Message Safety in Dart

Erik Ernst[1] Anders Møller[2] Mathias Schwarz[2] * Fabio Strocco[2]

Google, Aarhus, Denmark[1]
eernst@google.com

Aarhus University, Denmark[2]
{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, 12, 15, 19, 20, 22, 26, 27]. The Dart programming language [11] strikes an interesting new balance, with a type system far less restrictive than required for traditional 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 [14] and based on the most recent language specification [11].

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 the syntax, the small-step operational semantics, and the type system, together with a soundness theorem stating that message-safe programs do not cause message-not-understood errors in checked mode execution.

* Current affiliation: Uber, Inc.

This technical report is an extended version of E. Ernst, A. Møller, M. Schwarz, and F. Strocco. Message safety in Dart. In *Proc. 11th Dynamic Languages Symposium (DLS)*, October 2015.

- 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 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 described in Section 5. Section 6 discusses related work and Section 7 concludes.

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 [13] and C# [9], and others with JavaScript [10]. 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 [24].

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 [18]: 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 [15]. 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, $T_1\ \mathtt{x} = e$, 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 $T_1\ \mathtt{x} = e'$ is type correct. However, without transitivity, we cannot conclude that $T_1\ \mathtt{x} = e'$ 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 warnings[1] 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.[2]

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

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

[1] The 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.

[2] Our implementation (`http://www.brics.dk/fletch/`) required modifying less than 200 LOC in Google's *dartanalyzer* tool.
  ```

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.

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 modifications to the type checker:

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 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.

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 prevent 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 [7].

- ```
  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 2.3 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.

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`) in such a program, the declared type of the receiver object (`x`) ensures that the property (`f`) is defined. This aligns well with the use of nominal typing in object-oriented programming (unlike structural typing, which allows any two types that are structurally similar to be confused). 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 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 `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 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. 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.

2.7 A Two-Step Approach Toward Type Safety

The Dart language specification [11, 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*.

2.8 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 `invokedynamic` 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.

3. Fletch

Fletch[3] 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 [14] 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 ($b\,\texttt{?}\,e_1\texttt{:}e_2$, similar to e.g. Java), type tests (e `is` T, like `instanceof` in Java), and type promotion (allowing types to be refined based on type tests in conditionals), which are relevant for the Dart type system, but not essential when studying message safety.

Apart from a couple of trivial syntactic abbreviations and some extensions needed to describe runtime states, Fletch is a syntactic 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, $\overline{a}$ denotes the possibly empty list $a_1, \ldots, a_n$, $n \geq 0$.

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 (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, c, 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.

$$
\begin{array}{lcl}
CL & ::= & \texttt{class}\ c\ \texttt{<}\overline{X \lhd N}\texttt{>}\ \texttt{extends}\ N\ \{\overline{F}\ \overline{M}\} \\
F & ::= & G\ f; \\
M & ::= & T\ m(\overline{G\ x})\ \{\texttt{return}\ e;\} \\
e & ::= & y \mid e.p \mid e.p = e \mid x = e \mid e(\overline{e}) \mid \\
 & & \texttt{new}\ N() \mid \mathit{fn} \mid \boxed{[\![T, e]\!] \mid \tau = e \mid l} \\
T & ::= & G \mid \texttt{void} \\
G & ::= & X \mid N \mid \texttt{dynamic} \mid \bot \mid (\overline{G}) \to T \\
N & ::= & c\texttt{<}\overline{G}\texttt{>} \\
 & & \\
\mathit{fn} & ::= & T\ (\overline{G\ x})\ \Rightarrow e \\
y & ::= & x \mid \texttt{this} \mid \texttt{null} \\
p & ::= & f \mid m \\
l & ::= & \boxed{\iota \mid \tau}
\end{array}
$$

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

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 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* $\mathsf{CT} : c \hookrightarrow CL$, which maps a finite set of class names into class definitions ('$\hookrightarrow$' indicates a partial function). Each class definition $\texttt{class}\ c\ \texttt{<}\overline{X \lhd N}\texttt{>}\ \texttt{extends}\ N\ \{\overline{F}\ \overline{M}\}$ defines a class c with superclass N, fields $\overline{F}$, and methods $\overline{M}$. Each type parameter X has an upper bound N (the actual Dart syntax is X `extends` N).

A class table CT is *well-formed* iff $\texttt{Object} \notin \mathit{dom}(\mathsf{CT})$, but every other class name used in CT is defined, and inheritance is acyclic. A Fletch *program* is a pair (CT, e) where

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

$$
\begin{array}{lll}
o & : \mathsf{Obj} = G \times \mathsf{Fields} \times \mathsf{Methods} \\
\phi & : \mathsf{Fields} = f \hookrightarrow G \times \mathsf{LocH} \\
\mu & : \mathsf{Methods} = m \hookrightarrow \mathsf{LocH} \\
\sigma & : \mathsf{Heap} = \mathsf{LocH} \hookrightarrow \mathsf{Obj} \cup \mathit{fn} \\
\nu & : \mathsf{VarEnv} = \mathsf{LocV} \hookrightarrow G \times \mathsf{LocH} \\
s & : \mathsf{State} = \mathsf{VarEnv} \times \mathsf{Heap} \times e
\end{array}
$$

Figure 2. Semantic entities.

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 (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 $\iota \in \mathsf{LocH}$ to an object or a closure, and the latter maps each variable location $\tau \in \mathsf{LocV}$ 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 $\tau_{\texttt{this}}$.

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 $[]$ 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 [14].

The null location is predefined, which motivates the use of the base environments $\nu_{\text{base}} = []$ and $\sigma_{\text{base}} = [\iota_{\texttt{null}} \mapsto o_{\texttt{null}}]$, where $o_{\texttt{null}} = (\bot, [], [])$ represents the predefined null object. 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 Figure 3. Every successfully terminating expression evaluates to a heap location ι, 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 ι'. Similarly, $\nu(\tau) = \iota$ means that there exists a G such that $\nu(\tau) = (G, \iota)$. Evaluation of a variable location [E-VAR-READ] amounts to a lookup in ν for a location τ. Assignment to a variable lo-

cation τ [E-VAR-WRITE] updates the variable environment ν 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 ι is a subtype of the statically declared type of the variable location τ. Assignment to a field [E-FIELD-WRITE] looks up the object at ι_1 and creates a new heap σ' that differs from the old heap only at ι_1, which contains the object updated only at the selected field f to have the new value ι_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-FIELD-READ] or a method [E-METHOD-READ]s is straightforward, and the null literal [E-NULL] evaluates to the null heap address.

The `new` expression [E-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 $\tau_{\texttt{this}}$ 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, similar to `ftype` and `mtype` from Figure 11. We use `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 $M = T\ m(\overline{G\ x})\ \{\texttt{return}\, e;\ \}$. Similarly, $\texttt{type}(F)$ extracts the declared type for a field declaration.

Closure creation [E-FUNC] stores the given closure in the heap and evaluates to the corresponding heap location. Closure invocation [E-CALL] evaluates the body of the function in a new variable environment ν' created by combining the current variable environment ν 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 [E-RETURN] performs this check, if in checked mode, and produces the contained value.

Figure 4 shows how a failed runtime configuration can be detected, which is necessary in order to distinguish between an execution that stops with a subtype-violation or a null pointer error vs. one that stops by encountering a message-not-understood. The former is a configuration $\langle \nu, \sigma, e \rangle$ where $\nu; \sigma \vdash e$ `ACCEPTABLE_ERROR`; the latter is any other stuck configuration.

We omit the associated congruence rules, both in Figure 3 and in Figure 4, as they are entirely unsurprising.

As Figure 3 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 5 shows the definition of `typeof`, which takes a heap location ι or a variable location τ and determines the requested type.

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.[4] 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 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`. To keep Fletch simple, we model only the automatically generated getters and setters.

3.4 Subtyping

A type environment Δ is a finite map from type variables to class types. We use the notation $X_1 <: N_1, \ldots, X_n <: N_n$ 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 Δ iff $\Delta \vdash T_1 \ll T_2$ is provable according to Figure 6. 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 `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 $\Delta \vdash$ `List<int>` $<:$ `List<dynamic>` and $\Delta \vdash$ `List<dynamic>` $<:$ `List<String>`. If the rules had been transitive we could conclude $\Delta \vdash$ `List<int>` $<:$ `List<String>`, which should not hold. Transitivity only holds among class types, but not when the type `dynamic` is used.

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

[4] A *tear-off* in Dart is a closure that has been derived from a method [11].

$$\text{[E-VAR-READ]}\ \frac{\nu(\tau) = \iota}{\langle \nu, \sigma, \tau \rangle \longrightarrow \langle \nu, \sigma, \iota \rangle} \qquad \text{[E-VAR-WRITE]}\ \frac{\nu' = \nu[\tau \mapsto \iota] \qquad \boxed{\vdash \texttt{typeof}(\iota, \sigma) <: \texttt{typeof}(\tau, \nu)}}{\langle \nu, \sigma, \tau = \iota \rangle \longrightarrow \langle \nu', \sigma, \iota \rangle}$$

$$\text{[E-FIELD-WRITE]}\ \frac{\begin{array}{c}\sigma(\iota_1) = (c\texttt{<}\overline{G}\texttt{>}, \phi, \mu) \qquad \phi(f) = (G', \_) \\ \sigma' = \sigma[\iota_1 \mapsto (c\texttt{<}\overline{G}\texttt{>}, \phi[f \mapsto (G', \iota_2)], \mu)] \qquad \boxed{\vdash \texttt{typeof}(\iota_2, \sigma) <: G'}\end{array}}{\langle \nu, \sigma, \iota_1.f = \iota_2 \rangle \longrightarrow \langle \nu, \sigma', \iota_2 \rangle}$$

$$\text{[E-FIELD-READ]}\ \frac{\sigma(\iota_1) = (\_, \phi, \_) \qquad \phi(f) = (\_, \iota_2)}{\langle \nu, \sigma, \iota_1.f \rangle \longrightarrow \langle \nu, \sigma, \iota_2 \rangle} \qquad \text{[E-METHOD-READ]}\ \frac{\sigma(\iota_1) = (\_, \_, \mu) \qquad \mu(m) = \iota_2}{\langle \nu, \sigma, \iota_1.m \rangle \longrightarrow \langle \nu, \sigma, \iota_2 \rangle}$$

$$\text{[E-NULL]}\ \langle \nu, \sigma, \texttt{null} \rangle \longrightarrow \langle \nu, \sigma, \iota_{\texttt{null}} \rangle$$

$$\text{[E-NEW]}\ \frac{\begin{array}{c}\overline{F} = \texttt{fields}(c\texttt{<}\overline{G}\texttt{>}) \\ \overline{M} = \texttt{methods}(c\texttt{<}\overline{G}\texttt{>}) \qquad o = (c\texttt{<}\overline{G}\texttt{>}, [\overline{\texttt{name}(F) \mapsto (\texttt{type}(F), \iota_{\texttt{null}})}], [\overline{\texttt{name}(M) \mapsto \iota_{m_i}}]) \\ \sigma_0 = \sigma[\iota \mapsto o] \quad \text{where } \iota \text{ is fresh} \qquad \nu' = \nu[\tau_{\texttt{this}} \mapsto (c\texttt{<}\overline{G}\texttt{>}, \iota)] \quad \text{where } \tau_{\texttt{this}} \text{ is fresh} \\ \forall M_i \in \overline{M} \colon \sigma_i = \sigma_{i-1}[\iota_{m_i} \mapsto T_i\ m_i(\overline{G_i\ x_i})\ \Rightarrow\ [\tau_{\texttt{this}}/\texttt{this}]e_i] \\ \text{where } M_i = T_i\ m_i(\overline{G_i\ x_i})\ \{\texttt{return}\ e_i;\} \text{ and } \iota_{m_i} \text{ is fresh}\end{array}}{\langle \nu, \sigma, \texttt{new}\ c\texttt{<}\overline{G}\texttt{>}() \rangle \longrightarrow \langle \nu', \sigma_n, \iota \rangle}$$

$$\text{[E-FUNC]}\ \frac{\sigma' = \sigma[\iota \mapsto T\ (\overline{G\ x})\ \Rightarrow\ e] \quad \text{where } \iota \text{ is fresh}}{\langle \nu, \sigma, T\ (\overline{G\ x})\ \Rightarrow\ e \rangle \longrightarrow \langle \nu, \sigma', \iota \rangle}$$

$$\text{[E-CALL]}\ \frac{\sigma(\iota_0) = T\ (\overline{G\ x})\ \Rightarrow\ e \qquad \nu' = \nu[\overline{\tau \mapsto (G, \iota)}] \quad \text{where } \overline{\tau} \text{ is fresh} \qquad \boxed{\vdash \texttt{typeof}(\iota, \sigma) <: \overline{G}}}{\langle \nu, \sigma, \iota_0(\overline{\iota}) \rangle \longrightarrow \langle \nu', \sigma, [\![T, [\overline{\tau/x}]e]\!] \rangle}$$

$$\text{[E-RETURN]}\ \frac{\boxed{\vdash \texttt{typeof}(\iota, \sigma) <: T}}{\langle \nu, \sigma, [\![T, \iota]\!] \rangle \longrightarrow \langle \nu, \sigma, \iota \rangle}$$

Figure 3. Computational rules for expressions in Fletch. The boxed premises involving `typeof` are omitted for production mode execution, but included for checked mode execution.

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

With $\texttt{dynsub}(T)$, we can define the subtype relation as shown in Figure 7. Notice that [SUB-DYN-SUB] makes `dynamic` a subtype of all other types. This ensures $\Delta \vdash \texttt{List<dynamic>} <: \texttt{List<String>}$, as $\Delta \vdash \texttt{List<}\bot\texttt{>} \ll \texttt{List<String>}$, which solves the previously mentioned transitivity problem.

The notion of assignability in object-oriented languages often coincides with subtyping. As Figure 8 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 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

$$\nu;\sigma \vdash \iota_{\texttt{null}}.p \ \texttt{ACCEPTABLE\_ERROR}$$

$$\nu;\sigma \vdash \iota_{\texttt{null}}.f = \iota \ \texttt{ACCEPTABLE\_ERROR}$$

$$\nu;\sigma \vdash \iota_{\texttt{null}}(\overline{t}) \ \texttt{ACCEPTABLE\_ERROR}$$

$$\frac{\vdash \texttt{typeof}(\iota,\sigma) \not<: \texttt{typeof}(\tau,\nu)}{\nu;\sigma \vdash \tau = \iota \ \texttt{ACCEPTABLE\_ERROR}}$$

$$\frac{\sigma(\iota_1) = (\_,\phi,\_) \qquad \phi(f) = (G,\_) \qquad \texttt{typeof}(\iota_2,\sigma) = T \qquad \vdash T \not<: G}{\nu;\sigma \vdash \iota_1.f = \iota_2 \ \texttt{ACCEPTABLE\_ERROR}}$$

$$\frac{\texttt{typeof}(\iota,\sigma) = (\overline{G}) \to T \qquad \texttt{typeof}(\iota_i,\sigma) = T_i' \qquad \vdash T_i' \not<: G_i}{\nu;\sigma \vdash \iota(\overline{t}) \ \texttt{ACCEPTABLE\_ERROR}}$$

$$\frac{\vdash \texttt{typeof}(\iota,\sigma) \not<: T}{\nu;\sigma \vdash [\![T,\iota]\!] \ \texttt{ACCEPTABLE\_ERROR}}$$

Figure 4. Acceptable runtime errors in message-safe programs.

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

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

Figure 5. Definition of $\texttt{typeof}(\iota,\sigma)$, which looks up the dynamic type of a heap location ι in the heap σ, and $\texttt{typeof}(\tau,\nu)$, which looks up the declared type of a variable location τ in the variable environment ν.

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 [18]. 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

$$\Delta \vdash \bot \ll T \qquad \Delta \vdash T \ll \texttt{dynamic} \qquad \Delta \vdash T \ll T$$

$$\frac{\Delta \vdash T_1 \ll T_2 \quad \Delta \vdash T_2 \ll T_3}{\Delta \vdash T_1 \ll T_3}$$

$$\Delta \vdash X \ll \Delta(X) \qquad \frac{\Delta \vdash \overline{G_1} \ll \overline{G_2}}{\Delta \vdash c\langle\overline{G_1}\rangle \ll c\langle\overline{G_2}\rangle}$$

$$\frac{\mathsf{CT}(c) = \texttt{class}\ c\langle\overline{X \lhd N}\rangle\ \texttt{extends}\ d\langle\overline{G_1}\rangle\ \{\cdots\} \qquad \Delta \vdash \texttt{dynsub}(\overline{G_1}) \ll \overline{G_3}}{\Delta \vdash c\langle\overline{G_2}\rangle \ll [\overline{G_2/X}]d\langle\overline{G_3}\rangle}$$

Figure 6. Typing specificity.

$$[\text{SUB-DYN-SUB}]\ \frac{\Delta \vdash \texttt{dynsub}(T_1) \ll T_2}{\Delta \vdash T_1 <: T_2}$$

$$[\text{SUB-FUN}_s]\ \frac{\texttt{assignable}_\Delta(\overline{G_1},\overline{G_2}) \quad \boxed{\texttt{assignable}_\Delta(T_1,T_2)}\ \text{or}\ T_2 = \texttt{void}}{\Delta \vdash (\overline{G_1}) \to T_1 <: (\overline{G_2}) \to T_2}$$

$$[\text{SUB-FUN}_f]\ \frac{\texttt{assignable}_\Delta(\overline{G_1},\overline{G_2}) \quad \boxed{\Delta \vdash T_1 <: T_2}\ \text{or}\ T_2 = \texttt{void}}{\Delta \vdash (\overline{G_1}) \to T_1 <: (\overline{G_2}) \to T_2}$$

$$[\text{SUB-OBJECT}]\ \Delta \vdash (\overline{G_1}) \to T_1 <: \texttt{Object}$$

Figure 7. 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)).

$$\frac{\Delta \vdash T_1 <: T_2}{\texttt{assignable}_\Delta(T_1,T_2)}$$

$$\frac{\Delta \vdash T_2 <: T_1 \quad \neg\texttt{isfun}(T_1) \vee \neg\texttt{isfun}(T_2)}{\texttt{assignable}_\Delta(T_1,T_2)}$$

Figure 8. Assignability. The predicate $\texttt{isfun}(T)$ holds iff T is a function type.

(including `null` and `this`) to their declared types. The initial environments for an execution are ν_{base} and σ_{base} (see Section 3.2), $\Delta_{\text{base}} = \emptyset$, and $\Gamma_{\text{base}} = \{\texttt{null} : \bot\}$.

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

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

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

$$\boxed{\text{T-DYNAMIC-CALL}_s}\ \frac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : \texttt{dynamic} \qquad \nu;\sigma;\Delta;\Gamma \vdash \overline{e} : \overline{T}}{\nu;\sigma;\Delta;\Gamma \vdash e_0(\overline{e}) : \texttt{dynamic}} \qquad [\text{T-NEW}]\ \frac{\Delta \vdash N\ \texttt{OK}}{\nu;\sigma;\Delta;\Gamma \vdash \texttt{new}\ N() : N}$$

$$[\text{T-CALL}]\ \frac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : (\overline{G}) \to T \qquad \nu;\sigma;\Delta;\Gamma \vdash \overline{e} : \overline{T'} \qquad \boxed{\texttt{assignable}_\Delta(\overline{T'}, \overline{G})}}{\nu;\sigma;\Delta;\Gamma \vdash e_0(\overline{e}) : T}$$

$$[\text{T-FUNCTION}]\ \frac{\Delta \vdash \overline{G}\ \texttt{OK} \qquad \nu;\sigma;\Delta;\Gamma, \overline{x : G} \vdash e_0 : T' \qquad \boxed{\texttt{assignable}_\Delta(T', T)}}{\nu;\sigma;\Delta;\Gamma \vdash \boxed{T}\ (\overline{G\ x}) \Rightarrow e_0 : (\overline{G}) \to T}$$

$$[\text{T-RUNTIME-LOC}]\ \nu;\sigma;\Delta;\Gamma \vdash \boxed{\iota} : \texttt{typeof}(\iota, \sigma) \qquad [\text{T-RUNTIME-FRAME}]\ \frac{\nu;\sigma;\Delta;\Gamma \vdash e : T' \qquad \boxed{\vdash T' <: T}}{\nu;\sigma;\Delta;\Gamma \vdash \boxed{[\![T, e]\!]} : T}$$

$$[\text{T-RUNTIME-VLOC}]\ \nu;\sigma;\Delta;\Gamma \vdash \boxed{\tau} : \texttt{typeof}(\tau, \nu)$$

$$[\text{T-RUNTIME-VASSIGN}]\ \frac{\nu;\sigma;\Delta;\Gamma \vdash e : T \qquad \boxed{\texttt{assignable}_\Delta(T, \texttt{typeof}(\tau, \nu))}}{\nu;\sigma;\Delta;\Gamma \vdash \boxed{\tau = e} : T}$$

Figure 9. 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).

The Fletch type system differs from the Dart language specification in a couple of ways. In particular, in Figure 9 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. Furthermore, the message-safe variants of many rules encode requirements specific to message-safe programs.

As 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 9. These premises are treated specially in our message-safety soundness proof (Section 4).

The rules [T-VAR], [T-READ], [T-WRITE], [T-ASSIGN], and [T-CALL] 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 [T-NEW] rule is very simple because mutability allows us to omit constructors. [T-FUNCTION] is also standard, noting that the list $\overline{G\ x}$ 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 (see Figure 5).

Figure 10 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 $\texttt{bound}_\Delta(T)$ function replaces the type variables in the type T by their upper bound as defined in the type environment Δ.

Figure 11 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`

$$\frac{\texttt{ftype}(G_1, f) = G_2}{\texttt{accessor}(G_1, f) = G_2} \qquad \frac{\texttt{mtype}(G_1, m) = G_2}{\texttt{accessor}(G_1, m) = G_2}$$

$$\frac{\texttt{ftype}(\texttt{bound}_\Delta(N), f) = G_2 \quad \text{implies } \boxed{\Delta \vdash G_1 <: G_2}}{\texttt{foverride}_\Delta(f, N, G_1)}$$

$$\frac{\texttt{mtype}(m, \texttt{bound}_\Delta(N)) = (\overline{G_2}) \to T_2 \quad \text{implies } \boxed{\Delta \vdash \overline{G_1} \to T_1 <: \overline{G_2} \to T_2}}{\texttt{moverride}_\Delta(m, N, (\overline{G_1}) \to T_1)}$$

Figure 10. 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.

$$\frac{\mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd \ldots}\texttt{>} \texttt{ extends } N \ldots \{\overline{F} \ldots\} \quad \texttt{ftype}([\overline{G_1}/\overline{X}]N, f) = G_2 \qquad f \notin \overline{\texttt{name}(F)}}{\texttt{ftype}(c\texttt{<}\overline{G_1}\texttt{>}, f) = G_2}$$

$$\texttt{ftype}(\texttt{dynamic}, f) = \texttt{dynamic}$$

$$\frac{\mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd \ldots}\texttt{>} \ \ldots \{ \ldots \ G_2 \ f; \ldots \}}{\texttt{ftype}(c\texttt{<}\overline{G_1}\texttt{>}, f) = [\overline{G_1}/\overline{X}]G_2}$$

$$\frac{\mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd \ldots}\texttt{>} \texttt{ extends } N \ldots \{\ldots \overline{M}\} \quad \texttt{mtype}([\overline{G_1}/\overline{X}]N, m) = G_2 \qquad m \notin \overline{\texttt{name}(M)}}{\texttt{mtype}(c\texttt{<}\overline{G_1}\texttt{>}, m) = G_2}$$

$$\texttt{mtype}(\texttt{dynamic}, m) = \texttt{dynamic}$$

$$\frac{\mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd \ldots}\texttt{>} \ \ldots \{ \ldots \ T \ m(\overline{G_2 \ x}) \ \{ \ldots \} \}}{\texttt{mtype}(c\texttt{<}\overline{G_1}\texttt{>}, m) = [\overline{G_1}/\overline{X}](\overline{G_2}) \to T)}$$

Figure 11. Lookup definitions.

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

Figure 13 shows the top-level rules for typing of classes that causes all the other elements of type checking to be applied. The notation $\texttt{nodup}(\overline{a})$ indicates that the list $\overline{a}$ contains no duplicates.

$$\Delta \vdash \texttt{dynamic OK} \qquad \Delta \vdash \bot \texttt{ OK} \qquad \Delta \vdash \texttt{void OK}$$

$$\Delta \vdash \texttt{Object OK}$$

$$\frac{\Delta \vdash \overline{G} \texttt{ OK} \quad \Delta \vdash T \texttt{ OK}}{\Delta \vdash (\overline{G}) \to T \texttt{ OK}} \qquad \frac{X \in dom(\Delta)}{\Delta \vdash X \texttt{ OK}}$$

$$\frac{\mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd N}\texttt{> extends } N \ \{\cdots\} \quad \Delta \vdash \overline{G} <: [\overline{G}/\overline{X}]\overline{N} \quad \Delta \vdash \overline{G} \texttt{ OK}}{\Delta \vdash c\texttt{<}\overline{G}\texttt{> OK}}$$

Figure 12. Well-formed types.

$$\frac{\Delta = \overline{X <: N} \quad \Delta \vdash G \texttt{ OK} \quad \mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd N}\texttt{> extends } N \ \{\cdots\} \quad \texttt{foverride}_\Delta(f, N, G)}{G \ f \texttt{ OK } in \ c}$$

$$\frac{\Delta = \overline{X <: N} \quad \Delta \vdash T \texttt{ OK} \quad \Delta \vdash \overline{G} \texttt{ OK} \quad \emptyset; \nu_{\text{base}}; \Delta; \Gamma_{\text{base}}, \overline{x : G}, \texttt{this} : c\texttt{<}\overline{X}\texttt{>} \vdash e_0 : T_0 \quad \mathsf{CT}(c) = \texttt{class } c\texttt{<}\overline{X \lhd N}\texttt{> extends } N \ \{\cdots\} \quad \texttt{assignable}_\Delta(T_0, T) \quad \texttt{moverride}_\Delta(m, N, (\overline{G}) \to T)}{T \ m(\overline{G \ x})\{ \texttt{ return } e_0; \} \texttt{ OK } in \ c}$$

$$\frac{\Delta = \overline{X <: N} \quad \Delta \vdash \overline{N} \texttt{ OK} \quad \Delta \vdash N \texttt{ OK} \quad \texttt{nodup}(\overline{X}) \quad \texttt{nodup}(\overline{f}) \quad \texttt{nodup}(\overline{m}) \quad \overline{F} \texttt{ OK } in \ c \quad \overline{M} \texttt{ OK } in \ c}{\texttt{class } c\texttt{<}\overline{X \lhd N}\texttt{> extends } N \ \{\overline{F} \ \overline{M}\} \texttt{ OK}}$$

Figure 13. Typing of classes.

3.6 Modifications for Full Type Safety

Figure 14 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.

4. Soundness of Message Safety

Soundness is traditionally associated with Milner's phrase *well-typed programs cannot go wrong* [17], but message safety allows for subtype-violation errors (and null pointer errors), whereas message-not-understood must be ruled out.

$$\frac{\boxed{\overline{G_1} = \overline{G_2}}}{\Delta \vdash c\langle\overline{G_1}\rangle \ll c\langle\overline{G_2}\rangle}$$

$$\frac{\mathsf{CT}(c) = \texttt{class}\ c\langle\overline{X \lhd N}\rangle\ \texttt{extends}\ d\langle\overline{G_1}\rangle\ \{\cdots\} \quad \boxed{\overline{G_1} = \overline{G_3}}}{\Delta \vdash c\langle\overline{G_2}\rangle \ll [\overline{G_2/X}]d\langle\overline{G_3}\rangle}$$

$$\frac{\texttt{ftype}(\texttt{bound}_\Delta(N), f) = G_2 \quad \text{implies } \boxed{G_1 = G_2}}{\texttt{foverride}_\Delta(f, N, G_1)}$$

$$[\textsc{Sub-Fun}_f]\ \frac{\boxed{\Delta \vdash \overline{G_2} <: \overline{G_1}} \quad \Delta \vdash T_1 <: T_2 \text{ or } T_2 = \texttt{void}}{\Delta \vdash (\overline{G_1}) \to T_1 <: (\overline{G_2}) \to T_2}$$

Figure 14. Modifications that ensure full type safety. The boxed parts show the changes compared to the message-safe type system (cf. Figures 6, 7, and 10). In addition, the second rule of Figure 8 is removed.

As usual, the main steps on the way to a type soundness proof are progress and preservation.

As discussed in Section 2.3 we focus on checked mode execution. The following lemmas use a relaxed form of the message-safe type system where the boxed premises (expressing assignability) in Figure 9 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 σ OK means that every location in the heap σ is well-formed. (Figure 15 shows what it means for an object to be well-formed, 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 $\texttt{typeof}(\iota, \sigma)$ 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 Figure 4).

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 (CT, e), and $\Gamma = \Gamma_{\text{base}}$ because free variables (including this) are always substituted with variable locations during execution (due to rules [E-Call] and [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.

$$\frac{\overline{\texttt{fields}(c\langle\overline{G}\rangle) = \overline{H\ f}}}{\texttt{methods}(c\langle\overline{G}\rangle) = \overline{T\ m\ (\overline{G'\ x})\{\cdots\}}}$$
$$\emptyset \vdash \texttt{typeof}(\overline{\iota_f}, \sigma) <: \overline{H}$$
$$\texttt{typeof}(\overline{\iota_m}, \sigma) = (\overline{G'}) \to T$$
$$\nu; \sigma \vdash (c\langle\overline{G}\rangle, \overline{f : G \mapsto \iota_f}, \overline{m \mapsto \iota_m})\ \texttt{OK}$$

$$\frac{\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash T\ (\overline{G\ x})\ \Rightarrow e : (\overline{G}) \to T}{\nu; \sigma \vdash T\ (\overline{G\ x})\ \Rightarrow e\ \texttt{OK}}$$

$$\nu; \sigma \vdash o_{\texttt{null}}\ \texttt{OK}$$

Figure 15. Well-formed objects.

LEMMA 4.1 (Progress). ***If*** $\nu; \sigma; \emptyset; \Gamma_{base} \vdash e : T$ ***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 in the structure of the typing derivation $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e : T$. We here show a sketch of two of the cases and refer to the proof in Coq for further details.[5]

- Rule [T-Read] where $\nu; \sigma; \emptyset; \Gamma_{\text{base}} \vdash e.p : T$: If e is a value ι, 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 $\langle\nu, \sigma, \iota.p\rangle \longrightarrow \langle\nu, \sigma, \iota'\rangle$ for some ι', corresponding to condition b. Otherwise, we can apply the induction hypothesis to get either $\langle\nu, \sigma, e\rangle \longrightarrow \langle\nu', \sigma', e'\rangle$ 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 Γ_{base}, y must be null, so rule [E-Null] applies, hence condition b is satisfied. (As discussed above, variables x and this have been substituted earlier by rules [E-Call] and [E-New].) □

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 ei-

[5] The proof (http://www.brics.dk/fletch/) contains some unproven ('Admitted') lemmas, which are all at the level of what is typically also omitted in non-mechanically-checked proofs. For example, some of these lemmas involve preservation of environment well-formedness under execution, which is tedious to prove in Coq but intuitively not surprising and easy to check manually using the formalization in the paper.

ther lead to a expression whose type is a subtype of T or to an acceptable error.

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

1) σ' `OK` ***and*** $\sigma' \vdash \nu'$ `OK` ***and*** e', ν', *and* σ' *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) $\nu'; \sigma' \vdash e'$ `ACCEPTABLE_ERROR`.

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

- Rule [E-VAR-READ] where $\langle \nu, \sigma, \tau \rangle \longrightarrow \langle \nu, \sigma, \iota \rangle$: Condition 1 trivially holds, since the environments do not change during the evaluation of τ. Since the environments are well formed we have $\nu(\tau) = (G, \iota)$, so the type T of τ is $\texttt{typeof}(\tau, \nu) = G$. The type T' of ι is $\texttt{typeof}(\iota, \sigma)$, and $\sigma \vdash \nu$ `OK` implies that $\texttt{typeof}(\iota, \sigma)$ is a subtype of G, so condition 2a holds.
- Rule [E-VAR-WRITE] where $\langle \nu, \sigma, \tau = \iota \rangle \longrightarrow \langle \nu', \sigma, \iota \rangle$: The rule can only be applied if $\texttt{typeof}(\iota, \sigma)$ is a subtype of $\texttt{typeof}(\tau, \nu)$, so the update from ν into $\nu' = \nu[\tau \mapsto \iota]$ preserves environment well-formedness, so condition 1 is satisfied. Condition 2a holds because $\tau = \iota$ is well typed with type T, rule [T-ASSIGN] gives that ι has the same type T, and subtyping is reflexive.
- The congruence rule $\langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle \Rightarrow \langle \nu, \sigma, e.p \rangle \longrightarrow \langle \nu', \sigma', e'.p \rangle$: 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 $\texttt{accessor}(\texttt{bound}_\Delta(T_1), p) = T$. If e' is an acceptable error in the environments ν' and σ' 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 $\iota_{\texttt{null}}$ 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 `foverride` and `moverride` in the message-safety type system (Figure 10) we have $\texttt{accessor}(\texttt{bound}_\Delta(T_2), p) = T'$ where T' is a subtype of T. Rule [T-READ] $e'.p$ then has type T', meaning that condition 2a holds.

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. □

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 9 were included. As an example, consider the execution step $\langle \nu, \sigma, \tau_B = [\![A, \iota_C]\!].f \rangle \longrightarrow \langle \nu, \sigma, \tau_B = \iota_C.f \rangle$ (applying a congruence rule and [E-RETURN]) in environments where the type of τ_B is B, the type of ι_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 τ_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). 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 4.3 (Message-safety soundness). ***If*** $\nu; \sigma; \emptyset; \Gamma_{base} \vdash e : T$ ***and*** σ `OK` ***and*** $\sigma \vdash \nu$ `OK` ***and*** e, ν, ***and*** σ *do not contain* `dynamic` ***and*** $\sigma_{base} \subseteq \sigma$ ***and*** $\langle \nu, \sigma, e \rangle \longrightarrow^* \langle \nu', \sigma', e' \rangle$ ***and*** e' *is a normal form* ***then***

a) e' *is a value* ***or***

b) $\sigma'; \nu' \vdash e'$ `ACCEPTABLE_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 $\langle \nu, \sigma, e \rangle \longrightarrow^* \langle \nu', \sigma', e' \rangle$, applying lemma 4.1, lemma 4.2, 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 [SUB-FUN$_f$] (Figure 7) requires subtyping for function return types but only assignability for function parameters. This means that subtyping is not transitive, even when `dynamic` is not used. As an example, we may have an expression of type $T_1 = (\texttt{int}) \to \texttt{int}$, which in one step evaluates to an expression of type $T_2 = (\texttt{Object}) \to \texttt{int}$, which in turn evaluates to an expression of type $T_3 = (\texttt{String}) \to \texttt{int}$. 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 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

Although the focus of this paper 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.

5.1 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[6] 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*, *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 2.3. 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):

```
class ElementX extends Element {
  AnalyzableElement get analyzableElement {
    ...
  }
}

class CompilationUnitElementX
        extends ElementX
        implements CompilationUnitElement {
  Element get analyzableElement => ...;
}
```

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.[7]

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

```
abstract class _StreamImpl<T> extends Stream<T> {
  _BufferingStreamSubscription<T> _createSubscription(
      void onData(T data),
      Function onError,
      void onDone(),
      bool cancelOnError) {
    ...
  }
}

class _ControllerStream<T> extends _StreamImpl<T> {
  StreamSubscription<T> _createSubscription(
      void onData(T data),
      Function onError,
      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>`.

[6] The implementation and all benchmarks are available at `http://www.brics.dk/fletch/`.

[7] Discussions on the *dart2js* forum (`https://groups.google.com/a/dartlang.org/forum/#!topic/compiler-dev/DAcnoauqNNQ`) confirm this conclusion.

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));
  }
}
```

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

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

```
typedef ir.Node
         SubbuildFunction(
         IrBuilder builder);
abstract class IrBuilderMixin<N> {
  SubbuildFunction subbuild(N node) {
    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.Node 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` $\rightarrow$ `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*:

```
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 $(\mathtt{E},\mathtt{E}) \rightarrow \mathtt{E}$. The `reduce` method parameter has type $(\mathtt{T},\mathtt{T}) \rightarrow \mathtt{T}$ in `NumberEnumerable<T extends num>`, where `T` $<:$ `num`, and the type of the actual argument is $(\mathtt{num},\mathtt{num}) \rightarrow \mathtt{num}$, and the runtime type is $(\mathtt{num},\mathtt{num}) \rightarrow \mathtt{num}$. Since `T` $<:$ `num`, but `num` $\not<:$ `T`, we have that $(\mathtt{T},\mathtt{T}) \rightarrow \mathtt{T} <: (\mathtt{num},\mathtt{num}) \rightarrow \mathtt{num}$, and $(\mathtt{num},\mathtt{num}) \rightarrow \mathtt{num} \not<: (\mathtt{T},\mathtt{T}) \rightarrow \mathtt{T}$ so requirement 2(b) is violated. We can remove these warnings by changing the program as follows.

```
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 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.

5.2 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 7) 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 [18].

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

```
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 3.4, the program will fail at runtime at the assignment to `fb` since the result of the `downcast` invocation has type $() \to$ `A` that is not a subtype of $() \to$ `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 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 variant of Featherweight Java by Mackay et al. [16] 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 [16] 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 [16] or in the original work on Featherweight Java [14].

In R4RS [6] 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* [5, 26]. 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 [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 systems[8], 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 [12] 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 [20] uses conventional type annotations extended with '?', which corresponds to the Dart `dynamic` type. It builds on $\mathrm{Ob}_{<:}$ [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 [8, 22], 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 [27] 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 [23] and by Siek and Wadler [21]. 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* [19], 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. 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 [15], 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* [25] 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.

[8] `https://www.dartlang.org/slides/2011/11/stanford/dart-a-walk-on-the-dart-side.pdf`

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 We appreciate the comments and suggestions from Gianluca Mezzetti, Thomas Heinze, and the anonymous reviewers. This work was supported by the Danish Research Council for Technology and Production, by Google Faculty Research Award grants, and by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation program (grant agreement No 647544).

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] W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, July 1991.

[7] W. R. Cook. A proposal for making eiffel type-safe. In *Proc. 3rd European Conference on Object-Oriented Programming*, pages 57–70, 1989.

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

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

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

[11] Ecma International. *Dart Programming Language Specification, ECMA-408*, December 2015.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

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

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

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

[21] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 365–376, New York, NY, USA, 2010. ACM.

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

[23] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. Programming Languages and Systems, 18th European Symposium on Programming (ESOP)*, pages 1–16, 2009.

[24] 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`.

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

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

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