Wildcards

```java
public 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

```java
class C(P) extends D(D(? super C(L(P)))) {}
```

Efficiency

```java
interface Eq<T> {
    bool equals(T other);
}
interface List<T> : Eq<List<Eq<T>>> {
}
interface Tree : List<Tree> {}
```

Restrictions

- Inheritance Restriction
  - No use of `? super` in the inheritance hierarchy
- Parameter Restriction
  - When constraining type parameters, `? super` may only be used at covariant locations
Survey

9.2 Million Lines of Code Analyzed

Wildcards in Inheritance

<table>
<thead>
<tr>
<th>No Type Arguments</th>
<th>No Wildcards</th>
<th>Only Unconstrained Wildcards</th>
<th>Uses ? extends</th>
<th>Uses ? super</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wildcards in Constraints

<table>
<thead>
<tr>
<th>No Wildcards</th>
<th>Only Unconstrained Wildcards</th>
<th>Uses ? extends</th>
<th>Uses ? super</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.9%</td>
<td>3.7%</td>
<td>1.5%</td>
<td>1.8%</td>
</tr>
</tbody>
</table>

Industry Collaborations

Gavin King

Andrey Breslav

Materials and Shapes

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

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

13.5 Million Lines of Code Analyzed

Programmers are Humans

Library Designer

- 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

Library User

- 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

Library Implementation

- **void separate(middle, elements, smaller, bigger)**
  - foreach (element in elements)
  - (element < middle) ? smaller : bigger
  - .add(element);

Library Type

- `<T extends Comparable<T>> void separate(T middle, elements, Collection<T> smaller, Collection<T> bigger)`

Insufficient Flexibility

- **Formals**
  - `<T extends Comparable<T>> void separate(T middle, elements, Collection<T> smaller, Collection<T> bigger)`
- **Actuals**
  - Integer: x
  - 0: x
  - ints: x
  - ignore: x
  - positives: x

Wildcards

- `<T extends Comparable<? super T>> void separate(T middle, elements, Collection<? super T> smaller, Collection<? super T> bigger)`

Excessive Annotations

- `<T> void flatten(elements, into)`
  - for (element in elements)
  - (element < middle) ? smaller : bigger
  - .add(element);
- `<T extends Iterable<? extends T>> Collection<? super T> into)`
  - for (element : into) for (T elem : call) into.add(elem);
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>Expectations</th>
<th>Java</th>
<th>Scala</th>
<th>Default</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&lt;in T, out T&gt; is expressible</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C&lt;in T, out T&gt; is valid implies C&lt;T&gt; is valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C&lt;in T, out T&gt; is expressible</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C&lt;in T, out T&gt; is valid implies C&lt;T&gt; is valid</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

- `List(P) singleton(P elem) {return null;}`
- `var objs = singletonList("Hello");`
  - fails to type check
  - objs is inferred to be an `List(String)`
  - needs to be an `List(Object)`

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"), S) works

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());
}

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