Shape-Shifters: Type Classes for Mainstream Object-Oriented Languages

Abstract

Type classes are an excellent feature of Haskell [Hall et al. 1996]. 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 build on the recent concept of shapes [Greenman et al. 2014] to design a solution compatible with the type systems, implementation constraints, and software architectures of the current major object-oriented programming languages. Decidability is a surprisingly minor part of the challenge due to recent findings. The actual challenge lies in accommodating even the basic architectures that programmers expect from such a solution, especially in the context of declaration-site variance. We delve into key examples from industry and illustrate how they require a much more complex design than one would expect.

1. Introduction

When designing a statically typed object-oriented language, one of the most frustrating walls to hit is type-safe equality. Type-safe equality is the idea of catching nonsense equality comparisons at compile time. For example, if a program contains the expression `5 == "Five"`, the compiler would inform the programmer that there seems to be some mistake.

One way to accomplish this is to have classes provide an `equals` method, whose parameter's type indicates what the class is equatable to, and then make `==` simply a shorthand for calling that method. Unfortunately, such a solution does not interact well with parametric polymorphism, or generics as the concept is referred to in the object-oriented industry community [Thorup 1997] because it provides no way to indicate that some type variable must be equatable.

The next most obvious solution, then, is to define a generic interface for equality:

```
interface Equatable<T> {
    Boolean equals(T that)
}
```

A class like `String` would then have an inheritance clause `implements Equatable<String>` to indicate all strings are equatable with all other strings. Furthermore, a generic method can use `<T extends Equatable<T>>` to indicate that it is polymorphic over all types that are equatable with themselves.

Interestingly, we can generalize this solution a little bit by applying the concept of *variance*. Many common interfaces are either *covariant* or *contravariant*. The common intuition is that covariant interfaces are interfaces you only get things *out* of and contravariant interfaces are interfaces you only put things *in* to. `Equatable`, for example, is contravariant: we put a `T` in to `equals` and we do not get a `T` out of `equals`. One way to express the concept of variance is by using use-site variance [Thorup and Torgersen 1999], but industry widely perceives use-site variance to be a failed experiment, and we will illustrate that the deceptively simple formalization of type classes for use-site variance fails to work for key architectures from industry. We instead express variance by using declaration-site variance [Emir et al. 2006], annotating type parameters with `in` or `out`:

```
interface Equatable<in T> {
    Boolean equals(T that)
}
```

The advantage of contravariance is that, if `Foo` is a subtype of `Bar`, then it is safe to make `Equatable<Bar>` be a subtype of `Equatable<Foo>`: if an `Equatable` can safely accept `Bars` then it can also safely accept `Foos`. In particular, this means that if `Bar` implements `Equatable<Bar>`, then without any effort `Foo` automatically implements `Equatable<Foo>`, so self-equatability is inherited for free.

Sadly, this solution is still not sufficient. Consider a generic interface like `List<Element>`. A list has a sensible definition for `equals` *only if* its type argument `Element` has an appropriate `equals` method.

So, the next most obvious solution is to permit *conditional* inheritance and methods:

```
interface List<Element> {
    ...
} extends Equatable<List<Element>>
  given Element extends Equatable<Element> {
    Boolean equals(List<Element> that);
}
```

Yet, this solution still has problems. Consider the class `Tree`:

```
class Tree implements List<Tree> {...}
```

The question is, does `Tree` extend `Equatable<Tree>`? The answer turns out to depend on whether you allow infinite proofs. In particular, `Tree` extends `List<Tree>`, which ex-

tends `Equatable<List<Tree>>` (and `Equatable<Tree>` due to contravariance) *provided* `Tree` extends `Equatable<Tree>`. Notice that we have gone full circle. While in this case it is possible to detect the circle, in the more general case this is undecidable [Kennedy and Pierce 2007] and causes many compilers to crash [Tate et al. 2011] including ones that claim decidable type checking [Wehr and Thiemann 2009a].

Recently, Greenman et al. discovered a fact very relevant to this example: classes like `Tree` never occur in practice [Greenman et al. 2014]. And, relevant to the broader problem of type classes, interfaces and classes like `List` are used very differently than interfaces and classes like `Equatable`. They called `List`-like classes and interfaces *materials*, and `Equatable`-like classes *shapes*. They characterized the behaviors of materials versus shapes, and proved that no class or interface out of millions of lines of open-source generic-Java code ever mixes these behaviors, an observation they called Material-Shape Separation [Greenman et al. 2014]. These behaviors are easy to describe in Haskell terminology: materials correspond to types, whereas shapes correspond to type classes. Thus, they discovered that even though Java programmers have full freedom to define their own architectures, they all naturally separated their concepts into types and type classes. Furthermore, they were able to use this separation to define a well-founded measure on types such that typing algorithms reduce this measure as they recurse, guaranteeing termination.

In this paper, we integrate the separation between shapes and materials into the language, combining the concept of conditional type classes [Wadler and Blott 1989] with the concept of conditional inheritance. So far we have expressed type-theoretic concerns such as decidability, as have many prior works [Kennedy and Pierce 2007; Tate et al. 2011; Wehr and Thiemann 2009b], but there are many industry concerns to consider as well. These concerns include accommodating common architectures, permitting efficient implementations, avoiding user-perceived ambiguity, and targeting common usage patterns.

In Section 2, we present basic architectures from industry and illustrate how each one makes the design challenge much more complicated than one would realize simply starting a language from scratch. In Section 3, we present our design introducing a variety of features that, through their interactions, accommodate all these architectures. In Section 4, we illustrate how prior works unexpectedly fail to address these architectures, largely because they build upon use-site variance. Our design is the simplest possible for the sample architectures, so in Section 5 we discuss additional features that provide a more powerful adaptation of type classes. Throughout this paper, because the user experience is more constraining and challenging than the usual concerns of soundness and decidability, we focus our exposition on examples and reasoning rather than formalism.

```
shape Equatable {
    This equals(This that);
}

interface Container<Element> {
    Boolean contains(Element value);
}

interface Iterable<out Element> {
    Iterator<Element> iterator();
} extends Container<Element>
  given Element satisfies Equatable {
    Boolean contains(Element value);
}

class Recasings() implements Iterable<String> {
    String text;
    Iterator<String> iterator() {...}
    Boolean contains(String value) {
        return text.toLowerCase()
                .equals(value.toLowerCase());
    }
}
```

Figure 1. Containers and Iterables

2. Targeting Industry

In designing a language feature, it is important to consider the user base. For example, Material-Shape Separation informs us that we can have a separate treatment of materials and shapes without upsetting industry practice [Greenman et al. 2014]. Another finding was that the key type argument for shapes was always the same as the extending class, interface, or variable. That is, no class `Foo` ever had a clause `extends Equatable<Bar>` where `Bar` was not `Foo`. This suggests that, for shapes, we could instead have more convenient clauses like `Foo satisfies Equatable` without removing any expressiveness actually used by industry. The idea is that a shape `Equatable` is allowed to use a `This` type variable, and the clause `Foo satisfies Equatable` indicates that `Foo` implements `Equatable` with `Foo` in place of `This`.

2.1 Architectures and Variance

Now that we have an industry-motivated syntax, consider the architecture in Figure 1 that illuminates some of the usability and soundness challenges. A `Container` is something that can be asked if it contains values. Hashsets, arrays, and red-black trees are well-known examples. An `Iterable` is something that an iteration of values can be retrieved from. Any `Iterable` of `Equatable` values can be made into a `Container` by iterating over the values and checking them for equality one by one. For example, the implementing class `Recasings` represents all ways to recase the letters of a `text` and can specialize its implementation of `contains` for significant performance improvement by just comparing the lower-cased versions of `text` and `value`.

While this looks ideal, it is unfortunately unsound. The class `String` often implements `Iterable<Character>`. Since `Character` is `Equatable`, so is `Iterable<Character>`. However, arbitrary instances of `Iterable<Character>` do not have a `toLowerCase` method. Consequently, the following code would produce a run-time error:

```
Iterable<Iterable<Character>> listofchars
  = Recasings("Hello World");
Iterable<Character> chars
  = singleton<Character>('x');
listofchars.contains(chars);
```

The astute reader may have noticed the source of this unsoundness: the `contains` method in `Iterable` does not have a covariant signature. Consequently, due to the indirections made possible by subtyping, `Recasings` cannot be assured to be passed a `String` for its `contains` method. While one might say that this should simply not be permitted, it is important to recognize that equality is not covariant either, and the main motivating example for these features is type-safe equality. Thus a good design needs to accommodate conflicting variances.

2.2 Variance and Conditions

It turns out unsoundness can occur even if the proper variances are maintained. Consider the architecture in Figure 2. Object-oriented languages are typically stateful. Occasionally it is useful to get a new copy of some object whose state is independent from the original. A class satisfies `Cloneable` to indicate that this is possible. For example, an `Array` can be cloned by creating a new `Array` and filling it with clones of the original's elements. A class like `File`, though, whose state is tied to a physical location on disk, cannot be cloned into an independent `File` (at least not without duplicating the content on disk). So, `File` does not satisfy `Cloneable` directly, though it still implements the `clone` method required by the inherited `Iterable` because `String` is `Cloneable`. The inherited `clone` method only requires a `Iterable<String>`, so `File` can implement it by allocating an `Array` rather than another `File`.

Once again, this seems sound. Notice in particular that `clone` has a covariant signature, so we will not encounter the same problem as before. Nonetheless, the following code leads to a run-time error in a reified implementation:

```
Array<File> files = Array<File>(1);
files[0] = File("voodoo.exe");
Iterable<Iterable<String>> strings = files;
strings.clone();
```

The error occurs during `strings.clone()`. Since `strings` is actually an `Array<File>`, the first thing its clone method does is create a new `Array<File>`. Then, it sets the `File` at the 0^{th} index to be the clone of `strings`' 0^{th} index. However, that clone is not a `File` since `File` is not `Cloneable`; it is only an `Iterable<String>`. Thus the assignment into the `File` array causes a run-time exception. Note that this problem is not limited to reified languages [Douence and Südholt 2001]; a similar problem could happen in a language with type erasure [Bracha et al. 1998] if `Array`'s `clone` method were to leak the clone, say by assigning it to some field of `Array`.

```
shape Cloneable {
    This clone();
}

interface Iterable<out Element> {
    Iterator<Element> iterator();
} satisfies Cloneable
  given Element satisfies Cloneable {
    Iterable<Element> clone();
}

class Array<Element>(Integer length)
  implements Iterable<Element> {
    ...
} satisfies Cloneable
  given Element satisfies Cloneable {
    Array<Element> clone() {
        Array<Element> clone
          = Array<Element>(length());
        for (Integer i from 0 til length())
            clone[i] = this[i].clone();
        return clone;
    }
}

class File implements Iterable<String> {
    String filename;
    ...
    Iterable<String> clone() {
        Array<Element> clone
          = Array<Element>(length());
        for (Integer i from 0 til length())
            clone[i] = this[i].clone();
        return clone;
    }
}
```

Figure 2. Cloneable and Arrays

Again, this is an important use case, so a good design needs to accommodate at least some approximation of this architecture, even if its current rendition is unsound.

2.3 Locally Overriding Implementations

Although many classes come with an implementation of equality and other shapes, occasionally one does not want to use the built-in implementation. The classic example of this is sorted lists of strings. One would like to use the provided implementations for sorting lists, such as quicksort, but have them use non-built-in orderings such as reverse ordering or case-insensitive ordering. Similarly, one would like to use implementations of sorted lists, such as red-black trees, but have them use non-built-in orderings. In other words, programmers often need to be able to *locally override* the implementations of shape methods. Java has `Comparator` for

this purpose, but Java libraries must explicitly and manually enable programmers to provide a `Comparator` rather than use `Comparable` *and* suffer an extra layer of dynamic dispatch even when using the default implementations. Furthermore, these `Comparators` can be annoying to compose together when working with complex data structures.

Sometimes a class does not even have a built-in implementation of a shape but the programmer has one they want to use. For example, `java.util` provides `IdentityHashMap`, an implementation of `HashMap` that uses reference equality and address hashing and can be defined on any type. This suggests that programmers occasionally need to be able to *locally provide* implementations of shape methods.

A good design of shapes should provide a facility for locally overriding/providing shape implementations, preferably in a way that is unambiguous, adds no overhead when using default implementations, is convenient even for complex types. Note that local overriding is possible to implement efficiently because shapes are no longer types. In particular, with declaration-site variance, `Array<String>` would traditionally be a subtype of `Iterable<Equatable<String>>`, and it would be impossible to efficiently guarantee that those implementations of `Equatable` would use the locally provided implementation rather than the default one for `String`. For this reason, material inheritance cannot be locally overridden, at least not in a practical manner.

2.4 Overriding and Ambiguity

For the moment, let us discard the issues with variance and instead focus on ambiguity. As we have discussed, one can implement `contains` for any `Iterable` provided its `Element` satisfies `Equatable`. We might express this using conditional methods as in the following architecture:

```
interface Iterable<Element> {
    ...
    <Element satisfies Equatable>
      Boolean contains(Element value);
}
```

Now, consider an `Iterable<String>` variable `strings`. A programmer might expect `strings.contains("Hello World")` to indicate whether `strings` contains `"Hello World"`. However, if `strings` represents an instance of `IdentityHashMap<String>`, then this check would always fail because the constant has its own unique reference.

One might argue that `IdentityHashMap` is to blame for violating the principle of behavioral subtyping [Liskov and Wing 1994]. Another might argue that the programmer is to blame for making incorrect assumptions about the behavior of `Iterable.contains`. This situation is quite common, though, with many practical uses for both behaviors. We argue that the language design is to blame for conflating the concepts of *standard/generic implementations* and *built-in/specialized implementations*. That is, sometimes the programmer wants the implementation that makes sense for arbitrary instances of that interface, and other times the programmer wants the implementation specified by that specific instance. Ideally, a language design would be able to separate these concepts and provide a convenient and clear means for the programmer to indicate which concept they intend.

2.5 Local Overriding and Specialization

The above syntax for conditional methods is chosen to enable another feature: locally overriding the generic's own type argument's implementation of a shape. Suppose `hashes` is an `IdentityHashMap<String>`, and a programmer, well aware of its default implementation of `contains`, would like to check whether `hashes` contains a string equivalent to `"Hello World"`. Ideally, the programmer could do so by calling `hashes.<default>contains("Hello World")`, indicating that the implementation needs to use the default equality for strings rather than the equality built into `hashes`. Similarly, `hashes.<caseinsensitive>contains("Hello World")` would indicate that the programmer wants case-insensitive equality.

This introduces a new problem, though. `hashes` has a very efficient implementation of `contains` *provided* it is using reference equality. So there needs to be some mechanism for class implementations to know when they can use their preferred equality. This is especially important for sorted lists, because in addition to equality they need to rely on the fact that they are implemented for a type that is also comparable. Thus we have yet one more goal for our design in order to accommodate even the most common cases.

2.6 Erasure and Conditions

Last but not least, for efficiency purposes we would like to support erasure of shapes, at least whenever the default implementations are used. Otherwise, every `Array<Integer>` would have to somehow package information describing the `Integer`'s default implementations for various shapes, we would frequently need to construct these implementations as the program executes, and every call to a shape method would suffer an extra layer of dynamic dispatch. These concerns are especially important for languages that already have type erasure.

3. The Language Design

Once one considers the architectures that need to be accommodated, type classes become surprisingly challenging to integrate into object-oriented languages. What we present here is complicated, requiring adding multiple features that each interact with each other, but any simplification makes some important architecture impossible to implement, at least without the scapegoats of run-time casts and reflection.

Our design incorporates five features: integrated support for quantification over all supertypes/subtypes of a type parameter, conditional methods and inheritance clauses, local overriding via shape-shifters, extension methods and shape-shifters, and explicit delegation to the type-checker to auto-generate implementations. All these features are necessary to support the basic architecture in Figure 3 and its uses.

Shapes

```
inherited shape Equatable {
    Boolean equals(This that);
}

shape Cloneable {
    This clone();
}

inherited shape Comparable ext Equatable {
    Boolean leq(This that);
}

inherited shape Hashable ext Equatable {
    Integer hash();
}

shape Addable {
    This plus(This that);
}
```

Classes

```
class Array<E> impl List<E>
  <E sat Cloneable> sat Cloneable
  <E sat Addable> sat Addable {
    ... // implements methods without defaults
    <super E sat Cloneable>
      Array<super E> clone() {...}
    <super E sat Addable>
      Array<super E>
        plus(Indexed<super E> that) {...}
}

class HashSet<E sat Hashable>
  impl Set<E> sat Hashable
  <E sat Cloneable> sat Cloneable {
    ... // implements methods without defaults
    // specialized to use E's hash
    Boolean contains(E value) {...}
    <super E sat Cloneable>
      HashSet<super E> clone() {...}
}

class RedBlackTree<E sat Comparable> impl Set<E>
  <E sat Hashable> sat Hashable
  <E sat Cloneable> sat Cloneable
    ... // implements methods without defaults
    // specialized to use E's leq
    Boolean contains(E value) {...}
    <super E sat Cloneable>
      RedBlackTree<super E> clone() {...}
}
```

Shape-Shifters

```
interface Comparator<in T> shifts<T.Comparable> {
    Boolean areLeq(T left, T right)
      shifts<left.leq(right)>;
}

class Reverse<T sat Comparable> impl Comparator<T> {
    Boolean areEqual(T left, T right) {...}
    Boolean areLeq(T left, T right) {...}
}
```

Interfaces

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

interface Iterator<out E> {...}

interface Iterable<out E> {
    Iterator<E> iterator();
    Integer length() {...}
    <super E sat Equatable>
      Boolean contains(super E value) {...}
}

interface Indexed<out E> ext Iterable<E>
  <super E sat Equatable> sat Equatable
  <super E sat Hashable> sat Hashable {
    E get(Integer index);
    <super E sat Equatable>
      Boolean equals(Indexed<super E> that) {...}
    <super E sat Hashable> Integer hash() {...}
}

interface List<E> ext Indexed<E>
  <E sat Equatable> ext Container<E>
  <E sat Cloneable> sat Cloneable
  <E sat Addable> sat Addable {
    void add(E value);
    <super E sat Cloneable>
      List<super E> clone() {...}
    <super E sat Addable>
      List<super E>
        plus(Indexed<super E> that) {...}
}

interface Set<E> ext Iterable<E>, Container<E>
  sat Equatable {
    // specialized for Sets
    Boolean equals(Set<E> that) {...}
}
```

Extensions

```
Iterable<E sat Equatable>: sat Equatable {
    // can use the this keyword
    Boolean equals(Iterable<E> that) {...}
}
Iterable<E sat Hashable>: sat Hashable {...}
Iterable<E sat Cloneable>: sat Cloneable {...}
Iterable<E sat Addable>: sat Addable {...}

Indexed<E sat Cloneable>: sat Cloneable {...}
Indexed<E sat Addable>: sat Addable {...}

interface Equator<in T> shifts<T.Equatable> {
    Boolean areEqual(T left, T right)
      shifts<left.equals(right)>;
}

interface Hasher<in T> ext Equator<T>
  shifts<T.Hashable> {
    Integer hashOf(T value) shifts<value.hash()>;
}
```

Figure 3. Example architecture using our design (using abbreviations)

3.1 Shapes

We provide a variety of shapes in Figure 3. A shape can use the type variable `This`, which intuitively represents a self type [Bruce et al. 1998]. When a class or interface satisfies a shape, it must implementing the methods of the shape with `This` instantiated to the satisfying class or interface.

The shapes `Equatable`, `Comparable`, and `Hashable` all have signatures that are contravariant with respect to `This`. A contravariant shape has the property that, if a type has an appropriate implementation of the shape, then all subtypes can reuse that exact same implementation to satisfy the shape themselves. For this reason, contravariant shapes are permitted to be labeled `inherited`. If a shape is labeled `inherited`, and a class or interface has a clause for that shape, then all subclasses and subinterfaces automatically inherit that clause. This is why `List` does not specify a clause for `Equatable`; it inherits the clause from `Iterable`. Furthermore, because `Comparable` extends `Equatable`, any class or interface satisfying `Comparable` automatically also satisfies `Equatable`.

The shape `Cloneable` has a signature that is covariant with respect to `This`. This means that clauses cannot be automatically inherited, but a common practice is to restate the clause and reimplement the method with a stronger signature.

The shape `Addable` has a signature that is invariant with respect to `This`. This makes it unlikely for a subclass to satisfy `Addable` when a superclass satisfies `Addable` because their method signatures would often conflict. Such behavior may be more common in a language with multimethods, in which case a `Natural` could have a `plus` method that returns another `Natural` when given a `Natural` and otherwise returns an `Integer` when given an arbitrary `Integer`.

Although not needed in Figure 3, shapes can have type parameters. Those type parameters can have variance, though they cannot be constrained. In our design, constraints are never used for type validity, only for validating calls to constructors and ensuring methods are available, so there is no need for a shape or interface to constrain its type parameters. Because we permit conditional inheritance, the type arguments to shapes must be part of the relation that must be well-founded as per Material-Shape Separation. For example, a class satisfying a parameterized shape cannot use itself as a type argument to the shape, which is consistent with industry usage of shapes [Greenman et al. 2014].

Lastly, for the sake of convenience, shapes can provide default implementations of methods. For example, a full definition for `Comparator` would include convenience methods for checks such as strict inequality.

3.2 Polymorphism over Supertypes

Because so many common examples contradict the formally permitted uses of variance, we provide a way to circumvent the restrictions of declaration-site variance. We reuse the technique of making methods generic over a type variable whose lower bound is a covariant type parameter of the class in order to meet the requirements of covariance [Emir et al. 2006; Odersky 2014]. But we go one step further and actually integrate the technique into the language so that other features may recognize and specialize for these particularly common cases.

For example, `Iterable` cannot have the following method:

```
Iterable<E> concat(Iterable<E> tail);
```

This method signature is not covariant with respect to `E`. In Scala [Odersky 2014], this is addressed using lower bounds:

```
Iterable<S> concat<S super E>(Iterable<S> tail);
```

Surprisingly, this is covariant with respect to `E`. It ensures that any subclass's implementation of `concat` is not specialized to its specific instantiation of `E`.

In our design, we express the above with the following:

```
<super E> Iterable<super E>
  concat(Iterable<super E> tail);
```

The `<super E>` before the signature indicates that the method is polymorphic over any supertype of `E`. Every use of `super E` in the signature (and in the body of the implementation, if there is one) refers to the same supertype of `E`. In other words, `<super E>` is shorthand for Scala's technique, with every use of `super E` referencing the automatically generated type variable `S`.

Explicit Type-Arguments and Type-Argument Inference

In order to call such a method, one provides a type instantiation via the syntax `foo.<Type>concat(bar)`, then the type checker ensures that all the necessary constraints are satisfied. One might be tempted to use `foo.concat(bar)` and infer `Type`, but there are a few reasons this is problematic. First, in a reified language the actual `Type` affects how the program executes. For example, `foo` may allocate an `Array<Type>`, and `Array` is invariant, so this will affect various casts and operations down the road. This is especially problematic for languages with gradual typing, since then the inference would be done at run time and the behavior of the program would heavily rely on the specific dynamic types of `foo` and `bar`.

Another reason to require `Type` to be explicitly passed is that we want to allow `foo` to have an implementation of `concat` specialized to its *exact* type argument at allocation. We showed earlier how classes like `IdentityHashMap` and sorted lists rely on such a specialized implementation for `contains` to accomplish significant performance improvements. Thus, the unparameterized `foo.concat` must refer to `foo`'s specialized implementation of `concat`. In this case, since `Iterable` is covariant, such an invocation cannot be made safely. However, if a method is parameterized by `<super T>` and the containing class or interface is invariant with respect to `T`, then the specialized implementation can be safely invoked with `T` as the instantation of `super T`.

From an academic standpoint, this requirement of explicit type arguments seems necessary, but from an industry standpoint it is intolerable. There will typically be an

obvious type argument to use, and that type argument will typically be correct, so a programmer would be quite frustrated to always have to type out what seems obvious. Our compromise is to introduce a new way to invoke methods on an instance: `foo:concat(bar)`. The colon informs the type checker to automatically fill in the hole with what the obvious thing. More formally, it supplies the most precise `Type` such that `foo.<Type>concat(bar)` type checks statically, a process that is decidable thanks to Material-Shape Separation. In general, the idea is to use colon to indicate that the *static* type checker should step in, applying only the knowledge in scope at compile time. We will expand the uses of this operator as we introduce more features.

As we implied earlier, uses of `super T` are always covariant with respect to `T`. One could define a dual concept, `sub T`, uniformly quantifying over arbitrary subtypes of `T`. However, we found no practical use for it, and it causes complications with the features below, so we elide it.

3.3 Conditionals

A conditional method indicates that the method may only be implemented and called under certain conditions. We found it sufficient for these conditions to only depend on types satisfying shapes, and for consistency with other features we restrict them to that degree of expressiveness. Typically, the type being constrained is either a type parameter or a `super` of one. Of these two cases, Figure 3 illustrates that constraining `super` of some type parameter is quite common.

To better understand this practice, focus on `Array`'s `clone`:

```
class Array<E> {
    <super E satisfies Cloneable>
      Array<super E> clone() {...}
}
```

Note that `Array` is not covariant with respect to `E`, so it does not need to use `super E`. However, by using `super E` we get a more powerful signature. For example, given a variable `files` with type `Array<File>`, with this more expressive signature we can call `files.<List<String>>clone()` to get an `Array<List<String>>` (provided `File` implements `List<String>`).

Note that `Array<E>` does have an appropriate `clone` method when specifically `E` satisfies `Cloneable`. Furthermore, there is an established semantics for cloning arrays that every subclass of `Array` would abide by. Consequently, we can safely and sensibly say that `Array` satisfies `Cloneable` if `E` satisfies `Cloneable`. We express this with the conditional clause `<E satisifies Cloneable> satisfies Cloneable`.

These conditional clauses are used heavily. Another important example is that `Indexed<out E>` has the clause `<super E satisfies Equatable> satisfies Equatable`. This indicates that `Indexed<E>` is `Equatable` if any supertype of `E` is `Equatable`. Because `Indexed` is covariant with respect to `E`, it must express its conditions in terms of `super E`. Consequently, its `equals` method must only require that `super E` is `Equatable`. Also, because we do not permit `sub E`, this means there is no way to condition upon a contravariant type parameter, though we know of no application of such a feature.

The last important example is that the interface `List<E>` *extends* the interface `Container<E>` when `E` satisfies `Equatable`. This is motivated particularly by `Array`, which is an essential data structure that should be usable as a `Container` whenever a definition of equality on its elements can be given. Thus, it seems necessary to support conditional inheritance as well as conditional methods and satisfaction.

Generating Default Implementations

Because bounded polymorphism enables parts of the program to reference instances without realizing their methods are conditional, each such instance must have an implementation of those methods that does not require type-class-like evidence as an input. Consequently, in our language design, the implementation auto-generates unconditional implementations of all such methods. The unconditional implementation inlines the shape implementantions associated with the relevant type-argument at the allocation site of the instance. If the allocation site only uses the built-in implementations, then the unconditional implementation will just use the built-in implementations, fulfilling type erasure. If the allocation site opts to locally override the shape implementations, as discussed in Section 3.4, then the implementation will automatically allocate fields to store those implementations.

To be sound, the auto-generated unconditional implementation is only accessible where the condition is guaranteed to hold, say when abstracted by some type variable `X` that satisfies the associated shape, or when accessed directly as was described in Section 3.2. Importantly, this specialized version can be overridden. For example, `Set` provides a specialization of `equals` that takes advantage of the unconditional/specialized `contains` method.

Restricting Conditional Inheritance

The unconditional implementations are one of the two reasons we impose a restriction upon conditional inheritance and satisfaction clauses. That restriction is that, if a conditional inheritance or satisfaction clause constrains a (`super` of) a covariant type parameter, then the constraining shape must be `inherited`. This guarantees that the actual run-time type argument also satisfies the shape if the condition holds.

To demonstrate the need for this restriction, suppose you allocate an `Array<Integer with intadd>`, which informs the `Array` to use `intadd` in its own implementation of addition. Then, assign that array to a variable `nums` of type `Iterable<Number>`, where `Number` is a superclass of all the primitive number types and is `Addable`. If we permitted `Iterable` to be conditionally `Addable`, then we could add `nums` to some other `Iterable<Number>`, say a list of `Floats`. This would eventually result in passing a `Float` to `intadd`, which expects an `Integer`, demonstrating unsoundness.

The second reason for this restriction has to do with type checking. Suppose `Iterable` is conditionally `Cloneable`. Then, consider whether `Iterable<File>` is `Cloneable`. The naïve answer would be no, since `File` is not `Cloneable`. However, `Iterable` is `Cloneable` provided any *supertype* of `File` is `Cloneable`. In particular, `List<String>` is a `Cloneable` supertype of `File`. Thus, relaxing this restriction would significantly increase the complexity of basic type checking.

Lastly, we can extend the colon operator to accommodate conditional methods. As before, it first determines the most precise type argument that satisfies the constraints *except* for the conditions. Then, it determines the most precise supertype that also satisfies the conditions and automatically generates whatever evidence needs to be supplied given the defaults in the *current* scope. This avoids accidentally calling `.contains` on what is secretely an `IdentityHashMap` using a non-default equality; the colon operator would call `.<default>contains` which implements the general-purpose behavior. One thing to note is that determining the most precise supertype satisfying a constraint is decidable due to the measure provided by Material-Shape Separation. In particular, because conditions cannot depend on contravariant type parameters, it is actually possible to (in the worst case) enumerate all supertypes given only by covariance and check them for condition satisfaction.

3.4 Local Overriding with Shape-Shifters

A shape-shifter shows how to make a type satisfy a shape. For example, any implementation of Java's `Comparator<T>` shows how to make type `T` satisfy `Comparable`. There are many syntaxes a language may choose for defining and using shape-shifters and specifying a default shifter interface for a given shape; we illustrate one such syntax in Figure 3 with the default inferred from uniqueness.

To understand our syntax, examine `Equator<in T>`. It has a clause `shifts<T.Equatable>`, meaning it can be used wherever some condition requires `T` to be `Equatable`. Its method `areEqual` has a clause `shifts<left.equals(right)>`, meaning that `areEqual` shows how `left` implements the `equals` method required to satisfy `Equatable`.

Example Applications

The class `Reverse` in Figure 3 demonstrates how to implement a shape-shifter. One can use it in a construction `RedBlackTree<String with Reverse<String>()>()` to create a reverse-sorting list of strings. The class `RedBlackTree` requires its argument to satisfy `Comparable`. If one were to construct just `RedBlackTree<String>()`, then the resulting `RedBlackTree` implementation would simply use the implementations of `equals` and `leq` built into the `String` instances. But, by using `String with Reverse<String>()` as the type argument, the language implementation automatically generates a `RedBlackTree` that uses the allocated `Reverse<String>`'s `areEqual` and `areLeq` whenever `RedBlackTree` used the `equals` and `leq` methods on its type parameter's instances. Furthermore, if ever `RedBlackTree` in its implementation calls other methods or constructs instances of other classes relying on the fact that its type parameter satisfies `Comparable`, the auto-generated implementation will propagate the shape-shifter rather than revert to the built-in implementation.

This trick can also be applied to classes with conditional clauses. For example, one can construct an array via `Array<String with caseinsensitive>()` to get a simple implementation of a list of strings whose specialized methods use case-insensitive equality and hashing. One can even nest shape-shifter constructions. Given a variable `paper_authors` of type `Iterable<Iterable<String>>`, one can quickly whether check there is a paper with the poorly-capitalized author list `authors` by using the following method invocation:

```
paper_authors.<Iterable<String> with
               SetEquality<String with
               caseinsensitive>()>>(authors)
```

This example illustrates two things. First, we should let a shape-shifter be used as an abbreviation for the most-precise type it shifts along with itself. Then we could use the following much more concise method invocation:

```
strings.<SetEquality<caseinsensitive>()>(authors)
```

Second, we can exploit the fact that conditional methods are necessarily polymorphic in our design so that we can use our own custom implementation of a shape rather than the one built into `strings`. And, if `strings` were instead `RedBlackTree<Iterable<String>>`, then we still have its specialized implementation of `contains` available to us if we prefer that one.

In our design, shape-shifters are first-class citizens, granting the programmer a lot of freedom. This enables each column of a spreadsheet to have its own `Comparator` that can be changed as the program executes.

Disambiguating Multiple Implementations

Shape-shifters can also be used to resolve certain ambiguities that arise when implementing conditional methods. The condition might be that some type `Type` satisfies some shape `Shape`, *and* the environment may demonstrate that `Type` already satisfies `Shape`. The issue that arises is that the condition might be provided an implementation different from the one supplied by the environment.

By default, we forget the implementation provided by the environment: `Type` and all its subtypes are locally forgotten to have their method signatures with names coinciding with methods of `Shape`, and instead are limited to the (possibly weaker) signatures guaranteed by `Type` satisfying `Shape`. Furthermore, in this scope whenever `Type` (or its subtypes if `Shape` is inherited) is required to satisfy `Shape`, then we provide the implementation provided by the condition rather than the environment.

We use this default both because we found it to be more common and because we can provide a convenient syntax for locally overriding the default behavior. When implementing

a conditional method, we permit one to phrase the condition as `<shifter : Type satisfies Shape>`. In this case, `Type` retains its implementation of `Shape` provided by the environment. Whenever the programmer would prefer to use the implementation provided in the condition, they explicitly call the associated methods on `shifter`. It is also possible for `Type` to be something like `super Integer` because type parameters can be instantiated by inheriting classes. Such a term is ambiguous, so we require the programmer to explicitly name the implicit type parameter with the alternate clause `<S super Integer sat Equatable>`.

Shape-Shifters and Types

In our design, shape-shifters do not actually affect the type of an expression; they only affect the implementations of generic methods and classes. So both of the constructions `Array<Strings with caseinsensitive>()` and `Array<String with default>()` result in just an `Array<String>`. This means we do not have complications with allowing first-class shape-shifters, and we have no need to reason about shape-shifter equality. We can mix instances using different shape-shifters, and we do not have to distinguish between `Array<String with default>` and `Array<String with ?>`.

There is one subtlety to address, though. `Function<In,Out>` cannot generally implement equality. However, one can implement equality for the type `Function<Boolean,Integer>`, say as a shape-shifter `finitefun`. Then, suppose one constructs an array using `Array<Function<Boolean,Integer> with finitefun>`. This `Array` does have an implementation of `equals` and `contains`, and it safely satisfies `Equality` and implements `Container`. But, if we completely discard the shape-shifter in its type, we forget these facts because it would be unsound to assume an arbitrary `Array<Function<Boolean,Integer>>` has these methods. So, we allow shapes to be used as adjectives on type arguments indicating that the type argument was instantiated with an appropriate shape-shifter and the relevant conditonal clauses are satisfied. In this case, we would assign our constructed instance the static type `Array<Equatable Function<Boolean,Integer>>`, indicating that the `Equatable` condition holds, making it a subtype of `Container<Function<Boolean,Integer>>`.

3.5 Extension Shape-Shifters

One might be surprised to notice that `Iterable` does not satisfy `Equatable` in Figure 3, especially since `Equatable` is `inherited` and so there is no issue with the covariance of `Iterable`. The reason is that subinterfaces of `Iterable` have different interpretations of equality. `List` interprets equality as each index references the same elements — order is important. `Set`, on the other hand, interprets equality as both sets contain the same elements — order is irrelevant. Thus, an interface should only satisfy `Equatable` if all instances would have the same interpretation of equality.

One would still like to have an equality for arbitrary iterables, though. Even though it is unsound to tie the implementation to instances, it is possible to implement `Cloneable` for arbitrary iterables (by cloning into a new `Array`). Of course, one could use shape-shifters, but this would require manually specifying the shape-shifter every time.

We address this issue by introducing extension shape-shifters, as illustrated in Figure 3. The extension syntax `Iterable<E sat Equatable>: sat Equatable` indicates that, if `E` satisfies `Equatable`, then the type `Iterable<E>` satisfies `Equatable` using the contained implementation. The body provides implementations for the required methods, using `this` just as if the method were built into the instance.

We use a colon in the syntax because this feature is integrated with our colon operator. When using the colon operator to call conditional methods, it determines the most precise type that satisfies the condition. By providing these extension shape-shifters, that type can now be an `Iterable` of something. Similarly, when calling a conditional method and explicitly providing the type, if one provides `Iterable<String>` then the type checker will automatically generate the shape-shifter from the extension. Thus, the type `Iterable` effectively satisfies the shape even if the implementation is not built into its instances.

4. Related Work

Kaes invented a form of parametric polymorphism with overloading [Kaes 1988], and Wadler and Bloat devised a richer constraint language implemented with dictionary passing, well known as type classes [Wadler and Blott 1989]. This approach was later adopted by the Haskell language [Hall et al. 1996].

4.1 Self Types

Many languages have included explicit types for `This` or `self`, both in interfaces and classes. Trellis/OWL [Schaffert et al. 1986] and Eiffel [Meyer 1992] were two early languages that kept an explicit type for the variable `self`. Later, PolyTOIL [Bruce et al. 2003] and LOOJ [Bruce and Foster 2004] provided self types and exact types in a Java-like setting. The .NET CLR also maintained a type for each instantiation of a class [Kennedy and Syme 2001]. This was possible because the .NET virtual machine, unlike the JVM, was designed to support reified polymorphism and thus did not erase type variable information.

The most ubiquitous application of self types is binary methods, such as equality and addition. Bruce et al. [Bruce et al. 1995] give a comprehensive description of the problem. More information on the challenges associated with binary methods may be found in the works of Cook [Cook et al. 1989] and Torgersen [Torgersen 1998].

4.2 Conditional Inheritance

Work on optional methods dates back to the CLU language [Liskov et al. 1984]. Drawing on that work, Bank et al. proposed extended Java with conditional methods to better support parametric polymorphism [Bank et al. 1997].

Our design, however, is most closely related to cJ [Huang et al. 2007] and JavaGI [Wehr et al. 2007], both of which explore the concept of conditional inheritance. cJ extends Java with conditionals for statically restricting the supertypes, fields, and methods of a class or interface. It then erases everything at compile time, much like generic Java [Bracha et al. 1998]. JavaGI, on the other hand, uses a `This` type and provides ways to generate implementations much like our shape-shifters. Our design can be viewed as intersecting the approaches of these two proposals, providing a design capable of both erasure and custom implementations.

There are some key differences between our design and cJ and JavaGI, though. The conditions in cJ are conditioned upon subtyping, whereas we only condition upon shape satisfaction. Looking through cJ's motivating examples, they 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 seem to lose no critical expressiveness by not conditioning upon arbitrary subtyping, and we gain decidability properties not enjoyed by cJ.

JavaGI, on the other hand, suffers the same limitation as traditional type classes: every type can only have one implementation of a given shape. If a type were to have multiple, then there needs to be some system for resolving ambiguities, and a designer would have to work hand-in-hand with groups of programmers to design that resolution mechanism to match their expectations. This, of course, assumes there is one; programmers often adapt their expectation to the specific instance at hand rather than using a uniform algorithm for determing what to expect [Tate 2013].

There is one more significant difference between our design and cJ and JavaGI: they use use-site variance instead of declaration-site variance. Usually a design for use-site variance easily extends to declaration-site variance. Below we illustrate that such an extension is not so easy in the case of conditional inheritance. Or, possibly more accurately, the techniques relevant to programmers do not work well with just use-site variance.

4.3 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 you can only get numbers `out` of.

Java's wildcards are a form of use-site variance [Torgersen et al. 2004], which is why both cJ and JavaGI targeted use-site variance. More precisely, Java's wildcards are a form of existential types [Cameron et al. 2008; Pierce 2002], which can be used to emulate use-site variance, as done by Scala [Odersky and Zenger 2005].

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 architecture akin to the architectures provided by prior work:

```
interface Iterable<E>
  <E sat Equatable> sat Equatable
  <E sat Hashable> sat Hashable
  <E sat Cloneable> sat Cloneable
  <E sat Addable> sat Addable {
    ...
}
```

This architecture 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 `Addable`. The reason is that `Iterable<out Number>` stands for $\exists\alpha <: \texttt{Number}.\ \texttt{Iterable<}\alpha\texttt{>}$, 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 create a design that also addresses these broader issues.

5. Future Work

We have presented the smallest extension to generics that fulfills critical architectures benefiting from type classes. Here we discuss potential additional extensions one might.

5.1 Static Methods

In the examples we discussed, one always has access to an instance of the type satisfying the shape we want to use. This may not always be the case. One 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);
}
```

C# has a feature like this, taking advantage of reification. However, its feature is not locally overridable, limiting polymorphism over constructors to only the built-in constructors. While being able to default to the built-in constructors provides conveniences and improvse efficiency, one would like 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 [Wehr et al. 2007].

5.2 Type Families

Self types are not the only use of shapes; the other common use is type families [Ernst 2001]. 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 [Wehr et al. 2007], and it is related to higher-arity type classes [Duggan and Ophel 2002].

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: efficiently inferring the type arguments for the colon operator when the type parameters are bound together by a type family.

5.3 Higher Kinds

Material-Shape Separation has already demonstrated that higher-kinded subtyping is possible for generics [Greenman et al. 2014]. Consequently, it might be possible for our design. Consider the following speculative syntax illustrating the feature's potential:

```
shape<E> Collection
  ext<E sat Equatable> Equatable
  ext<E sat Hashable> Hashable
  ext<E sat Cloneable> Cloneable {
    static This<E> empty();
    <E> 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.

Of course, there is another higher-kinded shape that has been enormously successful in the functional-languages community:

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

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

6. Conclusion

Conditional inheritance seems like an easy feature to add, especially given the decidability measures granted by Material-Shape Separation. Unfortunately, it does not interact well with variance. With use-site variance, it seems to work until one writes programs actually using use-site variance as heavily as it used in practice (despite programmers' hatred of it). With declaration-site variance, it seems to work until one considers the natural variances in standard architectures. So, in theory conditional inheritance is a simple feature, but in practice it is quite complex.

Given that the difficulty lies in practice rather than theory, we have focused on key examples rather than on formalisms. To prove soundness, we can translate the type system to a structural type system with records, (irregular) coinduction, and impredicative bounded universal and existential quantification, much like one can use for other object-oriented type systems. To prove decidability, we adapt the well-founded measure provided by Material-Shape Separation and demonstrate that the relevant algorithms always decrease this measure as they recurse. These goals, however, were not the main challenges; they simply made reality hard to handle. The real challenges lay in accommodating the common architectures and their usage patterns. For this reason, we implemented a type checker for design and confirmed that it can type check the architecture in Figure 3, including its implementation and various use cases.

These use cases required a multitude of features:

Shapes, to explicitly integrate self types

Conditionals, to indicate certain functionality can only exist for suitable type arguments

`super` **Types,** to bypass the requirements of covariance while having the language automatically generates specializations upon instance allocation

Shape-Shifters, to locally override the default implementation of methods, or locally provide new functionality

Extension Shapes, to specify default shape implementation for a type when it should not be tied to the instances

The : **Operator,** to make common programming tasks convenient without ambiguity or loss of expressiveness

Each feature is necessary to accommodate some basic expectation of the language, and each restriction is necessary to guarantee soundness and prevent ambiguity.

Due to the complexity, we are uncertain whether this design will be adopted by practice. Our intent is to make it clear that there is no simpler design possible. The examples we have presented are enough to illustrate that every simplification suffers some significant limitation. Nonetheless, we hope that the benefits of our design will outweigh the inherent complexities it warrants, since at present these problems are haphazardly addressed by frequent run-time type checks and manual implementations to various specializations. We ourselves found the completed design convenient and intuitive to use with less need to be concerned about the tradeoffs between generality and performance.

References

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

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. *ACM SIGPLAN Notices*, 33, 1998.

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.

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

K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *ECOOP*, 1998.

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

N. Cameron, S. Drossopoulou, and E. Ernst. A model for java with wildcards. In *ECOOP*. Springer, 2008.

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

R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 2001.

D. Duggan and J. Ophel. Type-checking multi-parameter type classes. *Journal of functional programming*, 12(02):133–158, 2002.

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

E. Ernst. Family polymorphism. In *ECOOP*, 2001.

B. Greenman, F. Muehlboeck, and R. Tate. Getting F-bounded polymorphism into shape. In *PLDI*, 2014. doi: http://dx.doi.org/10.1145/2594291.2594308.

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

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

S. Kaes. Parametric overloading in polymorphic programming languages. In *ESOP*, 1988. URL http://dx.doi.org/10.1007/3-540-19027-9_9.

A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *ACM SigPlan Notices*, volume 36, 2001.

A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

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.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 1994.

B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

M. Odersky. The Scala language specification, version 2.9, 2014.

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

B. C. Pierce. *Types and programming languages*. MIT press, 2002.

C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *ACM Sigplan Notices*, volume 21, 1986.

R. Tate. Mixed-site variance. In *FOOL*, 2013. URL http://www.cs.cornell.edu/~ross/publications/mixedsite/.

R. Tate, A. Leung, and S. Lerner. Taming wildcards in Java's type system. In *PLDI*, 2011. URL http://www.cs.cornell.edu/~ross/publications/tamewild/.

K. K. Thorup. Genericity in Java with virtual types. In *ECOOP*, 1997.

K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP*, 1999.

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

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.

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

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

S. Wehr and P. Thiemann. On the decidability of subtyping with bounded existential types. In *Proceedings of the Seventh Asian Symposium on Programming Languages and Systems*, 2009b.

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