Self Type Constructors

Chieri Saito  
Graduate School of Informatics  
Kyoto University, Japan  
saito@kuis.kyoto-u.ac.jp

Atsushi Igarashi  
Graduate School of Informatics  
Kyoto University, Japan  
igarashi@kuis.kyoto-u.ac.jp

Abstract
Bruce and Foster proposed the language LOOJ, an extension of Java with the notion of MyType, which represents the type of a self reference and changes its meaning along with inheritance. MyType is useful to write extensible yet type-safe classes for objects with recursive interfaces, that is, ones with methods that take or return objects of the same type as the receiver.

Although LOOJ has also generics, MyType has been introduced as a feature rather orthogonal to generics. As a result, LOOJ cannot express an interface that refers to the same generic class recursively but with different type arguments. This is a significant limitation because such an interface naturally arises in practice, for example, in a generic collection class with method map(), which converts a collection to the same kind of collection of different elements. Altherr and Cremet and Moors, Piessens, and Odersky gave solutions to this problem but they used a highly sophisticated combination of advanced mechanisms such as abstract type members, higher-order type constructors, and F-bounded polymorphism.

In this paper, we give another solution by introducing self type constructors, which integrate MyType and generics so that MyType can take type arguments in a generic class. Our solution is simpler—it uses only first-order type constructors and neither abstract type members nor F-bounds. We demonstrate the expressive power of self type constructors by means of examples, formalize a core language with self type constructors, and prove its type safety.

Categories and Subject Descriptors  
D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.2 [Programming Languages]: Language Classifications—Object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features—Classes and objects; Polymorphism; F.3.3 [Logics and Meaning of Programs]: Studies of Program Constructs—Object-oriented constructs; Type structure

General Terms  
Design, Languages, Theory

Keywords  
binary methods, generics, MyType, type constructor polymorphism

1. Introduction

Background. It is well known that simple type systems (such as that of Java without generics) are not sufficiently expressive to make use of the inheritance mechanism in a type safe manner. One of the classical problems in this context is how to express binary methods [6]—methods that are supposed to take an object of the same type as the receiver, such as equals()—in statically typed languages. Ideally, in a class definition, the parameter type of a binary method has to change covariantly as the class extends so that subclasses refer to themselves. However, such covariant change of parameter types, if naively allowed, would break type safety and so C++ and Java disallow it. As a result, the parameter type of a binary method is fixed to a particular class name and this problem is often "solved" by typecasting. The use of typecasting, however, is not a real solution since they may fail at run-time, if used carelessly. A similar problem occurs when two or more classes are involved: paradigmatic examples are found in the implementation of node and edge classes for graphs [13] and also in the expression problem [39].

Over the last years, there have been many proposals to solve the problem above; a key idea common to them is to provide such inheritance and typing mechanisms that can properly preserve (mutual or self) "dependencies" among the interfaces of related classes. According to how dependencies are expressed in type systems, these proposals can be classified into two: one with dependent types [13, 14, 27, 28, 11, 31] and one without [20, 34, 19, 7, 8, 9, 5]. This paper focuses on the latter, which is admittedly less expressive but simpler and usually expressive enough.

MyType and its Extensions. The approach, in which dependent types are not used, is based on MyType [4], which represents the type of a self reference. MyType refers to the
class where it appears and changes its meaning covariantly when a member is inherited so as to refer to the class that inherits it. So, MyType can be used to give appropriate signatures to binary methods. Although MyType in earlier proposals can express only self-recursion within a single class, it has been extended to more general settings: mutually recursive classes [34, 8, 9, 5], class hierarchies [20], and arbitrarily nested groups of classes [19].

**LOOJ and Its Limitation.** Five years ago, Bruce and Foster proposed the language LOOJ, an extension of Java with MyType [7]. Although LOOJ also includes generics [23, 3], MyType has been introduced as a feature rather orthogonal to generics. As a result, LOOJ cannot express an interface that refers to the same generic class with different type arguments. For example, consider a collection class and its method map(), which takes a function from the element type to another and returns a new collection whose elements are obtained by applying the function to each element in the receiver collection object. It is natural to expect that this method returns the same kind of collection of a different element type but LOOJ types cannot express such an interface. This is a significant limitation because such an interface arises naturally in practice, namely, in generic collection classes, which are typical applications of generic classes.

In short, the limitation of LOOJ is that, if a parameterized class refers to itself recursively but with different type instantiations, it is impossible to give types that are covariantly refined along with inheritance to such recursive references.

**Our Contributions.** In this paper, we propose self type constructors, which integrate MyType and generics so that MyType is a type constructor, which can take type arguments just like ordinary generic class names. We demonstrate the expressive power of self type constructors by means of examples. In particular, we show collections with methods map() and flatMap(), which can be given the desired signatures by using self type constructors.

To rigorously show that our proposal is safe, we formalize a core language FLJ₁ of self type constructors by extending FLJ [7], a core calculus of LOOJ, which in turn extends Featherweight GJ [17], and prove its type soundness. Since the idea of self type constructors is simple, the type system of FLJ₁ is fairly straightforward (except for a few subtle restrictions, discussed in the paper) and the type soundness proof is easy to understand for the readers who are already familiar with that of Featherweight GJ.

Actually the problem pointed out here is not new; it has been tackled by other people [24, 1]. Their solutions required a highly sophisticated combination of advanced typing features including abstract type members [36, 16, 30], higher-order type constructors [24, 1], and F-bounded polymorphism [10]. Compared with them, our proposal is superior in the following points: (1) there is less boilerplate code because MyType automatically supports covariant change of method signatures and the way generic classes are parameterized is much simpler, (2) F-bounded polymorphism, which is a source of complication in the semantics and meta-theory, is not needed, (3) the “order” of type constructor polymorphism is lower.

We summarize our contributions as follows:

- the proposal of self type constructors with a demonstration of their expressive power by means of examples;
- a formalization of the type system of self type constructors; and
- a proof of type soundness.

For brevity, the proofs of the theorems are only sketched with the statements and proof sketches of required lemmas.

**The Rest of This Paper.** Section 2 reviews the idea of MyType and the type system of LOOJ and, then discusses a limitation of LOOJ with respect to generic classes with recursive interfaces. Section 3 informally describes the idea of self type constructors as a solution to the problem. Section 4 formalizes self type constructors as FLJ₁, which is an extension of FLJ, a small model of LOOJ. Section 5 investigates the interaction between self type constructors and other advanced typing features. Section 6 discusses related work and Section 7 concludes. Hereafter, we use the keyword This, as done in [19, 33], for MyType.

## 2. The Type System of LOOJ and Its Limitation

In this section, we first review the type system of LOOJ [7] and then describe its limitation caused by the fact that This stands for the current class with its type parameters.

### 2.1 This and Exact Types

In LOOJ, the keyword This (more precisely, ThisClass) represents the class in which it appears; moreover, when the class is inherited, the meaning of This changes covariantly. A typical use of This is in methods with recursive interfaces, that is, methods that take or return the same type as the receiver. Consider the following class definitions:

```java
class C {
    int field1;
    boolean isEqual(This that){
        return this.field1 == that.field1;
    }
}
class D extends C {
    int field2;
    boolean isEqual(This that){
        return super.isEqual(that) && this.field2 == that.field2;
    }
}
```

Class C declares a binary method (i.e., a method that takes the same type as the receiver [6]) isEqual() to compare the...
receiver with another object of the same type. This refers to class C in class C whereas it refers to class D in class D. So, in the overriding definition of isEqual(), the field access that.field2 in isEqual() in class D is legal.

When members are accessed on an object, the signatures of the members are obtained by replacing This with the (static) class name of the receiver object. For example, if isEqual() is invoked on an expression of type C, the signature is C—boolean. On the other hand, if isEqual() is invoked on type D, the signature is D—boolean.

One advantage of using This is that programmers can avoid unnecessary uses of typecasts, which sidestep static typechecking and may fail at run-time if used carelessly. For example, if we wrote isEqual() with the argument type being C, the access to field2 would require a typecast (D) for that, since D has to override the method with the same signature (in Java) but C does not have field2.

**Exact Types for Safe Binary Method Invocations.** However, it is not safe to allow the invocation of binary methods naively. Consider the following code:

```java
C c1, c2;
... c1.isEqual(c2); // unsafe
```

Although this invocation is well typed under the above-mentioned interpretation of This and the usual typing rule for method invocations, it can fail at run-time—if c1 refers to an object of class D and c2 refers to an object of class C, then the overriding definition of isEqual() will be executed and the field access that.field2 fails since c2 does not have field2. The problem here is that the run-time signature of isEqual() can be different from the compile-time one.

LOOJ introduces exact types [7, 19] in order to guarantee the safe invocations of binary methods such as isEqual(). While an ordinary type C means an object of class C or its subclasses, an exact type @C means the object of class C exactly, excluding its proper subclasses. In other words, a variable or field of type @C always refers to an instance of class C.

With the help of exact types, the safe typing rule for method invocations becomes: "the receivers of binary methods should have exact types." Thanks to this rule, the run-time and compile-time signatures of a binary method invocation will be the same1. The following method invocations illustrate type-checking under this rule:

```java
@C c1; C c2, c3;
c1.isEqual(c3); // 1: legal
c2.isEqual(c3); // 2: illegal
```

The first invocation is legal (and in fact safe) since the receiver is of @C, an exact type, and the argument type C is

1Obviously, this rule loses the benefit of dynamic dispatch—the method body called is determined at run-time. We have overcome this disadvantage by proposing local exactization [33], which will be shown in Section 3.

```java
interface Comparable {
  int compareTo(@This that);
}
interface Iterator<T> {
  @T next();
  @T peek();
  boolean hasNext();
}
```

**Figure 1.** Interfaces of Comparable and Iterator written in LOOJ.

a subtype of the parameter type C. The second invocation, which is unsafe as we have seen, is rejected by the type system since the receiver’s type is not exact. Hereafter, we call types without @ inexact.

2.2 Limitation of LOOJ

Although LOOJ has also generics, This is rather orthogonal to generics. When This appears in generic class C<X>, This means C<X> including the type parameter X of the generic class C. As a result, LOOJ cannot express an interface that refers to the same generic class recursively but with different type arguments. This is a significant limitation because such an interface naturally arises in practice, for example, in a generic collection class with method map(), which converts a collection to the same kind of collection of a different element type. We elaborate on this problem, which was also pointed out by Altherr and Cremet [1] and Moors, Piessens, and Odersky [24], below.

Figures 1 and 2 show our running example2 adapted from [24]. Suppose that we are developing a class hierarchy of collections by the iterable-and-iterator idiom. The top of the hierarchy is Iterable<T>, an abstract collection of the elements of type @T. (The element type needs to be exact to allow binary method compareTo() to be invoked on the elements in a subclass SortedList<T>.) List<T> is a concrete class implementing Iterable<T>. SortedList<T> is an extension of List<T> and the elements of its instance are expected to be sorted in an ascending order. Since sorting involves comparison between elements, the element type T in SortedList<T> is refined to be a subtype of Comparable, which declares binary method compareTo().

In this example, we omit the implementation of the data structure in each concrete class. So, the bodies of add() and iterator() are omitted. Method add() adds a new element to the data structure. Moreover, in SortedList<T>, add() keeps the elements being sorted. An invocation of iterator() on List<T> returns a new iterator object that iterates over the elements of the receiver; that on

2More precisely, LOOJ distinguishes MyType in classes and interfaces and uses ThisClass for class types and ThisType for public interfaces. In this paper, we use This for both. This is safe as long as the receiver of a binary method invocation has an exact class type, as pointed out in [7].
Now, consider that we try to write types for methods `map()` and `flatMap()` in class `List<T>`\(^3\). The requirement is as follows: the method `map()` is a polymorphic method that takes a type \(U\) and a function from type \(T\) to \(U\), and then returns a list whose element type is \(U\). Moreover, we naturally expect that the invocation of `map()` on a list returns a list and that on a sorted list returns a sorted list. The method `flatMap()`, which is also polymorphic and takes \(U\) as a type argument, takes a function from \(T\) to lists of \(U\) and returns a new list obtained by concatenating the results of applying the function to each element of the receiver list. Similarly, we expect that `flatMap()` returns the same kind of lists.

It is impossible, however, to express such a requirement with LOOJ types. The problem is that this cannot be used to express the requirement that the returned list is the same kind of list as the receiver because `this` always refers to the generic class instantiated with the declared type parameters: for example, in `List<T>`, this always refers to `List<T>` but cannot be used to refer to, say, `List<U>`.

Covariant refinement of return types introduced to Java since 5.0 does not solve the problem, either. All we can do at best is as follows:

```java
... void add(@T t) { ... }
@This @This create() { ... } // discussed later
@This append(@This that) {
    @This newList = create();
    for (@T t: that) newList.add(t);
    return newList;
}
@This append(@This that) {
    @This newList = create();
    for (@T t: this) newList.add(t);
    return newList;
}
<U> ??? map(T->U f) { ... }
<U> ??? flatMap(T->??? f) { ... }
}
class SortedList<T extends Comparable> extends List<T> {
    ... void add(@T t) { ... }
    @This @This create() { ... }
    @This append(@This that) {
        @This newList = create();
        Iterator<T> iter = that.iterator();
        while (iter.hasNext()) {
            if (iter.peek().compareTo(iter2.peek()) < 0)
                newList.add(iter.next());
            else newList.add(iter2.next());
        }
    }
    @This append(@This that) {
        @This newList = create();
        Iterator<T> iter = this.iterator();
        Iterator<T> iter2 = that.iterator();
        while (iter.hasNext() && iter2.hasNext()) {
            if (iter.peek().compareTo(iter2.peek()) < 0)
                newList.add(iter.next());
            else newList.add(iter2.next());
        }
        return newList;
    }
}
```

Figure 2. Collection classes written in LOOJ.

SortedList\(<T>\) returns a new object that iterates over the elements in the ascending order.

`Iterable<T>` declares binary method `append()` that takes the same kind of collection as the receiver, appends all the elements of the receiver and argument, and then returns a new collection of the same kind. Both `List<T>` and `SortedList<T>` implement `append()`. In `List<T>`, the elements of that are simply connected to the tail of those of this by iteration. In `SortedList<T>`, on the other hand, `append()` is overridden so that the elements of this and that are merge-sorted. The algorithm assumes that the elements of that has been sorted. This assumption is correct since that has type `@This` referring to `SortedList<T>`, a sorted list. The implementation of the factory method `create()` will be discussed later.

\(^3\)In this paper, we assume the existence of first-class functions, which can be seen in Scala [29], and use the notation \(S\to T\) for the type for functions from \(S\) to \(T\). They can be simulated in Java by representing functions by objects implementing a generic interface with method \(T\ apply(S\ x)\), where \(S\) and \(T\) are type parameters and function applications by invocation of `apply()`.
class List<T> extends Iterable<T> {
    @This<T> create() { ... }
    @This<T> append(@This<T> that) { ... }
    @This<U> map(T->@This<U> f) {
        @This<U> newList = this.<U>create();
        for (@T t: this)
            newList.add(f(t));
        return newList;
    }
    @This<U> flatMap(T->@This<U> f) {
        @This<U> newList = this.<U>create();
        for (@T t: this)
            newList = newList.append(f(t));
        return newList;
    }
}

Figure 3. The implementation of map() and flatMap() in class List<T> written by using self type constructors in the preliminary syntax.

3. Self Type Constructors

In this section, we introduce self type constructors to solve the problem described in the previous section. Self type constructors are the integration of This and generics where This can take type arguments in a generic class definition. First, we begin with a simple use of self type constructors and rewrite the classes in the previous section (Section 3.1). Second, we identify a subtlety that arises when the upper bound of a type parameter is refined in a subclass (Section 3.2). This leads us to distinguishing two kinds of type parameters: ones that are "tied" to self type constructors and whose upper bounds can be refined; and ones with opposite properties. We also show the use of other related mechanisms, including constructor-polymorphic methods [1, 24] (Section 3.3), local exactization [33] (Section 3.4), and non-heritable methods [33] (Section 3.5), which are useful in this context.

3.1 This as a Type Constructor

The idea of self type constructors is simple—it is just to consider that This refers to a generic class name where it appears, without type parameters, and that it can take type arguments just like ordinary generic class names. Since self types are now type constructors, namely, a type-level function that takes types to yield another type, we call This a self type constructor. Figure 3 shows class List<T> rewritten with self type constructors. In this class definition, This is used as a type constructor that takes one type. The return type of map() can now be expressed as @This<T> and, similarly, method flatMap() also has the signature that reflects our intention. Note that the argument type of append() is now @This<T> instead of @This because This always needs to take one argument so as to be a proper type. The factory method create() becomes a polymorphic method, parameterized by an element type of the list to be created, so that it can be called from map() and flatMap().

The following code illustrates invocations of map():

@List<Integer> intList=...;
@SortedList<Integer> intSList=...;
Integer->Float intToFloat=...;
@List<Float> floatList
    =intList.<Float>map(intToFloat); // 1
@SortedList<Float> floatSList
    =intSList.<Float>map(intToFloat); // 2

Each invocation returns a list of the same kind as the receiver, as expected, but the element type is converted from Integer to Float. The return type is obtained by replacing This with the constructor part (the simple class name, without types surrounded by < and >) of the receiver type. For example, the return type of the first invocation is obtained by replacing This in @This<Float> by List.

3.2 Type Parameters with Refinable/Fixed-Bounds

Although the first idea is quite simple, the interaction of self type constructors with the inheritance mechanism is subtler than it may have appeared. We will see that type parameters have to be distinguished into two kinds: one that allows upper bounds to be refined and the other with fixed upper bounds.

The following code reveals the problem:

class List<T> {
    @This<String> strlist;
}
class NumList<T extends Number> extends List<T> {
    ...
}

The class List<T> above is well defined—@This<String> is a well-formed type since the argument String is a subtype of Object, the (implicitly specified) upper bound of T. However, if strlist were inherited to NumList<T>, its type @This<String> would not be well formed any longer because String is not a subtype of Number, which is a new upper bound for T. In fact, the class SortedList suffers from the same problem since the upper bound of T is refined to Comparable. Similarly, it will be a problem to define a subclass that fixes a type parameter of its superclass

class IntList extends List<Integer> {...}

since @This in IntList takes no type arguments and @This<String> does not make sense, either.

In short, the problem is that, without any restriction, the range of acceptable type arguments for (or, even the arity of) self type constructors changes in a subclass and types in inherited members may become meaningless. Although such ill-formed types do no harm to type safety in the sense that execution does not get stuck by no-such-field or no-such-method errors (as long as the type system disallows the
instantiation of ill-formed types), we believe it is reasonable to prohibit them from appearing. A similar problem has also been pointed out in the context of Scala, by Moors, Piessens, and Odersky [26], who developed a mechanism to prevent ill-formed types from appearing.

In order to solve the problem, we distinguish two kinds of type parameters of a generic class: parameters where their upper bounds can be refined (simply called refinable parameters) and ones with fixed bounds (simply called fixed parameters). On the one hand, a refinable parameter (1) allows its upper bound to be refined covariantly (or can be fixed in a subclass as in IntList above), and (2) is included in the meaning for C: for example, when parameter T in class C<T> is refinable, this refers to C<T> instead of C. In LOOJ, all type parameters are considered refinable and inheritance obviously preserves well-formedness of type expressions. On the other hand, a fixed parameter (1) cannot be instantiated in an extends clause (as in IntList above), (2) requires its upper bound to be the same as that of the superclass, and (3) is not a part of the meaning of this. So, in order to be a proper type, this has to take as many type arguments as the number of fixed parameters. Since the upper bound does not change along with inheritance, well-formedness of types, especially the fact that the actual type arguments for C are subtypes of the upper bounds of the corresponding formal, are preserved.

To make the distinction clear in the syntax, we enclose refinable parameters by < and > before fixed parameters, which are enclosed by [ and ]. For example, we write

```java
class C<T>[U]{
	This[U] f;
<S> This[S] m();
}
```

Here, C has one fixed parameter U, so This takes one parameter, even though C has two type parameters in total. In this class, the self-type constructor This means that application to type S yields This[S], which is a subtype of C<T>[S]. We generalize the notion of the constructor part of the class to be a class name together with its refinable parameters. For example, the constructor part of C is C<T>.

This distinction does not affect subtyping, which is pointwise as in Java: if class C<T>[U] extends D<T>[U], for any types T1 and T2, C<T1>[T2] is a subtype of D<T1>[T2].

Using both kinds of type parameters, it is now possible to define SortedList, where the element type is a subtype of Comparable, as a subclass of List, without being bothered by ill-formed types. Figure 4 shows the new definitions, where the method bodies remain unchanged from the ones in Figure 3. In the new definition, class Iterable has one refinable parameter Bound and one fixed parameter T, bounded by Bound. The point is that, although T’s upper bound cannot be refined, it can actually be indirectly refined by changing Bound’s upper bound. The extension of List by SortedList is legal since the refinement of refinable parameters Bound is allowed and T has the same bound Bound as its superclass does. Note that in class List<Bound>[T], type @This[Integer] will not be well formed since String is not a subtype of Bound.

Although it is not possible that the number of fixed parameters decreases in a subclass (because a subclass cannot instantiate them), it is possible for a subclass to have more fixed (and also refinable) parameters. (In fact, such augmentation is even necessary because Object at the top of the class hierarchy has no type parameters.) When a subclass is given an additional fixed parameter, the signature of an inherited member must change. An easy example is as follows:

```java
class List<Bound>[T extends Bound] extends Object {
    @This[T] clone(){ ... } // overriding definition
}
```

Here, Object is assumed to have method clone() that returns @This to express the intention that it must return an object of the same type as the receiver. Now, clone() is inherited to List<Bound>[T], the return type changes to @This[T], not @This, which is not a proper type. So, the overriding definition must return type @This[T]. Although the return types look syntactically different, their meaning is the same, in that they refer to the constructor part of the class.

We summarize other restrictions on how two kinds of type parameters can be used in class definitions by giving the general form of the header of a class definition. Ill-formed types can be prevented if a class definition is of the form

```java
abstract class Iterable<Bound>[T extends Bound]
    extends Iterator<T>[T] { ... }
}
class List<Bound>[T extends Bound]
    extends Iterable<Bound>[T] { ... }
class SortedList<Bound extends Comparable>
    extends List<Bound>[T extends Bound] { ... }
```

Figure 4. Collections with self-type constructors, finally.
D’s fixed parameter is bounded by $B_3$, and
type $T_y$, which instantiates the refinable parameter of $D$,
does not include $T_3$.

As in the formal calculus given in the next section, it is easy
to extend this rule to sequences of type parameter declara-
tions.

3.3 Constructor-Polymorphic Methods
As studied in [8, 34, 19], it is convenient to allow method
declarations that work uniformly over different constructors.
The following example shows a method that takes two lists
of the same kind and returns a truth value if the two have
the same length and the elements at the same positions are
equal:

```java
static <T extends Comparable, L extends List<Comparable>> boolean isEqualLists(@L[T] n1, @L[T] n2){
    Iterator<T> iter=n2.iterator();
    for(@T t: n1){
        if(!iter.hasNext()) return false;
        if(t.compareTo(iter.next()) != 0)
            return false;
    }
    return !iter.hasNext();
}
```

The method above has two parameters. The first one $T$
is an ordinary type variable, which ranges over subtypes of
Comparable but the second one $L$ is a type constructor vari-
able—this is clear from its upper bound $List<Comparable>$,
which requires one more argument to be a proper type. The
method can be invoked, as follows:

```java
@SortedList<Comparable>[String] strList1, strList2;
...<String,SortedList<Comparable>>
isEqualLists(strList1, strList2);
```

Here, the actual arguments for the method invocation is ex-
roppedly specified. Since $String$ implements $Comparable$,
this method invocation is well typed. The development of an
inference algorithm is left for future work.

Constructor-polymorphic methods introduce type param-
eters which range over type constructors. In the formal type
system presented in the next section, this will be also re-
garded as an (implicit) type constructor parameter, which is
bounded by the constructor part of the class where it appears.
In this paper, we allow such “higher-order” type parameter
only for methods, not for classes, to avoid a type construc-
tor parameterized by another type constructor. However, this
is not an essential restriction—see Section 5. Our main mo-
tivation for the restriction is rather to show interesting pro-
gramming is possible without higher-order type constructors
as used in other work [1, 24].

3.4 Local Exactization for Invoking Binary Methods
on Inexact Types
The examples so far show only invocations on exact types.
Actually, with the help of local exactization [33], invocations
of binary methods$^4$ on inexact types are possible. Consider
the following example:

```java
class ReversibleList<Bound>[T extends Bound] extends Iterable<Bound>[T] {
    void reverse(){ ... }
}

static <L extends ReversibleList<Comparable>>
boolean isPalindrome(@L[String] strList)
    String->@L[Char] stringToChars=
        (String str) => { // first-class function
            @L[Char] newList=strList.<Char>create();
            for(Char c: str)
                newList.add(c);
            return newList;
        }
    @L[Char] charList=\strList.<Char>flatMap(stringToChars);
    @L[Char] charList2=charList.clone().reverse();
    return <Char,L>isEqualLists(cs, cs2);
}
```

The static method isPalindrome() judges if the given list
of strings represents a palindrome. For example, when it
takes the list of “borrow”, “or” and “rob” as an input, it
returns true. The argument is expected to be a reversible
list or its subclasses (whose definitions are not given here).
We wish to apply this method to $strList$ of inexact type
ReversibleList<Comparable>[String]. Local exacti-
zation is used to make an inexact type temporarily exact in a
local scope, similarly to wildcard capture [38], as follows:

```java
exact strList as x, X in {
    // x has type @X[String]
    // X extends ReversibleList<Comparable>
    <X>isPalindrome(x);
}
```

At compile-time, the type of $x$ is an exact type $@X[String]$.
where the type constructor variable $X$ is assumed to extend
ReversibleList<Comparable>, which comes from the
constructor part of the type of $strList$. At run-time, $x$ is
bound to the value of $strList$ and $X$ to the constructor part
of the run-time type of $strList$.

3.5 Nonheritable Methods for Implementing create()
There are several ways to implement create() in Figure 4.
The problem in defining create() is that one cannot return
an instance created from a concrete class such as List or
SortedList, since this is only a subtype of List<Bound>

$^4$Here, the meaning of “binary methods” is slightly expanded so that they
now refer to methods taking another instantiation of This; so the argument
type is not necessarily the same as the type of this.
or SortedList<Bound> but not vice versa. Here, we introduce two alternatives.

One is to use the abstract factory pattern [15]. First, we prepare abstract class Factory as the common interface for factories and a concrete factory ListFactory<Bound> and then augment class List<Bound>[T] with a reference to Factory<Bound,This>.

abstract class Factory<Bound,This> {
    <U extends Bound> @C[U] create();
}
class ListFactory<Bound> extends Factory<Bound,List<Bound>> {
    <U extends Bound> @List<Bound>[U] create() {
        return new List<Bound>[U](this);
    }
}
class List<Bound>[T extends Bound] extends Factory<Bound,This> {
    Factory<Bound,This> factory;
    List<Bound>This f) {
        this.factory = f;
    }
    <U extends Bound> @This[U] create() {
        return factory.<U>create();
    }
}

A factory must be implemented for each collection class and has to be supplied when a concrete collection is instantiated. Note that class Factory is parameterized by C, which is a constructor variable. So, Factory is a higher-order type constructor.

An alternative is to use nonheritable methods [33]. A nonheritable method is one that is not inherited to subclasses, which must rewrite the same methods. In exchange for this restriction, a nonheritable method allows This and the constructor part of the declaring class to be compatible, i.e., both types are subtypes of each other. This typing feature is, in a nutshell, the trick that allows object creations without constructor declarations, expressions, and values is given in Figure 5. The metavariables C and D range over class names; V, W, X, Y, and Z range over type constructor variables; f and g range over field names; m range over method names; x and y range over variables. The symbols $\triangle$ and $\triangledown$ are read extends and return, respectively.

flj is not a pure extension of FLJ. The calculus does not include nonheritable methods, but they are easy to add, as formalized in [33]. We omit interfaces for simplicity. Typecasts are dropped since we aim at safe and extensible programming without typecasting, a possibly unsafe operation. Section 4.1 defines the syntax; Sections 4.2 and 4.3 define the type system; Section 4.4 defines the operational semantics. Finally, we show type soundness in Section 4.5.

4. FLJ: A Formal Core Calculus

In this section, we formalize the idea described in the previous section as a small core calculus called FLJ, based on FLJ [7]. What we model here includes self type constructors, constructor-polymorphic methods, and local exactization, as well as the usual features of calculi of the FJ family, that is, fields, methods, object creations and recursion by this. The important restriction posed on FLJ is that classes can be parameterized only by proper types, but cannot by type constructors. So, type constructors in FLJ are first-order—type constructors takes only types, but not take type constructors. As the last section has shown, this first-order restriction still allows self type constructors to express the collection hierarchy if factory methods are implemented by using nonheritable methods. The relaxation of this restriction will be discussed in Section 5. Since F-bounded polymorphism is kicked out, FLJ is not a pure extension of FLJ. The calculus does not include nonheritable methods, but they are easy to add, as formalized in [33]. We omit interfaces for simplicity. Typecasts are dropped since we aim at safe and extensible programming without typecasting, a possibly unsafe operation. Section 4.1 defines the syntax; Sections 4.2 and 4.3 define the type system; Section 4.4 defines the operational semantics. Finally, we show type soundness in Section 4.5.

4.1 Syntax

The abstract syntax of types, class declarations, method declarations, expressions, and values is given in Figure 5. The metavariables C and D range over class names; V, W, X, Y, and Z range over type constructor variables; f and g range over field names; m range over method names; x and y range over variables. The symbols $\triangle$ and $\triangledown$ are read extends and return, respectively. Following the custom of FGJ, we put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing “$\mathbb{T}$ $\mathbb{T}$,” “$\mathbb{T}$ $\mathbb{T}$,” “$\mathbb{T}$ $\mathbb{T}$,” “$\mathbb{T}$ $\mathbb{T}$,” where n is the length of $\mathbb{T}$ and $\mathbb{T}$. Sequences of field declarations, parameter names, and method definitions are assumed to contain no duplicate names. We write the empty sequence as $\text{\textbullet}$ and concatenation of sequences using a comma. We write $\mid \mid$ for the length of a sequence. As in FGJ, every class has a single constructor that takes initial values of all the fields and assigns them; we omit constructor declarations for simplicity.

A type constructor K is either a type constructor variable X or a nonvariable type constructor C[H]. The application of type constructor K to a sequence of fields yields K[H], which can be also a type constructor since partial application of type constructors is allowed in FLJ. In what follows, we call H an inexact type when H does not take any arguments, in other words H is a fully-applied type constructor. A type is either
an inexact type or an exact type, which is obtained by adding \( \emptyset \) to an inexact type. Since this language is expression-based, the body of a method is a single return statement and the body of exact is a single expression, rather than a statement as in the previous section. The superscript \( i \) is introduced to exact just because of the proof of subject reduction, but programmers need not specify it. We assume that the set of (type) variables includes the special variable \( \textbf{this} \) (\( \textbf{this} \), resp.), which cannot be used as the name of a (type, resp.) parameter to a method.

A class table \( CT \) is a finite mapping from class names \( C \) to class declarations \( L \) and is assumed to satisfy the following sanity conditions: (1) \( CT(C) = \texttt{class } C \ldots \) for every \( C \in \text{dom}(CT) \); (2) \( \texttt{Object} \not\in \text{dom}(CT) \); (3) for every class name \( C \) (except \( \texttt{Object} \)) appearing anywhere in \( CT \), we have \( C \in \text{dom}(CT) \); and (4) there are no cycles in the inheritance relation induced by \( CT \). A program is a pair \((CT, e)\) of a class table and an expression. In what follows, we assume a \textit{fixed} class table \( CT \) to simplify the notation.

4.2 Lookup functions

We give functions to look up field or method declarations. The function \( \texttt{fields}(C<\Theta>) \) returns a sequence \( T \ T \) of field names of class type \( C<\Theta> \) with their types. The function \( \texttt{mtype}(C<\Theta>) \) takes a method name and a class type as input and returns the corresponding method signature of the form \( C<\Theta>\triangleright T \rightarrow T_0 \) in which \( Y \) are bound in \( T \) and \( T_0 \). The functions are defined by the rules below, which are essentially the same as those of FGV.

\[
\text{class } C<\Theta>\triangleright \{ Y<\Theta> \} \triangleleft [N[Z]\{ T \ T ; \ R \} \\
\text{fields}(C<\Theta>) = \{ Y ; g \} \\
\text{mtype}(m, C<\Theta>) = [F/X, \Theta/Y]([Z; U_0 \rightarrow U_0])
\]

We write \( [F/X, \Theta/Y] \) for the capture-avoiding simultaneous substitution of \( F_1 \) for \( X_1 \), \ldots, of \( F_n \) for \( X_n \), of \( G_1 \) for \( Y_1 \), \ldots, of \( G_m \) for \( Y_m \). Replacing \( X \) in \( [F/X] \) by an application \( \kappa(\Pi) \) yields \( \kappa(\Pi, T) \). In general, we identify \( \kappa(\Pi, [T]) \) and \( \kappa(\Pi, [T]) \). The type substitutions in these rules look a little more complicated since the two kinds of class parameters are separated syntactically. Here, \( m \not\in \mathbb{R} \) means that the method of name \( m \) not does exist in \( \mathbb{R} \).

4.3 Type System

The main judgments of the type system consist of one \( \Delta \vdash T \vdash S \) for subtyping, one \( \Delta \vdash I : I \) for type constructor well-formedness (\( k \) is a kind, defined later), one \( \Delta \vdash T \text{ ok} \) for type well-formedness, one \( \Delta_1 \vdash \Delta_2 \vdash \Gamma \) for bound environment well-formedness, and one \( \Delta; \Gamma \vdash e : T \) for expression typing. Here, \( \Delta \) is a \textit{bound environment}, which is a finite mapping from type (constructor) variables to type constructors, written \( X \not\in \Delta ; \Gamma \) is a \textit{type environment}, which is a finite mapping from variables to types, written \( X : T \). Following the custom of FGV [17], we abbreviate the sequence of judgments in the obvious way: \( \Delta \vdash S_1 : T_1, \ldots, \Delta \vdash S_n : T_n \Rightarrow \Delta \vdash \overline{S} : T ; \Delta \vdash I_1 : k_1, \ldots, \Delta \vdash I_n : k_n \Rightarrow \Delta \vdash \overline{I} ; \Delta \vdash T_1 ; \ldots, \Delta \vdash T_n \Rightarrow \Delta \vdash T \). And, \( \Delta; \Gamma \vdash e_1 : T_1, \ldots, \Delta; \Gamma \vdash e_n : T_n \Rightarrow \Delta; \Gamma \vdash \overline{e} : T \).

4.3.1 Bounds of Types

The function \( \texttt{bound}_{\Delta}(\), defined below, takes an inexact type as input and returns a class type, which is the least upper bound of the input type.

\[
\text{bound}_{\Delta, X < 1, \Delta_2} (X(\Pi)) = \text{bound}_{\Delta_1} (I(\Pi)) \\
\text{bound}_{\Delta} (N(\Pi)) = N(\Pi)
\]

If the input begins with a type constructor variable (the first rule), the function will be recursively applied to the output, in which, again, a variable can appear at the head.

---

**Figure 5.** FLJ. Syntax.
4.3.2 Subtyping

The subtyping judgment $\Delta \vdash S <: T$, read as “$S$ is a subtype of $T$ under $\Delta$,” is defined below. This relation is the reflexive and transitive closure of the `extends` relation with the rule that an exact type is a subtype of its inexact version. The notable rules are S-CLASS and S-APPLY. The former is defined to give subtyping between type constructors—the arguments for $Z$ are empty in both sides of the conclusion. This partial application does not cause a free occurrence of type variables in the right side of the conclusion since $N$ does not contain any of $Z$ (mentioned later in T-CLASS). S-APPLY says that if type constructors are in a subtyping relation, so are their applications to the same argument.

$$
\Delta \vdash T <: T \quad (S-REFL)
$$

$$
\Delta \vdash S <: T \quad \Delta \vdash T <: U \\
\Delta \vdash S <: U \quad (S-TRANS)
$$

$$
\Delta \vdash X <: \Delta(X) \quad (S-VAR)
$$

$$
\text{class } C<F>[\{Y,I,Z,O\}] <: \{T,I,T/I,H,I\} \quad (S-CLASS)
$$

$$
\Delta \vdash G <: H \\
\Delta \vdash G[I] <: H[I] \quad (S-APPLY)
$$

$$
\Delta \vdash \emptyset I <: I \quad (S-EXACT)
$$

4.3.3 Well-formedness

The well-formedness judgments consist of two judgments: one $\Delta \vdash I :: k$ for type constructors and one $\Delta \vdash T ok$ for proper types. In the former, $k$ is a kind of $I$. The kind of a type constructor represents the arity of the type constructor and the upper bound for each parameter. Thanks to kinds, it is possible to check if applications of type constructors are well-formed. The definition is:

$$
k :::= * \mid X <: I.k \text{ kinds}
$$

Kind $*$ is one for inexact types, which take no arguments. Kind $X <: I.$ is one for type constructors that take a single argument, which must be a subtype of $I$. We write kind $\forall n X <: I.*$, as an abbreviation of $X_1 <: I_1, \ldots, X_n <: I_n \ast$, for $n$-argument type constructors. Since type constructors are first-order, the upper bounds that appear in a kind always have kind $*$. The well-formedness judgment $\Delta \vdash I :: k$ for type constructors is read as “type constructor $I$ has kind $k$ under $\Delta$.” The rules are defined below. $\emptyset I$ has kind $*$. A type constructor variable $X$ has the same kind as its upper bound. Similarly to S-CLASS, WK-CLASS is defined so that the conclusion is a class that is partially applied, dropping the arguments for $Z$. The rule says that type constructor $C<F>(\mathbb{R})$, which takes as many arguments as the length of $\mathbb{R}$, has a kind if arguments $\mathbb{F}$ and $\mathbb{S}$ have kind $*$ and they conform with the corresponding upper bounds. WK-APPLY is the rule for application of a type constructor to a type constructor, which must have kind $*$ and be a subtype of the upper bound $I$, written in the kind $X <: I. k$ of the applied type constructor $G$.

$$
\Delta \vdash \text{Object}::<* \quad (WK-OBJECT)
$$

$$
\Delta \vdash \Delta(X) :: k \\
\Delta \vdash X :: k \quad (WK-VAR)
$$

$$
\text{class } C<F>[\{Y,I,Z,O\}] <: \{T,I,T/I,H,I\} \quad (WK-CLASS)
$$

$$
\Delta \vdash G ::= X <: I.k \quad \Delta \vdash H ::= * \\
\Delta \vdash H <: I \quad (WK-APPLY)
$$

The well-formedness judgment $\Delta \vdash T ok$ for types is read as “type $T$ is well formed under $\Delta$.” The rules are defined below. If type constructor $I$ has kind $*\rightarrow I$ is an inexact type, both exact type $\emptyset I$ and inexact type $I[]$ are well-formed types.

$$
\Delta \vdash I ::= * \quad \Delta \vdash I [] ok
$$

$$
\Delta \vdash I [] \quad \Delta \vdash I [] ok
$$

The bound environment well-formedness judgment $\Delta_1 \vdash \Delta_2 :: \mathbb{K}$, read as “type environment $\Delta_2$ has kinds $\mathbb{K}$ with respect to $\Delta_1$,” is defined as follows:

$$
\Delta \vdash \bullet ::= \bullet \quad \Delta_1 \vdash \Delta_2 :: \mathbb{K} \\
\Delta_1 \vdash (\Delta_2, X <: I) :: (\mathbb{K}, k')
$$

Note that the rules are defined so as to kick out F-bounded polymorphism [10] from FLJ. The scope of a type variable in a bound environment is the following type variable declarations. So, a type variable cannot appear in its upper bound.

4.3.4 Typing

Closing of Types. Before proceeding to expression typing, we define the judgment $\Delta \vdash T \square X <: T$, read as “type $T$ is enclosed under $X <: I$” or “$T$ is a minimal supertype of $S$ without $X$”, for closing of types [18]. The rules are defined below. This judgment is used in the typing rules for exact expressions to prevent the type variable introduced from escaping. The basic idea is to lift the type variable to its supertype so that it does not appear in the result. The left rule says that if type $T$ does not contain the variable $X$, the result is the same $T$. The other rules say that both $X[I]$ and $\emptyset X[I]$ close to $I[I]$ under $X <: I$ if $I$ does not contain $X$. Note that $\emptyset X[I]$ does not close to $\emptyset I[I]$ since the subtyping relation is $\emptyset < X <: I$, but $\emptyset X \not< \emptyset I$. Here, $f\iota(T)$ returns the set of type variables that appear in $T$. 
Expression Typing. The typing judgment for expressions of the form \( \Delta; \Gamma \vdash e : T \), read as “under bound environment \( \Delta \) and type environment \( \Gamma \), expression \( e \) has type \( T \),” defined below. The key rules are T-FIELD for field access and T-INVK for method invocation. Both rules restrict the receivers to be exact. (So, to access members on inexact types, we first exactize the types of the receivers.) The rule T-FIELD means that the type of field access is obtained by looking up field declarations from the bound of \( H_0 \) and then substituting \( I_0 \) for This in the type \( T_1 \) corresponding to \( f \). Note that \( I_0 \) is obtained by dropping some \( \Omega \) from \( H_0 \) so that the type constructors This in the class where \( T_1 \) is declared and \( I_0 \) have the same arity. The selection of \( I_0 \) is correct if substitution of \( I_0 \) for This in \( T_1 \) yields a well-formed type.

The rule T-INVK is similar to T-FIELD. First, the method signature is retrieved from the receiver’s type by using \( \text{mtype} \). Then, \( I_0 \) is selected so that \( I_0 \) and This is the nearest superclass where method \( m \) is declared has the same arity. Finally, it is checked if the type arguments \( \sigma \) have some kinds and are subtypes of the corresponding formal \( \kappa \), and if the types \( \sigma \) of the arguments \( \sigma \) are subtypes of those of the corresponding formal \( \kappa \). Since \( T \) and \( \kappa \) may contain This and \( \kappa \), type substitution is applied when the subtyping checks are done.

The rule T-NEW says that the type of a new expression is the exact type of the class being instantiated. Since different fields can come from different classes, which have the different numbers of fixed parameters, we have to choose a different \( H \) for each field.

There are two rules for local exactization whether the expression \( e_1 \) to be exactized is of type inexact \( I[H] \) or exact \( \Omega[I] \). If inexact (T-EXACT), the body expression is typed under \( \Delta \) extended by \( X : I \) and \( \Gamma \) extended by \( x : \Omega[I] \). The length of \( \Pi \) is determined by superscript \( i \) so that it holds \( \Delta \vdash T_0 \) ok. Since the result type \( U_0 \) may contain the type constructor variable \( X \), the type of the whole expression is obtained by closing \( U_0 \) under \( X : \Pi \), preventing \( X \) from escaping. The rule T-EXACT2 is for the case that the expression which will be exactized is already exact. This rule is required to show the subject reduction property since an expression of inexact type eventually reduces to one (typically a value) of an exact type at run-time.

To avoid cumbersome exactization in accessing members on inexact types, we could give the following derived rules, which can be obtained by the combination of T-FIELD/T-INVK and T-EXACT1.

Method Typing. The typing judgment for method declarations is written \( \forall X < \varnothing [T] \vdash m : \text{ok} \). The rule T-METHOD depends on a predicate override that checks valid overriding of method signatures. Both are defined below. override \((m, \forall X < \varnothing [T] \vdash m : \text{ok}) \) means that class \( \forall X < \varnothing [T] \) correctly overrides the method of name \( m \) in its superclass (if exists) and the overriding signature is \( \forall X < \varnothing [T] \vdash m : \text{ok} \). Note that method signatures are \( \alpha \)-convertible and return types can be covariantly refined along with extension.

The rule T-METHOD is straightforward. The method body \( e_0 \) is typed under the bound environment made from the parameterization clauses in the class and method declaration as well as This \( \in \forall X < \varnothing X \), and the type environment, in which this has type \( \forall \text{This}[\forall] \).
The operational semantics is given by the reduction relation of the form \( e \rightarrow e' \), read “expression \( e \) reduces to \( e' \) in one step.” We require another lookup function \( mbody(m,C,X,Y,F,T) \), defined below, for a pair of natural number and method body with formal (type) parameters, written \( i,\,\bar{x},\,e \), of given method and class names. \( \bar{x} \) and \( e \) are considered bound in \( e \). The natural number \( i \) counts the difference between the number of the fixed parameters in the class that the method receiver belongs to and that in the class where the method body comes from.

4.5 Type Soundness

The type system is sound with respect to the operational semantics, as expected. Type soundness is proved in the standard manner via subject reduction and progress [40, 17]. Here, we show only the results. See Appendix A for the required lemmas and proof sketches of the theorems.

**Theorem 1** (Subject Reduction). If \( \Delta;\Gamma \vdash e : T \) and \( e \rightarrow e' \), then \( \Delta;\Gamma \vdash e' : T' \), for some \( T' \) such that \( \Delta \vdash T < T' \).

**Proof.** By induction on the derivation of \( e \rightarrow e' \) with case analysis on the reduction rule used.

**Theorem 2** (Progress). If \( \emptyset;\emptyset \vdash e : T \) and \( e \rightarrow e' \), then \( e \rightarrow e' \), for some \( e' \).

**Proof.** By induction on the derivation of \( \emptyset;\emptyset \vdash e : T \) with case analysis on the last rule used.

**Theorem 3** (Type Soundness). If \( \emptyset;\emptyset \vdash e : T \) and \( e \rightarrow e' \) with \( e' \) a normal form, then \( e' \) is a value \( v \) with \( \emptyset;\emptyset \vdash v : T' \) and \( \Delta \vdash T' < T \).

**Proof.** Immediate from Theorems 1 and 2.
5. Interactions with Advanced Typing

Features

In this section, we discuss interactions between self type constructors and other advanced typing features.

Higher-Order Type Constructors. Type constructor polymorphism, which has been implemented in Scala [24] and formalized as FGJ[+], is a generalization of generics [3] so that class and method declarations can be parameterized by type constructors. So, it allows higher-order type constructors since classes can be parameterized by type constructors, as the following example shows:

```java
class List<T> { ... }
class List2<T> extends List<T> { ... }
class C<X<T>> extends List<X> { ... }
```

Class C is parameterized by a type constructor variable X, where L must be instantiated by a subtype of List, meaning that for any X, List<X> is a subtype of List<X>. So, C is a higher-order type constructor.

Not only can higher-order type constructors be used to simulate self type constructors (which will be detailed in Section 6), but also they have their own applications such as generalized algebraic data types, described in [1]. So, it is worthwhile to extend self type constructors to be higher-order in order to acquire the advantages of both.

Although we believe that such an extension is straightforward, we will need notations to specify kinds of type parameters, as in Scala and FGJ. For example, if a type parameter X of C ranges over a type constructor that takes another first-order type constructor, the class definition should be written like:

```java
class C<X<Y<Z<V>>> extends ... { .. }
```

Higher-order type constructors can also be extended to self type constructors so that variance can be specified for both refinable and fixed parameters. Note that the variance annotations for fixed parameters are also effective on this. For example, assume that class List[+T]{ ... }, then this[S] <: this[U] holds if S <: U in the class, but at the same time this[T] cannot appear in the parameter position of method signatures. The class definition for Lists below is a straightforward adaptation of one from [12] so that it uses self type constructors:

```java
abstract class List[+T]{
  T head;
  @This[T] tail;
  <U @This[U] create(U h, @This[U] t);
  <U super T @This[U] append(@This[U] that){
    return this.<U>create(head,
                   (tail==null ? that : tail.<U>append(that)));
  }
}
```

(Here, "<U super T>" means that type variable U has a lower bound T.) This definition satisfies the restriction on type parameters. So, in addition to the fact that append() of its subclasses will return the same kind of lists as the receiver, thanks to the use of @This, List is a covariant type operator. The example of parser combinators [25] can be similarly adapted.

Wildcard Types. Wildcard types [38], derived from variant parametric types [18], are introduced to Java to relax invariance on parameters in a generic class as well as definitionsite variance above. The difference is that variances annotations appear at each use of generic classes. Wildcards can be easily adapted to self type constructors and they are useful to write common interfaces for different instantiations of a generic class, for example:

```java
List<Comparable>[]? extends Number] list1;
```

Above, wildcard ? extends Number means a certain type that is a subtype of Number. So, list1 represents a list of the elements of a certain subtype of Number. So, this is a common interface of, for example, List<Comparable>[Integer], List<Comparable> [Float], and so on.

The introduction of wildcards to FLJ could relax the rules for closing of types in Section 4. The following example illustrates an expression untypeable with T-FIELD’ in FLJ.

```java
class List[T] {}
class C{ List[This] f; }
C c;
c.f; // untypeable
```

The type List[This] cannot be closed under This <: C since This appears in the argument position. In the presence of wildcards, List[This] would be closed as List[This] @ThisC List[? extends C] and c.f is given type List[? extends C].

Note that, in general, the introduction of wildcard types in this setting will lead to wildcards at a type constructor level such as D<? extends List>, where List is a type constructor.

---

5 We assume that Number implements Comparable, unlike in Java.
interface Comparable<T> {  
    int compareTo(T that);
}
interface Iterator<T> {  
    T next();
    T peek();
    boolean hasNext();
}

Figure 6. Interfaces Comparable<T> and Iterator<T>

abstract class Iterable<Bound<_,_>, T extends Bound<T>>{
    type Self<X extends Bound<X>> extends Iterable<Bound,X>;
    abstract Self<T> append(Self<T> that);
}
class List<Bound<_,_>, T extends Bound<T>>{
    type Self<X extends Bound<X>> extends List<Bound,X>;
    Self<T> append(Self<T> that){ ... }
    <U extends Bound<U>> Self<U> map(T->U f){ ... }
}
class SortedList<T extends Comparable<T>>{
    type Self<X extends Bound<X>> extends List<Comparable,X>;
    Self<T> append(Self<T> that){ ... }
    <U extends Comparable<_,_>> Self<U> map(T->U f){ ... }
}
class NumericList<T extends Number>{
    type Self<X extends Bound<X>> extends List<Number,X>;
    <U extends Number> Self<U> map(T->U f);
}

Figure 7. Collections in Scala

6. Related Work
Much recently, Altherr and Cremet [1] and Moors, Piessens, and Odersky [24] have introduced type constructor polymorphism into class-based object-oriented programming languages. They are partly motivated by the same problem discussed in this paper. As is shown below, programming similar to the one presented in this paper is indeed possible without self type constructors but requires even more complicated use of advanced language mechanisms, including abstract type members and F-bounded polymorphism (aside from type constructor polymorphism). The main idea of both solutions is to encode self type constructors manually. So, our work supports easier programming to solve the problem. In this section, we compare our solution with those in [24] and [1].

Comparison with Scala. We start with revisiting the definitions of Comparable and Iterator. In the absence of this and exact types, interfaces Comparable<T> and Iterator<T> have different signatures as Figure 6 shows: interface Comparable takes one argument, which is usually the class name that implements Comparable so that the receiver of compareTo() is compared with an object of the same kind; the element type of Iterator<T> is not exact.

Figure 7 shows the solution in Scala\(^6\) that Moors, Piessens, and Odersky gave at the last OOPSLA. They used a highly sophisticated combination of abstract type members [31], higher-order type constructors [24,1], and F-bounded polymorphism [10]. As classes in Figure 4, Iterator has two parameters: Bound and T. The difference is that Bound is a type constructor (<_>_ represents an unused type parameter) so as to be instantiated with Comparable. So, Iterator is higher-order. Another difference is that T is F-bounded—T appears in the upper bound of itself. The keyword type introduces abstract type member Self, which is a type constructor of one argument. Self is used for our This, as the signatures of the methods show. The covariant change is achieved by manually refining or fix the upper bound of Self in every subclass of Iterator: in List, the upper bound of Self<X> is refined to List<X>; in SortedList, Self<X> is fixed to SortedList<X>.

In NumericList, the upper bound of T is refined to type Number. Since Bound in superclass List is a type constructor, we have to adjust the arity of Number when it is given to List by using an anonymous type constructor <_>->Number\(^7\).

Comparison with FGJ\(\omega\). Figure 8 shows the solution in FGJ\(\omega\) [1] by Altherr and Cremet. In this solution, Self is the parameter of each generic class. This approach is based on the simulation of covariant change, described in [37,8,32], by using F-bounded polymorphism [10] and generics. So, as a natural result, this solution has the same disadvantage as the simulation in the following points:

1. type parameterization is much more complex,
2. fixed point classes are necessary for object creations,
3. selftyping is not suitable for recursion.

We elaborate the second and third points below.

It is impossible to create objects from these generic classes since there are no type (constructor)s that conform the upper bounds. So, fixed point classes have to be declared as the generators of objects, as follows:

\begin{verbatim}
class ListFix<Bound<_,_>, T extends Bound<T>>{
    extends List<Bound,T>{}
}
class SortedListFix<Bound<_,_>, T extends Bound<T>>{
    extends SortedList<Bound, T, SortedListFix>{}
}

We adapt Java's notation for familiarity. For example, we use abstract classes for traits.

\footnote{In fact, Scala does not support anonymous type constructors, which can be simulated by abstract type members. On the other hand, FGJ\(\omega\) formalizes them.}
Note that these fixed point classes are not in subtyping relation because they are not in inheritance relation. So, expressing their common interface requires wildcards [38]: List<Comparable,Integer,?>. FGJ does not formalize wildcards. In our proposal, inexact types play the role of such common interfaces.

In class List, this has type List<Bound,T,Self>, but not Self<T>. As a result, the following method cannot be well typed:

```java
class List<Bound<>, T extends Bound<T>, Self<X> extends List<Bound, X, Self>>
{
    void double(){
        append(this); // ill-typed
    }
}
```

The type List<Bound,T,Self> of argument this is not a subtype of Self<T>, the parameter type. In general, this cannot be given binary methods as the arguments in this programming style. There are several solutions to this problem. In Scala, self types can be explicitly annotated [31] by using requires clause. Another solution, invented independently by Saito and Igarashi [32] and Kamina and Tamai [21], is to extend generics a little so that self type constructors can have abstract types in a special case in exchange for a small restriction on subcasing.

7. Conclusion

In this paper, we propose self type constructors, which integrate This and generics so that This is a type constructor in a generic class. Self type constructors can express open recursion of at the level of type constructors. So, a generic class can be safely reused even if it has references to itself recursively but with different type instantiations. We expect that self type constructors can be applied not only to collections but also to programming with comprehensions [2, 22] and parser combinators [25]. We formalize self type constructors as a small calculus FLJ, and prove that the type system is sound with respect to operational semantics.

Main future work is to consider the integration of self type constructors with grouping mechanisms and path types [8, 34, 19], which support extensible yet type-safe mutually recursive classes since mutual recursion cannot be expressed by self type constructors. Such an integration will validate our decision to have thrown away F-bounded polymorphism, which has been used to express mutual recursion. It would not be difficult to show decidability of the present type system but it is left for future work. Other future work includes the development of a type inference algorithm for polymorphic method invocations.

Acknowledgments

We would like to thank members of the Kumiki project for fruitful discussions on this subject.

A. Proof Sketches of Theorems 1 and 2

We sketch the proofs of Theorems 1 and 2. (Theorem 3 is their easy consequence.) The structure of the proof of subject reduction is similar to those for Featherweight Java and Featherweight GJ [17]. So, we first prove various substitution lemmas, which are all proved by induction on the derivations, with other auxiliary lemmas.

**Lemma 1** (Weakening). Suppose $\bullet \vdash \overline{x} : T:: \overline{k}$ and $\Delta \vdash U ok$.

1. If $\Delta \vdash S : T$, then $\overline{x} : T, \Delta \vdash S : T$.
2. If $\Delta \vdash S ok$, then $\overline{x} : T, \Delta \vdash S ok$.
3. If $\Delta ; \Gamma \vdash e : T$, then $\Delta ; \Gamma , x : U and \overline{x} : T, \Delta ; \Gamma \vdash e : T$.

**Proof.** Each is proved by straightforward induction on the derivation of $\Delta \vdash S : T$, $\Delta \vdash S ok$, and $\Delta ; \Gamma \vdash e : T$, respectively.

**Lemma 2.** If $\Delta \vdash G : H and U \downarrow_{x<:T} T^' and U \downarrow_{x<:H} T$, then $\Delta \vdash T^' < T$.

**Proof.** By case analysis on U.

**Lemma 3.** If $\Delta , X <: H \vdash U^' : U and U \downarrow_{x<:H} T and U' \downarrow_{x<:H} T'$, then $\Delta \vdash T^' < T$.

**Proof.** By induction of the derivation of $\Delta , X <: H \vdash U^' : U$.

**Lemma 4.** If $U \downarrow_{x<:H} T$, then $\Delta \vdash [H/X]U <: T$ for some $\Delta$.

**Proof.** By case analysis on $U$. 

---

**Figure 8.** Collections in FGJ.
Lemma 5. If \( \Delta \vdash \Gamma \vdash k \), then \( \Delta \vdash \Gamma \vdash k \) for some \( k' \).

Proof. By induction on the derivation of \( \Delta \vdash \Gamma \vdash k \).

Lemma 6. If \( \Delta \vdash \Gamma \vdash k \), then \( \Gamma \vdash \kappa' \vdash k \) and \( \Gamma \vdash \kappa \vdash \kappa' \vdash \kappa \) for some \( k' \), \( \kappa' \), \( \kappa \), and \( \kappa \).

Proof. By induction on the derivation of \( \Delta \vdash \Gamma \vdash k \).

Lemma 7 (Type Substitution Preserves Typing). If \( \Delta_1 < I \), \( \Delta_2 < S \) and \( \Delta_1 \vdash H : I \) with \( \Delta_1 \vdash H : k \), then \( \Delta_1, [H/X]\Delta_2 \vdash [H/X]S < S < [H/X]T \).

Proof. By induction on the derivation of \( \Delta_1, X < I \), \( \Delta_2 \vdash S < T \).

Lemma 8. If \( \Delta \vdash S < T \) and \( \Delta \vdash S < k \), then \( \Delta \vdash T < k \).

Proof. By induction on the derivation of \( \Delta \vdash S < T \). Note that the last premise about the equality on upper bounds in the rule T-CLASS makes sense in the case of S-CLASS.

Lemma 9. If \( \Delta \vdash \Gamma \vdash * \), then there exist some \( \Xi \) and \( J \) such that \( \Delta \vdash \Xi < J \), \( * \), and \( \Delta \vdash \Pi \vdash [\Pi/\Xi] \) and \( \Delta \vdash \Pi \vdash \Xi \).

Proof. By induction on the derivation of \( \Delta \vdash \Gamma \vdash * \).

Lemma 10 (Type Substitution Preserves Well-Formedness). If \( \Delta_1 < I \), \( \Delta_2 \vdash T \) and \( \Delta_1 \vdash H : I \) with \( \Delta_1 \vdash H : k \), then \( \Delta_1, [H/X]\Delta_2 \vdash [H/X]T \).

Proof. By induction on the derivation of \( \Delta_1, X < I \), \( \Delta_2 \vdash T \) ok using Lemmas 8 and 9.

Lemma 11. If \( \Delta \vdash S \vdash T \) ok and \( \Delta \vdash \Xi \vdash e : \Xi \) for some well-formed bound environment \( \Delta \), then \( \Delta \vdash T \).

Proof. By induction on the derivation of \( \Delta \vdash \Gamma \) ok with a case analysis on the last rule used.

Lemma 12. If \( \Delta \vdash H < I \) and fields(bound_\( \Delta \)(H)) = \( T \vdash \Xi \), then fields(bound_\( \Delta \)(I)) = \( S \vdash \Xi \) and \( S_i = T_i \), and \( g_i = f_i \) for all \( i \leq |T| \).

By straightforward induction on the derivation of \( \Delta \vdash H < I \).

Lemma 13. If \( \Delta \vdash \kappa' \vdash \kappa \) and \( mtype(a, \text{bound}_\Delta(k(\kappa))) = \langle X_o \rangle \vdash T \vdash T_0 \), then \( mtype(a, \text{bound}_\Delta(k'(\kappa', \kappa))) = \langle X_o \rangle \vdash \kappa < T \vdash [\Pi/\Xi] \vdash [\Pi/\Xi]T \vdash (T, \Delta) \) and \( \Delta \vdash [\Pi/\Xi] \vdash [\Pi/\Xi/T]T_0 \).

Proof. By induction on the derivation of \( \Delta \vdash \kappa' \vdash \kappa \).

Lemma 14 (Type Substitution Preserves Typing). If \( \Delta_1, X < I \), \( \Delta_2 \vdash e : \Xi \) and \( \Delta_1 \vdash H : k \) and \( \Delta_1 \vdash H : I \), then \( \Delta_1, [H/X]\Delta_2 \vdash [H/X]e : S \) for some \( S \) such that \( \Delta_1, [H/X]\Delta_2 \vdash S < [H/X]T \).

Proof. By induction on the derivation of \( \Delta_1, X < I \), \( \Delta_2 \vdash \Gamma \vdash e : \Xi \).

Lemma 15 (Term Substitution Preserves Typing). If \( \Delta \vdash \Gamma \vdash [\Xi/\Xi]e : T_0 \) and \( \Delta \vdash \Gamma \vdash e : S \) where \( \Delta \vdash S < T \), then \( \Delta \vdash \Gamma \vdash [\Xi/\Xi]e : S \) for some \( S \) such that \( \Delta \vdash S < T_0 \).

Proof. By induction on the derivation of \( \Delta; \Gamma \vdash [\Xi/\Xi]e : T_0 \).

Thus, \( \Delta; \Gamma \vdash e : T_0 \) finishes the case.

A.1 Proof of Theorem 1

By induction on the derivation of \( e \rightarrow e' \) with a case analysis on the reduction rule used. We show only main cases.

Case R-FIELD: \( e = \text{new } \Pi \vdash \Phi, i, fields(\Pi) = T \vdash \Xi \), \( e' = e_i \).

Case R-INV: \( e = \text{new } \Pi \vdash \Phi, i, e \), \( mtype(\Pi, \Pi) = i, \Pi \vdash x, e_0 \).

By T-INV and T-NEW, we have

\( \Delta; \Gamma \vdash e : T \)

Then, by Lemma 16, there exist \( P \vdash T \) and \( S_0 \) such that

\( \Delta; \Gamma \vdash e \in S_0 \) and \( \Delta; \Gamma \vdash e : T_0 \).

We also have

\( \Delta; \Gamma \vdash e' : S_0' \), \( \Delta; \Gamma \vdash e' : S_0' \) finishes the case.
Case R-EXACT:  
\[ \begin{align*} 
e &= \text{exact}^i \text{new } N[H, I] \text{ as } x, X \text{ in } e_0 \\
H &= i \\
e' &= \text{new } N[H, I] J(\sigma)/x \in [N[H]/X] e_0 \]

By T-EXACT and T-NEW, we have
\[ \Delta \vdash N[H, I] \text{ ok} \]
\[ \Delta; \Gamma \vdash \text{new } N[H, I] J(\sigma)(\xi) : \text{ON}(N[H, I]) \]
\[ \Delta, X < N[H]; \Gamma, x : \text{ON}(H, I) \vdash e_0 : U \quad U \downarrow x < E[S] \]

By Lemmas 14 and 15, \( \Delta; \Gamma \vdash e' : U' \) such that \( \Delta \vdash U' < [N[H]/X] U \). By Lemma 4, \( \Delta \vdash [N[H]/X] U < S \). By S-TRANS, \( \Delta \vdash U' < S \), finishing the case.

A.2 Proof of Theorem 2

By induction on \( e \). We show only main cases.

Case:  
\[ e = e_0 \cdot f_i \]

If \( e_0 \) is not a value, by the induction hypothesis, \( e_0 \rightarrow e_0' \) for some \( e_0' \); then, the congruence rule shows \( e_0 \cdot f_i = e_0' \cdot f_i \).

On the other hand, if \( e_0 \) is a value \( N_0[H] (\xi) \), then, by T-FIELD, it must be the case that \( T \cdot f_i \in \text{fields}(N_0[H]) \). Then, \( e_0 \cdot f_i = v_1 \) by R-FIELD.

Case:  
\[ e = e_0 \cdot \langle G > m(\xi) \]

If \( e_1 \) is not a value, by the induction hypothesis, \( e_1 \rightarrow e_1' \) for some \( e_1' \); then, use the congruence rules to show
\[ e_0 \cdot \langle G > m(\xi) = e_0' \cdot \langle G > m(\xi) \text{ or } e_0 \cdot \langle G > m(...) = e_0' \cdot \langle G > m(...) \text{ or } ... \]

On the other hand, if \( e_0 \) is a value \( N_0[H] (\xi) \), then by T-INV, it must be the case that \( \text{mtype}(m, N_0[H]) = \langle X, \xi, X > S_0 \). By Lemma 16, \( \text{mbody}(m, N_0[H]) = i, X, \xi, \xi' \) where \( |\xi| = |\xi'| \). Then it is easy to show \([N[H]/This] \) is well defined where \( \xi = H, J \) and \( I = i \). Thus, we have
\[ e \rightarrow [\xi, X, \text{new } N_0[H] (\xi)/This] [\xi / \xi, N_0[H]/This] e' \]

finishing the case.

Case:  
\[ e = \text{exact}^i e_0 \text{ as } x, X \text{ in } e_1 \]

If \( e_0 \) is not a value, by the induction hypothesis, \( e_0 \rightarrow e_0' \) for some \( e_0' \); then, the congruence rules shows \( \text{exact}^i e_0 = x, X \text{ in } e_1 \rightarrow \text{exact}^i e_0' = x, X \text{ in } e_1 \).

On the other hand, if \( e_0 \) is a value \( N_0[H] (\xi) \), then by T-EXACT2, it must be the case that \( [N_0[H]/X] e_1 \) is well defined. Thus, we have
\[ e \rightarrow [\text{new } N_0[H, I(\tau)] / X] [N_0[H]/X] e_1 \]

finishing the case.

References


[17] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and


