USER-ORIENTED LANGUAGE DESIGN
Decidability
public class Variable {
    boolean value;
    /** Add this to the list corresponding to value */
    public void addTo( 
        List<? super Variable> trues,
        List<? super Variable> falses) {
            (value ? trues : falses).add(this);
        }
}
Subtyping

class C<P> extends D<\{? super C<L<P>>\}> {}

C<X> <: D<\? super C<X>>

D<D<\? super C<L<X>>\>> <: D<\? super C<X>>

D<D<\? super C<L<X>>\>> <: D<\? super C<L<X>>\>>

C<L<X>> <: D<\? super C<L<X>>\>>
interface Eq<in T> {
    bool equalTo(T other); }

interface List<out T> : Eq<List<Eq<T>>> { }

interface Tree : List<Tree> { }
Restrictions

**Inheritance Restriction**

No use of `? super` in the inheritance hierarchy

```
class C<P> extends D<D<? super C<L<P>>>>
```

**Parameter Restriction**

When constraining type parameters, `? super` may only be used at covariant locations

```
<P extends List<List<? super C<L<P>>>>
```
Survey

No Violations of Our Restrictions

9.2 Million Lines of Code Analyzed

Wildcards in Inheritance

Wildcards in Constraints

No Type Arguments 72.0%

No Wildcards 20.9%

Only Unconstrained Wildcards 3.7%

Uses ? extends 1.5%

Uses ? super 1.8%

All at covariant locations

No Type Arguments

No Wildcards

Only Unconstrained Wildcards

Uses ? extends

Uses ? super

# of Superclass Declarations
Industry Collaborations

Gavin King at redhat

Andrey Breslav at JetBRAINS

on Ceylon

on Kotlin by JetBrains
Materials and Shapes

- **Material**
  - List, Integer, Property, Comparator

- **Shape**
  - Comparable, Summable, Cloneable

No class/interface is both a material and a shape

13.5 Million Lines of Code Analyzed
Programmers are Humans
Want to provide a “separate” function

- Inputs: middle, elems, smaller, bigger
- Requirements:
  - Place all values in elems less than middle into smaller
  - Place all other values in elems into bigger

Goals

- Implement separate
- Provide maximally flexible type signature
Goal
- Place nonnegative values in “ints” into “positives”

Context
- “ignore” throws away all elements added to it

Implementation
- separate(0, ints, ignore, positives);
User Types

- **ints**: `Iterable<Integer>`
  - You can get things from iterables

- **positives**: `Collection<Integer>`
  - You can add things to collections

- **ignore**: `Collection<Object>`
  - You can add anything to it

- **Integer** implements `Comparable<Number>`
  - Integers can be compared with any number
void separate(middle, elems, smaller, bigger) {
    foreach (elem in elems) {
        (elem < middle ? smaller : bigger).
            add(elem);
    }
}
<T extends Comparable<T>>
void separate(T middle, 
Iterable<T> elems, 
Collection<T> smaller, Collection<T> bigger) {
    foreach (elem in elems) {
        if (elem < middle)
            smaller.add(elem);
        else
            bigger.add(elem);
    }
}
### Insufficient Flexibility

<table>
<thead>
<tr>
<th>Formals</th>
<th>Actuals</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;T extends Comparable&lt;T&gt;&gt;</code> void separate(T middle, Iterable&lt;T&gt; elems, Collection&lt;T&gt; smaller, Collection&lt;T&gt; bigger)</td>
<td>Integer 0 ints ignore positives</td>
</tr>
</tbody>
</table>
Wildcards

- `<T extends Comparable<? super T>>`
  
  ```java
  void separate(T middle,
                Iterable<? extends T> elems,
                Collection<? super T> smaller,
                Collection<? super T> bigger) {
    foreach (elem in elems)
      (elem < middle ? smaller : bigger)
        .add(elem);
  }
  ```
Excessive Annotations

- `<T>`

```java
definition
void flatten(
    Iterable<? extends Iterable<? extends T>> colls,
    Collection<? super T> into) {
    for (Iterable<? extends T> coll : colls) {
        for (T elem : coll) {
            into.add(elem);
        }
    }
```
<T extends Comparable<T>>
void separate(T middle, <T> elems, Collection<T> smaller, Collection<T> bigger) {
    foreach (elem in elems) {
        if (elem < middle) smaller.add(elem);
        else bigger.add(elem);
    }
}
**Insufficient Flexibility**

<table>
<thead>
<tr>
<th>Formals</th>
<th>Actuals</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;T extends Comparable&lt;T&gt;&gt;</code> void separate(T middle, Iterable&lt;T&gt; elems, Collection&lt;T&gt; smaller, Collection&lt;T&gt; bigger)`</td>
<td>Integer 0 ints ignore positives</td>
</tr>
</tbody>
</table>
<T extends Comparable<T>,
    U super T, V super T>
void separate(T middle,
    Iterable<T> elems,
    Collection<U> smaller,
    Collection<V> bigger) {
    foreach (elem in elems)
        (elem < middle ? smaller : bigger)
            .add(elem);
}
Mixed-Site Variance

- `<T extends Comparable<T>>`

`void separate(T middle, Iterable<T> elems, Collection<in T> smaller, Collection<in T> bigger) {`

`foreach (elem in elems)`

`(elem < middle ? smaller : bigger)
 .add(elem);`

```
Use+Declaration-Site

- The two address orthogonal roles
  - declaration-site for class/interface designers
  - use-site for class/interface users
- How do the two interact?
  - given interface Iterator<out T> {...}
  - what does Iterator<in Number> mean?
## Expectations vs. Designs

<table>
<thead>
<tr>
<th>Has declaration-site variance</th>
<th>Java</th>
<th>Scala</th>
<th>Default</th>
<th>Layer</th>
<th>Join</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C&lt;\tau&gt; &lt;: C&lt;\text{out } \tau&gt;$</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>$C&lt;\text{out } \tau&gt; &lt;: C&lt;\text{out } \tau'&gt;$ implies $\tau &lt;: \tau'$</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>instance of $C&lt;\text{out } \tau&gt;$ is instance of $C&lt;\tau'&gt;$ for some $\tau' &lt;: \tau$</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>instance of $C&lt;\text{out } \tau&gt;$ allocated as $C&lt;\tau'&gt;$ for some $\tau' &lt;: \tau$</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>$C&lt;\text{in } \tau \text{ out } \tau'&gt;$ is expressible</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>$C&lt;\text{in } \tau \text{ out } \tau&gt;$ is valid implies $C&lt;\tau&gt;$ is valid</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>$\text{Out}&lt;\text{in } \tau&gt;$ is valid</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>implicit constraints used in subtyping</td>
<td>✖️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Principal Types
Principal Type

- The principal type of an expression
  - a type for that expression that is better than all other types for that expression
- “Hello” has principal type String
  - “Hello” also has type Object, CharSequence, ...
  - String is a subtype of all those types
- A language has principal types
  - if every possible expression has a principal type
Java

- `assertEquals(5, Integer.valueOf(5))`  
  - ambiguous!
  - Is it two ints or two Integers?
- But the expression 5 is also an Integer
- And `Integer.valueOf(5)` is also an int
- Neither expression has a principal type
Ambiguous Semantics

\[
\langle P \rangle \text{ List}\langle P \rangle \text{ singleton}(P \text{ elem}) \{ \text{return } \text{list}; \}
\]
\[
\langle Q \text{ extends } \text{Comparable}\langle Q \rangle \rangle \{ \text{return list}; \}
\]

```
String typeName(Comparable\langle \_ \rangle c) {return "Comparable";}
String typeName(String s) {return "String";}
String typeName(Integer i) {return "Integer";}
String typeName(Calendar c) {return "Calendar";}
String ambiguous(boolean b) {
    return typeName(foo(singleton(b ? "Blah" : 1)));
}
```
Use-Site Inferability Check

- `<T> List<T> singletonList(T) {...}
- var objs = singletonList("Hello");
- objs.add(5);
  - fails to type check
  - objs is inferred to be an List<String>
  - needs to be an List<Object>
Declaration-Site Inferability

- `<T> List<T> singletonList(T)
  - T is not inferable because Array is invariant
  - singletonList(“Hello”)
    - could have type `List<String>` or `List<Object>`
    - no principal type

- `<T> Iterable<? extends T> singletonIterable(T)
  - T is inferable because `? extends` is covariant
  - singletonIterable(“Hello”)
    - has type `Iterable<? extends String>`
    - which is subtype of `Iterable<? extends Object>"
Gradual Types
Goal

- Mix static and dynamic type systems
  - e.g. Java with JavaScript

- Requirements
  - no implicit insertions of wrappers
  - dynamic code is just static code minus types
  - stripping types preserves or improves semantics
  - static code can assume type annotations are true
C#'s dynamic Type

- bool Equal(object left, object right) {
  return left == right;
}

- Equal(0, 0) returns false
C#'s dynamic Type

- interface Getter<T> { T get(); }
- class Five : Getter<int>, Getter<string> {
  int Getter<int>.get() {
    return 5;
  }
  double Getter<string>.get() {
    return 5.0;
  }
}
- void Print(Getter<int> getter) {
  Console.WriteLine(getter.get());
}
C#'s dynamic Type

- `List<T> Snoc<T>(IEnumerable<T> start, T end)`
  - `var elems = ToList(start);`
  - `elems.add(end);`
  - `return elems;`

- `Snoc(Singleton("Hello"), 5)` works

- `Snoc(Singleton("Hello"), 5)` works

- `Snoc(Singleton("Hello"), 5)` works

Crashes if made dynamic!
Prerequisite Language Properties

- Static Behavioral Subtyping
  - Using a more precise type for a subexpression improves the typability of the whole expression

- Decidability
  - Typing must be reliably doable at run time

- Principality
  - Every execution has a most precise typing