Type Soundness in the Dart Programming Language

Fabio Strocco

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Type Soundness in the Dart Programming Language

A Dissertation
Presented to the Faculty of Science and Technology of Aarhus University in Partial Fulfillment of the Requirements for the PhD Degree

by
Fabio Strocco
September 21, 2016
Abstract

Many mainstream programming languages are dynamically typed. This allows for rapid software development and programming flexibility because it gives programmers the freedom to use powerful programming patterns that are not allowed in statically typed programming languages. Nevertheless, this freedom does not come without drawbacks: static bugs detection, IDE support, and compiler optimization techniques are harder to implement. In the last decades, the research literature and mainstream programming languages have been aiming to reach a trade-off between statically typed and dynamically typed languages.

We investigate the trade-off, focusing on the area of optional typing, which allows programmers to choose when to use static type checking in parts of programs. Our primary focus is Dart, an optionally typed programming language with a type system that is unsound by design. What makes Dart interesting from a research point of view is that the static type system combines optional typing with nominal typing using subtyping rules that make the type system unsound even for fully annotated programs.

This dissertation contains three main contributions. We first show a formal model in Coq of a subset of the Dart type system and operational semantics. We prove in Coq that natural restrictions of the Dart type system guarantee the absence of runtime type errors caused by calls to missing methods and functions, without going all the way to full soundness. Our second main contribution concerns the design of a type safety analysis for the full Dart language, and experimental evaluations showing that the algorithm can be successfully applied to real-world programs. Our third main contribution consists of experimentally evaluating the benefits provided by each source of unsoundness in the Dart type system.
Resumé

Mange mainstream programmeringssprog er dynamisk typede. Dette muliggør hurtig softwareudvikling og programmeringsflexibilitet fordi det giver programmerører friheden til at bruge kraftfulde designmønstre som ikke er tilladt i statisk typede programmeringssprog. Ikke desto mindre, denne frihed kommer ikke uden ulemper: statisk fejl-detektion, IDE-understøttelse og programoptimerings teknikker er sværere at implementere. I de sidste århier, der har forskningslitteraturen og mainstream programmeringssprog sigtet efter at opnå et kompromis imellem statisk typede og dynamisk typede sprog.

Vi undersøger kompromiset, hvor vi fokuserer på området optional typing, som lader programmerøren at vælge hvornår der bruges statiske type checks i dele af programmer. Vores primære fokus er Dart, et programmerings sprog med optional typing som er designet til at være usundt. Det som gør Dart interessant fra et forskningssynspunkt er at det statiske type system kombinerer optional typing med nominelle typer med subtype-regler som gør type systemet usund for selv fuldt annoterede programmer.

Acknowledgments

First, I would like to thank Anders Møller for his supervision, and for having supported me with his positive attitude. Non secondly, I would like to thank Erik Ernst for supervising and contributing to my research. I would also like to thank my colleagues for their support during my PhD. In particular I am indebted with Thomas Heinze and Gianluca Mezzetti for the contribution on the research on Dart and for having reviewed this dissertation multiple times. A special thank also to Esben Andreasen for his support and the effort in reviewing this dissertation.

I visited the University of Maryland during the PhD. I would like to thank Jeff Foster for his daily supervision, and his students, who contributed to create a very nice and friendly working environment.

I am indebted to the Google team in Aarhus, for the talks and advise concerning Dart. I am also indebted to Fritz Henglein and Jeremy Siek for their final thesis assessment and the constructive conversation during the PhD defence. Finally, a special thank to my parents for their constant support during the PhD.

Fabio Strocco,
Aarhus, September 21, 2016.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Resumé</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Hypothesis</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Methodology</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Structure and Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Papers</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Optional Typing and Related Work</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Dart</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Coq</td>
<td>29</td>
</tr>
<tr>
<td><strong>3 Fletch</strong></td>
<td>41</td>
</tr>
<tr>
<td>3.1 Syntax</td>
<td>42</td>
</tr>
<tr>
<td>3.2 Operational Semantics</td>
<td>43</td>
</tr>
<tr>
<td>3.3 Type System</td>
<td>52</td>
</tr>
<tr>
<td><strong>4 Message Safety</strong></td>
<td>57</td>
</tr>
<tr>
<td>4.1 Message-Safe Programs</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Full Type Safety</td>
<td>61</td>
</tr>
<tr>
<td>4.3 Message Safety and Nominal Identity</td>
<td>64</td>
</tr>
<tr>
<td>4.4 Message Safety for Program Fragments</td>
<td>64</td>
</tr>
<tr>
<td>4.5 A Two-Step Approach Toward Type Safety</td>
<td>65</td>
</tr>
<tr>
<td>4.6 Message Safety for Other Languages</td>
<td>67</td>
</tr>
<tr>
<td>4.7 Soundness of Message Safety</td>
<td>69</td>
</tr>
<tr>
<td>4.8 Auxiliary Lemmas and Challenges in Coq</td>
<td>77</td>
</tr>
<tr>
<td>4.9 Local Message-Safety</td>
<td>82</td>
</tr>
</tbody>
</table>
5 Type Safety Analysis 93
  5.1 Optional Types & Flow Analysis 95
  5.2 Trusting Type Annotations 96
  5.3 The SafeDart Analysis 98
  5.4 A Variation of Fletch 100
  5.5 Abstract Domain for Types 101
  5.6 Type Inference Constraints 101
  5.7 Open World Assumption 109
  5.8 Discussion 111
  5.9 The Type Checking Phase 113
  5.10 Evaluation 115
  5.11 Related Work 121

6 Type Unsoundness in Practice 123
  6.1 Types and Dart 125
  6.2 Sources of Unsoundness in Dart 127
  6.3 Experiments 134
  6.4 Related Work 140

7 Conclusions 143

Bibliography 147
Chapter 1

Introduction

Types play an important role in programming languages. In general, types help to keep programs readable, structured, maintainable, to find bugs, and to allow for better compiler optimizations.

One of the most debated aspects of programming language design is about the role of types, especially whether and how languages should support types statically or dynamically, i.e., during source code analysis or during program execution. For example, in Java and C# an expression that has static type Object might evaluate to a value of dynamic type type String. A programming language is statically typed if it has a static type system, which is a set of rules used to determine the types of relevant programming components before program execution, and to reject programs that cannot be assigned a static type. This prevents a well defined class of runtime errors, depending on the design of the type system. For example, in Java a variable declared with type boolean cannot contain a value of dynamic type String at runtime, so preventing errors caused by assigning values to variables with incompatible types.

For many years, statically typed programming languages, such as C, C++ [45], Java [35], and C# [19], have been massively used. These languages require to explicitly provide type annotations. For example, Java programmers are forced to specify the type of each local variable and object field. Other statically typed languages, which are not widely used but well known in academia, for example OCaml [72], and Haskell [80], do not require programmers to provide type annotations. Such languages use a type inference algorithm instead, which infers, i.e., automatically deduces, the static type of expressions.

Statically typed languages can typically prevent a well defined class of runtime type errors. Although this guarantee is beneficial, type systems may reject programs that are useful in practice, sometimes preventing programmers from expressing powerful programming patterns used in real-world programs. This is particularly true in script programming and web development, which
CHAPTER 1. INTRODUCTION

became popular in the 1990s. The static typing programming discipline, for example, in the earliest versions of Java and C, did not fit with web application and script development. Thus, *dynamically typed* programming languages, i.e., languages without a static type system, such as JavaScript [20] and Ruby [73], have become very popular. Recently, also bytecodes typically used for compilation of statically typed languages are introducing instructions to facilitate dynamically typed language compilation. For example, the Java Virtual Machine [90] bytecode supports the “invokedynamic” instruction, which allows dispatching methods independently on the actual object receiver type.

The programming flexibility provided by dynamically typed programming languages does not come without drawbacks: efficient code refactoring, method navigation, code completion, static bug detection, and compiler optimizations techniques are harder to implement. For example, JavaScript IDEs provide poor support for programmers. Static program analysis can provide better tool support, although the design of this analysis is a complicated endeavor [29].

Modern programming languages and recent literature are trying to achieve a trade-off between statically typed languages and dynamically typed languages to provide a reasonable programming flexibility, while preserving some of the benefits provided by statically typed programming languages. For example, TypeScript [4, 57], a typed extension of JavaScript, and the latest versions of Python and Ruby, follow the discipline of *optional typing* [6, 75, 97]. An optionally typed language allows programmers omitting type annotations to selectively disable type checking. For example, if a programmer declares a variable without specifying any type annotation, the type system allows for any use of that variable in the program. Note that also some of the languages mentioned earlier, e.g., OCaml, Scala, and Haskell, allow selectively omitting type annotations. However, it does not mean that these languages are optionally typed: optional type systems disable type checking at places where type annotations are omitted, whereas the OCaml, Scala, and Haskell type systems infer a type at places where type annotations are missing.

Optional typing enables *gradual typing* [74, 75, 79], a typing discipline lying in the design space between statically typed and dynamically typed languages. Gradual type systems allow programmers choosing which regions of code are statically typed and which regions are dynamically typed, thereby enabling gradual evolution of code between dynamically typed and statically typed code. Clearly, gradual type systems are not sound in the traditional sense. In fact, dynamically typed parts of code can cause runtime type errors. However, gradual type systems still provide static guarantees [79]: 1) fully statically typed programs cannot fail with a runtime type error, 2) gradual type systems never reject dynamically typed parts of code, 3) if programs fail with a runtime type error, it is caused by dynamically typed parts of code, and 4) removing type annotations from well-typed programs does not cause programs to be rejected from the type system or to fail with a runtime type
error. The last guarantee is called gradual guarantee.

Many recent mainstream programming languages are following the direction of gradual typing. For example, TypeScript and Dart allow programmers choosing when to enable type checking. However, these languages violate some the principles of gradual typing presented above. This dissertation is focused on the area of optional typing and gradual typing.

Dart

We use Dart as subject programming language to conduct our research. Dart is an object oriented programming language designed by Google with the following goals: allowing for web application development, mobile applications, and supporting internet of things. Dart has a type system that follows optional typing discipline: it allows annotating declarations, e.g., variables and fields, with the special type dynamic, disabling type checking at uses or assignments involving these declarations. It is also possible to omit type annotations at declaration sites, but in most cases this is equivalent to annotate declarations with type dynamic. The Dart type system is unsound even for fully annotated programs, because of implicit downcasts, covariance for generics, bivariant function subtyping, and bivariant method overriding, as we will show in Section 2.2. What makes Dart interesting from a research point of view is that the type system combines the unsound features mentioned above with nominal typing.

In this dissertation we will use the following terminology for any program analysis tool for Dart, including the type system defined in the specification: A type error, or runtime type error, is an error raised by a runtime type checker causing the program execution to fail, a type warning is a warning message raised by a static program analysis tool at places where a potential runtime type error can occur, and a type warning raised at a program point is spurious if that program point does not cause a runtime type error.

Dart is standardized by ECMA [21]. Despite the specification being informal, it is exhaustive enough to allow research with respect to type systems and program analysis. An interesting aspect of Dart is that it has two execution modes: production mode and checked mode. The former ignores type annotations at runtime, with exception of reflection and type tests [21] pages 8,9, whereas the latter is an extension of production mode that takes into account type annotations provided by programmers by inserting runtime type checks. In particular, for the purpose of this dissertation, we only consider two categories of type errors that can occur during Dart program execution: message-not-understood and subtype-violation. The former, technically corresponding to a NoSuchMethodError exception, occurs at failed lookups or function calls, e.g., at x.foo, if foo is not a provided by the object assigned to x. The latter, technically corresponding to a TypeError exception, occurs at inconsistent write operations, e.g., x = y where x has static type int and
y has runtime type `String`. Message-not-understood errors can occur in both checked mode and production mode, whereas subtype-violation errors can only occur in checked mode. This raises the question whether a type system or a program analysis tool for Dart should take into account both execution modes, and what kind of guarantees a type system or program analysis tool should provide for checked mode and production mode. For example, could a program analysis tool or type system preventing message-not-understood errors in production mode be successfully applied to real-world Dart programs? Similarly, could an analysis tool preventing message-not-understood, and possibly subtype-violation, in checked mode be successfully applied to realistic Dart code?

Similarly to Dart, TypeScript [57] has an optional type system that is unsound even for fully annotated programs [4]. However, types do not affect TypeScript program execution, in the spirit of Dart’s production mode. The literature proposes an operational semantics similar to Dart’s checked mode [70], but runtime type checks are computationally expensive for TypeScript, because it supports structural typing, whereas Dart uses nominal typing.

Despite the Dart type system being unsound, the language specification [21, pages 136,137,144] encourages the development of stricter type checkers and program analysis tools; our work is indeed focused on investigating suitable type checkers and program analysis tools for Dart.

1.1 Hypothesis

The hypothesis of this dissertation has three parts, all pertaining to the design of Dart’s type system:

1. We believe that under some reasonable assumptions the Dart type system is able to ensure the absence of message-not-understood errors in checked mode execution. More specifically, we expect that it is possible to identify natural restrictions of the type system that ensure the absence of message-not-understood errors without going all the way to full soundness.

2. It is possible to design a sound program analysis that detects potential message-not-understood and subtype-violation errors, and the analysis can be successfully applied to real-world Dart programs. We expect the analysis to be precise enough to raise few spurious type warnings.

3. In agreement with the Dart design principles, we believe that each source of unsoundness introduced in the Dart type system is justified by the advantages it provides to programmers: For each source of unsoundness, switching to the sound alternative would result in a significant increase in
1.2 Methodology

This section describes the methodology that we use to investigate the hypothesis presented in Section 1.1. Regarding the first part of the hypothesis, we build a formal model of a subset of Dart according to the specification [21]. We only capture the features that are relevant to characterize the core of the Dart type system. We formalize the Dart syntax, operational semantics, and type system in Coq [43]: a proof assistant typically used to formalize programming languages, which allows encoding mechanical certified proofs. Using Coq to model programming languages is a standard choice; for example, a subset of Java has been formalized in Coq [50].

Our formal model is the starting point to identify restrictions of the type system that ensure the absence of message-not-understood errors, thereby testing the first part of the hypothesis. In particular, we encode a lemma in Coq stating that programs accepted by the formalized type system cannot fail with message-not-understood errors at runtime. We then iterate over the following two steps: 1) we start proving the lemma, 2) if Coq rejects our proof, we either modify the lemma assumptions or restrict the type system that is part of our formal model. Using Coq helps this process by ensuring to cover all the subtle corner cases in the proof. After trial-and-error, we find the right assumptions to prove the lemma mentioned above.

Regarding the second part of the hypothesis, we design and implement a type safety analysis to detect potential message-not-understood errors and subtype-violation errors in Dart programs. Our analysis is divided in two parts: a dataflow analysis and a type checker that uses the types inferred by the dataflow analysis to raise type warnings at places where message-not-understood and subtype-violation errors can occur at runtime. Our goal is to minimize the number of spurious type warnings that are raised by our analysis. To this end, we choose a set of benchmarks that are presumably stable to ensure that each warning is unlikely to correspond to a runtime type error. This allows measuring precision: a low number of type warnings raised in stable benchmarks means that the analysis is precise. We also measure relative precision by implementing a naive type safety analysis serving as baseline: a low number of type warnings compared to the baseline means high relative precision. We then iterate over the following three steps: 1) we apply our analysis on a set of benchmarks, 2) we manually investigate a selection of the type warnings raised by the analysis to understand their nature, 3) if a warning is caused by the Dart standard library we provide an ad-hoc solution.
otherwise we find general patterns in the warnings and, basing on that, we improve the analysis. After trial-and-error, our inference produces a relatively low number of warnings on the selected benchmarks.

Regarding the third part of the hypothesis, we classify the sources of unsoundness in the Dart static type system and we define a natural sound alternative for each unsoundness. Then, we modify the Dart static type checker accordingly. Whenever the type checker restrictions involve subtyping, we modify the runtime subtype checks in the virtual machine accordingly. For example, one source of the unsoundness is generic covariance; when we switch to the sound alternative, we restrict the static type system with generic invariance. Therefore, we also modify the subtype relation in the virtual machine to support generic invariance. The purpose of this implementation is to investigate the impact of restricting the static and runtime type checks to a sound alternative on both the Dart language implementors and programmers. The implementations can be configured so that only a subset of the sources of unsoundness are switched to their respective sound alternative. For each meaningful subset, we run the modified static type checker on each benchmark and the modified virtual machine on each testcase that we are able to successfully execute in our set of benchmarks. We then perform quantitative and qualitative analysis on the results. Regarding the quantitative analysis, we measure the impact on each subset of natural sound restrictions of the static and dynamic type checkers using the following metrics: the number of type warnings raised by the modified static type checker, the number of projects where at least one type warning is raised, and the number of projects where at least one testcase fails. A low number of warnings associated to a pool of natural restrictions of unsoundness means that it has little use in practice. For example, if switching covariance for generics to invariance introduces only few type warnings, then generic covariance might be ruled out from the type system in favor of generic invariance without a significant impact on Dart programmers. Regarding the qualitative analysis, we attempt to understand the nature of the type warnings by manually investigating a small portion of these warnings. We classify the warnings into two categories: warnings caused by a programmer design flaw and warnings that require a significant code refactoring or a more expressive type system.

1.3 Structure and Contributions

This dissertation is organized as follows:

- Chapter 2 contains some background material that provides a foundation for explaining the contributions of the remaining chapters. The chapter is logically divided in two parts: state of the art regarding optional typing, and technical background regarding Dart and Coq.
• Chapter 3 shows a formal model of a subset of Dart that we call Fletch. It is the starting point towards our understanding of the Dart type system, in order to test the first part of the hypothesis stated in Section 1.1. This chapter presents the following two contributions: we provide a formal model of a subset of the Dart syntax, operational semantics, and type system and we verify the formal model in Coq (see http://www.brics.dk/fletch/). This chapter is extracted from Ernst et al. [25] and some of the technical content has been extended. More specifically, we extend the content of Figure 3.9.

• Chapter 4 defines the notion of message-safe programs, which can be viewed as a natural level between dynamic and static typing. The main goal of this chapter is to test the first part of the hypothesis stated in Section 1.1. This chapter presents the following three main contributions: we define message-safety and show its potential role in software development, we prove that message-safe programs cannot fail with message-not-understood errors at runtime, and we discover a property of the runtime type system that was not intended by the Dart language designers. The message-safe type system and the proof of message-safety are encoded in Coq (see http://www.brics.dk/fletch/) This chapter is extracted from Ernst et al. [25] and extended with supplementary technical material. More specifically, we extend the content of Sections 4.1 and 4.7 as well as Figure 4.3. Moreover, Figures 4.1 and 4.5 as well as Sections 4.8 and 4.9 are not part of Ernst et al. [25].

• Chapter 5 shows a type safety analysis for Dart to prove the absence of message-not-understood and subtype-violation errors in Dart programs at runtime. It is the starting point to test the second part of the hypothesis stated in Section 1.1. This chapter presents the following two contributions: we show and implement different variants of the analysis (see http://www.brics.dk/safedart/) and we experimentally evaluate our approach to real world Dart programs. This chapter is extracted from Heinze et al. [40] and extended with supplementary technical material. In particular, Sections 5.6.1 and 5.7, Figure 5.3 and Tables 5.2, 5.3, 5.4, 5.5, 5.6 are technical material that is not part of Heinze et al. [40].

• Chapter 6 tests the third part of the hypothesis described in Section 1.1 by evaluating the pros and cons of each source of unsoundness in the Dart type system. This chapter presents five main contributions. First, we present an approach for experimentally evaluating the pros and cons of each source of unsoundness of the Dart type system. Our second contribution consists on classifying each source of unsoundness in the Dart type system. The third contribution consists on modifying the Dart static type checker to get full type soundness and restricting runtime
type checks to be consistent with the restrictions applied to the static checker. Our fourth contribution consists on experimentally evaluating the effect of applying the modified type checkers and runtime checks on real-world programs in terms of increment of static type warnings and runtime type errors. The complete experiment dataset is available online (see http://www.brics.dk/undart/). The fifth contribution is the following: we find out that some unsound features of the Dart type system have little use in practice. This chapter is extracted from Mezzetti et al. [56], and extended with Figure 6.3.

- Chapter 7 concludes by showing the results of the dissertation in relation to the hypothesis stated in Section 1.1. Most of the text in this chapter is extracted from the papers that are part of this dissertation: [25, 40, 56].

The reader of this dissertation is expected to be familiar with the basics of type theory [60] and static analysis [59].

1.4 Papers

The following papers were co-authored as part of the scientific work during the author’s Ph.D studies at Aarhus University and are part of this dissertation.

- Message Safety in Dart has been published in Dynamic Languages Symposium 2015 [24]. It is an extended version of Managing Gradual Typing with Message-Safety in Dart, published in Foundations of Object-Oriented Languages 2014 [23]. A journal version has been accepted for publication in Science of Computer Programming [25].

- Type Safety Analysis for Dart has been accepted for publication in Dynamic Languages Symposium 2016 [40].

- Type Unsoundness in Practice: An Empirical Study of Dart has been accepted for publication in Dynamic Languages Symposium 2016 [56].
Chapter 2

Background

As discussed in Chapter 1, statically typed programming languages provide many benefits, for example, helping static bug detection, IDE support, and compiler optimization techniques. Type systems typically ensure that well-typed programs cannot fail with a type error at runtime, by rejecting all the programs that cannot be proven to execute without runtime type errors. A type system that accepts only programs that cannot fail at runtime rejecting only programs that will definitely fail at runtime is undecidable. Therefore, type systems typically reject programs that could be correctly executed, including programs that may in fact be useful.

For this reason, dynamically typed programming languages, i.e., languages that do not have a static type system, have become popular. In these languages type errors typically occur dynamically, i.e., at runtime. Such programming languages are also called untyped languages [66]. Another term used in the literature is uni-typed languages [37], since from a theoretical perspective, dynamically typed programming languages have a static type system that accepts all the programs and each program component has the same static type.

Although dynamically typed languages allow for programming flexibility, static bug detection, IDE support and compiler optimization techniques are harder to implement than in statically typed languages. It is possible to provide an acceptable level of freedom also in statically typed languages. The literature shows two possible options. The first option consists on adding expressiveness to type systems, for example, by supporting dependent types [5]. However, it is hard to program in languages that uses dependent types for two main reasons: their type checkers might not terminate for some programs, because type checking with dependent types is not computable and dependent types require a huge type annotation burden. The second option consists on increasing programming flexibility by loosening some of the constraints occurring in traditional fully sound type systems. For example, in languages supporting optional typing, type annotations can be selectively omitted: if a
program component, such as, a variable, is declared without a type annotation, type checks at use or assignments to the variable are disabled. A related typing discipline is *gradual typing* [74][75][79], which allows gradual evolution of programs between static typing and dynamic typing by giving programmers the freedom to choose what parts of programs are statically typed and what parts are dynamically typed.

This chapter discusses the most relevant literature on optional typing and gradual typing, it introduces Dart, an optionally typed programming language, and it presents a brief introduction to Coq: a proof assistant that we use to build a formal model of Dart, which we describe in Chapter 3 and Chapter 4.

### 2.1 Optional Typing and Related Work

This section presents the most relevant literature related to this dissertation, which is about combining the advantages of statically typed languages with the programming flexibility provided by dynamically typed languages. More specifically, we focus on *optional typing*, where type annotations are indeed optional.

**Gradual Typing in Academia** Optionally typed languages enable the idea of *gradual typing* [74][75][79]. A gradual type system provides a special type with the same semantics of the type *dynamic* in optional typing, which is typically the default type when type annotations are omitted, and it allows disabling type checks at parts of programs. The type *dynamic* is typically called *any* or * in gradual typing [4][79], however we will use the world *dynamic* in this dissertation. We will say that a program is *fully annotated* to express that no program component has static type *dynamic*, whereas we will say that a program is *partially annotated* if some program component has static type *dynamic*, and *non-annotated* if every program component has static type *dynamic*.

According to Siek et al. [79], gradual type systems have four key properties: 1) every non-annotated program is well-typed, 2) well-typed fully annotated programs cannot fail with a runtime type error, 3) well-typed programs cannot fail with runtime type errors caused by fully annotated code, and 4) removing type annotations from well-typed programs does not change program behavior. The last property is called the *gradual guarantee*. We will demonstrate these properties of gradual typing by showing some examples.

```
1    square(n) {
2        return n * n;
3    }
```

Example 2.1: Non-annotated program.
A gradual type system accepts this program, according to the first property of gradual typing. However, the call `square("")` would cause a runtime error at line 2. Adding the `int` type annotation to the `n` parameter will allow to catch this error statically. A corresponding fully annotated program is the following.

```java
int square(int n) {
    return n * n;
}
```

Example 2.2: Fully annotated program.

According to the second property of gradual typing, this program is rejected by a gradual type system by raising a type error at line 8, because `String` is not a subtype of `int`.

A more interesting question is: how should a gradual type system behave when programs are partially annotated? Let us consider the following example.

```java
square(n) {
    return n.toString();
}
```

Example 2.3: Partially annotated program.

The function `isSquare` is fully annotated except that it contains a call to a non-annotated function `square`, which has return type `dynamic`. Typically, gradual type systems allow for implicit downcasts from `dynamic` to any type. This means that the program in Example 2.3 is accepted by a gradual type system, but the type system cannot ensure that the value computed by `square(n)` will be assignable to `s` at runtime. To overcome these limitations, gradually typed languages typically emit runtime type checks at write operations to ensure that the type annotations can be trusted. In the example, the execution of the call at line 18 will cause a runtime type error at line 14. In fact, assuming `n.toString()` returns a string, the call `square(n)` return a value of type `String`, which is not a subtype of `int`.

Basing on the examples presented so far, it would be possible to state that one of the key properties of gradual typing is that well-typed programs can only fail in non-annotated or partially annotated parts of code. In fact, considering the Example 2.3, the runtime type error at line 14 occurs in the `isSquare`
function, which is partially-annotated because of the call to \texttt{square(n)}. However, in some cases runtime type errors can occur in statically typed code. Let us consider the following example

```c
int square(int \texttt{f(int n)}, int n) {
    return \texttt{f(n) \* f(n)};
}
fun(n) {
    if(n > 0) return n;
    else return "Error";
}
...
square(fun, 1);
square(fun, -1);
```

Example 2.4: Partially annotated program.

The \texttt{square} function is fully annotated and it takes an higher-order parameter \texttt{f}, which is a function of type \texttt{(int)-> int}. We would expect that no runtime error occurs during the execution of \texttt{square}, because it is fully-annotated and well-typed. However, the \texttt{fun} function has type \texttt{(dynamic)-> dynamic}. Since gradual type systems typically allow for implicit downcasts from \texttt{dynamic} to any type, \texttt{fun} can be assigned to the \texttt{f} parameter, which has type \texttt{(int)-> int}, without causing a runtime type error. However, the call at line 29 will cause a runtime type error at line 20, because \texttt{f(-1)} returns a value of type \texttt{String}. Thus, the runtime type error occurs inside well-typed fully annotated code. However, it is possible track the dynamically typed code causing the error. Gradual typing introduces the notion of \textit{blame error}, which consists of a runtime type error that contains informations about the code causing the error. In Example 2.4 a blame error occurs at line 20 blaming line 29. With this notion of blame we can express the third principle of gradual typing by stating the \textit{blame-subtyping theorem}: \textit{if a blame error occurs, than it can only be caused by an implicit downcast at runtime}. Under the reasonable assumption that implicit downcasts at runtime only concern the type \texttt{dynamic}, and that these casts are only inserted at partially-annotated program fragments, clearly blame errors cannot blame fully annotated portions of code. The literature shows techniques for blame tracking for gradually typed languages \cite{30, 94}, possibly combining them with contracts \cite{78}.

The last key property of gradual typing is the \textit{gradual guarantee}, stating that gradually typed languages guarantee that removing type annotations from a well-typed program will not introduce static type errors. Additionally, runtime program execution should not change after removing type annotations, provided that no runtime error were occurring before removing the type annotations. This gradual guarantee allows gradual evolution from fully-annotated programs to non-annotated programs. Unfortunately, gradual typing does not provide the same notion when moving from dynamically
typed to statically typed code. However, the gradual guarantee still ensures that adding type annotations to well-typed programs does not cause them to be rejected by the type system, as long as the type annotations are correct. For example, the gradual program evolution from the program in Example 2.1 to the program in Example 2.2 and the converse does not change the program behavior and does not prevent the two programs from being well-typed. However, according to the second guarantee mentioned above, adding the wrong type annotations `String` to the `square` function in the program in Example 2.1 would cause a static type error.

**Gradual Typing in Industry**  Gradual typing is also being supported in industrial programming languages, which we will mention later, although they violate some of the four properties mentioned above. One of the main reasons for violating the principles is that the current technology for gradual typing can cause significant performance issues due to the cost of inserting type checks at runtime [85]. We will briefly show some examples of the application of gradual typing to industrial programming languages.

TypeScript [57], an extension of JavaScript with type annotations, provides a type system that supports the type `any`, which enables dynamic typing, thereby enforcing a gradual program evolution between static and dynamic typing. Unlike gradual typing, the TypeScript type system is unsound even when programs are fully annotated. Additionally, no extra type checks are inserted at write operations at runtime and there is no notion of blame error. However, the TypeScript type system still provides some guarantees: recent work shows that a well determined subset of TypeScript is actually sound [4] and [70] shows how to add runtime type checks at places where dynamically typed code flows into static type code. Moreover, TypeScript provides a local type inference at non-annotated program fragments.

Similarly, Flow [27], which is also an extension of JavaScript with type annotations, enables gradual typing along with a type inference that can be possibly applied at places where type annotations are missing. In contrast with TypeScript, which has been designed under the assumption that a large portion of programs does not contain type annotations, Flow relies on the fact that a large part of the code is fully annotated. This allows the Flow type inference trusting type annotations modularly. For example, the Flow type inference relies on type annotations and assumptions on types in presence of libraries massively using dynamic code evaluation. Moreover, to meet different programmer needs, Flow supports *weak mode*, meaning that the type inference is disabled at missing type annotations.

Hack [92], a dialect of PHP, also supports gradual typing. Unlike TypeScript and Flow, Hack program execution enables runtime type checks when type errors cannot be detected statically. The Hack language provides type hints, which differs from traditional type annotations because if a type hint
causes a runtime type error, program execution does not stop; instead, a warn-
ing corresponding to the runtime error is raised, and program execution can
continue. This allows gradually evolving programs replacing type hints to
actual type annotations.

The languages mentioned so far are designed as variations of existing dy-
namically typed programming languages with the purpose of retrofitting a
type system for these languages. Instead, Dart [21], which we will introduce
in Section 2.2 is an object oriented programming language completely de-
signed from scratch providing a type system that does not retrofit any other
language.

The idea of supporting optional type annotations was introduced in in-
dustrial programming languages before gradual typing. For example, pro-
gramming languages, such as, Cecil [16], Visual Basic .NET [54], Bigloo [51],
and ProfessorJ [47], could combine static and dynamic typing with optional
typing. However, gradual typing provides a solid foundation, which can in
principle be applied to these languages.

Variations of Gradual Typing The literature presents other variations of
gradual typing. For example, quasi-static typing [86] also supports the type
dynamic, which allows selectively disabling static type checking. Quasi-static
typing is based on the fact that ideally a type system should accept only
programs that will not fail with a runtime type error, and reject only pro-
grams that will definitely fail with a runtime type error. Such type system
is not computable. However, a quasi-static type checker implements an ap-
proximation of this type system by dividing programs into three categories:
1) well-typed programs, which cannot fail with a type error at runtime, 2) re-
jected programs, which will definitely fail with a runtime type error and
cannot be executed, and 3) ambivalent programs, for which no answer can
be provided statically, but can still be executed with additional runtime type
checks. In principle, a quasi-static type checker runs in two phases. During
the first phase, a type inference algorithm checks if the program is well-typed:
if the analyzed program is ill-typed, runtime type checks will be introduced
at runtime. The second phase, named plausibility phase, aims to prove that
runtime type errors will definitely occur; if it is the case, the program will be
rejected, otherwise the program will be accepted with runtime type checks.
Despite supporting the type dynamic, quasi-static typing differs from gradual
typing because it does not statically catch all the runtime type errors for fully
annotated programs.

A work that is closely related to quasi-static typing is hybrid type check-
ing [31], which uses dependent and refinement types. Similarly to quasi-static
typing, hybrid type checkers classify programs into well-typed, ill-typed, and
ambivalent. A dependent type is a function type that depends on some pro-
gram input. For example, given a function with parameter x, it can have the
following dependent type: \( x : \text{int} \rightarrow \text{if } x == 0 \text{ then String else int} \). This means that the function parameter has type \text{int}, and the return type depends on the \( x \) parameter value: if it is 0, the return type is \text{String}, alternatively the return type is \text{int}. A refinement type specifies a set of values that satisfy a predicate. For example, the type \( \{ x : \text{int} \mid x > 10 \} \) represents the integers greater than 10. Subtyping among refinement and dependent types is not decidable. Therefore, hybrid type checkers might not terminate classifying programs as ambivalent. In such case, a runtime type check is inserted.

Unlike quasi-static typing, the main goal of hybrid type checkers is to combine static and dynamic program verification with dependent and refinement types. When static verification is not possible, the type checker defers program verification at runtime. Another difference between hybrid type checking and quasi-static typing, is that the former does not support the type \text{dynamic}.

Recent literature shows many other alternatives to gradual typing. For example, like types \[97\] introduce an intermediate step between concrete type annotations and the type \text{dynamic}. Declarations of variables or instance fields with like types are checked against their use, but every value can be assigned to them. For example, if a variable \( x \) has type \text{like String}, the expression \( x * x \) is disallowed by the type checker, but \( x \) can be assigned to numeric values. The advantage of like types is that it is possible to obtain the same programming flexibility of structural typing into a nominal setting. Moreover, like types mitigate the runtime overhead compared to gradual typing because no type check is needed when assigning values to elements declared with a like type. Let us consider the following example.

```java
30 class Point {
31 ... 
32 }
33
34 class CartesianPoint extends Point {
35 int x;
36 int y;
37 }
38
39 class PolarPoint extends Point {
40 int x;
41 double angle;
42 }
43
44 int getX(like CartesianPoint p) {
45 return p.x;
46 }
47
48 getX(new CartesianPoint());
49 getX(new PolarPoint());
```

Example 2.5: A program that uses like types.
Chapter 2. Background

Assuming that some subclasses of `Point` do not provide the field `x`, the field declaration needs to be duplicated in `CartesianPoint` and `PolarPoint`, which are not related by subtyping. Because of the calls at lines 48 and 49, the `getX` method parameter `p`, can be assigned to both values of type `CartesianPoint` and values of type `PolarPoint`. Therefore, the parameter `p` should be declared with type `dynamic` in a gradually typed language. Instead, a type checker supporting like types would allow declaring `p` with type `like CartesianPoint` as in the example above, and the call at line 49 would be type correct without generating any runtime type errors. However, if we replace the return expression in the `getX` function with `p.angle`, the type checker would raise a type error because `angle` is not provided by `CartesianPoint`.

A similar approach is implemented in StrongScript [71], a restriction of TypeScript. StrongScript type annotations are divided in three main categories: 1) the type `any`, 2) optional types, and 3) concrete types. The first category corresponds to the type `dynamic` in gradually typed languages, the second category allows dynamically typed code to flow into statically typed code, whereas implicit downcasts from `any` concrete types are forbidden statically. The following example shows the difference between optional types and concrete types.

```
50 any foo() {
51     return "foo";
52 }
53
54 int! bar() {
55     return 0;
56 }
57
58 int x = <any>(foo());
59 int! y = <any>(foo());
60 int! z = bar();
```

The variable `x` has an optional type `int`, and the assignment at line 58 is well-typed because of the implicit downcast from `any` to `int`. A type check at runtime will stop program execution because the value computed by `foo()` has runtime type `String`. Conversely, the assignment at line 59 is rejected by the type checker, because `int!` is a concrete type: no implicit downcast from `any` to `int!` is allowed. Clearly, the assignment at line 60 is accepted by the type checker. Similarly to like types, concrete types do not require runtime type checks, thereby significantly improving runtime performances compared to gradually typed languages.

Most of the work related to gradual typing aims to combine the advantages of static and dynamic typing by designing type systems that are less restrictive than traditional fully sound type system. However, the flexibility required by a type system might vary depending on programmers and application domain. Progressive types [67] allow programmers choosing what class
of errors to check statically and what class to defer to runtime. For example, it is possible to statically check for function calls with wrong number of arguments, but to defer to runtime the type checks ensuring type compatibility when assigning values to variables. Despite being flexible, progressive type systems still have a well defined notion of soundness: if a type system is tuned so that a certain class of type errors \( C \) is statically checked, well-typed programs cannot fail with type errors of such class \( C \) at runtime. Progressive typing differs from gradual typing since type annotations must optional in gradual typing; conversely, progressive types demands a huge type annotation burden. A related work to progressive types is pluggable type systems \[6\].

The philosophy of pluggable type systems is that type annotations in programming languages should be seen as meta data, without affecting program execution and without rejecting ill-typed programs. Instead, multiple type checkers could be plugged into the language depending on programmer needs, to provide programming support. Note that also optional type systems are not mandatory. For example, the Dart type system allows ill-typed programs being executed. This in turns enables pluggable type systems.

The work we have discussed so far, can only be applied to programming languages that allow specifying type annotations. Conversely, soft typing \[10\] does not require languages to support type annotations. Soft typing is a typing discipline based on deferring type checking to runtime, often using a type inference that detects potential type errors statically, and introducing runtime type checks at program points where a type error is detected statically. The literature shows how to apply soft typing in languages like Scheme \[96\], a dynamically typed functional programming language. A type inference algorithm is executed on Scheme programs before their execution. If a program is ill-typed, the type checker emits informations about the potential type errors as well as runtime type check, but the program can still be executed. If such type error occurs at runtime, the program execution will fail. The significant difference between gradual typing and soft typing is that the former does still provide static guarantees, whereas the latter does not, but it allows for more programming flexibility. Moreover, soft typing can also be applied to programming languages that do not provide type annotations, whereas gradual typing does not differ from dynamic typing when type annotations are not specified. Similarly, quasi-static typing and hybrid type checking, only insert runtime type checks where the type checker is not able to classify a program as well-typed or ill-typed, whereas soft typing allows every program to be executed, and its type inference is computable. Similarly to soft typing, progressive types may defer type checks at runtime. However, according to progressive types,’ programmers have to choose when to defer errors to runtime, and progressive types rely on type systems and type annotations whereas soft typing applies a whole program analysis.

Several type systems that allow for gradual typing, soft typing, etc., have a well defined notion of soundness. Conversely, work on success typing shows
deliberately unsound type systems. In contrast with traditional sound type systems, which imply that well-typed programs cannot go wrong, success typing implies that ill-typed programs will definitely go wrong. In other words, if a type error occurs in a system that uses success typing, it will definitely occur at runtime. Such notion has been proven useful in practice: a type system based on success typing has been implemented in Erlang; it allowed discovering many bugs in real-world programs [48]. However, success typing requires a backward reasoning that can be only easily applied to functional programming languages. Related types [95] follow a similar approach by detecting dead code, thereby avoiding unnecessary static type errors.

Application of Type Inference to Gradual Typing The literature also shows how to combine type inference with gradual typing. For example, functional programming languages can support gradual along with a unification based type inference [76], although constraint resolution is significantly different compared to traditional unification based inference. Recent work also shows how to apply type inference for ActionScript [14], a programming language supporting gradual typing. The literature also shows how to apply type inference to automatically add type annotations in a context of gradual typing [69].

2.2 Dart

Dart is a programming language designed and developed by Google, originally aimed for Web application development, later enriched to support mobile application development and internet of things. Dart supports object oriented features, i.e., classes, interfaces, and mixins, as well as functional features, and it allows for client-side web programming, similarly to TypeScript [57] and Javascript [20]. Client-side web programmers, which are used to JavaScript, typically require a high degree of programming flexibility, to allow for rapid software development. To this end, Dart supports optional typing (see Section 2.1). However, its syntax and semantics are much closer to Java and C# and much cleaner than their competitor languages, such as, TypeScript and JavaScript. Additionally, Dart differs from typical web-oriented languages because it does not support JavaScript-like explicit dynamic code evaluation, although it supports a reflection system similar to Java and C#.

Dart is standardized by ECMA [21] and the Dart team has implemented the following set of tools supporting Dart: a virtual machine [88], a compiler from Dart to Javascript, named dart2js [33], which contains a static analyzer named dartanalyzer [34], which contains a type checker, a virtual machine for Chromium [87], and a plugin for IntelliJ [44].

One of the key aspects of the Dart type system is that it is unsound by design. In fact, the type system is unsound even for fully annotated programs,
2.2. DART

for example, because of generic covariance and implicit downcasts.

The following example shows one of the key aspects of the Dart type system’s unsoundness, and the reason why the Dart team purposely choose to introduce unsound features.

```dart
class Account extends DefaultAccount {
  int amount;
  String owner;

  Account(int amount, String owner) {
    this.amount = amount;
    this.owner = owner;
  }

  Map<String, Object> informations = {
    "currentUser": "John",
    "sessionId": 10
  };

  main() {
    Account acc = new Account(0,
      informations["currentUser"]) as Account;
    print(account.owner.lowerCase());
  }
}
```

Example 2.6: A Dart program

This Dart program defines a class Account with the amount and owner fields, a global variable informations mapping values of type String to values of type Object, and a main function that is the program execution starting point. The expression `{ "currentUser": "John", "sessionId": 10 }` denotes a map literal, which maps the string "currentUser" to the string "John" and the string "sessionId" to the number 10. The program in Example 2.6 would be rejected by a traditionally fully sound type system because of the second argument at line 78 in the constructor call. Indeed, the argument informations["currentUser"] has static type Object, since informations is a map from String to Object, whereas the corresponding owner constructor parameter has declared type String. However, the runtime value produced by the expression informations["currentUser"] will be "John", which has runtime type String. The Dart static type system accepts these kind of programming patterns by allowing implicit downcasts, thereby accepting the parameter passing at line 78 in Example 2.6. Although these programming patterns could also be allowed by a sound but more expressive type system, the Dart language designers aim to enable rapid software development, thereby discouraging complicated type systems or type inference mechanism that can confuse programmers.
Dart supports two execution modes: production mode, which does not take into account type annotations - besides few exceptions - and checked mode, which extends production mode with runtime type checks at write operations. This means that Dart has two type systems: a static type system and a dynamic type system. We will say that the former might raise static type warnings instead of type errors, thereby always allowing program execution, whereas the latter raises runtime type errors. Similarly, all alternative static type analysis for Dart described in this dissertation will raise static type warnings, whereas all alternative runtime system for Dart will raise runtime type errors. We will describe the Dart dynamic and static type systems in the remaining part of this section.

2.2.1 Dart Program Execution

Production Mode According to the Dart specification [21, pages 8, 9] when a program is executed in production mode type annotations do not affect program execution besides a few exceptions. More specifically, the specification states as follows:

In production mode, static type annotations have absolutely no effect on execution with the exception of reflection and structural type tests.... Type tests also examine the types in a program explicitly.... Production mode respects optional typing. Static type annotations do not affect runtime behavior.

When programs are released, they typically run in production mode because it is faster than checked mode, which introduces a significant overhead due to runtime type checks. Let us consider Example 2.6, assuming the programmer accidentally swaps the values mapped by informations getting the following map: \{ "currentUser": 10, "sessionId": "John"\}. The program execution would stop at line 79 in production mode because owner has a value of type int, which does not provide the lowercase method. Despite the owner field being declared with type String at line 65 it has the value 10 at runtime. However, this program is accepted by the Dart type checker and the program execution only stops when the numeric value 10 is incorrectly used, instead of being stopped because owner cannot contain integers.

In this dissertation, we will use the term message-not-understood errors, to denote the runtime error occurring at line 79. More specifically, a message-not-understood occurs in the following cases.

- An object field or method lookup operation fails, for example at x.p, if the object x does not have the specified property p.

- A function call fails, because the callee does not resolve to a closure, or its number of argument is wrong, for example at f() if f does not
2.2. **DART**

resolve to a closure, or the corresponding closure requires one or more parameters.

Technically, when a message-not-understood error occurs, the Dart program execution stops with a `NoSuchMethodError` exception.

**Checked Mode**  According to the Dart specification [21, page 9], checked mode execution is an extension of production mode with additional runtime type checks at write operations. More specifically, the specification states as follows:

*In checked mode, assignments are dynamically checked, and certain violations of the type system raise exceptions at run time.... Checked mode utilizes static type annotations and dynamic type information aggressively yet selectively to provide early error detection during development.*

Programs are usually run in checked mode during the testing phase in order to discover some of the type errors the static type system does not catch.

Let us revisit Example 2.6, assuming the programmer accidentally swaps the values mapped by `informations` getting the following map:

\[
\{ \text{"currentUser": 10, "sessionId": "John"} \}
\]

As mentioned earlier, despite the program in the example being rejected by a traditionally fully sound type system, it is accepted Dart type system and it fails in production mode at line 79 with a message-not-understood error. Instead, program execution fails at line 78 in checked mode, because the dynamic type of 10 is `int`, which is not a subtype of the type of owner, which is `String`.

In this dissertation, we will use the term subtype-violation errors to denote the runtime error occurring at line 78. More specifically, a *subtype-violation* occurs in the following cases.

- An object with runtime type $T$ is assigned to a variable, parameter, or field with declared type $S$, and $T$ is not a subtype of $S$.

- An object with runtime type $T$ is returned from a closure with return type $S$, and $T$ is not a subtype of $S$.

Technically, when a subtype-violation error occurs, the Dart program execution stops with a `TypeError` exception. Since checked mode is an extension of production mode, message-not-understood errors can occur in both production mode and checked mode, whereas subtype-violation can only occur in checked mode.
2.2.2 Dart Features and Static Type System

This section shows the most important aspects of Dart to understand the content of this dissertation, in particular Chapters 3, 4, 5, and 6. Additionally, this section provides insight on the Dart static type system and shows why it is unsound.

Classes and Objects  Dart has a notion of classes similar to Java and C#. A class defines fields, methods, and constructors that generate object instances. Unlike Java and C#, in Dart every value is an object, including `null`, numbers, and closures. Indeed, Dart provides a predefined class `Null`, `int`, and `Function`. Every class is a subclass of `Object`, which provides methods and fields that can be invoked from any object, e.g., `toString` and `hashCode`.

Let us consider the following example:

```dart
void myFunction() {}  

void main() { 
  int x = 42;
  print(x.toString());
  print(null.toString());
  print(myFunction.toString());
}
```

This Dart program can be compiled and executed without errors, because every value is an instance of `Object`, including closures, thereby allowing calls to `toString` method even on null references. The program prints the string representations of the number `42`, the null reference, and the closure executing the code in `myFunction`.

Dart allows declaring getters and setters, as well as fields. Nevertheless, the specification [21] state that a class implicitly provides a getter and a setter for each field. Therefore, each direct access to a field (resp. assignment to a field) is converted to a call to the corresponding implicit getter (resp. implicit setter). Similarly to JavaScript each method has an implicit getter too. For example, given a class C with a `foo` method, the expression `new C().foo.toString()` is legal and the `toString` call returns a string representation of the closure extracted from `foo`.

An example of unsound feature of the Dart type system is that it allows bivariant overriding of methods and fields, as the following example shows.

```dart
class C { 
  int x;
  int m(Object x) { return 0; }
}

class D extends C { 
  Object x;
  Object m(int x) { return 0; }
}
```
Another unsound feature of the Dart related to class types is the support for covariant subtyping for generic types. For example $\text{List<int>}$ is a subtype of $\text{List<Object>}$.[2.2] We will discuss the consequences of these unsoundness in Chapter 4.

Another interesting aspect of Dart is that it supports higher-order functions and it allows defining callable classes. Each instance of a class $C$ defining a method with signature $T \text{ call}(T_1 x_1, \ldots, T_n x_n)$ is a callable object: Given an expression $e$ of type $C$, the following expression is accepted by the type system $e(e_1, \ldots, e_n)$, and the expression is equivalent to $e\text{.call}(e_1, \ldots, e_n)$. Let us consider the following example.

```dart
class F {
  call(x, y) { ... }
}
...
F fun = new F();
fun(1, 2);
```

The class $F$ defines a call method, thereby allowing the call at line 103 both at compile time and at runtime.

**Functions** Dart supports higher-order functions with escaping variables, as the following example shows.

```dart
foo(x) {
  return (y) { x++; return x + y; }
}

var f = foo(10);
print(f(10));
```

The $\text{foo}$ call at line 108 returns a function. The function returned by $\text{foo}$ contains both parameter $x$ and $y$, the former is outside of the lexical scope at line 108. At runtime the $\text{foo}$ function returns closure. A closure is a function with an environment that binds variables outside the scope with their current value. In Dart a closure is also an object, more specifically every closure is an instance of the class $\text{Function}$. Closures can be generated by anonymous functions, methods, global functions or nested functions.

Dart supports two different kind of anonymous functions. The first kind is function expressions. For example, the expression $(\text{int } x) \Rightarrow x$, is a function expression of type $(\text{int}) \rightarrow \text{int}$. The second kind of function is function blocks. For example, the function $(\text{int } x)\{ \text{return } x; \}$ is a function block of type $(\text{int}) \rightarrow \text{dynamic}$. Similarly to method overriding, function subtyping is also bivariant. For example, $(\text{int}) \rightarrow \text{Object}$ is a subtype of $(\text{Object}) \rightarrow \text{int}$.
**Futures**  The Dart standard library provides support for asynchronous computation with futures. A *future* is an object of type *Future* wrapping values that might be computed in the future. A future registers two callbacks: a function that computes the value and a function that is called when the value has been computed. For example, consider the following synchronous code.

```dart
110 main() {
111   print(add(2, 3));
112 }
113
114 int add(x, y) {
115   return x + y;
116 }
```

The `add` function is synchronous. The asynchronous version of `add` would wrap its body in an object of type `Future<int>`, which is a future that will asynchronously compute a value of type `int` and eventually return it, as the following code snippets shows:

```dart
117 main() {
118   Future<int> addResult = add(2, 3);
119   addResult.then((int value) {
120     print(value);
121   });
122 }
123
124 Future<int> add(x, y) {
125   return new Future<int>(() => x + y);
126 }
```

Example 2.7: A Dart program using futures.

The `add` function wraps the computation of `x + y` into a callback that is passed to the future. The `add` return value is now a future. The code at line 119 calls the `then` method from the future passing a callback that will print the computed value, i.e., 5, when it will be ready.

**Async**  Dart provides a language feature that allows simplifying the implementation of asynchronous methods. For example, the program in Example 2.7 can be encoded as follows.

```dart
127 main() {
128   Future<int> addResult = add(2, 3);
129   addResult.then((int value) {
130     print(value);
131   });
132 }
133
134 Future<int> add(x, y) async {
135   return x + y;
```
2.2. DART

The `add` function is automatically wrapped into a future. Asynchronous methods with return expressions of type `T` must declare their return type as `dynamic` or `Future<T>` as return type.

Sometimes asynchronous methods or functions need to be called with a blocking call, i.e., calls that stop until the value of a future is available. It is possible to perform a blocking call by using `await`. Let us consider Example 2.8. We can replace the asynchronous call as follows.

```dart
137 main() async {
138   print(await(add(2, 3)));
139 }
140
141 Future<int> add(x, y) async {
142   return x + y;
143 }
```

Example 2.9: A Dart program using an `await` call.

The expression `await(add(2, 3))` ensures that the program execution in the `main` function stops until the value computed by the future returned by `add(2, 3)` is ready.

2.2.3 Strong Mode

Recently, the Dart team has introduced strong mode [55], which is, in terms of programming flexibility, a more restrictive variant of the Dart static and dynamic type checkers described in the specification [21], which provides more guarantees. The Dart team has formally defined the strong mode type system and operational semantics [84] but, since it is not part of ECMA, it is not going to replace the actual Dart type system, which plays an important role in this dissertation. Instead, this novel type system is meant to provide additional support to programmers that need more static guarantees. The reason why strong mode is not part of the Dart standard is that it would not allow for enough programming flexibility, which goes against the Dart design principles. However, Gilad Bracha, one of the main contributor of the Dart specification, claims that there are different opinions among the Dart team: some of the engineers believe that Dart should provide more static guarantees, whereas some of the engineers think that Dart should provide more programming flexibility [65]. Although there is no specific plan to change the Dart type system in favor of a stricter system, if the Dart designer will decide that the language should provide more static guarantees, strong mode would be the best candidate as a sound type system.

Strong mode has two main goals: to allow trusting type annotations and to minimize the number of runtime type checks, still enabling optional typing and
allowing for programming flexibility. Nevertheless, in order to maintain backward compatibility, each warning caused by the standard Dart type system is turned into an error in strong mode.

Moreover, in order to guarantee that type annotations can be trusted, implicit upcasts are disabled at compile time. For example, the following assignment `int x = new Object()` is allowed by the standard type checker, because `Object` is a supertype of `int`, but disallowed in strong mode, because `Object` is not a subtype of `int`. Moreover, function subtyping and method overriding has contravariant input and covariant output. To summarize, the following Dart program is well-typed according to the standard type checker, but disallowed in strong mode.

```dart
typedef int F(Object);
void main() {
  int x = new Object();
  F y = (int p) => new Object();
}
```

The program is rejected because of two errors at lines 147 and 148, the latter implied by the restriction imposed to subtyping, i.e., `(int) → Object` is not a subtype of `F = (Object) → int`. Another important restriction concerns generic subtyping: circularity introduced by `dynamic` in the standard Dart subtype relation is ruled out in some cases. For example, although still `List<int> <: List<dynamic>`, `List<dynamic> !<: List<int>` in strong mode. Consider the following example.

```dart
main() {
  List<int> x = <dynamic>[''];
  int y = x[0];
  print(y * 2);
}
```

This program is well-typed according to the standard type system, and it fails at line 153 in both checked mode and production mode. However, the assignment at line 151 hides that, despite the variable `x` being of type `List<int>`, its value is a list that can actually contain elements of any type at runtime. The strong mode stricter subtyping relation address this issue: a type error is raised at line 151 because `List<dynamic>` is not a subtype of `List<int>`.

Besides a stricter subtyping relation, strong mode enables a type inference for non annotated variables: omitting a type annotation means inferring type, which might be `dynamic` in some cases, but if the type `dynamic` is specified explicitly, than no inference is performed. Instead, the standard type system considers omitting a type annotation semantically equivalent to using the type `dynamic`. Strong mode supports inference for top-level static fields, instance fields and methods, local variables, constructor calls and literals, and generic
method invocations. For example, the following Dart program is well-typed according to the standard type checker, but ill-typed in strong mode.

```dart
main() {
  var x = 0;
  var y = 0;
  var z = x + y;
  String str = z;
  Map<String, List<String>> m = { "id" : [ z ] };
}
```

The type inference assigns the type `int` to `x` and `y`, and it combines such informations to infer `int` for the variable `z`, resulting in a subtype error at line 159. Similarly, a type error is raised at line 160 because the right-hand side of the `m` assignment is a map from strings to a list that contains an integer, whereas the declared type of `m` specifies a map from strings to lists of strings. The type inference is also used to check against wrong overriding patterns. Consider the following example.

```dart
class Base {
  int f;
  var g = "g";
  int m() { return 0; }
}

class Derived extends Base {
  var f;
  var g;
  m() { return 0; }
}
```

Strong mode uses two different kind of type inference for overriding type checks. The first consists of propagating field and method declared types from a class to all its derived classes to complete missing type annotations. In the example above, the field `f` and the `m` return type in `Derived` have type `int`. It allows checking that, despite the lack of type annotations, elements are properly overridden. The second kind of inference only applies to instance variables. In the example above, `g` has type `String` because it is initialized as a string in class `Base`.

Strong mode uses an unsound and local type inference, so that, for example, the following program is type-correct.

```dart
foo(x) { return x; }
main() {
  int x = foo("name");
  print(x / 2);
}
```

Another important feature introduced by strong mode is generic methods. Consider the following example.
CHAPTER 2. BACKGROUND

```dart
179 List<int> unsafeCast(List<String> lst) {
180   return lst.map((String x) => x).toList();
181 }
182
183 void main() {
184   List<String> lstString = new List<String>();
185   lstString.add("hello");
186   List<int> lstInt = unsafeCast(lstString);
187   print(lstInt[0] / 2);
188 }
```

The `map` method has been defined in the Dart standard library as `Iterable<dynamic> map(f(E element))` within the `Iterable<E>` class, because maps create iterable structures that may contain elements of any type, including types not related to `E`. Thus, the expression `lst.map((String x) => x).toList()` has type `List<dynamic>`. However, because of strong mode, the `map` method can be redefined as `Iterable<T> map(f(E e))` in the class `Iterable<E>`. In the example above, the expression `lst.map((String x) => x).toList()` has type `List<String>` in strong mode, therefore it fails at line 180.

Strong mode is not only a restriction of the standard Dart type checker, but also a restriction of the checked mode execution. The main aim of restricting runtime type checks is that, combined with stricter type checks, it is possible to trust type annotations. As a consequence, if a Dart program can be compiled in strong mode, there are more static guarantees, which allows reducing the number of runtime type checks. In contrast, the standard type checker does not prevent Dart programs from being executed. Therefore, in order to reduce the number of runtime type checks, a very accurate type inference should be applied.

Strong mode execution and type system have the same restricted subtyping relation but, since checked mode and strong mode have different subtyping, the runtime behavior of the two modes are different, as the following example shows.

```dart
189 var x = new List<dynamic>();
190 if(x is List<int>) {
191   print("True Branch");
192 } else {
193   print("False Branch");
194 }
```

This program will print 'True Branch' in checked mode, because `List<dynamic>` is a subtype of `List<int>` in checked mode, but it will print 'False Branch' in strong mode, because `List<dynamic>` is not a subtype of `List<int>` in strong mode. In order to keep backward compatibility with checked mode and production mode, strong mode will raise a runtime error.
2.3. COQ

Strong mode semantics also refines types for generic classes and function types to blame type errors earlier than checked mode. Consider the following Dart program.

```dart
typedef int F(int);
main() {
    F f = (x) => x;
}
```

The right-hand side of the assignment at line 198 has type `(dynamic) → dynamic`, so that the assignment to `f` is allowed in both checked mode and strong mode. However, strong mode refines the runtime type of the closure `(x) => x / 2` to `(int) → int` after it is assigned to `f`.

### 2.3 Coq

Formal models and proofs are typically error-prone. To this end, researchers are starting using proof assistants, i.e., software tools to assist with the development of formal models and formal proofs by human-machine collaboration. There are several proof assistants, the most used by researchers are Coq, Agda, Isabelle, and ACL2. This section describes Coq, the proof assistant that we use to encode our formal model and proof described respectively in Chapter 3 and Chapter 4. Coq is a rather standard choice to model programming languages. For example, our formal model is inspired from Mackay et al. [50], which also uses Coq to encode the formal model of a dialect of Java. Coq gives the benefit of a machine-checked proof, since the proof assistant rejects all the proofs that contain errors. Moreover, Coq provides some mechanism to automatically prove parts of the proofs. However, automatic theorem proving is not decidable when considering the full intuitionistic logic; in most cases at least part of a proof has to be manually encoded.

#### 2.3.1 Introduction

Coq [43] is a proof assistant developed by INRIA, supporting an intuitionistic higher-order logic, higher-order functions, and dependent types. The formal models and proof encoded in Coq are first compiled to Gallina, a functional dialect of OCaml without side effects, and then executed. Figure 2.1 shows a proof session using a Coq IDE. The left side of the screen display a lemma and the respective proof. The green highlighted area represents the part of the proof that has already been verified by Coq to be correct, whereas the right side shows current state of the proof, i.e., the proof assumptions and the remaining statements to prove.

**Definitions and Types** At a first glance, Coq seems to be an ordinary ML-like functional programming language. As every programming languages
it provides literals, control structures, expressions and definitions. Let us consider the following Coq program that computes the factorial.

```coq
Definition fact n :=
    match n with
    | 0 ⇒ 1
    | _ ⇒ n * fact (n - 1)
end.

definition fact_8 := fact 8.
eval compute in fact_8.
```

Example 2.10: Factorial definition in Coq.

This program defines a `fact` function that computes the `n` factorial, using pattern matching and recursion, which is typical of functional programming languages. The definition of `fact_8` corresponds to the expression `fact 8`, which is only evaluated when executing `Eval compute in fact_8`. Similarly to OCaml, Coq supports a type inference that automatically infers the types of `fact`, `n`, and `fact_8`. It is possible to add type annotations to the program in Example 2.10 by obtaining the following Coq program.
2.3. COQ

```
Definition fact (n : nat) : nat :=
match n : nat with
| 0 ⇒ 1
| _ ⇒ n * fact (n-1)
end.

Definition fact_8 : nat := (fact : nat → nat) 8.

Eval compute in fact_8 : nat.
```

Example 2.11: Factorial definition in Coq.

The type nat → nat denotes a function type from natural numbers to natural numbers. While checking a formal model, the Check instruction is very useful to investigate the type of expressions. For example, the Check fact_8 will display nat.

Kinds Coq supports kinds, which is one of the most important features that allows for proof reasoning. We have shown in Example 2.11 that the factorial function parameter n has type nat. The term nat is a type constructor, i.e., an expression that allows specifying a type. A kind is a type of a type constructor. For example, the kind of nat is Set. Similarly, Set has kind Type. This can be checked by using the Check nat and Check Set instructions in Coq.

The most two relevant kinds are Set and Prop: any type denoting a value has the former kind, whereas any type denoting a proposition or logical predicated has the latter kind. We will summarize this with the following example:

```
Definition value : nat * nat : Set := (1, 2).
Definition property : (True ∨ True) : Prop := or_introl I.
```

Example 2.12: This example shows a simple use of kinds in Coq.

The meaning of the definition at line 218 is the following: value has type nat * nat, which has kind Set and its value is the pair of natural numbers (1, 2). The meaning of the definition at line 219 is the following: property has propositional type True ∨ True, which has kind Prop and its value is or_introl I, which is a proof of the property True ∨ True. The code at line 219 is equivalent to the following Coq lemma and proof.

```
Lemma property : True ∨ True.
Proof.
left. exact I.
Qed.
```

Example 2.13: This example shows a simple Coq proof.

The proof proceeds as follows: The left-hand side of True ∨ True, which is True, holds. We can show that True holds, since there exists a value I of type True. The fact that this lemma and proofs are equivalent to the definition at
line 219 in Example 2.12 means the following: proving that \( \text{True} \lor \text{True} \) holds, is equivalent to provide a value of that type. Coq exploit this equivalence to compile the lemma and proof in Example 2.13 to a definition similar to line 219 in Example 2.12.

### 2.3.2 Induction

The induction principle is native in Coq and this is what makes it easy to define programming language. We have seen in Section 2.3.1 how to define terms and predicates using the `Definition` construct. It is possible to express definition by induction using the `Inductive` construct. We can use the inductive definitions to define new types, more precisely new sets or new propositions, by specifying their inhabitants by induction.

**Inductive Sets**  The following example shows how natural numbers are defined in Coq.

```coq
Inductive nat : Set :=
| 0 : nat
| S : nat → nat.
```

It is equivalent to the following mathematical definition.

\[
\begin{align*}
0 & \in \mathbb{N} \\
n & \in \mathbb{N} \rightarrow S(n) \in \mathbb{N}
\end{align*}
\]

where \( S : \mathbb{N} \rightarrow \mathbb{N} \). The following example shows how to define the lambda-calculus syntax in Coq using the inductive definition.

```coq
Inductive Lambda : Set :=
| x : nat → Lambda
| abs : nat → Lambda → Lambda
| app : Lambda → Lambda → Lambda.
```

That is equivalent to the following.

\[
\begin{align*}
n & \in \mathbb{N} \rightarrow x_n \in \lambda \\
n & \in \mathbb{N} \land t \in \lambda \rightarrow abs(n, t) \in \lambda \\
t_1 & \in \lambda \land t_2 \in \lambda \rightarrow app(t_1, t_2) \in \lambda
\end{align*}
\]

The point of such inductive Coq definition is that lambda terms are expressed as functions that return a lambda term, so that it is possible to combine infinitely many times lambda terms. For example, the identity function \( \lambda x.x \) can be encoded in Coq according to our syntax as `abs 0 (var 0)` where 0 is the index that identifies the \( x \) variable. It is easy to see that such term, still has type `Lambda`. 
## 2.3. COQ

### Inductive Properties

In Coq it is possible to define properties by induction too. The following example shows how to define the property `contains_app` by induction, that holds if and only if a lambda term contains a lambda application.

```coq
Inductive contains_app : Lambda → Prop :=
| c_abs : ∀ n t, contains_app t → contains_app (abs n t) |
| c_app : ∀ t1 t2, contains_app (app t1 t2).
```

The property `contains_app` is defined as a function from a `Lambda` to `Prop`, that is true or false basing on the lambda term. The two inductive cases here are values whose types are properties that describes when `contains_app` holds. More specifically `c_abs` shows that `contains_app` holds if and only if the lambda term `λx.n.t` contains an application, that is `t` contains an application, and `c_app` shows that `contains_app` holds for each lambda application `t1 t2`, that is each lambda application contains a lambda application (that is trivial).

### 2.3.3 Dependent Types

Like many functional programming languages, Coq does not have subtyping. Suppose we want to implement the following function.

```coq
Definition pred (n : nat) :=
match n with
| 0 ⇒ false
| _ ⇒ n - 1
end.
```

This function computes the predecessor of a natural number, or `false` if the input is zero. This term is not well-typed because the function return type can be both `nat` or `bool` and the type inference is not powerful enough to infer the right type. Can we provide a type manually without using union types (that would make the function implementation more complex)? Yes, using the dependent types. A dependent type can be seen as a function type that depend on one or more of the function parameters. Coming to our example, the following type is the right dependent type.

```coq
Definition pred_type :=
∀ n : nat, match n with
| 0 ⇒ bool
| _ ⇒ nat
end.
```

```coq
Definition pred : pred_type :=
fun n =>
```

match n with
| 0 ⇒ false
| _ ⇒ n - 1
end.

The pred_type type can be read as: “function type from a natural number n to num if n is zero, to a boolean otherwise”.

Dependent Types for Set

We can use dependent types to emulate polymorphic parameter types like the following example shows.

Definition identity : ∀ T : Set, T → T :=
fun (T : Set) (x : T) ⇒ x.

The identity function defined here has type ∀ T : Set, T → T and it is encoded as a curried function whose first parameter is the type, and the second parameter is the identity parameter whose type depend on the first parameter. This is another example of dependent types. The identity parameter can be any set, that is a number, a boolean, a list, a function, etc.

Dependent Types for Prop

It is possible to provide the same identity function for any type, and for any proposition.

Definition identity : ∀ X : Prop, X → X :=
fun (X : Prop) (x : T) ⇒ x.

We can give are two different interpretations for this definition

• identity is a function of type ∀ X : Prop, X → X and implementation
  fun (X : Prop) (x : T) ⇒ x
• identity is a name for the theorem ∀ X : Prop, X → X with proof fun (X : Prop) (x : T) ⇒ x

So, accordingly to the curry-howard correspondence, in Coq a proof P for a theorem T, is just the value P of a name of type T. Indeed, here we have just proven that each property implies itself!

2.3.4 How to Build Proofs

We have just seen that ∀ X : Prop, X → X can be proven by providing a term of such type. It is not the way proofs are built, but it is useful to understand how Coq internally works. Coq provides a set of instruction to elaborate a proof, called tactics. We can rewrite our identity definition as follow.

Theorem identity :
∀ X : Prop, X → X.
Proof.
intro X.
This definition is equivalent to the definition given in the previous section. Indeed, the instruction `Print identity` prints the same Coq term defined in the previous section.

Similarly to an handwritten proof, a Coq proof consists of a set of goals, which are prepositions to prove, and a set of assumptions, which are prepositions assumed to hold because of a proven lemma, axiom, or as a result of decomposing a goal. Considering the `identity` proof we have the following steps.

1. `intro X`: introduces in the assumptions that we have a variable $X$ of type $\text{Prop}$. The goal to prove becomes $X \rightarrow X$. It is equivalent to the forall introduction rule in the sequent calculus for the intuitionistic logic.

2. `intro x`: introduces in the assumptions that we have a variable $x$ of type $X$, that is, we assume to have a proof of $x$, that is a property of type $X$. The goal becomes $x$.

3. `exact x`: shows that we can prove $x$ since we have it in the assumption

Coq provides many different tactics, as well as proof scripts, and third party libraries. We will mention just the most used tactics.

- `intro`: introduces the first implication premise or universally quantified variable from the goal proposition into the theorem assumptions. For example, given the goal $\forall x. x > 1 \rightarrow x + 1 > 1$, `intro` changes the goal into $x > 1 \rightarrow x + 1 > 1$ and introduces the assumption that there is a variable $x$ of type $\text{nat}$.

- `intros`: the same as `intro` but it introduces all the premises.

- `exact`: allows proving a goal which is an assumption. For example, given the goal $x = y$, and an assumption $H : x = y$, i.e., an assumption named $H$ that states $x = y$, then `exact H` proves the given goal.

- `apply`: applies the consequent of a rule.

- `inversion`: applies an inverse pattern matching in an inductive predicate in one of the assumptions. For example, given the assumption $H : x > 0$, since the $>$ relation is defined inductively on the structure of natural numbers, `inversion H` split the current goal into a goal where $x = 0$ and a goal where $x = S(m)$ for some natural number $m$. This tactic is very useful for case analysis proof.
CHAPTER 2. BACKGROUND

• induction: applies the induction principle to the inductive definitions. This tactic is similar to inversion but it also adds induction hypothesis in the assumptions.

• auto: this tactic tries to automatically prove the current goal, if it is possible.

• absurd: start a proof for absurd.

• left, right: respectively states that the left-hand side and the right-hand side of a disjunctive goal hold. For example, given the goal \( A \lor B \), where \( A \) and \( B \) are two properties, left changes the goal into \( A \).

• split: split a conjunctive goal into two subgoals. For example, given the \( A \land B \), where \( A \) and \( B \) are two properties, split produces two subgoals: \( A \) and \( B \). Intuitively, this tactic says, “we will prove first that \( A \) holds, and then that \( B \) holds”.

We will now show an example on how Coq can be used to formalize and prove properties about programming languages. The following Coq code inductively defines a programming language that we call “exp”.

```
Inductive term : Set :=
| t_true : term
| t_false : term
| t_if : term → term → term → term
| t_num : nat → term.
```

This language contains the boolean terms \( t_{\text{true}} \) and \( t_{\text{false}} \), the conditional expression \( t_{\text{if}} \, \text{cond} \, t_{\text{true}} \, t_{\text{false}} \) defined as a function from three terms (\( \text{cond}, \text{true}, \text{false} \)) to a term, and the numeric expression \( t_{\text{num}} \, n \) where \( n \) is a natural number.

The following Coq fragment defines the types syntax for “exp”.

```
Inductive term_type : Set :=
| ty_bool : term_type
| ty_nat : term_type.
```

We include the boolean and natural number types, respectively \( \text{ty\_bool} \) and \( \text{ty\_nat} \). Follows the definition of the typing judgment for “exp”.

```
Inductive t_typing : term → term_type → Prop :=
| ty_true : t_typing t_true ty_bool
| ty_false : t_typing t_false ty_bool
| ty_if : ∀ \text{cond} \, \text{true} \, \text{false} \, \text{T1} \, \text{T2},
  t_typing \text{cond} \, \text{ty\_bool} \, \rightarrow
  t_typing \text{true} \, \text{T1} \, \rightarrow
  t_typing \text{false} \, \text{T2} \, \rightarrow
  \text{T1} = \text{T2} \, \rightarrow
  t_typing \,(\text{t\_if} \, \text{cond} \, \text{true} \, \text{false}) \, \text{T1}
```

This Coq fragment is equivalent to the following mathematical definition of the typing judgment $t : T$, i.e., $\text{t_typing } t T$, where $t$ is a term and $T$ a type.

\[
\begin{align*}
\text{[TY_TRUE] } & \quad t_{\text{true}} : ty_{\text{bool}} \\
\text{[TY_FALSE] } & \quad t_{\text{false}} : ty_{\text{bool}} \\
\text{[TY_IF] } & \quad \frac{\text{cond} : ty_{\text{bool}} \quad \text{true} : T \quad \text{false} : T}{t_{\text{if cond true false}} : T} \\
\text{[TY_NUM] } & \quad \frac{n \in \mathbb{N}}{t_{\text{num n}} : ty_{\text{nat}}} 
\end{align*}
\]

We will now define the $\text{no_nums}$ predicate, which holds only if the term does not contain any occurrence of $t_{\text{num n}}$.

\[
\begin{align*}
\text{Inductive no_nums : term } \rightarrow \text{ Prop :=} \\
| \text{no_nums_in_true : no_nums t_{\text{true}}} \\
| \text{no_nums_in_false : no_nums t_{\text{false}}} \\
| \text{no_nums_in_if : } \forall \text{ cond true false}, \quad \text{no_nums cond } \rightarrow \quad \text{no_nums true } \rightarrow \quad \text{no_nums false } \rightarrow \quad \text{no_nums (t_{\text{if cond true false})}.}
\end{align*}
\]

The inversion and induction tactics applied on the typing derivation enforce a case analysis by pattern matching a typing judgment with all the possible rules that can show this derivation hold. This is crucial in making most of the proofs of soundness, which also explain why Coq fits well with formalization of programming languages. The following lemma states that if a term in the “exp” language has type $ty_{\text{bool}}$ no number can occur (the comments show assumptions, goals, and the effect of a tactic, step by step).

\[
\begin{align*}
\text{Lemma } \text{no_nums_in_bools : } \forall t, \quad & \text{t_typing } t ty_{\text{bool}} \rightarrow \text{no_nums } t. \\
\text{Proof.} \\
\text{intros } t \ \text{Typing}. \\
\text{By induction on the structure of the term } t \quad \text{*)} \\
\text{induction } t. \\
\text{Case } t = t_{\text{true}} \\
\end{align*}
\]
CHAPTER 2. BACKGROUND

- **Assumptions:** t_typing t_true ty_bool
- **Goal:** no_nums t_true

- **constructor.**
  (* From the constructor no_nums_in_true, it follows
   * that t_true does not contain any numbers *)

- **constructor.** (* Similarly, t_false does not contain
  any number *)

- **Case** t = t_if cond true false
  (* Assumptions
   * Typing: t_typing (t_if t1 t2 t3) ty_bool
   * by induction hypothesis
   * IHt1: t_typing t1 ty_bool → no_nums t1
   * by induction hypothesis
   * IHt2: t_typing t2 ty_bool → no_nums t2
   * by induction hypothesis
   * IHt3: t_typing t2 ty_bool → no_nums t3
   * Goal: no_nums (t_if t1 t2 t3)

- **inversion Typing.** (* We generate one subgoal
  for each typing rule where
  t_if t1 t2 t3 has type
  bool, i.e.,
  the rule ty_if *)

- **subst.** (* After inversion Typing, the pattern
  matching
  a lot of equalities. subst apply the
  equalities to the premises. *)

- **assert(A := IHt1 H2).**
- **assert(B := IHt2 H3).**
- **assert(C := IHt3 H5).**

- (* We can now decompose our goal no_nums (t_if t1 t2 t3)
  applying the rule
  no_nums_in_if which leads to 3 subgoals*)
2.3. **COQ**

350   no_nums t1  
351   no_nums t2  
352   no_nums t3 *)
353     constructor.
354   (* These three subgoals now coincides with A, 
355       B, and C *)
356     exact A.  
357     exact B.  
358     exact C. 
359   (* Case t = (t_num n) for some n 
360     * Assumptions 
361     * Typing: t_typing (t_num n) ty_bool 
362     * Goal: no_nums (t_num n) 
363     *)
364   (* Clearly, no typing rule allow concluding 
365     * that t_num n has type ty_bool 
366     * therefore our assumption will never hold. 
367     * The tactic inversion is aware of 
368     * it and it therefore invalidate this goal 
369     *)
370     inversion Typing. 
371     Qed.
Chapter 3

Fletch

The first part of the hypothesis of this dissertation stated in Section 1.1 is that, despite the Dart type system being unsound, it is possible to identify natural restrictions of the type system that ensure the absence of message-not-understood errors. In order to test this part of the hypothesis, we formalize a language called Fletch, which faithfully models a subset of Dart, including syntax, operational semantics, and type system. We will use this formal model in Chapter 4 as a starting point to identify the natural restriction of the Dart type system mentioned above and to investigate the guarantees it provides.

Fletch includes just enough elements from Dart to characterize the core of the Dart type system and the associated dynamic semantics. The formal model is inspired by Featherweight Java [42]. Additionally, we provide a Coq formal model (see http://www.brics.dk/fletch/) inspired by the Coq model of FJ+AI [50], an extension of Featherweight Java with assignment, mutable, and immutable objects.

This chapter presents the following contributions:

- We present a core calculus of Dart called Fletch, thereby elucidating the type system of the Dart language.
- We encode the formal model of the syntax, the small-step operational semantics for both checked mode and production mode, and the type system in Coq.

This chapter is extracted from Ernst et al. [25]. The operational semantics rules presented in this chapter are extended with the congruence rules shown in Figure 3.9. Moreover, the subtyping rules in Figure 3.6 defined in this chapter models the Dart type system, whereas Ernst et al. [25] present the rules together with a restriction of the type system, which we will show in Chapter 4. The remaining parts of Ernst et al. [25] are in Chapter 4.
Figure 3.1: Fletch syntax. Boxed parts occur only at runtime.

3.1 Syntax

Figure 3.1 shows the syntax of Fletch. The Fletch syntax faithfully models the Dart syntax in [21] apart from a small modification in anonymous functions that we will discuss later and the boxed parts that is an extension of the Dart syntax used to define the runtime semantics. This is the standard approach to model calculi (see for example [42]) and typed closures.

The declaration categories $CL$, $M$, and $F$ define classes, methods, and fields, and they are unsurprising. As usual, $a$ denotes the possibly empty list $a_1, ..., a_n$, $n \geq 0$.

Expressions ($e$) specify computations including variable and property lookup, assignments, function invocations, object creation, anonymous functions, and runtime expressions.

Variables ($y$) denote method arguments ($x$) and the predefined names `this` and `null`. Locations ($l$) are variable locations ($\tau$) or heap locations ($\iota$), which we will discuss in Section 3.2 Names of fields, methods, classes, method arguments, type parameters, variable locations, and heap locations are disjoint, and denoted by $f$, $m$, $c$, $x$, $X$, $\tau$, and $\iota$ respectively. In a slight abuse of notation we will use grammar nonterminals to indicate sets of terms; for example, $e$ stands for the set of all syntactic expressions and we also use $e$ as a metavariable that ranges over this set.

Frame expressions $[T, e]$ arise when a function is invoked. Such an expression carries the declared return type of the invoked function. This enables a check on the type of the returned value, as required for checked mode execution.

The anonymous function syntax $T (\overline{G x}) \Rightarrow e$ is slightly different from the
corresponding syntax in Dart, which omits the return type $T$. It would be easy to introduce a prepossessing phase that obtains the statically known type of the returned expression $e$ and adds it as the explicit return type. In other words, the explicitly declared return types for Fletch anonymous functions do not add essential information to programs. However, they do eliminate the need for some complicated machinery to compute the statically known return type whenever needed—which includes the dynamic semantics in checked mode. We deviate slightly from Dart here to avoid unnecessary complexity.

The class definitions in a program are modeled as a class table $CT : c \mapsto CL$, which maps a finite set of class names into class definitions. We use ‘$\mapsto$’ to indicate a partial function.

A class table $CT$ is well-formed iff $\text{Object} \notin \text{dom}(CT)$, but every other class name used in $CT$ is defined, and inheritance is acyclic. A Fletch program is a pair $(CT, e)$ where $CT$ is a class table and $e$ is an expression, and it is well-formed iff $CT$ is well-formed and both $e$ and all expressions within all classes in $CT$ contain only well-formed types and identifiers that are defined in the relevant environment.

### 3.2 Operational Semantics

As we explain in Section 2.2, the Dart specification [21] defines two execution modes: checked mode and production mode. The former is an extension of the latter with additional runtime type checks at write operations. This section shows a formalization of both checked mode and production mode executions with small step operational semantics rules.

#### 3.2.1 Semantic Entities

The operational semantics of Fletch requires more complex semantic entities than many other calculi. We need to model a heap in order to express mutability, which we cannot ignore, because the semantics of lexically scoped closures and checked mode execution depend substantially on being in a mutable rather than an immutable setting. We need an extra level of indirection on method arguments in order to model first class closures and lexical nesting. Since local variables would be given the same treatment as method arguments, had they been included in the model, we will use the word *variable* as interchangeable with method arguments.

We model the heap by the maps denoted by $\sigma$, and the indirection for variables by the maps denoted by $\nu$. The former maps each heap location $\iota \in \text{LocH}$ to an object or a closure, and the latter maps each variable location $\tau \in \text{LocV}$ to a type and a heap location, as shown in Figure 3.2. We use the word *heap* to designate the former, *variable environment* to designate the latter, and *environment* to designate any of the two. $\text{LocH}$ and $\text{LocV}$ are disjoint, countably infinite sets.
A good intuition about $\nu$ is that it is a log that models all the local state used in the execution so far. Each variable $x$ is systematically replaced by an invocation specific variable location $\tau$, which ensures that variables are aliased across all nested scopes for each invocation of a method, but distinct for different method invocations.

We illustrate this using an example. Assume that a method $m$ is invoked and returns an object containing two closures $cl_1$ and $cl_2$, where $cl_1$ will mutate a variable $x$ and $cl_2$ will use $x$. An execution of $cl_1$ changing $x$ must then work such that $cl_2$ evaluates $x$ to the new value. On the other hand, no such interaction is allowed between $cl_2$ and a closure created from the same expression as $cl_1$ during a different invocation of $m$. By the use of variable environments, all occurrences of $x$ will be replaced by a variable location $\tau_1$ in the first invocation, and by $\tau_2 \neq \tau_1$ in the other invocation. Mutations of $x$ will modify the given variable environment to map $\tau_1$, resp. $\tau_2$, to new heap locations.

In this way, we model all the bindings in the runtime stack, including the ones in activation records that have already been discarded. An alternative approach would be to model the runtime stack directly. Our approach enables a significant simplification: we avoid modeling migration of variables to the heap in case a closure using variables in an activation record escapes out of the corresponding method invocation, and we avoid specifying how to detect that situation.

To be able to express checked mode execution, variable environments $\nu$ provide not only a heap location for every variable location, but also the statically declared type of the corresponding variable, as represented by the syntactic metavariable $G$ from Figure 3.1.

We also introduce objects, closures, field maps, and method maps. An object $o$ contains its runtime type $G$, a map $\phi$ from field names to declared types and heap locations, and a map $\mu$ from method names to heap locations. A closure is simply represented by an anonymous function $fn$. There is no need to equip a closure with an environment: upon invocation it contains no free variables, because they are all replaced by variable locations, and $this$ is replaced during object creation by a variable location $\tau_{this}$.
3.2. OPERATIONAL SEMANTICS

$$\text{CT}(c) = \text{class } c \langle X < \ldots > \text{ extends } N \{ F \ldots \}$$

$$F_1 = [G_1 / X] F \quad F_2 = \text{fields}([G_1 / X] N) - F_1$$

$$\text{fields}(c<G_1>) = F_1 \cup F_2$$

$$\text{CT}(c) = \text{class } c \langle X < \ldots > \text{ extends } N \{ \ldots M \}$$

$$M_1 = [G_1 / X] M \quad M_2 = \text{methods}([G_1 / X] N) - M_1$$

$$\text{methods}(c<G_1>) = M_1 \cup M_2$$

Figure 3.3: Semantics lookup definitions. With a small abuse of notation, the $A - B$ operator excludes all the fields/methods in $A$ whose name occurs in $B$.

$$\text{typeof}(\iota, \sigma) = \begin{cases} G & \text{if } \sigma(\iota) = (G, \_ , \_) \\ \{ G \} \rightarrow T & \text{if } \sigma(\iota) = T (G \_ x) \Rightarrow e \end{cases}$$

$$\text{typeof}(\tau, \nu) = G \text{ if } \nu(\tau) = (G, \_)$$

Figure 3.4: Definition of $\text{typeof}(\iota, \sigma)$, which looks up the dynamic type of a heap location $\iota$ in the heap $\sigma$, and $\text{typeof}(\tau, \nu)$, which looks up the declared type of a variable location $\tau$ in the variable environment $\nu$.

Notationally, $[\tau/y]e$ denotes capture avoiding substitution in a Fletch expression $e$: all free occurrences of $y$ in $e$ are replaced by $\tau$. The same notation is used for substitution of types, etc. We also use brackets to denote maps of any types, i.e., finite, partial functions, listing each binding in the map. For instance, $[\tau \mapsto (G, \iota)]$ is the map that maps $\tau$ to $(G, \iota)$, and $[]$ is the map that is everywhere undefined.

The state of a Fletch program during execution is represented by $s$ (see Figure 3.2). The class table, $\text{CT}$, is frequently consulted during execution. It is constant throughout any program executions so we will leave it implicit, as is common in object calculi since Featherweight Java [42].

Some locations are predefined, e.g., the null pointer, which motivates the use of the base environments $\nu_{\text{base}} = []$ and $\sigma_{\text{base}} = [\iota_{\text{null}} \mapsto o_{\text{null}}]$, where $o_{\text{null}} = (\bot, [], [])$ represents the predefined object $\text{null}$. Every runtime environment will extend one of these.

3.2.2 Auxiliary Definitions

This section shows the definitions of auxiliary predicates and functions that we use in the operational semantics. First, we need to define $\text{fields}$ and
\[ \Delta \vdash \bot \ll T \quad \Delta \vdash T \ll \text{dynamic} \quad \Delta \vdash T \ll T \]
\[ \Delta \vdash T_1 \ll T_2 \quad \Delta \vdash T_2 \ll T_3 \]
\[ \Delta \vdash T_1 \ll T_3 \]
\[ \Delta \vdash X \ll \Delta(X) \]
\[ \Delta \vdash c < G_1 \ll c < G_2 \]
\[ \text{CT}(c) = \text{class } c < X < N \text{ extends } d < G_1 \{ \cdots \} \]
\[ \Delta \vdash c < G_2 \ll [G_2/X]d < \text{dynsub}(G_1) \]

Figure 3.5: Typing specificity.

methods. Figure 3.3 show the definition. As Figure 3.8 shows, the dynamic semantics of Fletch requires the ability to answer certain simple type-related questions. It must be possible to determine the runtime types of objects and closures and the statically declared types of variables. Figure 3.4 shows the definition of typeof, which takes a heap location \( \iota \) or a variable location \( \tau \) and determines the requested type.

3.2.3 Subtyping

Before defining the operational semantics, we need to define the subtyping relation, which is used in both the static type system and the checked mode runtime type checks. A type environment \( \Delta \) is a finite map from type variables to class types. We use the notation \( X_1 <: N_1, \ldots, X_n <: N_n \) for explicit listings, where \( <: \) is also used for the subtyping relation described later. Each element \( X <: N \) indicates that \( X \) must be bound to a subtype \( N' \) of \( N \).

Typing specificity is a partial order on types. We say that \( T_1 \) is more specific than \( T_2 \) in the type environment \( \Delta \) iff \( \Delta \vdash T_1 \ll T_2 \) is provable according to Figure 3.5. Note that the rules follow the declared extends relationship, but they leave some special cases to subtyping (defined below).

Type rules for type specificity do not describe the full subtype relation for Fletch types. The special type annotation \text{dynamic} allows the programmer to leave a type unspecified in the program, and unchecked by the compiler. The type \text{dynamic} behaves as a supertype and as a subtype of any other type in the language, and no type warnings ever appear for expressions of type \text{dynamic}. Generic type parameters may also be declared as \text{dynamic}.

An unfortunate side effect of the type \text{dynamic} is that the subtype relation in Fletch is not transitive. For example, it is the case that \( \Delta \vdash \text{List}<\text{int}> <: \text{List}\langle\text{dynamic}\rangle \) and \( \Delta \vdash \text{List}\langle\text{dynamic}\rangle <: \text{List}<\text{String}> \). If the rules had been transitive we could conclude \( \Delta \vdash \text{List}<\text{int}> <: \text{List}<\text{String}> \), which should not hold. Transitivity only holds among class types, but not when the type \text{dynamic} is used.
3.2. OPERATIONAL SEMANTICS

We need to define a simple syntactic transformation of types to promote `dynamic` to the bottom type:

\[
\text{dynsub}(T) = \begin{cases} 
\bot & \text{if } T = \text{dynamic} \\
\text{assignable}(G_1, G_2) & \text{if } T = \text{assignable}(T_1, T_2) \text{ or } T_2 = \text{void} \\
T & \text{otherwise}
\end{cases}
\]

With `dynsub(T)`, we can define the subtype relation as shown in Figure 3.6. Notice that `[Sub-Dyn-Sub]` makes `dynamic` a subtype of all other types. This ensures \( \Delta \vdash \text{List<dynamic>} \prec \text{List<String>} \), as \( \Delta \vdash \text{List<\bot>} \prec \text{List<String>} \), which solves the previously mentioned transitivity problem.

We also use `dynsub` in the definition of typing specificity for class inheritance (last rule in Figure 3.5). As an example, for a class defined by `class DynList extends List<dynamic> {}`, we have `DynList ≪ List<\bot> ≪ List<int>`, and therefore `DynList ≪ List<int>` as one would expect.

The notion of assignability in object-oriented languages often coincides with subtyping. As Figure 3.7 shows, the assignability relation in Fletch is strictly larger than the subtyping relation: types are assignable if either of them is a subtype of the other. Type parameters are treated likewise. While this clearly allows programmers to assign values to variables that cause runtime failures in checked mode, the static type checker does reject direct assignments between unrelated types. As an example, the following program is type correct by these rules:

```java
class C<X,Y> {
    int x;
    C<String,Object> y;
    void initX() {this.x = new Object();}
}
```

\(^1\)The Dart language specification erroneously omits this substitution of `dynamic`; the language designers have confirmed that this is indeed an error, and Google’s implementation agrees.
3.2.4 Dynamic Semantics

We specify the dynamic semantics of Fletch in terms of a small-step operational semantics $\rightarrow$ that relates States to States, that is, each configuration is a triple $(\nu, \sigma, e)$. The rules for expression evaluation in Fletch are shown in Figure 3.8 and Figure 3.9. Every successfully terminating expression evaluates to a heap location $\iota$, which is the only kind of value that Fletch supports. Expression evaluation may have side effects in terms of updates to the heap or the variable environment.

We use the shorthand $\nu[\tau \mapsto \iota]$ standing for $\nu[\tau \mapsto (G, \iota)]$ where $\nu(\tau) = (G, \iota')$ for some $\iota'$. Similarly, $\nu(\tau) = \iota$ means that there exists a $G$ such that $\nu(\tau) = (G, \iota)$. Evaluation of a variable location $[E\text{-Var-Read}]$ amounts to a lookup in $\nu$ for a location $\tau$.

Assignment to a variable location $[E\text{-Var-Write}]$ updates the variable environment $\nu$ to map that variable location to the given value. The subtype check in the premise is included iff the execution uses checked mode, in which case it is enforced that the runtime type of the new value $\iota$ is a subtype of the statically declared type of the variable location $\tau$. Assignment to a field $[E\text{-Field-Write}]$ looks up the object at $\iota_1$ and creates a new heap $\sigma'$ that differs from the old heap only at $\iota_1$, which contains the object updated only at the selected field $f$ to have the new value $\iota_2$. Note that field assignment requires the field to exist, both in checked mode and in production mode. In checked mode it is also enforced that the new field value conforms to the declared type. Evaluation of a field $[E\text{-Field-Read}]$ or a method $[E\text{-Method-Read}]$ is straightforward, and the null literal $[E\text{-Null}]$ evaluates to the null heap address.

The new expression $[E\text{-New}]$ creates and initializes a fresh object based on the given class, with a null valued fields map, and with closures corresponding to the method declarations in the methods map. Occurrences of this in method bodies are replaced by the location $\tau_{\text{this}}$ of the new object; the method arguments will be similarly replaced upon invocation of each method. The auxiliary functions fields and methods collect the set of fields and methods,
respectively, for a given type, taking class inheritance and type parameter substitution into account, as shown in Figure 3.3.

We use \texttt{name} to extract the field names and method names, that is \texttt{name}(F) = f for a field declaration \( F = G f \) and \texttt{name}(M) = m for a method declaration \( M = T m(G x) \) \{ \texttt{return} e; \}.

Closure creation \texttt{[E-Func]} stores the given closure in the heap and evaluates to the corresponding heap location. Closure invocation \texttt{[E-Call]} evaluates the body of the function in a new variable environment \( \nu' \) created by combining the current variable environment \( \nu \) with bindings from the formal to the actual arguments of the invocation, replacing variables by fresh variable locations in the body. In checked mode, the dynamic types of the actual arguments are checked against the formal argument types. The resulting expression packages the declared return type \( T \) of the closure together with the closure body, which is needed in order to be able to check that the dynamic return value conforms to the declared return type. The return step \texttt{[E-Return]} performs this check, if in checked mode, and produces the contained value.

Every terminating expression evaluates to a heap location \( \iota \), which is the only kind of values that Fletch supports. Expression evaluation may have side effects in terms of updates to the heap or the variable environment.

The Dart language includes getter and setter methods. They can be explicitly declared, but otherwise for each declared field the compiler automatically provides a getter and a setter, and for each method a getter returning a “tear-off” closure for the method. Although all fields are private in Dart, they can be accessed from other classes by implicit uses of getters and setters. For instance, if class \( C \) contains field \( f \) then \texttt{new C().f} will call the automatically generated getter method named \( f \) that returns the value of the field \( f \). Similarly, \texttt{new C().f = e} will call the generated setter method named \( f= \) that sets the field \( f \) to the value of its argument \( e \). Since Dart does not introduce any significant novelties about getters and setters, we only model automatically generated getters and setters. For simplicity we do this by means of primitive field read/write and method read operations. This does differ from the language specification, but it is a faithful model of the core of the language.
\[
\begin{align*}
\text{[E-Var-Read]} & \quad \frac{\nu(\tau) = \iota}{(\nu, \sigma, \tau) \rightarrow (\nu, \sigma, \iota)} \\
\text{[E-Var-Write]} & \quad \frac{\nu' = \nu[\tau \mapsto \iota]}{(\nu, \sigma, \tau = \iota) \rightarrow (\nu', \sigma, \iota)} \quad \frac{\quad \vdash \text{typeof}(\iota, \sigma) <: \text{typeof}(\tau, \nu)}{}
\end{align*}
\]

\[
\begin{align*}
\sigma(t_1) &= (c \triangleright G, \phi, \mu) \quad \phi(f) = (G', \_)
\sigma' &= \sigma|_{t_1} \mapsto (c \triangleright G, \phi[f \mapsto (G', \iota_2)], \mu) \\
\frac{\quad \vdash \text{typeof}(\iota_2, \sigma) <: G'}{(\nu, \sigma, \iota_1.f = \iota_2) \rightarrow (\nu, \sigma', \iota_2)}
\end{align*}
\]

\[
\begin{align*}
\text{[E-Field-Write]} & \quad \frac{\sigma(t_1) = (\_, \phi, \_)}{(\nu, \sigma, \iota_1.f) \rightarrow (\nu, \sigma, \iota_2)} \\
\text{[E-Field-Read]} & \quad \frac{\sigma(t_1) = (\_, \phi, \_)}{(\nu, \sigma, \iota_1.f) \rightarrow (\nu, \sigma, \iota_2)} \\
\text{[E-Method-Read]} & \quad \frac{\sigma(t_1) = (\_, \_, \mu)}{(\nu, \sigma, \iota_1.m) \rightarrow (\nu, \sigma, \iota_2)}
\end{align*}
\]

\[
\text{[E-Null]} \quad \frac{\langle \nu, \sigma, \text{null} \rangle \rightarrow (\nu, \sigma, \iota_{\text{null}})}{}
\]

\[
\begin{align*}
F &= \text{fields}(c \triangleright G) \quad M &= \text{methods}(c \triangleright G) \\
o &= (c \triangleright G, \{\text{name}(F) \mapsto (\text{type}(F), \iota_{\text{null}}), \text{name}(M) \mapsto \iota_{m_i}\}) \\
\sigma_0 &= \sigma[\iota \mapsto o] \quad \text{where } \iota \text{ is fresh} \\
\nu' &= \nu[\tau_{\text{this}} \mapsto (c \triangleright G, \iota)] \\
\forall M_i \in M: \sigma_i &= \sigma_{i-1}[\iota_{m_i} \mapsto T_i m_i(G_i x_i) \Rightarrow [\tau_{\text{this}/\text{this}} e_i] \text{ where } M_i = T_i m_i(G_i x_i) \{\text{return } c_i;\} \text{ and } \iota_{m_i} \text{ is fresh}} \\
\frac{\langle \nu, \sigma, \text{new } c \triangleright G() \rangle \rightarrow \langle \nu', \sigma_n, \iota \rangle}{\text{[E-New]}}
\end{align*}
\]

\[
\begin{align*}
\text{[E-Func]} & \quad \frac{\sigma' = \sigma[\iota \mapsto T(G x) \Rightarrow e]}{(\nu, \sigma, T(G x) \Rightarrow e) \rightarrow (\nu, \sigma', \iota)} \\
\text{[E-Call]} & \quad \frac{\nu' = \nu[\tau \mapsto (G, \iota)] \quad \text{where } \tau \text{ is fresh}}{(\nu, \sigma, \iota_0(\tau)) \rightarrow (\nu', \sigma, [\tau/\tau][e])} \quad \frac{\quad \vdash \text{typeof}(\iota, \sigma) <: G}{(\nu, \sigma, \iota_0(\tau)) \rightarrow (\nu', \sigma, [\tau/\tau][e])} \\
\text{[E-Return]} & \quad \frac{\langle \nu, \sigma, \iota_0(\tau) \rangle \rightarrow (\nu, \sigma, \iota)}{(\nu, \sigma, \iota_0(\tau)) \rightarrow (\nu, \sigma, \iota)}
\end{align*}
\]

Figure 3.8: Computational rules for expressions in Fletch. The boxed premises involving typeof are omitted for production mode execution, but included for checked mode execution.
3.2. OPERATIONAL SEMANTICS

\[ \langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle \]

\[ \langle \nu, \sigma, e.p \rangle \longrightarrow \langle \nu', \sigma', e'.p \rangle \]

\[ \langle \nu, \sigma, e_1 \rangle \longrightarrow \langle \nu', \sigma', e'_1 \rangle \]

\[ \langle \nu, \sigma, e_1.f = e_2 \rangle \longrightarrow \langle \nu', \sigma', e'_1.f = e'_2 \rangle \]

\[ \langle \nu, \sigma, e_2 \rangle \longrightarrow \langle \nu', \sigma', e'_2 \rangle \]

\[ \langle \nu, \sigma, \iota_1.f = e_2 \rangle \longrightarrow \langle \nu', \sigma', \iota_1.f = e'_2 \rangle \]

\[ \langle \nu, \sigma, e_0(\iota, e, \iota) \rangle \longrightarrow \langle \nu', \sigma', e_0(\iota, e', \iota) \rangle \]

\[ \langle \nu, \sigma, e_0(\tau) \rangle \longrightarrow \langle \nu', \sigma', e'_0(\tau) \rangle \]

\[ \langle \nu, \sigma, \tau = e \rangle \longrightarrow \langle \nu', \sigma', \tau = e' \rangle \]

\[ \langle \nu, \sigma, \llbracket T, e \rrbracket \rangle \longrightarrow \langle \nu', \sigma', \llbracket T, e' \rrbracket \rangle \]

Figure 3.9: Congruence rules for expressions in Fletch.
CHAPTER 3. FLETCH

3.3 Type System

In this section we formalize the Fletch type system, which models part of the type system defined in the Dart specification [21]. The formalization includes some of the additional typing rules Fletch artifacts that we will use in Chapter 4 to prove the soundness theorem.

3.3.1 Auxiliary Definitions

Figure 3.12 defines a few auxiliary functions: accessor is a convenient short-hand for property lookup, foverride defines requirements on re-declaring a field in a subclass, and moverride defines requirements on method overriding. The last two predicates use the bound function. It replaces all the type variables occurring in the type T to their upper bound as defined in the type environment Δ.

\[
\Delta \vdash \text{dynamic} \ OK \quad \Delta \vdash \bot \ OK \quad \Delta \vdash \text{void} \ OK \quad \Delta \vdash \text{Object} \ OK
\]

\[
\frac{\Delta \vdash G \ OK \quad \Delta \vdash T \ OK}{\Delta \vdash (G) \rightarrow T \ OK}
\]

\[
\frac{X \in \text{dom(Δ)}}{\Delta \vdash X \ OK}
\]

CT(c) = class c<X<N> extends N {···}

\[
\Delta \vdash G : [G/X]N \quad \Delta \vdash G \ OK
\]

\[
\Delta \vdash c<G> \ OK
\]

Figure 3.10: Well-formed types.

Finally, Figures 3.14 shows the top-level rules for typing of classes that causes all the other elements of type checking to be applied.

Figure 3.11 shows the definition of ftype and mtype, that respectively lookup the list of declared field and declared methods in a given class type.

The only nonstandard element of ftype is the treatment of the receiver type dynamic where all field names are considered to be defined and having the type dynamic. Similarly, the only nonstandard part of mtype is that a receiver of type dynamic is considered to have all methods, each of which also has the type dynamic.

Figure 3.10 defines what it means for a type T to be well-formed in a type environment Δ, written Δ \vdash T OK. Note that type well-formedness requires subtyping for type parameters rather than assignability: if we have a class definition class c<X<String> {···} then c<Object> is not a well-formed type, since X must be a subtype of String. Type well-formedness is used in the top-level rules for definition typing (Figure 3.14).
3.3. TYPE SYSTEM

3.3.2 Expression Typing

The typing judgment $\nu; \sigma; \Delta; \Gamma \vdash e : T$ indicates that the expression $e$ is well-typed with the type $T$ in the environments $\nu$, $\sigma$, $\Delta$ and $\Gamma$. Here, $\nu$ maps variable locations to heap locations, $\sigma$ maps heap locations to objects or closures, $\Delta$ maps type variables to their upper bounds, and $\Gamma$ maps variables to their declared types. The initial environments for an execution are $\nu_{\text{base}} = \emptyset$, $\sigma_{\text{base}} = [\iota_{\text{null}} \mapsto o_{\text{null}}]$, $\Delta_{\text{base}} = \emptyset$, and $\Gamma_{\text{base}} = \{\text{null} : \bot\}$.

The Fletch type systems differ from the Dart type system in a couple of ways. In particular, in Figure 3.13 there are several type rules concerned with runtime expressions, e.g., heap locations, that are absent in the Dart specification because it does not formalize the dynamic semantics. The [T-Function] rule contains the return type, which is absent in the Dart syntax; we gave reasons for having it in Section 3.1.

The rules [T-Var], [T-Read], [T-Write], and [T-Assign] are unsurprising apart from the assignability checks, which allow some types to be both subtypes and supertypes where typical type systems would require a subtype.

The rule [T-Call] is also unsurprising, apart from the fact that it allows for supertypes for the actual arguments. The [T-New] rule is very simple because mutability allows us to omit constructors. [T-Function] is also standard, noting that the list $Gx$ cannot contain any duplicate variable names. Finally, the rules [T-Runtime-Loc], [T-Runtime-Frame] and [T-Runtime-VAssign] are simple extrapolations.
\[
\begin{align*}
\text{CT}(c) &= \text{class } c\langle X < \ldots \rangle \text{ extends } N \ldots \{ F \ldots \} \\
\text{ftype}([G_1/X]N, f) &= G_2 \quad f \notin \text{name}(F) \\
\text{ftype}(c\langle \bar{G}_1 \rangle, f) &= G_2 \\
\text{ftype}(\text{dynamic}, f) &= \text{dynamic} \\
\text{CT}(c) &= \text{class } c\langle X < \ldots \rangle \ldots \{ \ldots G_2 f; \ldots \} \\
\text{ftype}(c\langle \bar{G}_1 \rangle, f) &= [G_1/X]G_2 \\
\text{CT}(c) &= \text{class } c\langle X < \ldots \rangle \text{ extends } N \ldots \{ \ldots M \} \\
\text{mtype}([G_1/X]N, m) &= G_2 \quad m \notin \text{name}(M) \\
\text{mtype}(c\langle \bar{G}_1 \rangle, m) &= G_2 \\
\text{mtype}(\text{dynamic}, m) &= \text{dynamic} \\
\text{CT}(c) &= \text{class } c\langle X < \ldots \rangle \ldots \{ \ldots T \quad m(G_2 x) \{ \ldots \} \} \\
\text{mtype}(c\langle \bar{G}_1 \rangle, m) &= [G_1/X][(G_2) \rightarrow T] \\
\end{align*}
\]

Figure 3.11: Lookup definitions.

\[
\begin{align*}
\text{[FIELD-ACCESSOR]} \quad &\text{ftype}(G_1, f) = G_2 \\
&\text{accessor}(G_1, f) = G_2 \\
\text{[METHOD-ACCESSOR]} \quad &\text{mtype}(G_1, m) = G_2 \\
&\text{accessor}(G_1, m) = G_2 \\
\text{[FIELD-OVERRIDING]} \quad &\text{ftype}(\text{bound}_\Delta(N), f) = G_2 \\
&\text{implies assignable}_\Delta(G_1, G_2) \\
&\text{foverride}_\Delta(f, N, G_1) \\
\text{[METHOD-OVERRIDING]} \quad &\text{mtype}(m, \text{bound}_\Delta(N)) = (G_2) \rightarrow T_2 \\
&\text{implies assignable}_\Delta(G_1 \rightarrow T_1, G_2 \rightarrow T_2) \\
&\text{moverride}_\Delta(m, N, (G_1) \rightarrow T_1) \\
\end{align*}
\]

Figure 3.12: Property accessor and field/method overriding definitions.
3.3. **TYPE SYSTEM**

\[ [T-\text{VAR}] \nu; \sigma; \Delta; \Gamma \vdash y : \Gamma(y) \]

\[ [T-\text{READ}] \frac{\nu; \sigma; \Delta; \Gamma \vdash e : T}{\nu; \sigma; \Delta; \Gamma \vdash e.p : G} \quad \text{accessor} (\text{bound}_\Delta(T), p) = G \]

\[ [T-\text{WRITE}] \frac{\nu; \sigma; \Delta; \Gamma \vdash e_1 : T_1}{\nu; \sigma; \Delta; \Gamma \vdash e_1.f = e_2 : T_2} \quad \text{assignable}_\Delta(T_2, G) \]

\[ [T-\text{ASSIGN}] \frac{\nu; \sigma; \Delta; \Gamma \vdash e : T}{\nu; \sigma; \Delta; \Gamma \vdash x = e : T} \quad \text{assignable}_\Delta(T, \Gamma(x)) \]

\[ [T-\text{DYNAMIC-CALL}] \frac{\nu; \sigma; \Delta; \Gamma \vdash e_0 : \text{dynamic}}{\nu; \sigma; \Delta; \Gamma \vdash e_0(\sigma) : \text{dynamic}} \]

\[ [T-\text{NEW}] \frac{\Delta \vdash N \text{ OK}}{\nu; \sigma; \Delta; \Gamma \vdash \text{new } N() : N} \]

\[ [T-\text{CALL}] \frac{\nu; \sigma; \Delta; \Gamma \vdash e_0 : (G) \rightarrow T}{\nu; \sigma; \Delta; \Gamma \vdash e_0(\sigma) : T} \quad \text{assignable}_\Delta(T, G) \]

\[ [T-\text{FUNCTION}] \frac{\Delta \vdash \text{G OK}}{\nu; \sigma; \Delta; \Gamma \vdash e : T' \rightarrow T} \quad \text{assignable}_\Delta(T', T) \]

\[ [T-\text{RUNTIME-LOC}] \nu; \sigma; \Delta; \Gamma \vdash \lceil i \rceil : \text{typeof}(i, \sigma) \]

\[ [T-\text{RUNTIME-FRAME}] \frac{\nu; \sigma; \Delta; \Gamma \vdash e : T' \quad T' \prec ; T}{\nu; \sigma; \Delta; \Gamma \vdash \lceil [T, e] \rceil : T} \]

\[ [T-\text{RUNTIME-VLOC}] \frac{\nu; \sigma; \Delta; \Gamma \vdash \lceil \tau \rceil : \text{typeof}(\tau, \nu)}{\nu; \sigma; \Delta; \Gamma \vdash \tau = e : T} \quad \text{assignable}_\Delta(T, \text{typeof}(\tau, \nu)) \]

Figure 3.13: Expression typing. Boxed parts in conclusions are Fletch artifacts that do not occur in the actual Dart syntax.
\[ \Delta = X <: N \quad \Delta \vdash G \text{ OK} \]
\[
\text{CT}(c) = \text{class } c<X < N> \text{ extends } N \{ \cdots \}
\]
\[
\text{foverride}_\Delta(f, N, G)
\]
\[ G \ f \text{ OK in } c \]
\[ \Delta = X <: N \quad \Delta \vdash T \text{ OK} \quad \Delta \vdash C \text{ OK} \]
\[
\emptyset; \nu_{\text{base}}; \Delta; \Gamma_{\text{base}}; x : G, \text{this} : c<X > \vdash e_0 : T_0
\]
\[
\text{CT}(c) = \text{class } c<X < N> \text{ extends } N \{ \cdots \}
\]
\[
\text{assignable}_\Delta(T_0, T) \quad \text{moverride}_\Delta(m, N, (G) \rightarrow T)
\]
\[ T \ m(G x)\{ \text{return } e_0; \} \text{ OK in } c \]
\[ \Delta = X <: N \quad \Delta \vdash N \text{ OK} \quad \Delta \vdash N \text{ OK} \]
\[
\text{nodup}(X) \quad \text{nodup}(f) \quad \text{nodup}(m) \quad F \text{ OK in } c \quad M \text{ OK in } c
\]
\[
\text{class } c<X < N> \text{ extends } N \{ F \ M \} \text{ OK}
\]

Figure 3.14: Typing of classes.
Chapter 4

Message Safety

In Chapter 3 we have introduced Fletch: a formal model of a subset of the Dart operational semantics and type system. The purpose of Fletch is to test the first part of the hypothesis stated in Section 1.1 that is, despite the Dart type system being unsound, it is possible to identify natural restrictions of the type system that ensure the absence of message-not-understood errors. This chapter shows the restrictions by introducing the notion of message-safety, along with a proof that message-safe programs cannot fail with message-not-understood errors at runtime. The proof is also available in Coq (see http://www.brics.dk/fletch/) besides preventing message-not-understood errors, message-safety is also a natural intermediate point between the Dart type checking and a traditionally sound type system. We will use the term message-safe type system to denote the type system ensuring message-safety, obtained by slightly restricting the standard Dart type system.

The proof of message-safety differs from traditional soundness proofs, since the message-safe type system is more permissive, allowing for subtype-violation errors. This raises unique challenges in the proof. A fact worth noting is that assignability, formalized in Section 3.2.3, is not transitive. The following program fragment is accepted by the Dart type checker, because both assignments satisfy the assignability requirement (integers are objects in Dart), but an int value is not assignable to a String variable, and hence a checked mode execution will fail:

```
Object obj = 1;
String s = obj; // fails at runtime in checked mode
```

The lack of transitivity makes assignability quite inconvenient to work with in a formal model. For example, it invalidates the typical line of reasoning in a type soundness proof: Assume that we consider a variable declaration with an associated initialization expression, $T_1 x = e$, and that we have a proof that $e$ has the type $T_2$, which is a subtype of $T_1$. Typing succeeds, because it is allowed to initialize $x$ with a value whose type is a subtype of the declared type
CHAPTER 4. MESSAGE SAFETY

Now assume that a step is taken in the execution of the program, changing $e$ to $e'$, and assume that we have a proof that $e'$ has the type $T'_2$, which is a subtype of $T_2$. At this point, the standard proof (of the type preservation part of soundness) proceeds to use the transitivity of subtyping to conclude that $T_1 \ x = e'$ is type correct. However, without transitivity, we cannot conclude that $T_1 \ x = e'$ is type correct.

Interestingly, we have succeeded in obtaining our message-safety soundness result using a less restrictive type system where the assignability requirements in the standard Dart type rules have been omitted. In the same vein, the Dart language specification includes the notion of a type being more specific than another type, which amounts to a slightly modified version of subtyping. This relation is transitive (Section 3.2.3), and we use it directly in our treatment of soundness.

This chapter presents the following contributions:

- We define the notion of message-safe programs, which can be viewed as a natural level between dynamic and static typing. The significance and relevance of message-safe programs are motivated by their potential role in practical software development. To support gradual evolution from dynamically typed to message-safe programs, we outline a generalization of message safety from complete programs to program fragments.

- We show a soundness theorem stating that message-safe programs do not cause ‘message-not-understood’ errors in checked mode execution.

- As part of the proving process, we discovered a property of the type rule for function subtyping that was not intended by the Dart language designers and that affects message safety. We argue empirically that this can easily be fixed. We additionally report on initial experimental results that support the use of message safety in software development.

This chapter is extracted from Ernst et al. [25] and extended with additional technical content. The last paragraph of Section 4.1 shows how to restrict the Fletch type system described in Section 3.3 to obtain message-safety (see Figure 4.1), whereas the paper integrates these modifications directly in the Fletch formal model. Figure 4.3 is extended with congruence rules, and we add Figure 4.5 to show acceptable error configurations for full type-safety. Section 4.7 is extended with a more detailed proof of progress and preservation for message-safety and the soundness theorem for full type-safety. We introduce Section 4.8 that contains some of the lemmas used in the progress and preservation proofs in Coq. Section 4.9 extends the discussion in Section 4.4 with more details and a lemma stating that message-safety can be applied to program fragments.
4.1 Message-Safe Programs

Under which conditions can a Dart programmer be certain that his program will not raise any ‘message-not-understood’ error during checked mode execution? This section presents the core concept that can lead to such a guarantee. Surprisingly, this can be achieved without taking the full step to traditional type soundness.

We define a message-safe Dart program as one that satisfies the following requirements:

1. The annotation `dynamic` does not occur, neither explicitly nor implicitly. Specifically, all fields, method signatures, and variables have type annotations, and type parameters cannot be omitted.
2. Type checking the program produces no static type warnings using the standard Dart type checker with the following modifications:
   a) Overriding methods must have covariant return types. That is, if a superclass $C_1$ contains a method $m$ with return type $T_1$ and a subclass $C_2$ of $C_1$ contains a method $m$ with return type $T_2$ then $T_2$ must be a subtype of $T_1$. Similarly, the types of overriding fields must be covariant (overriding for fields makes sense because all accesses use getters and setters).
   b) Subtyping among function types requires covariant return types. That is, the type of a function with return type $T_1$ is a subtype of one with return type $T_2$ only if $T_1$ is a subtype of $T_2$.

One of our key contributions is to demonstrate that these requirements suffice, as shown in the following sections for a core language. Requirement 1 is not surprising, as `dynamic` effectively disables static type checking. Informally, requirement 2(a) is motivated by the fact that a method override with an unrelated return type could easily cause a ‘message-not-understood’ error for a property looked up on the returned value, and similarly for 2(b). Clearly, it is not hard to implement a checker that decides for any given Dart program whether it is message safe.

Examples We show three small programs that demonstrate the need for the message-safety requirements. Consider the following class definitions:

```dart
class A {
    A f;
    A m(Object x) {
        return new A();
    }
}
```

Our implementation [http://www.brics.dk/fletch/](http://www.brics.dk/fletch/) required modifying less than 200 LOC in Google’s `dartanalyzer` tool.
class B extends A {
  Object f = new Object();
  Object m(int x) {
    return new Object();
  }
}

All of the following programs fail with a ‘message-not-understood’ runtime error in checked mode execution, for different reasons. The standard Dart type checker emits no warnings, whereas our message-safety type checker catches the error in each case.

- A x = new B();
  x.m(42).f;

  The overriding method m violates requirement 2(a). A runtime error occurs because x.m(42) returns a value of type Object, which does not have an f property. It is not surprising that this causes a ‘message-not-understood’ error that we must prevent. Statically we expect x.m to have type Object → A, but dynamically we encounter a function of type int → Object, which is not a subtype of the former, and in particular it violates the standard requirement that the return type of a function type is covariant. Note, however, that we do not have to require contravariance for argument types, because the associated failure will be a ‘subtype violation’ error, which is allowed.

- A x = new B();
  x.f.m(117);

  The overriding field f violates requirement 2(a). A runtime error occurs because x.f yields a value of type Object, which does not have an m method. Noting that x.f semantically is a getter, i.e., a function that gets the value of the field named f, it is easy to see that the situation is the same as for the previous example.

- typedef A MyFunType(Object x);
  MyFunType g = (String x) => new Object();
  g("foo").f;

  The type of the anonymous function stored in g is not assignable to g, as requirement 2(b) is violated. A runtime error occurs because the function returns a value of type Object, which does not have an f property. Once again, the underlying issue is that we must enforce return type covariance for functions, in this case applied to a first class function value.
4.2. **FULL TYPE SAFETY**

Although our primary focus is on message-safety *full type safety*, where neither ‘message-not-understood’ nor ‘subtype violation’ errors are possible, can be ensured statically by the following requirements in addition (1) and (2) for message-safety:

3. Every assignability check is replaced by a subtype check (or, equivalently, assignability is redefined to coincide with subtyping).

4. Generic class subtyping requires invariance rather than covariance.

5. a) Method overriding requires contravariant argument types, and field overriding must be invariant.

b) Function subtyping requires contravariant argument types.

We formalize these modifications in Figure 4.2. Requirement 3 is obtained by restricting the assignability relation so that it coincides with the subtyping
CHAPTER 4. MESSAGE SAFETY

Figure 4.2: Modifications that ensure full type safety. The boxed parts show the changes compared to the message-safe type system (cf. Figure 4.1). In addition, the second rule of Figure 3.7 is removed.

relation. The first rule is required for generic invariance, as specified by requirement 4. The second rule is required for method overriding contravariance for parameters, according to requirement 5(a) and the last rule is required for function subtyping contravariance for parameters, as specified by requirement 5(b).

These additional requirements, especially those involving invariance, may obviously cause many useful Dart programs to be rejected. Even though it is possible to replace invariance by less restrictive (but more complex) variants, this observation supports our argument that message safety is a flexible and simple alternative to full type safety.

Examples To motivate requirements 3–5, consider the following class definitions:

class A {
    Object m(Object x) {
        return new A();
    }
}
class B<X extends A> extends A {
    X f;
}
class C extends A {
    Object m(int x) {
        return new Object();
    }
}

The following four programs fail with a ‘subtype violation’ runtime error in checked mode execution, for different reasons. The standard Dart type checker and the message-safety type checker emit no warnings since they do not pre-
4.2. **FULL TYPE SAFETY**  

vent ‘subtype violation’ errors, but with full type safety each error is caught statically.

- **C x = new A();**  
The assignment violates requirement 3. A runtime error occurs because the right-hand side of the assignment has type A that is not a subtype of the declared type C. This is unsurprising: it is a standard requirement for sound typing that assignments admit subtypes, but not supertypes.

- **B<A> x = new B<B<A>>()**;  
  *x.f = new C();*  
The first assignment violates requirement 4. The program is well-typed by the message-safe type system since x.f has static type A and the right-hand side of the field assignment has type C that is a subtype of A. At runtime, x will have type B<B<A>>, but the right-hand side of the field assignment has type C that is not a subtype of B<A>. This situation where a generic class is considered to be covariant in a type argument that occurs in a contravariant position (the argument type of the setter for f) is also a well-known source of soundness violations, documented by Cook in 1989 [15].

- **A x = new C();**  
  *x.m("");*  
The overriding method m in class C violates requirement 5(a). A runtime error occurs because m at runtime belongs to the class C, so the formal parameter has type int but the argument has type String. The underlying issue is again function type subtyping. In Section 4.1 it was sufficient to enforce covariance for the return types of functions, but in order to maintain full type safety we must also enforce contravariance for argument types, which is violated for the method m in class C.

- **typedef int MyFunType(Object x);**  
  MyFunType x = (int x) => 0;  
  *x("");*  
The x assignment violates requirement 5(b). At runtime the formal parameter of x is int and the argument has type String that is not a subtype of int. As explicitly stated, the underlying issue is again argument type contravariance, this time concerned with a first class function value.

In summary, even though message-safety allows for a substantially more flexible approach to typing than traditional, sound type rules, the steps needed to go from message-safety to full type safety are simple and unsurprising.
4.3 Message Safety and Nominal Identity

A useful intuition about message-safe programs is that they make programmers decide on a specific choice of the meaning of every property (method or field) that is used in the program. More concretely, for every property lookup (e.g., \(x.f\)) in such a program, the declared type of the receiver object \(x\) ensures that the property \(f\) is defined. Since Dart types are nominal, we say that message-safe programs enforce the commitment to a specific nominal identity for each property lookup operation. Such a nominal identity determines the location in the source code where a definition of the property is given. The documentation about how to use or redefine this property (type annotations, informal comments, etc.) should reside there, or in a statically known superclass. Late binding may cause the invocation of a method, e.g., \(x.m(y)\), to execute a method implementation in a proper subtype of the one that contains the statically known declaration. However, both the programmer writing the invocation and the programmer redefining the property will know statically where to find the appropriate documentation of the semantics. This helps maintaining consistency.

Of course, that documentation may be absent, misleading, or just informal, but compared to the non-message-safe situation where a given property being looked up could resolve to many different declarations in a large software system (essentially any declaration with the right name), we believe that the static commitment to a nominal identity is a powerful tool for clarification of the intended use and semantics, thus promoting well-understood and correct software.

4.4 Message Safety for Program Fragments

The notion of message safety also makes sense for program fragments, not only for complete programs. In fact, such a generalization is almost trivial in most cases. Consider a property access expression of the form \(x.f\) or \(x.m(...)\) where \(x\) is a local variable or a formal argument to a method; in this case a local check on the declared type of the receiver \(x\) suffices to ensure that the property access will never cause a message-not-understood error in checked mode at runtime. For the field access we just check that the receiver type declares a field (or getter) named \(f\), and for the method call we check that the method exists, with the given arity. If \(x\) is a field in \texttt{this} object we check that its declared type includes the requested property. Similarly, for an access expression applied to a returned value, e.g., \(x.m(...).f\) or \(x.m(...).n(...)\), we check that the return type of \(m\) declares that property. For every class we encounter, the covariance check in 2(a) is applied modularly (i.e., to that class alone), which ensures that every looked up property based on a field or a returned value has the statically declared type in every superclass. Finally,
first-class closures in Fletch support a direct inspection of their dynamic type (as opposed to the approaches using blame assignment where checks must be delayed because the type of a higher-order value cannot be inspected dynamically), which makes it possible to treat them just like objects when considering message safety. Clearly, this is just as modular as a standard type check, e.g., in the Java programming language.

One minor complication arises due to the fact that in Dart (unlike other languages with gradual typing) the type `dynamic` may appear in the runtime type of entities, which may cause violations of the type annotations in the program fragment under consideration. Modular message safety checking therefore includes the constraint that type parameters in the runtime type of generic instances and the return type of function closures cannot be `dynamic`.

From a software engineering point of view, a developer who is working with a large program can use a modular message-safety check on one property lookup at a time, for example focusing on a critical program fragment and thereby obtaining the benefits of message safety for that fragment, without requiring the conditions from Section 4.1 to be satisfied for the entire program. This aligns well with the concept of gradual typing that is a cornerstone of the Dart design. If a lookup pertains to a receiver whose type is declared outside the program fragment of interest (consider, for example, an expression `x.y.z` where `y` is declared in the class of `x` and is the receiver for the lookup of `z`), it may be useful to make remote adjustments (changing the type of `y`), or it may be better to introduce a local variable with a suitable type, holding a reference to that receiver (`x.y`). The choice will depend on which of the two adjustments fits better into the given software development context.

### 4.5 A Two-Step Approach Toward Type Safety

The Dart language specification [21, page 124] suggests that a sound type checker can be implemented and used, for example, as a stand-alone tool. This is a rather well-understood undertaking, and we will only briefly discuss full type safety in this dissertation. Instead, we observe that message-safe programs constitute an intermediate form between dynamic typing and full static type safety, which enables a structured evolution toward type safe programs. The set of message-safe programs separates such a transformation into a predominantly local step that considers the usage of object properties at property lookup operations where message-not-understood errors may occur, and a global step that considers subtype constraints at assignments and other dataflow operations where subtype-violation errors may occur.

As an example, consider the following untyped program:

```dart
class Account {
  var balance = 0;
  withdraw(amount) {
```
The first step toward a type safe program is to make the program message-safe, the main part of which is adding type annotations. For the programmer, a useful way to think about this transformation is that every lookup operation (as in `x.f`) enforces a sufficiently informative type (of `x`) to ensure that the corresponding lookup (of `f`) will succeed. In the example above, the use of `account.withdraw(amt)` thus forces `account` to have a sufficiently informative type to ensure that it has a `withdraw` method with one argument. Here is a corresponding message-safe program (changes highlighted):

```java
class Account {
    int balance = 0;
    void withdraw(int amount) {
        balance -= amount; return amount;
    }
}
Object pay(Account account, Object amt) {
    return account.withdraw(amt) == amt;
}
Object make() { return new Account(); }
void main() { var acc=make(); pay(acc,10); }
```

Note that `acc` can have type `Object` because no properties are used via this variable, in contrast to `account`. It is not required for message-safe programs that all types are as general as possible (e.g., `pay` could return type `bool`), but it is likely to be a practical and maintainable style to commit only to the types required for property lookups.

The second step in the transformation to a type safe program is to propagate types according to the dataflow that takes place in assignments and argument passing operations. Whenever a value is passed from some expression into a variable, the expression must have a type that is a subtype of that variable, and similarly for function arguments and return values. This is achieved by replacing declared types by subtypes in a process similar to constraint propagation, until the program satisfies the standard subtype constraint everywhere. A corresponding statically safe program is as follows:

```java
class Account {
    int balance = 0;
    int withdraw(int amount) {
```
4.6 MESSAGE SAFETY FOR OTHER LANGUAGES

    balance -= amount; return amount;
    }
    }
    Object pay(Account account, int amt) {
        return account.withdraw(amt) == amt;
    }
    Account make() { return new Account(); }
    void main() { Account acc = make(); pay(acc, 10); }

    In general, both steps may require restructuring of the program code itself, not just insertion or adjustment of type annotations: e.g., the code may be inherently type unsafe (such that some executions will produce a message-not-understood error at runtime), or it may be safe only according to a structural typing discipline (such that some property accesses will succeed with different unrelated nominal types at different times). But for programs that have a safe nominal typing, it seems plausible that the constraint solving step could be performed automatically. However, exploring algorithms for that is future work.

    Note that the type annotations in the first step can be chosen entirely based on the local use of features of each object, without any global considerations. This fits nicely with the expected importance of IDE support for code completion. The message-safe program may raise subtype-violation type errors at runtime, but it will not raise message-not-understood errors. Hence, in message-safe programs, the type annotations justify the actual property lookups, while implicit downcasts are still allowed, which enables a more flexible flow of data compared to traditional sound typing.

4.6 Message Safety for Other Languages

The essence of message safety is strict treatment of lookups and flexible treatment of dataflow. It would be straightforward with, for example, the Java and C# programming languages to allow for part of the flexibility that Dart offers by modifying compilers to insert downcasts, rather than rejecting the program as untypable when a downcast is needed but not specified.

    Message safety then corresponds to the standard type checks applied to lookups, and the constraints on programs would rule out the dynamic type in C# and anything in the Java language that relies on the invokédynamic byte code. Thus, at the technical level there are no deep difficulties in providing the same combination of nominal safety and dataflow flexibility that we are proposing for Dart.
Base cases

\[ \nu; \sigma \vdash \eta_{\text{null}}.p \quad \text{ERROR} \quad \nu; \sigma \vdash \eta_{\text{null}}.f = \iota \quad \text{ERROR} \quad \nu; \sigma \vdash \eta_{\text{null}}(\iota) \quad \text{ERROR} \]

\[ \vdash \text{typeof}(\iota, \sigma) \not< \text{typeof}(\tau, \nu) \]

\[ \nu; \sigma \vdash \tau = \iota \quad \text{ERROR} \]

\[ \sigma(\iota_1) = (\_, \phi, \_); \phi(f) = (G, \_) \quad \text{typeof}(\iota_2, \sigma) = T \quad \vdash T \not< G \]

\[ \nu; \sigma \vdash \iota_1.f = \iota_2 \quad \text{ERROR} \]

\[ \text{typeof}(\iota, \sigma) = (G) \to T \quad \text{typeof}(\iota_i, \sigma) = T_i' \quad \vdash T_i' \not< G_i \]

\[ \nu; \sigma \vdash \iota(T) \quad \text{ERROR} \]

\[ \vdash \text{typeof}(\iota, \sigma) \not< T \]

\[ \nu; \sigma \vdash [T, \iota] \quad \text{ERROR} \]

Inductive cases

\[ \nu; \sigma \vdash e_1 \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e_1.p \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e_1 \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e_1.p = e_2 \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e_2 \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e_1.p = e_2 \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e(\overline{e}) \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e_i \quad \text{ERROR} \quad \text{for some } e_i \in \overline{e} \]

\[ \nu; \sigma \vdash e(\overline{e}) \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e \quad \text{ERROR} \]

\[ \nu; \sigma \vdash [T, e] \quad \text{ERROR} \]

\[ \nu; \sigma \vdash e \quad \text{ERROR} \]

\[ \nu; \sigma \vdash \tau = e \quad \text{ERROR} \]

Figure 4.3: Acceptable runtime errors in message-safe programs.
4.7  Soundness of Message Safety

Soundness is traditionally associated with Milner’s phrase *well-typed programs cannot go wrong* [58], but message safety allows for subtype-violation errors (and null pointer errors), whereas message-not-understood must be ruled out. As usual, the main steps on the way to a type soundness proof are progress and preservation.

As discussed in Section 4.1 we focus on checked mode execution. The following lemmas use a relaxed form of the message-safe type system, which is obtained with the following two logical steps. First, the typing rules defined in Figure 3.13 are modified according to the rules in Figure 4.1 to obtain the message-safe type system. Then, assignability in the premises of the typing rules and subtyping in the premise of rule \[ \text{T-Runtime-Frame} \] are omitted.

We thereby avoid the problems with non-transitivity of assignability, and, perhaps surprisingly, message-safety soundness still holds in this weakened type system. As a simple corollary, soundness also holds for the message-safe type system where the assignability premises are present.

The notation $\sigma \succeq$ means that every location in the heap $\sigma$ is well-formed. (Figure 4.4 shows what it means for an object to be well-formed, and a similar criterion applies for closures.)

The notation $\nu; \sigma \vdash \nu; \sigma$ OK means that each variable location in $\nu$ is mapped to a pair $(G, \iota)$ such that $\text{typeof}(\iota, \sigma)$ is a subtype of $G$. The notation $\nu; \sigma \vdash e$ ERROR means that the configuration $\langle \nu, \sigma, e \rangle$ is a subtype-violation or null pointer error (defined formally in Figure 4.3).

In the lemmas and the soundness theorem, it suffices to consider only the base environments, $\Delta = \emptyset$ because we are only interested in soundness for complete programs $(\text{CT}, e)$, and $\Gamma = \Gamma_{\text{base}}$ because free variables (including $\text{this}$) are always substituted with variable locations during execution (due to rules $[\text{E-Call}]$ and $[\text{E-New}]$).

Informally, the progress lemma says that for any well-typed expression in a well-formed environment, a) the expression is a value, b) evaluation can proceed, or c) evaluation is stuck but not due to message-not-understood.

**Lemma 4.7.1 (Progress)** *If $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T$ and $\sigma \succeq$ and $\sigma \vdash \nu \succeq$ and $e, \nu,$ and $\sigma$ do not contain dynamic then*

a) $e$ is a value (i.e., a heap location) or

b) $\langle \nu, \sigma, e \rangle \rightarrow \langle \nu', \sigma', e' \rangle$ for some $\nu', \sigma', e'$ or

c) $\nu; \sigma \vdash e$ ERROR.
\[
\begin{align*}
\text{fields}(c\langle G \rangle) &= H f \\
\text{methods}(c\langle G \rangle) &= T \ m \ (G' x) \{ \cdots \}
\end{align*}
\]

\[
\emptyset \vdash \text{typeof}(\iota f, \sigma) <: H
\]

\[
\text{typeof}(\iota m, \sigma) = (G') \rightarrow T
\]

\[
\nu; \sigma \vdash (c\langle G \rangle, f : G \mapsto \iota f, m \mapsto \iota m) \ \text{OK}
\]

\[
\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T \xrightarrow{\nu} e (G) \rightarrow T
\]

\[
\nu; \sigma; o_{\text{null}} \ \text{OK}
\]

**Figure 4.4:** Well-formed objects.

**Proof** By induction in the structure of the typing derivation \(\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T\). We here show a sketch of eleven of the cases and refer to the proof in Coq for further details.

- **Rule [T-Read]** where \(\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e.p : T\): If \(e\) is a value \(\iota\), since \(\iota.p\) is well-typed and all the environments are well formed, \(\sigma(\iota)\) is an object that has the property \(p\), so we can conclude that \(\langle \nu, \sigma, \iota.p \rangle \rightarrow \langle \nu, \sigma, \iota' \rangle\) for some \(\iota'\), corresponding to condition b. Otherwise, we can apply the induction hypothesis to get either \(\langle \nu, \sigma, e \rangle \rightarrow \langle \nu', \sigma', e' \rangle\) or \(e\) is an acceptable error configuration, which by congruence gives conditions b and c, respectively. Notice in particular that the evaluation cannot result in a message-not-understood error.

- **Rule [T-Var]** where \(\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash y : T\): Since \(y\) type checks in the environment \(\Gamma_{\text{base}}\), \(y\) must be \(\text{null}\), so rule [E-Null] applies, hence condition b is satisfied. (As discussed above, variables \(x\) and \(\text{this}\) have been substituted earlier by rules [E-Call] and [E-New].)

- **Rule [T-Write]** where \(\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e_1.f = e_2 : T\). There are 3 cases:
  1. By induction hypothesis, \(\langle \nu, \sigma, e_1 \rangle \rightarrow \langle \nu', \sigma', e'_1 \rangle\) or \(e_1\) is an error. Therefore, we can either reduce \(e_1.f = e_2\) by rule [E-Cong-Field-Write-Left] or prove that \(e_1.f = e_2\) is an error configuration.
  2. By induction hypothesis, \(\langle \nu, \sigma, e_2 \rangle \rightarrow \langle \nu', \sigma', e'_2 \rangle\) or \(e_2\) is an error. Therefore we can reduce \(e_1.f = e_2\) or the expression is an error configuration.

---

2The proof [http://www.brics.dk/fletch/](http://www.brics.dk/fletch/) contains some unproven (using ‘Admitted’ or ‘admit’) but plausible lemmas. Technically, soundness has been reduced to those lemmas. Many of those state that extending or updating a well-formed heap or variable environment preserves well-formedness.
3. $e_1$ and $e_2$ are values, respectively $\iota_1$ and $\iota_2$, by Lemma 4.8.2, $\iota_1.d = \iota_2$ can be evaluated or it is an error configuration.

- Rule [T-Assign] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash x = e : T$. This case can never happen, since in order for this expression to be well-typed, it would require $x$ is in the domain of $\Gamma_{base}$ which is not true since the domain of $\Gamma_{base}$ only contains `null`.

- Rule [T-New] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash \text{new } N() : N$. This case holds by Lemma 4.8.4.

- Rule [T-Call] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash e_0(e) : (T) \rightarrow T$. We can apply the inductive steps similarly to the points above. The base case, that is, $e = \iota$ and $\bar{e} = (\bar{\iota})$, is proven by Lemma 4.8.3.

- Rule [T-Function] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash T(Gx) = e : (G) \rightarrow T$. This case is proven by Lemma 4.8.5.

- Rule [T-Runtime-Loc] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash \iota : \text{typeof}(\iota, \sigma)$. This case is trivial since $\iota$ is a value in Fletch.

- Rule [T-Runtime-Frame] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash J e_0, T K : T$. Similarly to the cases above, we can prove this case by combining the induction hypothesis with Lemma 4.8.7.

- Rule [T-Runtime-VLoc] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash \tau : \text{typeof}(\tau, \nu)$. Point 1 is trivial. Point 2 always hold, since by the environment well formendess we can apply the rule [E-Var-Read], proving $\langle \nu, \sigma, \tau \rangle \rightarrow \langle \nu, \sigma, \iota \rangle$, where $\Delta \vdash \text{typeof}(\iota, \sigma) < \text{typeof}(\tau, \nu)$ by environment well formendess.

- Rule [T-Runtime-VAssign] where $\nu; \sigma; \emptyset; \Gamma_{base} \vdash \tau = e : T$. Similarly to the cases above, this case can be proven by combining the induction hypothesis with Lemma 4.8.6.

Cases [T-Assign], [T-New], [T-Call], [T-Function], [T-Runtime-Loc], [T-Runtime-Frame], [T-Runtime-VLoc], and [T-Runtime-VAssign] are similar. The proof relies on additional lemmas for the base cases of this proof by induction, the terms can be evaluated or they are an error configuration. For example, one of these lemmas states that, if $\iota.p$ is well-typed, and all the environments are well formed, then $\iota.p$ can be evaluated. Section 4.8 describes most of these lemmas in more details.

The preservation lemma says that performing an execution step for a well-typed expression of type $T$ in a well-formed environment will preserve well-formedness and either lead to an expression whose type is a subtype of $T$ or to an acceptable error.
Lemma 4.7.2 (Preservation) If \( \nu; \sigma; \emptyset; \Gamma_{base} \vdash e : T \) and \( \sigma \) \( OK \) and \( \nu \) \( OK \) and \( e, \nu, \) and \( \sigma \) do not contain dynamic and \( \sigma_{base} \subseteq \sigma \) and \( \langle \nu, \sigma, e \rangle \rightarrow \langle \nu', \sigma', e' \rangle \) then both of the following hold:

1) \( \sigma' \ OK \) and \( \sigma' \vdash \nu' \) \( OK \) and \( e', \nu' \), and \( \sigma' \) do not contain dynamic and \( \sigma_{base} \subseteq \sigma' \) and either

2a) \( \nu'; \sigma'; \emptyset; \Gamma_{base} \vdash e' : T' \) where \( \emptyset \vdash T' <: T \) or

2b) \( \nu'; \sigma' \vdash e' \) ERROR.

Proof By induction in the execution derivation \( \langle \nu, \sigma, e \rangle \rightarrow \langle \nu', \sigma', e' \rangle \). We briefly show three cases; again, see the proof in Coq for further details.

- Rule [E-Var-Read] where \( \langle \nu, \sigma, \tau \rangle \rightarrow \langle \nu, \sigma, \iota \rangle \): Condition 1 trivially holds, since the environments do not change during the evaluation of \( \tau \). Since the environments are well formed we have \( \nu(\tau) = (G, \iota) \), so the type \( T \) of \( \tau \) is \( typeof(\tau, \nu) = G \). The type \( T' \) of \( \iota \) is \( typeof(\iota, \sigma) \), and \( \sigma \vdash \nu \) \( OK \) implies that \( typeof(\iota, \sigma) \) is a subtype of \( G \), so condition 2a holds.

- Rule [E-Var-Write] where \( \langle \nu, \sigma, \tau = \iota \rangle \rightarrow \langle \nu', \sigma, \iota \rangle \): The rule can only be applied if \( typeof(\iota, \sigma) \) is a subtype of \( typeof(\tau, \nu) \), so the update from \( \nu \) into \( \nu' = \nu[\tau \mapsto \iota] \) preserves environment well-formedness, so condition 1 is satisfied. Condition 2a holds because \( \iota = \iota \) is well-typed with type \( T \), rule [T-Assgn] gives that \( \iota \) has the same type \( T \), and subtyping is reflexive.

- Rule [E-Field-Read] where \( \langle \nu, \sigma \rangle \rightarrow \langle \nu', \sigma', e' \rangle \Rightarrow \langle \nu, \sigma, e.p \rangle \rightarrow \langle \nu', \sigma', e'.p \rangle \): Condition 1 follows directly from the premise of the rule and the induction hypothesis. Rule [T-Read] gives that \( e \) has some type \( T_1 \) where \( accessor(\text{bound}_\Delta(T_1), p) = T \). If \( e' \) is an acceptable error in the environments \( \nu' \) and \( \sigma' \) then by congruence so is \( e'.p \), corresponding to condition 2b. Otherwise, \( e' \) type checks with some type \( T_2 \) that is a subtype of \( T_1 \). If \( e' \) is \( t_{null} \) then \( e'.p \) is not well-typed but it is an acceptable error, corresponding to condition 2b (notice that this case shows why condition 2b is relevant in the lemma, unlike traditional preservation lemmas). Otherwise, due to the definitions of override and overridex in the message-safety type system (Figure 3.12) we have \( accessor(\text{bound}_\Delta(T_2), p) = T' \) where \( T' \) is a subtype of \( T \). Rule [T-Read] \( e'.p \) then has type \( T' \), meaning that condition 2a holds.

- Rule [E-Field-Write] where \( \langle \nu, \sigma, \iota_1, f = \iota_2 \rangle \rightarrow \langle \nu', \sigma', \iota_2 \rangle \). The only non trivial thing to prove for point 1, is that \( \sigma \vdash \nu' \) \( OK \). By rule [E-Field-Write], \( \nu' = \nu[\iota_1 \mapsto (c<G>, \phi[f \mapsto (G, \iota_2)], \mu)] \). By combining our hypothesis with Lemma 4.8.10 we get that \( (c<G>, \phi[f \mapsto (G, \iota_2)], \mu) \)
4.7. SOUNDNESS OF MESSAGE SAFETY

is well formed and this conclusion can be used together with our hypothesis, and Lemma 4.8.9 to conclude that $\nu' \vdash \sigma 0K$. Point 2a always hold since $\iota_2$ cannot be an error. From the premises of the rule [E-Field-Write] and since $\sigma$ is well formed, and $\nu; \sigma; 0; \Gamma_{base} \vdash \iota_1.p = \iota_2 : \text{typeof}(\iota_2, \sigma)$, we can conclude that $\nu; \sigma; 0; \Gamma_{base} \vdash \iota_2 : \text{typeof}(\iota_2, \sigma)$ and $0 \vdash \text{typeof}(\iota_2, \sigma) <: \text{typeof}(\iota_2, \sigma)$.

- Rule [E-Null] where $\langle \nu, \sigma, \text{null} \rangle \rightarrow \langle \nu, \sigma, \iota_{\text{null}} \rangle$. This case is trivial. Similarly to the previous case, point 1 holds. Point 2a holds because $\sigma_{\text{base}} \subseteq \sigma$ and therefore $\iota_{\text{null}} \in \sigma$, so that we can apply rule [T-Runtime-Loc]. Thus, we have that $\nu; \sigma; 0; \Gamma_{base} \vdash \text{null} : \bot$ and $\nu; \sigma; 0; \Gamma_{base} \vdash \iota_{\text{null}} : \bot$ and clearly $0 \vdash \bot <: \bot$.

- Rule [E-New] where $\langle \nu, \sigma, \text{new}<G> \rangle \rightarrow \langle \nu', \sigma', \iota \rangle$. This case will not be elaborated on detail since it contains many irrelevant details due to the complex definition of the rule [E-New]. Proving that point 2a always holds is trivial, but the non trivial part is proving point 1, with particular respect to $\sigma' 0K$ and $\nu' \vdash \sigma' 0K$. The operational semantics specifies a set of steps to create $\sigma'$ and $\nu'$. Each step generate a new environment and the Coq proof contains all the lemmas showing that at each step, the new environments are well formed.

- Rule [E-Func] where $\langle \nu, \sigma, (Gx) = e \rangle \rightarrow \langle \nu, \sigma', \iota \rangle$. Point 1 is straightforward, except proving that $\sigma' 0K$ and $\nu' \vdash \sigma' 0K$. It can be proven by applying the premises of rules [E-Func] and Lemma 4.8.9. Point 2a always hold since the type of $\iota$ is the same as the the type of $(Gx) = e$.

- Rule [E-Call] where $\langle \nu, \sigma, \iota_0(\tau) \rangle \rightarrow \langle \nu', \sigma, [T, \tau/e] \rangle$. Point 1 can be proven by combining the premises of the rule [E-Call] with Lemma 4.8.8. Point 2a always hold because of Lemma 4.8.11.

- Rule [E-Return] where $\langle \nu, \sigma, [T, \ell] \rangle \rightarrow \langle \nu, \sigma, \iota \rangle$. This case is straightforward.

- Rule [E-Cong-Field-Write-Left] where $\langle \nu, \sigma, e_1.p = e_2 \rangle \rightarrow \langle \nu', \sigma', e'_1.p = e_2 \rangle$. Points 1 and 2b follow from the induction hypothesis. Point 2a is more simple than in standard preservation proofs. Indeed, in order to prove that the evaluated expression $e'_1,p = e_2$ is well-typed and it preserves the type, we start as usual by deriving the typing rule for $e_1.p = e_2$ in our type system, using the rule [T-Write].

\[
\begin{array}{c}
\nu; \sigma; 0; \Gamma_{base} \vdash e_1 : T_1 \\
\nu; \sigma; 0; \Gamma_{base} \vdash e_2 : T_2 \\
\end{array}
\]

\[
\frac{\nu; \sigma; 0; \Gamma_{base} \vdash \text{accessor}(\text{bound}_0(T_1), f) = G}{\nu; \sigma; 0; \Gamma_{base} \vdash e_1.f = e_2 : T_2}
\]
By induction hypothesis we get \( \nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_1' : T_1', \emptyset \vdash T_1 < : T_1' \), and by Lemma 4.8.12 \( \nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_2 : T_2', \emptyset \vdash T_2 < : T_2' \). It is immediate to prove that

\[
\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_2 : T_2', \emptyset \vdash T_2 < : T_2'\]

and

\[
\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_2 : T_2'.
\]

It is immediate to prove that

\[
\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_2 : T_2'.
\]

Thus, we can conclude that \( \nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_1'.p = e_2 : T_2' \) that coincides point 2a. The derivation is the following.

\[
\begin{array}{c}
\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_1' : T_1'
\end{array}
\]

\[
\begin{array}{c}
\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_2 : T_2'
\end{array}
\]

\[
\begin{array}{c}
\text{accessor}(\text{bound}(T_1'), f) = G'
\end{array}
\]

\[
\begin{array}{c}
\text{assignable}(T_2', G')
\end{array}
\]

\[
\begin{array}{c}
\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e_1.f = e_2 : T_2'
\end{array}
\]

This typing derivation is correct because the “relaxed” type system used in this proof, does not include the boxed assignability rule in the premise. Since we do not need such premise, as we know that the evaluated term might eventually stop in a subtype violation error, we can rule it out. If we had included such premise, we would have had to prove that

- if \( \text{assignable}(T_2, G) \), and
- \( \text{assignable}(G, G') \) (we can prove this point since we know that \( G' \) is a subtype of \( G \)), and
- \( \text{assignable}(G', T_2) \), then
- \( \text{assignable}(T_2, T_2') \)

\[\text{Rules [E-Cong-Property-Read], [E-Cong-Property-Write-Right], [E-Cong-Call-Args], [E-Cong-Call-Function], [E-Cong-Var-Write], [E-Cong-Method-Read], and [E-Cong-Return] are similar to the previous cases.}\]

The proof relies on several minor lemmas, such as, reflexivity of subtyping (which is not defined as a rule for subtyping, but it is derivable), weakening, and lemmas stating that the environment updates during program execution preserve well-formedness and type annotations (see Section 4.8). □

Subexpressions may change type arbitrarily during evaluation (because both upcasts and downcasts are allowed), so the preservation lemma generally does not hold if the assignability premises from Figure 3.13 were included. As an example, consider the execution step \( \langle \nu, \sigma, \tau_B = [A, \iota_C].f \rangle \rightarrow \langle \nu, \sigma, \tau_B = \iota_C.f \rangle \) (applying a congruence rule and [E-Return]) in environments where the type of \( \tau_B \) is \( B \), the type of \( \iota_C \) is \( C \), the types \( B \) and \( C \) are both subtypes of \( A \), \( C \) is not assignable to \( B \), and the field \( f \) is defined with type \( A \) in the class \( A \) and with type \( C \) in the class \( C \). The first expression \( \tau_B = [A, \iota_C].f \) type checks using rules [T-Runtime-VAssign], [T-Read], and [T-Runtime-Frame], but after the execution step the expression \( \tau_B = \iota_C.f \) is ill-typed if including the assignability constraints, because \( C \) is not assignable.
to $B$, which is the type of $\tau_B$. Also, the configuration after the execution step is not an acceptable error because the right hand side of the assignment is not a value (see Figure 4.3). Execution will eventually reach a subtype-violation error, but in this case one additional execution step is needed.

From the progress and preservation lemmas we can obtain the soundness result: if an expression $e$ of type $T$ reduces to a normal form $e'$, then $e'$ is a value or $e'$ is stuck at an acceptable error (that is, not at a message-not-understood). Note that this soundness result applies to the message-safety type system with the assignability premises (although we prove it using the lemmas that consider the type system without those premises).

**Theorem 4.7.3 (Message-safety soundness)**

*If $\nu;\sigma;\emptyset;\Gamma_{\text{base}} \vdash e : T$ and $\sigma \text{ OK}$ and $\sigma \vdash \nu \text{ OK}$ and $e$, $\nu$, and $\sigma$ do not contain $\text{dynamic}$ and $\sigma_{\text{base}} \subseteq \sigma$ and $\langle \nu, \sigma, e \rangle \rightarrow^* \langle \nu', \sigma', e' \rangle$ and $e'$ is a normal form then

a) $e'$ is a value or

b) $\sigma';\nu' \vdash e' \text{ ERROR.}$

**Proof** We first show the desired property for the weakened form of the type system without the assignability premises. This property follows by induction in the derivation sequence $\langle \nu, \sigma, e \rangle \rightarrow^* \langle \nu', \sigma', e' \rangle$, applying Lemma 4.7.1, Lemma 4.7.2, and a minor lemma showing that all acceptable error configurations are normal forms (i.e. cannot be evaluated further). As a final step, soundness trivially also holds for the stronger type system where the assignability premises are present. □

Perhaps surprisingly, in the message-safety soundness theorem when the resulting expression $e'$ is a value, the type of $e'$ is not always a subtype of $T$. Recall that rule \textit{[Sub-Fun]} (Figure 3.6) requires subtyping for function return types but only assignability for function parameters. This means that subtyping is not transitive, even when $\text{dynamic}$ is not used. As an example, we may have an expression of type $T_1 = (\text{int}) \rightarrow \text{int}$, which in one step evaluates to an expression of type $T_2 = (\text{Object}) \rightarrow \text{int}$, which in turn evaluates to an expression of type $T_3 = (\text{String}) \rightarrow \text{int}$. We have $T_3 <: T_2 <: T_1$ but not $T_3 <: T_1$.

The message-safety soundness theorem is only concerned with terminating computations, but since every configuration that corresponds to a message-not-understood error is a normal form, it follows trivially that well-typed expressions can never lead to such an error, even in a non-terminating computation.

The variant with full type safety (Section 4.2) additionally rules out subtype-violation errors, and applies to both checked and production mode execution.
CHAPTER 4. MESSAGE SAFETY

Base cases

\[ \nu; \sigma \vdash t_{null}.p \quad \text{TYPESAFE ERROR} \quad \nu; \sigma \vdash t_{null}.f = t \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash t_{null}(t) \quad \text{TYPESAFE ERROR} \]

Inductive cases

\[ \nu; \sigma \vdash e_1 \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e_1.p \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e_1 \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e_1.p = e_2 \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e_2 \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e_1.p = e_2 \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e(t) \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e_1 \quad \text{TYPESAFE ERROR} \quad \text{for some } e_i \in \pi \]

\[ \nu; \sigma \vdash e(t) \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash [T, e] \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash e \quad \text{TYPESAFE ERROR} \]

\[ \nu; \sigma \vdash \tau = e \quad \text{TYPESAFE ERROR} \]

Figure 4.5: Acceptable runtime errors in full type safe programs.
The following theorem states full type safety for the strict type system. Figure 4.5 shows the acceptable error configuration for the full type safety, and the judgment $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash^{TS} e : T$ reads as follows: the expression $e$ is well-typed with type $T$ using the full type safe system (see Figure 4.2).

Conjecture 4.7.4 (Full Type-safety)

If $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash^{TS} e : T$ and $\sigma \text{ OK and } \sigma \vdash \nu \text{ OK}$ and $\sigma_{\text{base}} \subseteq \sigma$ and $e$, $\nu$, and $\sigma$ do not contain dynamic and $\langle \nu, \sigma, e \rangle \rightarrow^* \langle \nu', \sigma', e' \rangle$ and $e'$ is a normal form then.

a) $e'$ is a value or

b) $\sigma'; \nu' \vdash e'$ TYPESAFE ERROR.

Moreover, in both cases, $\sigma' \text{ OK and } \sigma' \vdash \nu' \text{ OK and } e'$, $\sigma'$, and $\nu'$ do not contain dynamic and $\sigma_{\text{base}} \subseteq \sigma'$.

Since our main focus is on message-safety, we omit the proof of full type safety.

4.8 Auxiliary Lemmas and Challenges in Coq

As we mention in Section 4.7, progress and preservation use additional auxiliary lemmas. In this section we show some of these lemmas along with a brief discussion of the main challenges arising when proving type soundness for message-safety in Coq. We provide a complete formal proof in Coq for soundness of message-safety, progress, preservation, and the some of the relevant auxiliary lemmas, along with a more exhaustive description here: http://www.brics.dk/fletch/. Some of the auxiliary lemmas are left unproven because they are trivial, while involving unnecessary extra Coq machinery. For example, some of these unproven lemmas show that inserting well-formed elements in well-formed environments preserves environment well-formedness, but this follows from the well-formedness definition for environments. We will now briefly discuss some of the lemmas used in the progress, preservation, and soundness proof.

Progress Auxiliary Lemmas The progress proof proceeds by induction on the typing derivation. Due to the complicated nature of the Fletch formal model, the base cases of the progress proof have been split in the lemmas described below.

The following lemma states that if a property access $i.p$ is well-typed, then it can be evaluated without causing a runtime type error.
Lemma 4.8.1 If $\sigma \trianglerighteq$ and $\sigma \vdash \nu \trianglerighteq$ and $\nu;\sigma;\Delta;\Gamma_{\text{base}} \vdash \iota.p : T$ then $\langle \nu,\sigma,\iota.p \rangle \rightarrow \langle \nu,\sigma,\iota' \rangle$ for some $\iota'$.

Similarly to Lemma 4.8.1, the following lemma states that if the expression $\iota_0.p = \iota_1$ is evaluated, then assignment does not causes a message-not-understood error, despite it can cause a subtype-violation error.

Lemma 4.8.2 If $\sigma \trianglerighteq$ and $\sigma \vdash \nu \trianglerighteq$ and $\nu;\sigma;\Delta;\Gamma_{\text{base}} \vdash \iota_0.p = \iota_1 : T$ then $\langle \nu,\sigma,\iota_0.p = \iota_2 \rangle \rightarrow \langle \nu,\sigma',\iota_2 \rangle$ for some $\sigma'$, or $\nu;\sigma \vdash \iota_0.p = \iota_2 \text{ERROR}$.

The following lemma states that a closure call can be successfully executed or fail with a subtype-violation error.

Lemma 4.8.3 If $\sigma \trianglerighteq$ and $\sigma \vdash \nu \trianglerighteq$ and $\nu;\sigma;\Delta;\Gamma_{\text{base}} \vdash \iota_0(\iota) : T$ then $\langle \nu,\sigma,\iota_0(\iota) \rangle \rightarrow \langle \nu,\sigma',\iota \iota'_T,\iota'_K \rangle$ for some $\sigma'$, $\iota'$, $T'$ or $\nu;\sigma \vdash \iota_0(\iota) \text{ERROR}$.

The following lemma states that constructors evaluation cannot fail with a runtime type error.

Lemma 4.8.4 If $\sigma \trianglerighteq$ and $\sigma \vdash \nu \trianglerighteq$ and $\nu;\sigma;\Delta;\Gamma_{\text{base}} \vdash \text{new} \quad N() : N$ then $\langle \nu,\sigma,\text{new} \quad N() \rangle \rightarrow \langle \nu,\sigma',\iota \rangle$ for some $\sigma'$, and $\iota'$.

The following lemma says that storing a closure in the heap never causes a runtime type error.

Lemma 4.8.5 If $\sigma \trianglerighteq$ and $\sigma \vdash \nu \trianglerighteq$ and $\nu;\sigma;\Delta;\Gamma_{\text{base}} \vdash (Gx) = e : T$ then $\langle \nu,\sigma,(Gx) = e \rangle \rightarrow \langle \nu,\sigma',\iota' \rangle$ for some $\sigma'$, and $\iota'$.

Preservation Auxiliary Lemmas The preservation lemma proceeds by induction on the semantic derivation and it involves several proof cases, due to the complex operational semantics rules (see Section 3.2). Therefore, we split the preservation proof into several lemmas, most of them concerning the store and heap environments.

We will now state the most relevant lemmas used in the preservation proof. The following lemma states that consistent updates of a well-formed store preserve the store well-formedness.
Lemma 4.8.8 For each \((\iota, \_ ) \in \sigma\), if \(\sigma \text{ OK and } \Delta \vdash \text{typeof}(\iota, \sigma) <: \text{typeof}(\tau, \nu)\) then \(\sigma \vdash \nu[\tau \mapsto \iota] \text{ OK}\).

The following lemma states that updating a well-formed heap with a well-formed object preserves the heap well-formedness.

Lemma 4.8.9 For each well-formed object \(o\), if \(\sigma \text{ OK and } \sigma \vdash \nu \text{ OK}\) then \(\sigma[\iota \mapsto o] \text{ OK and } \sigma[\iota \mapsto o] \vdash \nu \text{ OK}\).

The following lemma states that updating a field of a well-formed object with a value whose type is a subtype of the declared field type preserves the object well-formedness.

Lemma 4.8.10 If \(\sigma \text{ OK, and } (\iota_1, (c<\overline{G}>, \phi, \mu)) \in \sigma, \text{ and } (\iota_2, \_) \in \sigma, \text{ and } \emptyset \vdash \text{typeof}(\iota_2, \sigma) <: G, \text{ then } (c<\overline{G}>, \phi[f \mapsto (G, \iota_2)], \mu) \text{ is well formed.}\)

The following lemma is similar to weakening [42, 50, 66], except that it concerns runtime environments, i.e., store and heap, instead of typing environments.

Lemma 4.8.12 [Weakening-Like] If \(\sigma \text{ OK, and } \sigma \vdash \nu \text{ OK, and } \langle \nu, \sigma, e_1 \rangle \rightarrow \langle \nu', \sigma', e'_1 \rangle, \text{ and } \nu; \sigma; \Delta; \Gamma \vdash e_2 : T, \text{ then } \nu'; \sigma'; \Delta; \Gamma \vdash e_2 : T' \text{ and } \Delta \vdash T' <: T\).

The following lemma is called lemma of uniqueness of types and it expresses that expressions can only have one type. This lemma is standard in soundness proofs [66].

Lemma 4.8.13 (Uniqueness of Types) If \(\nu; \sigma; \emptyset; \Gamma \vdash e : T_1 \text{ and } \nu; \sigma; \emptyset; \Gamma \vdash e : T_2 \text{ then } T_1 = T_2\).

The following lemma shows that subtyping is reflexive. Although the proof of this lemma is trivial, it does not directly follow from the definition of subtyping, which is defined from typing rules in Figure 3.6.

Lemma 4.8.14 (Subtyping Reflexivity) For each type \(T, \Delta \vdash T <: T\).
CHAPTER 4. MESSAGE SAFETY

\[ [\text{T-List-Base}] \nu; \sigma; \Delta; \Gamma \vdash \emptyset : \emptyset \]

\[ [\text{T-List-Inductive}] \]

\[
\frac{
\nu; \sigma; \Delta; \Gamma \vdash \overline{\pi} : T
}{
\nu; \sigma; \Delta; \Gamma \vdash e : T
}\]

\[
\frac{
\nu; \sigma; \Delta; \Gamma \vdash e, e : T, T
}{
\vdash e, e : T
}\]

Figure 4.6: Expression typing for lists.

**Induction in Coq for Progress and Preservation** Coq requires a much deeper reasoning compared to hand-written formal models and proofs. For example, lists require careful considerations. Our formal model of Fletch contains function calls \( e_0(\overline{\pi}) \) accepting a list of arguments \( \overline{\pi} \). Although lists of expressions are simply represented with the \( \overline{\pi} \) notation in this dissertation, Coq requires a separate definition of list of expressions. This also requires a separate typing definition \( \nu; \sigma; \Delta; \Gamma \vdash e : T \) for lists (see Figure 4.6).

The progress proof requires two additional proof cases, not mentioned in Section 4.7. The first case involves proving that a well typed empty list of expressions can be executed without a message-not-understood error, whereas the second point requires to prove that a well typed list of expressions can be executed without causing message-not-understood. The Coq induction tactic (see Section 2.3) is not able to automatically generate induction cases suitable for proving the lemma, thereby requiring to manually specify the induction principle.

The same considerations apply to the preservation lemma: the operational semantics is defined by mutual induction and the preservation proof proceeds by mutual induction on the operational semantics.

**Soundness of Message-Safety in Coq** We conclude this section by showing the message-safety soundness theorem and proof in Coq, in order to give an intuition on how the Fletch formal model and proofs are represented in Coq.

\[
\text{Theorem type_soundness : } \forall \nu \nu' \sigma \sigma' e\ e', T,
\]

\[
(* \text{ PREMISES } *)
\]

\[
\text{fully_annotated_expression } e \rightarrow
\text{fully_annotated_subst } nu \rightarrow
\text{fully_annotated_store } \sigma
\rightarrow \text{well_formed_envs } nu \sigma \Gamma_0 \rightarrow
\text{well_formed_store } nu \sigma
\rightarrow
\text{lenv_base_superset } \sigma
\rightarrow \sigma \vdash_S nu
\rightarrow
nu; \sigma; \Delta_0; \Gamma_0 \vdash e : T
\rightarrow
<< nu, sigma, e >> || << nu', sigma', e' >>
\]
4.8. AUXILIARY LEMMAS AND CHALLENGES IN COQ

irred nu' sigma' e' \rightarrow
(* CONCLUSIONS *)
(error nu' sigma' e' \lor
(value e' \land \exists S, nu' ; sigma' ; Delta0 ; Gamma0 \vdash e' : S))
\land fully_annotated_expression e'
\land fully_annotated_subst nu'
\land fully_annotated_store sigma'
\land well_formed_envs nu' sigma' Gamma0
\land well_formed_store nu' sigma' \land sigma' |--S nu'
\land lenv_base_superset sigma'.

We will now briefly show how the definitions occurring in this theorem are represented in Conjecture 4.7.4. The predicates fully_annotated_expression e, fully_annotated_subst nu, and fully_annotated_store sigma correspond respectivelly to e, \nu, and \sigma do not contain dynamic. The predicate well_formed_envs nu sigma Gamma0 state that the environments are well formed mappings. The predicates well_formed_store nu sigma, sigma |--S nu, and lenv_base_superset sigma correspond respectively to, \sigma is well formed, \sigma \vdash \nu OK, and \sigma_{base} \subseteq \sigma.

We represent expression typing in Coq as follows: nu ; sigma ; Delta0 \vdash e : T. The predicate \langle \nu, \sigma, e \rangle \rightarrow^{\star} \langle \nu', \sigma', e' \rangle, and irred nu' sigma' e' means that e' is a normal form.

The conclusions of the theorem show that error nu' sigma' e', denoting \sigma'; \nu' \vdash e' ERROR, or value e' and \exists S, nu'; \sigma'; Delta0 ; Gamma0 \vdash e' : S, meaning that the expression e' is a value and it has type S. Moreover, we conclude that the evaluating the expression e preserves environment well-formedness. The conclusions of Conjecture 4.7.4 do not mention environment well-formedness and that computed values are well-typed. The rationale is that our main focus is on proving that well-typed programs, according to the message-safe type system, cannot fail with message-not-understood errors. However, our Coq theorem contains extra information, which is standard in soundness proofs [66]. We will now show part of the proof of the message-safety extracted from the Coq sources (see http://www.brics.dk/fletch/).

Proof.
intros nu nu' sigma sigma' e e' T FA_e FA_nu
FASigma
WFEnvs WFStore SigmaSSigma0 WFSigmaNu
Exp_typing RedStar Irred.
generalize dependent T.
induction RedStar; intros.
- Case "e \rightarrow^{\star} e".
CHAPTER 4. MESSAGE SAFETY

Our theoretical results summarized in Section 4.1 show that the message-safe type system guarantees the absence of message-not-understood errors for fully annotated programs. However, if a program that is accepted by the message-safe type system is partially annotated, message-not-understood errors can occur. Most of the real-world Dart code is partially annotated, as the experimental results in Chapter 6 show, since many real-world Dart programs massively use the type `dynamic`. Automatic code refactoring techniques might help adding type annotations to programs, but it might not be possible to add type annotations in many real-world Dart programs without introducing static type warnings or runtime type errors (see Chapter 6). However, it is still possible to show the absence of message-not-understood errors for fully-annotated parts of programs.

This section introduces the notion of local message safety, which consists on statically ensuring that message-not-understood errors cannot occur in program fragments. Section 4.4 already shows that message-safety is a local property, thereby allowing to show the absence of message-not-understood errors for program fragments. This section explains in more details the conditions program fragments need to satisfy in order to obtain message-safety. We now define more precisely the notion of program fragment.

**Definition** Given a Dart program $P$ with the corresponding concrete syntax tree $\text{ST}(P)$, the Dart program $P'$, represented by a list of concrete syntax trees
\( \mathcal{ST}(P') \), is a program fragment of \( P \) iff for each \( S \in \mathcal{ST}(P') \), \( S \) is a subtree of \( \mathcal{ST}(P) \).

For example, consider the following Dart program.

```dart
413 class C {
414     C foo(C x, dynamic y) {
415         var z = y.isEven;
416         x.bar().isEven;
417         print(x.foo(new C(), 0).bar().isEven);
418         return z;
419     }
420
421     int bar() => 0;
422
423     int add(int x) { return x + 1; }  
424 }
425
426 class D extends C {
427     Object bar() => ";
428     dynamic add(int x) { return "";
429 }
430     ...
431   new C().foo(new D(), 0);
```

Example 4.1: A partially-annotated Dart program that defines two classes.

The following parts of the program in the example are valid program fragments: the class \( C \), the method \( \text{foo} \), the declaration \( \text{var} \ z = y.\text{isEven} \), and both the expressions \( x.\text{bar()} \) and \( x.\text{bar()}.\text{isEven} \). From a practical perspective the most relevant program fragments are libraries, classes, and methods. For example, it is useful to ensure that some fully annotated libraries and classes are guaranteed not to fail with message-not-understood, even in programs that contain partially-annotated or non-annotated libraries.

The following definition shows when a type can be trusted, i.e., it provides meaningful informations that allow the message-safe type system to prevent message-not-understood.

**Definition** A type \( T \) can be trusted with respect to a property \( p \) if the following conditions hold:

- \( T \) does not contain any class type with generics
- \( T \) is not \texttt{dynamic}
- \( T \) is not a type parameter

We now define the notion of local message safety.
**Definition** A *program fragment* $F$ of a program $P$ is *message-safe* if the following holds:

1. Type-checking $P$ with the message-safe checker does not produce any type warnings in $F$, the fragment $F$ is fully annotated, and for each expression $\text{exp}.p$ in $F$, the static type of $\text{exp}$ can be trusted. Moreover, for each expression $\text{exp}(\ldots)$, $\text{exp}$ is a method invocation.

2. For each $\text{exp}.p$ in $F$, where $\text{exp}$ has type $T$, if $T$ is a class type, all the declaration of $p$ occurring in $T$ and all its subclasses, respect the requirement 2(a) defined in Section 4.1.

Let us revisit Example 4.1. A programmer might need to check if the program fragment identified by lines 416 and 431 is message-safe. Type-checking the fragment with the message-safe type checker does not raise any type warnings. However, the call $x.bar().isEven$ at line 416 is not message-safe because it violates the requirement 2 of local message-safety: $x$ has type $C$, and condition 2(a) of message-safety presented in Section 4.1 is not matched by the overriding method $\text{bar}$ defined in the class $D$, which is a subclass of $C$. Indeed, executing the code at line 423 would cause a message-not-understood error at line 416. Conversely, despite the $\text{add}$ method is overridden in the class $D$ with return type $\text{dynamic}$, and the both the classes $C$ and $D$ are not fully annotated, the method definition at line 423 is message-safe, enforcing that message-safety for program fragments is a local property, thereby involving only the specific program fragments and the related parts of code.

### 4.10 Experiments

Although the focus of this chapter is on the theoretical development of message safety, we have performed experiments to assess the impact of our approach for the Dart language. These experiments give insight into how existing Dart code violates the message safety requirements and whether such code is affected by the proposed modification of function subtyping.

#### 4.10.1 Violations of Message Safety

We expect that many existing Dart programs are already message safe and that violations are likely to indicate programming errors. To investigate whether this is correct, we applied our implementation\(^3\) to a range of publicly available Dart codebases: dart2js, dartanalyzer, Dart SDK, ace.dart, angular.dart, bot.dart, chrome.dart, dark, DartRocket, episodes.dart, force.dart, GoogleMaps, json.dart, MongoDB, PostgreSQL, presentation.dart, Protobuf,

---

\(^3\)The implementation and all benchmarks are available at [http://www.brics.dk/fletch/](http://www.brics.dk/fletch/)
Redstone, three.dart, XML, DartURL, and SecurityMonkey. These programs are, not surprisingly, only partly annotated with types, so they contain many violations of requirement 1 from Section 4.1. More interestingly, we find a small number of violations of requirements 2(a) (covariant return types for method overriding) and 2(b) (covariant return types for function subtyping): 17 of the 22 programs satisfy both requirements, 4 programs contain a total of 62 violations of 2(a) (most of them in dart2js and Dart SDK), and 2 programs contain a total of 4 violations of 2(b) (in dart2js and bot.dart). In the following we show some typical cases. In all cases where we propose a fix to a message safety violation and where a test suite is available, running the test suite on the modified program confirms that no subtype-violation errors are introduced by the changes.

Violations of 2(a) (Method Overriding)

We first describe two warnings among the 62 caused by non-covariant return types of overriding methods.

Example 1 The following code has been extracted from the file modelx.dart in dart2js (we highlight the most important parts):

```dart
432 class ElementX extends Element {
433   AnalyzableElement get analyzableElement {
434     ...
435   }
436 }
437
class CompilationUnitElementX
438 extends ElementX
439 implements CompilationUnitElement {
440   Element get analyzableElement => ...;
441 }
```

Our type checker warns that Element is not a subtype of AnalyzableElement, because of the return type of the overriding method analyzableElement. We have AnalyzableElement <: Element, which is enough to satisfy the ordinary Dart type checker, but message safety would require Element <: AnalyzableElement. We can easily fix this by changing the return type from Element to AnalyzableElement. Many others warnings follow this pattern, and they clearly indicate design oversight.

Example 2 The following code has been extracted from stream_controller.dart in the dart:async library in Dart SDK.

4Discussions on the dart2js forum [https://groups.google.com/a/dartlang.org/forum/#!topic/compiler-dev/DAcnoaugNHQ] confirm this conclusion.
abstract class _StreamImpl<T> extends Stream<T> {
  _BufferingStreamSubscription<T> _createSubscription(
      void onData(T data),
      Function onError,  // Note: Void function type check is performed.
      void onDone(),
      bool cancelOnError) {
    ...
  }
}

class _ControllerStream<T> extends _StreamImpl<T> {
  StreamSubscription<T> _createSubscription(
      void onData(T data),
      Function onError,  // Note: Void function type check is performed.
      void onDone(),
      bool cancelOnError) =>
    _controller._subscribe(onData, onError,
        onDone, cancelOnError);
}

At the overriding method _createSubscription, our type checker reports that the return type StreamSubscription<T> is not a subtype of _BufferingStreamSubscription. In this case, it may be problematic to specialize the return type of _createSubscription in the _ControllerStream class to _BufferingStreamSubscription since its method body might return an instance of StreamSubscription. Instead, it is safe to generalize the return type in the super-class to StreamSubscription<T>.

Violations of 2(b) (Function Subtyping)

Only 4 warnings are caused by non-covariant function return types. In each case, the fix is straightforward.

Example 3 The following code appears in dart2js in the file cps_ir_builder_visitor.dart:

class IrBuilderVisitor extends ResolvedVisitor<ir.Primitive>
    with IrBuilderMixin<ast.Node> {
  ir.Primitive visitConditional(ast.Conditional node) {
    return irBuilder.buildConditional(
        build(node.condition),
        subbuild(node.thenExpression),
        subbuild(node.elseExpression));
  }
}
4.10. EXPERIMENTS

Our type checker gives two warnings, at the second and third argument of the call to `buildConditional`: Both arguments have type `IrBuilder → Node`, and the formal parameters of `buildConditional` have type `IrBuilder → Primitive`. Since `Primitive <: Node`, the ordinary Dart type checker does not raise any warning, but `Node !:< Primitive`, so requirement 2(b) is violated.

The following code shows the definitions of `Primitive`, `Node`, `subbuild`, and `buildConditional`:

```dart
typedef ir.Node
typedef ir.SubbuildFunction(
    IrBuilder builder);
abstract class IrBuilderMixin<N> {
  return (IrBuilder builder) => 
    withBuilder(builder, () => build(node));
}
withBuilder(IrBuilder builder, f()) {
  assert(builder != null);
  IrBuilder prev = _irBuilder;
  _irBuilder = builder;
  var result = f();
  _irBuilder = prev;
  return result;
}
ir.Primitive build(N node) => node != null ?
    visit(node) : null;
ir.Primitive visit(N node);
}
ir.Primitive buildConditional(
  ir.Primitive condition,
  ir.Primitive buildThenExpression(IrBuilder builder),
  ir.Primitive buildElseExpression(IrBuilder builder)) {
  ...
}
abstract class Primitive
    extends Definition<Primitive> { ... }
abstract class Definition<T extends Definition<T>>
    extends Node { ... }
abstract class Node { ... }
```

The runtime type of the `buildConditional` arguments (i.e., the return value of `subbuild` function) will always be `IrBuilder → Primitive`, or else a subtype-violation would occur at runtime during the call to `visit`. We can safely change the return type of `SubbuildFunction` from `Node` to `Primitive`. This makes the program fragment message safe. No new type warnings appear after the change, and running the `dart2js` test suite does not break any tests.
Example 4 The following code has been extracted from the file number_enumerable.dart in bot.dart:

```dart
abstract class NumberEnumerable<T extends num> extends IterableBase<T> {
  num max() => this.reduce((num a, num b) => math.max(a, b));
  num min() => this.reduce((num a, num b) => math.min(a, b));
}
abstract class IterableBase<E> implements Iterable<E> {
  E reduce(E combine(E value, E element)) {
    Iterator<E> iterator = this.iterator;
    if (!iterator.moveNext()) {
      throw IterableElementError.noElement();
    }
    E value = iterator.current;
    while (iterator.moveNext()) {
      value = combine(value, iterator.current);
    }
    return value;
  }
}
```

Our type checker reports a warning at the arguments to the two calls to `reduce` in `NumberEnumerable`. The `max` and `min` methods call the `reduce` method, which is implemented in `IterableBase<E>`, with the `combine` parameter of type `(E, E) → E`. The `reduce` method parameter has type `(T, T) → T` in `NumberEnumerable<T extends num>`, where `T <: num`, and the type of the actual argument is `(num, num) → num`, and the runtime type is `(num, num) → num`. Since `T <: num`, but `num ≈ T`, we have that `(T, T) → T <: (num, num) → num`, and `(num, num) → num ≈ (T, T) → T` so requirement 2(b) is violated. We can remove these warnings by changing the program as follows.

```dart
abstract class NumberEnumerable<T extends num> extends IterableBase<T> {
  num max() => this.reduce((T a, T b) => (math.max(a, b) as T));
  num min() => this.reduce((T a, T b) => (math.min(a, b) as T));
}
```

If the `NumberEnumerable` is instantiated with `T = int`, then the `max` method could then fail with a cast error (in Dart, casts are written using the `as` operator). The difference is that, before the change the program could only fail in checked mode, and after the change it can in principle also fail in...
4.10. EXPERIMENTS

production mode (casts are also checked in production mode). The change does not introduce any new type warning and causes no failures of the bot.dart test suite. A more robust solution that does not involve cast operations can perhaps be obtained if Dart is extended with generic methods, which is already being considered for a future revision of the language.

4.10.2 Modifying Function Subtyping

Since both the static type system and checked mode program execution share the same subtyping relation, obtaining message safety guarantees in Dart requires not only modifying the static type system but also, which is more controversial, adjusting the rule for function subtyping (Figure 3.6) in checked mode runtime execution. The Dart language designers have confirmed that the consequences of the rule for function subtyping in the current language standard were not intended [64].

To demonstrate the need for the change, consider the following program:

```dart
class A {}
class B extends A { Object b; }
typedef A FA();
typedef B FB();
class C<X, Y extends X> {
  Y downcast(X x) { return x; }
}
A foo() => new A();

void main() {
  FA fa = foo;
  FB fb = new C<FA, FB>().downcast(fa);
  fb().b;
}
```

The `downcast` function implicitly performs a downcast from `FA` to `FB` without directly comparing function types. The program is type correct according to the message-safe type system, but it will fail in checked mode execution (as defined by the Dart specification, that is, with bivariant input function subtyping) during the `fb().b` field access, since the runtime type of `fb` will be `A` that does not provide the `b` field. By restricting the checked mode subtyping relation as suggested in Section 3.2.3, the program will fail at runtime at the assignment to `fb` since the result of the `downcast` invocation has type `() -> A` that is not a subtype of `() -> B`. The essence of the problem with the original semantics is that the type annotation `FB` of `fb` cannot be trusted. Changing the rule for function subtyping only statically, and not also in the runtime semantics, would not solve the problem, because type checking the `downcast` function does not use function subtyping. Of course, the error could also be caught statically by disallowing implicit downcasts entirely in the static type system, but that would result in a large number of spurious type warnings.
Now, the question is how to implement the proposed change and whether it will affect existing Dart code. For the first part, we find that the change requires only one new line of code in the Dart virtual machine and only one new line of code in the compiler runtime environment, with no measurable effect on the running time of either. For the second part, we tested if the change affects the dart2js compiler, which is presumably among the most complex Dart programs that exist. More specifically, we performed the following experiment using the co19 compiler test suite, which consists of 10264 tests. We exercised the dart2js compiler by running co19 on the modified virtual machine. If correctness of dart2js had relied on the original function subtyping rule, this would likely have caused some of the tests to fail. Nevertheless, even with such a complex program and an extensive test suite, not a single test case is affected.

Running the virtual machine test suite on the modified virtual machine resulted in 13 “failed” test cases, all related to subtyping of functions, which shows that the test suite is sufficiently extensive to detect the changed semantics and indicates that the change has no unforeseen consequences.

We have presented these results to the Dart language designers who now consider the proposal for an upcoming revision of the language standard.

4.11 Related Work

The variant of Featherweight Java by Mackay et al. [50] specifies a core of Java with mutable references. We have used that formalization as an inspiration for the overall approach in the creation of our Coq formalization of Fletch. Many parts are very different, however. In particular, we model first-class closures and unlimited lexical nesting. Access to mutable state in enclosing scopes is supported, and it uses a notion of an execution log rather than a traditional stack. The Dart approach to variance is very different from the approach taken in Java, but in the report [50] there is no notion of variance so we have added covariant generics to the model. Finally, our core conceptual contribution, message safety, puts the focus on the value of a consistent nominal commitment to lookup in a gradual typing context, and there is nothing similar in the report [50] or in the original work on Featherweight Java [42].

In R4RS [13] there is a dynamic semantics where lambda parameters are mapped to locations, which makes it possible to model mutable parameters in lexically nested scopes. Our approach differs from this in that we use substitution to make multiple usages of the same parameter distinct, and we use an ever-growing log to ensure that the lifetime of each parameter extends beyond the termination of the invocation that created it. We believe that these two models can be transformed into each other, but note that our model fits rather well in an object-oriented context because it corresponds closely to an implementation where stack frames are allocated in the heap (we just abstain
from modeling garbage collection).

Many papers present approaches to typing that allow for more flexibility than full type safety. We briefly present the most influential ones and the relations to our work.

An early approach which aims to reconcile the flexibility of dynamic typing with the safety of static typing is soft typing [10, 96]. The basic idea is that an expression whose type does not satisfy the requirements by the context is wrapped in a type cast, thus turning the static type error into a dynamic check. The Dart concept of assignability makes the same effect a built-in property of the dynamic semantics.

Strongtalk [7] is an early system with a similar goal, supporting very expressive (but not statically decidable) type specifications for Smalltalk. The Dart type system may have inherited the trait of being optional from there.

Pluggable type systems [6] are optional type systems that may be used with its target language as needed. The Dart language has been designed to enable the use of pluggable type systems (see [https://www.dartlang.org/slides/2011/11/stanford/dart-a-walk-on-the-dart-side.pdf]), e.g., by insisting that the dynamic semantics does not depend on type annotations (except for checked mode errors). This allows for a separate, strict type checker, and it also prepares the ground for the use of a message-safety checker.

Hybrid typing [31] combines static type checking with dynamic checking of type refinements based on predicates (boolean expressions). Of special interest is the potential for statically deciding some predicate based relations (e.g., the implication \( p_1 \Rightarrow p_2 \)), thus surpassing the static guarantees of traditional type safety. Given that this is concerned with strict static typing enhanced with dynamic predicates, there is little overlap with Dart typing.

Gradual typing [75] uses conventional type annotations extended with ‘?’, which corresponds to the Dart dynamic type. It builds on \( \text{Ob}_{<} \) [1] (i.e., it uses structural type equivalence and does not include recursive types), and hence the foundations differ substantially from Fletch. Their notion of type consistency does not have a corresponding concept in Fletch nor in Dart, but is replaced by our inclusion of dynamic in the subtype rules.

Contracts may contain executable code, thus checking of a contract may involve arbitrary computation (and hence, no static checking) in Scheme [15], with a special emphasis on tracking blame for first-class functions that only reveal typing violations when invoked. Neither Fletch nor Dart supports blame tracking, but it is not needed because the type of first-class functions can be checked when they are passed as an argument or assigned to a variable (by construction, they carry tags specifying the type).

Like types [97] were introduced recently, where usage of a like typed variable is checked statically, but it is checked dynamically that the value of such a variable actually supports the operations applied to it. It could be claimed that the point of the work on like types is to support structural typing to some extent, and no such support is present in Dart — checked mode checks
will fail for an assignment to an unrelated type, no matter whether the object in question would be able to respond to the messages actually sent.

The notion of type specificity in Dart is somewhat similar to the notion of naive subtyping used by Wadler and Findler [94] and by Siek and Wadler [77]. However, they differ in that Dart specificity is concerned with generic classes whereas those papers are concerned with function types, and Dart uses a different (and more permissive) rule for function types. Even the slightly more restrictive rule that we propose for Dart is still more permissive than naive subtyping in those papers.

Another recent paper presents progressive types [67], letting programmers tune the typing to allow or prevent certain kinds of runtime errors. Our work is similar in the sense that it enables programmers to rule out one kind of runtime type errors (‘message-not-understood’) and allow another (‘subtype violation’), but it differs because we start from a type system that is unsound, whereas a progressive type system with an empty $\Omega$ is a fully type safe system.

Finally, TypeScript [4] enables optional type annotations in JavaScript programs. Using structural types and coinductive subtype rules, the foundations differ substantially from Dart and Fletch. Moreover, TypeScript does not have a notion of checked mode execution.

All of these approaches aim to give various trade-offs between dynamic and static typing. However, none of them present a specific intermediate level of typing strictness similar to our notion of message-safe programs. Moreover, we believe our work is the first formalization of the core of Dart.

Success typing is a way to design complete but unsound type systems [48], that is, type systems where a statically detected type error corresponds to a problem in the code that definitely causes a runtime error if reached; the ‘normal’ is the converse, namely soundness, where programs with no static type errors will definitely not raise a type error at runtime. The point is that a complete (but unsound) type systems will avoid annoying programmers with a large number of unnecessary static type errors, and just focus on certain points that are genuinely problematic. The notion of related types [95] has a similar goal and approach, detecting useless code, such as if-statements that always choose the same branch, because the test could never (usefully) evaluate to true. The use of message-safe programs resembles a complete type system, but it is not identical: It is certainly possible to write a program that produces static type warnings in Dart which will run without type errors (so the typing is both unsound and incomplete), but the fact that message-safe programs prevent ‘message-not-understood’ errors offers a different kind of guarantee that success typing does not.
Chapter 5

Type Safety Analysis

According to the second part of the hypothesis of this dissertation, stated in Section 1.1, it is possible to design a sound program analysis that detects potential message-not-understood and subtype-violation errors, and the analysis can be successfully applied to real-world Dart programs.

This chapter shows the type safety analysis, which statically detects both message-not-understood and subtype-violation errors even for Dart code that contains type `dynamic`. Experimental evaluations show that the analysis can be applied to real-world programs producing only few false positives. The type safety analysis runs in two phases: a dataflow analysis providing an over-approximation of the set of types of each value computed by a program point, and a type checking phase using the type information provided by the dataflow analysis to raise static type warnings at places where message-not-understood or subtype-violation errors can occur. The dataflow analysis is sound, i.e., if a value of a type \( T \) is computed by a program point at runtime, the analysis infers a the type \( S \) for that program component, where \( T \) is a subtype of \( S \). Similarly, the type checking phase is sound: if a message-not-understood or a subtype-violation error can occur at runtime at a program point, then the type checker raises a static type warning at that component. For example, if the dataflow analysis infers that the variable \( x \) can contain both values of type `int` and values of type `String`, each runtime value assigned to the variable will have either dynamic type `int` or `String`, thereby allowing the type checking phase to predict all the possible message-not-understood and subtype-violation errors involving \( x \).

Alternatives to statically prevent message-not-understood and subtype-violation errors have been explored in the previous chapters, and the literature proposes different dataflow analysis techniques. We will briefly discuss these alternatives highlighting how they differ from the type safety analysis presented in this chapter. Chapter 3 shows a core of Dart and Chapter 4 clarifies its type system and identifies the precise causes of potential type errors. Chapter 4 also established two variations of the Dart 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 these type systems is that it provides no guarantees for program code that uses type *dynamic*.

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 [76] is based on unification, which is insufficient for object-oriented languages with subtyping, and the technique for ActionScript by Rastogi et al. [69] 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.

This chapter presents the following contributions:

- 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 design of this analysis is how to incorporate type annotations in a sound manner. We explore two main approaches: one that uses type annotations for *filtering* the flow of types through the program, and one that uses the annotations as specifications allowing more *modular* reasoning. Furthermore, we explain how such analysis techniques are affected if 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 [62], 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 [74]. 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
5.1. 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 [32], Self [2], or JavaScript [46].

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 a simple motivating example, consider the following program.

```dart
f(x) { 
    return x + " , world!";
}

String g(String y) {
    return f(y);
}

main() {
    print(g("Hello"));
}
```

Example 5.1: Optional type annotations.

A pure flow-analysis based technique would ignore the `String` annotations on line 550 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...
CHAPTER 5. TYPE SAFETY ANALYSIS

returned from \( g \), so that the return operation is also guaranteed to succeed (i.e., without a subtype-violation error in checked mode execution).

In contrast to such a whole-program flow analysis, an analysis that takes the type annotations into consideration 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 \[17\] 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 \[46\] is more suitable.

5.2 Trusting Type Annotations

If type annotations could always be trusted, integrating the type annotations would be a straightforward modification of the flow analysis: when modelling 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 modelling 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.

```dart
class Cell<T> { T f; }
void main() {
  Cell x = new Cell();
  Cell<String> y = x;
  x.f = 42;
  var z = y.f.substring(2);
  print(z);
}
```
5.2. TRUSTING TYPE ANNOTATIONS

Example 5.2: 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 560, 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 559 and the use of y on line 561, 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 message-safe type system in Chapter 4 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 559 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 558. Another candidate is line 560; 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 of those options, we choose for our analysis design to report the errors where they may occur at runtime, in this case line 561.

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.

Example 5.3: Dynamic return type.

The assignment in line 566 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 567 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 \texttt{String}. Furthermore, a consequence of a design flaw in the type rule for function subtyping is that function return types cannot be trusted, even without type \texttt{dynamic} (see Chapter 4). More generally, if a variable \( x \) in a Dart program has (non-\texttt{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, the field \( f \) in the subclass \texttt{B2} has a different type than in the superclass \texttt{B1}, which is allowed in Dart:

```dart
569 class A1 {}
570 class A2 extends A1 { String s = "foo"; }
571 class B1 { A2 f = new A2(); }
572 class B2 extends B1 { A1 f = new A1(); }
573 void main() {
574    B1 x = new B2();
575    print(x.f.s);
576 }
```

Example 5.4: Contravariant field overriding.

The declared type \texttt{B1} of \( x \) suggests that \( x.f.s \) exists, but the program fails in line 575 at runtime because \( x.f \) is in fact of type \texttt{A1}. If we want to catch this error statically, we could require invariant field overriding, as we explain in Chapter 4, and issue a warning at line 572. However, that might be too restrictive, and not in the Dart spirit. A better option is to issue a warning at line 575 where the runtime error occurs. This can be achieved based on the reasoning that \( x \) has declared type \texttt{B1}, so it may hold a value of type \texttt{B2} and therefore \( x.f \) may have type \texttt{A1}, which is not a subtype of \texttt{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 \texttt{B2}.

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

### 5.3 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 5.10). 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.
5.4. A VARIATION OF FLETCH

\[
\text{class } \quad C ::= \begin{aligned}
\text{class } \; N<\U \text{ extends } T \rangle \; \text{ extends } \; N \{ \\
\text{T } f = e \\
T \; m(T \; x) \Rightarrow F \; e
\} \\
\text{typedef } D ::= \begin{aligned}
\text{typedef } \; T \; F(T)
\end{aligned}
\]

\[
\text{expr. } \quad e ::= x \mid e.f \mid e.m \mid \text{new } N<\T>() \mid x = e \mid e.f = e \mid e(e) \mid T \; (T \; x) \Rightarrow F \; e
\]

\[
\text{type } \quad T ::= N<\T> \mid F \mid U \mid \text{dynamic}
\]

Figure 5.1: Syntax of the simplified Dart language.

**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 5.1. 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 5.1.

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, modelling 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.
5.4 A Variation of Fletch

We begin by defining a variation of Fletch (see Chapter 3) that we use for presentation of our analysis. Figure 5.1 shows the syntax of the language: some language features have been ruled out because they are not relevant in describing our analysis, whereas other features have been introduced with the purpose of simplifying the description of the analysis. The calculus used in this chapter 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). Instead, Fletch uses the symbols \( c, d \) to denote class names and \( X \) for type parameters. We occasionally misuse notation slightly and use e.g. \( N \) as a metavariable ranging over the set of class names rather than as the set itself. We use the notation \( X \) as in Chapters 3 and 4 to denote a possibly empty list \( X_1, \ldots, X_n \). 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. Similarly to Fletch, classes are parameterized by types, written in \( < \ldots > \), 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 \( N<T> \), a type parameter \( U \) defined for the surrounding class, a function type \( F \) (defined by a typedef), or the type \( \text{dynamic} \). Note that our core language restricts Fletch functions to one parameter, and it includes a label \( F \) which plays two roles: is denotes function type names in typedef declarations and it labels methods and anonymous functions. A method \( T r (T x x) \Rightarrow F e \) 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. We assume the analysis is applied to well-formed programs; we use the same notion of well-formedness introduced in Chapter 3. We also use the same types defined in Chapter 3, but we rule out the type \( \perp \) and \( \text{void} \), while still including the predefined type \( \text{Object} \). To simplify the presentation, we also assume that every function and method has a single parameter, 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. Our implementation described in Section 5.10 naturally covers the full Dart language.

We omit a formalization of the language semantics due to the limited space; Chapter 3 already shows a formalization of the related language Fletch. Note that, as in Fletch, a dynamic programming style can be emulated by using
5.5. ABSTRACT DOMAIN FOR TYPES

\[
\begin{align*}
\text{inferred type} & \quad \tau ::= N<P>^\kappa \mid F<P>^\kappa \mid U^\kappa \\
\text{kind} & \quad \kappa ::= C \mid A \\
\text{type arguments} & \quad P ::= T \mid ? \\
\text{type sets} & \quad S ::= \{\tau\}
\end{align*}
\]

Figure 5.2: Abstract domain.

the type dynamic everywhere.

5.5 Abstract Domain for Types

We express our analysis using set constraints on a collection of type variables [39, 62]. 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 \( F.x, F.r \) to denote its parameter and its return expression, respectively.

Type variables range over sets \( S \) of inferred types shown in Figure 5.2. An inferred type \( \tau \) is an object type \( N<P>^\kappa \), a function type \( F<P>^\kappa \), or a type parameter \( U^\kappa \). 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 \( T \) 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 \( P \).

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.

5.6 Type Inference Constraints

The inference phase of the analysis is expressed using constraints on the type variables as shown in Figure 5.4. 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 [39, 62, 82].

To explain the different analysis modes \( m \in \{\text{Flow, Filter, Modular}\} \) in a uniform manner, we make use of some auxiliary definitions shown in Figure 5.5. Intuitively, \( \text{flow}_m \) and \( \text{decl}_m \) determine what types can be inferred on the basis of dataflow and type annotations, respectively. The operator

\[ F<P> \] in our notation corresponds to \( \text{type}(F.x, P) \to \text{type}(F.r, P) \) in Chapter 3.
\( \prec \) is a variant of Dart’s subtyping relation that takes kinds (C and A) and type parameters into account. The bind<sub>m</sub> function used in some of the constraint rules models binding of type parameters in inferred types, as explained informally in the following subsections.

5.6.1 Constraint Rules

The constraint rules in Figure 5.4 present the three modes of our dataflow analysis. To keep the presentation simple, we will first show the core of our dataflow analysis, omitting details about all those functions parameterized by the analysis mode \( m \), e.g., flow and decl. The simplified rules are presented in Figure 5.3. Sections 5.6.2, 5.6.3, and 5.6.4 show the variations of the constraints for each analysis modes.

Rule [Method], models the flow of types for methods. The inferred type for a method definition \( m \) in the class \( N \) is the concrete function type \( F<?>^C \). The type \( F<?> \) denotes a function type, where \( F \) is the label of the method \( m \) (see Section 5.5) and all the type parameters defined in the class \( N \) and occurring in the type of the method \( m \) with label \( F \) are replaced by \( ? \). Rule [FIELD] models the dataflow for fields. The flow of the field initializers expression \( e \) is propagated to the field \( f \) defined in the class \( N \). The following example attempt to clarify the effect of these two constraints rules on the analysis.

577 class Base<\( X \)> {
578  X foo(int x) => \( F \) x;
579  dynamic f = 0;
580 }

By Rule [Method] we can conclude \([\text{Base}.\text{foo}] = \{F<?>^C \}\), where the inferred type denotes a function type whose input type is \( \text{int} \) and output type is \( ? \). Similarly, by Rule [FIELD] we can conclude \([f] = \{\text{int} \}\).

Given a class \( M \) extending a class \( N \) and overriding a field \( f \), Rule [Inh1] models field inheritance by propagating the flow of types in the overriding field \( N.f \) to the overridden field \( M.f \) in the subclass. Similarly, Rule [Inh2] models method inheritance. According to Rule [Obj], the inferred type for the instance creation \( \text{new} \ N<\overline{T}>() \) is the class type \( N \) whose generic arguments depend on the analysis mode, as we will show later in Sections 5.6.2, 5.6.3, and 5.6.4. Rule [FUN] models the flow of functions similarly to Rule [Method].

Since Dart supports higher order functions and methods are semantically modeled as tear-off closures, Rule [READMETHOD] infers function types for each method lookup. Inferred types for method calls, e.g., \( e.m(\epsilon') \), are thereby obtained in two phases: first rule [READMETHOD] infers function types for \( e.m \), then the general rule for function application [App] infers the call type basing on each function return type inferred for \( e.m \). The premises of Rule [READMETHOD] captures all the inferred types \( N<P>^C \) for the method target \( e \), including all the subtypes of the abstract types. For each of these class types with class name \( N \), the analysis builds an implicit
5.6. **TYPE INFERENCE CONSTRAINTS**

```plaintext
class N ... {
    T m(T x x) ⇒ F e
    T f = e
}

class M ... extends N ...

for each non-overridden field

for each non-overridden method

new N<T>()

T (T x x) ⇒ F e

e.m

e.f

x = e

e.f = e'

e(e')
```

Figure 5.3: Core constraints for the pure dataflow analysis of our simplified Dart language, without taking into account the analysis modes.

call graph by statically dispatching each class N to the method N.m. Then, the analysis propagates the inferred types for each obtained constraint variable [N.m] to the method lookup e.m. Rule [ReadField] is a variation of Rule [ReadMethod] for fields. Rule [WriteVar] models the flow of variable write. The dataflow of the right-hand side expression e is propagated to the left-hand side x and to the assignment x = e. Rule [WriteField] models the flow of field assignments. The inferred types of the right-hand side e′ are propagated to the field assignments e.f = e′ and the respective field f declarations. Similarly to rule [ReadField], the analysis first models each receiver field N.f, based on the inferred class types in e, and subsequently propagates the flow of e′ to N.f. Since our analysis is not context sensitive, we model
a single type variable \([N.f]\) for field \(f\) defined in the class \(N\). However, we will show in Section 5.6.3 that we can regain some precision with an implicit form of context sensitivity in filter mode. Rule \([\text{App}]\) infers types for function calls \(e(e')\). For each inferred function type \(F<P>\) of \(e\), the analysis propagates the flow of the argument \(e'\) in the function parameter \(F.x\), and similarly propagates the flow of the return expression \(F.r\) to the call \(e(e')\).

Algorithm 1 CHA Algorithm

1: function CHA(T)
2:    result ← ∅
3:    for each \(C ∈ \text{Subclasses}(T)\) do
4:        \(C' ← \text{ApplySubstitution}(C,T)\)
5:        if \(C' <: T\) then
6:            result ← result ∪ \{C'\}
7:        end if
8:    end for
9:    return result
10: end function

Class Hierarchy Analysis Our dataflow analysis implements class hierarchy analysis [17], which consists on finding all the subtypes of a type. Algorithm 1 shows our implementation. Since Dart supports generic types, we propagate generic arguments substitution to the list of subclasses resulting from class hierarchy analysis, as the following example shows.

581 class A<X, Y> {  
582     ...
583 }  
584 class B<Z> extends A<int, Z> {  
585     ...
586 }  

When the algorithm that implements our type safety analysis applies class hierarchy analysis, it will call a function similar to the CHA function described in Algorithm 1. Suppose our inference algorithm calls \(\text{CHA}(A<\text{int}, \text{bool}>)\). The call to \(\text{Subclasses}(A<\text{int, bool}>)\) results in the list containing the sole element \(B<Z>\). Despite \(B\) being a subclass of \(A\), \(B<Z>\) is not a subtype of \(A<\text{int, bool}>\). However, the call to \(\text{ApplySubstitution}(B<Z>, A<\text{int, bool}>)\) instantiates the \(Z\) type parameter according to \(A<\text{int, bool}>\) using the following steps. First, since we are applying class hierarchy analysis on \(A<\text{int, bool}>, i.e., X and Y, have respectively type \text{int} and \text{bool}. The class \(B\) with parameter \(Z\) extends \(A<\text{int, Z}>, meaning that \(Z = Y\). Since \(Y = \text{bool}\), then \(Z = \text{bool}., and
5.6. TYPE INFERENCE CONSTRAINTS

```plaintext
class N ... {
    T m(Tx x) ⇒F e {F<?>C} ⊆ [N.m], decl_m(Tx) ⊆ [x] [METHOD]
    T f = e flow_m([e], type(N.f, ?)) ⊆ [N.f] [FIELD]
}

class M ... extends N ... flow_m([N.f], type(N.f, ?)) ⊆ [M.f] [INH1]
for each non-overridden field f
[N.m] ⊆ [M.m] [INH2]
for each non-overridden method m

new N<T>() bind_m({N<T>}, T) ⊆ [new N<T>()] [OBJ]
T (Tx x) ⇒F e {F<?>C} ⊆ [T (Tx x) ⇒F e], decl_m(Tx) ⊆ [x] [FUN]
e.m
  τ ∈ [e] N<P>^e ≤ τ
  T = type(N.f, P) bind_m(flow_m([N.f], T), P) ⊆ [e.m] [READMETHOD]
e.f
  τ ∈ [e] N<P>^e ≤ τ
  T = type(N.f, P) bind_m(flow_m([N.f], T), ?) ∪ decl_m(T) ⊆ [e.f] [READFIELD]
e.x = e
flow_m([e], type(x, ?)) ⊆ [x] ∩ [x = e] [WRITEVAR]
e.f = e'
  τ ∈ [e] N<P>^e ≤ τ
  T = type(N.f, P)
  T_x = type(F.x, P) bind_m(flow_m([e'], T), ?) ∪ decl_m(T) ⊆ [N.f],
  T_x = type(F.x, P) flow_m([e'], T) ∪ decl_m(T) ⊆ [e.f = e'] [WRITEFIELD]

F<P>^e ∈ [e] T_x = type(F.x, P)
  T_x = type(F.x, P) bind_m(flow_m([e'], T_x), ?) ⊆ [F.x],
  bind_m(flow_m([F.r], T_x), P) ⊆ [e.f = e'] [APP]
e(e')
```

Figure 5.4: Core constraints for the simplified Dart language, using mode
m ∈ {Flow, Filter, Modular}.

ApplySubstitution(B<Z>, A<int, bool>) = B<bool>. Since B<bool> is a sub-
type of A<int, bool>, the call to CHA(A<int, bool>) will result in
{A<int, bool>, B<bool>}. 
We improve precision of our analysis using rapid type analysis \[3\], which rules out all the classes that are never instantiated in programs. Let us consider the following Dart program.

```dart
588 class Base {}
589 class Derived1 extends Base {}
590 class Derived2 extends Base {}
591 main() {
592   print(new Derived1());
593 }
```

The call to CHA(\text{Base}) will result in \{\text{Base, Derived1}\}. Indeed, despite Derived2 being a subtype of \text{Base}, it is never instantiated in the program above.

### 5.6.2 Constraints for flow mode

Flow mode ignores type annotations and only propagates concrete types, similar to an Andersen-style points-to analysis \[82\] although inferring types instead of abstract heap locations. For \(m = \text{Flow}\), \text{bind}_m and \text{flow}_m are simply the identity function in their first argument and ignore the second argument, \text{decl}_m always returns the empty set of types, and \(\preceq\) is the identity relation.

In Figure 5.4, the constraint rule \[
\text{Obj}
\] 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 \[
\text{Fun}
\] and \[
\text{Method}
\] similarly infer concrete function types for function expressions and method declarations, respectively. \[
\text{WriteVar}
\] models the flow of types at variable assignments. Method and field reads (including method dispatch) are modeled by \[
\text{ReadMethod}
\] and \[
\text{ReadField}
\], respectively. Field assignments are similarly handled by \[
\text{Field}
\] and \[
\text{WriteField}
\], and \[
\text{Inh1}
\] and \[
\text{Inh2}
\] model the flow of types for inherited but not overridden fields and methods. Finally, \[
\text{App}
\] 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.

### 5.6.3 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 \[82\]. Revisiting Figure 5.5 with \(m = \text{Filter}\), the function \text{flow}_m(S, T) now performs this filtering: it only admits types from the type set \(S\) that are subtypes of \(T\), thereby modelling the effect of subtype checks in Dart checked mode execution. The relation \(<:\) denotes subtyping, with special treatment of \texttt{dynamic} and with covariant generics, as formalized in Fletch (see Chapter 3). We make use of a static type resolution mechanism, \texttt{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 type annotation or \texttt{dynamic} by default. We must be careful with generic types as demonstrated in Section [5.2]. In rule [WriteVar], \texttt{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 [5.8].)

We therefore conservatively interpret the type parameters as \texttt{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 [5.9].) An alternative to replacing type parameters with \texttt{dynamic} is to use the bounds of the type parameters, but that would be unsound since they can be invalidated by the use of \texttt{dynamic} as type argument, so we only do that if ‘optimistic’ is enabled.

Type parameters occurring in [e], for example if \( e \) is \texttt{new Cell<T>} inside the \texttt{Cell} class, are treated similarly.

Type parameters in object types are now relevant, unlike in flow mode. The \texttt{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 \texttt{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 \texttt{Cell<T> h(Cell<T> p)} \Rightarrow \texttt{F_3} \ p in the \texttt{Cell} class from Example [5.2] and that we have \([x] = \{\texttt{Cell<int>}^{C}\}\) for some variable \( x \). The \texttt{bind}_m function then ensures that the expression \( x.h(x) \) has type \texttt{Cell<int>}^{C}: applying rules [Method] and [ReadMethod] gives \( \texttt{bind}_m([\texttt{Cell.h}], \texttt{int}) = \texttt{bind}_m([\texttt{F_3<int>}^{C}], \texttt{int}) = \{\texttt{F_3<int>}^{C}\}, \) therefore \([x.h] = \{\texttt{F_3<int>}^{C}\}\) and finally \([x.h(x)] = \{\texttt{Cell<int>}^{C}\}\) 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 \texttt{N<T}> in \([e]\) (\( \Rightarrow \) is still just the identity function), \texttt{type(N.f, P)} gives us the type of the \( f \) field in \( N \) using generic type arguments \( P \). The use of \texttt{bind}_m here effectively erases any type parameters that may appear in \([e']\) by substituting them by \texttt{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 \texttt{N<T>} 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 \texttt{flow}_m to filter the types in \([N.f]\) according to \( P \). As an example, consider an expression \( x.f \) in a situation where \([x] = \{\texttt{Cell<String>}^{C}\}\) and \([\texttt{Cell.f}] = \{\texttt{String}^{C}, \texttt{int}^{C}\}\) (i.e. in some instantiations of \texttt{Cell}, the \( f \) field holds a string
flow_m(S, T) = \begin{cases} 
S & \text{if } m = \text{Flow} \lor T = \text{dynamic} \lor T = \text{Object} \\
\{X^e \in S \mid \text{stype}(X, ?) <: \text{stype}(T, ?) \land (m = \text{Filter} \lor ftype(X))\} & \text{if } m = \text{Filter} \lor T = \text{dynamic} \lor T = \text{Object} \\
\emptyset & \text{otherwise} 
\end{cases}

\text{decl}_m(T) = \begin{cases} 
\{T^A\} & \text{if } m = \text{Modular} \land (tpar(T) \lor (T = N<^TT> \land T \neq \text{Object} \land \text{opt})) \\
\{N<^T?>^A\} & \text{if } m = \text{Modular} \land (T = N<^TT> \land T \neq \text{Object} \land \neg \text{opt}) \\
\emptyset & \text{otherwise} 
\end{cases}

\tau_1 \ll \tau_2 \text{ if } \tau_1 = \tau_2 \lor (\tau_1 = T^1 \land \tau_2 = T^2 \land \text{stype}(T_1, ?) <: \text{stype}(T_2, ?))

Figure 5.5: 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 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 Jx.fK = \{String\}.

5.6.4 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 = dynamic, flow_m and decl_m behave as in flow mode. There are two important exceptions to these rules, however (see Figure 5.5):

First, we must take into account that not all type annotations can be trusted (cf. Section 5.2). If T is an object type N<^TT> we therefore replace 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 FC). In principle, we could instead use abstract function types (of form FA), which might be more in the spirit of modular inference, however,
5.7. OPEN WORLD ASSUMPTION

Our analysis can run in modular, filter, and flow mode. Additionally, each of these modes can run with either closed world assumption or open world assumption. The former is based on assuming that all the analyzed Dart code is visible, and it corresponds to the analysis described in Figure 5.4, whereas the latter assumes the same part of the analyzed program can be used by invisible code. Open world assumption is an extension of closed world assumption, and it is necessary to analyze libraries, since they can be used by any application, which is not necessary visible at analysis time. Closed world assumption is used to analyze applications instead, assuming any external Dart code can call its functions. When running open world assumption, the inference can infer an additional kind of type: external types. We thereby extend the abstract domain defined in Figure 5.2 with the kind $E$, denoting types coming from external flow. This allows separating causes of potential errors while issuing type warnings (see Section 5.9), since the type checker
can show if a warning is caused by external or internal flow. The significant
difference between abstract types $T^A$ and external types $T^E$ is that our analysis
can apply rapid type analysis for the former, but not for the latter, since the
set of all instantiate class is unknown.

Let us consider the following example.

```plaintext
594  _bar(y) => y;
595  _bar2(z) => z;
596
597  foo(x) {
598     x++;
599     _bar(0);
600     _bar2(x);
601     return x;
602 }
603
604  foo(10);
```

If we assume that the code above is part of a library, since the `foo` function
is public, it can be called by any client application that is not visible at analysis
time. Because our analysis is sound, we have to infer all the possible types
that can flow at a program point, which implies that our analysis should infer
the type `Object` if external invisible values can flow into a program point. In
close world assumption, i.e., assuming that all the code is visible, our analysis
would infer $[x] = \{\text{int}^C\}$. If we use this type information, we can show
that message-not-understood errors cannot occur at line 598, because integer
numbers supports the increment operator. However, if a client application
performed the call `foo("")`, a message-not-understood error would occur at
line 598 because `String` does not provide the increment operator. If `x` can
be assigned to every value at runtime in open world assumption, our analysis
would have to infer $[x] = \{\text{int}^C, \text{Object}^E\}$. Note that the, since `foo` can be
called with any input value, the type `Object` is external, meaning that any
possible subtype of `Object` can flow in `x`.

The `_bar` and `_bar2` functions are private, because any function or vari-
able with a name starting with the `_` symbol is private in Dart. Clearly, the
closed world assumption applies for private variables and function parameters,
because all the code that can potentially assign values to these variables or
functions is visible during the analysis. For example, we infer $[y] = \{\text{int}^C\}$
because of the call at line 599. However, values flowing from invisible code to
`x`, can also flow to `z`. Therefore, despite `z` being the parameter of a private
function, our analysis should infer $[z] = \{\text{Object}^E\}$. Note that it is possible
to track the public member causing an external type flowing into `z`; in the
example the parameter `x`.

Since Dart supports higher order functions, we also must take into account
escaping functions, as the following example shows.

```plaintext
605  class PublicClass {
```
5.8 DISCUSSION

The analysis infers \([x] = \{\text{int}^C\}\) in closed word. The callback field defined in PublicClass is public, so it can be accessed by external invisible code in open world assumption. Despite _square begin private, an external application including the code that declares PublicClass can contain an expression, such as, new PublicClass().callback(""), causing a message-not-understood error at line 607. Therefore, inferring external types at declaration sites is not sufficient. Our analysis can detect each escaping function, in our example _square is assigned to the field callback, and infer the respective external object type for the function parameters. In this case, the result would be \([x] = \{\text{int}^C, \text{Object}^E\}\). We are able to track the external flow of types in for all escaping functions, which occur at field initialization and variable initialization - the latter unsupported by our core language, but supported by the Dart language - and at methods and function return.

5.8 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 5.2, we would then have distinct type constraint variables \([\text{Cell<String>}.f]\) and \([\text{Cell<dynamic>}.f]\) for the \(f\) field of Cell, whereas we now only have one, \([\text{Cell}.f]\). Although our simple analysis design is capable of recovering some precision at property access operations by the use of filtering, as discussed in Section 5.6.3, 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<??>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 [11].

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 $\text{Object}$. 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 dissertation.

**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 5.10.

**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 in Chapter 4 is another approach to sound type checking for Dart. Its precision is incomparable to our type safety analysis—even when restricting to Dart programs with no occurrences (explicitly or implicitly) of $\text{dynamic}$ in type annotations, type arguments, or closure return types. As an example, the full type safety system rejects Example 5.4 even if removing line 575, 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.
5.9 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 \( \text{Object}^A \). In Dart, function calls and method calls technically involve looking up the \( \text{call} \) 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

\[
f(A1 \ x)\{ \text{var } y = x; \text{ return } y.s; \}
\]

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 \( \tau \) in \([e]\) is not a subtype of the declared type of \( x \). Occurrences of \( ? \) or \( \text{dynamic} \) in an inferred abstract type \( \tau \), as e.g. in \( \text{Cell}<?^A> \), can denote any object and are accordingly treated as \( \text{Object} \) 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 \( \text{Object}^A \).) 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 \( \text{Object} \) in the current class, and where the possible values of \( y \) may be of type \( \text{int} \) or \( \text{String} \). Filter mode inference essentially ignores the type annotation for \( y \), as explained in Section 5.6.3, yielding \([y]=\{\text{int}^C,\text{String}^C\}\). 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 \([y]=\{T^A\}\). 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 5.8, 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 2.2 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 5.2. Analyzing Example 5.2 with filter mode gives $J_y = \{ \text{Cell}\langle \text{dynamic}\rangle \}$ and $J_y.f = \{ \text{int} \}$, and $\text{int}$ does not have the property $\text{substring}$, so a message-not-understood warning is generated at line 561. In modular mode we have $J_x = J_y = \{ \text{Cell}\langle ?^>^A \rangle \}$, $J_y.f = \{ \text{Cell}.f \}$, resulting in the same warning but also a (spurious) subtype-violation warning at line 559. Notice that modular mode inference in this case automatically falls back to resolve $J_y.f$ based on dataflow due to the use of $\text{dynamic}$.

Considering Example 5.3, let $F_3$ be the label of the anonymous function in line 566. In both filter mode and modular mode, using rules $[\text{Fun}]$ and $[\text{WriteVar}]$ gives $[f] = \{ F_3<?^>^> \}$ and $[\text{New}]$ gives $[F_3,r] = \{ \text{Object}\}$ (since $F_3.r$ is the expression $\text{new Object()}$). Rule $[\text{App}]$ now gives $[fO] = \{ \text{Object}\}$ in both modes, and therefore our type checker raises a warning on the assignment at line 567, because $\text{Object}$ is not a subtype of $\text{String}$.

Analyzing Example 5.4, which does not involve type $\text{dynamic}$, with filter mode gives $[x] = \{ \text{B2}\}$ and $[x.f] = \{ \text{A1}\}$, and $\text{A1}$ does not have the property $\text{s}$, again resulting in a message-not-understood warning. With modular mode we instead have $[x] = \{ \text{B1}\}$ and $[x.f] = \{ \text{A2}, \text{A1}\}$, reaching the same conclusion. Notice in this last case that looking up $f$ in $\text{B1}$ involves not only the $\text{B1}$ class itself but also its subclass $\text{B2}$ (cf. rule $[\text{ReadField}]$ and the defi-
5.10 Evaluation

Our overall hypothesis, as described in Section 1.1, is that it is possible to integrate type annotations into a sound dataflow analysis to effectively check absence of message-not-understood and subtype-violation errors in real-world Dart programs.

More specifically, we perform a set of experimental evaluation to answer the following questions.

1. **Precision of flow mode, filter mode, and modular mode**: our analysis supports 3 major modes, namely flow mode, filter mode, and modular mode. The first is a pure dataflow analysis, the second is a restriction of flow mode where type annotations are used as filters, and the latter trusts type annotations modularly. Although we know that filter mode is always more precise than flow mode and modular mode, we want to investigate the impact of these analysis versions on real-world programs in term of potential message-not-understood and potential subtype-violation detected by our analysis.

2. **Optimistic assumptions**: what is the effect of enabling ‘optimistic’ treatment of type annotations in modular mode? This configuration is only conditionally sound, so it is also interesting to investigate whether the condition is satisfied in practice.

3. **Intersecting modular mode and filter mode**: modular mode type inference is always less precise than the filter mode inference. However, if a subtype-violation error is raised in modular mode, it might not be raised in filter mode. Nevertheless, since both analysis are sound, if a subtype-violation error is not detected by both analysis modes, it can be safely omitted, that is, it will never occur at runtime. We want to investigate how much our analysis precision improves in terms of subtype-violation potential errors by counting the intersection of subtype-violation errors raised by both analysis.

4. **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. Some warnings, especially in modular mode, may also indicate fragile code.

Although precision is our main objective, analysis time is of course also relevant.
### Table 5.1: Benchmarks used for the evaluation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC excl./incl. deps.</th>
<th>Baseline MUN / SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>dart2js</td>
<td>102 718 / 104 414</td>
<td>60 431 / 82 571</td>
</tr>
<tr>
<td>analyzer</td>
<td>83 410 / 86 297</td>
<td>37 213 / 53 876</td>
</tr>
<tr>
<td>devcompiler</td>
<td>66 247 / 159 883</td>
<td>74 158 / 11 034</td>
</tr>
<tr>
<td>dartstyle</td>
<td>3 591 / 73 765</td>
<td>3 291 / 3 415</td>
</tr>
<tr>
<td>linter</td>
<td>2 236 / 77 535</td>
<td>938 / 1 343</td>
</tr>
<tr>
<td>petitparser</td>
<td>2 065 / 3 280</td>
<td>1 155 / 1 559</td>
</tr>
<tr>
<td>bzip2</td>
<td>1 105 / 2 280</td>
<td>665 / 1 207</td>
</tr>
<tr>
<td>coverage</td>
<td>847 / 3 586</td>
<td>446 / 667</td>
</tr>
<tr>
<td>markdown</td>
<td>697 / 1 846</td>
<td>441 / 701</td>
</tr>
<tr>
<td>crypt</td>
<td>161 / 1 199</td>
<td>118 / 197</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>263 077 / 514 805</strong></td>
<td><strong>112 156 / 155 672</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Experimental results on checking for potential message-not-understood and subtype violation errors in *modular mode*.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Warnings</th>
<th>Percentage</th>
<th>Times</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>dart2js</td>
<td>976</td>
<td>1.62%</td>
<td>433s</td>
<td>1s/270s</td>
</tr>
<tr>
<td>analyzer</td>
<td>306</td>
<td>0.9%</td>
<td>124s</td>
<td>5s/24s</td>
</tr>
<tr>
<td>devcompiler</td>
<td>312</td>
<td>4.2%</td>
<td>840s</td>
<td>8s/247s</td>
</tr>
<tr>
<td>dartstyle</td>
<td>183</td>
<td>5.6%</td>
<td>134s</td>
<td>3s/9s</td>
</tr>
<tr>
<td>linter</td>
<td>26</td>
<td>2.8%</td>
<td>264s</td>
<td>1s/4s</td>
</tr>
<tr>
<td>petitparser</td>
<td>178</td>
<td>15.4%</td>
<td>51s</td>
<td>0s/11s</td>
</tr>
<tr>
<td>bzip2</td>
<td>31</td>
<td>4.6%</td>
<td>37s</td>
<td>0s/1s</td>
</tr>
<tr>
<td>coverage</td>
<td>24</td>
<td>5.4%</td>
<td>45s</td>
<td>0s/0s</td>
</tr>
<tr>
<td>markdown</td>
<td>28</td>
<td>6.4%</td>
<td>35s</td>
<td>0s/0s</td>
</tr>
<tr>
<td>crypt</td>
<td>1</td>
<td>0%</td>
<td>9s</td>
<td>0s/0s</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>2 036</strong></td>
<td><strong>1.8%</strong></td>
<td><strong>1972s</strong></td>
<td><strong>32s/566s</strong></td>
</tr>
</tbody>
</table>

The evaluation is conducted on a range of real-world open source Dart programs, where we exclude programs that use mirrors [8] (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 5.1 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 naïve 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.\footnote{http://www.brics.dk/safedart/}
### 5.10. EVALUATION

#### Table 5.3: Experimental results on checking for potential message-not-understood and subtype violation errors in modular mode with optimistic assumption.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Warnings</th>
<th>Percentage</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MNU</td>
<td>SV</td>
<td>(t_{\text{inf}})</td>
</tr>
<tr>
<td>dart2js</td>
<td>681</td>
<td>5 452</td>
<td>13s/198s</td>
</tr>
<tr>
<td>analyzer</td>
<td>134</td>
<td>1 959</td>
<td>4s/10s</td>
</tr>
<tr>
<td>devcompiler</td>
<td>266</td>
<td>738</td>
<td>6s/104s</td>
</tr>
<tr>
<td>linter</td>
<td>25</td>
<td>75</td>
<td>3s/2s</td>
</tr>
<tr>
<td>petitparser</td>
<td>171</td>
<td>64</td>
<td>1s/10s</td>
</tr>
<tr>
<td>analyzer</td>
<td>67</td>
<td>1 501</td>
<td>0s/0s</td>
</tr>
<tr>
<td>devcompiler</td>
<td>73</td>
<td>460</td>
<td>4s/32s</td>
</tr>
<tr>
<td>dartstyle</td>
<td>10</td>
<td>34</td>
<td>1s/2s</td>
</tr>
<tr>
<td>linter</td>
<td>10</td>
<td>59</td>
<td>1s/2s</td>
</tr>
<tr>
<td>petitparser</td>
<td>168</td>
<td>44</td>
<td>1s/2s</td>
</tr>
<tr>
<td>bazip2</td>
<td>3</td>
<td>16</td>
<td>0s/0s</td>
</tr>
<tr>
<td>coverage</td>
<td>7</td>
<td>54</td>
<td>0s/0s</td>
</tr>
<tr>
<td>markdown</td>
<td>11</td>
<td>9</td>
<td>0s/0s</td>
</tr>
<tr>
<td>crypt</td>
<td>0</td>
<td>0</td>
<td>0s/0s</td>
</tr>
<tr>
<td>total</td>
<td>1 349</td>
<td>8 463</td>
<td>28s/328s</td>
</tr>
</tbody>
</table>

Table 5.4: Experimental results on checking for potential message-not-understood and subtype violation errors in filter mode.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Warnings</th>
<th>Percentage</th>
<th>Times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MNU</td>
<td>SV</td>
<td>(t_{\text{inf}})</td>
</tr>
<tr>
<td>petitparser</td>
<td>249</td>
<td>215.5%</td>
<td>0s</td>
</tr>
<tr>
<td>bazip2</td>
<td>33</td>
<td>5%</td>
<td>0s</td>
</tr>
<tr>
<td>coverage</td>
<td>36</td>
<td>8%</td>
<td>0s</td>
</tr>
<tr>
<td>markdown</td>
<td>44</td>
<td>10%</td>
<td>0s</td>
</tr>
<tr>
<td>crypt</td>
<td>0</td>
<td>0%</td>
<td>0s</td>
</tr>
<tr>
<td>total</td>
<td>318</td>
<td>0.3%</td>
<td>0s</td>
</tr>
</tbody>
</table>

Table 5.5: Experimental results on checking for potential message-not-understood and subtype violation errors in flow mode.

To solve type inference constraints we use a basic fixpoint algorithm with difference propagation \[82\]. 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 \(\{\text{Object}\}\), 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 5.3 to the
full Dart language involves several interesting issues, some of which are crucial for making the algorithms practical. Programs are represented using SSA, which regains some degree of flow sensitivity [38]. 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 [21]. 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.

### 5.10.1 Precision of the Analysis

We first measure the ability of the analysis to check absence of message-not-understood errors. Tables 5.2, 5.4, 5.5, and 5.3 respectively show the experimental results for modular mode, filter mode, flow mode, and modular mode with optimistic assumption. The MNU and SV columns below Warnings shows respectively the number of MNU warnings and the numbers of SV warnings. Similarly, the columns below Percentage shows the respective percentage of warnings compared to the baselines defined in Table 5.1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(\text{MNU}<em>{\text{modular}} - \text{MNU}</em>{\text{filter}}) Warnings</th>
<th>Percentage</th>
<th>(\text{SV}<em>{\text{modular}} \cap \text{SV}</em>{\text{filter}}) Warnings</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dart2js</td>
<td>538</td>
<td>0.9%</td>
<td>4771</td>
<td>5.8%</td>
</tr>
<tr>
<td>analyzer</td>
<td>281</td>
<td>0.8%</td>
<td>1231</td>
<td>2.3%</td>
</tr>
<tr>
<td>decompiler</td>
<td>238</td>
<td>3.2%</td>
<td>382</td>
<td>3.4%</td>
</tr>
<tr>
<td>dartstyle</td>
<td>175</td>
<td>5.3%</td>
<td>30</td>
<td>0.9%</td>
</tr>
<tr>
<td>linter</td>
<td>15</td>
<td>1.6%</td>
<td>57</td>
<td>4.2%</td>
</tr>
<tr>
<td>petitparser</td>
<td>9</td>
<td>0.8%</td>
<td>41</td>
<td>2.6%</td>
</tr>
<tr>
<td>bzip2</td>
<td>28</td>
<td>4.2%</td>
<td>12</td>
<td>1%</td>
</tr>
<tr>
<td>coverage</td>
<td>17</td>
<td>3.8%</td>
<td>49</td>
<td>7.4%</td>
</tr>
<tr>
<td>markdown</td>
<td>24</td>
<td>5.4%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>crypt</td>
<td>1</td>
<td>0.9%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>total</td>
<td>1326</td>
<td>1.2%</td>
<td>6573</td>
<td>4.2%</td>
</tr>
</tbody>
</table>

Table 5.6: Results on the message-not-understood warnings caused by broken type annotations, and subtype-violation warnings obtained intersecting modular and filter mode subtype-violation warnings. The set \(\text{MNU}_{\text{modular}}\) contains all the message-not-understood warnings raised in modular mode for a given benchmark. Similar considerations apply to \(\text{MNU}_{\text{filter}}, \text{SV}_{\text{modular}},\) and \(\text{SV}_{\text{filter}}\).
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 5.9.

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. We therefore performed an experiment where we identify broken type annotations by removing all the message-not-understood warnings raised in filter mode from the set of message-not-understood warnings raised in modular mode. Table 5.6 shows that 1,326 out of 2,036 warnings raised in modular mode are caused by broken type annotations, which amounts to 65% of the message-not-understood warnings.

The corresponding numbers for subtype-violation warnings are 5.1% for filter mode and 10.1% for modular mode. It is beneficial for subtype-violation checking to combine the two modes; table 5.6 shows that this reduces the number of warnings to 4.2%, thus demonstrating that this trick does lead to improved precision in practice.

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. Table 5.5 excludes some of our benchmarks, because flow mode execution is too slow when analyzing large projects. One way to interpret this result is that incorporating type annotations adds significant value compared to merely tracking dataflow.

Tables 5.2, 5.4, 5.5, and 5.3 also show the time for inference (t_{inf}), message-not-understood checking (t_{MNUchk}), subtype violation checking (t_{SVchk}), 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 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.
5.10.2 Optimistic assumptions

The ‘optimistic’ configuration allows the inference phase to trust type annotations, which is particularly relevant in modular mode. As evident from Table 5.3, this configuration has a significant effect on precision compared to modular mode, and it also improves analysis time. The number of message-not-understood warnings is reduced from 1.8% to 1.2%, and the number of subtype-violation warnings is reduced 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.

Typical causes of type warnings

We have manually investigated a selection of the potential message-not-understood and subtype-violation errors reported by our inference, to understand the causes of these errors. Although precisely classifying such errors is difficult, we observe some interesting patterns that may provide opportunities for future improvements of the analysis.

- Generic classes are a typical source of imprecision. Although our type inference is able to track type parameters, it cannot track tricky cases such as the Dart’s async and collection library. However, context sensitivity could recover some precision.

- Another important fraction of errors 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 appear 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 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.
5.11 Related Work

The idea of applying static analysis to check for potential type errors in dynamically typed languages has a long history [2, 10, 27, 32, 46, 69, 76]. Common to most of these techniques is that they do not incorporate type annotations. A notable exception is the algorithm by Siek and Vachharajani [76], however, being based on unification, it does not work in presence of subtyping. The more recent algorithm by Rastogi et al. [69] 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 [27]. The Mypy project [89] 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 our work, 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 [61], and for an extension of Dylan with function types and parametric polymorphism [53].

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 [74]. A recent survey by Siek et al. [79] 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 [78], but the language is gaining momentum in industrial software development unlike e.g. Typed Racket [85].

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 [33, 34]. 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 [55].
Chapter 6
Type Unsoundness in Practice

Chapter 4 shows the sources of unsoundness in the Dart type system and proposes message-safety and full type safety as natural sound alternative. This chapter takes these theoretical results as starting point to tests the third part of the hypothesis stated in Section 1.1, i.e., that programmers take advantage on each source of unsoundness. As observed in Chapter 4, for each source of unsoundness in the Dart 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. This gives us a spectrum between the unsound standard type system formalized in Section 3.3 and the fully sound alternative formalized in Section 4.2. For each category we may choose either the unsound or the sound variant. The hypothesis that we test in this chapter is the following:

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 legitamate 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 of those sources of unsoundness, which may be useful information when developing future versions of the language, or for design of new languages. The focus of this chapter 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). We attempt to provide a posteriori empirical justifi-
This chapter presents the following contributions:

- 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 the theoretical results of Chapter 4 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 [74, 79].

- 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 programmer mistakes.

This chapter is extracted from [56]. The technical content of the paper is extended with additional experimental results shown in Figure 6.3. Additionally, Section 6.1 has been shortened removing the concepts of Dart introduced in Section 2.2 while still keeping the concepts specific to this chapter. Part of the experiments in this chapter are similar to the experiments performed in Section 4.10. However, the former aims to investigate when programmers take advantage of the sound feature, whereas the latter serves to support message-safety.
6.1 Types and Dart

This section introduces some terminology that will be used in the remaining parts of this chapter. Using the definitions for Fletch, introduced in Chapter 3, we write $T <: S$ if $T$ is a subtype of $S$, and $\text{assignable}(T, S)$ is an abbreviation of $T <: S \lor (S <: T \land (\neg \text{isfun}(T) \lor \neg \text{isfun}(S)))$. As described in Section 2.2, two kinds of runtime type errors can occur: 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 chapter, we also consider a third kind of error: a type-mismatch (TM) 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. Absence of SV errors does not imply absence of TM errors.

**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 [9]. 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 TM errors are generally untrapped (but some are prevented by the SV checks). For example, the following program erroneously passes a number to a variable of type String:

```dart
613 String x = 5;
614 print(x.length);
```

In checked mode, the error causes the program to stop in line 613 due to the SV check, while in production mode, the error is unnoticed until line 614 where it causes the program to stop with an MNU error. Untrapped errors are often more problematic than trapped ones because of the potentially large distance between where the errors occur and where they are detected. For the remainder of this chapter, we focus on checked mode, and we shall later see examples of TM errors.

The purpose of a type system is to check statically whether a given category of errors, called forbidden errors, can occur at runtime [9]. 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 chapter, unless otherwise noted we select MNU and SV as the forbidden errors, while still allowing TM 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 type checker warning 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 may 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 in 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. [79] 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:

1. A well-typed fully annotated program (not containing dynamic) cannot encounter MNU nor SV error 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 TM 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 of the criteria for gradually typed soundness mentioned above, for reasons explained in the following section.

---

1MNU and TM correspond to TypeError in Siek et al. [79], and SV corresponds to CastError; moreover, Siek et al. express the criteria slightly differently via a blame tracking mechanism, which Dart does not have.

2One could choose to also treat TM errors as forbidden, in which case TM should be included here alongside MNU and SV errors.
6.2 Sources of Unsoundness in Dart

In Chapter 4, we 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. [25] 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 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, for each variant of the type system we discuss, the same subtype relation is used for type checking and at runtime.

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

- Type dynamic (optional typing)
  - at lookups (DYNL)
  - with ground types (DYNG)
  - with generic types (DYNP)
  - with function types (DYNF)
- Symmetric assignability (SYA)
- Covariant generics (COG)
- Function subtyping
  - at parameter types (CFP)
  - at return types (CFR)
- Method overriding (including field overriding)
  - at parameter types (CFPM)
  - at return types (CFRM)

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

6.2.1 Unsoundness Caused by Optional Typing

We distinguish between four sources of unsoundness concerning the optional typing mechanism (and hence the type dynamic). In comparison, message-safety simply forbids dynamic altogether (see Chapter 4), which is unnecessarily strict.

**Type dynamic at lookups (DYNL)** Quoting the Dart language specification [21] 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 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.

```dart
615 var x = 5;
616 x.method();
```

This is clearly unsound as an MNU error appears at line 616.

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 (DYNG)** 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 DYNG 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 DYNP and DYNF. Such assignments can obviously break soundness, as shown by the following example.

```dart
617 var x = 5;
618 num y = x;
619 String z = x;
620 print(z + z);
621 z.substring(0, 5);
```

The variable $x$ has static type dynamic. The assignments in line 618 and 619 and the invocation in line 621 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 618 and 619. The first runs fine, because int <: num, but an SV error occurs in line 619. In production mode, no subtype checks are performed, but an MNU error occurs at line 621. Note that the operator + is present in both the int and String classes (operators are methods in Dart), so line 620 gives no error and 10 is printed.

If optional typing is used for ground types only, then in checked-mode execution all the errors at the boundary between untyped and typed code will be trapped, fulfilling criterion 2 for gradually typed soundness (see Section 6.1).

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 DYNG unsoundness requires two modifications: we simply disallow those two cases when $T$ is a ground type. However, to reduce overlap with DYNF this change does not apply when assignability is used for checking subtyping of function types.
Type dynamic with generic types (DYNP) 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 DYNP 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 DYNP may lead to unsoundness.

```dart
622 List<dynamic> y = new List<dynamic>();
623 y.add(true);
624 List<int> x = y;
625 x.first.toDouble();
```
The program is well-typed but fails with an MNU error at line 625 because toDouble is not defined for booleans. The subtype check at line 624 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 a TM error: reading \( x.first \) in line 625 yields a boolean, which does not match the declared type int.

Notice that the DYNP category of unsoundness is different from DYNG. The subtype check performed at line 624 (in checked mode) is insufficient for catching the error, making it untrapped and unnoticed until the lookup of toDouble in line 625 where it causes the MNU error. In contrast, DYNG can only result in trapped errors. The DYNP category therefore violates criterion 2 for gradually typed soundness.

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

Type dynamic with function types (DYNF) As for generic types, function types can contain dynamic. The DYNF category is exactly like DYNP except using function types instead of generic types. For example, DYNF comprises assignments \( x = e \) where \( x \) has type int\( \rightarrow \)int and \( e \) has type dynamic or int\( \rightarrow \)dynamic. (Only the parts in positive position in the function types are considered, so we do not include the case where \( e \) has type dynamic\( \rightarrow \)int, for example.) Those cases are allowed by the type checker but may cause unsoundness, as the following example shows.

```dart
typedef int intmap(int
626 dynamic intmapimpl(int x) {
627 return true;
628 }
629 } intmap x = intmapimpl;
630 x(5).toDouble();
```
The program is well-typed but fails with an MNU error at line 631 in both production mode and checked mode. The static type of `intmapimpl` in line 630 is `int → dynamic`, and `x` has type `int → int`. Similar to the `DYNP` example, the checked-mode runtime type checks are not strong enough to trap this mismatch early at line 630.

To make the situation even worse, according to the language specifications, it is not granted that the declared type of `intmapimpl` is reified as `int → dynamic`, which one might expect [21, Section 9.3]: “It is up to the implementation to choose an appropriate representation for functions. Implementations may also use different classes for functions based on arity and or type. ... this specification only guarantees that function objects are instances of some class that is considered to implement `Function`.” A consequence of this partial type erasure is that function type errors may not only be undetected by the type checker; they may also remain untrapped at runtime.

To fix the `DYNF` category of unsoundness, we modify subtyping and assignability in the same way as for `DYNG` and `DYNP`, except this time for function subtyping.

### 6.2.2 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 6.1).

**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 `assignable(T, S)`.

The following program shows why this is unsound.

```dart
632 int x = new Object();
633 x + 5;
```

The type checker raises no warnings. In checked mode, an SV error appears at line 632. In production mode, reading `x` in line 633 gives a TM 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 (COG)** 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 type specificity rather than subtyping). Consider the following class hierarchy where `B <: A` due to the inheritance.

```dart
634 class A { }
```
6.2. SOURCES OF UNSOUNDNESS IN DART

The following code is now well-typed.

```dart
635 class B extends A {
636   void m() {}
637 }
```

The following code is now well-typed.

```dart
638 List<A> x = new List<B>();
639 x.add(new A());
640 x.first.m();
```

The assignment on line 638 is allowed statically because of the covariant generics rule. In checked mode execution an SV error appears at line 639 and in production mode the execution proceeds to line 640 where an MNU error appears. Notice that with checked mode, COG unsoundness cannot introduce untrapped errors (i.e., TM errors), in contrast to DYNP and DYNF unsoundness.

The traditional sound choice for generics, if variance annotations are not used, is to require invariance, that is, $T<E> <: T<K>$ if $E = K$ (see Chapter 4). To reduce the overlap with DYNP we choose a slightly more fine-grained alternative that also allows $E = dynamic$. In this way, our sound choice for COG will reject `List<A> x = new List<B>()` but allow `List<A> x = new List<dynamic>()`.

**Function subtyping (CFP and CFR)** 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 $T \rightarrow S$ is a subtype of another one $T' \rightarrow S'$ if $\text{assignable}(T, T')$ and $\text{assignable}(S, S')$ (or $S = void$).

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

```dart
641 typedef int fun(Object x);
642 Object funimpl(int x) {
643   return x.isOdd;
644 }
645 fun x = funimpl;
```

The program is well-typed, and the assignment in line 645 passes the subtype check in checked-mode execution. Because of CFR, the following invocation is well-typed, but it encounters a TM 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.

```dart
646 x(5).toDouble();
```

Likewise, because of CFP 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.
CHAPTER 6. TYPE UNSOUNDNESS IN PRACTICE

647 \texttt{x(true);} \\

The natural way to fix these kinds of unsoundness is to require contravariant parameter types (for \texttt{CFP}) and covariant return types (for \texttt{CFR}) in function subtyping.

Method overriding (\texttt{CFPM and CFRM}) The type rule for overriding of methods in subclasses uses bivariance, in the same way as function subtyping; if a method \texttt{m} has type \( T \rightarrow S \) in a class \texttt{A} and \texttt{m} is overridden with type \( T' \rightarrow S' \) in a sub-class \texttt{B} of \texttt{A}, it is required that \texttt{assignable}(\( T, T' \)) and \texttt{assignable}(\( S, S' \)). Since fields are implicitly 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 \texttt{CFPM} and \texttt{CFRM}, respectively. Examples corresponding to lines 641–647 can therefore be made using method overriding:

\begin{verbatim}
648 class A {
649     int m(Object x) => 0;
650 }
651 class B extends A {
652     Object m(int x) => x.isOdd;
653 }
654 A s = new B();
655 s.m(5).toDouble();
656 s.m(true);
\end{verbatim}

Line 655 demonstrates \texttt{CFRM}, similar to line 646 for \texttt{CFR}, and line 656 demonstrates \texttt{CFPM} (assuming the preceding line is removed), similar to line 647 for \texttt{CFP}. To fix these kinds of unsoundness, we accordingly restrict the type rule for method overriding to contravariant parameter types (for \texttt{CFPM}) and covariant return types (for \texttt{CFRM}).

6.2.3 Discussion

Figure 6.1 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 TM errors are untrapped, possibility of TM implies possibility of MNU and SV. The figure also shows which fixes affect the subtyping relation (and thereby also the meaning of SV and TM and the runtime type checks performed in checked-mode execution\footnote{Fixing \texttt{DYNG} does affect subtyping, but not runtime execution since no object has type \texttt{dynamic}.}), 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 sug-
6.2. SOURCES OF UNSOUNDNESS IN DART

Figure 6.1: Sources of unsoundness. A \(\bullet\) 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 \(\star\) indicates that subtyping is affected (and thereby also the meaning of SV and TM) 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).

<table>
<thead>
<tr>
<th>category</th>
<th>MNU</th>
<th>SV</th>
<th>TM</th>
<th>&lt;:</th>
<th>type checker</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNL</td>
<td>•</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>54</td>
<td>-</td>
</tr>
<tr>
<td>DYNG</td>
<td>-</td>
<td>•</td>
<td>-</td>
<td>((\star))</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>DYNP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>DYNF</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>28</td>
<td>13</td>
</tr>
<tr>
<td>SYA</td>
<td>-</td>
<td>•</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>COG</td>
<td>-</td>
<td>•</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>CFR</td>
<td>-</td>
<td>•</td>
<td>-</td>
<td>•</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>CFPM</td>
<td>•</td>
<td>•</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>CFRM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>-</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>total LOC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>121</td>
<td>42</td>
</tr>
</tbody>
</table>

It follows from Figure 6.1 that some configurations have 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 6.1).

**message safety** We obtain the notion of message safety (see Chapter 4), meaning that well-typed programs cannot have MNU errors, if we apply the fixes for DYNL, DYNP, DYNF, CFR, and CFRM.

**gradual safety** We can satisfy the two criteria for gradually typed soundness from Section 2.2 by applying the fixes for all categories of unsoundness, except DYNG. The resulting type system will hence only allow optional typing for ground types.

**TM safety** Absence of TM errors can be guaranteed by the type checker if we apply the fixes for (only) DYNP, DYNF, CFR, and CFRM. This is interesting because TM 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.

---

\(\text{dynamic}\) as an example, the following code is well-typed in our sound type system, even though it uses \text{dynamic}: dynamic x = "string"; (x as String).substring(0, 3);
CHAPTER 6. TYPE UNSOUNDNESS IN PRACTICE

6.3 Experiments

In order to test our hypothesis we have downloaded the latest version of every library and application hosted by the Dart ‘pub’ repository, consisting of 2093 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 1883 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 $\{\text{COG, SYA}\}$ 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 6.1, 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 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 warning 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.

We report only highlights of the experiments. Our entire dataset, including details about warnings and errors in different type system configurations, is available online.

6.3.1 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 6.2 shows the resulting distributions for all configurations of size 1 and for TM safety $\text{TM} = \{\text{DYNP, DYNF, CFR, CFRM}\}$ and “co-/contravariance” $\text{CC} = \{\text{CFP, CFR, CFPM, CFRM}\}$; we later report on the results for configurations that involve other combinations of categories. We also measure

\footnote{https://pub.dartlang.org/ (May 2nd 2016)}
\footnote{http://www.brics.dk/undart/}
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 6.2 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 6.2 for $\{\text{DYNL}\}$, $\{\text{DYNG}\}$, and $\{\text{DYNP}\}$. It is even a recommended programming style to omit type annotations at local variables, which obviously results in many warnings, especially for DYNL and DYNG. For someone with little experience with Dart, probably the most surprising numbers are those for $\{\text{DYNP}\}$. More than 50% of the benchmarks contain more than 21 (and in some cases beyond 100) occurrences of DYNP per KLOC. Section 6.3.2 provides a possible explanation. The numbers for $\{\text{DYNF}\}$ 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 $\{\text{CFP}\}$, $\{\text{CFR}\}$, $\{\text{CFPM}\}$, and $\{\text{CFRM}\}$ are more

https://www.dartlang.org/effective-dart/usage/
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 \{SYA\} and \{COG\} not conclusive. These configurations give 2.4 and 5.5 warnings/KLOC and significantly affect 9% and 18% of the benchmarks, respectively.

Our observations so far are for size 1 configurations, however, there are cases where combinations of fixes 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 \(\emptyset\), \{SYA\}, or \{COG\}, but a warning appears with \{SYA, COG\}. Our fix to COG 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 cc 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 cc can be attributed to such effects. Since each of CFP, CFR, CFPM, and CFRM can be fixed individually without leading to a significant increase in warnings, it is interesting to notice that the combination cc also has this property: the cc configuration results in more than 5 new warnings compared to Dart’s standard type system at only 2.8% of the benchmarks. Similarly, the TM configuration is interesting for the reason explained in Section 6.2.3. However, due to the large number of warnings that result from our fix for DYNP unsoundness, TM affects many benchmarks significantly. Figure 6.3 shows the resulting distributions for the configurations with the best average number of warnings per KLOC. The result confirms that \{CFR\}, \{CFRM\}, \{CFP\}, and \{CFPM\} are among the features causing the lowest average number of warnings per benchmark.

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 configuration 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 6.3.2). 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 6.3.3).
6.3. EXPERIMENTS

6.3.2 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.

- **A good** warning is one that indicates a programmer mistake that is easy to fix by only changing a few type annotations. (As most of our benchmark programs are presumably thoroughly tested already, most likely when such warnings appear in our benchmark programs, they do not indicate serious bugs but 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.\footnote{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.} 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.
CHAPTER 6. TYPE UNSOUNDNESS IN PRACTICE

Table 6.1: Classification of warnings.

<table>
<thead>
<tr>
<th>origin</th>
<th>good</th>
<th>bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>{CFP}</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>{CFR}</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>{CFPM}</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>{CFRM}</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>{COG}</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>{SYA}</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>{DYNP}</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{COG,FYNG}</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>{COG,CFPM}</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>{COG,SYA}</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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 the origin of the warning. Note that a warning can have multiple origins.

Table 6.1 summarizes the results. The listed 11 origins account for all the 51 warnings. For example, we find that all 9 warnings examined from the \{CFRM\} 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 from the super-class:

```java
1 class Displ implements Display {
2     num get viewOffset;
3 ...
4 }
```

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 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 \{DYNP\}. Such warnings could clearly be avoided with a slightly less naive language design, without sacrificing soundness. The signature of the List.map
method is \texttt{Iterable\textlessdynamic\textgreater map(dynamic f(E e))} due to the lack of generic methods. This has the unfortunate consequence that \texttt{dynamic} appears frequently as type parameter, making it practically impossible to write Dart programs that do not use \texttt{dynamic}. We also see examples where generic methods could avoid warnings related to \texttt{CFR}.

Another interesting type system artifact is that \{\texttt{COG}\} forbids the use of asynchronous expressions, as the following example shows.

5 \texttt{Future\textlt String\textgt generate(Element element) async {}
6 \quad if (element is! LibraryElement)
7 \quad return null;
8 \quad ...}
9 }

Asynchronous expressions and the type \texttt{Future\lt T\textgt} are treated specially by Dart’s type system. If an expression \(e\) of type \(T\) is returned from an asynchronous function, the type system assigns the type \texttt{Future\lt T\textgt} to \(e\). In the example above, \texttt{Future\lt Null\textgt} requires \texttt{COG} unsoundness to match the \texttt{generate} function return type, which is \texttt{Future\lt String\textgt}, leading to a warning at line 7 for the configuration \{\texttt{COG}\}. However, since type \texttt{Future\lt T\textgt} already has a special meaning in the type system, and futures with value \texttt{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 has been designed with symmetric assignability in mind. For example, it is common to write \texttt{DivElement d = new Element.tag("div")} knowing that the constructor returns an object of type \texttt{DivElement} although its static return type is the super-type \texttt{Element}. However, it would be easy for the programmer to instead create the object by \texttt{new DivElement()} (or use an explicit type cast) and thereby not rely on \texttt{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.

### 6.3.3 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 5 relevant sources of unsoundness according to Figure \ref{fig:unsoundness}: \texttt{DYNP}, \texttt{DYNF}, \texttt{COG}, \texttt{CFP}, and \texttt{CFR}.

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 contain multiple test procedures.) This gives us a total of 1,032 succeeding test files among 390 of the benchmark projects.

<table>
<thead>
<tr>
<th>configuration</th>
<th>affected</th>
<th>configuration</th>
<th>affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>{DYNP}</td>
<td>67.3% (69.2%)</td>
<td>{CFP}</td>
<td>1.6% (3.3%)</td>
</tr>
<tr>
<td>{DYNF}</td>
<td>50.9% (50.1%)</td>
<td>{CFR}</td>
<td>0.7% (1.8%)</td>
</tr>
<tr>
<td>{COG}</td>
<td>81.3% (77.7%)</td>
<td>{CFP, CFR}</td>
<td>1.8% (4.1%)</td>
</tr>
</tbody>
</table>

Table 6.2: Percentage of tests affected in different configurations (projects affected are shown in parentheses).

Table 6.2 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 cc configuration (which is equivalent to \{CFP, CFR\} 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 DYNP, DYNF, and COG, the numbers of affected tests are much higher. The number for DYNP reinforces the observation in Section 6.3.2 that with the current design of lists, maps, and sets in the standard library, most such objects inevitably hold a \texttt{dynamic} type argument. This also shows that in many assignments of the kind \texttt{C<T> x = y}, the type of y is \texttt{C<dynamic>}, which means that the T annotation is completely superfluous from a type safety perspective.

### 6.4 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 [83] “seriously question the scientific validity of any claim made by a language designer that does not have actual data in hand.” According to Markstrum [52], “We should be more aware of valiant and effective efforts for supplying evidence to support language design claims.” Murphy-Hill and Grossman [60] 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.
6.4. RELATED WORK

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 [68].

Souza and Figueiredo [81] have studied Groovy projects to investigate how programmers use optional typing, and Eshkevari et al. [26] 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. [63]. 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 [41].

Several controlled experiments have been conducted to study the potential benefits of static typing, most notably by Hanenberg et al. [22, 36]. We are aware of only one such study involving Dart: Faldborg and Nielsen [28] 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 2.1, 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 [70, 85, 93]. 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 [55]. This goes considerably further than the minor adjustments we have explored in this chapter, 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
Unsoundness in static analysis  Deliberate unsoundness is rare in type systems, but less so in static analyzers [49]. Recently, Christakis et al. [12] 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.
Chapter 7
Conclusions

In order to test the first part of the hypothesis stated in Section 1.1, we have introduced Fletch as a core of the Dart programming language to expose the central aspects of its type system (Chapter 3). Moreover, we have proposed the notion of message-safe programs as a natural intermediate point between dynamically typed and statically typed Dart programs (Chapter 4). Based on Fletch we have expressed appropriate progress and preservation lemmas and a type soundness theorem, which demonstrates the fundamental property that message-safe programs never encounter ‘message-not-understood’ errors. This result provides new insights into the design space between dynamic and static typing. At this point, the theoretical foundation of message safety has been established. Message safety can be obtained for Dart code under two assumptions: 1) the code is fully-annotated, and 2) we need to restrict the Dart subtype relation in both the static and dynamic type systems. Although the first assumption invalidates message-safety for real-world Dart programs, which are typically not fully-annotated, message-safety can be ensured for fully-annotated program fragments, which are present in Dart code. Moreover, the second assumption is justified by our preliminary experimental results in Section 4.10 and the change required in the runtime subtyping relation has been confirmed as a design flaw by the Dart team. Therefore, we can conclude that the two assumptions required to get message-safety are reasonable. Moreover, we believe Fletch and our formalization may be useful in further studies of Dart and related programming languages.

The second part of the hypothesis stated in Section 1.1 is confirmed by the type safety analysis presented in Chapter 5. More specifically, we have demonstrated that it is possible for a realistic programming language with optional typing to incorporate type annotations into a flow analysis to provide static type checking and thereby increase confidence in program correctness before the programs are put to use. 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 5.8 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.

According to the third part of the hypothesis stated in Section 1.1, we expect that programmers take advantage of each source of unsoundness in the Dart type system. We found out that some of the sources of unsoundness in the Dart type system are of little use, thereby contradicting the third part of the hypothesis. However, 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 for full Dart programs\footnote{Despite message safety cannot be easily obtained for full Dart programs, there still exists real-world Dart program fragments that are message-safe.}, or even gradual safety (as defined in Section 6.2.3) 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 and method overriding 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 pro-
ductivity, 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.
Bibliography


[40] Thomas S. Heinze, Anders Møller, and Fabio Strocco. Type safety analysis for Dart. In *Proc. 12th Symposium on Dynamic Languages (DLS)*, 2016. 7, 8, 95


[64] Personal communication. In relation to a Google Faculty Awards research effort, involving Gilad Bracha and the Google Team in Aarhus, Denmark, 2013–2014. 89

[65] Personal communication. In relation to the the discussion about strongmode on the Dart forum, involving Gilad Bracha and Bob Nystrom, 2016. URL https://groups.google.com/a/dartlang.org/forum/#!searchin/analyzer-discuss/strong/analyzer-discuss/yqjrj17mSAM/YHlcJRIkYCAJ 25

[66] Benjamin C. Pierce. Types and Programming Languages. MIT Press, 2002. 8, 9, 79, 81


[72] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. The Definition of Standard ML, Revised, 1977. 1


