Shape-Shifters: Type Classes for Declaration-Site Variance

Abstract

Type classes are an excellent feature of Haskell. In particular, they are one of the best solutions for type-safe equality. Unfortunately, they have proven difficult to adapt to object-oriented programming.

In this paper, we investigate this problem applied to the common architecture of collection classes and interfaces using declaration-site variance. At the same time, we enable developers to build a default implementation of a type class into a class or interface, such that the compiler can erase type classes whenever their default implementation is used. This investigation makes apparent two additional requirements for a type-class feature design to be practical: an ability to supply a non-built-in type-class implementation, and a need to explicitly distinguish between callee's and caller's type-class implementations.

We present a design that accommodates these requirements. This design is regrettably complex, though the complexity seems unavoidable once one identifies all of the design constraints. The goal of this paper is to educate readers about aspects of the design space that have been ignored in the past, and to offer a concrete step towards addressing those aspects.

1. Introduction

Type classes are a renowned feature of the Haskell programming language [18] that would be a great to integrate into mainstream object-oriented languages. In particular, well-behaved type-safe equality remains an open problem in mainstream object-oriented languages, and type classes solved this same problem for Haskell. But three challenges repeatedly arise in transferring this language technology:

Subtyping To be convenient, the implementation of a type class needs to be inferred from the static type information of the relevant expressions, but subtyping implies that those expressions can have multiple types simultaneously, potentially giving rise to ambiguity.

Generics Generic (as in parametrically polymorphic) collections run into the issue that they can only satisfy a type class if its type argument does so as well, requiring features akin to conditional inheritance [19].

Erasure For many classes and interfaces, the default implementation of a type class can be built into the instance's virtual-method table, so one wants to be able to erase type classes when appropriate rather than dynamically passing around and constructing redundant implementations.

There are proposals that address these challenges [10, 17, 24, 30, 41], but most are incompatible with another feature that has become an industry standard: declaration-site variance. Declaration-site variance is the ability to declare that a class or interface is covariant or contravariant with respect to its type parameter. For example, whenever every foo is also a bar, then every iterator of foos is also an iterator of bars, making iterator a covariant interface. This property can be verified automatically by simply analyzing the type signature of the class or interface.

To see the first fundamental problem with combining type classes with declaration-site variance, consider the following interface concept with type-safe equality:

```
interface Iterable<out E>
    <if E satisfies Equatable> satisfies Equatable {
  Iterator<E> iterator();
  <if E satisfies Equatable> Boolean equals(Iterable<E> that);
}
```

This is the first specification that comes to mind when considering type classes and collections. The problem, though, is that the signature of `equals` is *not* covariant; the signature for `equals` is actually *contravariant*. And though the standard general-purpose implementation of `equals` happens to work correctly, more specialized implementations can encounter run-time type errors due to this variance mismatch.

In this paper we examine how to address this use case among many others, focusing on classes and interfaces taken from the standard libraries of current major industry languages. We first demonstrate how the above challenges and these use cases impose even more requirements on designs for type classes, and at the same time we argue why those additional requirements should be considered anyways for the object-oriented paradigm. We then present a language design that satisfies the original challenges but ignores their derived requirements. Afterwards, we add features addressing those derived requirements, illustrating the subtle complexities caused by those requirements. We demonstrate the tradeoffs of addressing these challenges by comparing to the prior work in the area, both in industry and in academia. Lastly, we reflect on the lessons learned and how we might expand the capabilities of our design.

2. Design Requirements

Here we outline the design space that we are working in. First, we describe the kind of languages we are targeting, along with common use cases for developers of those languages. Then, we demonstrate how this setting necessarily imposes requirements additional to those already mentioned.

To begin, we are targeting object-oriented languages with generics (i.e. parametric polymorphism for both classes/interfaces and methods) and at least declaration-site variance. Declaration-site variance is used to specify how parametric polymorphism should interact with subtype polymorphism [3]. In particular, if a class were declared `Covariant<out T>`, then the `out` annotation indicates that if τ is a subtype of τ', then `Covariant<`τ`>` is a subtype of `Covariant<`τ'`>`. On the other hand, if a class were declared `Contravariant<in T>`, then the `in` annotation indicates that if τ is a subtype of τ', then `Contravariant<`τ'`>` is a subtype of `Contravariant<`τ`>` (note the reversal). `Iterator` is a classic example of a covariant interface, `Property` is a classic example of a contravariant interface, and `Array` is a classic example of an *invariant* interface. Invariance means that `Invariant<`τ`>` is a subtype of `Invariant<`τ'`>` if and only if τ and τ' are subtypes of each other, i.e. equivalent.

In order to be sound, restrictions must be placed on which classes and interfaces can be declared covariant or contravariant. In particular, the body of the class or interface must likewise be covariant or contravariant, as determined by the contained types and their positions. For example, since `Iterable` is covariant, then `equals` below is contravariant with respect to `E`, `clone` is covariant, and `plus` is invariant:

```
Boolean equals(Iterable<E> that);
Iterable<E> clone();
Iterable<E> plus(Iterable<E> that);
```

And interestingly, although the first signature below is contravariant, the second signature below is in fact covariant:

```
Boolean contains(E element);
Boolean contains<T super E>(T element);
```

The `<T super E>` component indicates that the method can be used for any type `T` that is a supertype of `E`.

What we would like to add to such a language is type classes. Whereas a type reasons about what structure an instance has, a type class reasons about what structure a type has [37]. Common examples that we will be using throughout this paper are `Equatable`, `Cloneable`, and `Summable`. A type is `Equatable` if all of its instances can be compared to each other for some notion of semantic equality; similarly, `Comparable` is for types with some decidable total ordering on its instances. A type is `Cloneable` if all of its instances can be replicated while duplicating the state so that any mutations to the clone do not affect the original. A type is `Summable` if all of its instances can be "added" to each other to produce some other instance of the type. We choose these three examples because the first turns out to be covariant, the second contravariant, and the last invariant.

Besides specifying type classes, we want to incorporate them into parametric polymorphism. For example, a generic `sort` method could be defined on any `Array` whose type argument is `Comparable`. We express this with the following:

```
void sort<E satisfies Comparable>(Array<E> array);
```

We can extend this concept to generic classes and interfaces. But the problem that quickly arises is that a class like `Array` can satisfy `Equatable`, but only if its type argument satisfies `Equatable`. Thus we encounter the need for conditional satisfaction, much like in Haskell [20]. However, unlike in Haskell, here we have concepts like inheritance and variance to deal with.

To see why this is a challenge, consider the interface `Iterable`. There is an obvious way to define equality on iterables (provided the elements are themselves equatable), and many applications want to use this equality on iterables. After all, at the type level, C#'s LINQ feature is all about manipulating `IEnumerables` [26], which is C#'s variant of `Iterable`. But `Iterable` is meant to be a common superinterface of a variety of collection interfaces, including `Set`. And `Set` has a different notion of equality than the obvious one for `Iterable`.

This means that an instance of `Set` has two sensible notions of equality defined on it. The question then becomes how to resolve this ambiguity. One way to resolve it would be use the dynamic type of the instance, but this means programmers must know the possible dynamic types of a given `Iterable` to reason about its behavior, thereby ruining modularity. Another way would be to use the static type of the instance, but this means the implementation must pass around run-time information indicating which view to use, thereby ruining erasure of type classes. Furthermore, and arguably more importantly, it prevents data structures from optimizing for the specific view they were created with. After all, a red-black tree can only exploit its structure when it knows its users are using the same notions of equality and comparison.

Thus, in addition to the aforementioned design requirements, we recognize that a design must accommodate both the dynamic and static resolution approaches while still remaining unambiguous to the programmer (without the programmer having to know complex behind-the-scenes algorithms for things like ambiguity resolution). We call this *The Perspective Challenge*, as it boils down to resolving the fact that every method call on an instance has two possible perspectives of how a type class should be implemented: the perspective of the instance, i.e. the callee, and the perspective of the environment, i.e. the caller. This challenge of type classes is arguably specific to object-oriented programming due to its philosophy that objects represent data and interpretation that data, which may be fundamentally at odds with type classes given the above common use cases. Regardless of these philosophical questions, we will illustrate the challenges of accommodating these requirements, and propose one design solution that manages to satisfy them.

```
shape Equatable {
  Boolean equals(This that);
}
shape Comparable extends Equatable {
  Boolean leq(This that);
  Boolean lt(This that) { return leq(that) && !equals(that); }
}
shape Hashable extends Equatable {
  Integer hash();
}
shape Cloneable {
  This clone();
}
shape Summable {
  This plus(This that);
}
```

Figure 1. Examples of shapes

3. Shaping Type Classes

Here we detail how we use shapes to address the initial design requirements we had set out for. That is, we present the design one would arrive it if one did not recognize or opted to disregard The Perspective Challenge. As we do so, we will also illustrate challenges and motivations related to each component. In the next section we will add shape-shifters to address The Perspective Challenge and at the same time improve the traditional functionality of type classes.

3.1 Shapes

Greenman et al. recently observed that industry developers already subconsciously divide classes and interfaces into what Haskell people would call types and type classes [16]. They used the term *material* for classes and interfaces that were used as type arguments, field types, variable types, parameter types, and return types. That is, materials are the classes and interfaces that are explicitly passed around, such as data structures, GUI elements, and exceptions. They used the term *shapes* for classes and interfaces that were used recursively in inheritance clauses and type constraints. That is, shapes are the classes and interfaces that are used to specify structure on types rather than individual instances, i.e. type classes. In order to avoid confusion with past literature and preexisting terminology, we will adopt the term "shapes".

In our design, materials and shapes are explicitly separated both in syntax and in usage. All types in our design are materials, and all classes and interfaces are materials. Because Greenman et al. demonstrated that programmers in practice already exhibit a well-founded inheritance structure on materials, we will impose that structure as a requirement in our design. For the most part this technicality can be ignored; we use it solely to make type checking decidable while preserving practical expressiveness.

```
class Integer(int value)
    satisfies Comparable, Hashable, Cloneable, Summable {
  Boolean equals(Integer that) { return value == that.value; }
  Boolean leq(Integer that) { return value <= that.value; }
  Integer hash() { return this; }
  Integer clone() { return this; }
  Integer summable(Integer that) {
    return new Integer(value + that.value);
  }
}
```

Figure 2. Example class satisfying shapes

As for shapes, some examples in our design can be seen in Figure 1. Shapes are essentially interfaces whose signatures have access to a distinguished type variable `This`. The `This` represents the type whose structure the shape is describing, and each method in a shape is free to use `This` in covariant and contravariant positions. Shapes can extend other shapes, as demonstrated by `Comparable` and `Hashable`. Shapes can have default implementations of methods, as demonstrated by `lt` of `Comparable`, and these can be declared non-overrideable. Although declaration-site variance and default implementations in shapes are not necessary aspects of our design, we present them to make potential adopters aware of what is possible. What is important, though, is that shapes are never used as *types*; they are only used as type *constraints*.

3.2 Satisfying Shapes

A class or interface can declare itself to satisfy a shape as in Figure 2. For the declaration to be valid, the class or interface must implement all the methods declared in the shape (and its supershapes) with the uses of the `This` type variable replaced by the implementing class or interface's type. Thus we ensure that shape methods can be invoked on any instance of the class or interface and will accept and return instances of the appropriate type.

Satisfaction does not work like inheritance, though. In particular, if `Foo` inherits `Bar` which satisfies `Cloneable`, that does not imply that `Foo` also satisfies `Cloneable`. After all, unless `Foo` overrides `Bar`'s implementation of `clone`, the method might return a `Bar` instance that is not necessarily a `Foo` instance. So, if `Foo` should be `Cloneable`, the programmer must declare it as such and *override* its implementation of `clone` to always return a `Foo` instance. It is important that the method is overridden rather than overloaded, otherwise we run into issues with The Perspective Challenge, with subtype polymorphism introducing ambiguities.

A language designer might consider requiring that all classes and interfaces satisfy the shapes satisfied by their superclasses and superinterfaces. However, there are many practical use cases that would fail to satisfy this requirement. For example, it makes sense for a (mutable) `List` interface to be `Cloneable`, but one can imagine a subclass `File` that implements a list by backing it with a file on a hard drive, in which case requiring `File` to be `Cloneable` would force its `clone` method to copy the file contents to a new file on the hard drive so that the new file can back the new `File` instance. Another example is that one might want to specialize `Integer` with a `One` subclass, but 1 plus 1 is not 1, so `One` could not satisfy `Summable`.

Inherited Shapes There is one exception we can make to the limitation between inheritance and satisfaction. Suppose `Foo` extends `Bar` which satisfies `Equatable`. Then `Bar` has an `equals` method that accepts any `Bar` instance as its argument and returns a `Boolean`. Because `Foo` is a subtype of `Bar`, this method also accepts any `Foo` instance as its argument and returns a `Boolean`. Consequently, `Foo` inherits an `equals` method from `Bar` that automatically satisfies the requirements for `Foo` to satisfy `Equatable`. This pattern holds for any shape whose signature is contravariant with respect to the `This` type variable, which holds for `Comparable` and `Hashable` as well. We permit such shapes to be declared `inherited`, and we say that any type that has a supertype satisfying an `inherited` shape then also satisfies that shape.

3.3 Constraining with Shapes

The whole point of this effort is to enable generic methods to be restricted to types with extra structure. We do so by allowing `satisfies` clauses to constrain type variables, as in the following:

```
Boolean contains<E satisfies Equatable>(Iterable<E> elems, E value) {
  for each (elem in elems)
    if (elem.equals(value))
      return true;
  return false;
}
```

When a type variable is constrained by a shape, its instances are assumed to have the methods of the shape with `This` replaced by the type variable. Hence the `contains` method can assume the `E` instance `elem` has an `equals` method that accepts the `E` instance `value`.

When calling a generic method, one supplies the type argument and the type checker ensures that the type argument satisfies the appropriate shapes. For the purposes of this proposal, we do not consider the challenge of type-argument inference. However, this proposal does ensure that all valid type arguments would have the same implementation of the relevant shapes wherever they overlap. This is one reason why we require subclasses to override shape methods rather than overload them.

3.4 Conditional Methods

Many generic classes and interfaces have methods that can only be implemented (or only make sense) when the type argument satisfies certain additional requirements. For example, an array can only clone itself if whatever it is an array of can also clone itself. One could implement an `Array` class and a `CloneableArray` class, with the latter extending the former, imposing additional constraints on its type parameter, and providing an additional method. But this strategy is tedious and blows up as more and more combinations of side conditions are introduced. Furthermore, it still will not be the case that an `Array<String>` (where `String` is cloneable) is necessarily a `CloneableArray` because it may not have been allocated as such for reasons that hark to The Perspective Challenge.

Thus, with the addition of type classes comes a pressing need for *conditional* methods [25]. We demonstrate our design for this with the following conditional method for `Array<E>`, with syntax inspired by cJ [19]:

```
<E satisfies Cloneable> Array<E> clone() {
  return new Array<E>(index => this[index].clone());
}
```

The prefix `<E satisfies Cloneable>` of the signature indicates that the method is only invokable if the type argument of the `Array` at the call site satisfies the condition, which in this case means it must be `Cloneable`. Inside the method implementation, the prefix enables the method to assume that the type parameter satisfies the condition, in this case permitting `clone` to be invoked on all instances of `E`.

It turns out that constraining a type parameter is an invariant use of the type parameter. The exception is when it is constrained to satisfy only `inherited` shapes, in which case it is a covariant use of the type parameter. One would like covariant interfaces to have conditional methods like `clone`, along with many other methods which traditionally do not have covariant signatures, so we address this limitation of declaration-site variance with the feature below.

3.5 Quantifying over Supertypes of Type Arguments

Declaration-site variance has some inherent problems with a variety of common use cases. Consider the interface `Iterable`. It is naturally a covariant interface: something that iterates a bunch of strings also iterates a bunch of objects. The fact that `Iterable` is covariant also means that iterable comprehensions, e.g. `{ for i in ints yield i+1 }`, can have a principal type. But then consider one of the most obvious methods one would want for `Iterable<E>`:

```
Boolean contains(E elem);
```

Ignoring for now the fact that one can only implement this if `E` is `Equatable`, notice that this method is not covariant! Libraries typically get around this by using run-time type casts, but that defeats the point of static type checking and the technique is known to have let many bugs go unnoticed. So we would like something that addresses the use case while still being type safe.

Our proposal is the following method for `Iterable<E>`:

```
<super E satisfies Equatable> Boolean contains(super E elem);
```

For now, this can be viewed as a shorthand for the following:

```
Boolean contains<T super E satisfies Equatable>(T elem);
```

It turns out that this signature is covariant! This is a known technique [14] and is often employed manually in Scala libraries [28], but by integrating it into the language design we will be able to make specialized accommodations for this common use case.

To demonstrate the utility of this, consider the following method implementation for `Iterable<E>`:

```
<super E satisfies Summable>
    Iterable<super E> plus(Iterable<super E> that) {
  return { for (e1, e2) in zip(this, that) yield e1.plus(e2) };
}
```

This demonstrates that addition can be extended to iterables componentwise. The signature and its implementation are conceptually shorthand for the following:

```
Iterable<T> plus<T super E satisfies Summable>(Iterable<T> that) {
    return { for (e1, e2) in zip(this, that) yield e1.plus(e2) };
}
```

Every use of `super E` within the scope of the method is replaced with the same fresh type variable `T` that is constrained to be a supertype of `E` and satisfy all the shapes that `super E` was required to satisfy. With this we achieve a convenient and sound technique for circumventing the limitations imposed by declaration-site variance.

3.6 Conditional Satisfaction

Now we have the tools for enabling classes and interfaces to conditionally satisfy a shape. Consider the examples illustrating our design in Figure 3.

The first thing one might notice is that `Iterable` satisfies nothing, even conditionally; we will discuss the architectural reasons for this in the next section. The next thing to notice is that we now allow conditionals in satisfaction clauses, here used by `Indexed` and `Array` (we will discuss conditional inheritance clauses later). For example, `Array` indicates that the type is cloneable only when the type argument is cloneable. These conditional clauses are valid if the body would have the appropriate methods if the declared conditions held. In the case of `Array`, the condition on its `clone` method is actually more permissive than necessary for the conditional satisfaction clause, but this is necessary for it to implement `Iterable` whose `clone` method must be parameterized over `super E` in order to be covariant.

The condition for `Indexed` to satisfy `Equatable` is probably not what the reader expected. It indicates that `Indexed<`τ`>` is equatable if *any* supertype of τ is equatable. This may seem complex, but it is necessary to accommodate the covariance of `Indexed`. Just like for methods, conditioning on a type parameter `T` in satisfaction clauses is an invariant use of `T`, so one must instead condition on `super T` for `T` to be covariant. Because methods can explicitly condition upon `super T`, we can check if the appropriate methods exist when only `super T` can be assumed to satisfy the shapes listed in the satisfaction clause's condition.

```
interface Iterable<out E> {
  Integer size {
    Integer s = 0;
    for each (elem in this)
      s++;
    return s;
  }
  <super E satisfies Equatable>
      Boolean equals(Iterable<super E> that) {
    if (!length.equals(that.length))
      return false;
    for each ((e1, e2) in zip(this, that))
      if (!e1.equals(e2))
        return false;
    return true;
  }
  <super E satisfies Cloneable> Iterable<super E> clone();
}
interface Indexed<out E> extends Iterable<E>
    <super E satisfies Equatable> satisfies Equatable {
  E get(Integer index);
}
class Array<E> implements Indexed<E>
    <E satisfies Cloneable> satisfies Cloneable {
  <super E satisfies Cloneable> Array<super E> clone() {
    return new Array<super E>(index => this[index].clone());
  }
}
```

Figure 3. Example of conditional satisfaction

Lastly, only `inherited` shapes can be used to condition upon `supers` in satisfaction clauses, regardless of the variance of the type parameter. This ensures that the actual type argument also satisfies the shape, as necessary for soundness. It also makes type checking simple because one can determine whether the condition holds for any supertype of τ just by checking whether it holds for τ. Otherwise, one has to consider all possible supertypes of a type argument to check this condition, of which there can be infinite.

3.7 Specializing Class Implementations

Suppose `Iterable<E>` had the following method:

```
<super E satisfies Equatable> Boolean contains(super E value) {
  for each (elem in this)
    if (elem.equals(value))
      return true;
  return false;
}
```

An implementing subclass like `HashSet` would require its type argument to satisfy `Hashable`, and consequently `Equatable`. Thus it would know that this condition is satisfied *and* it would want to specialize its implementation of `contains` to exploit the hashing structure. Consequently, the class would have an implementation as follows:

```
class HashSet<E satisfies Hashable> implements Iterable<E> {
  Boolean contains(E value) { /* make use of value.hash() */ }
}
```

Our design permits these specializations. However, note that `HashSet` must *also* have a non-specialized implementation. Suppose `Foo` were `Hashable` but extended `Bar` which is only `Equatable`. A `HashSet<Foo>` can be cast to an `Iterable<Bar>`, so it still needs an implementation of `contains` that can accept `Bar` instances even though they have no `hash` method. In this case, that implementation happens to be inherited from `Iterable`. As we will see, the presence of both generalized and specialized implementations will be key to helping us address The Perspective Challenge.

4. Shifting Shapes to Local Perspectives

We now have the basics of type classes adapted to the setting of declaration-site variance. At this point, we work to address The Perspective Challenge. Recall that in Figure 3 we did not make `Iterable` (conditionally) satisfy `Equatable`. This is because different `Iterable` instances will belong to different kinds of collections with different notions of equality. On the other hand, for `Indexed` the notion of equality is fixed to be componentwise. Nonetheless, even though `Iterable` *instances* may vary across their interpretation of equality, one still wants to be able to treat the `Iterable` *type* as though it satisfied equality componentwise.

In other words, we need a way to specify a type-class implementation that is *not* built into the instance, as well as a way to make code using type classes use such an implementation. This will solve our problem for `Iterable` because we can specify and use the componentwise implementation rather than the instances' implementations. But such a solution will also greatly improve the flexibility of type classes, because we could use it to *locally override* a type class's standard implementation for a type. For example, we could indicate to use the reverse ordering for some `Comparable` type rather than the built-in ordering.

Many major libraries already have a way to accomplish this, though through manual means. For example, Java's standard library includes `Comparator`, which can specify a comparison operation on a type without it being built into the type. Right now, whenever one wants to use the non-built-in comparison, one has to hope the library provides a class or method implemented using an explicit `Comparator` instance rather than an implicit `Comparable` subtyping constraint.

This is not always the case, though, and a famous example is the `IdentityHashMap` that had to be added to Java's standard library. `HashMap` (and the `Map` contract in general) does not provide a way to use an equality and hash function different from the one built into the instances. However, people often need hashmaps to use reference equality, so Java added an `IdentityHashMap` to the library for this specific functionality. Note that there still is no such variant for case-insensitive string equality, so one can find a variety of hacks for accomplishing this online, including wrapper classes, using two hashmaps side by side, and using `TreeMap` instead simply because it accepts a `Comparator`. With our proposal, none of these hacks will be necessary, and developers can still get improved performance when using the defaults by exploiting erasure.

4.1 Shape-Shifters

Our design enables programmers to specify alternate implementations of a shape by using *shape-shifters*:

```
object reverse<T satisfies Comparable> implements T.Comparable {
  Boolean (T t).leq(T that) { return that.leq(t); }
}
```

This shows how an object can provide an alternate implementation of a shape for a given type. The interface `T.Comparable` indicates that the target shape is `Comparable` and the target type is `T`. The method signature `Boolean (T t).leq(T that)` indicates it is providing type `T`'s implementation for `Comparable`'s `leq` method, with the variable `t` representing the `T` instance that the method is being called on.

In order to prevent ambiguities in other parts of the design, a shape-shifter can only shift for one type (though for an arbitrary number of shapes). As part of such, the position of that type is considered to be invariant. Consequently, even though the type parameter of `reverse` could theoretically be contravariant, in our design it must be invariant.

4.2 Locally Overriding Shape Implementations

Consider a conditional method. So far we have interpreted the conditional to mean "if this condition is satisfied, then this method is accessible". But we can also interpret the conditional to mean "supply me an implementation demonstrating the condition holds, and this method will execute using that implementation". That is, in addition to the erasable interpretation, there is also the run-time-evidence interpretation. Traditionally designs have chosen one or the other, but we will choose both!

Let us consider how to implement the run-time-evidence interpretation. In the implementation of the conditional method, we can identify method invocations that correspond to the condition. For example, invocations of `equals` on instances of a type parameter correspond to the `satisfies Equatable` condition on that type parameter. There can be some complications with identifying those invocations, but we will address those later.

Next, suppose one supplies a shape-shifter as the run-time evidence that the condition holds. We can then rewrite the implementation to redirect the appropriate invocations to the shape-shifter instead of the instance, and supply the instance that the method was being invoked on as the pseudo-`this` argument of the shape-shifter's method.

For example, the following implementation

```
<super E satisfies Equatable> Boolean contains(super E value) {
  for each (elem in this)
    if (elem.equals(value))
      return true;
  return false;
}
```

could, behind the scenes, be translated to

```
Boolean contains<T super E>(T.Equatable equator, T value) {
  for each (elem in this)
    if (equator.(elem)equals(value))
      return true;
  return false;
}
```

With this in mind, in our design one always specifies how the condition should be specified. If the programmer wants to use the built-in implementations, then they simply supply the type. If they want to use a different implementation, then they supply their preferred shape-shifter, which uniquely indicates the actual type argument being shifted.

So, given an iterable of strings `strings`, the programmer can use `strings.<String>contains("Hello World")` to check whether the collection contains `"Hello World"` according to standard string equality. In this case, the compiler could dispatch to an implementation of the method that is optimized for erasure. Alternatively, the programmer could use `strings.<caseless>contains("Hello World")` to check whether `string` contains a case-variant of `"Hello World"`, where `caseless` provides case-insensitive equality (and hashing, and ordering). In this case, the compiler would dispatch to the implementation that uses a run-time-supplied shape-shifter.

Note that this requires that every class provide a generalized implementation of its conditional methods. That is, the generalized implementation needs to work properly even if the constraint were satisfied using a different type-class implementation than what the class was designed for. Thus, even in cases where classes can provide a specialized implementation of a conditional method, we require that a generalized implementation always be provided as well. This way `strings.<caseless>contains("Hello World")` works correctly regardless of whether `strings` is implemented by an `IdentityHashSet` or a case-sensitive `TreeSet`. The obvious downside to this is that these generalized implementations are often necessarily less efficient. This makes it clear that designs that always favor the caller's perspective, although they enable localized reasoning, effectively disable many efficient implementations when multiple implementations of type classes are possible. Our design favors neither perspective, which we demonstrate after discussing overriding class implementations.

4.3 Overriding Class Implementations

Shape-shifters can be supplied as type arguments to a class when constructing an instance, since constructors are generic functions after all. For example, `new HashSet<caseless>()` would create a hashset that uses case-insensitive equality and hashing instead of the built-in implementation for strings.

This is done by creating behind-the-scenes subclasses of `HashSet` with shape-shifters as hidden fields. The methods of these secret subclasses reroute their type-class invocations on instances of the type parameter to the shape-shifter stored in the appropriate hidden field. In Java, classes like `TreeMap` do this manually by accepting and storing a `Comparator` during construction. And, in order to avoid redundant code, there is no version of `TreeMap` optimized for simply using the built-in comparison operations. In our design, the developer gets both the erased and overriding implementations, and the compiler hides the redundancy by generating both implementations from the code.

Furthermore, for every conditional method in which the type argument and shape-shifters satisfy the condition exactly, the compiler generates an implementation specialized to that implementation (unless the class already has specialized implementations specified, as for `contains` in `HashSet`). Thus, in the end, every conditional method has essentially three implementations: one using the built-in operations for the conditioned type parameter, one using caller-supplied shape-shifters for the conditioned type parameter, and one using allocation-supplied shape-shifters (or built-in operations when none are supplied) for the conditioned type parameter. This enables efficiency when simply using built-in operations, *and* it can address The Perspective Challenge with one more feature.

4.4 Distinguishing Perspectives

As discussed earlier, the caller specifies which shape implementation to use each time they call a conditional method. But occasionally the caller wants to use the shape implementation encapsulated in the instance. The instance may already have a semantic interpretation of its own data, or the instance may be optimized for a particular shape implementation on its data. So we provide a way to explicitly invoke the method using the callee's interpretation rather than the caller's interpretation.

The syntax is simple: we omit the type argument. If `strings` is an `Iterable<String>`, then `strings.contains("Hello World")` calls `contains` using `strings`'s implementation of equality on strings. So `strings.<String>contains` uses the built-in implementation of equality on strings, `strings.<caseless>contains` uses `caseless`'s implementation of equality on strings, and `strings.contains` uses `strings`'s implementation of equality on strings. If `strings` were a `HashMap`, then `strings.contains` would call the specialized implementation that the programmer manually specified to exploit its hashing structure.

Thus we address The Perspective Challenge by enabling library users to explicitly indicate which perspective to use depending on their domain-specific needs, and by enabling library writers to optimize for their perspective while often automatically generating the adaptations to other perspectives. Contrast this with Java's libraries that typically prefer the callee's perspective, or with Scala's implicits that typically prefer the caller's perspective. Of course programmers in both languages could manually implement both perspectives, but this leads to a lot of redundant code, and the developer is out of luck when provided with a library that only accommodates one perspective.

Moving back to technical points, we must restrict when callee-perspective methods can be invoked. Note that the specialized implementations of conditional methods are generated only when the argument *exactly* satisfies the condition. To understand what this means, suppose `Array<E>` had a `clone` method conditioned upon `<super E satisfies Cloneable>`. Next, suppose `Foo` inherits `Bar` but only `Bar` satisfies `Cloneable`. Then a specialized implementation of `clone` would *not* be generated for an `Array<Foo>` because, although `Foo` does have a supertype that satisfies `Cloneable`, the type `Foo` does not itself satisfy `Cloneable`. This may seem overly restrictive, but `Foo` can actually have multiple supertypes that satisfy `Cloneable`, in which case it is not clear how the specialization should be implemented, nor would it be clear how to make the specialization available in a way that works for all supertypes of `Array<Foo>`. The complications get even worse when considering invariant methods like `plus`.

For this reason, we only permit callee-perspective method invocations when we can guarantee it is implemented by the target instance and with the appropriate signature. This is the case when the static type argument satisfies the condition and the corresponding type parameter is either invariant or it is covariant and the static type argument has no strict subtypes. Thus our use of `strings.contains` above is only valid/sound in languages where `String` has no subtypes.

4.5 Shape-Shifters in Types

We should clarify how shape-shifters affect the types in our type system: we do not incorporate shape-shifters into the types. Shape-shifters can be dynamically generated (such as row orderings in GUI tables); they can depend on inputs (such as equality modulo some radix); they can be redundant (such as two libraries unintentionally implementing conceptually the same shape-shifter); they can have common subfunctionality (such as `reverse` having the same equality as the original), and they can have all the complications inherent to imperative languages (such as state and exceptions). In short, shape-shifters are simply objects implementing interfaces that happen to have specialized language support. To incorporate shape-shifters into the types would require equational reasoning over a complex space or developing some ad hoc approximation of it with ways to override when the approximation fails to work correctly. Plus developers may intentionally mix together implementations using different shape-shifters, which would require some sort of existential quantification. We believe the complexity and/or inconsistency of any such design is not worth the semantic guarantees it provides.

There is one small way we do change our types, though. Sometimes shape-shifters can be used to give a type structure that it would not otherwise have. For example, functions in general do not have decidable equality, but functions with a finite domain do. So we permit type arguments to class and interface types to be amended with `+Equatable` and other shapes to indicate that the related callee-perspective conditional methods can be invoked (provided the aforementioned requirements also hold). These can easily be incorporated into the subtyping rules, though interestingly the rules are the same regardless of the variance of the corresponding type parameter.

4.6 Optimizing Caller-Perspective Invocation

At present, caller-perspective invocations are burdensome to write because one has to always supply the type argument, even when just using the built-in operations. This burden is necessary to avoid ambiguities, but we might as well optimize for at least one convention. We do so by introducing the syntax `strings:contains("Hello World")`.

With colon invocation, we use the static type information *currently* available to determine the type argument to supply. We say current because providing more detailed static type information could change the run-time semantics. Nonetheless, in our experience programmers often view an expression's principal type as *the* type for the expression rather than just one of many possible types for the expression. So we infer the most precise valid type argument, and fail if there is no such type either due to a lack of any valid type argument or due to a lack of a most precise one.

We implement this inference soundly and completely with the following steps:

1. Collect the subtyping constraints on the potential type argument imposed by the method signature and the invocation arguments.
2. Use Greenman et al.'s sound and complete algorithm to compute the join of the collected lower bounds [16]. Note that by adopting their Material-Shape Separation, we do not have F-bounded polymorphism [9], thereby avoiding the complications incurred by recursive constraints.
3. Test whether the candidate type satisfies the appropriate shapes. If so, continue to the next step. If not, expand the candidate type according to inheritance and repeat this step. Our design has the property that exploring expansions possible due to variance will not improve the satisfiability of a type, thereby making the search space finite.
4. Test whether the candidate type is a subtype of the collected upper bounds. If so, use it as the type argument. If not, there is no valid type argument.

In the case of languages with intersection types or multiple inheritance, there might be multiple expansions that satisfy the constraints. If so, compare them to determine which one is principal among them, if any.

Conceptually, the programmer can now use `strings:contains` for default local perspective, `strings.contains` for the target's perspective, and `strings.<caseless>contains` for non-standard perspective. Thus we have a clear syntax for distinguishing callee versus caller perspective, as well as functionality for non-standard perspectives.

4.7 Extension Shape-Shifters

We still have not quite addressed the original issue we identified: one would like to be able to use `Iterable` as if it satisfied `Equality`. We can now plug into the above features to make such functionality convenient.

We allow programmers to specify a conditional default shape-shifter, much like an extension method [38]:

```
Iterable<E satisfies Equatable>: satisfies Equatable { //note the ":"
  Boolean equals(Iterator<E> that) {
    if (!length.equals(that.length))
      return false;
    for each ((e1, e2) in zip(this, that))
      if (!e1.equals(e2))
        return false;
    return true;
  }
}
```

Then whenever one uses `Iterator` as a type argument, this shape-shifter is automatically included (if the conditions apply) so that the type argument is `Equatable`. This includes whenever the colon operator infers types: when checking whether a type satisfies a shape, it also checks for conditional default shape-shifters. Thus our original motivating challenge has become a specialized case of a powerful set of features.

4.8 Avoiding Ambiguities

We mentioned before that in order to automatically generate the various implementations of a conditional method, we need to be able to identify which invocations in its code correspond to the conditional. Occasionally this, as well as some of our syntactic shorthands, can be ambiguous. Here we discuss how to identify and resolve those ambiguities.

One ambiguity can arise from the following:

```
interface Bar<out T1, out T2> {
  <super T1, super T2> super T1 baz(super T1 t1, super T2 t2);
}
class Foo<T> implements Bar<T,T> {
  <super T, super T> super T baz(super T t1, super T t2) {
    return t2;
  }
}
```

The signature of `Foo`'s `baz` method is generated using standard substitution. And it type checks. However, it is unsound. A `Foo<Integer>` could be cast to a `Bar<Integer,Object>` on which `bar(1,"One")` could be invoked. The return type would be `Integer`, but the returned value would be `"One"`. The issue comes from the fact that `super T` is supposed to represent a single type variable, but in this case it represents two different type variables. The same ambiguity can occur if substitution would result in the type `super Integer` or other non-appropriately-scoped type variables.

We resolve this by rejecting such ambiguous signatures and by enabling developers to name the variable to manually resolve the ambiguity:

```
class Foo<T> implements Bar<T,T> {
  <T1 super T, T2 super T> T1 baz(T1 t1, T2 t2) {
    return t2; // now the unsoundness is obvious
  }
}
```

The other source of ambiguity arises when the methods granted by the constraining shapes would be available to the type parameter or its subtypes through some other means. These ambiguities can be witnessed in the following:

```
interface Bar<out T> {
  <super T satisfies Cloneable> Pair<super T,super T> qux();
}
interface Foo implements Bar<Integer> {
  <T super Integer satisfies Cloneable> Pair<T,T> qux() {
    T t = 5;
    return (t.clone(), 6.clone()); // which clone for 6?
  }
}
interface Collection<E> {
  <E satisfies Equatable> Boolean contains(E value);
}
class HashSet<E satisfies Hashable> {
  // generalized implementation
  <E satisfies Equatable> Boolean contains(E value) {
    for each (elem in this)
      if (elem.equals(value)) // which equals?
        return true;
    return false;
  }
}
```

In `Foo`'s `qux`, `6` can be cloned in two ways: using the method built into integers, or casting to `T` and using the implicitly supplied shape-shifter. In `HashSet`'s `contains`, it is unclear type-theoretically whether the `equals` call should use the implementation provided at the allocation of the `HashSet` or the implementation provided at the invocation. We could arbitrarily choose a way to resolve these ambiguities, but we may not make the right choice, and the programmer may not even realize there is an ambiguity. Furthermore, some experience suggests that the programmers will expect whatever behavior makes sense to them, regardless of whether this matches the language specification [34].

Our design rejects such ambiguous signatures and enables the programmer to explicitly name the shape-shifter:

```
interface Foo implements Bar<Integer> {
  <T super Integer with shifter satisfies Cloneable> Pair<T,T> qux() {
    T t = 5;
    return (shifter.(t).clone(), shifter.(6).clone());
    //return (shifter.(t).clone(), 6.clone());
  }
}
class HashSet<E satisfies Hashable> {
  // generalized implementation
  <E with shifter satisfies Equatable> Boolean contains(E value) {
    for each (elem in this)
      if (shifter.(elem).equals(value)) //elem.equals(value)
        return true;
    return false;
  }
}
```

Thus in the rare cases where subtyping combined with shape-shifters causes ambiguity, we simply require the programmer to explicitly use the shape-shifter rather than have the compiler attempt to make guesses. Note that this ambiguity can also happen if a type parameter is constrained with an upper bound rather than a lower bound.

At this point in our design, our proposal has only one point of ambiguity: which type argument to infer in colon invocation. This ambiguity is resolved through a simple principle: the most precise candidate is chosen. The programmer does not have to know the algorithm used to derive the solution. Thus our design provides a principled and powerful solution to a complex intersection of subtype polymorphism, parametric polymorphism, and ad hoc polymorphism.

5. Related Work

Kaes invented a form of parametric polymorphism with overloading [21], and Wadler and Blott devised a richer constraint language implemented with dictionary passing, well known as type classes [37]. Wadler and Blott's approach was adopted by the Haskell language [18], where it has proven extremely useful for statically-checked type dispatch.

Our design seeks to unite this work with techniques that developed concurrently within the object-oriented community, both in academia and industry. From academia, we build on proposals for self types, conditional dispatch, and binary methods. From industry, we recognize that declaration-site variance and ad-hoc polymorphism are highly desirable features; furthermore, we leverage the observed tendency towards well-founded inheritance and the separation between types and type constraints to guide our formulation of type classes as shapes.

5.1 Self Types and Binary Methods

We introduced a type `This` to express the fact that shapes act solely as constraints on a material type. But the idea of an explicit type for the self-referential variable within a class or interface has appeared in many earlier languages. Trellis/OWL [31] and Eiffel [27] were two of the first to keep an explicit type for the variable `self`. Later, PolyTOIL [7] and LOOJ [6] provided self types and exact types in a Java-like setting, and JavaGI (discussed below) included self types as one of its many innovations [41]. As for other contemporary object-oriented languages, the .NET CLR maintains a type for each instantiation of a class at run time [22] and Scala has a reserved variable `this` with a field `type` for expressing self types [29].

The most ubiquitous application of self types is for reconciling binary methods with subtype polymorphism. Bruce et al. [5] give a comprehensive description of the problem and offer a few design alternatives. In particular, their suggestion to replace binary methods with external functions is similar to our use of extension methods. More information on the challenges associated with binary methods may be found in the works of Cook [11] and Torgersen [35].

5.2 Conditional Inheritance

Work on conditional methods dates back to the CLU language [25]. Bank et al. later introduced conditional methods to Java in their design for parametric polymorphism [4].

More recently, cJ [19] features both conditional methods and conditional inheritance that erases into Java. Whereas we restrict to shape satisfaction, cJ allows conditioning on arbitrary subtying judgments. However their motivating examples fall into two categories: conditioning upon shape satisfaction, or conditioning upon a bit set encoded into the inheritance hierarchy, which could just as easily be done with empty shapes. Thus we seemingly lose no critical expressiveness with our restriction, and additionally we gain decidability.

5.3 Generalized Interfaces

JavaGI [41] extends Java 5 with so-called generalized interfaces. These generalized interfaces provide the core features of type classes as well as self types, binary methods, conditional methods, and conditional and retroactive interface implementation. Furthermore their formal development includes a small performance study [40] and proofs of decidability, soundness, and completeness [39].

We address similar features in the context of declaration-site variance (as opposed to use-site variance, discussed in the next section) but also extend the traditional notion of a type class to let a single type have multiple implementations of the same type class. JavaGI, like Haskell, requires each type to have at most one implementation of any given type class. We originally strove to provide this same behavior, but realized it was insufficient for accommodating common use cases.

Moreover, JavaGI adds a number of coherency restrictions related to subtyping. For instance, two implementations of the same interface at different types must be accompanied by a third, downward closed implementation and conditions on a subtype's parameter must be implied by conditions on the supertype's parameter [39]. We faced similar challenges, but adopt a different solution: rather than constraining inheritance, we make shape satisfaction explicit and separate from the established concepts of class extension and interface implementation.

5.4 Use-Site Variance

In use-site variance, one does not label type *parameters* as covariant or contravariant. Instead, one labels type *arguments* as covariant or contravariant. For example, the type `Array<out Number>` denotes the covariant subcomponent of `Array`'s signature instantiated with type `Number`. Intuitively, this is an array one can only get numbers `out` of.

Java's wildcards are a form of use-site variance [36], which is why both cJ and JavaGI targeted use-site variance. More precisely, Java's wildcards are a form of existential types [8], which can be used to emulate use-site variance, as done by Scala [29].

Although Java's wildcards are quite powerful, they are broadly considered by industry to be a failed experiment. For one, many find them (and existential types or use-site variance in general) difficult to reason about. More importantly, they are quite burdensome to use. For example, every single use of `Iterable`, and every other interface with a covariant signature, should have its type argument annotated with `out` to provide maximum usability.

Industry trends aside, use-site variance does not address the architectural needs we discussed earlier. It only works well for `inherited` shapes.

Consider the following interface declaration, akin to the architectures provided by prior work:

```
interface Iterable<E>
    <E satisfies Equatable> satisfies Equatable
    <E satisfies Hashable> satisfies Hashable
    <E satisfies Cloneable> satisfies Cloneable
    <E satisfies Summable> satisfies Summable {
  ...
}
```

This seems ideal, until one tries to actually use it with use-site variance. For example, `Iterable<out Number>` has no `clone` method, nor a `plus` method, despite the fact that `Number` satisfies both `Cloneable` and `Summable`. The reason is that `Iterable<out Number>` stands for $\exists\alpha <:$`Number`. `Iterable<`α`>`, and one cannot guarantee that the unknown α satisfies the appropriate shape. So now the advice that `Iterable<Number>` should always be replaced with `Iterable<out Number>` is no longer valid, making one of the few applications of use-site variance that the majority of Java programmers actually understand no longer easy to reason about.

On top of these concerns, designs relying on the simplicity of use-site variance do not address many other issues we discussed. For example, there is still no way to distinguish between the implementation that uses the instance's shape-shifter and the implementation that uses the environment's shape-shifter. So, addressing the issues for declaration-site variance forced us to consider these broader issues as well.

5.5 Haskell's Type Classes

The success of Haskell's type classes are of course what inspired our design [18, 37]. While we envy the elegant simplicity of Haskell's model, we believe our complexities are necessary given our design space. For example, one rarely bundles functionality with data in Haskell, whereas the fundamental notion of an object is to identify data with related methods. Consequently, The Perspective Challenge does not arise much.

Despite the challenges, we are able to make some improvements to Haskell's type classes, again due to the differences in the design space. Haskell's type classes are not overrideable, and the use of inference makes it difficult to add such a feature without introducing ambiguities. Consequently, one needs to manually implement every function using a type class twice in order to provide full functionality: one implementation using the default evidence for the type class and another implementation using manually provided evidence for the type class. And global type-class coherence has proved to be a significant roadblock to a strong module system [23]. Thus some of our motivations are already known problems in the Haskell community.

5.6 Scala's Implicits

Scala's implicits is the object-oriented solution to type classes most adopted by industry. Interestingly, a similar approach to type classes is taken by Agda [12] and Coq [33], and recently proposed for OCaml [42]. The idea is that methods can specify implicit parameters, and programmers can specify rules for generating arguments to implicit parameters [30]. Behind the scenes, a resolution algorithm matches generated instances to implicit arguments. In terms of type classes, the implicit parameter is the implementation of the type class for the evidence to use, and the implicit generators are the generated implentations of type classes.

Although Scala's implicits are quite flexible, our design gains a number of advantages by specializing to type classes. First, all algorithms used by our design are sound and complete thanks to the Material-Shape Separation and to careful consideration on what form constraints can take. Second, Scala's implicits make no effort to prevent or make the programmer aware of ambiguities, and the algorithm/principle for resolving ambiguities is notoriously difficult to understand and has historically been silently buggy. Third, our design can implement erasure when using built-in implementations. Lastly, Scala's implicits use caller perspective because the evidence is generated by the caller's environment, so implicits do not address any of the issues associated with The Perspective Challenge.

5.7 C++'s Concepts

Concepts are a technique for formalizing the expectations a C++ template has on its type parameters [17], similar to how F-bounded polymorphism constrains the type arguments of a class or interface. Their goal was to catch template programming errors before template generation. While the design is in flux, we do not not know of any proposal for conditional concepts [32]. Furthermore, C++ does not have variance or erasure, so concepts do not share the primary challenges we discuss in the paper.

5.8 Rust's Traits

Traits are similar to our shapes and extension methods, also having a notion of a self type and being used to constrain a type [2]. They declare (and optionally implement) a set of functions for the self type and can be used to bound type parameters in other structs, functions, and traits. Like Haskell type classes, Rust traits can be inherited and may only be instantiated once for each type. This global coherence simplifies their implementation, as traits are monomorphized at invocation sites, but consequently avoids many of the challenges and opportunities we mention above.

6. Future Work

We have presented the smallest extension to generics that accommodates common architectures benefiting from type classes. Here we discuss potential extensions to explore.

6.1 Parameterized Shapes

To flexibly accommodate linear algebra, one would want the following shape:

```
shape Scalable<S> {
  This scale(S scalar);
}
```

The parameter enables one to adjust the scalar type to the application, with `Double` and `Complex` probably being the most common candidates. But enabling such a parameter opens a number of questions.

The most interesting question is whether to permit the following:

```
class Double satisfies Scalable<Double> { ... }
```

Of course it makes sense for `Double` to be scalable by itself, but this would suggest that type parameters should also be constrained in terms of themselves, as in `<T satisfies Scalable<T>>`. This some challenges for other features; for example, colon invocation's algorithm relied on the absence of recursive constraints. So it is not clear what tradeoffs should be made in deciding which type arguments to shapes are permissible. Rather than burden the designer with details at this point, we opt to defer parameterized shaped to future work.

6.2 Static Methods

One obvious limitation of our current design in comparison to Haskell is that all shape methods need an instance of the constrained type in order to be accessed. But many type classes in Haskell have functions for creating values from scratch, and many shapes would benefit from the ability to provide instances on the fly. For example, `Iterable` might want a `sum` method when the contained type is `Summable`. But in the case that the `Iterable` is empty, `sum` would need a way to provide the zero value without any other instances present.

Another particularly important example is factories: ways to create new instances of a type. We could accomplish this with the following shape using a *static* method:

```
shape Constructible<P> {
  static This construct(P param);
}
```

Then if a type parameter `T` is known to satisfy `Constructible<Integer>`, one could construct a `T` instance by using `T.construct(0)`, viewing `construct` as a method of the type rather than a method of its instances.

C# has a feature like this [1], taking advantage of reification to create instances. However, its feature is not locally overridable, limiting polymorphism over constructors to only the built-in constructors. While being able to default to built-in constructors provides conveniences and improves efficiency, one would like to apply shape-shifters to the feature. This can already be easily accomplished with the current design with minimal changes. In fact, JavaGI already has this feature [41].

6.3 Conditional Inheritance

In our design, we allow conditional satisfaction clauses, but unlike cJ [19] we do not allow conditional inheritance clauses. The reason is that decidability relies on the fact that shape-satisfaction quickly turns into subtype-checking, which is self contained and terminates due to well-founded inheritance usage [16]. By adding conditional inheritance clauses, then subtype-checking now depends on shape-satisfaction, creating a worrisome loop.

There seems to be one very compelling use case for conditional inheritance, though. We have seen that containment is a fundamentally contravariant concept. And for this reason, often one cannot access an instance's specialized `contains` method through common covariant interfaces. So, for applications that only need containment and not iteration, it makes sense to have an interface specialized to that purpose so that it can serve efficient implementations:

```
interface Container<in E> {
  Boolean contains(E value);
}
```

But many common data structures cannot always implement this interface. They need their data to have additional structure in order to do so, such as with arrays:

```
class Array<E> implements Iterable<E>
    <E satisfies Equatable> implements Container<E> {
  <E satisfies Equatable> Boolean contains(E value) {
    for (index from 0 until length)
      if (this[index].equals(value))
        return true;
    return false;
  }
}
```

Without conditional inheritance, arrays cannot be used as containers even when the appropriate functionality is available. Similar problems occur for collections like linked lists that do not need equality for most of their functionality.

Although general conditional inheritance may be undecidable, it is likely that there is some practical subset that is decidable. We have a guess as to what that subset may be, but unlike our design so far where each future is easily checked to be sound and decidable in isolation, this subset requires developing a whole new well-founded measure that crosscuts features, so we defer it to future work.

6.4 Type Families

Self types are not the only use of shapes; the other common use is type families [15]. Type families are a group of classes or interfaces that coevolve. A particularly common example is graphs, vertices, and edges. JavaGI also supports this feature [41], and it is related to higher-arity type classes [13].

Type families should be easy to incorporate into our design. The biggest challenge might be developing an intuitive syntax. One thing worth noting, though, is that we know of no applications of conditional inheritance for type families. This might bypass the one technical concern: inferring the type arguments for colon invocation when the type parameters are bound together by a type family.

Interestingly, according to Greenman et al.'s analysis of industry code bases [16], if we manage to accommodate type families and parameterized shapes, then the resulting design would contain all of the expressiveness of F-bounded polymorphism that is actually used in practice. This indicates that the various restrictions we impose in our design align with rather than restrict realistic behavior.

6.5 Higher Kinds

Material-Shape Separation has already demonstrated that higher-kinded subtyping is possible for generics [16]. Consequently, it might be possible for our design. Consider the following speculative syntax for the feature:

```
shape<E> Collection
    <E satisfies Equatable> satisfies Equatable
    <E satisfies Hashable> satisfies Hashable
    <E satisfies Cloneable> satisfies Cloneable {
  static This<E> empty();
  void addAll(This<E> that);
}
```

For many data structures, operations using multiple structures can be implemented more efficiently when the structures all have the same implementation. Higher-kinded shapes could help bring homogeneity to the typically heterogenous object-oriented setting.

And, of course, there is one more example to recognize:

```
shape<T> Monad {
  static This<T> unit(T point);
  This<U> bind<U>(Function<T,This<U>> op);
}
```

Thus higher-kinded shapes may be a new way to bring monadic programming to the object-oriented community.

7. Conclusion

Greenman et. al introduced the terms *material* and *shape* to describe the results of their survey, but they refrained from giving a formal definition, noting only that the concepts were orthogonal, and that their separation was sufficient to render subtyping with variance decidable [16]. This work argues that a shape is simply a type class, adapted to the setting of object-oriented programming. We support this point by formalizing shapes as type-level constraints and demonstrate their usefulness through a variety of examples.

A number of difficulties arose during development, prompting the need for quantification over supertypes and overriding with shape-shifters, but in the end we arrived at a design more powerful than what we had set out for. We recognized that the fundamental issue making type classes so difficult to adapt to object-oriented programming is The Perspective Challenge: the fact that the caller and callee may have different perspectives on how types implement a shape. To address this challenge, we enable programmers to conveniently express which perspective they intend to use, and we automatically generate implementations for each choice to avoid redundancy.

We have focused our discussion on design issues rather than formal concerns because much of the formalization can be addressed by building upon powerful techniques provided by the community. Soundness can be demonstrated by translating our programs to use evidence explicitly, much like what is done for type classes. Constrained methods translate to methods requiring explicit evidence, and satisfying classes and interfaces translate to provide static methods constructing that evidence. Erasure is accomplished by recognizing and optimizing for the evidence implemented by simply invoking the corresponding method built into the instances. Decidability follows from the well-founded measure provided by Material-Shape Separation.

While there are many details of our design that we did not discuss, those details can easily be derived by using these formalisms as guidelines when considering the particulars of the choice at hand. Indeed, this project has been especially interesting because the usability challenges far surpassed the technical challenges. The difficulty has been in devising and justifying a comprehensive design.

Our intent is not to push a design upon language developers, but rather to help language developers create their own designs with a comprehensive understanding of the space. In creating our design, we identified challenges overlooked in the past, investigated their underlying principles, and developed new technologies to overcome those challenges. Each of these steps developed a comprehensive understanding, one necessary to anyone considering mixing type classes with declaration-site variance.

References

[1] `new()` constraint (C# reference). https://msdn.microsoft.com/en-us/library/sd2w2ew5.aspx. Accessed: 2015-03-25.

[2] The Rust reference, 1.0.0-alpha.2. http://doc.rust-lang.org/1.0.0-alpha.2/reference.html. Accessed: 2015-03-24.

[3] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA/ECOOP*, 1990.

[4] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.

[5] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, B. Pierce, et al. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, Dec. 1995.

[6] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In *ECOOP*, 2004.

[7] K. B. Bruce, A. Schuett, R. Van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *TOPLAS*, 2003.

[8] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *ECOOP*. Springer, 2008.

[9] P. S. Canning, W. R. Cook, W. L. Hill, W. G. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, 1989.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA*, 2000.

[11] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL*, 1989.

[12] D. Devriese and F. Piessens. On the bright side of type classes: instance arguments in Agda. *ICFP*, 2011.

[13] D. Duggan and J. Ophel. Type-checking multi-parameter type classes. *JFP*, 12(02):133–158, 2002.

[14] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In *ECOOP*. 2006.

[15] E. Ernst. Family polymorphism. In *ECOOP*, 2001.

[16] B. Greenman, F. Muehlboeck, and R. Tate. Getting F-bounded polymorphism into shape. In *PLDI*, 2014. .

[17] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA*, 2006.

[18] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *TOPLAS*, 1996.

[19] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *AOSD*, 2007.

[20] S. P. Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.

[21] S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP*, 1988.

[22] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *PLDI*, 2001.

[23] S. Kilpatrick, D. Dreyer, S. Peyton Jones, and S. Marlow. Backpack: Retrofitting Haskell with interfaces. In *POPL*, 2014.

[24] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE*, 2006.

[25] B. Liskov, E. Moss, A. Snyder, R. Atkinson, J. C. Schaffert, T. Bloom, and R. Scheifler. *CLU Reference Manual*. Springer-Verlag New York, Inc., 1984.

[26] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[27] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[28] M. Odersky and L. Spoon. The architecture of Scala collections. http://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html. Accessed: 2015-03-24.

[29] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[30] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.

[31] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *OOPSLA*, 1986.

[32] J. G. Siek. The C++0x "concepts" effort. *CoRR*, 2012.

[33] M. Sozeau and N. Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*. 2008.

[34] R. Tate. Mixed-site variance. In *FOOL*, 2013.

[35] M. Torgersen. Virtual types are statically safe. In *FOOL*, 1998.

[36] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the Java programming language. In *SAC*, 2004.

[37] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, 1989.

[38] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA*, 2006.

[39] S. Wehr. *JavaGI: a language with generalized interfaces*. PhD thesis, University of Freiburg, 2010. URL http://www.freidok.uni-freiburg.de/volltexte/7678/.

[40] S. Wehr and P. Thiemann. JavaGI in the battlefield: Practical experience with generalized interfaces. In *GPCE*, 2009.

[41] S. Wehr, R. Lämmel, and P. Thiemann. JavaGI : Generalized interfaces for Java. In *ECOOP*, 2007.

[42] L. White and F. Bour. Modular implicits. http://www.lpw25.net/ml2014.pdf. DRAFT: February 22, 2015.