Self Type Constructors

Atsushi Igarashi
Kyoto University
Joint work with Chieri Saito
My Research Interests

Type systems

• for static analysis
  – 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

```java
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

  This <: C, that: This, this: This ⊨ this.f == that.f : 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

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

• One solution is noninheritable methods [I. & Saito’09], in which
  – This is \textit{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: \texttt{map()} in generic collection classes

- **Bag** implements \texttt{map()}
  - \texttt{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

\begin{center}
\begin{tikzpicture}

\node[black] (bag) at (0, 0) {\texttt{Bag<Float>}};
\node[black] (set) at (0, -2) {\texttt{Set<Float>}};
\node[black] (bag_int) at (3, 0) {\texttt{Bag<Integer>}};
\node[black] (set_int) at (3, -2) {\texttt{Set<Integer>}};

\node (float1) at (1, 0) {1.2, 2.1, 3.4, 3.5};
\node (float2) at (1, -2) {1.2, 2.1, 3.4, 3.5};
\node (int1) at (4, 0) {1, 2, 3, 3};
\node (int2) at (4, -2) {1, 2, 3};

\draw[orange, -stealth] (float1) -- (float2);
\draw[orange, -stealth] (float2) -- (int1);
\draw[orange, -stealth] (float2) -- (int2);

\node[black] at (1.5, -1) {\texttt{.map(floor)}};
\node[black] at (2.5, -1) {floor: \texttt{Float} \rightarrow \texttt{Integer}};
\end{tikzpicture}
\end{center}
Skeletons of Bag and Set classes

class Bag\textless{}T\textgreater{} {  
    void add(T t) { ... }  

    \textless{}U\textgreater{} Bag\textless{}U\textgreater{} create(){  
        return new Bag\textless{}U\textgreater{}();  
    }  

    \textless{}U\textgreater{} ? map(T\rightarrow{}U f) {  
        ? tmp=create();  
        for(T t: this) tmp.add(f(t));  
        return tmp;  
    }  
}  

class Set\textless{}T extends Comparable extends Bag\textless{}T\textgreater{} {  
    // overriding to prevent duplicate elements  
    void add(T t) { ... }  

    \textless{}U\textgreater{} Set\textless{}U\textgreater{} create(){  
        return new Set\textless{}U\textgreater{}();  
    }  

    // no redefinition of map()  
}  

interface Comparable {  
    int compare(This that);  
}
Covariant Refinement of Return Types is \textit{not} a Solution

- \textbf{Set} must override \texttt{map()}
- Downcasts would fail at run time if \texttt{create()} were not overridden in \texttt{Set}

```java
class Bag<T> {
    <U> Bag<U> map(T\rightarrow U f) { ... }
}

class Set<T> extends Bag<T> {
    <U> Set<U> map(T\rightarrow 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

```java
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;
    }
}
```
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()`

```java
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;
  }
}
```

Example:
- `Set<String>` `"this", "is", "high"`.flatMap(str2char)
  - `str2char: String→Set<Character>
    - Set<Character> 't', 'h', 'i', 's', 'g'`
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-kindled)

```java
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
}
```

- So, we should prohibit refinement of bounds
- How can we declare **Set**, then?
How the body of map() is typed

• Bag: \(* \rightarrow \ast\), T: \ast\, \text{This} <: \text{Bag}, U: \ast\, \text{f: } T \rightarrow U\), this: \text{This}<T> \mid body : \text{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: \(\forall (X <: \text{Comparable}) \rightarrow \ast\)
    • Subtype-constrained dependent kind
If a type parameter is *not* included in the meaning of **This**, its bound must be *fixed*.

- **Object**
  - T's range
  - `class Bag<T>`

- **Object**
  - T's range
  - `class Set<T>`

The subclassing from `Bag<T>` to `Set<T>` is undesirable due to the fixed bound.
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

```java
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
}
```
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.
Introducing two kinds of type variables may solve the problem!

The meaning of This

25
Indeed, it solves the problem!

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

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

```java
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

```java
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

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

The meaning of **This**
FGJ\textsubscript{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
  - \texttt{exact} statements
  - and the usual features of FJ family

- Kinding is a bit complicated

- FGJ\textsubscript{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

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

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

• requires fixed point classes

```java
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

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

Scala in Java-like syntax

```scala
class Bag[Bound;T extends Bound> {
}
```

Our solution
Scala 2.8.0 β1 (as of Feb., 2010)

- `map()` takes
  - the result type as another type parameter
  - A factory object which returns an object of the result type
- Compiler will supply the factory

```scala
class Bag[T]<U, That>(implicit Factory[U,That] f){ ... }

class Set[T] extends Bag[T] { Set(implicit T => Comparable[T] c){ ... } // constructor

Bag<Integer>.map(abs)
-2, 1, 2, -1
2, 1, 2, 1

Set<Integer>
-2, 1, 2, -1
2, 1

Static types affect the result
```
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