Managing Gradual Typing with Message-Safety in Dart

Erik Ernst [*] [†] Anders Møller [*] Mathias Schwarz [‡] Fabio Strocco

Aarhus University, Denmark

{eernst,amoeller,schwarz,fstrocco}@cs.au.dk

Abstract

This paper establishes a notion of *message-safe programs* as a natural intermediate point between dynamically typed and statically typed Dart programs.

Unlike traditional static type checking, the type system in the Dart programming language is unsound by design. 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. We show that, despite unsoundness, judicious use of type annotations can ensure useful properties of the runtime behavior of Dart programs. This supports evolution from a dynamically typed program to a strictly statically typed form.

We present a formal model of Dart that elucidates how a core of the language and its standard type system works. This allows us to characterize message-safe programs and present a theorem stating that such programs will never cause 'message not understood' errors, which is generally not guaranteed for Dart programs that pass the standard type checker. The formal model has been expressed in Coq.

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

General Terms Languages

Keywords Dart, type systems

1. Introduction

Most mainstream object-oriented languages are statically typed, with well-understood 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, 10, 13, 16–18, 21, 22]. The Dart programming language [9] 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 does not support type soundness in the traditional sense: even for fully annotated programs, the static type checker may miss some type-related errors. However, the type system has been designed in a principled manner, and it does guarantee interesting properties at runtime.

Support for evolution of programs from a dynamically typed form into a statically typed form is valuable. To organize this software evolution process in a more structured manner, we propose *message-safe programs* as an intermediate form between one that passes the standard Dart type checker, and one that is type correct according to a traditional, sound type system.

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 [12] and based on the most recent language specification [9]. We state a theorem that message-safe programs will not incur 'message not understood' errors (technically, `NoSuchMethodError` exceptions) at runtime.

The contributions of this paper are as follows:

- We present a core calculus of Dart called *Fletch*, specifying its syntax, dynamic semantics, and typing, thereby elucidating how the core of the Dart language works.
- 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 evolution from dynamically typed to message-safe programs, we outline a generalization of message-safety from complete programs to program fragments.

[*] Supported by Google Faculty Research Award grants.

[†] Current affiliation: Google, Inc.

[‡] Current affiliation: CLC Bio.

- We give a specification of the system in Coq, including a partial proof of the soundness property that message-safe programs do not cause ‘message not understood’ errors.[1]

The paper is organized as follows: Section 2 presents our underlying conceptual analysis, defines the notion of message-safe programs, and describes a practical approach to structure the transformation of a program from an untyped to a typed form. Next, Section 3 introduces our formalization of a core of Dart, and Section 4 presents the soundness results. Finally, Section 5 discusses related work and Section 6 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 [11] and C# [7], and others with JavaScript [8]. Although the Dart language is primarily aimed at web programming, it is a general purpose language, and our results are applicable independently of any application domain. The language is class based, and objects do not change class nor add or remove members during their lifetime, which positions the language near the Java style of mainstream object-orientation; the family resemblance is also strong in the area of 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, which positions it closer to many dynamically typed languages, e.g., JavaScript. That connection is underscored by the fact that one of the main techniques used to execute Dart programs is to translate them into JavaScript programs. A native Dart virtual machine is also available, but currently not widely deployed [19].

The Dart language offers a useful trade-off between dynamically and statically typed object-orientation, and its type system deserves a more in-depth discussion.

2.2 The Dart Approach to Typing

We consider two kinds of type-related errors:

- A *‘message not understood’* error may occur at object property (field or method) lookup operations, e.g., 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, calls, and return operations (note that these are the operations with an associated data-flow), e.g., `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, calls, 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 traditional 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 [15]: 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 [13]. 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 three main 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.

[1] The proof, `http://cs.au.dk/~fstrocco/fletch/qpalds.html`, is partial in the sense that the current Coq source code include a small set of unproven (’Admitted’) but plausible lemmas. Technically, the top level results have been reduced to those unproven lemmas.

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, 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 in checked mode
```

The lack of transitivity makes assignability quite inconvenient to work with in a formal model. The reason is that, in general, a non-transitive typing relation fails to preserve relations over multiple steps. We illustrate how this invalidates the typical line of reasoning in a type soundness proof:

Assume that we consider a variable declaration with an associated initialization expression, T `x` $= e$, and that we have a proof that e has the type S, which is a subtype of T. Typing succeeds, because it is allowed to initialize `x` with a value whose type is a subtype of the declared type T. Now assume that one more 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 S', which is a subtype of S. At this point, the standard proof (of the type preservation part of soundness) proceeds to use the transitivity of subtyping to conclude that T `x` $= e'$ is type correct. However, without transitivity, we cannot conclude that T `x` $= e'$ is type correct.

This kind of phenomenon breaks the standard line of reasoning about type soundness. Hence, our formal model has been created such that it avoids the concept of assignability. Interestingly, we have succeeded in obtaining our soundness result using an even 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 3.6).

2.3 Message-Safe Programs

Under which conditions can a Dart programmer be certain that her 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.

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

1. Every field, method argument, and local variable has an explicitly declared type, and this type is never `dynamic`.
2. Type checking the program produces no static type warnings[2] using the standard Dart type checker with the following modifications:
   (a) The type signature of an overriding method must be identical to that of the overridden method.
   (b) Subtyping among function types requires contravariant argument types and covariant return types.
   (c) If `f` is the name of a field declared in a class `C` and also inherited from its superclass `D` then the type of `f` in `C` must be a subtype of the type of `f` in `D`.
   (d) An expression `e` of type `Function`, `Object`, or `dynamic` cannot be invoked (as in `e(`$\texttt{e}_1$`..`$\texttt{e}_k$`)`).

[2] The notion of a *static type warning* in the Dart specification corresponds to a type error in most other languages; the point is that the type checker rejects the program, but it is possible to run the program anyway.

One of our key contributions is to demonstrate that these requirements suffice, as shown in the following sections. 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. Similarly, a 'message not understood' error arises if the number of arguments is wrong. For simplicity, in order to stay close to traditional OO typing, and in order to prepare for strict static typing, we have not allowed for covariant return types nor for arbitrary argument types in this requirement, but that would be an easy generalization. Requirement 2(b) is motivated by the same line of reasoning as 2(a), once again staying close to traditional function type rules and omitting the easy generalization to allow arbitrary argument types. Requirement 2(c) could be loosely described as "field overriding must be covariant"; overriding for fields makes sense because all accesses use getters and setters. Requirement 3(d) is necessary because these particular cases implicitly introduce the `dynamic` type, and the arity is left unchecked. Clearly, it would not be hard to implement a checker that decides for any given Dart program whether it is message-safe or not.

2.4 Message-Safety and Nominal Identity

A useful intuition about message-safe programs is that they make programmers decide on a specific choice of the meaning of every property (method or field) that is used in the program. More concretely, for every property lookup (e.g., `x.f`) in such a program, the declared type of the receiver object (`x`) ensures that the property (`f`) is defined. Since the 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 provides the location in the source code where the property is defined and where the documentation about how to use or redefine this property should reside.

Of course this is all informal, and the property documentation may be absent or misleading, 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 (such as "any declaration with the right name"), we believe that the static commitment to a nominal identity (which is essentially the same thing as the static commitment to one particular declaration with the right name) is a powerful tool for clarification of the intended use and semantics of that property, and for communication among developers, thus helping them toward writing well-understood and correct software.

Late binding may cause the invocation of a method, e.g., `x.m(y)`, to execute a method implementation in a subclass, `D`, of the one, `C`, that contains the statically known implementation of `m` for that invocation. However, even though the runtime value of `m` is from `D`, the documentation about `m` is likely to be found in `C` or one of its superclasses. Hence, the statically known declaration of `m` still serves as a commitment to a specific nominal identity for `m`, also when late binding is taken into account.

2.5 Message-Safety for Program Fragments

The notion of message-safety also makes sense for program fragments, not only for complete programs. In fact, such a generalization is almost trivial in most cases. Consider a property access expression on the form `x.f` or `x.m(...)` 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, the relevant fragment of the program is the enclosing class and all its subclasses; the subclasses must be inspected in order to verify that there is no subclass that violates requirement 2(c) with respect to the field `x`. For an access expression applied to a returned value, e.g., `x.m(...).f` or `x.m(...).n(...)`, we must check that the return type of `m` declares the requested property. 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 dynamic type of a closure cannot be inspected without calling it), which makes it possible to treat them just like objects when considering message-safety.

Although we focus on complete programs in this paper, this suggests that the concept of message-safety can be generalized to a modular setting. From a software engineering point of view, a developer who is working with a large program could then use such 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.

2.6 A Two-Step Approach toward Type Safety

The Dart language specification [9, page 112] 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 not focus on sound type systems in this paper. Instead, we observe that message-safe programs constitute an intermediate form between dynamic typing and sound static typing, 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 features at property lookup operations where 'message not understood' errors may occur, and a global step that considers subtype constraints at assignments and other data-flow 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:

```
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 features 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;
  }
}
bool 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, and in this paper we will focus on message-safe programs.

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 intermediate message-safe program may raise 'subtype violation' type errors at runtime, but it will not raise 'message not understood' errors. The conceptual significance of this is that, in message-safe programs, *the type annotations justify the actual property lookups* but *there is no guarantee that the flow of data conforms to those type annotations*.

3. Fletch

Fletch 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 specify two distinct type systems for Fletch: the *standard type system*, which, being the core of the Dart type system, elucidates 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 the caption of each figure to produce the two variants.

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

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

The calculus supports a 'typeless' style of programming: 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 Sect. 2.3.

As is typical for such calculi, many features have been omitted, e.g., general statements. A notable omission is conditional expressions ($b?e_1{:}e_2$), which would require union types in order to be integrated into Fletch. Apart from a couple of trivial syntactic abbreviations and some extensions needed to describe runtime states, Fletch is a syntactic and semantic subset of the Dart language, such that Fletch programs can easily be adapted to run on a Dart interpreter, with the same behavior.

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 $a_1, ..., a_n$, $n \geq 0$ of elements denoted by a; $\texttt{nodup}(\overline{a})$ indicates that the list $\overline{a}$ contains no duplicates.

Expressions (e) specify computations, including variable and property lookups, 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.

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. We use '$\hookrightarrow$' to indicate a partial function.

A class table CT is *well-formed* iff $\texttt{Object} \notin dom(\mathsf{CT})$, but every other class name used in CT is defined, and inheritance is acyclic. A Fletch *program* is a tuple (CT, e) where CT is a class table and e is an expression, and it is *well-formed* iff CT is well-formed and both e and all expressions within all classes in CT contain only well-formed types (cf. Section 3.7) 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 an immutable versus a mutable 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 argument.

We model the heap by the maps denoted by the metavariable 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 — a steadily growing map — modeling 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

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

Figure 2. Semantic entities.

variables are aliased across all nested scopes for each invocation of a method, but distinct for distinct 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 C_1' 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 use the following shorthands: $\nu[\tau \mapsto \iota]$ stands 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 $\nu(\tau) = (G, \iota)$.

We also introduce objects, closures, field maps, and method maps. An object o contains its runtime type N (a class applied to a suitable type argument list), a map ϕ from field names to 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 have all been replaced by variable locations, and $\texttt{this}$ has been replaced during object creation by a variable location $\tau_{\texttt{this}}$.

Notationally, $[\tau/y]e$ denotes standard capture avoiding substitution in a Fletch expression e: all free occurrences

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

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

$$[\textsc{Field-Dynamic}]\ \texttt{ftype}(\texttt{dynamic}, f) = \texttt{dynamic}$$

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

$$[\textsc{Method-Super}]\ \frac{\begin{array}{c}\mathsf{CT}(c) = \texttt{class}\ c\texttt{<}\overline{X \lhd \ldots}\texttt{>}\ \texttt{extends}\ N \ldots\ \{\ldots \overline{M}\} \\ \texttt{mtype}([\overline{G_1}/\overline{X}]N, m) = G_2 \qquad m \notin dom(\overline{M})\end{array}}{\texttt{mtype}(c\texttt{<}\overline{G_1}\texttt{>}, m) = G_2}$$

$$[\textsc{Method-Dynamic}]\ \texttt{mtype}(\texttt{dynamic}, m) = \texttt{dynamic}$$

Figure 3. Lookup definitions.

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 'empty' map).

The state of a Fletch program during execution is represented by s (see Figure 2). The class table, CT, is frequently consulted during execution, however, it is constant throughout any program execution, so we will generally leave it implicit, as is common in object calculi since Featherweight Java [12].

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

3.3 Auxiliary Functions

Figure 3 defines field and method type lookup by the functions `ftype` and `mtype`. This will be used for expression typing as in other object calculi, but it will also be used in the operational semantics for checked mode. Hence, we need to introduce them at this point, before we discuss the semantics. The only nonstandard element of `ftype` is the treatment of the receiver type `dynamic` where all field names are considered to be defined and having the type `dynamic`. Similarly, the only nonstandard part of `mtype` is that a receiver of type `dynamic` is considered to have all methods, each of which also has the type `dynamic`.

3.4 Dynamic Semantics

We specify the dynamic semantics of Fletch in terms of a small-step operational semantics that relates States to States, that is, each configuration is a triple $\langle \nu, \sigma, e \rangle$. Figure 4 shows the rules for expression evaluation in Fletch. Every expression evaluates to a heap location ι, which is the only kind of values that Fletch supports. Expression evaluation may have side effects in terms of updates to the heap or the variable environment.

The Dart language includes getter and setter methods. They can be explicitly declared, but otherwise for each declared field the compiler automatically provides a getter and a setter, and for each method a getter. 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`. Since Dart does not introduce any significant novelties about getters and setters, we only model automatically generated getters and setters. For simplicity we do this by means of primitive field read/write and method read operations. This does differ from the language specification, but it is a faithful model of the core of the language.

Evaluation of a variable location [E-VAR-READ] amounts to a lookup in ν for a location τ. Assignment to a variable location 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

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

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

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

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

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

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

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

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

Figure 4. Computational rules for expressions in Fletch. The boxed premises involving `typeof` are omitted for production mode execution, but included for checked mode execution.

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]) 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 $\tau_{\mathtt{this}}$ 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 3. We use `name` to extract the field names and method names, that is, $\mathtt{name}(F) = f$ for a field declaration $F = G\ f$ and $\mathtt{name}(M) = m$ for a method declaration $M = T\ m(\overline{G\ x})\ \{\mathtt{return}\, e;\ \}$.

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.

$$[\textsc{Err-NullRead}]\ \nu;\sigma \vdash \iota_{null}.p\ \texttt{ERROR}$$

$$[\textsc{Err-NullWrite}]\ \nu;\sigma \vdash \iota_{null}.f = \iota\ \texttt{ERROR}$$

$$[\textsc{Err-Var-Write}]\ \frac{\begin{array}{c}\tau \in dom(\nu)\\ \vdash \texttt{typeof}(\iota,\sigma) \not<: \texttt{typeof}(\tau,\nu)\end{array}}{\nu;\sigma \vdash \tau = \iota\ \texttt{ERROR}}$$

$$[\textsc{Err-Field-Write}]\ \frac{\begin{array}{c}\sigma(\iota_1) = (N,\phi,\mu)\\ f \mapsto (S,\iota') \in \overline{F}\\ \texttt{typeof}(\iota_2,\sigma) = T \qquad \vdash T \not<: S\end{array}}{\nu;\sigma \vdash \iota_1.f = \iota_2\ \texttt{ERROR}}$$

$$[\textsc{Err-Call}]\ \frac{\begin{array}{c}\texttt{typeof}(\iota,\sigma) = (\overline{G}) \to T\\ \texttt{typeof}(\iota_i,\sigma) = S \qquad \vdash S \not<: G_i\end{array}}{\nu;\sigma \vdash \iota(\overline{\iota})\ \texttt{ERROR}}$$

$$[\textsc{Err-Return}]\ \frac{\vdash \texttt{typeof}(\iota,\sigma) \not<: T}{\nu;\sigma \vdash [\![T,\iota]\!]\ \texttt{ERROR}}$$

Figure 5. Acceptable runtime errors in message-safe programs.

$$\texttt{typeof}(\iota,\sigma) = \begin{cases} \bot & \text{if } \iota = \iota_{\texttt{null}}\\ c\texttt{<}\overline{G}\texttt{>} & \text{if } \sigma(\iota) = (c\texttt{<}\overline{G}\texttt{>},\phi,\mu)\\ (\overline{G}) \Rightarrow T & \text{if } \sigma(\iota) = T\ (\overline{G\ x}) \Rightarrow e \end{cases}$$

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

Figure 6. 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.

Figure 5 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 $\nu;\sigma \vdash e$ `ERROR`; the latter is any other stuck configuration.

We omit the associated congruence rules, both in Figure 4 and in Figure 5, as they are entirely unsurprising.

3.5 Typing Support for Evaluation

As Figure 4 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 6 shows the definition of `typeof`, which takes a heap location ι or a variable location τ and determines the requested type.

$$[\textsc{ST-Bottom}]\ \Delta \vdash \bot \ll T$$

$$[\textsc{ST-Dynamic}]\ \Delta \vdash T \ll \texttt{dynamic}$$

$$[\textsc{ST-Var}]\ \Delta \vdash X \ll \Delta(X)$$

$$[\textsc{ST-Covariance}]\ \frac{\Delta \vdash \overline{G_1} \ll \overline{G_2}}{\Delta \vdash c\texttt{<}\overline{G_1}\texttt{>} \ll c\texttt{<}\overline{G_2}\texttt{>}}$$

$$[\textsc{ST-Cls}]\ \frac{CT(c) = \texttt{class}\ c\texttt{<}\overline{X \lhd N}\texttt{>}\ \texttt{extends}\ N\ \{...\}}{\Delta \vdash c\texttt{<}\overline{G}\texttt{>} \ll [\overline{G}/\overline{X}]N}$$

$$[\textsc{ST-Refl}]\ \Delta \vdash T \ll T$$

$$[\textsc{ST-Trans}]\ \frac{\Delta \vdash T_1 \ll T_2 \qquad \Delta \vdash T_2 \ll T_3}{\Delta \vdash T_1 \ll T_3}$$

Figure 7. Specific typing definition.

A type environment Δ is a finite map from type variables to class types. We use the notation $X_1 <: N_1, ..., X_n <: N_n$ 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.

3.6 Subtyping

Specific typing is a partial order on types. We say that $\Delta \vdash T_1 \ll T_2$ if T_1 is a more specific type than T_2 in the type environment Δ, as defined in Figure 7. Note that the rules follow the declared `extends` relationship between classes in the program, but leaves several special cases to subtyping (defined below), e.g., a rule [Sub-Dyn-Sub] that makes `dynamic` a subtype of all other types.

Type rules for specific typing do not describe the full subtype relation for Fletch types. The language has a special type annotation `dynamic` that 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 typing rules for `dynamic` 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.

$$[\textsc{Sub-Dyn-Sub}]\ \frac{\Delta \vdash \texttt{dynsub}(T_1) \ll T_2}{\Delta \vdash T_1 <: T_2}$$

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

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

$$[\textsc{Sub-Object}]\ \Delta \vdash (\overline{G_1}) \to T_1 <: \texttt{Object}$$

Figure 8. Subtyping definition. The standard type system (most closely modeling Dart) and the operational semantics use [$\textsc{Sub-Fun}_s$], and the message-safe type system uses [$\textsc{Sub-Fun}_f$]. The boxes just point out the differences (cf. Section 2.3, requirement 2(b)).

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

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

With $\texttt{dynsub}(T)$, we can define the subtype relation as shown in Figure 8. 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 9 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>();}
}
```

3.7 Type Well-Formedness

Figure 10 defines what it means for a type T to be well-formed in a type environment Δ, written $\Delta \vdash T$ OK. Note that type well-formedness requires subtyping for type parameters rather than assignability: if we have a class definition `class` $c\texttt{<}X \lhd \texttt{String}\texttt{>} \{\ldots\}$ then $c\texttt{<Object>}$ is not a well-formed type, since X must be a subtype of `String`. Type well-formedness is used in the top-level rules for definition typing (Figure 13) and class typing (Figure 14).

$$[\textsc{Assign-Up}]\ \frac{\Delta \vdash T_1 <: T_2}{\texttt{assignable}_\Delta(T_1, T_2)}$$

$$[\textsc{Assign-Down}]\ \frac{\Delta \vdash T_2 <: T_1}{\texttt{assignable}_\Delta(T_1, T_2)}$$

$$[\textsc{Assign-Gen}]\ \frac{\texttt{assignable}_\Delta(\overline{G_1}, \overline{G_2})}{\texttt{assignable}_\Delta(c\texttt{<}\overline{G_1}\texttt{>}, c\texttt{<}\overline{G_2}\texttt{>})}$$

Figure 9. Assignability.

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

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

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

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

Figure 10. Well-formed types.

3.8 Expression Typing

The typing judgment $\nu; \sigma; \Delta; \Gamma \vdash e : T$ indicates that the expression e is well typed with the type T in the environments ν, σ, Δ and Γ. Here, ν maps variable locations to heap locations, σ maps heap locations to objects or closures, Δ maps type variables to their upper bounds, and Γ maps variables to their declared types. When type checking a program, ν and σ will be empty, but they are required for type checking a program state during execution, i.e., in the soundness proof. The initial environments for an execution are $\nu_{\text{base}} = \emptyset$, $\sigma_{\text{base}} = [\iota_{\texttt{null}} \mapsto o_{\texttt{null}}]$, $\Delta_{\text{base}} = \emptyset$, and $\Gamma_{\text{base}} = \{\texttt{null} : \bot\}$.

The Fletch type systems differ from the Dart type system in a couple of ways. In particular, in Figure 12 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.

$$\frac{\texttt{ftype}(G, f) = H}{\texttt{accessor}(G, f) = H}$$

$$\frac{\texttt{mtype}(G, m) = H}{\texttt{accessor}(G, m) = H}$$

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

$$\frac{\begin{array}{c}\texttt{mtype}(m, \texttt{bound}_{\Delta}(N)) = (\overline{H}) \to S \\ \text{implies } \boxed{\overline{G} = \overline{H}}\ \boxed{T = S}\end{array}}{\texttt{moverride}_{\Delta}(m, N, (\overline{G}) \to T)}$$

Figure 11. Auxiliary definitions. Boxed parts enforce properties required in message-safe programs (cf. Section 2.3, requirement 2(a) and 2(c)).

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

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

Figure 13. Definition typing.

Finally, the message-safe variant of many rules encode some requirements specific to message-safe programs, e.g., that a method can only override another method if they have the same signature.

So far, the standard type system only differs from the message-safe type system in unsurprising ways, introducing more strict requirements in the areas where Dart typing has the greatest degree of built-in unsoundness. However, as we mentioned in Section 2 there is a conflict between the use of assignability and proofs of soundness, which is the reason why the assignability premises are boxed in Figure 12. These premises are included in the standard type system, but omitted in the message-safe type system. This may come as a surprise because this makes message-safe typing more *flexible* than standard typing, rather than more strict.

In fact, it only makes message-safe typing more flexible during some steps of execution. If standard typing would have failed at some point — e.g., because the type of an actual argument to a method invocation is not a subtype of the declared argument type — then message-safe typing will also fail at the point where the method is invoked, but it will allow the evaluation of the actual arguments to proceed to that point, whereas standard typing would fail as soon as an actual argument obtains a type that is not a subtype of the formal argument type. No errors are suppressed, but they may be detected later when using the message-safe type system.

The reason why the relaxation of the message-safe type system is necessary in the first place is that the standard type system "predicts" errors much earlier, and it is then not true that computation can proceed all the way until an easily recognizable error configuration has been reached. This means that the progress property only holds if specified in a complex manner (that does not offer any additional insight, and by the way does not match the behavior of an actual implementation where argument evaluation would also be allowed to finish before an error is detected). As we shall see, usage of the message-safe type system actually produces a soundness result in terms of the standard type system as an easy corollary.

The rules [T-VAR], [T-READ], [T-WRITE], and [T-ASSIGN] are unsurprising apart from the assignability checks, which allow some types to be both subtypes and supertypes where typical type systems would require a subtype. When assignability is omitted, even unrelated types are allowed.

The rule [T-CALL] is also unsurprising, apart from the fact that it allows for supertypes (with assignability) or unrelated types (without assignability) for the actual arguments. The [T-NEW] rule is very simple because mutability allows us to omit constructors. [T-FUNCTION] is also standard, noting that the list $\overline{G\ x}$ 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.

Figure 11 defines a few auxiliary functions: `accessor` is a convenient short-hand for property lookup, `foverride` defines requirements on redeclaring a field in a subclass, and `moverride` defines requirements on method overriding. The last two predicates use the $\texttt{bound}_{\Delta}(T)$ function. It replaces all the type variables occurring in the type T to their upper bound as defined in the type environment Δ.

Finally, Figures 13 and 14 show the top-level rules for typing of classes that causes all the other elements of type checking to be applied.

4. Properties of Fletch

Soundness is traditionally associated with Milner's phrase *well-typed programs can't go wrong*, but we need to allow for subtype violation errors, whereas message not understood must be ruled out. We have defined two type systems

$$[\textsc{T-Var}]\ \nu;\sigma;\Delta;\Gamma \vdash y : \Gamma(y) \qquad [\textsc{T-Read}]\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : T \qquad \texttt{accessor}(\texttt{bound}_{\Delta}(T), p) = G}{\nu;\sigma;\Delta;\Gamma \vdash e_0.p : G}$$

$$[\textsc{T-Write}]\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : T \qquad \texttt{accessor}(\texttt{bound}_{\Delta}(T), f) = G \qquad \nu;\sigma;\Delta;\Gamma \vdash e_1 : S \qquad \boxed{\texttt{assignable}_{\Delta}(S, G)}}{\nu;\sigma;\Delta;\Gamma \vdash e_0.f = e_1 : S}$$

$$[\textsc{T-Assign}]\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : T \qquad \boxed{\texttt{assignable}_{\Delta}(T, \Gamma(x))}}{\nu;\sigma;\Delta;\Gamma \vdash x = e_0 : T}$$

$$\boxed{\textsc{T-Dynamic-Call}_s}\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : T \qquad T \in \{\texttt{Object}, \texttt{Function}, \texttt{dynamic}\} \qquad \nu;\sigma;\Delta;\Gamma \vdash \overline{e} : \overline{T}}{\nu;\sigma;\Delta;\Gamma \vdash e_0(\overline{e}) : \texttt{dynamic}}$$

$$[\textsc{T-New}]\ \dfrac{\Delta \vdash N\ \texttt{OK}}{\nu;\sigma;\Delta;\Gamma \vdash \texttt{new}\ N() : N}$$

$$[\textsc{T-Call}]\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e_0 : (\overline{G}) \to T \qquad \nu;\sigma;\Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \boxed{\texttt{assignable}_{\Delta}(\overline{S}, \overline{G})}}{\nu;\sigma;\Delta;\Gamma \vdash e_0(\overline{e}) : T}$$

$$[\textsc{T-Function}]\ \dfrac{\Delta \vdash \overline{G}\ \texttt{OK} \qquad \nu;\sigma;\Delta;\Gamma, \overline{x : G} \vdash e_0 : S \qquad \boxed{\texttt{assignable}_{\Delta}(S, T)}}{\nu;\sigma;\Delta;\Gamma \vdash \boxed{\mathrm{T}}\ (\overline{G\ x}) \Rightarrow e_0 : (\overline{G}) \to T}$$

$$[\textsc{T-Runtime-Loc}]\ \nu;\sigma;\Delta;\Gamma \vdash \boxed{\iota} : \texttt{typeof}(\iota, \sigma) \qquad [\textsc{T-Runtime-Frame}]\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e : S \qquad \boxed{\texttt{assignable}_{\Delta}(S, T)}}{\nu;\sigma;\Delta;\Gamma \vdash \boxed{[\![T, e]\!]} : T}$$

$$[\textsc{T-Runtime-VLoc}]\ \nu;\sigma;\Delta;\Gamma \vdash \boxed{\tau} : \texttt{typeof}(\tau, \nu)$$

$$[\textsc{T-Runtime-VAssign}]\ \dfrac{\nu;\sigma;\Delta;\Gamma \vdash e : T \qquad \boxed{\texttt{assignable}_{\Delta}(T, \texttt{typeof}(\tau, \nu))}}{\nu;\sigma;\Delta;\Gamma \vdash \boxed{\tau = e} : T}$$

Figure 12. Expression typing. Boxed elements in conclusions are extensions relative to the Dart language, boxed premises (all on assignability) are included in the standard type system and omitted in the message-safe type system. 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 2(d)).

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

Figure 14. Class typing.

for Fletch, and in this section we shall use the message-safe type system. As usual, the main step on the way to a soundness proof is progress and preservation:

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

- *e is a value,* ***or***
- *$\langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle$ for some ν', σ', e',* ***or***
- *$\nu;\sigma \vdash e$* `ERROR`

Proof By induction on the typing derivation $\nu;\sigma;\emptyset;\Gamma_{\text{base}} \vdash e : T$. The Coq sources contain a proof of this lemma using several unproven but plausible lemmas. $\square$

LEMMA 4.2 (Preservation). ***If*** *e does not contain* `dynamic`*, $\nu;\sigma;\emptyset;\Gamma_{base} \vdash e : T$, σ* `OK`*, $\sigma \vdash \nu$* `OK`*, $\langle \nu, \sigma, e \rangle \longrightarrow$*

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

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

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

Figure 15. Well-formed objects.

$\langle \nu', \sigma', e' \rangle$ *ν and σ do not contain* `dynamic`, ***and*** $\sigma_{base} \subseteq \sigma$ ***then*** *both of the following hold:*

- σ' OK, $\sigma' \vdash \nu'$ OK, *ν' and σ' do not contain* `dynamic`, ***and***
- $\nu';\sigma';\emptyset;\Gamma_{base} \vdash e' : S$, $\emptyset \vdash S <: T$ *for some S* ***or***
- $\nu';\sigma' \vdash e'$ ERROR, $\sigma_{base} \subseteq \sigma'$

Proof Induction on the derivation $\langle \nu, \sigma, e \rangle \longrightarrow \langle \nu', \sigma', e' \rangle$. The Coq sources contain a proof of this lemma using several unproven but plausible lemmas. □

In these lemmas, 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.

From these lemmas we can obtain the soundness result:

THEOREM 4.3 (Type soundness). ***If*** *e does not contain* `dynamic`, $\nu;\sigma;\emptyset;\Gamma_{base} \vdash e : T$, σ OK, $\sigma \vdash \nu$ OK, $\sigma_{base} \subseteq \sigma$, *$\nu$ and σ do not contain* `dynamic`, $\langle \nu, \sigma, e \rangle \longrightarrow^* \langle \nu', \sigma', e' \rangle$, ***and*** *$e'$ is a normal form* ***then*** *either*

- *e' is a value,* $\nu';\sigma';\emptyset;\Gamma_{base} \vdash e' : T'$, ***and*** $\emptyset \vdash T' <: T$, ***or***
- $\sigma';\nu' \vdash e'$ ERROR

and in both cases σ' OK, $\sigma' \vdash \nu'$ OK, ***and*** $\sigma_{base} \subseteq \sigma'$.

Proof By induction on $\langle \nu, \sigma, e \rangle \longrightarrow^* \langle \nu', \sigma', e' \rangle$. See the proof in Coq for further details. □

The connection back to the standard type system is easy to establish: it is trivial to see that when a typing exists in the standard type system then by omitting assignability a typing is derived in the message-safe type system. This means that soundness holds for all programs which are standard typable, because the final result is a value, whose typing is identical in the two type systems. The Coq sources contain a small corollary proving this result. In summary, we have demonstrated that Fletch does ensure the fundamental property that message-safe programs will never encounter a 'message not understood' error.

5. Related Work

The Featherweight Java formalization [14] specifies a core of Java with mutable references. We have used that formalization as inspiration for the overall approach in the creation of our Coq formalization of Fletch; all the details are very different, of course.

Many papers present approaches to typing that allow for more flexibility than traditional, sound type systems. We briefly present the most influential ones, and position our work relative to each of them.

An early approach which aims to reconcile the flexibility of dynamic typing with the safety of static typing is *soft typing* [5], later complemented by [21]. The basic idea is that expressions whose type do not satisfy the requirements by the context are wrapped in a type cast, thus turning the static type error into a dynamic check. The Dart concept of assignability makes the same effect a built-in property of the dynamic semantics. Strongtalk [4] is an early system with a similar goal, supporting very expressive (but not statically decidable) type specifications for Smalltalk. The Dart type system may have inherited the trait of being optional from there. *Pluggable type systems* [3] are optional type systems that may be used with its target language as needed. The Dart language has been designed to enable the use of pluggable type systems[3], 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* [10] 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 checking. Given that this is concerned with strict static typing *enhanced* with dynamic predicates, there is little overlap with Dart typing. *Gradual typing* [17] uses conventional type annotations extended with '?', which corresponds to the Dart `dynamic` type. It builds on $\mathbf{Ob}_{<:}$ [1] (i.e., it uses structural type equivalence and does not include recursive types), and hence the foundations differ substantially from Fletch. Their notion of *type consistency* does not have a corresponding concept in Fletch nor in Dart, but is replaced by our inclusion of `dynamic` in the subtype rules. *Contracts* allow for general computation (and hence, no static checking) in Scheme [6, 18], with a special emphasis on tracking blame for first-class functions that only reveal typing violations when invoked. Neither Fletch nor Dart support contracts, but in a sense they are not needed because the type of first-

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

class functions can be checked dynamically. *Like types* [22] were introduced recently, where usage of a like typed variable is checked statically, but it is checked dynamically that the value of such a variable actually supports the operations applied to it. It could be claimed that the point of this work is to support structural typing to some extent, and no such support is present in Dart — checked mode checks will fail for an assignment to an unrelated type, no matter whether the object in question would be able to respond to the messages actually sent. Another recent paper presents *progressive types* [16], letting Racket programs tune the typing to allow or prevent certain kinds of runtime errors. Our work is slightly similar, in the sense that it enables programmers to rule out a particular kind of run-time type errors. Finally, TypeScript [2] enables optional type annotations in JavaScript programs. Using structural types and coinductive subtype rules, the foundations differ substantially from Dart and Fletch.

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

Success typing is a way to design complete but unsound type systems [13], 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* [20] 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.

6. 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 in Dart never encounter 'message not understood' errors.

This result provides new insights into the design space between dynamic and static typing. In future work, we plan to implement tool support to guide Dart programmers toward type safe programs via message-safe programs. Also, we believe Fletch and our formalization in Coq may be useful in further studies of Dart and related programming languages.

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 *Proceedings of 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 *Proceedings of OOPSLA*, pages 215–230, 1993.

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

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

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

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

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] Personal communication. In relation to a Google Faculty Awards research effort, involving Gilad Bracha and the Google Team in Aarhus, Denmark, 2013–2014.

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

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

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

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

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

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

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