Getting F-Bounded Polymorphism into Shape

Ben Greenman
Cornell University
blg59@cornell.edu

Fabian Muehlboeck
Cornell University
fabianm@cs.cornell.edu

Ross Tate
Cornell University
ross@cs.cornell.edu

Abstract

We present a way to restrict recursive inheritance without sacrificing the benefits of F-bounded polymorphism. In particular, we distinguish two new concepts, *materials* and *shapes*, and demonstrate through a survey of 13.5 million lines of open-source generic-Java code that these two concepts never actually overlap in practice. With this *Material-Shape Separation*, we prove that even naïve type-checking algorithms are sound and complete, some of which address problems that were unsolvable even under the existing proposals for restricting inheritance. We illustrate how the simplicity of our design reflects the design intuitions employed by programmers and potentially enables new features coming into demand for upcoming programming languages.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Inheritance, Polymorphism

General Terms Algorithms, Design, Languages

Keywords Materials, Shapes, Separation, Subtyping, F-Bounded Polymorphism, Variance, Decidability, Joins, Higher-Kinded Types

1. Introduction

Generics were a long-awaited addition to Java and C#. They finally gave industry developers access to the benefits of polymorphism*. But polymorphism was not originally designed for object-oriented languages, rather it was tried and tested primarily in functional languages [14]. There is one fundamental difference between typical instances of these language classes: subtyping. The design and algorithms for polymorphism were centered around unification [1], a technique that only works smoothly in type systems without subtyping. Yet subtyping is a key part of Java, C#, Scala, and numerous other object-oriented languages, and the question of how to combine polymorphism and subtyping has yet to arrive at a solution that is capable of expressing the various idioms used in practice while still providing sound and complete algorithms for type checking. Here we provide the foundations of such a solution, one based on reshaping F-bounded polymorphism [4] to properly match how it is used in practice.

Plain bounded polymorphism is the ability to specify the range of types a type variable can represent. Typically this is done with an upper bound, i.e. a constraint indicating what classes/interfaces instantiations of a type variable must implement. This allows the programmer to guarantee the presence of various methods, such as requiring a type variable to extend `Formattable` so that the programmer can safely use the `format` method. Thus bounded polymorphism enables programmers to impose the same requirements and guarantees on type arguments that they can impose on function parameters and returns.

F-bounded polymorphism is the ability to constrain a type variable by a type expressed in terms of the type variable itself [4]. In other words, F-bounded polymorphism is the ability to use recursive constraints. This subtle addition significantly increases the power of type-variable constraints. In particular, F-bounded polymorphism addresses the issue of *binary methods*, the pattern that operations such as comparison and addition need both arguments to have the same type. With F-bounded polymorphism, one can require a type parameter `T` to extend `Comparable<T>`, where `Comparable<T>` has a comparison method that only accepts arguments of type `T`. This way types such as `Integer` and `String` can be compared to themselves but not to other types that happen to also have a comparison method. Java's equality design does not adopt this paradigm, instead declaring equality to exist between all objects. Consequently, the type checker cannot help identify cases where the wrong types of objects are being compared, and most implementations of `equals` have to first cast its parameter to the correct type.

The main drawback of F-bounded polymorphism is that it requires inheritance to be recursive. For example, `String` implements `Comparable<String>`, so the inherited type is defined in terms of the inheriting type itself. On its own, this is a simple feature, but generics typically also have some form of *variance*. That is, a `List<String>`† can safely be treated as a `List<Object>`, an ability that is rather useful in practice, and consequently languages such as C# and Scala provide a way for programmers to declare that `List` is *covariant* [8, 15]. Dually, something that is comparable to arbitrary objects can safely be compared to integers, making `Comparable` *contravariant*. Unfortunately, the combination of variance and recursive inheritance greatly complicates many type-checking algorithms. Indeed, Kennedy and Pierce proved that even just subtyping is undecidable in languages supporting these two features [11].

Our key insight is that we can recover decidability and algorithmic simplicity by restricting recursive inheritance to how it is actually used in practice. We call the classes/interfaces used for recursive inheritance, such as `Comparable`, *shapes* because they describe the higher-level shape of the type using recursive inheri-

* In this paper, we use polymorphism to refer to parametric polymorphism and not subtype polymorphism.

† We use `List` to represent some read-only list interface.

PLDI'14, June 09–11, 2014, Edinburgh, United Kingdom

ACM 978-1-4503-2784-8/14/06…$15.00
`http://dx.doi.org/10.1145/2594291.2594308`

tance. What we recognize is that shapes are used in a very restricted fashion in practice. In particular, shapes are never used in parameter types, return types, field types, and type arguments. Instead, we call the classes/interfaces used in those locations *materials*, because they are the types actually used for material exchanges across the components of a program. Our fundamental finding is that, should one require materials and shapes to be disjoint sets, the 13.5 million lines of generic-Java code we analyzed would be unaffected except for where the analysis identified flaws in the designs. We call this observed property *Material-Shape Separation*.

With this newfound understanding of industry code, we are able to formalize a decidable type system that is backwards compatible with Java as it exists in practice. The key insight is that most algorithms need only be defined on materials, since shapes are only ever used as constraints. Because shapes encapsulate all recursive inheritance, inheritance amongst materials is well founded, so even naïve strategies are guaranteed to terminate. With this, we solve open problems such as computable joins, and we solve them with simple, direct, and efficient machinery.

In summary, we make the following contributions:

Section 2 anecdotal evidence suggesting that Material-Shape Separation is already an unrecognized idiom

Section 3 a type-theoretic formalization of materials and shapes and Material-Shape Separation

Section 4 a large survey of industry code demonstrating the compatibility of Material-Shape Separation with practice

Section 5 type-checking algorithms exploiting Material-Shape Separation to achieve simplicity and decidability

Section 6 potential applications of Material-Shape Separation to open type-checking challenges and new type-system features

We illustrate how our findings may enhance related work in Section 7, following with high-level lessons from our experiences in Section 8.

2. Background

Polymorphism and subtyping make a powerful combination, and as such both have been widely adopted by statically-typed major industry languages. They also make for a troublesome combination, as Kennedy and Pierce have shown that even subtyping with variant generics is undecidable without restriction [11]. Consequently, Kennedy and Pierce provided various restrictions that ensure decidability, the most notable of which is banning expansive inheritance, which has been adopted by C# [8]. But that solution requires a complicated algorithm, has poor blame properties, and does not work for more powerful systems like Java's wildcards [26]. Tate et al. proposed an alternative restriction that guarantees decidability for wildcards and uses a more efficient algorithm [20], but their restriction is less accommodating of contravariance. Regardless of which one might be better, both solutions grew from algorithmic perspectives, recognizing current practice only insofar as to show backwards compatibility with existing code. Thus, their acceptability is conditioned on there not being any compelling counterexamples. However, the following interface is such a compelling counterexample to both solutions:

```
interface List<out E>
          extends Equatable<List<Equatable<E>>> {}
```

Here the definition uses the `Equatable` interface to express type-safe equality. Equality is a binary method, and so modern object-oriented practice suggests it be formulated using F-bounded polymorphism and recursive inheritance. Thus the signature guarantees that all lists implement type-safe equality. Ideally, we would require that lists of `E` are equatable only when `E` extends `Equatable<E>`; however, most modern languages do not support such conditional inheritance, a feature we will discuss in more detail in Section 6.1. We bypass this limitation by making `List<E>` be equatable to lists of `Equatable<E>`, which will only exist when `E` extends `Equatable<E>`. Also, when `E` extends `Equatable<E>`, then `List<E>` will actually be a subtype of `Equatable<List<E>>` due to the covariance of `List` (hence the `out` annotation on the type parameter `E`) and the contravariance of `Equatable` (typically expressed with an `in` annotation). Thus, `List<String>` will be equatable with itself and so can be used as, say, the type of keys for hash maps, which require equality to be defined on their keys. Consequently, this design presents a solution to the important open problem of type-safe equality on lists. In fact, we know of no alternative solution to this problem using just the expressiveness of Java or C#'s generics.

This solution is rejected by both of the existing proposals to restrict generics for decidability. It uses expansive inheritance by having `List<E>` use `List<Equatable<E>>` in its inherited type, thereby violating Kennedy and Pierce's requirement [11]. It also uses nested contravariance, with `Equatable` being used at a non-covariant position in the inherited type, thereby violating Tate et al.'s requirement [20]. Yet this design is being rejected for reasons that industry developers would view as purely academic. In other words, the common case is being sacrificed for the corner case. So, to design a more practical restriction to generics, one must better understand the common case.

To that end we presented this design to our industry collaborators, and to our surprise they were strongly opposed to it. Despite the lack of any type-safe alternatives, and even admitting they found it to be a clever exploitation of features, they rejected it because they felt like it violated unwritten, and up to that point unrecognized, design principles. In particular, to them `Equatable` is only meant to describe types via constraints; it is not something to be passed around in lists. Using `Equatable` as a type argument violates the accepted use of the interface. From this we developed the concept of shapes, e.g. `Equatable`, and materials, e.g. `List`, and we designed a type system and typing algorithms based on the separation of these two concepts, i.e. the *Material-Shape Separation*.

Using Material-Shape Separation, we were able to develop a simple sound and complete subtyping algorithm, one capable of incorporating type equivalence even in invariant types, a problem raised by Tate et al. [20] and not well addressed by any of the existing proposals for restricting generics. More importantly, we developed a sound and complete algorithm for computing the join of two types, a problem raised by Smith and Cartwright [18] and also not well addressed by any of the existing proposals. We could even add higher-kinded constrained type variables and type lambdas with pointwise higher-kinded subtyping [17] and still maintain decidability of all these features. Finally, to justify that all this was indeed compatible with widespread industry practice and not just limited to our collaborators, we surveyed 13.5 million lines of open-source generic-Java code and found no violations of our design assumptions. Thus we have a decidable type system, with simple and efficient algorithms, that matches hitherto-unwritten design principles of industry practitioners.

3. Materials and Shapes

In this section, we define materials and shapes in full detail. This section culminates with the formalization of Material-Shape Separation, the key observation enabling the algorithms presented in Section 5. But first, we must establish the formal setting we are working within.

When discussing generics, variance is an important challenge. In Section 2, we used declaration-site variance, which is used by

C# and Scala [8, 15]. However, here we will be using use-site variance, a simplification of Java wildcards [6]. Tate discusses the relationships between these systems [19], but for this paper one need only understand that use-site variance is more expressive than declaration-site variance and discards the implicit constraints of wildcards, since the complications of implicit constraints far outweigh their usefulness [20].

In our simplified formalism, classes/interfaces $\mathcal{C}$ have exactly one type parameter; all the rules, algorithms, and proofs will be extended to arbitrary type parameters in Section 5.5. More importantly, when supplying a type argument to a class/interface, one provides both an `in` bound and an `out` bound. In terms of arrays, the `in` bound is what can be put into the array, and the `out` bound is what can be taken out of the array. Formally, the `in` bound is the argument to the contravariant portion of the class/interface and the `out` bound is the argument to the covariant portion of the class/interface. We also use $\bot$ and $\top$ as the subtype and supertype of all types. That way Java's $\mathcal{C}\langle$`? extends` $\tau\rangle$ can translate to $\mathcal{C}\langle$`in` $\bot$ `out` $\tau\rangle$, and $\mathcal{C}\langle$`? super` $\tau\rangle$ to $\mathcal{C}\langle$`in` τ `out` $\top\rangle$.

3.1 Materials

Materials are the classes/interfaces exchanged between separate components of a program and stored within the components of a program. More formally, they are the classes/interfaces that are used as parameter and return types for functions/methods/constructors as well as types of fields. Consequently, most classes/interfaces are materials.

Supposing $\mathcal{M}$ is the subset of classes/interfaces $\mathcal{C}$ that are materials, we define the grammar of our parameterized types as follows:

$$\dot{\tau} ::= \bot \mid \top \mid \mathcal{M}\langle \texttt{in}\ \dot{\tau}\ \texttt{out}\ \dot{\tau}\rangle \mid \cdot$$

The $\cdot$ represents the single parameter of the inheriting type; a complete discussion of type variables appears in Section 5.4.

Observation that we make parameterized types $\dot{\tau}$ be comprised of only materials. However, any class/interface can inherit any other; only the type arguments are restricted to materials. Thus we formalize inheritance as a relationship of the following form:

$$\mathcal{C}\langle\cdot\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle$$

We do not impose a grammar for specifying inheritance, rather we leave that to the language and assume it provides one. Consequently, we demand the following three properties in order to accurately model inheritance:

Transitivity
$$\begin{array}{c} \mathcal{C}\langle\cdot\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle \quad \wedge \quad \mathcal{C}'\langle\cdot\rangle <:: \mathcal{C}''\langle\dot{\tau}''\rangle \\ \Downarrow \\ \mathcal{C}\langle\cdot\rangle <:: \mathcal{C}''\langle\dot{\tau}''[\cdot \mapsto \dot{\tau}']\rangle \end{array}$$

Finiteness For all classes/interfaces $\mathcal{C}$ and $\mathcal{C}'$, the set of parameterized types $\dot{\tau}'$ such that $\mathcal{C}\langle\cdot\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle$ holds is finite.

Acyclicity There is no $\mathcal{C}$ and $\dot{\tau}'$ such that $\mathcal{C}\langle\cdot\rangle <:: \mathcal{C}\langle\dot{\tau}'\rangle$ holds.

Typically this relationship will be derived from some simpler one via transitive closure, but we require transitivity in order to simplify many of our formalisms. Nonetheless, with a little care one can easily reformulate our system for a non-transitive inheritance relationship. On a related note, we will use $\leqslant::$ to denote the reflexive closure of $<::$.

3.2 Shapes

Shapes capture the recursive aspects of inheritance and are the reason we need F-bounded polymorphism [4] rather than just plain bounded polymorphism. For example, the common Java interface `Comparable` is a shape because classes such as `Integer` implement `Comparable<Integer>`, a type defined in terms of the class/interface inheriting it. In current practice, only a few classes/interfaces are shapes, but those classes/interfaces are often used widely throughout the project.

```
interface Graph<G extends Graph<G,E,V>,
                E extends Edge<G,E,V>,
                V extends Vertex<G,E,V>> {
    List<V> getVertices();
}
interface Edge<G extends Graph<G,E,V>,
               E extends Edge<G,E,V>,
               V extends Vertex<G,E,V>> {
    G getGraph();
    V getSource();
    V getTarget();
}
interface Vertex<G extends Graph<G,E,V>,
                 E extends Edge<G,E,V>,
                 V extends Vertex<G,E,V>> {
    G getGraph();
    List<E> getIncoming();
    List<E> getOutgoing();
}

class Map extends Graph<Map,Road,City> {...}
class Road extends Edge<Map,Road,City> {...}
class City extends Vertex<Map,Road,City> {...}
```

Figure 1. A type family for graphs, edges, and vertices

From our observations, shapes arise in practice for two main reasons. The primary one is to encode a form of self types [2]. That is, the type parameter of the shape is meant to represent the type implementing that shape. This is useful for binary methods, such as comparisons and equalities, as well as algebraic operations, such as addition, negation, and multiplication. Negation is an important example because it illustrates that self types are not just used for binary methods.

The second use of shapes is type families [5]. A type family is a codependent group of classes/interfaces. A classic example is graphs, edges, and vertices. A graph consists of edges and vertices; edges connect vertices and reside within a graph; and vertices have connecting edges and reside within a graph. The challenge is designing this group such that when one extends it, say with mutability, then all components can refer to the other components and know they are also mutable. To accomplish this with shapes, each interface takes three type parameters, one for graphs, one for edges, and one for vertices, and all bounded to indicate so. Extensions of the type family then impose additional constraints on the type parameters to indicate the guaranteed additional functionality. We illustrate this design pattern in Figure 1.

To formalize the recursive nature of inheritance with shapes, we first define a labeled graph describing how classes/interfaces are used in inheritance:

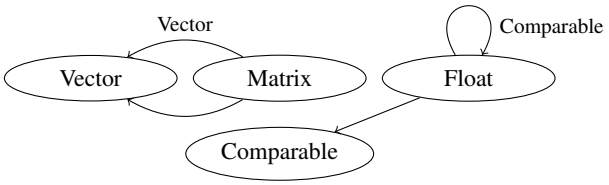
$$\frac{\mathcal{C}\langle\cdot\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle}{\mathcal{C} \to \mathcal{C}'} \qquad \frac{\mathcal{C}\langle\cdot\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle \quad \mathcal{C}'' \text{ occurs in } \dot{\tau}'}{\mathcal{C} \xrightarrow{\mathcal{C}'} \mathcal{C}''}$$

If one were to require classes/interfaces to inherit only types that are already defined, then this usage graph would be acyclic, and subtyping can be proven decidable by using a topological ordering of the classes/interfaces. However, in a system with recursive inheritance, such a topological ordering does not exist. Shapes $\mathcal{S}$ are the classes/interfaces such that if the edges labeled with shapes were removed from the usage graph then it would be acyclic. Thus shapes are the classes/interfaces preventing the topological ordering that would make subtyping easily decidable.

As an example, consider the following class declarations.

```
interface Comparable<E> {}
class     Vector<E> {}
class     Matrix<E extends Comparable<E>>
          extends Vector<Vector<E>> {}
class     Float extends Comparable<Float> {}
```

These result in the following usage graph.



The unlabeled edge from `Matrix` to `Vector` is due to the direct extension `class Matrix extends Vector<...>` and the labeled edge is due to `Vector<X>` being the type argument to `Vector<...>` in that extension. The `Float` class has a self loop labeled `Comparable`, creating a cycle in the usage graph containing `Comparable`, indicating that `Comparable` is a shape. Note that the constraint on `Matrix`'s parameter `E` has no effect on this graph; we discuss the role of type variables in Section 5.4.

3.3 Separating Materials and Shapes

While it is theoretically possible to have a class/interface be used as both a material and a shape, our aforementioned interaction with developers suggests there is a natural tendency to keep these two patterns separate. Here we formalize that assumption.

Material-Shape Separation. *Let $\mathcal{M}$ be all classes/interfaces used as type arguments. For some set $\mathcal{S}$ of classes/interfaces such that removing all edges labeled with an element of $\mathcal{S}$ from the usage graph results in an acyclic graph, $\mathcal{M}$ and $\mathcal{S}$ are disjoint.*

Although this formalization does not need $\mathcal{M}$ and $\mathcal{S}$ to cover $\mathcal{C}$, it is convenient to simply define $\mathcal{M}$ as all non-shapes. In this way, unless a programmer explicitly declares a class/interface to be a shape, they are free to use that class/interface without restriction outside the class/interface hierarchy.

Under this design, our `List` example is still rejected, but for more intuitive reasons. First, the designer would specify that `Equatable` is a shape. Then, when defining `List`, our system would indicate that `Equatable` cannot be used as an argument to `List` due to being a shape. Hence the cause and effect are clear to the designer, who can then focus on finding a type-safe alternative. Regrettably, this leaves our problem with `Equatable` unsolved, which we defer to future work as discussed in Section 6, but it prevents programmers from creating unconventional designs, assuming that such designs are indeed unconventional, which we verify in the next section.

4. Industry Compatibility

To support our claim that Material-Shape Separation captures an industry-wide idiom, we present the findings of our scientific inquiry into current practices. Over 13.5 million lines of generic-Java code across a total of 60 open-source projects, taken primarily from the Qualitas Corpus [21], show no alarming cases where separation was broken. A table of all projects we analyzed and some relevant statistics we collected can be found in the technical report [7]. Projects ranged in scale and function from the jFin finance library to the massive NetBeans IDE, the median size being approximately 60,000 lines of code. As such, our sample set contains a wide range of styles and design principles. Nevertheless, the projects conformed to our system, suggesting that one can enforce Material-Shape Separation in existing languages, as well as in new ones, without breaking compatibility with existing code bases.

4.1 Methodology

After forming our collection of projects, we modified the source code of `openjdk` to generate the usage graphs of Section 3.2 from the classes/interfaces of each project. From these graphs, we extracted the labels of the edges constituting simple cycles. These labels formed our set of shapes $\mathcal{S}$, and all other classes/interfaces formed our set of materials $\mathcal{M}$. Another compiler pass then searched for occurrences, if any, of shapes being used as materials, thereby violating Material-Shape Separation.

4.2 Findings

Barring a few caveats discussed below, the entire body of 60 projects never violated Material-Shape Separation. In fact, every shape we encountered was either an encoding of self types or type families, as we had expected. The type family we encountered happened to be precisely for representing graphs. In the `findbugs` project, interfaces `GraphVertex` and `GraphEdge`, and classes `AbstractVertex` and `AbstractEdge`, constituted type families at the interface level, and at the class level, similar to the design in Figure 1. Some custom shapes, `hadoop`'s `WritableComparable` as well as `findbugs`'s `AnnotationEnumeration`, are simple extensions of `Comparable`. In fact, `WritableComparable` is actually just an encoding of an intersection type, which will be discussed in Section 5.2. As for self types, `Comparable` and `Enum` are the two incorporated into Java's libraries and are the most widely used. Moreover, the remaining nine remaining shapes were all custom applications of self types. All counted, there were 17 shapes in total, listed in the technical report [7], none of which were used as materials.

Caveats

The above statements make a few simplifications, namely eliding technicalities caused by programmer errors and Java limitations. First, there were uses of the shapes outside of inheritance and type-variable constraints. However, all these uses were in the form of raw types (except in one case where the type argument was simply an unconstrained wildcard, thereby not utilizing the type argument). That is, the programmers used shapes as materials only when bypassing Java's type system, sacrificing type safety. These were either results of poor utilization of generics (e.g. failing to use F-bounded polymorphism in order to ensure type safety) or involved casting wherein Java can only enforce raw types due to type erasure [6].

Because none of these uses of shapes as materials actually used their type argument, it is still possible to incorporate them into our system. For each shape, we can associate a new parameterless material inherited by the shape. This material is not inherited recursively, so it is not a shape. We can substitute all the above raw (or wildcarded) misuses of the shape with the new parameterless material inherited by that shape. Thus, since the arguments of shapes are never used in the code bases, through this encoding they still all satisfy Material-Shape Separation. Regardless, it is better to view these few instances as abuses of the type system rather than as reflective of design principles.

The second caveat is due to the following class in `openjdk`:

```
public class Env<A> implements Iterable<Env<A>> {}
```

Because of this one class, we originally inferred `Iterable` to be a shape, even though this inheritance clause is never actually made use of by the code base nor exposed by the API and so should not have been present. `Iterable` is used widely as a material, so this

inference caused many false alarms, demonstrating the danger of inferring shapes rather than having them be explicitly identified by the programmer.

4.3 Ceylon

One might be surprised by how few shapes we discovered in use: roughly one shape per million lines of code. However, every shape had a key and distinct role in its respective architecture design. We simply have recognized these as special cases and classified their distinction. Nonetheless, one might worry that our observations may not persist over more designs given the limited sample we draw our conclusions from here. Similarly, our observations might only apply to Java because of the burden Java imposes upon using generics. To address this issue, we have adapted our analysis to Ceylon, a language recently designed and released by Red Hat that fully embraces generics. Self types and type families are directly supported by Ceylon, and Ceylon uses shapes to support features such as operator polymorphism [3]. Thus, shapes appear much more frequently in Ceylon than in Java, providing a denser sample.

We presented Material-Shape Separation and our corresponding results to the Ceylon team. They found the analysis and applications compelling and simple enough that within a day they had implemented a branch of their compiler that enforced Material-Shape Separation. They decided to treat precisely the self types and type families as shapes instead of using our inference technique. They used the modified compiler on all the committed code that had been developed in the language, either by the designers implementing core modules or by contributors adding new modules to the open-source project, and found only one counterexample to Material-Shape Separation. This counterexample was a labeled-tree design similar to the problematic `Tree` example to be discussed in Section 5.1. It was a quickly-drafted practical implementation of a JSON API, and its design was already in contention at the time. Furthermore, this instance is easily resolved by adding a `children` attribute to the class in place of the extension clause, similar to the example we include in Section 6. The designers have continued to confirm that unconstrained programmers still naturally adhere to Material-Shape Separation even with their more expressive type system. Their current stance is that they will likely integrate Material-Shape Separation into Ceylon 2.0.

5. Applications

Having introduced the formal definitions of materials and shapes and demonstrated their compatibility with existing code bases, we now describe how we can exploit our newfound Material-Shape Separation to design simple, sound, and complete type-checking algorithms. This section presents five results immediately realizable through shapes: the decidability of subtyping, the support for non-syntactic type equivalence, the existence of joins, the ability to constrain type variables, and the incorporation of higher-kinded types. In Section 6, we will discuss additional existing challenges and new features we hope to address in future work by extending the techniques we present here.

5.1 Decidability of Subtyping

Recall the example `List` design:

```
interface List<out E>
          extends Equatable<List<Equatable<E>>> {}
```

`javac`[‡] handles most uses of this design correctly. However, this design violates both Kennedy and Pierce's and Tate et al.'s restrictions on generics [11, 20], and consequently we can use it to cause `javac` to stack overflow.

[‡] When we refer to `javac` we mean the OpenJDK `1.7.0_25` type checker.

Consider the following use of the `List` design:

```
class Tree extends ArrayList<Tree> {}
```

In one line, it implements a mutable unlabeled tree. Furthermore, since `ArrayList` implements `List`, we also get the correct equality implementation for trees with no additional effort. But upon actually equating two trees, `javac` throws a `StackOverflowError`.

Understanding why the type checker fails is crucial to understanding the surprising challenges behind generics. To check a use of the equality operation, the type checker needs to verify that the left type implements `Equatable` of the right type. Here this reduces to checking that `Tree` is a subtype of `Equatable<Tree>`. This simple question evolves into the following infinite progression of subtyping reductions:

$$\begin{array}{c}
\texttt{Tree} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{ArrayList<Tree>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{List<Tree>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{Equatable<List<Equatable<Tree>>>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{contravariance}) \\
\texttt{Tree} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{ArrayList<Tree>} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{List<Tree>} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{covariance}) \\
\texttt{Tree} <: \texttt{Equatable<Tree>} \\
\vdots
\end{array}$$

Note that the final state of the above is the same as the initial state, forming a loop that causes the infinite digression. What is surprising is that this infinite digression corresponds to a *valid* infinite proof of subtyping (refer to Tate et al. for more details [20]). These infinite proofs are what make subtyping so difficult to decide, since they imply that an algorithm can in fact make good progress in each step but still never be able to finish. However, with Material-Shape Separation, all proofs of subtyping are finite, so any such algorithm is guaranteed to terminate.

To demonstrate this, we first formalize *extended* types σ. Extended types are not used in practice, but we can guarantee decidable subtyping even for extended types, so we present them here to provide more informed options for language designers.

$$\sigma := \bot \mid \top \mid \mathcal{C}\langle \texttt{in } \sigma \texttt{ out } \sigma \rangle$$

The primary difference between σ and $\dot{\tau}$ is that σ allows arbitrary classes/interfaces $\mathcal{C}$ rather than just materials $\mathcal{M}$. Consequently, extended types may use shapes, even as type arguments. The intuition behind this is that, for subtyping, our separation of materials and shapes need only be imposed upon the class/interface hierarchy and not on types elsewhere in the program. The second difference is that σ is not parameterized; we will address the issue of type variables shortly in Section 5.4.

Figure 2 formalizes subtyping on extended types. The rule for subtyping classes/interfaces combines inheritance and use-site variance into one step. The subtlety here is substitution, described in the table in Figure 2 above the subtyping rules, which has to deal with the fact that there are two type arguments for a single type parameter. This substitution replaces all contravariant uses of the type parameter with the `in` argument, and all covariant uses with the `out` argument. This technique combines the subtyping and tight-approximation algorithms of Tate [19] into one rule.

The subtyping rules are syntax directed and so specify a sound and complete decision algorithm *provided* we can guarantee the

$$\dot\tau \mapsto \dot\tau[\,\cdot \mapsto \sigma_i; \sigma_o]$$

$$\begin{array}{c}
\bot \mapsto \bot \\
\top \mapsto \top \\
\cdot \mapsto \sigma_o \\
\mathcal{M}\langle \texttt{in}\ \dot\tau_i\ \texttt{out}\ \dot\tau_o\rangle \mapsto \mathcal{M}\langle \texttt{in}\ \dot\tau_i[\cdot \mapsto \sigma_o; \sigma_i]\ \texttt{out}\ \dot\tau_o[\cdot \mapsto \sigma_i; \sigma_o]\rangle
\end{array}$$

$$\boxed{\vdash \sigma <: \sigma}$$

$$\frac{}{\vdash \bot <: \sigma} \qquad \frac{}{\vdash \sigma <: \top}$$

$$\frac{\begin{array}{c}\mathcal{C}\langle\,\cdot\,\rangle \leqslant:: \mathcal{C}'\langle\dot\tau'\rangle \\ \vdash \sigma_i' <: \dot\tau'[\,\cdot \mapsto \sigma_o; \sigma_i] \qquad \vdash \dot\tau'[\,\cdot \mapsto \sigma_i; \sigma_o] <: \sigma_o'\end{array}}{\vdash \mathcal{C}\langle \texttt{in}\ \sigma_i\ \texttt{out}\ \sigma_o\rangle <: \mathcal{C}'\langle \texttt{in}\ \sigma_i'\ \texttt{out}\ \sigma_o'\rangle}$$

Figure 2. Algorithmic subtyping rules

process terminates. Our finiteness assumption on $<::$ prevents infinite branching at any point. Consequently, the only remaining source of non-termination is the potential for infinite proofs, much like in our example earlier. This brings us to our main theorem.

Theorem 1. *Under Material-Shape Separation, all proofs of subtyping as specified in Figure 2 are finite.*

Proof. The major insight is that Material-Shape Separation implies that new uses of shapes are never introduced when applying inheritance in subtyping since shapes can never occur in the type arguments of inherited classes/interfaces. Thus we can define a well-founded two-part measure on extended types σ.

The first part, $\lfloor\sigma\rfloor$ formalized in Figure 3, is the maximum layering depth of shapes in the extended type, where a layer is a shape occurring syntactically inside a type argument to another shape. This part of the measure is completely agnostic to the inheritance hierarchy, since we know that inheritance cannot introduce new layers of shapes so long as it satisfies Material-Shape Separation. Thus recursion in inheritance causes no problems.

The second part, $|\sigma|$ formalized in Figure 3, specifies the maximum number of proof steps that can be taken from any situation where σ is on either side of a subtyping judgement until reaching a shape at the top level (thereby next reducing the first part of the measure) or terminating. The challenge is to prove that this measure is well defined; in other words, the calculation $|\sigma|$ must terminate. This is clear by structural induction *provided* each parameterized measure $M_{\mathcal{M}}$ is well defined. The parameterized measure $M_{\mathcal{M}}$ is a function on measures indicating how applying inheritance affects the measure of a $\mathcal{M}$ type. The key observation is that $M_{\mathcal{M}}$ only uses the parameterized measures of inherited *materials*. Due to Material-Shape Separation, well foundedness of material inheritance enables us to assume that those parameterized measures are already well defined, thereby making $M_{\mathcal{M}}$ well defined.

This two-part measure on types can be adapted into a measure on subtyping judgements. We define the measure of a judgement $\vdash \sigma <: \sigma'$ as the lexicographic ordering $(\lfloor\sigma\rfloor + \lfloor\sigma'\rfloor)$ followed by $(|\sigma| + |\sigma'|)$. One can easily verify that for each rule the measure of the premises is always strictly less than the measure of the conclusion, thereby guaranteeing that any proof will be finite even if infinite proofs were permitted. □

Corollary 1. *Under Material-Shape Separation, subtyping as specified in Figure 2 is decidable.*

What is remarkable about this result is that our subtyping algorithm is more naïve than prior solutions and yet is still both sound and complete under Material-Shape Separation. For example, Kennedy and Pierce's prohibition against expansive inheritance

$$\sigma \mapsto \lfloor\sigma\rfloor : \mathbb{N}$$

$$\begin{array}{c}
\bot \mapsto 0 \\
\top \mapsto 0 \\
\mathcal{M}\langle \texttt{in}\ \sigma_i\ \texttt{out}\ \sigma_o\rangle \mapsto \max(\lfloor\sigma_i\rfloor, \lfloor\sigma_o\rfloor) \\
\mathcal{S}\langle \texttt{in}\ \sigma_i\ \texttt{out}\ \sigma_o\rangle \mapsto 1 + \max(\lfloor\sigma_i\rfloor, \lfloor\sigma_o\rfloor)
\end{array}$$

$$\sigma/\dot\tau \mapsto |\sigma/\dot\tau| : \mathbb{N}/\dot{\mathbb{N}}$$

$$\begin{array}{c}
\cdot \mapsto \cdot \\
\bot \mapsto 0 \\
\top \mapsto 0 \\
\mathcal{M}\langle \texttt{in}\ \sigma_i/\dot\tau_i\ \texttt{out}\ \sigma_o/\dot\tau_o\rangle \mapsto 1 + M_{\mathcal{M}}[\,\cdot \mapsto \max(|\sigma_i/\dot\tau_i|, |\sigma_o/\dot\tau_o|)] \\
\mathcal{S}\langle \texttt{in}\ \sigma_i/\dot\tau_i\ \texttt{out}\ \sigma_o/\dot\tau_o\rangle \mapsto 0
\end{array}$$

$$M_{\mathcal{M}} = \max(\,\cdot\,, \max_{\mathcal{M}\langle\,\cdot\,\rangle <:: \mathcal{M}'\langle\dot\tau\rangle} M_{\mathcal{M}'}[\,\cdot \mapsto |\dot\tau|])$$

We implicitly lift max *and* $1+$ *to parameterized integers.*

Figure 3. Measures for extended/parameterized types

does not prevent infinite proofs; it only ensures all infinite proofs eventually cycle thanks to results from Viroli [27]. Therefore, their algorithm requires keeping a list of all the subtyping judgements that arose earlier in the recursion stack and checking them against the current judgement for syntactic identity before proceeding to process the judgement as usual in order to determine if they are in an infinite cyclic proof [11]. While not as computationally difficult, Tate et al. prevent infinite recursion by treating invariant types as a special case using syntactic unification [20]. Notice that both these approaches rely on syntactic identity, whereas we only use recursion, which brings us to our next contribution.

5.2 Equivalences

Syntactic identity of types can be troublesome for type systems in which there are multiple ways to express the same type. In practice, this has not been a large problem because many existing type systems have the property that all equivalent types are syntactically identical. However, newer and more expressive languages cannot rely on syntactic identity. Tate et al. presented some issues with this flawed assumption in Java, illustrating that semantically equivalent types can be written differently and that consequently `javac` rejects programs due to such shallow syntactic differences [20]. Tate et al.'s own algorithm actually also relies on syntactic identity, so they describe a complex multipass process for canonicalizing types. Ideally such complications would not be necessary because they can be rather brittle and sensitive to changes in the language design. Our system has no such problem. Syntactic identity is never used in our subtyping algorithm, so type equivalences are already incorporated and decidable.

To describe the circumstances more formally, let us suppose we make the following extension to our types:

$$\begin{array}{rcl}
\dot\tau & ::= & \cdots \mid \mathcal{M}\langle \texttt{inv}\ \dot\tau\rangle \\
\sigma & ::= & \cdots \mid \mathcal{C}\langle \texttt{inv}\ \sigma\rangle
\end{array}$$

The `inv` annotation indicates an invariant usage of the type argument. In many systems, this is the default. We previously used only the `in` and `out` arguments because `inv` represents the special case where both arguments are the same; however, existing type systems are more accurately formalized with an `inv` annotation.

In such systems, subtyping is specified with the following additional rules:

$$\frac{\begin{array}{c}\mathcal{C}\langle\,\cdot\,\rangle \leqslant:: \mathcal{C}'\langle\dot\tau'\rangle \\ \vdash \sigma_i' <: \dot\tau'[\,\cdot \mapsto \sigma] \qquad \vdash \dot\tau'[\,\cdot \mapsto \sigma] <: \sigma_o'\end{array}}{\vdash \mathcal{C}\langle \texttt{inv}\ \sigma\rangle <: \mathcal{C}'\langle \texttt{in}\ \sigma_i'\ \texttt{out}\ \sigma_o'\rangle}$$

$$\frac{\mathcal{C}\langle\cdot\rangle \leqslant:: \mathcal{C}'\langle\dot{\tau}'\rangle \quad \dot{\tau}'[\cdot \mapsto \sigma] = \sigma'}{\vdash \mathcal{C}\langle\texttt{inv}\ \sigma\rangle <: \mathcal{C}'\langle\texttt{inv}\ \sigma'\rangle}$$

The second rule uses syntactic identity. This is the status quo in many type systems and algorithms, but it does not interact well with other type features.

To demonstrate the problem, suppose we were to add intersection types. To do so, we would make the following extensions to our system:

$$\begin{array}{rcl} \dot{\tau} & ::= & \cdots \mid \dot{\tau}\,\&\,\dot{\tau} \\ \sigma & ::= & \cdots \mid \sigma\,\&\,\sigma \end{array}$$

$$\frac{\vdash \sigma_i <: \sigma'}{\vdash \sigma_1\,\&\,\sigma_2 <: \sigma'} \qquad \frac{\vdash \sigma <: \sigma'_1 \quad \vdash \sigma <: \sigma'_2}{\vdash \sigma <: \sigma'_1\,\&\,\sigma'_2}$$

With intersection types one can require a field to be both iterable and serializable using the type `Iterable<T>&Serializable`. Such a type is not expressible in Java, and consequently programmers often opt to leave the `Serializable` requirement implicit and manually cast when necessary, somewhat defeating the purpose of a static type system.

Given such a feature, one might eventually obtain an object `Array<Iterable<T>&Serializable>`, where `Array` is an invariant type (we have slipped to declaration-site variance for sake of clarity). Similarly, a function might demand an object of type `Array<Serializable&Iterable<T>>`. The question is whether the former can be used for the latter. The answer seems to be obviously yes, since `&` is a commutative operator, but the type systems and algorithms using syntactic identity would reject such a coercion since the two intersections are written differently. At first this might seem easy to fix, but the problem is subtler than it appears. In particular, `Serializable` and `Iterable` have no direct connection to each other, making this is an easy example. But we could also have the type `Iterable<T>&List<T>`, in which case the left is a supertype of the right and therefore redundant. That is, `Iterable<T>&List<T>` is equivalent to `List<T>`. Thus determining equivalences of intersections relies on subtyping, and determining proper subtyping relies on determining equivalences, producing a circularity.

This troublesome circularity is best illustrated with the following class definition:

$$\texttt{class Foo extends Array<Foo\&Array<Foo>>}\ \{\}$$

Now consider whether `Foo` is a subtype of `Array<Foo>`. The subtyping holds iff `Foo&Array<Foo>` is equivalent to `Foo`, and that equivalence holds iff `Foo` is a subtype of `Array<Foo>`. Thus we have a circular dependency, so we can answer yes to both or no to both and in either case we have a consistent system. This situation is due to the problematic infinite proofs we discussed earlier.

Fortunately, having observed Material-Shape Separation, we recognize that the above example is impractical and need not be addressed. Moreover, our encoding of invariant types replaces the rule using syntactic identity with the following rule:

$$\frac{\mathcal{C}\langle\cdot\rangle \leqslant:: \mathcal{C}'\langle\dot{\tau}'\rangle \quad \vdash \sigma' <: \dot{\tau}'[\cdot \mapsto \sigma] \quad \vdash \dot{\tau}'[\cdot \mapsto \sigma] <: \sigma}{\vdash \mathcal{C}\langle\texttt{inv}\ \sigma\rangle <: \mathcal{C}'\langle\texttt{inv}\ \sigma'\rangle}$$

Hence our system already uses type equivalence rather than syntactic identity. Plus, our strategy for guaranteeing all proofs are finite easily extends to incorporate intersections. Put together, these properties give a sound and complete subtyping algorithm with intersections that uses type equivalence rather than syntactic identity.

5.3 Joins

Whereas our subtyping results applied to extended types, our remaining findings only apply to non-extended types τ:

$$\tau ::= \bot \mid \top \mid \mathcal{M}\langle\texttt{in}\ \tau\ \texttt{out}\ \tau\rangle$$

Given a pair of types τ_1 and τ_2, their join $\tau_1 \sqcup \tau_2$ is their most-precise common supertype. Joins are useful for the type checker, particularly in operations that combine expressions. For example, consider the following program:

```
<T extends Comparable<in T>>
void separate(T middle,
              Iterable<out T> elems,
              ArrayList<in T> smaller,
              ArrayList<in T> bigger) {
   foreach (T elem in elems)
      (elem < middle ? smaller : bigger).add(elem);
}
```

Each element of the list is added to `smaller` or `bigger` depending on how it compares with `middle`. To type check `? :`, though, one needs to combine the types of `smaller` and `bigger` into a common supertype. If the most precise such common supertype is computed, then the subsequent method call `.add(elem)` is rejected only if the program is invalid.

In this case, such a most-precise common supertype seems easy to determine since the types being joined are in fact the same. However, we chose this example because it both arose from practice and broke `javac`. The program, once translated into Java's syntax, is valid but `javac` incorrectly rejects it.

The reason is that `javac` uses an imprecise join algorithm that discards any uses of `? super` (i.e. `in`) in the types being joined. It does so because Java's type system does not have joins, and even when they exist they can be difficult to determine. For this reason, Smith and Cartright proposed simply adding *union* types [18], trivially guaranteeing joins because the rules for union types actually define them as the join of the types being unioned together. However, such a fix is shallow, since then one needs to extend all other type-checking rules to handle union types. For example, Tate et al. demonstrate that Smith and Cartwright's approach does not address capture conversion [20], an important feature for using wildcards with generic methods [26]. Tate et al. instead use lazy existential types as a regrettably complex solution.

As an example of the intricacies of this problems, suppose we need to join together the two simple types `Integer` and `Float`. To simplify matters, further suppose that `Integer` only implements `Summable<Integer>` and `Float` only implements `Summable<Float>`. One common supertype of `Integer` and `Float` is `Summable<?>`, but so is `Summable<out Summable<?>>` and `Summable<out Summable<out Summable<?>>>`, and each is more precise than the one before it. In fact, we can continue this chain forever, demonstrating that there is no most-precise common supertype for this simple practical example. That is, the join of `Integer` and `Float` does not exist.

Now we apply Material-Shape Separation to this problem. Notice that `Summable` is a shape; it appears in two cases of recursive inheritance. Consequently, `Summable<?>` is not a valid type τ in our system because `Summable` is not a material. In fact, none of the above common supertypes are valid types in our system. `Summable` is only permitted in inheritance and type-variable constraints, not as the type of an expression. Thus in our system the join of `Integer` and `Float` is simply $\top$. The following proof demonstrates that all joins are similarly easy to compute in our system, provided we have intersection types.

Theorem 2. *Under Material-Shape Separation, our type system extended with intersection types has computable joins for all types τ_1 and τ_2 with respect to other types τ.*

Proof. The algorithm is the following: (1) if either τ_j is $\bot$, then the join is τ_k where $j \neq k$; (2) if either τ_j is $\top$, then the join is $\top$; (3) if either τ_j is $\tau \,\&\, \tau'$, then the join is $(\tau \sqcup \tau_k) \,\&\, (\tau' \sqcup \tau_k)$ where $j \neq k$; (4) otherwise, each τ_j must be of the form $\mathcal{M}_j\langle \texttt{in}\ \tau_i^j\ \texttt{out}\ \tau_o^j\rangle$, and the join is

$$\bigwedge_{\substack{\mathcal{M}_1\langle\cdot\rangle \leqslant:: \mathcal{M}'\langle\dot\tau_1'\rangle,\\ \mathcal{M}_2\langle\cdot\rangle \leqslant:: \mathcal{M}'\langle\dot\tau_2'\rangle}} \mathcal{M}' \left\langle \begin{array}{l} \texttt{in}\ \ \dot\tau_1'[\cdot \mapsto \tau_o^1; \tau_i^1] \,\&\, \dot\tau_2'[\cdot \mapsto \tau_o^2; \tau_i^2] \\ \texttt{out}\ \dot\tau_1'[\cdot \mapsto \tau_i^1; \tau_o^1] \sqcup \dot\tau_2'[\cdot \mapsto \tau_i^2; \tau_o^2] \end{array} \right\rangle$$

This algorithm can easily be shown to terminate reusing the second component of the measure used for subtyping. Once again, well-foundedness of material inheritance is the critical feature. Note that the large intersection only ranges over inherited materials rather than all classes/interfaces, which is safe to do because other types τ are only comprised of materials. This is how we avoid the issue of recursive inheritance via shapes.

The key step for proving the algorithm correct is proving that joins distribute through intersections. Ignoring $\bot$ and $\top$ at the moment for simplicity, any type τ is essentially of the form $\bigwedge_{\mathcal{L}_i} \tau_i$ where each τ_i is an instantiation of some material and i ranges over some finite number. Given two such types $\bigwedge_{\mathcal{L}_i} \tau_i$ and $\bigwedge_{\mathcal{L}_k} \tau_k''$, it is easy to prove that $\bigwedge_{\mathcal{L}_i} \tau_i <: \bigwedge_{\mathcal{L}_k} \tau_k''$ can only hold if for all k there exists some i such that τ_i is a subtype of τ_k''. So, if $\bigwedge_{\mathcal{L}_k} \tau_k''$ is a common supertype of $\bigwedge_{\mathcal{L}_i} \tau_i$ and $\bigwedge_{\mathcal{L}_j} \tau_j'$, then for each k there exists some i and some j such that τ_k'' is a common supertype of τ_i and τ_j'. Thus $\bigwedge_{\mathcal{L}_k} \tau_k''$ is a common supertype of $\bigwedge_{\mathcal{L}_{i,j}} \tau_i \sqcup \tau_j'$, from which the result follows. □

The reader might take issue with our use of intersection types, which allowed us to avoid computing the *meet*, or least-precise common subtype of two types. Indeed, many languages impose restrictions on multiple inheritance, and unrestricted intersection types can be used to violate invariants that would otherwise hold such as single-instantiation inheritance for arbitrary types. Additional subtleties surrounding intersection types include uninhabitable intersections, which a precise type system would replace with $\bot$. These issues are rather specific to details of a given language design, so their discussion lies outside the scope of this paper, but we have found Material-Shape Separation to be useful in these settings. Here we used unrestricted intersection types because they are necessary for handling arbitrary multiple inheritance.

Note that, although in the case of subtyping we only provided an alternative to existing approaches to guaranteeing decidability, in the case of joins none of those existing approaches guarantee the existence, let alone the computability, of joins. Even the simple example before with `Integer` and `Float` proved problematic in those systems.

5.4 Type Variables and Constraints

So far we have managed to avoid the issue of type variables, a rather important concept given the topic of F-bounded polymorphism. We did so because we can view type variables simply as abstract classes/interfaces. Upper-bound constraints on type variables translate to inheritance clauses on these type variables. There is a technical issue with the use of top-level use-site variance permitted in constraints but not in inheritance clauses, but this is purely grammatical and easy to accommodate. Lower-bound constraints on type variables can sometimes translate to locally adding inheritance clauses to the constraining class/interface.

To illustrate our perspective, recall the code from Section 3.2:

$$\begin{array}{l} \kappa ::= * \mid \langle\overline{\kappa}\rangle \to \kappa \\ \tau ::= \bot \mid \top \mid X \mid \mathcal{M} \mid \tau\langle\overline{\texttt{in}\ \tau\ \texttt{out}\ \tau}\rangle \mid \lambda\overline{X}.\tau \\ \Theta ::= \overline{X : \kappa} \end{array}$$

$\boxed{\Theta \vdash \tau : \kappa}$

$$\frac{}{\Theta \vdash \bot : *} \qquad \frac{}{\Theta \vdash \top : *} \qquad \frac{X : \kappa \in \Theta}{\Theta \vdash X : \kappa}$$

$$\frac{\mathcal{M} : \langle\overline{\kappa}\rangle \to *}{\Theta \vdash \mathcal{M} : \langle\overline{\kappa}\rangle \to *} \qquad \frac{\Theta, \overline{X : \kappa} \vdash \tau : \kappa'}{\Theta \vdash \lambda\overline{X}.\tau : \langle\overline{\kappa}\rangle \to \kappa'}$$

$$\frac{\Theta \vdash \tau : \langle\overline{\kappa}\rangle \to \kappa' \quad \overline{\Theta \vdash \tau_i : \kappa} \quad \overline{\Theta \vdash \tau_o : \kappa}}{\Theta \vdash \tau\langle\overline{\texttt{in}\ \tau_i\ \texttt{out}\ \tau_o}\rangle : \kappa'}$$

Figure 4. Higher-kinded types

```
interface Comparable<E> {}
class     Vector<E> {}
class     Matrix<E extends Comparable<E>>
          extends Vector<Vector<E>> {}
class     Float extends Comparable<Float> {}
```

Inside the body of `Matrix`, the type variable `E` is in scope. Various types will reference `E`, and subtyping will need to take its constraint into account. To integrate this into our formalism, note that if `E` were a class/interface $\mathcal{C}$ with the inheritance clause `E <:: Comparable<E>` then Material-Shape Separation would still hold. Thus, subtyping will still be decidable. If `E` had a lower bound such as `Integer`, then subtyping would still be decidable since adding the inheritance clauses `Integer <:: E` and (transitively required) `Integer <:: Comparable<E>` still satisfies Material-Shape Separation.

Note that a lower bound such as `Vector<Integer>` cannot be translated like above into our formalization of inheritance, so our current proof does not extend to such lower bounds. However, our formalism could be extended to handle such lower bounds by treating them like inheritance clauses when generating the usage graph (extending the definition of Material-Shape Separation) and when defining the measure of variables (extending our proof strategy). The one caveat is that shapes cannot be used in lower bounds for this strategy to work. Indeed, if `Foo` inherited `Shape<Foo>` and `X` had lower bound `Shape<out X>`, then there would be an infinite proof that `Foo` is a subtype of `X`.

5.5 Higher Kinds

With this strategy of viewing type variables as abstract classes/interfaces, we can extend type variables to having higher kinds since class/interface names are essentially higher-kinded types. One could declare a parameterized type variable `C<X>` and require it to extend `Iterable<X>` so that `C` represents some iterable generic class/interface. One could even further constrain `C<X>` to extend `Equatable<C<X>>` so as to ensure this kind of collection comes with a semantics and decision algorithm for equality. One only needs to prove that higher-kinded subtyping [17] is decidable.

Unfortunately, higher-kinded subtyping is not decidable with extended types. To understand why, consider the following definitions (using declaration-site variance for the sake of convenience):

```
shape    Shape<in P : *> {}
material Mayhem<Q : * -> *>
         extends Shape<Q<Mayhem<Q>>> {}
```

$$\boxed{\tau \rightsquigarrow \tau}$$

$$\frac{}{(\lambda \overline{X}.\tau)\langle \overline{\texttt{in } \tau_i \texttt{ out } \tau_o} \rangle \rightsquigarrow \tau[\overline{X \mapsto \tau_i; \tau_o}]}$$

$$\frac{\tau \rightsquigarrow \tau'}{\tau \langle \overline{\texttt{in } \tau_i \texttt{ out } \tau_o} \rangle \rightsquigarrow \tau' \langle \overline{\texttt{in } \tau_i \texttt{ out } \tau_o} \rangle}$$

$$\boxed{\Theta \vdash \tau <: \tau : \kappa \qquad \Theta \vdash \tau <: \mathcal{S} \langle \overline{\texttt{in } \tau \texttt{ out } \tau} \rangle : *}$$

$$\frac{}{\Theta \vdash \bot <: \tau : *} \qquad \frac{}{\Theta \vdash \tau <: \top : *}$$

$$\frac{X : \overline{\langle \overline{\kappa} \rangle \to *} \in \Theta \quad \overline{\overline{\Theta \vdash \tau_i' <: \tau_i : \kappa}} \quad \overline{\overline{\Theta \vdash \tau_o <: \tau_o' : \kappa}}}{\Theta \vdash X \, \overline{\langle \overline{\texttt{in } \tau_i \texttt{ out } \tau_o} \rangle} <: X \, \overline{\langle \overline{\texttt{in } \tau_i' \texttt{ out } \tau_o'} \rangle} : *}$$

$$\frac{\begin{array}{c}\mathcal{M} : \langle \overline{\kappa} \rangle \to * \quad \mathcal{C} : \langle \overline{\kappa} \rangle \to * \quad \mathcal{M} \langle \overline{X} \rangle \leqslant:: \mathcal{C} \langle \overline{\tau} \rangle \\ \overline{\overline{\Theta \vdash \tau_i' <: \tau[\overline{X \mapsto \tau_o; \tau_i}] : \kappa}} \\ \overline{\overline{\Theta \vdash \tau[\overline{X \mapsto \tau_i; \tau_o}] <: \tau_o' : \kappa}}\end{array}}{\Theta \vdash \mathcal{M} \langle \overline{\texttt{in } \tau_i \texttt{ out } \tau_o} \rangle <: \mathcal{C} \langle \overline{\texttt{in } \tau_i' \texttt{ out } \tau_o'} \rangle : *}$$

$$\frac{\tau \rightsquigarrow \hat{\tau} \quad \Theta \vdash \hat{\tau} <: \tau' : *}{\Theta \vdash \tau <: \tau' : *} \qquad \frac{\tau' \rightsquigarrow \hat{\tau}' \quad \Theta \vdash \tau <: \hat{\tau}' : *}{\Theta \vdash \tau <: \tau' : *}$$

$$\frac{\Theta, \overline{X : \kappa} \vdash \tau \langle \overline{\texttt{in } X \texttt{ out } X} \rangle <: \tau' \langle \overline{\texttt{in } X \texttt{ out } X} \rangle : \kappa'}{\Theta \vdash \tau <: \tau' : \langle \overline{\kappa} \rangle \to \kappa'}$$

Figure 5. Subtyping rules for higher-kinded types

Note that `Shape` itself has kind $* \to *$, so `Mayhem<Shape>` is a valid type of kind $*$. Consider, then, whether `Mayhem<Shape>` is a subtype of `Shape<Mayhem<Shape>>>`. We can prove this with the following infinite derivation (making intermediate steps explicit):

`Mayhem<Shape>` $<:$ `Shape<Mayhem<Shape>>>`
$\Downarrow$ (*inheritance*)
`Shape<Shape<Mayhem<Shape>>>` $<:$ `Shape<Mayhem<Shape>>`
$\Downarrow$ (*contravariance*)
`Mayhem<Shape>` $<:$ `Shape<Mayhem<Shape>>>`
$\vdots$

This example exploits the fact that `Shape` can be used as an argument to a higher-kinded parameter that can be used without restriction in order to violate our invariant that shapes are never introduced by expanding inheritance.

Fortunately, due to Material-Shape Separation we can adapt our earlier proof strategy to a higher-kinded type system without nested shapes. We formalize this higher-kinded type system in Figure 4 and its subtyping rules in Figure 5. For the sake of algorithmic simplicity, we present the minimal form of type-level computation necessary for the system to work as expected; this minimality is not necessary for our proof strategy below. We use $\overline{\ }$ to indicate "some number of", being consistent with that unknown number across multiple uses of $\overline{\ }$ within a rule, and similarly for $\overline{\overline{\ }}$. For example, in the rule for variables, $\overline{\overline{\ }}$ represents the number of applications to the variable X, and $\overline{\ }$ represents the number of arguments in each of those applications. The premises indicate that each corresponding pair of `in` (or `out`) arguments of each corresponding pair of applications must be supertypes (or subtypes).

$$\begin{array}{|c|}\hline \tau : \kappa \mapsto |\tau| : \kappa[* \mapsto \mathbb{N}] \\ \hline \bot \mapsto 0 \\ \top \mapsto 0 \\ X \mapsto X \\ \mathcal{M} \mapsto M_{\mathcal{M}} \\ \tau \langle \overline{\texttt{in } \tau_i \texttt{ out } \tau_o} \rangle \mapsto 1 + |\tau| \left(\overline{\max(|\tau_i|, |\tau_o|)}\right) \\ \lambda \overline{X}.\tau \mapsto \lambda \overline{X}.|\tau| \\ \hline \end{array}$$

$$M_{\mathcal{M}:\langle \overline{\kappa} \rangle \to *} = \lambda \overline{X}.\, \max\left(\max(\overline{\zeta_\kappa(X)}), \max_{\mathcal{M}\langle \overline{X} \rangle <:: \mathcal{M}' \langle \overline{\tau} \rangle} M_{\mathcal{M}'}(|\overline{\tau}|)\right)$$
$$\text{where} \qquad \zeta_*(m) = m \qquad \zeta_{\langle \overline{\kappa} \rangle \to \kappa'}(m) = \zeta_{\kappa'}(m)(\overline{0})$$

We implicitly lift $\max$, $1+$, *and* 0 *when applied to functions.*

Figure 6. Measure for higher-kinded types

Theorem 3. *Under Material-Shape Separation, all proofs of subtyping as specified in Figure 5 are finite.*

Proof. As before, our strategy is to identify a measure for types such that the sum of the measures of the types being compared always decreases as the syntax-directed algorithm progresses. We no longer need to consider nested uses of shapes, but now we must consider higher-kinded types. To do so, we assign a type τ of kind κ a measure $|\tau|$ of type $\kappa[* \mapsto \mathbb{N}]$. For example, a type $\hat{\tau}$ of kind $\langle *, * \rangle \to *$ is assigned a measure $|\hat{\tau}|$ of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The intuition is that if $\hat{\tau}$ were applied to types with measures m and n then $|\hat{\tau}|\,(m, n)$ is the measure of the applied type.

We define this measure in Figure 6 (reusing type-variable names as measure-variable names). The challenge is to prove that this measure is well defined. To do so, observe that the definition of the measure function $M_{\mathcal{M}}$ for a material constructor $\mathcal{M}$ only references measure functions for materials used in the inheritance clauses of $\mathcal{M}$. Due to Material-Shape Separation, we can assume that those material functions are terminating, thereby making $M_{\mathcal{M}}$ a terminating function as well. Structural induction then easily demonstrates that the measure is well defined on all types.

This measure on types can be adapted into a measure on subtyping judgements. We define the measure of a subtyping judgement $\overline{X : \kappa} \vdash \tau <: \tau' : *$ to be $(|\tau| + |\tau'|)[\overline{X \mapsto 0}]$. Note that we only define this measure for subtyping judgements of kind $*$. This is because we view the only rule applicable to other kinds as intermediate since by structural induction on the kind it simply introduces fresh variables until the types being compared have kind $*$.

Finally, one can easily verify that for each rule the measure of the premises (after processing the rule for introducing fresh variables) is always strictly less than the measure of the conclusion, thereby guaranteeing that any proof must be finite even if infinite proofs were permitted. □

The proof for computable joins extends similarly. Thus, by separating materials and shapes we are able to add a powerful, fully functional feature to our type system with minimal effort.

6. Future Work

We have shown that the separation of materials and shapes is practical, with a broad survey demonstrating its compatibility with existing code and with anecdotes offering insight into why this pattern arises. We have also shown that this separation simplifies and improves various core typing algorithms even in the presence of in-

tersection types and higher-kinded type variables. Now we present new type features that may be made possible by our findings.

6.1 Conditional Inheritance

Although Material-Shape Separation solves a number of open type-checking problems, our initial motivating use case remains unsolved. Recall that we wanted a type-safe way to make `Lists` have equality whenever their elements have equality. We believe we could apply our findings to conditional inheritance to produce an effective solution. Here is how our example might look like using conditional inheritance:

```
interface List<out T> {...}
          extends Equatable<List<T>>
          given T extends Equatable<T> {...}
```

This seems ideal, something akin to Haskell's type classes [28], but now consider our `Tree` specification once again. The question again is whether `Tree` extends `Equatable<Tree>`. Since `Tree` extends `List<Tree>`, this holds provided `List<Tree>` extends `Equatable<Tree>`. According to the above specification, `List<Tree>` extends `Equatable<List<Tree>>` (a subtype of `Equatable<Tree>`) *provided* `Tree` extends `Equatable<Tree>`. And now we are back where we started. Once again we are building a valid, yet infinite proof.

cJ and JavaGI have already made an effort to incorporate conditional inheritance [9, 29]. However, cJ has no proof for decidable type checking (and appears to be undecidable), and the above example causes the JavaGI compiler to stack overflow even though the language has a proof of decidability. Most likely this is because a decision algorithm for the above would at least need to track all entailments and subtypings currently being processed and continually check these for repeats in order to identify when inside a cyclic infinite proof, a rather expensive and complicated process. Even if implemented correctly, this approach is most likely brittle and may not be able to extend to systems where type equivalence does not imply syntactic identity.

We believe conditional inheritance would be decidable in our system. In particular, we disallow the problematic recursive specification of `Tree` in Section 5.1, instead encouraging the following:

```
class Tree extends Equatable<Tree> {
      List<Tree> children() {...}
      Boolean equals(Tree that) {
        return children().equals(that.children());
      }
}
```

This implementation provides predictable, understandable behavior. Furthermore, in the case of shapes, we believe it would be possible to override a default implementation locally without ever producing any semantic inconsistency via variance and subtyping, since shapes may not occur as type arguments. Nonetheless, there are many subtleties to explore both in terms of type checking and in terms of run-time implementation, so we defer detailed investigation to future work.

6.2 Decidable Intraprocedural Type Inference

With computable joins, we have the beginnings of decidable intraprocedural type inference. Ideally one would be able to take a function whose context is well established, including types for parameters and an explicit return type, and determine whether it type checks without needing any type annotations in its body. There are two major challenges we foresee for completing this goal. First, one must design an object-oriented type system with principal types, which requires addressing practical issues such as overloading, as well as theoretical issues such as type-argument inference. Second, one must infer the types of loop variables whenever typeable. This latter challenge may prove very difficult since, even given Material-Shape Separation, subtyping is still not well founded despite all proofs being finite. For example, `Array<in Object>` is a subtype of `Array<in Array<in Array<in Object>>>`, which is only the beginning of an infinite progression. Nonetheless, Material-Shape Separation drastically simplifies the forms that subtyping constraints can take, so we believe it may be a first step towards decidable intraprocedural type inference for object-oriented languages. Such a feature would not only make programming in statically-typed languages more convenient, but would also enable one to dynamically incorporate untyped code into typed systems by type checking it at run time without error.

6.3 Virtual Types

Virtual types are, in summary, the idea that objects can have types as members [12, 13]. For example, each graph object could have a member `V` indicating the type of its own vertices. Self types are a special form of virtual types, and type families are a means to approximate virtual types with F-bounded polymorphism.

There are many ways to implement the concept of virtual types within a type system [2, 10, 16, 23–25]. For example, with significant effort one can encode virtual types using the implicit constraints [20] of Java's wildcards combined with wildcard capture [26], or much more simply one can use Scala's path-dependent types [16]. Regardless of the specifics, any encoding of virtual types must address their many subtleties, such as those relating to wildcards as described by Tate et al. [20]. We posit that Material-Shape Separation may alleviate these subtleties. For example, by incorporating the constraints on the virtual types of a class/interface into the usage graph and measure, we might be able to extend the definition of Material-Shape Separation and the proof of decidable subtyping to virtual types. With further investigation, one might be able to support constrained virtual types without sacrificing principles such as decidability.

7. Related Work

Previous work in this area has focused primarily on algorithmic issues. Kennedy and Pierce mapped the boundary of decidable subtyping, giving three forms of restrictions each of which would guarantee decidability [11]. These provided subsequent works, ours included, a firm basis for future explorations.

Wehr et al. built JavaGI, adding conditional inheritance to the type system [29], and incorporating Kennedy and Pierece's results to achieve the decidability missing from cJ [9]. However, probably due to the complexity of the underlying algorithms, the implementation of their type checker does not match the specification. Our results suggest that acknowledging the separation between materials and shapes might help to repair and simplify their implementation.

Smith and Cartwright identified problems specific to type-argument inference in Java and proposed an extension to the type system with corresponding algorithms [18]. In particular, Java wildcards do not admit joins, so Smith and Cartwright proposed adding union types to Java's type system, though these introduce complications elsewhere in the type system [20]. Our finding is that, by not allowing shapes as type arguments, we admit and can compute joins without the need for union types. This change could be incorporated into Smith and Cartwright's algorithms.

Most recently, Tate et al. identified nested contravariance as a source of complications [20]. Removing it, they found, would make subtyping decidable in a manner compatible with existing code bases. Yet their restrictions significantly restrict contravariance and are strongly influenced by corner cases. Like Smith and Cartwright, Tate et al.'s proposal does not admit joins, nor does it extend to the many features we have addressed in this paper.

8. Conclusion

Our approach differed from the aforementioned past endeavors in that we did not take algorithmic considerations as our imperative. Rather we considered the question from the perspective of the language user, trying to determine a solution that would gracefully handle the common cases and clearly isolate edge cases. All of the existing solutions we mentioned are incomparable to each other and to our own in terms of formal expressiveness, but ours is the only solution admitting more advanced type features and providing a clear separation of concepts that can be explicitly integrated into a language design. Our system is simple to understand, implement, and extend.

Beginning with insights from our colleagues in industry, we defined two different kinds of classes and interfaces: materials and shapes. Materials are the classes/interfaces exchanged throughout the programs; shapes are the classes/interfaces used to describe types via recursive inheritance. We then presented Material-Shape Separation, which prohibits shapes from being used as type arguments. This split was motivated by the theoretical desire to pinpoint the source of recursion in the type system and the practical intuition that shapes are fundamentally different from materials. Subsequently, we justified Material-Shape Separation through the systematic analysis of 13.5 million lines of open-source generic-Java code. This was compelling evidence that shapes are never in practice used outside of inheritance and type-parameter constraints.

Next we proved the decidability of subtyping with type equivalences even under naïve algorithms, showed the computability of joins, and demonstrated the capability to handle constrained higher-kinded type variables. These results all followed naturally from Material-Shape Separation, reasserting its usefulness and encouraging us of its future prospects. Indeed, we identified several avenues for future work where separating shapes from materials may make even more advanced features feasible.

Since the barrier for adopting the separation of materials and shapes is very low, especially when contrasted with the gains in both decidability and simplicity, we believe that it can easily be incorporated into new and existing statically-typed object-oriented languages. As evidence, the designers of Ceylon have already taken interest in our design, and are likely to integrate Material-Shape Separation into Ceylon 2.0. Given the benefits of enforcing Material-Shape Separation, there is every reason to scale down F-bounded polymorphism and get it into shape.

Acknowledgements

The task of finding a large body of existing software projects was greatly facilitated by the Qualitas Corpus, a massive collection of open-source Java projects [21]. We thank Professor Ewan Tempero from the University of Auckland for maintaining the Qualitas Corpus and for providing advice and feedback as we worked with the projects. Additionally, we drew from the compiled *Qualitas.class Corpus* and wish to thank its maintainers for providing a truly valuable resource [22].

We are indebted to Avik Chaudhuri, Basil Hosmer, and Michael Vitousek for posing the crazy idea of removing F-bounded polymorphism altogether, and for providing feedback as we made that thought (with compromises) into reality. We are also grateful to our colleagues at Cornell for providing insights into the cultural context of the paper and for providing feedback on how to better convey the material to a broader audience. We greatly appreciate Stephen Longfield in particular for his assistance in the final crunch. Lastly, we thank the anonymous reviewers for identifying where we needed to clarify our writing and suggesting how we might strengthen our arguments.

References

[1] Franz Baader and Wayne Snyder. Unification theory. *Handbook of Automated Reasoning*, 1:445–532, 2001.

[2] Kim B Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP*, 1998.

[3] The Ceylon language specification, version 1.0. `ceylon-lang.org/documentation/1.0/spec/`, November 2013.

[4] Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, 1989.

[5] Erik Ernst. Family polymorphism. In *ECOOP*, 2001.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley Professional, 2005.

[7] Benjamin Greenman, Fabian Muehlboeck, and Ross Tate. Getting F-bounded polymorphism into shape. Technical report, Cornell University, March 2014.

[8] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with safe type conditions. In *AOSD*, 2007.

[10] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *ECOOP*, 1999.

[11] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

[12] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the BETA programming language. In *POPL*, 1983.

[13] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes - a powerful mechanism in object-oriented programming. In *OOPSLA*, 1989.

[14] Robin Milner, Lockwood Morris, and Malcolm Newey. *A Logic for Computable Functions with Reflexive and Polymorphic Types*. Department of Computer Science, University of Edinburgh, 1975.

[15] Martin Odersky. The Scala language specification, version 2.9, March 2014.

[16] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[17] Benjamin C. Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1-2):235–282, 1997.

[18] Daniel Smith and Robert Cartwright. Java type inference is broken: Can we fix it? In *OOPSLA*, 2008.

[19] Ross Tate. Mixed-site variance. In *FOOL*, 2013.

[20] Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in Java's type system. In *PLDI*, 2011.

[21] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas Corpus: A curated collection of Java code for empirical studies. In *APSEC*, 2010.

[22] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[23] Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP*, 1997.

[24] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP*, 1999.

[25] Mads Torgersen. Virtual types are statically safe. In *FOOL*, 1998.

[26] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. Adding wildcards to the Java programming language. In *SAC*, 2004.

[27] Mirko Viroli. On the recursive generation of parametric types. Technical Report DEIS-LIA-00-002, University of Bologna, Italy, September 2000.

[28] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, 1989.

[29] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI : Generalized interfaces for Java. In *ECOOP*, 2007.