Self Type Constructors

Atsushi Igarashi
Kyoto University
Joint work with Chieri Saito

My Research Interests

Type systems

- for static ananysis
  - Linear types, resource usage analysis, etc.
- for object-oriented languages
  - Generics, wildcards, union types, self types, gradual typing, etc.
  - Using Featherweight Java
- for multi-stage programming
  - Curry-Howard isomorphisms for modal logic

Typical Type Systems for Class-Based Object-Oriented PLs

- Class names as types
- Inheritance as subtyping

➢ Resulting in difficulty in reusing classes with *recursive interfaces* by inheritance
  - Standard (non)solution: downcasts
  - Self types (often called MyType [Bruce et al.])
  - OCaml

Today’s Talk

- Review of MyType
- Challenge in programming generic collection classes
- Self Type Constructors: Extending MyType to the type constructor level
  - …with unpleasant complication(!)

MyType in LOOJ [Bruce et al. 04]

- Keyword “This” represents the class in which it appears
  - Its meaning changes when it is inherited

```
class C {
   int f;
   boolean isEqual(This that){ // binary method
      return this.f == that.f;
}   }
class D extends C {
   int g;
   boolean isEqual(This that){
      return super.isEqual(that) && this.g == that.g; // well-typed
}   }
```

Exact Types to Avoid Unsoundness

- Covariant change of argument types is unsound under inheritance-based subtyping

```
D d = …;   C c1 = d;   C c2 = …;
c1.isEqual(c2);
```

- LOOJ has “exact types” @C
  - @C stands for only C objects (not a subclass of C)
  - isEqual() can be invoked only if the receiver type is exact

Typing rule for MyType

- A method body is typed under the assumption that This is a subtype of the current class

$$\text{This} <: \text{C}, \text{that}:\text{This}, \text{this}:\text{This} \vdash \text{this.f} == \text{that.f} : \text{bool}$$

- So that the method can be used any subclass of C

“This” is indeed a Polymorphic Type Variable!

```
class C<This extends C<This>> { // F-bounded polymorphism
   int f;
   boolean isEqual(This that){ // binary method
      return this.f == that.f;
}  }
class D<This extends D<This>> extends C<This> {
   int g;
   boolean isEqual(This that){
      return super.isEqual(that) && this.g == that.g;
}  }
class FixC extends C<FixC> {}   // Corresponding to @C
class FixD extends D<FixD> {}  // No subtyping btw. @C and @D
```

Digression: clone() with MyType

- Doesn’t quite work
  - This is an (unknown) subtype of C, not vice versa

```
class C {
  This clone() { return new C(); }
}
```

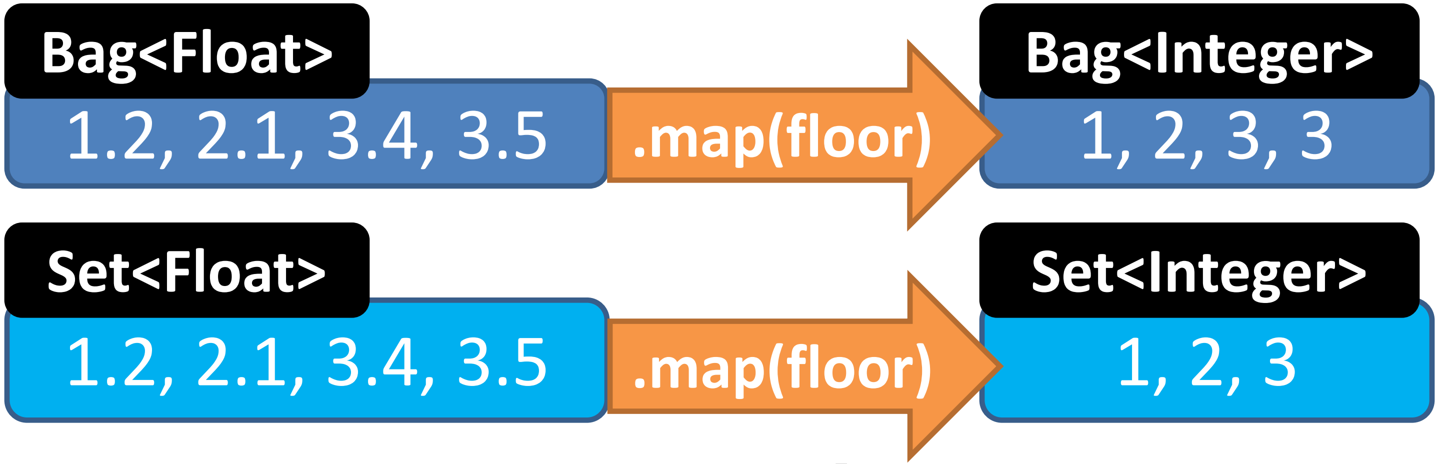
- One solution is nonheritable methods [I. & Saito’09], in which
  - This is *equal* to the current class, but
  - Every subclass has to override them

Today’s Talk

- Review of MyType
- Challenge in programming generic collection classes
- Self Type Constructors: Extending MyType to the type constructor level
  - …with unpleasant complication(!)

Today’s challenge:
map() in generic collection classes

- **Bag** implements **map()**
  - **map()** returns the same kind of collection as the receiver
- **Set** is a subclass of **Bag**
  - **Set** reuses **Bag**'s implementation as much as possible
- **Set** prohibits duplicate elements



Bag<Float>
1.2, 2.1, 3.4, 3.5
.map(floor)
Bag<Integer>
1, 2, 3, 3

Set<Float>
1.2, 2.1, 3.4, 3.5
.map(floor)
Set<Integer>
1, 2, 3

floor: **Float→Integer**

Skeletons of **Ba**

```
interface Comparable {
  int compare(This that);
}
```

```
class Bag<T> {

   void add(T t) { ... }

   <U> Bag<U> create(){
      return
   }

   <U> ? map(T→U f) {
      ? tmp=create();
      for(T t: this) tmp.add(f(t));
      return tmp;
   }
}
```

What is the return type of **map()**?

```
class Set<T extends Comparable>
         extends Bag<T> {

   // overriding to prevent
   // duplicate elements
   void add(T t) { ... }

   <U> Set<U> create(){
      return new Set<U>();
   }

   // no redefinition of map()
}
```

T's bound is refined

Covariant Refinement of Return Types is *not* a Solution

- **Set** must override **map()**
- Downcasts would fail at run time if **create()** were not overridden in **Set**

```
class Bag<T> {
   <U> Bag<U> map(T→U f) { ... }
}

class Set<T> extends Bag<T> {
   <U> Set<U> map(T→U f) {
      return (Set<U>) super.map(f);
}  }
```

MyType and Generics in LOOJ

- The meaning of *MyType* in a generic class includes *the formal type parameters*
  - e.g. **This** in class **Bag<T>** means **Bag<T>**
- So, *MyType* cannot be used for the return type of **map()**

Today's Talk

- Review of MyType
- Challenge in programming generic collection classes
- Self Type Constructors: Extending MyType to the type constructor level
  - …with unpleasant complication(!)

Self Type Constructors: *MyType* as a Type Constructor

- **This** means a class name, without type parameters

The meaning of **This**

```
class Bag<T> {

   <U> This<U> create() { … }  // should be nonheritable

   <U> This<U> map(T→U f) {
      This<U> tmp=create();
      for(T t: this) tmp.add(f(t));
      return tmp;
   }
}
```

This takes one argument

General use case of Self Type Constructors

- Writing the interface of a generic class that refers to itself recursively but with *different* type instantiations
  - e.g. collection with **flatMap()**

Set<String>

"this", "is", "high"

.flatMap(str2char)

Set<Character>

't', 'h', 'i', 's', 'g'

```
class Bag<T> {

<U> This<U> flatMap(T→This<U> f) {
      This<U> tmp=create();
      for(T t: this) tmp.append(f(t));
      return tmp;
}  }
```

str2char:
String→Set<Character>

Today’s Talk

- Review of MyType
- Challenge in programming generic collection classes
- Self Type Constructors: Extending MyType to the type constructor level
  - …with unpleasant complication(!)

Refining bounds can yield ill-formed types in subclasses

- **map()** inherited to **Set** is not safe (ill-kinded)

```
class Bag<T> {
   <U> This<U> map(T→U f) { … }
}
class Set<T extends Comparable> extends Bag<T> {
   // <U> This<U> map(T→U f) { … }
   // This<U> is ill-formed here
}
```

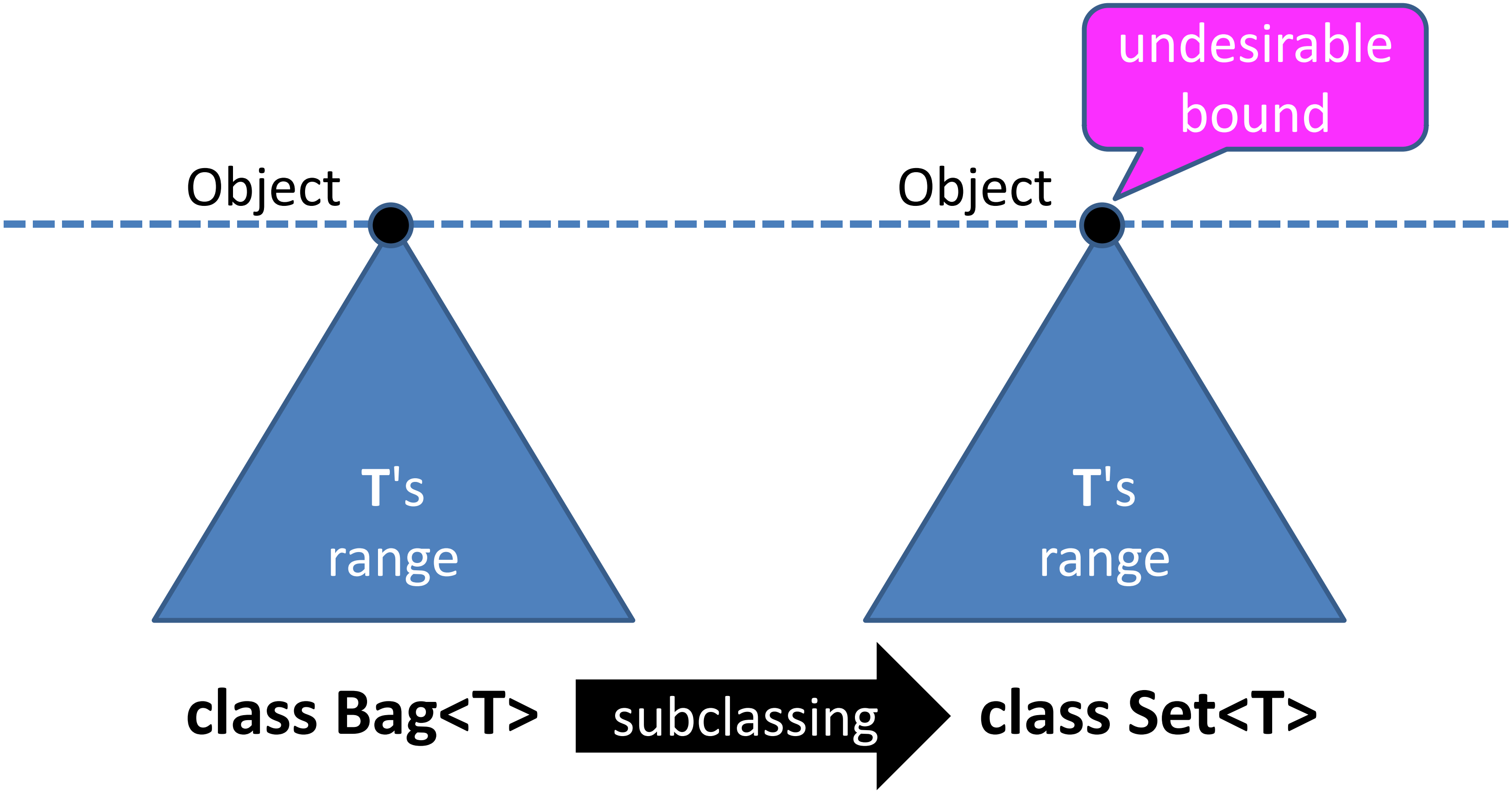
inherited

- So, we should prohibit refinement of bounds
- How can we declare **Set**, then?

How the body of map() is typed

- Bag: $* \to *$, T: $*$, This <: Bag, U: $*$,
  f: $T \to U$, this: This<T> $\vdash$ *body* : This<U>
- If Set is a subtype of Bag, then *body* will remain well typed (and can be inherited)
- But, actually, it's not!
  - Set: $\forall (X <: \text{Comparable}) \to *$
    - Subtype-constrained dependent kind

If a type parameter is *not* included in the meaning of **This**, its bound must be *fixed*



undesirable bound

Object

Object

T's range

T's range

class Bag<T>

subclassing

class Set<T>

It *is* OK to refine bounds *in LOOJ*

- since the meaning of **This** includes type parameters
  - in other words, **This** does not take any arguments

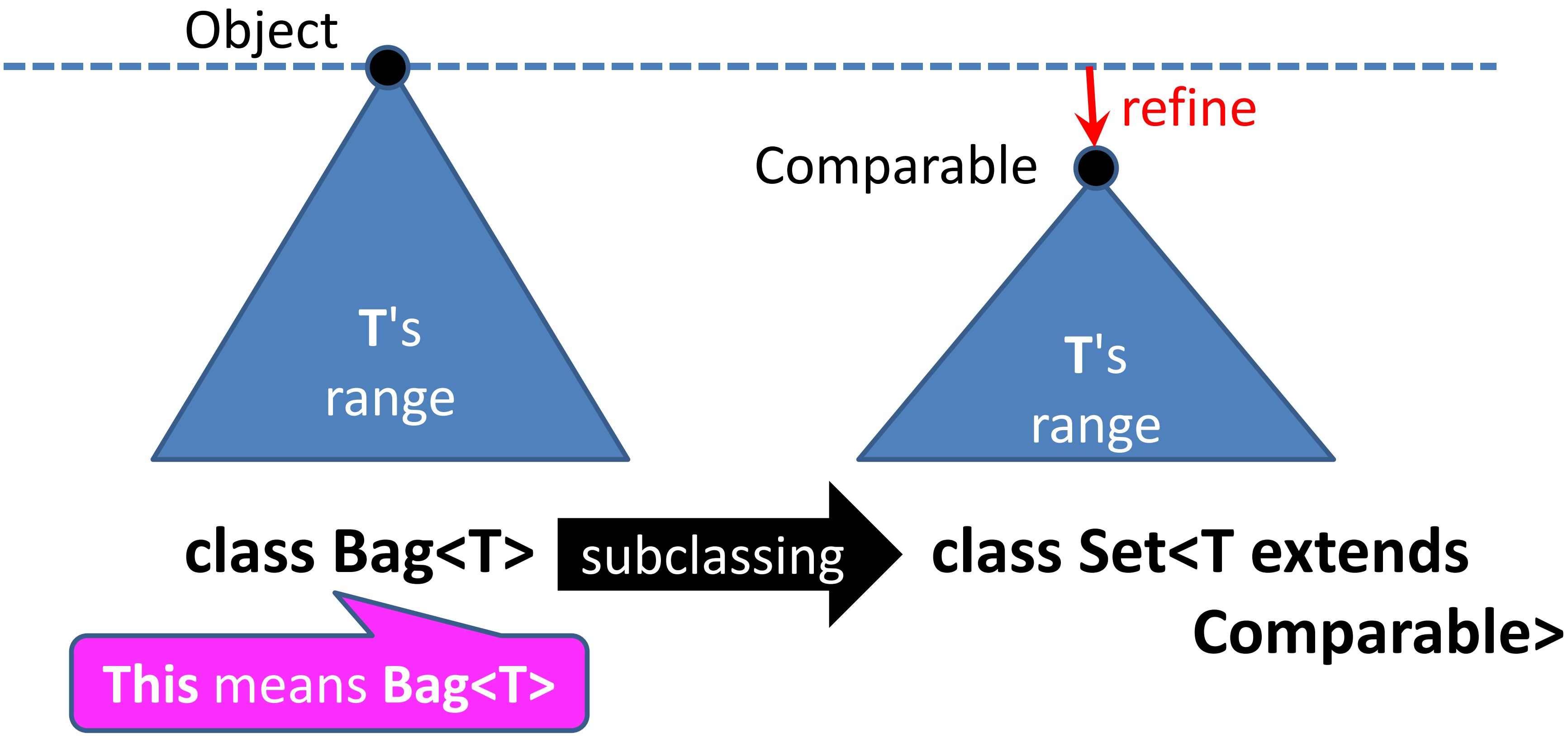
```
class Bag<T> {
   This map(T→T f) { ... } // monomorphic map()
}

class Set<T extends Comparable> extends Bag<T> {
   // This map(T→T f) { ... }
   // This is well formed
}
```

inherited

How the body of map() is typed

- Bag: *→*, T: *, This <: Bag<T>, f: T→T, this: This ⊢ *body* : This
- Set is not a subtype of Bag, but …
- Set<T> *is* a subtype of Bag<T> for any type T!
  - It's declared to be so
- So, *body* remains well-typed when the upper bound of This is replaced with Set<T>

If a type parameter *is* included in the meaning of **This**, its bound can be *refined*



Object

refine

Comparable

T's range

T's range

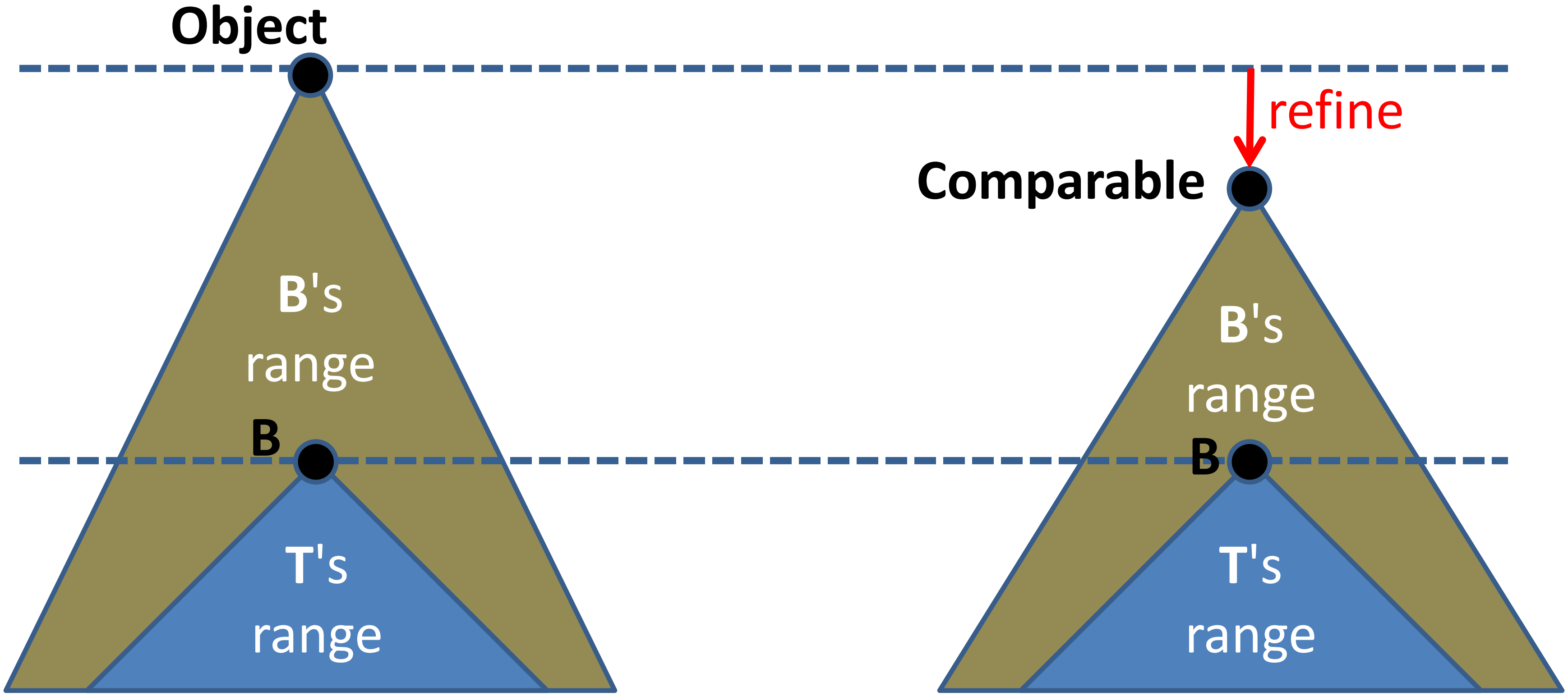
class Bag<T>

subclassing

class Set<T extends Comparable>

This means Bag<T>

Introducing two kinds of type variables may solve the problem!



class
Bag<B,T extends B>

The meaning of **This**

subclassing

class
Set<B extends
Comparable,
T extends B>

Indeed, it solves the problem!

- Bag: $\forall(B{:}*) \rightarrow \forall(T <: B) \rightarrow *$
- Set: $\forall(B <: \text{Comparable}) \rightarrow \forall(T <: B) \rightarrow *$
- $B{:}*,\ T <: B,\ \text{This} <: \text{Bag}\langle B\rangle,\ U <: B,$
  $f: T \rightarrow U,\ \text{this}: \text{This}\langle T\rangle \vdash \textit{body} : \text{This}\langle U\rangle$
- Again, Set is not a subtype of Bag, but…
- Set<B> is a subtype of Bag<B> for any B, which is a subtype of Comparable
- Replacing the bounds for B and This with subtypes (i.e., Comparable and Set<B>) leads to what we want

Correct **Bag** and **Set** classes

The meaning of **This**

```
class Bag<B; T extends B> {
   <U extends B> This<U> map(T→U f) { ... }
}
```

This takes one argument

inherited

```
class Set<B extends Comparable; T extends B>
             extends Bag<B,T> {
   //<U extends B> This<U> map(T→U f) { ... }
   // This<U> is well formed
}
```

Signature resolution in client code

- **This** in the return type is replaced with the class name and refinable-bound params of the receiver

```
Bag<Number,Float> floatbag=... ;
Set<Number,Float> floatset=... ;

Bag<Number,Integer> integerbag=floatbag.map(floor);
```

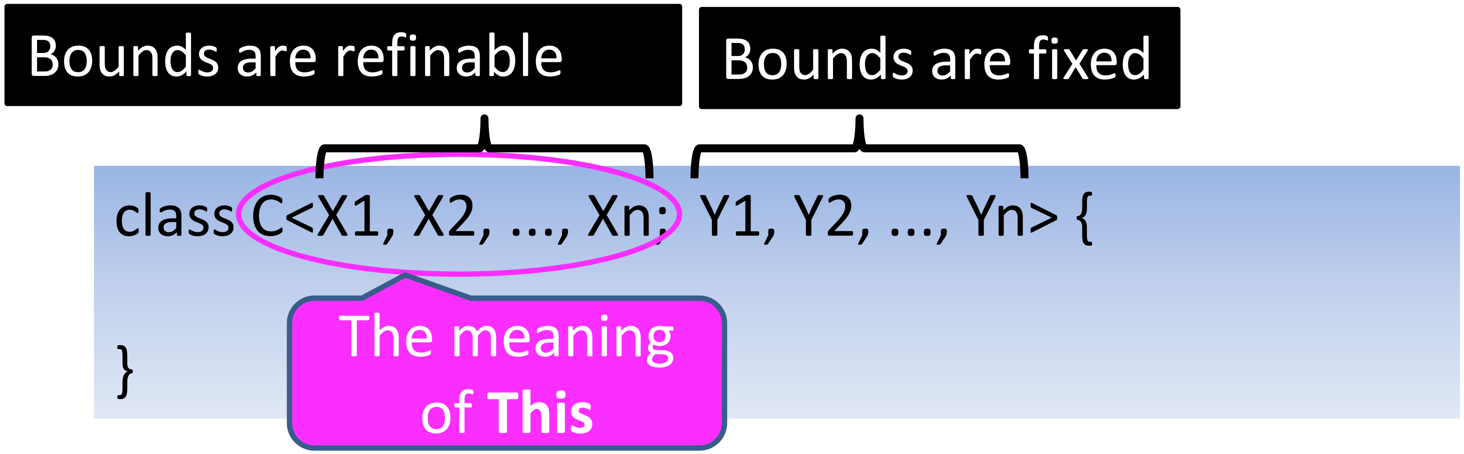
= This<U>{U:=Integer}{This:=Bag<Number>}

```
Set<Number,Integer> integerset=floatset.map(floor);
```

= This<U>{U:=Integer}{This:=Set<Number>}

Summary of Self Type Constructors

- **This** in a generic class is a type constructor, which
  - takes arguments as many as the number of parameters before a semicolon
  - means a class name with parameters before the semicolon



Bounds are refinable

Bounds are fixed

```
class C<X1, X2, ..., Xn; Y1, Y2, ..., Yn> {

}
```

The meaning of **This**

FGJ_{stc}: A Formal Core Calculus of Self Type Constructors

- Extension of Featherweight GJ [I., Pierce, Wadler'99] w/
  - self type constructors
  - exact types
  - constructor-polymorphic methods
  - **exact** statements
  - and the usual features of FJ family
- Kinding is a bit complicated
- FGJ_{stc} enjoys type soundness
  - subject reduction theorem
  - progress theorem

Encoding self type constructors with higher-order type constructors

- Higher-order type constructors
  - Classes can be parameterized by type constructors
- Type declarations become (even) more complicated
  - FGJω [Altherr and Cremet. J. Object Technology 08]
  - Scala [Moors, Piessens and Odersky. OOPSLA08]

Encoding in FGJω

- by combination of
  - Higher-order type constructors
  - F-bounded polymorphism

```
class Bag<Bound: *→*, T extends Bound<T>,
          This extends λ(X extends Bound<X>).Bag<Bound,X,This>> {
}
```

FGJω

- requires fixed point classes

```
class FixBag<Bound<_>, T extends Bound<T>>
              extends Bag<Bound,T,FixBag> { }
```

```
class Bag<Bound;T extends Bound> {
}
```

Our Solution

Encoding in Scala

- by combination of
  - – Higher-order type constructors
  - – Abstract type members [Odersky et al. 03]
  - – F-bounded polymorphism [Canning et al. 89]
    - A type variable appears in its upper bound

```
class Bag<Bound<_>, T extends Bound<T>> {
  type Self<X extends Bound<X>> extends Bag<Bound,X>
}
```

Scala in Java-like syntax

```
class Bag<Bound;T extends Bound> {
}
```

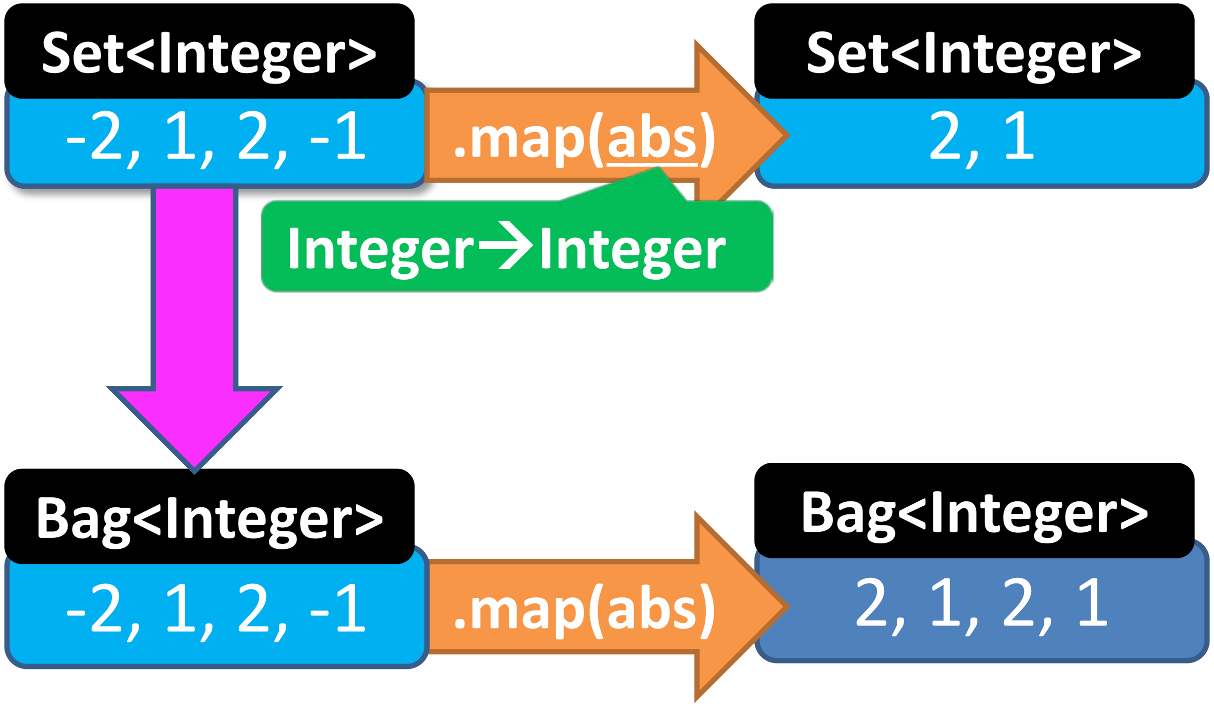
Our solution

Scala 2.8.0 β1 (as of Feb., 2010)

```
class
  <U                                              t){ … }
}
class
  Set
}
```

syntax

Static types affect the result



- **ma**
  -
  - type
- Compiler will supply the factory

Conclusion

- Self Type Constructors
  - for the interface of a generic class that refers to itself recursively but different type instantiations
  - Useful for **map()**, **flatMap()**, and so on
- Idea looks simple but more complicated than expected