# Getting F-Bounded Polymorphism into Shape


Cornell University

*Ben Greenman, Fabian Muehlboeck, & Ross Tate*


## Problem

### Type checking with generics, variance, and recursive inheritance is challenging.

There are many difficult corner cases and even subtyping is undecidable [1].

#### Example 1: Undecidable Subtyping

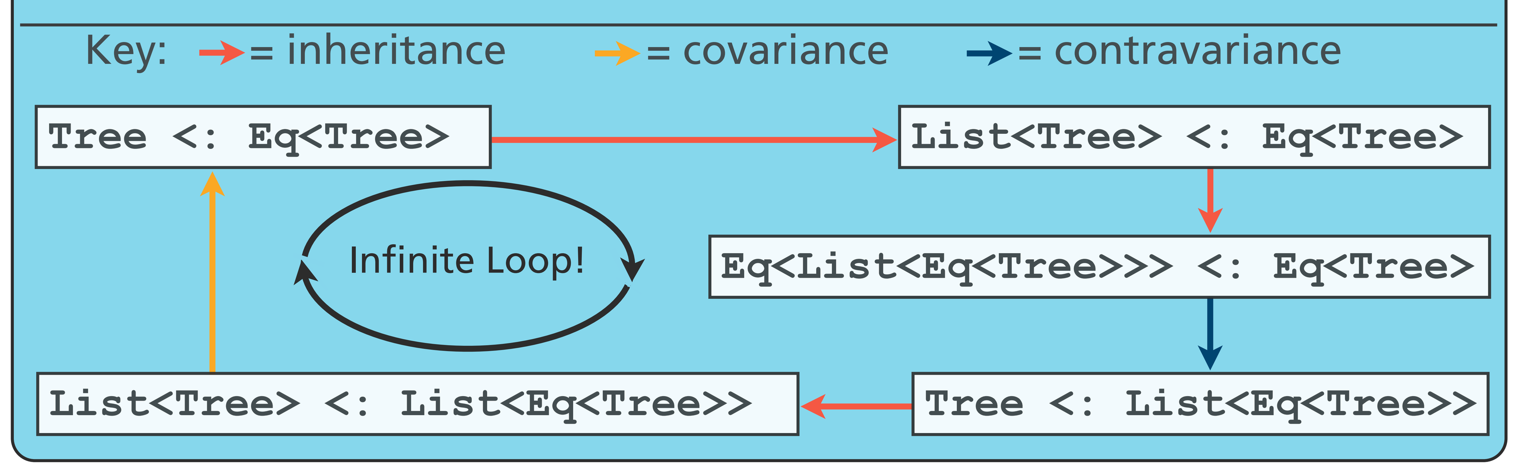
We attempted to provide type-safe equality on lists by using generics to enforce that list elements support type-safe equality.

```
class List<out T> extends Eq<in List<out Eq<in T>>>
```

Next, we thought to define *n*-ary trees with type-safe equality by extending our `List` interface.

```
class Tree extends List<out Tree>
```

But the OpenJDK compiler (version 1.7) crashed when we added variance annotations asked if `Tree` was a subtype of `Eq<Tree>`.




#### Ex 2: Syntactic Identity

In type systems with syntactic identity, intersection commutes

✓ `A & B = B & A`

but not within type arguments.

✗ `Array<A & B> = Array<B & A>`

#### Ex 4: Imprecise Joins

A language without joins would incorrectly reject this program:

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

#### Ex 3: Undecidable Equality

Given the following declaration:

```
class Foo extends
    Array<Foo & Array<Foo>>>
```

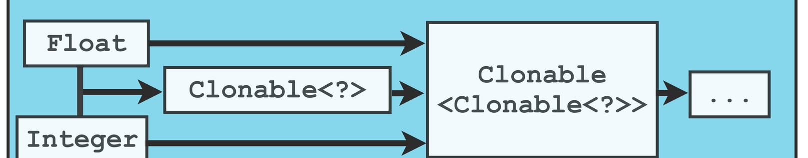
We cannot decide if `Foo` is a subtype of `Array<Foo>`.


#### Ex 5: Imprecise Joins

Suppose we have three classes:

```
class Clonable<out T> { }
class Integer extends Clonable<Integer> { }
class Float extends Clonable<Float> { }
```

The join of `Integer` and `Float` does not exist.




## Observation

### Programmers separate constraints from data. So should the compiler.

**Example:** The interface `Comparable<T>` is very different from most familiar types.

>> `Comparable` is **only used in inheritance or as a constraint.**

>> A programmer never wants a `List<Comparable<X>>`, but rather a `List<T>` where the `T` extends `Comparable<T>`.

**Consequence:** We recognize two disjoint groups of classes & interfaces, formalized as **Material-Shape Separation**.

### Materials

**Summary:** Materials are the data transmitted and shared by program components.

**Used for:**
>> Parameter types
>> Return types
>> Field types
>> Type arguments

**Examples:** `Object, Integer, String, List<T>, Map<K,V>, HashSet<T>, ...`

### Shapes

**Summary:** Shapes define the higher-level structure of a type via recursive inheritance.

**Used for:**
>> Inheritance / Type definitions
>> Type variable constraints

**Examples:** `Clonable<T>, Enum<T>, Equatable<T>, Comparable<T>, Addable<T>, GraphEdge<E,V>`

### Industry Survey

**13.5 million** lines of Java code from **60** open-source projects* show these results.

>> Parameterized shapes were **never used** as materials
>> Exactly **one** project used a material in inheritance, but this definition was never used or exposed by an API.
>> Approximately 30% of projects used raw/wildcarded shapes as materials. Our system can provide this functionality by creating for each shape a parameterless material superclass.
>> In total, we found 15 project-specific shapes, each encoding a self type or a type family.

**Conclusion:** Material-Shape Separation is compatible with modern industry practices.

*All projects were written for Java 1.5 or later. Thanks to the Qualitas Corpus [2] for hosting many of the projects we used.

## Applications

### Material-Shape Separation simplifies type-checking.

The restriction provides a solid foundation for type-system enhancements.

#### Decidable Subtyping

Material-Shape Separation **limits** the power of recursive type definitions **to match practical use.** Cyclic and infinitely expansive inheritance are no longer possible and we have simple, decidable subtyping.

#### Type Equivalence

Our subtyping rules do not rely on syntactic identity, so reliable type equivalence is a free consequence.

Material-Shape Separation also eliminates troublesome corner cases.

✗ `class Foo extends Array<Foo & Array<Foo>>`

is nonsensical because the material `Array` should **never** be used to create a recursive definition.

#### Computable Joins

Joins need only be defined on the acyclic hierarchy of Materials. For example, the least common supertype of `Integer` and `Float` in our system is `Object` because `Clonable<?>` is not a Material.

Separating concepts lets us use a simple join algorithm without sacrificing the power of recursive type constraints.

#### Higher-Kinded Types

The well-founded measure we use to prove decidable subtyping and computable joins generalizes naturally to higher-kinded types.

#### Ceylon Integration

The Ceylon [3] team at Red Hat was our primary industry collaborator. They provided valuable insight and feedback throughout this project.

Material-Shape Separation is compatible with the entire Ceylon codebase and will likely be incorporated into Ceylon 2.0.


[1] Kennedy & Pierce, FOOL/WOOD 2007.
[2] http://qualitascorpus.com/ [3] http://ceylon-lang.org/


Read more: