

Message Safety in Dart

Erik Ernst¹ Anders Møller² Mathias Schwarz^{2*} Fabio Strocchio²
Google, Aarhus, Denmark¹ Aarhus University, Denmark²
ernst@google.com {amoeller,schwarz,fstrocco}@cs.au.dk

Abstract

Unlike traditional static type checking, the type system in the Dart programming language is unsound by design, even for fully annotated programs. The rationale has been that this allows compile-time detection of likely errors and enables code completion in integrated development environments, without being restrictive on programmers.

Despite unsoundness, judicious use of type annotations can ensure useful properties of the runtime behavior of Dart programs. We present a formal model of a core of Dart with a focus on its type system, which allows us to elucidate the causes of unsoundness. Our main contribution is a characterization of *message-safe programs* and a theorem stating that such programs will never encounter ‘message not understood’ errors at runtime. Message safety is less restrictive than traditional type soundness, and we argue that it forms a natural intermediate point between dynamically typed and statically typed Dart programs.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

Keywords type systems; optional types

1. Introduction

Most mainstream object-oriented languages are statically typed, with soundness properties ensuring that certain errors cannot occur at runtime. It is also well known that dynamically typed languages without type annotations can offer great flexibility at the cost of potential type related errors at runtime. Many intermediate levels have been proposed and studied, e.g. [2, 3, 5, 11, 14, 17–19, 22, 23]. The Dart programming language [9] strikes an interesting new balance, with a type system far less restrictive than required for tradi-

tional soundness. Dart permits programmers to provide type annotations selectively and thereby decide which parts of the program should be statically type checked. The type system is not sound in the traditional sense: even for fully annotated programs, the static type checker may miss some type-related errors. Hence, two kinds of such errors may appear at runtime: ‘message not understood’ if an object property lookup operation fails, and ‘subtype violation’ if a value assigned to a variable or field does not match the declared type.

By separating these two kinds of errors and slightly restricting Dart’s type system, we show that there exists a natural intermediate point between the existing type checking in Dart and a traditional sound type system. In a *message-safe program*, which passes this modified type system, ‘message not understood’ runtime errors cannot occur, but ‘subtype violation’ errors are still possible. We argue that message safety is a valuable step when evolving programs from dynamic to static typing, because it establishes a firm connection from each lookup to the corresponding declaration.

To show that message-safe programs can be defined precisely and that they have the desired properties, we have created a formal model of a core of the Dart programming language, in the style of Featherweight Java [13] and based on the most recent language specification [9].

The contributions of this paper are as follows:

- We present a core calculus of Dart called *Fletch*, thereby elucidating the type system of the Dart language and the causes of unsoundness.
- 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 formalize *Fletch* and a soundness theorem stating that message-safe programs do not cause ‘message not understood’ errors in checked mode execution.
- As part of the formalization 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

* Current affiliation: CLC Bio, Qiagen Aarhus.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DLS ’15, October 25–30, 2015, Pittsburgh, PA, USA.
Copyright © 2015 ACM 978-1-4503-3690-1/15/10...\$15.00.
<http://dx.doi.org/10.1145/10.1145/2816707.2816711>

easily be fixed. We additionally report on initial experimental results that support the use of message safety in software development.

The paper is organized as follows. Section 2 presents our conceptual analysis, defines the notion of message-safe programs, and describes a practical approach to manage gradual typing using message safety. Next, Section 3 introduces our formalization of a core of Dart, and Section 4 presents the soundness results. Our practical experiments are briefly described in Section 5. Section 6 discusses related work and Section 7 concludes. A small-step operational semantics for Fletch, additional motivating examples for message safety, and more details of our experiments are provided in a technical report [10].

2. Analysis and Background

This section briefly describes the Dart language with a focus on the type system design. We then define message-safe programs and outline a two-step approach to structure transformations from untyped to typed programs.

2.1 The Dart Programming Language

The Dart language is a recently introduced object-oriented programming language that shares many traits with Java [12] and C# [7], and others with JavaScript [8]. Although the Dart language is primarily aimed at web programming, it is a general purpose language, and our results are applicable independently of any application domain. The language is class based, and objects do not change class nor add or remove members during their lifetime. This positions the language near the Java style of mainstream object-orientation. The family resemblance is also strong in the syntax, and with many other details.

However, a fundamental difference is that type annotations are optional in Dart programs, and the dynamic semantics of the language is independent of the type annotations. This creates strong connections to many dynamically typed languages, e.g., JavaScript. That connection is underlined by the fact that one of the main techniques to execute Dart programs is via translation into JavaScript. A native Dart virtual machine is also available [20].

The Dart language offers a useful trade-off between dynamic and static typing, and its type system deserves a more in-depth discussion.

2.2 The Dart Approach to Typing

We consider two kinds of type-related runtime errors:

- A *‘message not understood’* error may occur at object property (field or method) lookup operations; for example, at $x.p$ if the object x does not have the specified property p , or at $x.m(y, z)$ if $x.m$ does not resolve to a closure, or the number of arguments is wrong. Technically, this is a `NoSuchMethodError` exception.

- A *‘subtype violation’* error may occur at assignments, parameter passing, and return expressions (i.e., the operations with an associated dataflow); for example, at $x = y$ if the runtime type of y is not a subtype of the declared type of x . Technically, this is a `TypeError` exception.

Dart typing involves the dynamic semantics, which has two modes of execution. *Production mode* execution proceeds without any use of type annotations. It will never fail due to a ‘subtype violation’ error, but it might fail due to a ‘message not understood’ error.

Checked mode execution includes subtyping tests at assignments, parameter passing, and return operations at runtime to detect ‘subtype violation’ errors. Checked mode can also have ‘message not understood’ errors, and both modes can of course have other errors, e.g., divide by zero. The idea is that checked mode execution may be used by programmers during development to catch type-related errors as with static typing, whereas production mode will continue to execute if at all possible.

There is a significant difference between sound static typing and the level of type checking that the standard Dart type system employs. As mentioned, Dart type checking is so permissive that it allows for many programs that cause runtime type errors. This is the consequence of a conscious trade-off by the language designers [16]: Sound type systems require programmers to handle a large amount of complexity in order to enable a sufficiently expressive style of programming. Conversely, a type system that is not sound can be simpler and more flexible. In general, the Dart type system detects obviously wrong typing situations instead of guaranteeing type correctness, which makes it somewhat similar to success typing [14]. However, the lack of soundness does not make type declarations less useful for other purposes. In particular, types can be very helpful in making the programmer’s expectations and intentions explicit, thus enabling type sensitive lookup and completion features in integrated development environments (IDEs).

There are several causes of type unsoundness in Dart. First, initialization, assignment, and argument passing must satisfy an *assignability* check rather than a subtype check; the difference is that both subtypes and supertypes are allowed, but unrelated types are rejected. This means that the type system accepts code that might work, but rejects code that will definitely not work (in checked mode—it might still work in production mode). This is of course not sound, but it does single out the cases where the types are obviously wrong and hence require attention. Second, generic types are considered covariant (e.g., `List<Car>` is a subtype of `List<Vehicle>` iff `Car` is a subtype of `Vehicle`). This is not sound, but the trade-off is useful and meaningful, as known from arrays in Java. Third, function types require only assignability for the argument types and for the return type, rather than the usual sound scheme where argument types are contravariant and return types covariant. Similarly,

method overriding only requires assignability for argument and return types.

A fact worth noting is that assignability 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 T_1 . 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, and we use it directly in our treatment of soundness (Section 3.4).

2.3 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 warnings¹ using the standard Dart type checker with the following modifications:

¹The notion of a *static type warning* in the Dart specification corresponds to a type error in most other languages; the point is that even though the type check fails, it is possible to run the program.

- (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). Some small example programs that motivate these requirements are provided in the technical report [10]. Clearly, it is not hard to implement a checker that decides for any given Dart program whether it is message safe.²

2.4 Full Type Safety

Although we focus on message safety in this paper, *full type safety*, where neither ‘message not understood’ nor ‘subtype violation’ errors are possible, can be ensured statically by the following additional requirements:

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. Method overriding requires contravariant argument types, and field overriding must be invariant.
6. Function subtyping requires contravariant argument types.

We formalize these modifications in Section 3.6.

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.

2.5 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., $x.f$)

²Our implementation (<http://www.brics.dk/fletch/>) required modifying less than 200 LOC in Google’s *dartanalyzer* tool.

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.

2.6 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 on the form $x.f$ or $x.m(\dots)$ 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 `this` object we check that its declared type includes the requested property. Similarly, for an access expression applied to a returned value, e.g., $x.m(\dots).f$ or $x.m(\dots).n(\dots)$, 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 2.3 to be satisfied for the entire program. This aligns well with the concept of gradual typing that is a cornerstone of the Dart design.

2.7 A Two-Step Approach Toward Type Safety

The Dart language specification [9, 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 paper. 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:

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

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):

```
class Account {
  int balance = 0;
```

```

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

```

class Account {
    int balance = 0;
    int withdraw(int amount) {
        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.

3. Fletch

Fletch³ is a calculus that aims to capture the essence of the Dart language, including the interaction between types and checked mode execution. Fletch includes just enough elements from Dart to faithfully characterize the core of the Dart type system and the associated dynamic semantics.

We first specify two distinct type systems for Fletch: the *standard type system*, which faithfully models the core of the Dart type system, elucidating how Dart typing works; and the *message-safe type system*, which embodies the additional constraints required for making programs message safe. The type systems are so similar that we specify them using a single set of type rules; highlighted elements in the type rules should then be omitted or included as described in captions. The message-safe type system is the main type system that is used in the soundness proof, and the standard type system is used to show how that relates to Dart. In Section 3.6 we briefly describe a third variant that ensures full type safety.

The calculus supports ‘typeless’ programs: put `dynamic` in all locations where a type is required. It also supports message-safe programs: the message-safe type system enforces programs with no occurrences of `dynamic` to be message safe, i.e., it embeds the requirements from Section 2.3.

As Fletch is designed to model a core of Dart, similar in spirit to how Featherweight Java [13] models the full Java language, it is practically infeasible to guarantee that Fletch precisely follows the semantics of Dart using testing or formal verification. Instead, we base our confidence on the fact that the description has been checked by central members of the Dart team at Google.

Like in other calculi, many features have been omitted, e.g., general statements. Notable omissions are conditional expressions ($b ? e_1 : e_2$) and type tests ($e \text{ is } T$), which are relevant for the Dart type system, but not essential when studying the idea of message safety. Apart from a couple of trivial syntactic abbreviations and some extensions needed to describe runtime states, Fletch is a syntactic and semantic subset of the Dart language, such that Fletch programs can easily be adapted to become Dart programs.

3.1 Syntax

Dart is an imperative language with classes, whose syntax builds on the family of languages that includes Java, C++ and C#. Figure 1 shows the syntax of Fletch. The declaration categories CL , M , and F define classes, methods, and fields, and they are unsurprising. As usual, \bar{a} denotes the possibly empty list $a_1, \dots, a_n, n \geq 0$.

³The first version of our Fletch calculus was published before the unrelated Google experimental runtime system of the same name.

$$\begin{aligned}
CL &::= \text{class } c \langle \overline{X} \triangleleft \overline{N} \rangle \text{ extends } N \{ \overline{F} \overline{M} \} \\
F &::= G f; \\
M &::= T m(\overline{G} x) \{ \text{return } e; \} \\
e &::= y \mid e.p \mid e.p = e \mid x = e \mid e(\overline{e}) \mid \\
&\quad \text{new } N() \mid fn \mid \boxed{\llbracket T, e \rrbracket \mid \tau = e \mid l} \\
T &::= G \mid \text{void} \\
G &::= X \mid N \mid \text{dynamic} \mid \perp \mid (\overline{G}) \rightarrow T \\
N &::= c \langle \overline{G} \rangle \\
fn &::= T(\overline{G} x) \Rightarrow e \\
y &::= x \mid \text{this} \mid \text{null} \\
p &::= f \mid m \\
l &::= \boxed{\iota \mid \tau}
\end{aligned}$$

Figure 1. Fletch syntax. Boxed parts occur only at runtime.

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 (τ) or heap locations (ι), 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, e, x, X, τ , and ι 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 $\llbracket T, e \rrbracket$ 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 preprocessing 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 (‘ \mapsto ’ indicates 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-*

$$\begin{aligned}
o &: \text{Obj} = G \times \text{Fields} \times \text{Methods} \\
\phi &: \text{Fields} = f \mapsto G \times \iota \\
\mu &: \text{Methods} = m \mapsto \iota \\
\sigma &: \text{Heap} = \text{LocH} \mapsto \text{Obj} \cup fn \\
\nu &: \text{VarEnv} = \text{LocV} \mapsto G \times \text{LocH} \\
s &: \text{State} = \text{VarEnv} \times \text{Heap} \times e
\end{aligned}$$

Figure 2. Semantic entities.

formed iff CT is well-formed and both e and all expressions within all classes in CT contain only well-formed types (see Section 3.5) and identifiers that are defined in the relevant environment.

3.2 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 σ , and the indirection for variables by the maps denoted by ν . 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 2. We use the word *heap* to designate the former, *variable environment* to designate the latter, and *environment* to designate any of the two. LocH and LocV are disjoint, countably infinite sets.

A good intuition about ν 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 τ , 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 τ_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 τ_1 , resp. τ_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 ν 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 1.

We also introduce objects, closures, field maps, and method maps. An object o contains its runtime type G , a map ϕ from field names to declared types and heap locations, and a map μ 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 τ_{this} .

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

The state of a Fletch program during execution is represented by s (see Figure 2). The class table, CT, is frequently consulted during execution. It is constant throughout any program execution so we will leave it implicit, as is common in object calculi since Featherweight Java [13].

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

3.3 Dynamic Semantics

We specify the dynamic semantics of Fletch in terms of a small-step operational semantics \longrightarrow that relates States to States, that is, each configuration is a triple $\langle \nu, \sigma, e \rangle$. The rules for expression evaluation in Fletch are shown in the technical report [10]. Every terminating expression evaluates to a heap location ι , 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 [9]. 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 `new C().f` will call the automatically generated getter

$$\begin{array}{c}
\Delta \vdash \perp \ll T \quad \Delta \vdash T \ll \text{dynamic} \quad \Delta \vdash T \ll T \\
\\
\frac{\Delta \vdash T_1 \ll T_2 \quad \Delta \vdash T_2 \ll T_3}{\Delta \vdash T_1 \ll T_3} \\
\\
\Delta \vdash X \ll \Delta(X) \quad \frac{\Delta \vdash \overline{G_1} \ll \overline{G_2}}{\Delta \vdash c \langle \overline{G_1} \rangle \ll c \langle \overline{G_2} \rangle} \\
\\
\frac{\text{CT}(c) = \text{class } c \langle \overline{X} \triangleleft \overline{N} \rangle \text{ extends } N \{ \dots \}}{\Delta \vdash c \langle \overline{G} \rangle \ll [\overline{G}/\overline{X}]N}
\end{array}$$

Figure 3. Typing specificity.

method named `f` that returns the value of the field `f`. Similarly, `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.

3.4 Subtyping

A type environment Δ is a finite map from type variables to class types. We use the notation $X_1 <: N_1, \dots, X_n <: N_n$ 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 Δ iff $\Delta \vdash T_1 \ll T_2$ is provable according to Figure 3. Note that the rules follow the declared `extends` relationship, but they leave some special cases to subtyping (defined below). E.g., [SUB-DYN-SUB] makes `dynamic` a subtype of all other types.

Type rules for type specificity do not describe the full subtype relation for Fletch types. The special type annotation `dynamic` allows the programmer to leave a type unspecified in the program, and unchecked by the compiler. The type `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 `dynamic`. Generic type parameters may also be declared as `dynamic`.

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

$$\begin{array}{c}
\text{[SUB-DYN-SUB]} \frac{\Delta \vdash \text{dynsub}(T_1) \ll T_2}{\Delta \vdash T_1 <: T_2} \\
\\
\text{[SUB-FUN}_s\text{]} \frac{\text{assignable}_\Delta(\overline{G}_1, \overline{G}_2) \quad \boxed{\Delta \vdash T_1 <: T_2} \text{ or } T_2 = \text{void}}{\Delta \vdash (\overline{G}_1) \rightarrow T_1 <: (\overline{G}_2) \rightarrow T_2} \\
\\
\text{[SUB-FUN}_f\text{]} \frac{\text{assignable}_\Delta(\overline{G}_1, \overline{G}_2) \quad \boxed{\Delta \vdash T_1 <: T_2} \text{ or } T_2 = \text{void}}{\Delta \vdash (\overline{G}_1) \rightarrow T_1 <: (\overline{G}_2) \rightarrow T_2} \\
\\
\text{[SUB-OBJECT]} \Delta \vdash (\overline{G}_1) \rightarrow T_1 <: \text{Object}
\end{array}$$

Figure 4. Subtyping. The standard type system (most closely modeling Dart) uses [SUB-FUN_s], and the message-safe variant uses [SUB-FUN_f]. The boxes just point out the differences (cf. Section 2.3, requirement 2(b)).

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

$$\text{dynsub}(T) = \begin{cases} \perp & \text{if } T = \text{dynamic} \\ c < \text{dynsub}(\overline{G}) > & \text{if } T = c < \overline{G} > \\ T & \text{otherwise} \end{cases}$$

With $\text{dynsub}(T)$, we can define the subtype relation as shown in Figure 4. This ensures $\Delta \vdash \text{List} < \text{dynamic} > <: \text{List} < \text{String} >$, as $\Delta \vdash \text{List} < \perp > \ll \text{List} < \text{String} >$, which solves the previously mentioned transitivity problem.

The notion of assignability in object-oriented languages often coincides with subtyping. As Figure 5 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:

```

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

```

The careful reader may notice a subtle detail in the subtyping relation: as subtyping is used not only for static type checking, but also in checked mode runtime execution, the necessary modification of the function subtyping rule also affects the runtime semantics. Providing alternative static

$$\begin{array}{c}
\frac{\Delta \vdash T_1 <: T_2}{\text{assignable}_\Delta(T_1, T_2)} \\
\\
\frac{\Delta \vdash T_2 <: T_1 \quad T_1, T_2 \text{ are not function types}}{\text{assignable}_\Delta(T_1, T_2)} \\
\\
\frac{\text{assignable}_\Delta(\overline{G}_1, \overline{G}_2)}{\text{assignable}_\Delta(c < \overline{G}_1 >, c < \overline{G}_2 >)}
\end{array}$$

Figure 5. Assignability.

type checkers for Dart is explicitly encouraged by the language specification, in accordance with the choice by the language designers that presence of static type warnings does not prevent execution; however, changing the runtime semantics is another matter. Interestingly, the Dart language designers are seriously considering our proposal to replace [SUB-FUN_s] by [SUB-FUN_f], also at runtime [16]. Also note that message safety can be achieved even without this change to the runtime semantics by letting the compiler apply a simple program transformation using extra local variables for results returned by first-class closures.

3.5 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 ν, σ, Δ and Γ . Here, ν maps variable locations to heap locations, σ maps heap locations to objects or closures, Δ maps type variables to their upper bounds, and Γ maps variables to their declared types. When type checking a program, ν and σ will be empty, but they are required for type checking a program state during execution, i.e., in the soundness proof. The initial environments for an execution are ν_{base} and σ_{base} (see Section 3.2), $\Delta_{\text{base}} = \emptyset$, and $\Gamma_{\text{base}} = \{\text{null} : \perp\}$.

The Fletch type systems differ from the Dart type system in a couple of ways. In particular, in Figure 6 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. Finally, the message-safe variants of many rules encode requirements specific to message-safe programs.

The message-safe type system differs from the standard type system by introducing more strict requirements in the areas where Dart typing has the greatest degree of built-in unsoundness. However, as we mentioned in Section 2 there is a conflict between the use of assignability and proofs of soundness, which is the reason why the assignability premises are boxed in Figure 6. These premises are treated specially in our message-safety soundness proof (Section 4).

In fact, omitting these assignability premises only makes typing more flexible during some steps of execution. If stan-

$$\begin{array}{c}
\text{[T-VAR]} \nu; \sigma; \Delta; \Gamma \vdash y : \Gamma(y) \qquad \text{[T-READ]} \frac{\nu; \sigma; \Delta; \Gamma \vdash e_0 : T \quad \text{accessor}(\text{bound}_\Delta(T), p) = G}{\nu; \sigma; \Delta; \Gamma \vdash e_0.p : G} \\
\\
\text{[T-WRITE]} \frac{\nu; \sigma; \Delta; \Gamma \vdash e_0 : T_0 \quad \text{accessor}(\text{bound}_\Delta(T_0), f) = G \quad \nu; \sigma; \Delta; \Gamma \vdash e_1 : T_1 \quad \boxed{\text{assignable}_\Delta(T_1, G)}}{\nu; \sigma; \Delta; \Gamma \vdash e_0.f = e_1 : T_1} \\
\\
\text{[T-ASSIGN]} \frac{\nu; \sigma; \Delta; \Gamma \vdash e : T \quad \boxed{\text{assignable}_\Delta(T, \Gamma(x))}}{\nu; \sigma; \Delta; \Gamma \vdash x = e : T} \\
\\
\boxed{\text{T-DYNAMIC-CALL}_s} \frac{\nu; \sigma; \Delta; \Gamma \vdash e_0 : \text{dynamic} \quad \nu; \sigma; \Delta; \Gamma \vdash \bar{e} : \bar{T}}{\nu; \sigma; \Delta; \Gamma \vdash e_0(\bar{e}) : \text{dynamic}} \qquad \text{[T-NEW]} \frac{\Delta \vdash N \text{ OK}}{\nu; \sigma; \Delta; \Gamma \vdash \text{new } N() : N} \\
\\
\text{[T-CALL]} \frac{\nu; \sigma; \Delta; \Gamma \vdash e_0 : (\bar{G}) \rightarrow T \quad \nu; \sigma; \Delta; \Gamma \vdash \bar{e} : \bar{T}' \quad \boxed{\text{assignable}_\Delta(\bar{T}', \bar{G})}}{\nu; \sigma; \Delta; \Gamma \vdash e_0(\bar{e}) : T} \\
\\
\text{[T-FUNCTION]} \frac{\Delta \vdash \bar{G} \text{ OK} \quad \nu; \sigma; \Delta; \Gamma, \bar{x} : \bar{G} \vdash e_0 : T' \quad \boxed{\text{assignable}_\Delta(T', T)}}{\nu; \sigma; \Delta; \Gamma \vdash \boxed{T}(\bar{G} \bar{x}) \Rightarrow e_0 : (\bar{G}) \rightarrow T} \\
\\
\text{[T-RUNTIME-LOC]} \nu; \sigma; \Delta; \Gamma \vdash \boxed{l} : \text{typeof}(l, \sigma) \qquad \text{[T-RUNTIME-FRAME]} \frac{\nu; \sigma; \Delta; \Gamma \vdash e : T' \quad \boxed{\vdash T' <: T}}{\nu; \sigma; \Delta; \Gamma \vdash \boxed{[T, e]} : T} \\
\\
\text{[T-RUNTIME-VLOC]} \nu; \sigma; \Delta; \Gamma \vdash \boxed{\tau} : \text{typeof}(\tau, \nu) \\
\\
\text{[T-RUNTIME-VASSIGN]} \frac{\nu; \sigma; \Delta; \Gamma \vdash e : T \quad \boxed{\text{assignable}_\Delta(T, \text{typeof}(\tau, \nu))}}{\nu; \sigma; \Delta; \Gamma \vdash \boxed{\tau = e} : T}
\end{array}$$

Figure 6. Expression typing. Boxed parts in conclusions are Fletch artifacts that do not occur in the actual Dart syntax; boxed premises are treated specially in the proof of message-safety soundness (see Section 4). The rule [T-DYNAMIC-CALL_s] is included in the standard type system and omitted in the message-safe type system (cf. Section 2.3, requirement 1).

dart typing would have failed at some point — for example, because the type of an actual argument to a method invocation is not a subtype of the declared argument type — then such a relaxed variant of message-safe typing where these assignability premises are omitted will also fail at the point where the method is invoked, but it will allow the evaluation of the actual arguments to proceed to that point. In contrast, standard typing would fail as soon as an actual argument obtains a type that is not a subtype of the formal argument type. In other words, no errors are suppressed, but they may be detected later when using the relaxed variant of the message-safe type system.

The reason why the relaxation of the message-safe type system is necessary in the first place is that the standard type system “predicts” errors much earlier, and it is then not true that computation can proceed all the way until an easily recognizable error configuration has been reached. This means that the progress property only holds if specified in a complex manner (that does not offer any additional

insight, and by the way does not match the behavior of an actual implementation where argument evaluation would also be allowed to finish before an error is detected).

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. When assignability is omitted, even unrelated types are allowed.

The rule [T-CALL] is also unsurprising, apart from the fact that it allows for supertypes (with assignability) or unrelated types (without assignability) 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 $\bar{G} \bar{x}$ cannot contain any duplicate variable names.

Finally, the rules [T-RUNTIME-LOC], [T-RUNTIME-FRAME] and [T-RUNTIME-VASSIGN] are simple extrapolations from programs to runtime expressions, to be used in the soundness proof. The `typeof` function determines the type of a heap location or variable location.

$$\begin{array}{c}
\frac{\text{ftype}(G_1, f) = G_2}{\text{accessor}(G_1, f) = G_2} \quad \frac{\text{mtype}(G_1, m) = G_2}{\text{accessor}(G_1, m) = G_2} \\
\\
\frac{\text{ftype}(\text{bound}_\Delta(N), f) = G_2 \quad \text{implies } \boxed{\Delta \vdash G_1 <: G_2}}{\text{foverride}_\Delta(f, N, G_1)} \\
\\
\frac{\text{mtype}(m, \text{bound}_\Delta(N)) = (\overline{G_2}) \rightarrow T_2 \quad \text{implies } \boxed{\Delta \vdash \overline{G_1} \rightarrow T_1 <: \overline{G_2} \rightarrow T_2}}{\text{moverride}_\Delta(m, N, (\overline{G_1}) \rightarrow T_1)}
\end{array}$$

Figure 7. Auxiliary definitions. Boxed parts enforce properties required in message-safe programs (cf. Section 2.3, requirement 2(a)); the standard type system uses assignability instead of subtyping in those rules.

$$\begin{array}{c}
\frac{\text{CT}(c) = \text{class } c < \overline{X} \triangleleft \dots > \text{ extends } N \dots \{ \overline{F} \dots \} \quad \text{ftype}([\overline{G_1}/\overline{X}]N, f) = G_2 \quad f \notin \text{dom}(\overline{F})}{\text{ftype}(c < \overline{G_1} >, f) = G_2} \\
\\
\text{ftype}(\text{dynamic}, f) = \text{dynamic} \\
\\
\frac{\text{CT}(c) = \text{class } c < \overline{X} \triangleleft \dots > \dots \{ \dots G_2 f; \dots \}}{\text{ftype}(c < \overline{G_1} >, f) = [\overline{G_1}/\overline{X}]G_2} \\
\\
\frac{\text{CT}(c) = \text{class } c < \overline{X} \triangleleft \dots > \text{ extends } N \dots \{ \dots \overline{M} \} \quad \text{mtype}([\overline{G_1}/\overline{X}]N, m) = G_2 \quad m \notin \text{dom}(\overline{M})}{\text{mtype}(c < \overline{G_1} >, m) = G_2} \\
\\
\text{mtype}(\text{dynamic}, m) = \text{dynamic} \\
\\
\frac{\text{CT}(c) = \text{class } c < \overline{X} \triangleleft \dots > \dots \{ \dots T m(\overline{G_2} x) \{ \dots \} \}}{\text{mtype}(c < \overline{G_1} >, m) = [\overline{G_1}/\overline{X}](\overline{G_2} \rightarrow T)}
\end{array}$$

Figure 8. Lookup definitions.

Figure 7 defines a few auxiliary functions: `accessor` is a shorthand for property lookup, `foverride` defines requirements on redeclaring a field in a subclass, and `moverride` defines requirements on method overriding. The `boundΔ(T)` function replaces the type variables in the type T by their upper bound as defined in the type environment Δ .

Figure 8 defines field and method type lookup by the functions `ftype` and `mtype`. 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`.

$$\begin{array}{c}
\Delta \vdash \text{dynamic OK} \quad \Delta \vdash \perp \text{ OK} \quad \Delta \vdash \text{void OK} \\
\\
\Delta \vdash \text{Object OK} \\
\\
\frac{\Delta \vdash \overline{G} \text{ OK} \quad \Delta \vdash T \text{ OK}}{\Delta \vdash (\overline{G}) \rightarrow T \text{ OK}} \quad \frac{X \in \text{dom}(\Delta)}{\Delta \vdash X \text{ OK}} \\
\\
\frac{\text{CT}(c) = \text{class } c < \overline{X} \triangleleft \overline{N} > \text{ extends } N \{ \dots \} \quad \Delta \vdash \overline{G} <: [\overline{G}/\overline{X}]\overline{N} \quad \Delta \vdash \overline{G} \text{ OK}}{\Delta \vdash c < \overline{G} > \text{ OK}}
\end{array}$$

Figure 9. Well-formed types.

$$\begin{array}{c}
\frac{\Delta = \overline{X} <: \overline{N} \quad \Delta \vdash G \text{ OK} \quad \text{CT}(c) = \text{class } c < \overline{X} \triangleleft \overline{N} > \text{ extends } N \{ \dots \} \quad \text{foverride}_\Delta(f, N, G)}{G f \text{ OK in } c} \\
\\
\frac{\Delta = \overline{X} <: \overline{N} \quad \Delta \vdash T \text{ OK} \quad \Delta \vdash \overline{G} \text{ OK} \quad \emptyset; \nu_{\text{base}}; \Delta; \Gamma_{\text{base}}, x: \overline{G}, \text{this} : c < \overline{X} > \vdash e_0 : T_0 \quad \text{CT}(c) = \text{class } c < \overline{X} \triangleleft \overline{N} > \text{ extends } N \{ \dots \} \quad \text{assignable}_\Delta(T_0, T) \quad \text{moverride}_\Delta(m, N, (\overline{G}) \rightarrow T)}{T m(\overline{G} x) \{ \text{return } e_0; \} \text{ OK in } c} \\
\\
\frac{\Delta = \overline{X} <: \overline{N} \quad \Delta \vdash \overline{N} \text{ OK} \quad \Delta \vdash N \text{ OK} \quad \text{nodup}(\overline{X}) \quad \text{nodup}(\overline{f}) \quad \text{nodup}(\overline{m}) \quad \overline{F} \text{ OK in } c \quad \overline{M} \text{ OK in } c}{\text{class } c < \overline{X} \triangleleft \overline{N} > \text{ extends } N \{ \overline{F} \overline{M} \} \text{ OK}}
\end{array}$$

Figure 10. Typing of classes.

Figure 9 defines what it means for a type T to be well-formed in a type environment Δ , written $\Delta \vdash T \text{ OK}$. Type well-formedness requires subtyping for type parameters rather than assignability: if we have a class definition `class $c < X \triangleleft \text{String} > \{ \dots \}$` then `$c < \text{Object} >$` is not a well-formed type, since X must be a subtype of `String`.

Figure 10 shows the top-level rules for typing of classes that causes all the other elements of type checking to be applied. The notation `nodup(\overline{a})` indicates that the list \overline{a} contains no duplicates.

3.6 Modifications for Full Type Safety

Figure 11 formalizes the additional modifications from Section 2.4 that ensure full type safety. Note that class subtyping is modified via the typing specificity definition, and method overriding is modified via the function subtyping definition.

$$\begin{array}{c}
\boxed{\overline{G_1} = \overline{G_2}} \\
\hline
\Delta \vdash c < \overline{G_1} > \ll c < \overline{G_2} >
\end{array}
\quad
\begin{array}{c}
\text{ftype}(\text{bound}_\Delta(N), f) = G_2 \\
\text{implies } \boxed{G_1 = G_2} \\
\hline
\text{foverride}_\Delta(f, N, G_1)
\end{array}$$

$$\begin{array}{c}
\boxed{\Delta \vdash \overline{G_2} <: \overline{G_1}} \\
\hline
\text{[SUB-FUN}_f\text{]} \frac{\Delta \vdash T_1 <: T_2 \text{ or } T_2 = \text{void}}{\Delta \vdash (\overline{G_1}) \rightarrow T_1 <: (\overline{G_2}) \rightarrow T_2}
\end{array}$$

Figure 11. Modifications that ensure full type safety. The boxed parts show the changes compared to the message-safe type system (cf. Figures 3, 7, and 4). In addition, the second and third rules of Figure 5 are removed.

4. Soundness of Message Safety

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

The following lemmas use a relaxed form of the message-safe type system where the boxed premises in Figure 6 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 corollary, message-safety soundness also holds for the message-safe type system.

In the lemmas, the notation σ OK means that every location in the heap σ is well-formed. (The meaning of object well-formedness is defined in the technical report [10], and a similar criterion applies for closures.) The notation $\sigma \vdash \nu$ OK means that each variable location in ν is mapped to a pair (G, ι) such that $\text{typeof}(\iota, \sigma)$ is a subtype of G . The notation $\nu; \sigma \vdash e$ ACCEPTABLE_ERROR means that the configuration $\langle \nu, \sigma, e \rangle$ is a subtype violation or null pointer error (defined formally in the technical report [10]).

LEMMA 4.1 (Progress). *If $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T$ and σ OK and $\sigma \vdash \nu$ OK and e, ν , and σ do not contain dynamic then*

- a) e is a value (i.e., a heap location) or
- b) $\langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle$ for some ν', σ', e' or
- c) $\nu; \sigma \vdash e$ ACCEPTABLE_ERROR.

Proof By induction on the typing derivation $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T$. See the proof in Coq for further details.⁴ \square

⁴The proof (<http://www.brics.dk/fletcher/>) contains some unproven (‘Admitted’) but plausible lemmas. Technically, soundness has been reduced to those lemmas.

LEMMA 4.2 (Preservation). *If $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T$ and σ OK and $\sigma \vdash \nu$ OK and $\langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle$ and e, ν , and σ do not contain dynamic and $\sigma_{\text{base}} \subseteq \sigma$ then both of the following hold:*

- 1) σ' OK and $\sigma' \vdash \nu'$ OK and $\sigma_{\text{base}} \subseteq \sigma'$ and e', ν' , and σ' do not contain dynamic and either
- 2a) $\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e' : T'$ where $\emptyset \vdash T' <: T$ or
- 2b) $\nu'; \sigma' \vdash e'$ ACCEPTABLE_ERROR.

Proof Induction on the derivation $\langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle$ (see the Coq sources). \square

From these lemmas we can obtain the soundness result:

THEOREM 4.3 (Message-safety soundness).

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

- a) e' is a value and $\nu'; \sigma'; \emptyset; \Gamma_{\text{base}} \vdash e' : T'$ and $\emptyset \vdash T' <: T$ or
- b) $\sigma'; \nu' \vdash e'$ ACCEPTABLE_ERROR.

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

Proof Induction on $\langle \nu, \sigma, e \rangle \longrightarrow^* \langle \nu', \sigma', e' \rangle$ (see the Coq sources). \square

The variant with full type safety (Section 3.6) additionally rules out subtype violation errors, and applies to both checked and production mode execution. We omit a formalization of this property.

5. Experiments

Violations of message safety Although the focus of this paper is on the theoretical development of message safety, we have performed an initial experiment to assess the impact of our approach for the Dart language. The overall results are as follows. We modified the *dartanalyzer* tool to perform message-safety checking and used it on the Dart tool chain, the core libraries, several other libraries, and some applications. We found that violations of the message safety requirements 2(a) and 2(b) (see Section 2.3) are generally design flaws. For instance, 4 bugs were accepted and fixed by the Dart team. More details of this experiment are described in the technical report [10].

Modifying function subtyping As explained in Section 3.4, 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 4) 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 [16].

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. 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 10 264 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.

6. Related Work

The Featherweight Java formalization [15] 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. In the report [15] there is no notion of variance. Moreover, the Dart approach to variance is very different from the approach taken in Java. 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 [15] or in the original work on Featherweight Java [13].

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* [5], later complemented by [22]. The basic idea is that expressions whose type do not satisfy the requirements by the context are 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 [4] 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 [3] 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 systems⁵, 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 [11] 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 $p1 \Rightarrow p2$), 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 [18] uses conventional type annotations extended with ‘?’, which corresponds to the Dart `dynamic` type. It builds on $Ob_{<}$: [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 allow for general computation (and hence, no static checking) in Scheme [6, 19], with a special emphasis on tracking blame for first-class functions that only reveal typing violations when invoked. Neither Fletch nor Dart support contracts, but in a sense they are not needed because the type of first-class functions can be checked dynamically.

Like types [23] 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 this work 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.

Another recent paper presents *progressive types* [17], 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 Ω is a fully type safe system.

Finally, TypeScript [2] enables optional type annotations in JavaScript programs. Using structural types and coinductive subtype rules, the foundations differ substantially from Dart and Fletch.

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 sim-

⁵<https://www.dartlang.org/slides/2011/11/stanford/dart-a-walk-on-the-dart-side.pdf>

ilar to our notion of message-safe programs. Moreover, we believe this is the first formalization of the core of Dart.

Success typing is a way to design complete but unsound type systems [14], 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* [21] 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.

7. Conclusion

We have introduced Fletch as a core of the Dart programming language to expose the central aspects of its type system. Moreover, we have proposed the notion of message-safe programs as a natural intermediate point between dynamically typed and statically typed Dart programs. 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. In future work we plan to explore experimentally how tool support can guide Dart programmers toward type safe programs via message-safe programs. Also, we believe Fletch and our formalization may be useful in further studies of Dart and related programming languages.

Acknowledgments

This work was supported by the Danish Research Council for Technology and Production and by Google Faculty Research Award grants.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *ECOOP*, pages 257–281, 2014.

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

[4] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

[5] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[6] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*, pages 215–226, 2011.

[7] Ecma International. *C# Language Specification, ECMA-334*, June 2006.

[8] Ecma International. *ECMAScript Language Specification, ECMA-262*, June 2011.

[9] Ecma International. *Dart Programming Language Specification, ECMA-408*, December 2014.

[10] E. Ernst, A. Møller, M. Schwarz, and F. Stocco. Message safety in Dart. Technical report, Department of Computer Science, Aarhus University, 2015. <http://www.brics.dk/fletch/>.

[11] C. Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.

[12] J. Gosling, B. Joy, G. L. Steele Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[13] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[14] T. Lindahl and K. F. Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178, 2006.

[15] J. Mackay, H. Mehnert, A. Potanin, L. Groves, and N. Cameron. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. Technical report, Victoria University of Wellington, 2012.

[16] Personal communication with Gilad Bracha and the Dart team at Google, 2015.

[17] J. G. Politz, H. Q. de la Vallee, and S. Krishnamurthi. Progressive types. In *Onward!*, pages 55–66, 2012.

[18] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, pages 151–175, 2007.

[19] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

[20] K. Walrath and S. Ladd. *dart: The Standalone VM*. Google, June 2014. <https://www.dartlang.org/docs/dart-up-and-running/contents/ch04-tools-dart-vm.html>.

[21] J. Winther and M. I. Schwartzbach. Related types. In *ECOOP*, pages 434–458, 2011.

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

[23] T. Wrigstad, F. Z. Nardelli, S. Lebesne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, pages 377–388, 2010.