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 a different element type. 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. Self type constructors are tailored to writing recursive interfaces more concicely than previous solutions. 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

OOPSLA'09, October 25–29, 2009, Orlando, Florida, USA.

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 [15] and also in the expression problem [42].

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 [15, 16, 30, 31, 12, 34] and one without [23, 37, 22, 7, 8, 9, 5] [1]. This paper focuses on the latter, which is admittedly less expressive but simpler and usually expressive enough.

[1] In spite of what the title of the paper [23] suggests, Concord is not really equipped with dependent types in the traditional type-theoretic sense.

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 [37, 8, 9, 5], class hierarchies [23], and arbitrarily nested groups of classes [22].

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 [26, 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 such recursive references types that are covariantly refined along with inheritance.

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[2]. 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 FGJ_{stc} of self type constructors by extending FLJ [7], a core calculus of LOOJ, which in turn extends Featherweight GJ [20], and prove its type soundness.

Actually the problem pointed out here is not new; it has been tackled by other people [27, 1]. Their solutions required a highly sophisticated combination of advanced typing features including abstract type members [39, 19, 33], higher-order type constructors [27, 1], and F-bounded polymorphism [11]. Compared with them, our proposal differs in that (1) there is less boilerplate code to write recursive interfaces because *MyType* automatically supports covariant change of method signatures, (2) F-bounded polymorphism, which is often a source of complication in the semantics and metatheory, is not needed, and (3) no second- or higher-order type constructors are needed. Actually, in order to bring these advantages, we need the help (or, one might say, additional complexity) of our recent proposal to make self types more applicable [36]. Nevertheless, we believe our whole proposal offers a simpler solution to writing extensible, recursive, and generic interfaces than the previous work.

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 and main lemmas are only sketched.

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 FGJ_{stc}, 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 [22, 36], 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:

```
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;
```

[2] Actually, Bruce once pointed out this idea as a solution to extensible parameterized visitors [5]. However, he did not investigate it any further due to its expected complexity.

```
  }
}
```

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:

```
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, 22] 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 same[3]. The following method invocations illustrate type-checking under this rule:

```
@C c1; C c2, c3;
```

[3] Obviously, this rule loses the benefit of dynamic dispatch—the method body called is determined at run time. This problem will be overcome by *local exactization*, which we proposed in the previous work [36]—See Section 3.

```
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.

```
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 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 [27], below.

Figures 1 and 2 show our running example[4] adapted from [27]. 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>. Obviously, this prohibits heterogeneous collections, even for List. Its relaxation, which we do not incorporate in this paper to simplify the discussion, will be sketched in Section 7.) 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().

[4] More 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].

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

Figure 2. Collection classes written in LOOJ.

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 sorted. An invocation of `iterator()` on `List<T>` returns a new iterator object that iterates over the elements of the receiver; that on `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.

Now, consider that we try to write types for methods `map()` and `flatMap()` in class `List<T>`. The requirement is as follows: the method `map()` is a polymorphic method that takes a type `U` and a function[5] 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:

```
class List<T> {
  <U> List<U> map(T->U f){ ... }
}
class SortedList<T extends Comparable>
                               extends List<T> {
  <U extends Comparable> SortedList<U> map(T->U f) {
    ...
  }
}
```

The return types are correctly refined since `SortedList<U>` is a subtype of `List<U>`. However, this code is not very satisfactory. First, it is a programmer's responsibility to refine the return type every time he or she defines a subclass of `List<T>`. Second, these signatures do not really satisfy the requirement above since, for example, when `map()` is invoked on a plain list, we are not sure whether a list or its subclasses will be returned—in other words, the return type is inexact. Note that it is impossible to give `@List<U>` and `@SortedList<U>` as the return types of `map()` to classes `List<T>` and `SortedList<T>`, respectively, since `@SortedList<U>` is not a subtype of `@List<U>`.

[5] In this paper, we assume the existence of first-class functions, which can be seen in Scala [32], and use the notation `S->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> {
  <U> @This<U> create(){ ... }

  @This<T> append(@This<T> that) { ... }

  <U> @This<U> map(T->U f){
    @This<U> newList=this.<U>create();
    for(@T t: this)
      newList.add(f(t));
    return newList;
  }
  <U> @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.

In general, in LOOJ, if a generic class refers to itself recursively but with different type instantiations, it is impossible to give such recursive references types that are automatically refined along with extension.

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, in Section 3.1, we begin with a simple use of self type constructors and rewrite the classes shown in the previous section. 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, 27] (Section 3.3), `exact` statements [36] (Section 3.4), and non-heritable methods [36] (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<U>` 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 [29], 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 `This`: 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 `This` 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

```
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: if `class C<T>[U]` extends `D<T>[U]`, for any types T_1 and T_2, `C<`T_1`>[`T_2`]` is a subtype of `D<`T_1`>[`T_2`]`.

By 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 parameter `Bound` is allowed and `T` has the same bound `Bound` as its superclass does. Note that in class `List<Bound>[T]`, type `@This[String]` will not be well formed since `String` is not a subtype of `Bound`.

```
abstract class Iterable<Bound>[T extends Bound]{
  abstract Iterator<T> iterator();
  abstract void add(@T t);
  abstract @This[T] append(@This[T] that);
}
class List<Bound>[T extends Bound]
                      extends Iterable<Bound>[T]{
  <U extends Bound> @This[U] create(){ ... }
  @This[T] append(@This[T] that){ ... }
  <U extends Bound> @This[U] map(T->U f){ ... }
  <U extends Bound>
  @This[U] flatMap(T->@This[U] f){ ... }
}
class SortedList<Bound extends Comparable>
               [T extends Bound]
                      extends List<Bound>[T]{
  @This[T] append(@This[T] that){ ... }
}
```

Figure 4. Collections with self type constructors, finally.

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 changes. An easy example is as follows:

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

Here, `Object` is assumed to have method `clone()` that returns `@This` to express the intention that it must returns an object of the same type as the receiver. Now, `clone()` is inherited to `List<Bound>[T]`, then the return type changes to `@This[T]`, not `@This`, which is not a proper type. So, if `List` overrides `clone()`, the return type must be written `@This[T]`. Although the return types look syntactically different, their meaning is the same, in that each represents the type of `this` in its declaring 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. Suppose there is a class `C`

```
class C<X1 extends B1>[X2 extends B2] ...
```

and we are going to declare a subclass D with a new fixed parameter. Then, ill-formed types can be avoided if a class definition is of the form

```
class D<Y1 extends B1'>
       [Y2 extends B2', Y3 extends B3']
    extends C<T>[Y3]  ...
```

obeying the following rules:

- upper bounds $\mathtt{B}'_i$ contain only type variables $\mathtt{Y}_1, \ldots \mathtt{Y}_{i-1}$ (for $i = 1, 2, 3$);
- $\mathtt{B}'_3$ is equal to a substitution instance of $\mathtt{B}_2$, in which $\mathtt{X}_1$ and $\mathtt{X}_2$ are replaced by T and $\mathtt{Y}_3$, respectively; and
- type T, which instantiates the refinable parameter $\mathtt{X}_1$ of C, does not include $\mathtt{Y}_3$.

The first rule amounts to the absence of F-bounded polymorphism. Fixed parameter $\mathtt{Y}_3$ corresponds to $\mathtt{X}_2$ in the superclass, so it must instantiate $\mathtt{X}_2$ and have the same bound "modulo instantiation of $\mathtt{X}_1$" (as enforced by the second rule). $\mathtt{Y}_2$ is the newly introduced fixed parameter; since it cannot depend on the existing fixed parameter(s), its declaration is put before them. In the formal calculus given in the next section, the rules above are extended so that they can deal with sequences of type parameter declarations.

3.3 Constructor-Polymorphic Methods

As studied in [8, 37, 22], 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:

```
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 variable*—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:

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

Here, the actual arguments for the method invocation is explicitly specified. Since String implements Comparable, this method invocation is well typed. The development of an algorithm to infer these type arguments is left for future work.

Constructor-polymorphic methods introduce type parameters which range over type constructors. In the formal type system presented in the next section, This will be also regarded 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 constructor parameterized by another type constructor. However, this is not an essential restriction—see Section 5. Our main motivation for the restriction is rather to show interesting programming is possible without higher-order type constructors as used in other work [1, 27].

3.4 exact Statements for Invoking Binary Methods on Inexact Types

The examples so far show invocations on only exact types. Actually, with the help of exact *statements* (which were called *local exactization* [36]), invocations of binary methods[6] on inexact types are possible. Consider the following example:

```
class ReversibleList<Bound>[T extends Bound]
                       extends List<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;
    }
  // flatMap is a binary method!
  @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. This method is constructor-polymorphic (just as isEqualLists) to work for a reversible list or any subclasses (whose definitions are not given here). However, the argument type has to be exact, since binary methods are in-

[6] 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.

voked on the argument in the method body, and it is impossible to apply it directly to strList of inexact type ReversibleList<Comparable>[String].

We use exact statements to make an inexact type temporarily exact in a local scope. This typing feature is considered unpacking of existential types, when an inexact type C is considered an existential type ∃X<:C.@X, where X can be thought of a run-time class [36, 10]. It is also similar to *wildcard capture* [41]. Using an exact statement, isPalindrome() can be applied to strList as

```
boolean b;
exact <X>(@X[String] x = strList) {
  // X extends ReversibleList<Comparable>
  b = <X>isPalindrome(x);
}
```

"<X>" after the keyword exact declares a constructor variable to be used in the type declaration for x and the body (inside the braces). At run time, x is bound to the value of strList and X to the constructor part of the run-time type of strList and the body is executed. At compile time, the type constructor variable X is assumed to extend ReversibleList<Comparable>, which comes from the constructor part of the type of strList and the type of x is an exact type @X[String], so isPalindrome() can be called with x.

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> or SortedList<Bound> but not vice versa. Here, we introduce two alternatives.

One is to use the *abstract factory pattern* [17, 5]. 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,
                       C extends List<Bound>> {
  <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 Iterable<Bound>[T] {
  Factory<Bound,This> factory;

  List(Factory<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 type constructor variable. So, Factory is a higher-order type constructor.

An alternative is to use *nonheritable methods* [36]. 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 written in factory classes to be put into factory methods, as the following code shows:

```
class List<Bound>[T extends Bound]
                    extends Iterable<Bound>[T] {
  nonheritable <U extends Bound> @This[U] create() {
    return new List<Bound>[U]();
  }
}
class SortedList<Bound>[T extends Bound]
                    extends List<Bound>[T] {
  nonheritable <U extends Bound> @This[U] create() {
    return new SortedList<Bound>[U]();
  }
}
```

The modifier nonheritable is essential for each method above. Without it, they would be ill-typed—for example, List<Bound>[U] is not a subtype of @This<U> in class List<Bound>[U]. Note that higher-order type constructors do not appear in this code.

4. FGJ$_{stc}$: A Formal Core Calculus

In this section, we formalize the idea described in the previous section as a small core calculus called FGJ$_{stc}$ based on FLJ [7]. What we model here includes self type constructors, constructor-polymorphic methods, and exact statements, 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 FGJ$_{stc}$ is that classes can be parameterized only by proper types, but cannot by type constructors. So, type constructors in FGJ$_{stc}$ are first-order—type constructors takes only types, but does 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, FGJ$_{stc}$ is not a pure extension of FLJ. The calculus does not include nonheritable methods,

but they are easy to add, as formalized in [36]. 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 $\lhd$ and $\uparrow$ are read `extends` and `return`, respectively. Following the custom of FJ, we put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$;" for "$\mathtt{T}_1\ \mathtt{f}_1; \ldots\ \mathtt{T}_n\ \mathtt{f}_n$;", where n is the length of $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$. Sequences of field declarations, parameter names, and method definitions are assumed to contain no duplicate names. We write the empty sequence as $\bullet$ and concatenation of sequences using a comma. We write $|\cdot|$ for the length of a sequence. As in FJ, 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<`$\overline{\mathtt{H}}$`>`. The application of type constructor `K` to a sequence of $\overline{\mathtt{H}}$ yields `K[`$\overline{\mathtt{H}}$`]`, which can be also a type constructor since partial application of type constructors is allowed in FGJ$_{\mathsf{stc}}$. In what follows, we call `H` an *inexact type* when `H` does not take any more 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 `@` to an inexact type. Since this language is expression-based, the body of a method is a single `return` statement, rather than a compound statement as in the previous section. An expression is either a variable, a field access, a method invocation, an object creation, or an `exact` *expression*, whose body is an expression. We assume that the set of (type) variables includes the special variable `this` (`This`, resp.), which cannot be used as the name of a (type, resp.) parameter to a method.

We use a different syntax for `exact` expressions to simplify the notation. An `exact` statement

```
exact <X>(@X[String] x = strList){ ... }
```

corresponds to `exact`^{1} `strList as x, X in ...` here. In general, we write `exact`^{i} $\mathtt{e}_0$ `as x, X in` $\mathtt{e}_1$, where the variables `x` and `X` are bound in the body expression $\mathtt{e}_1$, and the superscript i represents the arity of `X`.

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`) = `class C ...` for every `C` $\in$ *dom*(*CT*); (2) `Object` $\notin$ *dom*(*CT*); (3) for every class name `C` (except `Object`) appearing anywhere in *CT*, we have `C` $\in$ *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 *fixed* class table *CT* to simplify the notation.

4.2 Lookup functions

We give functions to look up field or method declarations. The function *fields*(`C<`$\overline{\mathtt{G}}$`>[`$\overline{\mathtt{H}}$`]`) returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of field names of class type `C<`$\overline{\mathtt{G}}$`>[`$\overline{\mathtt{H}}$`]` with their types. The function *mtype*(`m`, `C<`$\overline{\mathtt{G}}$`>[`$\overline{\mathtt{H}}$`]`) takes a method name and a class type as input and returns the corresponding method signature of the form `<`$\overline{\mathtt{X}}\lhd\overline{\mathtt{I}}$`>`$\overline{\mathtt{T}}\to\mathtt{T}_0$, in which $\overline{\mathtt{X}}$ are bound in $\overline{\mathtt{T}}$ and $\mathtt{T}_0$. The functions are defined by the rules below, which are essentially the same as those of FGJ.

$$\mathit{fields}(\mathtt{Object<>[]}) = \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{H}}\mathtt{>[}\overline{\mathtt{Y}}\lhd\overline{\mathtt{I}}\mathtt{]}\lhd\mathtt{N[}\overline{\mathtt{Z}}\mathtt{]\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \quad \mathit{fields}([\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}](\mathtt{N[}\overline{\mathtt{Z}}\mathtt{]})) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C<}\overline{\mathtt{F}}\mathtt{>[}\overline{\mathtt{G}}\mathtt{]}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, [\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}} \qquad \text{(F-CLASS)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{H}}\mathtt{>[}\overline{\mathtt{Y}}\lhd\overline{\mathtt{I}}\mathtt{]}\lhd\mathtt{N[}\overline{\mathtt{V}}\mathtt{]\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \quad \mathtt{<}\overline{\mathtt{Z}}\lhd\overline{\mathtt{J}}\mathtt{>U_0\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e_0;\}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{F}}\mathtt{>[}\overline{\mathtt{G}}\mathtt{]}) = [\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}](\mathtt{<}\overline{\mathtt{Z}}\lhd\overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U_0})} \qquad \text{(MT-CLASS)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{H}}\mathtt{>[}\overline{\mathtt{Y}}\lhd\overline{\mathtt{I}}\mathtt{]}\lhd\mathtt{N[}\overline{\mathtt{Z}}\mathtt{]\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \quad \mathtt{m} \notin \overline{\mathtt{M}} \quad \mathit{mtype}([\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}](\mathtt{N[}\overline{\mathtt{Z}}\mathtt{]})) = \mathtt{<}\overline{\mathtt{W}}\lhd\overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U_0}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{F}}\mathtt{>[}\overline{\mathtt{G}}\mathtt{]}) = \mathtt{<}\overline{\mathtt{W}}\lhd\overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U_0}} \qquad \text{(MT-SUPER)}$$

We write $[\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}]$ for the capture-avoiding simultaneous substitution of $\mathtt{F}_1$ for $\mathtt{X}_1$, ..., of $\mathtt{F}_n$ for $\mathtt{X}_n$, of $\mathtt{G}_1$ for $\mathtt{Y}_1$, ..., of $\mathtt{G}_m$ for $\mathtt{Y}_m$. Replacing `X` in `X[`$\overline{\mathtt{I}}$`]` by an application `K[`$\overline{\mathtt{H}}$`]` yields `K[`$\overline{\mathtt{H}}$`,`$\overline{\mathtt{I}}$`]`. In general, we identify `K[`$\overline{\mathtt{H}}$`][`$\overline{\mathtt{I}}$`]` and `K[`$\overline{\mathtt{H}}$`,`$\overline{\mathtt{I}}$`]`. The type substitutions in these rules look a little more complicated since the two kinds of class parameters are separated syntactically. Here, $\mathtt{m} \notin \overline{\mathtt{M}}$ means that the method of name `m` does not exist in $\overline{\mathtt{M}}$.

4.3 Type System

The main judgments of the type system consist of $\Delta \vdash \mathtt{S} <: \mathtt{T}$ for subtyping, $\Delta \vdash \mathtt{I} :: k$ for type constructor well-formedness (k is a kind, defined later), $\Delta \vdash \mathtt{T}$ ok for type well-formedness, $\Delta_1 \vdash \Delta_2 :: \overline{k}$ for bound environment well-formedness, and $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ for expression typing. Here, Δ is a *bound environment*, which is a finite mapping from type (constructor) variables to type constructors, written $\overline{\mathtt{X}} <: \overline{\mathtt{I}}$; Γ is a *type environment*, which is a finite mapping from variables to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$. Following the custom of FJ [20], we abbreviate the sequence of judgments in the obvious way: $\Delta \vdash \mathtt{S}_1 <: \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n <: \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$; $\Delta \vdash \mathtt{I}_1 :: k_1, \ldots, \Delta \vdash \mathtt{I}_n :: k_n$ to $\Delta \vdash \overline{\mathtt{I}} :: \overline{k}$; $\Delta \vdash \mathtt{T}_1$ ok, ..., $\Delta \vdash \mathtt{T}_n$ ok to $\Delta \vdash \overline{\mathtt{T}}$ ok, and $\Delta; \Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Delta; \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

$$
\begin{array}{rcll}
\mathtt{N},\mathtt{P} & ::= & \mathtt{C<}\overline{\mathtt{H}}\mathtt{>} & \text{nonvariable type constructors} \\
\mathtt{K} & ::= & \mathtt{X} \mid \mathtt{N} & \text{type constructors} \\
\mathtt{E},\mathtt{F},\mathtt{G},\mathtt{H},\mathtt{I},\mathtt{J} & ::= & \mathtt{K}[\overline{\mathtt{H}}] & \text{applications of type constructors} \\
\mathtt{S},\mathtt{T},\mathtt{U} & ::= & \mathtt{H} \mid \mathtt{@H} & \text{types} \\
\mathtt{L} & ::= & \mathtt{class\ C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{I}}\mathtt{>[}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{I}}\mathtt{]\lhd N[}\overline{\mathtt{X}}\mathtt{]\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} & \text{class declarations} \\
\mathtt{M} & ::= & \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{I}}\mathtt{>T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow \mathtt{e;\}} & \text{method declarations} \\
\mathtt{d},\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.<}\overline{\mathtt{I}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ N[}\overline{\mathtt{I}}\mathtt{](}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{exact}^{i}\ \mathtt{e\ as\ x,\ X\ in\ e} & \text{expressions} \\
\mathtt{v} & ::= & \mathtt{new\ N[}\overline{\mathtt{I}}\mathtt{](}\overline{\mathtt{v}}\mathtt{)} & \text{values}
\end{array}
$$

Figure 5. FGJ$_{\mathsf{stc}}$: Syntax.

4.3.1 Bounds of Types

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

$$\mathit{bound}_{\Delta_1,\mathtt{X}\mathtt{<:}\mathtt{I},\Delta_2}(\mathtt{X}[\overline{\mathtt{H}}]) = \mathit{bound}_{\Delta_1}(\mathtt{I}[\overline{\mathtt{H}}])$$

$$\mathit{bound}_{\Delta}(\mathtt{N}[\overline{\mathtt{H}}]) = \mathtt{N}[\overline{\mathtt{H}}]$$

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.

4.3.2 Subtyping

The subtyping judgment $\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$, read as "S is a subtype of T under Δ," 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 $\overline{\mathtt{Z}}$ are not given in either side 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 $\overline{\mathtt{Z}}$ (as discussed in Section 3.2 and will be formalized later in T-CLASS). S-APPLY says that if type constructors are in a subtyping relation, so are their applications to the same argument.

$$\Delta \vdash \mathtt{T} \mathtt{<:} \mathtt{T} \qquad \text{(S-REFL)}$$

$$\frac{\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{T} \qquad \Delta \vdash \mathtt{T} \mathtt{<:} \mathtt{U}}{\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{U}} \qquad \text{(S-TRANS)}$$

$$\Delta \vdash \mathtt{X} \mathtt{<:} \Delta(\mathtt{X}) \qquad \text{(S-VAR)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{H}}\mathtt{>[}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{I}}\mathtt{,}\overline{\mathtt{Z}}\mathtt{\lhd}\overline{\mathtt{J}}\mathtt{]\lhd N[}\overline{\mathtt{Z}}\mathtt{]\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{F}}\mathtt{>[}\overline{\mathtt{G}}\mathtt{]} \mathtt{<:} [\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}]\mathtt{N}} \qquad \text{(S-CLASS)}$$

$$\frac{\Delta \vdash \mathtt{G} \mathtt{<:} \mathtt{H}}{\Delta \vdash \mathtt{G[I]} \mathtt{<:} \mathtt{H[I]}} \qquad \text{(S-APPLY)}$$

$$\Delta \vdash \mathtt{@I} \mathtt{<:} \mathtt{I} \qquad \text{(S-EXACT)}$$

4.3.3 Well-formedness

The well-formedness judgments consist of two judgments: one $\Delta \vdash \mathtt{I} :: k$ for type constructors and one $\Delta \vdash \mathtt{T}$ 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:

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

Kind $\mathtt{*}$ is one for inexact types, which take no arguments. Kind $\mathtt{X} \mathtt{<:} \mathtt{I.*}$ is one for type constructors that take a single argument, which must be a subtype of I. We write kind $\overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{I}}\mathtt{.*}$, as an abbreviation of $\mathtt{X}_1 \mathtt{<:} \mathtt{I}_1 \mathtt{.} \ldots \mathtt{.X}_n \mathtt{<:} \mathtt{I}_n\mathtt{.*}$, for n-argument type constructors. Since type constructors are first-order, the upper bounds that appear in a kind always have kind $\mathtt{*}$.

The well-formedness judgment $\Delta \vdash \mathtt{I} :: k$ for type constructors is read as "type constructor I has kind k under Δ." The rules are defined below. `Object<>` has kind $\mathtt{*}$. 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 type that is partially applied, dropping the arguments for $\overline{\mathtt{Z}}$. The rule says that type constructor $\mathtt{C<}\overline{\mathtt{F}}\mathtt{>[}\overline{\mathtt{G}}\mathtt{]}$, which takes as many arguments as the length of $\overline{\mathtt{Z}}$, has a kind if arguments $\overline{\mathtt{F}}$ and $\overline{\mathtt{G}}$ have kind $\mathtt{*}$ and they conform to the corresponding upper bounds. WK-APPLY is the rule for application of a type constructor to a type constructor, which must have kind $\mathtt{*}$ and be a subtype of the upper bound I, written in the kind $\mathtt{X<:I}.k$ of the applied type constructor G.

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

$$\frac{\Delta \vdash \Delta(\mathtt{X}) :: k}{\Delta \vdash \mathtt{X} :: k} \qquad \text{(WK-VAR)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{H}}\mathtt{>[}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{I}}\mathtt{,}\overline{\mathtt{Z}}\mathtt{\lhd}\overline{\mathtt{J}}\mathtt{]\lhd N[}\overline{\mathtt{Z}}\mathtt{]\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \quad \Delta \vdash (\overline{\mathtt{F}},\overline{\mathtt{G}}) :: \overline{*} \qquad \Delta \vdash (\overline{\mathtt{F}},\overline{\mathtt{G}}) \mathtt{<:} [\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}](\overline{\mathtt{H}},\overline{\mathtt{I}})}{\Delta \vdash \mathtt{C<}\overline{\mathtt{F}}\mathtt{>[}\overline{\mathtt{G}}\mathtt{]} :: \overline{\mathtt{Z}} \mathtt{<:} [\overline{\mathtt{F}}/\overline{\mathtt{X}},\overline{\mathtt{G}}/\overline{\mathtt{Y}}]\overline{\mathtt{J}}\mathtt{.*}} \qquad \text{(WK-CLASS)}$$

$$\frac{\Delta \vdash \mathtt{G} :: \mathtt{X} <: \mathtt{I}.k \qquad \Delta \vdash \mathtt{H} :: * \qquad \Delta \vdash \mathtt{H} <: \mathtt{I}}{\Delta \vdash \mathtt{G[H]} :: [\mathtt{H}/\mathtt{X}]k} \quad \text{(WK-APPLY)}$$

The well-formedness judgment $\Delta \vdash \mathtt{T}$ ok for types is read as "type T is well formed under Δ." The rules are defined below. If type constructor I has kind $*$, that is, I is an inexact type, both exact type @I[] and inexact type I[] are well-formed types.

$$\frac{\Delta \vdash \mathtt{I} :: *}{\Delta \vdash \mathtt{@I[]}\ \text{ok}} \qquad \frac{\Delta \vdash \mathtt{I} :: *}{\Delta \vdash \mathtt{I[]}\ \text{ok}}$$

The bound environment well-formedness judgment $\Delta_1 \vdash \Delta_2 :: \overline{k}$, read as "type environment Δ_2 has kinds $\overline{k}$ with respect to Δ_1," is defined as follows:

$$\Delta \vdash \bullet :: \bullet \qquad \frac{\Delta_1 \vdash \Delta_2 :: \overline{k} \qquad \Delta_1, \Delta_2 \vdash \mathtt{I} :: k'}{\Delta_1 \vdash (\Delta_2, \mathtt{X} <: \mathtt{I}) :: (\overline{k}, k')}$$

Note that the rules are defined in such a way that F-bounded polymorphism [11] is kicked out from FGJ_{stc}. 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

Expression Typing. The typing judgment for expressions of the form $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$, read as "under bound environment Δ and type environment Γ, expression e has type T," defined as follows:

$$\Delta; \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \quad \text{(T-VAR)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{@H}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \mathtt{H}_0 = \mathtt{I}_0\mathtt{[}\overline{\mathtt{H}}\mathtt{]} \qquad \Delta \vdash [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_i\ \text{ok}\end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_i} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{@H}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{H}_0)) = \mathtt{<}\overline{\mathtt{X}} \mathtt{\lhd} \overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{T}} \mathtt{\to} \mathtt{T}_0 \\ \mathtt{H}_0 = \mathtt{I}_0\mathtt{[}\overline{\mathtt{H}}\mathtt{]} \qquad \Delta \vdash [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_0\ \text{ok} \\ \Delta \vdash \overline{\mathtt{G}} :: \overline{k} \qquad \Delta \vdash \overline{\mathtt{G}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{I}_0/\mathtt{This}]\overline{\mathtt{J}} \\ \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{I}_0/\mathtt{This}]\overline{\mathtt{T}}\end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{<}\overline{\mathtt{G}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{I}_0/\mathtt{This}]\mathtt{T}_0} \quad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{N}\mathtt{[}\overline{\mathtt{I}}\mathtt{]}\ \text{ok} \qquad \mathit{fields}(\mathtt{N}\mathtt{[}\overline{\mathtt{I}}\mathtt{]}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \\ \text{for all } i \le |\overline{\mathtt{T}}|.(\Delta \vdash [\mathtt{N}\mathtt{[}\overline{\mathtt{G}}\mathtt{]}/\mathtt{This}]\mathtt{T}_i\ \text{ok and} \\ \Delta \vdash \mathtt{U}_i <: [\mathtt{N}\mathtt{[}\overline{\mathtt{G}}\mathtt{]}/\mathtt{This}]\mathtt{T}_i \text{ where } \overline{\mathtt{I}} = \overline{\mathtt{G}}, \overline{\mathtt{H}} \text{ for some } \overline{\mathtt{G}}, \overline{\mathtt{H}})\end{array}}{\Delta; \Gamma \vdash \mathtt{new\ N}\mathtt{[}\overline{\mathtt{I}}\mathtt{]}\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{@N}\mathtt{[}\overline{\mathtt{I}}\mathtt{]}} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_1 : \mathtt{I}\mathtt{[}\overline{\mathtt{H}}\mathtt{]} \qquad |\overline{\mathtt{H}}| = i \\ \Delta, \mathtt{X} <: \mathtt{I}; \Gamma, \mathtt{x} : \mathtt{@X}\mathtt{[}\overline{\mathtt{H}}\mathtt{]} \vdash \mathtt{e}_0 : \mathtt{U}_0 \\ \Delta, \mathtt{X} <: \mathtt{I} \vdash \mathtt{U}_0 <: \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0\ \text{ok}\end{array}}{\Delta; \Gamma \vdash \mathtt{exact}^i\ \mathtt{e}_1\ \mathtt{as\ x,\ X\ in\ e}_0 : \mathtt{T}_0} \quad \text{(T-EXACT1)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_1 : \mathtt{@I}\mathtt{[}\overline{\mathtt{H}}\mathtt{]} \qquad |\overline{\mathtt{H}}| = i \\ \Delta; \Gamma, \mathtt{x} : \mathtt{@I}\mathtt{[}\overline{\mathtt{H}}\mathtt{]} \vdash [\mathtt{I}/\mathtt{X}]\mathtt{e}_0 : \mathtt{U}_0 \qquad \Delta \vdash \mathtt{U}_0\ \text{ok}\end{array}}{\Delta; \Gamma \vdash \mathtt{exact}^i\ \mathtt{e}_1\ \mathtt{as\ x,\ X\ in\ e}_0 : \mathtt{U}_0} \quad \text{(T-EXACT2)}$$

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 make the types of the receivers exact.) The rule T-FIELD means that the type of field access is obtained by looking up field declarations from the bound of $\mathtt{H}_0$ and then substituting $\mathtt{I}_0$ for This in the type $\mathtt{T}_i$ corresponding to $\mathtt{f}_i$. Note that $\mathtt{I}_0$ is obtained by dropping some $\overline{\mathtt{H}}$ from $\mathtt{H}_0$ so that the type constructors This in the class where $\mathtt{T}_i$ is declared and $\mathtt{I}_0$ have the same arity. The selection of $\mathtt{I}_0$ is correct if substitution of $\mathtt{I}_0$ for This in $\mathtt{T}_i$ yields a well-formed type.

Assume that $\Delta; \Gamma \vdash \mathtt{x} : \mathtt{@C[T]}$ and $\Delta; \Gamma \vdash \mathtt{y} : \mathtt{@D[U,S]}$ for some Δ and Γ with the following class declarations:

```
class C[X]{ @This[X] f; }
class D[Y,X] extends C[X]{ @This[Y,X] g; }
```

Then, examples of typing field accesses are shown:

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{x} : \mathtt{@C[T]} \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{C[T]})) = \mathtt{@This[T]\ f} \\ \Delta; \Gamma \vdash [\mathtt{C}/\mathtt{This}]\mathtt{@This[T]}\ \text{ok}\end{array}}{\Delta; \Gamma \vdash \mathtt{x.f} : [\mathtt{C}/\mathtt{This}]\mathtt{@This[T]}(= \mathtt{@C[T]})}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{y} : \mathtt{@D[U,S]} \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{D[U,S]})) = \mathtt{@This[S]\ f, @This[U,S]\ g} \\ \Delta; \Gamma \vdash [\mathtt{D[U]}/\mathtt{This}]\mathtt{@This[S]}\ \text{ok}\end{array}}{\Delta; \Gamma \vdash \mathtt{y.f} : [\mathtt{D[U]}/\mathtt{This}]\mathtt{@This[S]}(= \mathtt{@D[U,S]})}$$

In the first derivation C is substituted for This in the result type while in the second D[U] is for This.

The rule T-INVK is similar to T-FIELD. First, the method signature is retrieved from the receiver's type by using *mtype*. Then, $\mathtt{I}_0$ is selected so that $\mathtt{I}_0$ and This in the nearest superclass where method m is declared have the same arity. Finally, it is checked if the type arguments $\overline{\mathtt{G}}$ have some kinds and are subtypes of the corresponding formal $\overline{\mathtt{J}}$, and if the types $\overline{\mathtt{U}}$ of the arguments $\overline{\mathtt{e}}$ are subtypes of those of the corresponding formal $\overline{\mathtt{T}}$. Since $\overline{\mathtt{T}}$ and $\overline{\mathtt{J}}$ may contain This and $\overline{\mathtt{X}}$, 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 $\mathtt{N}[\overline{\mathtt{G}}]$ for each field. Under the assumption of the environments and class declarations above, an example of typing a new expression is shown:

$$\frac{\begin{array}{c}\Delta \vdash \texttt{D[U,S]}\ \text{ok} \\ \mathit{fields}(\mathit{bound}_\Delta(\texttt{D[U,S]})) = \texttt{@This[S] f, @This[U,S] g} \\ \Delta;\Gamma \vdash [\texttt{D[U]}/\texttt{This}]\texttt{@This[S]} (= \texttt{@D[U,S]})\ \text{ok} \\ \Delta \vdash \texttt{@D[U,S]} <: \texttt{@D[U,S]} \\ \Delta;\Gamma \vdash [\texttt{D}/\texttt{This}]\texttt{@This[U,S]} (= \texttt{@D[U,S]})\ \text{ok} \\ \Delta \vdash \texttt{@D[U,S]} <: \texttt{@D[U,S]}\end{array}}{\Delta;\Gamma \vdash \texttt{new D[U,S](y,y)} : \texttt{@D[U,S]}}$$

`D[U]` is substituted for `This` in the type of `f` while `D` is for `This` in the type of `g`.

There are two rules for `exact` expressions, depending on whether the type of $\texttt{e}_1$ is inexact $\texttt{I}[\overline{\texttt{H}}]$ or exact $\texttt{@I}[\overline{\texttt{H}}]$. If inexact (T-EXACT1), the body expression is typed under Δ extended by $\texttt{X} <: \texttt{I}$ and Γ extended by $\texttt{x}:\texttt{@X}[\overline{\texttt{H}}]$. The length of $\overline{\texttt{H}}$ is determined by superscript i. Since the result type $\texttt{U}_0$ may contain the type constructor variable `X`, the type of the whole expression is obtained by taking a supertype $\texttt{T}_0$ of $\texttt{U}_0$ so that $\texttt{T}_0$ does not contain `X`. The rule T-EXACT2 is for the other case. This rule, which will not be used in ordinary programs, 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 `exact` 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.

$$\frac{\Delta;\Gamma \vdash \texttt{e}_0 : \texttt{I}_0[\overline{\texttt{H}}] \qquad \mathit{fields}(\mathit{bound}_\Delta(\texttt{I}_0[\overline{\texttt{H}}])) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \Delta, \texttt{This} <: \texttt{I}_0 \vdash \texttt{T}_i <: \texttt{T} \qquad \Delta \vdash \texttt{T}\ \text{ok}}{\Delta;\Gamma \vdash \texttt{e}_0.\texttt{f}_i : \texttt{T}}$$

(T-FIELD')

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \texttt{e}_0 : \texttt{H}_0 \qquad \mathit{mtype}(\texttt{m}, \mathit{bound}_\Delta(\texttt{H}_0)) = \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{J}}\texttt{>}\overline{\texttt{T}}\texttt{→}\texttt{T}_0 \\ \overline{\texttt{T}} \text{ and } \overline{\texttt{J}} \text{ do not contain } \texttt{This} \qquad \Delta \vdash \overline{\texttt{G}} :: \overline{k} \\ \Delta \vdash \overline{\texttt{G}} <: [\overline{\texttt{G}}/\overline{\texttt{X}}]\overline{\texttt{J}} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \qquad \Delta \vdash \overline{\texttt{U}} <: [\overline{\texttt{G}}/\overline{\texttt{X}}]\overline{\texttt{T}} \\ \texttt{H}_0 = \texttt{I}_0[\overline{\texttt{H}}] \qquad \Delta, \texttt{This} <: \texttt{I}_0 \vdash \texttt{T}_0 <: \texttt{T} \qquad \Delta \vdash \texttt{T}\ \text{ok}\end{array}}{\Delta;\Gamma \vdash \texttt{e}_0.\texttt{<}\overline{\texttt{G}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} : [\overline{\texttt{G}}/\overline{\texttt{X}}]\texttt{T}}$$

(T-INVK')

The condition that $\overline{\texttt{T}}$ and $\overline{\texttt{J}}$ do not contain `This` means that the method invoked is not binary—binary methods should be invoked on receivers of exact types.

Closing of Types. In the rule T-EXACT1, $\texttt{T}_0$ is not unique for given `X`, `I`, $\texttt{U}_0$ and Δ. We could give a set of rules to determine a minimal `X`-free supertype of $\texttt{U}_0$ as a basis for implementing typecheckers. We introduce the judgment *closing of types* [21], written $\texttt{S} \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}$, meaning that "`T` is a minimal supertype of `S` without `X` under Δ". We also say that "`S` closes to `T` under $\texttt{X} <: \texttt{I}$ and Δ". In T-EXACT1, $\texttt{U}_0 \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}_0$ can be used for $\Delta, \texttt{X} <: \texttt{I} \vdash \texttt{U}_0 <: \texttt{T}_0$. Similarly, $\texttt{T}_i \Downarrow^{\Delta}_{\texttt{This}<:\texttt{I}_0} \texttt{T}$ for $\Delta, \texttt{This} <: \texttt{I}_0 \vdash \texttt{T}_i <: \texttt{T}$ in T-FIELD' and $\texttt{T}_0 \Downarrow^{\Delta}_{\texttt{This}<:\texttt{I}_0} \texttt{T}$ for $\Delta, \texttt{This} <: \texttt{I}_0 \vdash \texttt{T}_0 <: \texttt{T}$ in T-INVK'. The rules of closing of types are as follows:

$$\frac{\texttt{X} \notin \mathit{fv}(\texttt{T})}{\texttt{T} \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}} \qquad \frac{\begin{array}{c}\texttt{X} \in \mathit{fv}(\texttt{H}) \\ \texttt{H} \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}\end{array}}{\texttt{@H} \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}} \qquad \frac{\texttt{I}[\overline{\texttt{H}}] \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}}{\texttt{X}[\overline{\texttt{H}}] \Downarrow^{\Delta}_{\texttt{X}<:\texttt{I}} \texttt{T}}$$

$$\frac{\begin{array}{c}\texttt{X} \neq \texttt{Y} \\ \texttt{Y} \in \mathit{fv}(\overline{\texttt{H}}) \\ \Delta(\texttt{X})[\overline{\texttt{H}}] \Downarrow^{\Delta}_{\texttt{Y}<:\texttt{I}} \texttt{T}\end{array}}{\texttt{X}[\overline{\texttt{H}}] \Downarrow^{\Delta}_{\texttt{Y}<:\texttt{I}} \texttt{T}} \qquad \frac{\begin{array}{c}\texttt{X}_0 \in \mathit{fv}(\overline{\texttt{F}}, \overline{\texttt{G}}) \\ \texttt{class C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{]◁N[}\overline{\texttt{Z}}\texttt{]\{...\}} \\ [\overline{\texttt{F}}/\overline{\texttt{X}}, \overline{\texttt{G}}/\overline{\texttt{Y}}]\texttt{N[}\overline{\texttt{Z}}\texttt{]} \Downarrow^{\Delta}_{\texttt{X}_0<:\texttt{I}_0} \texttt{T}\end{array}}{\texttt{C<}\overline{\texttt{F}}\texttt{>[}\overline{\texttt{G}}\texttt{]} \Downarrow^{\Delta}_{\texttt{X}_0<:\texttt{I}_0} \texttt{T}}$$

Here, $\mathit{fv}(\texttt{T})$ returns the set of type variables that appear in `T`. There are three rules in the upper row. The left rule says that if type `T` does not contain the variable `X`, the result is the same `T`. The center rule says that if an exact type contains `X`, the result is what its inexact version closes to. The right rule says that if `X` appears at the head of the type to close, then the result is what $\texttt{I}[\overline{\texttt{H}}]$, the bound of $\texttt{X}[\overline{\texttt{H}}]$, closes to. There are two rules in the lower row. The left rule says that if the arguments for `X` contain `Y`, then the result is what $\Delta(\texttt{X})[\overline{\texttt{H}}]$, the bound of $\texttt{X}[\overline{\texttt{H}}]$, closes to. The right rule says that if the arguments of a class type contain $\texttt{X}_0$, the result is what the supertype of the class type closes to. Note that every well-formed type always closes to a certain type, at worst to `Object<>[]`, which is no informative type.

Method Typing. The typing judgment for method declarations is written $\texttt{C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{]} \vdash \texttt{M}\ \text{ok}$. The rule T-METHOD, defined below, is straightforward. The method body $\texttt{e}_0$ is typed under the bound environment derived from the parameterization clauses in the class and method declaration as well as $\texttt{This} <: \texttt{C<}\overline{\texttt{X}}\texttt{>}$, and the type environment, in which `this` has type $\texttt{@This[}\overline{\texttt{Y}}\texttt{]}$. The last premise, using the predicate *override*, checks valid overriding of method signatures—$\mathit{override}(\texttt{m}, \texttt{C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{]}, \texttt{<}\overline{\texttt{Z}}\texttt{◁}\overline{\texttt{J}}\texttt{>}\overline{\texttt{T}}\texttt{→}\texttt{T}_0)$ means that class $\texttt{C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{]}$ correctly overrides the method of name `m` in its superclass (if exists) and the overriding signature is $\texttt{<}\overline{\texttt{Z}}\texttt{◁}\overline{\texttt{J}}\texttt{>}\overline{\texttt{T}}\texttt{→}\texttt{T}_0$. The rule is also below. Note that method signatures are α-convertible and return types can be covariantly refined along with extension.

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{,}\overline{\texttt{Z}}\texttt{◁}\overline{\texttt{J}}\texttt{]◁N[}\overline{\texttt{Z}}\texttt{]\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} \\ [\texttt{This[}\overline{\texttt{Y}}\texttt{]}/\texttt{This}](\mathit{mtype}(\texttt{m}, \texttt{N[}\overline{\texttt{Z}}\texttt{]})) = \texttt{<}\overline{\texttt{V}}\texttt{◁}\overline{\texttt{F}}\texttt{>}\overline{\texttt{U}}\texttt{→}\texttt{U}_0 \\ \text{implies } \overline{\texttt{G}}, \overline{\texttt{T}} = [\overline{\texttt{W}}/\overline{\texttt{V}}](\overline{\texttt{F}}, \overline{\texttt{U}}) \text{ and } \overline{\texttt{W}} <: \overline{\texttt{G}} \vdash \texttt{T}_0 <: [\overline{\texttt{W}}/\overline{\texttt{V}}]\texttt{U}_0\end{array}}{\mathit{override}(\texttt{m}, \texttt{C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{,}\overline{\texttt{Z}}\texttt{◁}\overline{\texttt{J}}\texttt{]}, \texttt{<}\overline{\texttt{W}}\texttt{◁}\overline{\texttt{G}}\texttt{>}\overline{\texttt{T}}\texttt{→}\texttt{T}_0)}$$

(OVERRIDE)

$$\frac{\begin{array}{c}\Delta_1 = \overline{\texttt{X}} <: \overline{\texttt{H}}, \overline{\texttt{Y}} <: \overline{\texttt{I}}, \texttt{This} <: \texttt{C<}\overline{\texttt{X}}\texttt{>} \qquad \Delta_2 = \overline{\texttt{Z}} <: \overline{\texttt{J}} \\ \Delta_1 \vdash \Delta_2 :: \overline{k} \qquad \Delta = \Delta_1, \Delta_2 \qquad \Delta \vdash \overline{\texttt{T}}, \texttt{T}_0\ \text{ok} \\ \Gamma = \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{@This[}\overline{\texttt{Y}}\texttt{]} \qquad \Delta;\Gamma \vdash \texttt{e}_0 : \texttt{U}_0 \\ \Delta \vdash \texttt{U}_0 <: \texttt{T}_0 \qquad \mathit{override}(\texttt{m}, \texttt{C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{]}, \texttt{<}\overline{\texttt{Z}}\texttt{◁}\overline{\texttt{J}}\texttt{>}\overline{\texttt{T}}\texttt{→}\texttt{T}_0)\end{array}}{\texttt{C<}\overline{\texttt{X}}\texttt{◁}\overline{\texttt{H}}\texttt{>[}\overline{\texttt{Y}}\texttt{◁}\overline{\texttt{I}}\texttt{]} \vdash \texttt{<}\overline{\texttt{Z}}\texttt{◁}\overline{\texttt{J}}\texttt{>T}_0\ \texttt{m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{}\uparrow \texttt{e}_0\texttt{;\}}\ \text{ok}}$$

(T-METHOD)

Class Typing. The typing judgment for classes is written $\vdash \texttt{L}\ \text{ok}$. The rule is defined below. The type environment derived from the parameterization clause is checked if each upper bound has kind $*$. If it succeeds, it will be guaranteed

that the scope of a type variable is the following bindings. So, the upper bounds $\overline{\mathtt{H}}$ contain none of $\overline{\mathtt{Y}}$ and $\overline{\mathtt{Z}}$, and $\overline{\mathtt{I}}$ none of $\overline{\mathtt{Z}}$, as expected. As mentioned before, $\overline{\mathtt{N}}$ does not contain any of $\overline{\mathtt{Z}}$. These conditions correspond to the general pattern of class parameterization discussed at the end of Section 3.2. The well-formedness of field types $\overline{\mathtt{T}}$ is checked under the bound environment with `This <: C<`$\overline{\mathtt{X}}$`>`. The last conditional premise says that the upper bounds of type variables $\overline{\mathtt{Z}}$ must be the same as those of the superclass.

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{H}}, \overline{\mathtt{Y}} <: \overline{\mathtt{I}}, \overline{\mathtt{Z}} <: \overline{\mathtt{J}} \qquad \bullet \vdash \Delta :: \overline{*} \\ \overline{\mathtt{X}}<:\overline{\mathtt{H}}, \overline{\mathtt{Y}}<:\overline{\mathtt{I}} \vdash \mathtt{N} : \overline{\mathtt{Z}}<:\overline{\mathtt{J}}.* \qquad \Delta, \mathtt{This} <: \mathtt{C}\langle\overline{\mathtt{X}}\rangle \vdash \overline{\mathtt{T}}\ \mathrm{ok} \\ \mathtt{C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle[\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{I}}, \overline{\mathtt{Z}}\triangleleft\overline{\mathtt{J}}] \vdash \overline{\mathtt{M}}\ \mathrm{ok} \\ \text{if } \mathtt{D} \neq \mathtt{Object} \text{ where } \mathtt{N} = \mathtt{D}\langle\overline{\mathtt{E}}\rangle \text{ and} \\ \mathtt{class\ D}\langle\overline{\mathtt{W}}\triangleleft\overline{\mathtt{F}}\rangle[\overline{\mathtt{V}}\triangleleft\overline{\mathtt{G}}]\triangleleft \ldots\{\ldots\}, \\ \text{then } [\overline{\mathtt{E}}/\overline{\mathtt{W}}][\overline{\mathtt{Z}}/\overline{\mathtt{V}}]\overline{\mathtt{G}} = \overline{\mathtt{J}}\end{array}}{\vdash \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle[\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{I}}, \overline{\mathtt{Z}}\triangleleft\overline{\mathtt{J}}]\triangleleft\mathtt{N}[\overline{\mathtt{Z}}]\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathrm{ok}} \quad \text{(T-CLASS)}$$

A class table (*CT*) is ok, if all the class definitions in it are ok.

4.4 Operational Semantics

The operational semantics is given by the reduction relation of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression e reduces to $\mathtt{e}'$ in one step." We require another lookup function $\mathit{mbody}(\mathtt{m}, \mathtt{C}\langle\overline{\mathtt{G}}\rangle[\overline{\mathtt{H}}])$, defined below, for a pair of natural number and method body with formal (type) parameters, written i, $\overline{\mathtt{X}}.\overline{\mathtt{x}}.\mathtt{e}$, of given method and class names. $\overline{\mathtt{X}}$ and $\overline{\mathtt{x}}$ 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.

$$\frac{\begin{array}{c}\mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle[\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{I}}]\triangleleft\mathtt{N}[\overline{\mathtt{W}}]\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \\ \langle\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{J}}\rangle\mathtt{U}_0\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\uparrow \mathtt{e}_0;\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}\langle\overline{\mathtt{F}}\rangle[\overline{\mathtt{G}}]) = 0, [\overline{\mathtt{F}}/\overline{\mathtt{X}}, \overline{\mathtt{G}}/\overline{\mathtt{Y}}](\overline{\mathtt{Z}}.\overline{\mathtt{x}}.\mathtt{e}_0)} \quad \text{(MB-CLASS)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle[\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{I}}, \overline{\mathtt{Z}}\triangleleft\overline{\mathtt{J}}]\triangleleft\mathtt{N}[\overline{\mathtt{Z}}]\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \\ \mathit{mbody}([\overline{\mathtt{E}}/\overline{\mathtt{X}}, \overline{\mathtt{F}}/\overline{\mathtt{Y}}, \overline{\mathtt{G}}/\overline{\mathtt{Z}}](\mathtt{N}[\overline{\mathtt{Z}}])) = i, \overline{\mathtt{W}}.\overline{\mathtt{x}}.\mathtt{e}_0\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}\langle\overline{\mathtt{E}}\rangle[\overline{\mathtt{F}}, \overline{\mathtt{G}}]) = (i + |\overline{\mathtt{F}}|), \overline{\mathtt{W}}.\overline{\mathtt{x}}.\mathtt{e}_0} \quad \text{(MB-SUPER)}$$

The reduction rules are given below. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1$, ..., $\mathtt{x}_n$ with $\mathtt{d}_n$, and y with e. This simultaneous substitution is capture-avoiding. The first three rules are one for field access, one for method invocation, which are straightforward, thanks to lookup functions, and one for `exact` expressions. The only non-trivial point is that, in R-INVK, `This` is replaced by $\mathtt{N}[\overline{\mathtt{H}}]$, which is obtained by dropping $\overline{\mathtt{I}}$ from the receiver type. Since the length of $\overline{\mathtt{H}}$ is the same as the number of fixed parameters introduced between the receiver class and the class where the method is defined, $\mathtt{N}[\overline{\mathtt{H}}]$ is a subtype of the constructor part of the class where the method is defined.

$$\frac{\mathit{fields}(\mathtt{N}[\overline{\mathtt{H}}]) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ N}[\overline{\mathtt{H}}](\overline{\mathtt{e}}).\mathtt{f}_i \longrightarrow \mathtt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]) = i, \overline{\mathtt{X}}.\overline{\mathtt{x}}.\mathtt{e}_0 \qquad |\overline{\mathtt{H}}| = i}{\begin{array}{l}\mathtt{new\ N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}}).\mathtt{m}\langle\overline{\mathtt{G}}\rangle(\overline{\mathtt{d}}) \longrightarrow \\ [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}})/\mathtt{this}][\overline{\mathtt{G}}/\overline{\mathtt{X}}, \mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{e}_0\end{array}} \quad \text{(R-INVK)}$$

$$\frac{|\overline{\mathtt{I}}| = i}{\begin{array}{l}\mathtt{exact}^i\ \mathtt{new\ N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}})\ \mathtt{as\ x,\ X\ in\ e}_0 \\ \longrightarrow [\mathtt{new\ N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}})/\mathtt{x}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{X}]\mathtt{e}_0\end{array}} \quad \text{(R-EXACT)}$$

The reduction rules may be applied at any point in an expression, so we also need the congruence rules.

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0'.\mathtt{f}} \quad \text{(RC-FIELD)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{m}\langle\overline{\mathtt{H}}\rangle(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0'.\mathtt{m}\langle\overline{\mathtt{H}}\rangle(\overline{\mathtt{e}})} \quad \text{(RC-INVK-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0.\mathtt{m}\langle\overline{\mathtt{H}}\rangle(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}\langle\overline{\mathtt{H}}\rangle(\ldots,\mathtt{e}_i',\ldots)} \quad \text{(RC-INVK-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ N}[\overline{\mathtt{H}}](\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new\ N}[\overline{\mathtt{H}}](\ldots,\mathtt{e}_i',\ldots)} \quad \text{(RC-NEW)}$$

$$\frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_1'}{\begin{array}{l}\mathtt{exact}^i\ \mathtt{e}_1\ \mathtt{as\ x,\ X\ in\ e}_0 \\ \longrightarrow \mathtt{exact}^i\ \mathtt{e}_1'\ \mathtt{as\ x,\ X\ in\ e}_0\end{array}} \quad \text{(RC-EXACT)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\begin{array}{l}\mathtt{exact}^i\ \mathtt{e}_1\ \mathtt{as\ x,\ X\ in\ e}_0 \\ \longrightarrow \mathtt{exact}^i\ \mathtt{e}_1\ \mathtt{as\ x,\ X\ in\ e}_0'\end{array}} \quad \text{(RC-EXACT-BODY)}$$

We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

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 [43, 20]. Here, we show only the results. See Appendix A for the required lemmas and proof sketches of the theorems.

THEOREM 1 (Subject Reduction). *If* $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\Delta; \Gamma \vdash \mathtt{e}' : \mathtt{T}'$*, for some* $\mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$.

Proof. By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with case analysis on the reduction rule used. □

THEOREM 2 (Progress). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* e *is not a value, then* $\mathtt{e} \longrightarrow \mathtt{e}'$*, for some* $\mathtt{e}'$.

Proof. By induction on the derivation of $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ with case analysis on the last rule used. □

THEOREM 3 (Type Soundness). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* v *with* $\emptyset;\emptyset \vdash \mathtt{v} : \mathtt{T}'$ *and* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$.

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 [27] and formalized as FGJω [1], 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:

```
class List<T> { ... }
class List2<T> extends List<T> { ... }
class C<L<X> extends List<X>> { .. L<String> f; .. }
```

Class `C` is parameterized by a type constructor variable `L`, where `L` must be instantiated by a subtype of `List`, meaning that for any `X`, `L<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 [1], polymorphic embedding of DSLs [18], and modular visitor components [13]. 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 FGJω. 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:

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

FGJ_{stc} in Section 4 does not need such machinery since the order of type constructors is restricted.

Definition-Site Variance. *Definition-site variance* [14], whose theory is based on *polarity* [38] in $F_{\omega}^{<:}$, is a typing feature that can relax invariance on parameters in a generic class with respect to subtyping. For example, if we write `class List<+T> { }`, `List` is covariant with respect to `T`—`List<S> <: List<U>` holds if `S <: U`. In exchange for flexible subtyping, certain restrictions are posed on where the type parameter `T` can appear. For example, in `List<+T>`, neither `T` nor `List<T>` can be used as parameter types of a method.

Definition-site variance can also be easily adapted 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 Emir et al. [14] so that it uses self type constructors:

```
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 [28] can be similarly adapted.

Wildcard Types. Wildcard types [41], derived from variant parametric types [21], are introduced to Java to relax invariance on parameters in a generic class as well as definition-site 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[7]:

```
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>[Float]`, `List<Comparable>[Integer]`, and so on.

The introduction of wildcards to FGJ_{stc} could give more informative types for expressions. The following field access typed by T-FIELD' in FGJ_{stc} illustrates this:

```
class List[T] {}
class C{ List[This] f; }
C c;
c.f; // : Object<>[] in STC
```

[7] 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>`

The type of `c.f` is `Object<>[]`, which is the only supertype of `List[This]`. In the presence of wildcards, `List[This]` could have a less trivial supertype `List[? extends C]`, which at least means that the result value is a `List`.

6. Related Work

Much recently, Altherr and Cremet [1] and Moors, Piessens, and Odersky [27] 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. However, in these solutions, writing recursive interfaces requires more boilerplate code due to complicated use of advanced language mechanisms, including abstract type members and F-bounded polymorphism (aside from type constructor polymorphism) to encode self type constructors *manually*. In this section, we compare our solution with those in [27] 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[8] that Moors, Piessens, and Odersky gave at the last OOPSLA. They used a highly sophisticated combination of abstract type members [34], higher-order type constructors [27, 1], and F-bounded polymorphism [11]. As classes in Figure 4, `Iterable` has two parameters: `Bound` and `T`. The difference is that `Bound` is a type constructor parameter (`<_>` represents an unused type parameter) so as to be instantiated with `Comparable`. So, `Iterable` 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

```
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>>
                     extends Iterable<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>>
                     extends List<Comparable, T>{
  type Self<X extends Comparable<X>>
       = SortedList<X>;
  <U extends Comparable<U>> Self<U> map(T->U f);
}
class NumericList<T extends Number>
                     extends List<<_>->Number,T>{
  type Self<T extends Number> = NumericList<T>;
  <U extends Number> Self<U> map(T->U f);
}
```

Figure 7. Collections in Scala

upper bound of `Self` in every subclass of `Iterable`: 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`[9].

Scalina [29] is an object-oriented calculus that provides the formal underpinning for the implementation of higher-order type constructors in Scala. Like refinable/fixed parameters, this calculus distinguishes abstract type members into two: members and un-members to prevent ill-formed type from appearing. While members are similar to refinable parameters in that both are covariant, un-members are a little different from fixed parameters—un-members are contravariant but fixed parameters are invariant. If fixed parameters were contravariant, type soundness of FGJ_{stc} would be lost since superclass types can be ill formed.

Comparison with FGJω. Figure 8 shows the solution in FGJω [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 [40, 8, 35], by using F-bounded polymorphism [11] and generics. So, as a natural result, this solution has the same disadvantages as the simulation in the following points:

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

[9] In fact, Scala does not support anonymous type constructors, which can be simulated by abstract type members. On the other hand, FGJω has them.

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

Figure 8. Collections in FGJω

1. type parameterization is much more complex,
2. fixed point classes are necessary for object creations, and
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 constructors that conform to the upper bounds. So, fixed point classes have to be declared as the generators of objects, as follows:

```
class ListFix<Bound<_>, T extends Bound<T>>
      extends List<Bound, T, ListFix> {}
class SortedListFix<Bound<_>, T extends Bound<T>>
      extends SortedList<Bound, T, SortedListFix> {}
```

Note that these fixed point classes are not in subtyping relation because they are not in the inheritance relation. Wildcards [41], which FGJω does not have, can be used to express common interfaces of their instances. For example, `List<Comparable,Integer,?>` is a common interface of `ListFix<Comparable,Integer>` and `SortedListFix<Comparable,Integer>`. 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:

```
class List<Bound<_>, T extends Bound<T>,
           Self<X extends Bound<X>>
                  extends List<Bound,X,Self>>
                 extends Iterable<Bound,T,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 passed to binary methods as the arguments in this programming style. There are several solutions to this problem. In Scala, self types can be explicitly annotated [34] by using `requires` clause. Another solution, invented independently by Saito and Igarashi [35] and Kamina and Tamai [24], is to extend generics a little so that `this` can have abstract types in a special case in exchange for a small restriction on subclassing.

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 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, 25] and parser combinators [28]. We formalize self type constructors as a small calculus FGJ_{stc} 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, 37, 22], 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. We conjecture that the present type system is decidable but showing it is left for future work. Other future work includes the development of a type inference algorithm for polymorphic method invocations.

We believe that the limitation that `List`'s element type had to be exact can be easily addressed by allowing exact types (as well as inexact types) to be passed as type arguments. Fig 9 sketches a solution. There, the element type in the declarations of `Iterator` and `List` is inexact. Class `SortedList` inherits `List` and the element type is instantiated by exact type `@T`. So, the inherited method `iterator()` returns an `Iterator` of the exact type. All in all, `List` is a heterogeneous collection while `SortedList` is homogeneous, as desired. The formalization of this solution is left for future work, too.

```
interface Iterator<T>{
  T next();
  ...
}

class List<Bound>[T extends Bound]
         extends Iterable<Bound>[T]{
  Iterator<T> iterator(){...}
}

class SortedList<Bound extends Comparable>
                [T extends Bound]
      extends List<Bound>[@T]{
                      //  ^^ exact-type instantiation
  Iterator<@T> iterator(){
    ...
  } // if overridden (not necessary)
}
```

Figure 9. Definitions of heterogeneous `List` and homogeneous `SortedList`.

Acknowledgments

Comments from anonymous reviewers of OOPSLA2009 helped us improve the presentation of the present paper. We would like to thank members of the Kumiki project for fruitful discussions on this subject. This work was supported in part by Grant-in-Aid for Scientific Research No. 18200001 and Graint-in-Aid for Young Scientists (B) No. 18700026 from MEXT of Japan (Igarashi). Saito is a research fellow of the Japan Society for the Promotion of Science for Young Scientists.

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 [20]. So, we first prove various substitution lemmas, which are all proved by induction on the derivations, with other auxiliary lemmas.

LEMMA 1 (Weakening). *Suppose* $\bullet \vdash \overline{\mathtt{X}} <: \overline{\mathtt{I}} :: \overline{k}$ *and* $\Delta \vdash \mathtt{U}$ ok.

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

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

LEMMA 2. *If* $\Delta \vdash \mathtt{I}[\overline{\mathtt{H}}] :: k$, *then* $\Delta \vdash \mathtt{I} :: k'$ *for some* k'.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{I}[\overline{\mathtt{H}}] :: k$. □

LEMMA 3. *If* $\Delta \vdash \mathtt{H}_0 <: \mathtt{I}_0$, *then* $\mathtt{H}_0 = \mathtt{K}'[\overline{\mathtt{H}}, \overline{\mathtt{I}}]$ *and* $\mathtt{I}_0 = \mathtt{K}[\overline{\mathtt{I}}]$ *for some* $\mathtt{K}'$, $\mathtt{K}$, $\overline{\mathtt{H}}$, *and* $\overline{\mathtt{I}}$.

Proof. By induction of the derivation of $\Delta \vdash \mathtt{H}_0 <: \mathtt{I}_0$. □

LEMMA 4 (Type Substitution Preserves Subtyping). *If* $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2 \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta_1 \vdash \mathtt{H} <: \mathtt{I}$ *with* $\Delta_1 \vdash \mathtt{H} :: k$, *then* $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{S} <: [\mathtt{H}/\mathtt{X}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2 \vdash \mathtt{S} <: \mathtt{T}$. □

LEMMA 5. *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \mathtt{S} :: k$, *then* $\Delta \vdash \mathtt{T} :: k$.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$. Note that the last premise about the equality on upper bounds in the rule T-CLASS is used in the case of S-CLASS. □

LEMMA 6. *If* $\Delta \vdash \mathtt{I}[\overline{\mathtt{H}}] :: *$, *then there exist some* $\overline{\mathtt{X}}$ *and* $\overline{\mathtt{J}}$ *such that* $\Delta \vdash \mathtt{I} :: \overline{\mathtt{X}} <: \overline{\mathtt{J}}.*$ *and* $\Delta \vdash \overline{\mathtt{H}} <: [\overline{\mathtt{H}}/\overline{\mathtt{X}}]\overline{\mathtt{J}}$ *and* $\Delta \vdash \overline{\mathtt{H}} :: \overline{*}$.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{I}[\overline{\mathtt{H}}] :: *$. □

LEMMA 7 (Type Substitution Preserves Well-Formedness). *If* $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2 \vdash \mathtt{T}$ ok *and* $\Delta_1 \vdash \mathtt{H} <: \mathtt{I}$ *with* $\Delta_1 \vdash \mathtt{H} :: k$, *then* $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash [\mathtt{H}/\mathtt{X}]\mathtt{T}$ ok.

Proof. By induction on the derivation of $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2 \vdash \mathtt{T}$ ok using Lemmas 5 and 6. □

LEMMA 8. *If* $\Delta \vdash \overline{\mathtt{T}}$ ok *and* $\Delta; \overline{\mathtt{x}}{:}\overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$ *for some well-formed bound environment* Δ, *then* $\Delta \vdash \mathtt{T}$ ok.

Proof. By induction on the derivation of $\Delta; \overline{\mathtt{x}}{:}\overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$ with case analysis on the last rule used. □

LEMMA 9. *If* $\Delta \vdash \mathtt{H} <: \mathtt{I}$ *and fields*$(\mathit{bound}_\Delta(\mathtt{I})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, *then fields*$(\mathit{bound}_\Delta(\mathtt{H})) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$ *and* $\mathtt{S}_i = \mathtt{T}_i$ *and* $\mathtt{g}_i = \mathtt{f}_i$ *for all* $i \leq |\overline{\mathtt{f}}|$.

Proof. By straightforward induction on the derivation of $\Delta \vdash \mathtt{H} <: \mathtt{I}$. □

LEMMA 10. *If* $\Delta \vdash \mathtt{K}'[\overline{\mathtt{G}}, \overline{\mathtt{H}}] <: \mathtt{K}[\overline{\mathtt{H}}]$ *and mtype*$(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{K}[\overline{\mathtt{H}}])) = \mathtt{<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{I}}\mathtt{>}\overline{\mathtt{T}}\mathtt{\to}\mathtt{T}_0$, *then mtype*$(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{K}'[\overline{\mathtt{G}}, \overline{\mathtt{H}}])) = \mathtt{<}\overline{\mathtt{Y}}\mathtt{\triangleleft}\overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ *and* $[\mathtt{This}[\overline{\mathtt{G}}]/\mathtt{This}][\overline{\mathtt{Y}}/\overline{\mathtt{X}}](\overline{\mathtt{I}}, \overline{\mathtt{T}}) = (\overline{\mathtt{J}}, \overline{\mathtt{U}})$ *and* $\Delta \vdash \mathtt{U}_0 <: [\mathtt{This}[\overline{\mathtt{G}}]/\mathtt{This}][\overline{\mathtt{Y}}/\overline{\mathtt{X}}]\mathtt{T}_0$.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{K}'[\overline{\mathtt{G}}, \overline{\mathtt{H}}] <: \mathtt{K}[\overline{\mathtt{H}}]$. □

LEMMA 11 (Type Substitution Preserves Typing). *If* $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta_1 \vdash \mathtt{H} :: k$ *and* $\Delta_1 \vdash \mathtt{H} <: \mathtt{I}$, *then* $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2; [\mathtt{H}/\mathtt{X}]\Gamma \vdash [\mathtt{H}/\mathtt{X}]\mathtt{e} : \mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\Delta_1, [\mathtt{H}/\mathtt{X}]\Delta_2 \vdash \mathtt{S} <: [\mathtt{H}/\mathtt{X}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2; \Gamma \vdash \mathtt{e} : \mathtt{T}$. □

LEMMA 12 (Term Substitution Preserves Typing). *If* $\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{T}_0$ *and* $\Delta; \Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$ *where* $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$, *then* $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{S}_0$ *for some* $\mathtt{S}_0$ *such that* $\Delta \vdash \mathtt{S}_0 <: \mathtt{T}_0$.

Proof. By induction on the derivation of $\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{T}_0$. Note that if $\Delta \vdash \mathtt{S} <: \mathtt{@H}$, then $\mathtt{S} = \mathtt{@H}$. □

LEMMA 13. *If* $\Delta \vdash \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]$ ok *and mtype*$(\mathtt{m}, \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]) = \mathtt{<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}_0$ *and mbody*$(\mathtt{m}, \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]) = |\overline{\mathtt{H}}|, \overline{\mathtt{X}}.\overline{\mathtt{x}}.\mathtt{e}_0$, *then*

there exist $\mathtt{P}$, $\overline{\mathtt{I}}$ *and* $\mathtt{S}_0$ *such that* $\Delta \vdash \mathtt{N}[\overline{\mathtt{H}}] <: \mathtt{P}$ *and* $\Delta \vdash \mathtt{P}[\overline{\mathtt{I}}]$ ok *and* $\Delta, \mathtt{This} <: \mathtt{P}, \overline{\mathtt{X}} <: \overline{\mathtt{J}} \vdash \mathtt{S}_0 <: \mathtt{U}_0$ *and* $\Delta, \mathtt{This} <: \mathtt{P}, \overline{\mathtt{X}} <: \overline{\mathtt{J}} \vdash \mathtt{S}_0$ ok *and* $\Delta, \mathtt{This} <: \mathtt{P}, \overline{\mathtt{X}} <: \overline{\mathtt{J}}; \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{@This}[\overline{\mathtt{I}}] \vdash \mathtt{e}_0 : \mathtt{S}_0$ *and* $\Delta \vdash [\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{U}_0$ ok.

Proof. By induction on the derivation of $\mathit{mbody}(\mathtt{m}, \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]) = |\overline{\mathtt{H}}|, \overline{\mathtt{X}}.\overline{\mathtt{x}}.\mathtt{e}_0$ using Lemma 11. □

LEMMA 14 (Narrowing). *If* $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2 \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta_1 \vdash \mathtt{I}' <: \mathtt{I}$, *then* $\Delta_1, \mathtt{X} <: \mathtt{I}', \Delta_2 \vdash \mathtt{e} : \mathtt{T}'$ *for some* $\mathtt{T}'$ *such that* $\Delta_1, \mathtt{X} <: \mathtt{I}', \Delta_2 \vdash \mathtt{T}' <: \mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1, \mathtt{X} <: \mathtt{I}, \Delta_2 \vdash \mathtt{e} : \mathtt{T}$. □

A.1 Proof of Theorem 1

We show only main cases. Other cases, that is, RC-INVK-ARG, RC-NEW and RC-EXACT-BODY, are easy.

Case R-FIELD**:** $\mathtt{e} = \mathtt{new}\ \mathtt{N}[\overline{\mathtt{I}}](\overline{\mathtt{e}}).\mathtt{f}_i$, $\mathit{fields}(\mathtt{N}[\overline{\mathtt{I}}]) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$, $\mathtt{e}' = \mathtt{e}_i$

By T-FIELD, and T-NEW, we have

$\Delta; \Gamma \vdash \mathtt{e}_i : \mathtt{S}_i$ $\quad$ $\overline{\mathtt{G}}, \overline{\mathtt{H}} = \overline{\mathtt{I}}$
$\Delta \vdash \mathtt{S}_i <: [\mathtt{N}[\overline{\mathtt{G}}]/\mathtt{This}]\mathtt{T}_i$ $\quad$ $\Delta \vdash [\mathtt{N}[\overline{\mathtt{G}}]/\mathtt{This}]\mathtt{T}_i$ ok
$\mathtt{T} = [\mathtt{N}[\overline{\mathtt{G}}]/\mathtt{This}]\mathtt{T}_i$

Thus, $\Delta; \Gamma \vdash \mathtt{e}_i : \mathtt{S}_i$ finishes the case.

Case R-INVK**:** $\mathtt{e} = \mathtt{new}\ \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}}).\mathtt{<}\overline{\mathtt{G}}\mathtt{>m}(\overline{\mathtt{d}})$, $\mathit{mbody}(\mathtt{m}, \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]) = i, \overline{\mathtt{X}}.\overline{\mathtt{x}}.\mathtt{e}_0$, $|\overline{\mathtt{H}}| = i$, $\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}})/\mathtt{this}]\ [\overline{\mathtt{G}}/\overline{\mathtt{X}}, \mathtt{N}[\overline{\mathtt{G}}]/\mathtt{This}]\mathtt{e}_0$

By T-INVK and T-NEW, we have

$\Delta \vdash \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]$ ok
$\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}}) : \mathtt{@N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]$
$\mathit{mtype}(\mathtt{m}, \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]) = \mathtt{<}\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{J}}\mathtt{>}\overline{\mathtt{T}}\mathtt{\to}\mathtt{T}_0$ $\quad$ $\Delta \vdash \overline{\mathtt{G}} :: \overline{k}$
$\Delta \vdash \overline{\mathtt{G}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\overline{\mathtt{J}}$ $\quad$ $\Delta; \Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$
$\Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\overline{\mathtt{T}}$
$\mathtt{T} = [\overline{\mathtt{G}}/\overline{\mathtt{X}}, \mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{T}_0$
$\Delta \vdash [\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{T}_0$ ok

Then, by Lemma 13, there exist $\mathtt{P}$, $\overline{\mathtt{I}}$ and $\mathtt{S}_0$ such that

$\Delta \vdash \mathtt{N}[\overline{\mathtt{H}}] <: \mathtt{P}$
$\Delta, \mathtt{This} <: \mathtt{P}, \overline{\mathtt{X}} <: \overline{\mathtt{J}}; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{@This}[\overline{\mathtt{I}}] \vdash \mathtt{e}_0 : \mathtt{S}_0$
$\Delta, \mathtt{This} <: \mathtt{P}, \overline{\mathtt{X}} <: \overline{\mathtt{J}} \vdash \mathtt{S}_0 <: \mathtt{T}_0$.

Then, by Lemma 11 and the fact that none of $\overline{\mathtt{X}}$ appears in $\overline{\mathtt{I}}$, there exists $\mathtt{S}_0'$ such that

$\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\overline{\mathtt{T}}, \mathtt{this} : \mathtt{@N}[\overline{\mathtt{H}}, \mathtt{I}] \vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{e}_0 : \mathtt{S}_0'$
$\Delta \vdash \mathtt{S}_0' <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{S}_0$.

We also have

$\Delta \vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{S}_0 <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{This}]\mathtt{T}_0$

by Lemma 4. Finally, by Lemma 12, there exists $\mathtt{S}_0''$ such that

$\Delta; \Gamma \vdash \mathtt{e}' : \mathtt{S}_0''$ $\quad$ $\Delta \vdash \mathtt{S}_0'' <: \mathtt{S}_0'$.

Finally, by S-TRANS, $\Delta \vdash \mathtt{S}_0'' <: \mathtt{T}$, finishing the case.

Case R-EXACT**:** $\mathtt{e} = \mathtt{exact}^i\ \mathtt{new}\ \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}})\ \mathtt{as\ x,\ X\ in}\ \mathtt{e}_0$, $|\overline{\mathtt{I}}| = i$, $\mathtt{e}' = [\mathtt{new}\ \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}})/\mathtt{x}][\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{X}]\mathtt{e}_0$

By T-EXACT and T-NEW, we have

$\Delta \vdash \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]$ ok
$\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}](\overline{\mathtt{e}}) : \mathtt{@N}[\overline{\mathtt{H}}, \overline{\mathtt{I}}]$
$\Delta, \mathtt{X} <: \mathtt{N}[\overline{\mathtt{H}}]; \Gamma, \mathtt{x} : \mathtt{@X}[\overline{\mathtt{I}}] \vdash \mathtt{e}_0 : \mathtt{U}$
$\Delta, \mathtt{X} <: \mathtt{N}[\overline{\mathtt{H}}] \vdash \mathtt{U} <: \mathtt{S}$ $\quad$ $\Delta \vdash \mathtt{S}$ ok

By Lemmas 11 and 12, $\Delta; \Gamma \vdash \mathtt{e}' : \mathtt{U}'$ such that $\Delta \vdash \mathtt{U}' <: [\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{X}]\mathtt{U}$. By Lemma 4 and the fact that $[\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{X}]\mathtt{S} = \mathtt{S}$, $\Delta \vdash [\mathtt{N}[\overline{\mathtt{H}}]/\mathtt{X}]\mathtt{U} <: \mathtt{S}$. By S-TRANS, $\Delta \vdash \mathtt{U}' <: \mathtt{S}$, finishing the case.

Case RC-FIELD**:** $\mathtt{e} = \mathtt{e}_0.\mathtt{f}$ $\quad$ $\mathtt{e}' = \mathtt{e}_0'.\mathtt{f}$ $\quad$ $\mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

By the rule T-FIELD, we have

$\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{@H}_0$ $\quad$ $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{H}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\mathtt{H}_0 = \mathtt{I}_0[\overline{\mathtt{H}}]$ $\quad$ $\Delta \vdash [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_i$ ok
$\mathtt{T} = [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_i$

By the induction hypothesis, $\Delta; \Gamma \vdash \mathtt{e}_0' : \mathtt{T}_0'$ for some $\mathtt{T}_0'$ such that $\Delta \vdash \mathtt{T}_0' <: \mathtt{@H}_0$. Since $\mathtt{T}_0' = \mathtt{@H}_0$, $\Delta; \Gamma \vdash \mathtt{e}_0'.\mathtt{f} : [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_i$ by T-FIELD. Letting $\mathtt{T}' = [\mathtt{I}_0/\mathtt{This}]\mathtt{T}_i$ finishes the case.

Case RC-INVK-RECV**:**

Similar to the case RC-FIELD.

Case RC-EXACT**:** $\mathtt{e} = \mathtt{exact}^i\ \mathtt{e}_0\ \mathtt{as\ x,\ X\ in}\ \mathtt{e}_1$, $\mathtt{e}' = \mathtt{exact}^i\ \mathtt{e}_0'\ \mathtt{as\ x,\ X\ in}\ \mathtt{e}_1$, $\mathtt{e}_0 \longrightarrow \mathtt{e}_0'$

Case analysis on the typing rule used.

Subcase T-EXACT1**:** $\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{I}[\overline{\mathtt{H}}]$, $|\overline{\mathtt{H}}| = i$, $\Delta, \mathtt{X} <: \mathtt{I}; \Gamma, \mathtt{x} : \mathtt{@X}[\overline{\mathtt{H}}] \vdash \mathtt{e}_1 : \mathtt{U}_1$, $\Delta, \mathtt{X} <: \mathtt{I} \vdash \mathtt{U}_1 <: \mathtt{T}_1$, $\Delta \vdash \mathtt{T}_1$ ok, $\mathtt{T} = \mathtt{T}_1$

By the induction hypothesis, we have $\Delta; \Gamma \vdash \mathtt{e}_0' : \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0 <: \mathtt{I}[\overline{\mathtt{H}}]$. Case analysis on $\mathtt{S}_0$.

Subsubcase: $\mathtt{S}_0 = \mathtt{I}'[\overline{\mathtt{H}}]$ $\quad$ $\Delta \vdash \mathtt{I}' <: \mathtt{I}$

By Lemma 14, $\Delta, \mathtt{X} <: \mathtt{I}'; \Gamma, \mathtt{x} : \mathtt{@X}[\overline{\mathtt{H}}] \vdash \mathtt{e}_1 : \mathtt{U}_1'$ for some $\mathtt{U}_1'$ such that $\Delta, \mathtt{X} <: \mathtt{I}' \vdash \mathtt{U}_1' <: \mathtt{U}_1$. By Lemma 4, $\Delta, \mathtt{X} <: \mathtt{I}' \vdash \mathtt{U}_1 <: \mathtt{T}_1$. By S-TRANS, $\Delta, \mathtt{X} <: \mathtt{I}' \vdash \mathtt{U}_1' <: \mathtt{T}_1$. By T-EXACT1, $\Delta; \Gamma \vdash \mathtt{exact}^i\ \mathtt{e}_0'\ \mathtt{as\ x,\ X\ in}\ \mathtt{e}_1 : \mathtt{T}_1$. Letting $\mathtt{T}' = \mathtt{T}_1$ finishes the case.

Subsubcase: $\mathtt{S}_0 = \mathtt{@I'[\overline{H}]} \qquad \Delta \vdash \mathtt{I'<:I}$

By Lemma 11 and the fact that $\mathtt{X}$ does not appear in $\overline{\mathtt{H}}$ and Γ, $\Delta;\Gamma,\mathtt{x} : \mathtt{@I'[\overline{H}]} \vdash \mathtt{e}_1 : \mathtt{U}_1{}'$ for some $\mathtt{U}_1{}'$ such that $\Delta \vdash \mathtt{U}_1{}'\mathtt{<:}[\mathtt{I'/X}]\mathtt{U}_1$. By T-EXACT2, $\Delta;\Gamma \vdash \mathtt{exact}^i\ \mathtt{e}_0{}'\ \mathtt{as\ x,\ X\ in\ e}_1 : \mathtt{U}_1{}'$. By Lemma 4 and the fact that $[\mathtt{I'/X}]\mathtt{T}_1 = \mathtt{T}_1$, $\Delta \vdash [\mathtt{I'/X}]\mathtt{U}_1\mathtt{<:T}_1$. By S-TRANS, $\Delta \vdash \mathtt{U}_1{}'\mathtt{<:T}_1$. Letting $\mathtt{T'} = \mathtt{U}_1{}'$ finishes the case.

Subcase T-EXACT2**:**

Easy.

A.2 Proof of Theorem 2

We show only main cases. Other cases are easy.

Case: $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i$

If $\mathtt{e}_0$ is not a value, by the induction hypothesis, $\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$ for some $\mathtt{e}_0{}'$; then, RC-FIELD shows $\mathtt{e}_0\mathtt{.f}_i \longrightarrow \mathtt{e}_0{}'\mathtt{.f}_i$.

On the other hand, if $\mathtt{e}_0$ is a value $\mathtt{new\ N}_0\mathtt{[\overline{I}]}\mathtt{(\overline{v})}$, then, by T-FIELD, it must be the case that $\mathtt{T\ f}_i \in \mathit{fields}(\mathtt{N}_0\mathtt{[\overline{I}]})$. Then, $\mathtt{e}_0\mathtt{.f}_i \longrightarrow \mathtt{v}_i$ by R-FIELD.

Case: $\mathtt{e} = \mathtt{e}_0\mathtt{.<\overline{G}>m(\overline{e})}$

If $\mathtt{e}_i$ is not a value, by the induction hypothesis, $\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'$ for some $\mathtt{e}_i{}'$; then, use RC-INVK-RECV or RC-INVK-ARG to show $\mathtt{e}_0\mathtt{.<\overline{G}>m(\overline{e})} \longrightarrow \mathtt{e}_0{}'\mathtt{.<\overline{G}>m(\overline{e})}$ or $\mathtt{e}_0\mathtt{.<\overline{G}>m(..,e}_i\mathtt{,..)} \longrightarrow \mathtt{e}_0\mathtt{.<\overline{G}>m(..,e}_i{}'\mathtt{..)}$, respectively.

On the other hand, if $\mathtt{e}_0$ is a value $\mathtt{new\ N}_0\mathtt{[\overline{H}]}\mathtt{(\overline{v})}$, then by T-INVK, it must be the case that $\mathit{mtype}(\mathtt{m},\mathtt{N}_0\mathtt{[\overline{H}]}) = \mathtt{<\overline{X}\lhd\overline{F}>\overline{S}\to S}_0$. By Lemma 13, $\mathit{mbody}(\mathtt{m},\mathtt{N}_0\mathtt{[\overline{H}]}) = i, \mathtt{\overline{X}.\overline{x}.e'}$ where $|\mathtt{\overline{x}}| = |\mathtt{\overline{e}}|$. We let $(\mathtt{\overline{I},\overline{J}}) = \mathtt{\overline{H}}$ where $|\mathtt{\overline{I}}| = i$. By R-INVK, we have

$$\mathtt{e} \longrightarrow [\mathtt{\overline{e}/\overline{x}},\ \mathtt{new\ N}_0\mathtt{[\overline{H}]}\mathtt{(\overline{v})/this}][\mathtt{\overline{G}/\overline{X}},\ \mathtt{N}_0\mathtt{[\overline{I}]/This}]\mathtt{e'}$$

finishing the case.

Case: $\mathtt{e} = \mathtt{exact}^i\ \mathtt{e}_0\ \mathtt{as\ x,\ X\ in\ e}_1$

If $\mathtt{e}_0$ is not a value, by the induction hypothesis, $\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$ for some $\mathtt{e}_0{}'$; then, RC-EXACT shows $\mathtt{exact}^i\ \mathtt{e}_0\ \mathtt{as\ x,\ X\ in\ e}_1 \longrightarrow \mathtt{exact}^i\ \mathtt{e}_0{}'\ \mathtt{as\ x,\ X\ in\ e}_1$.

On the other hand, if $\mathtt{e}_0$ is a value $\mathtt{new\ N}_0\mathtt{[\overline{G},\overline{H}]}\mathtt{(\overline{v})}$ where $|\mathtt{\overline{H}}| = i$, then by R-EXACT we have

$$\mathtt{e} \longrightarrow [\mathtt{new\ N}_0\mathtt{[\overline{G},\overline{H}]}\mathtt{(\overline{v})/x}][\mathtt{N}_0\mathtt{[\overline{G}]/X}]\mathtt{e}_1$$

finishing the case.

References

[1] Philippe Altherr and Vincent Cremet. Adding type constructor parameterization to Java. *Journal of Object Technology*, 7(5):25–65, June 2008. Special Issue: Workshop on FTfJP 2007.

[2] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Comega. In *Proc. of ECOOP*, volume 3586 of *LNCS*, pages 287–311, 2005.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA '98*, pages 183–200, 1998.

[4] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[5] Kim B. Bruce. Some challenging typing issues in object-oriented languages. In *Proc. of WOOD'03*, volume 82 of *ENTCS*, 2003.

[6] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[7] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *Proc. of ECOOP 2004*, volume 3086 of *LNCS*, pages 390–414, June 2004.

[8] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proc. of ECOOP '98*, volume 1445 of *LNCS*, pages 523–549, 1998.

[9] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proc. of MFPS XV*, volume 20 of *ENTCS*, 1999.

[10] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *Proc. of FTfJP*, Genoa, Italy, July 2009.

[11] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. of ACM Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 273–280, London, England, September 1989. ACM Press.

[12] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *Proc. of AOSD'07*, pages 121–134, 2007.

[13] Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *Proc. of ECOOP 2009*, pages 269–293, July 2009.

[14] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *Proc. of ECOOP2006*, volume 4067, pages 279–303, 2006.

[15] Erik Ernst. Family polymorphism. In *Proc. of ECOOP 2001*, volume 2072 of *LNCS*, pages 303–326, 2001.

[16] Erik Ernst. Higher-order hierarchies. In *Proc. of ECOOP 2003*, volume 2743 of *LNCS*, pages 303–328, 2003.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sotfware*. Addison-Wesley, 1995.

[18] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proc. of International Conference on Generative Programming and Component Engineering 2008*, pages 137–148, New York, NY, USA, 2008. ACM.

[19] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In Rachid Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer*

Science, pages 161–185, Lisbon, Portugal, June 1999. Springer-Verlag.

[20] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. A preliminary summary appeared in Proc. of OOPSLA'99.

[21] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, September 2006. A preliminary version appeared under the title "On Variance-Based Subtyping for Parametric Types" in Proc. of ECOOP2002.

[22] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 113–132, Montreal, QC, October 2007.

[23] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proc. of FTfJP 2004*, June 2004.

[24] Tetsuo Kamina and Tetsuo Tamai. Lightweight scalable components. In *Proc. of 6th International Conference on Generative Programming and Component Engineering(GPCE'07)*, pages 145–154, October 2007.

[25] Erik Meijer. Confessions of a used programming language salesman. In *Proc. of OOPSLA*, pages 677–694, 2007.

[26] Bertrand Meyer. Genericity versus inheritance. In *Proc. of OOPSLA*, pages 391–405, 1986.

[27] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Proc. of OOPSLA (OOPSLA'08)*, pages 423–438, 2008.

[28] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Katholieke Universiteit Leuven, Belgium, 2008.

[29] Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in scala. In *Proc. of International Workshop on Foundations of Object-Oriented Languages (FOOL'08)*, January 2008.

[30] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proc. of OOPSLA '04*, pages 99–115, October 2004.

[31] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. of OOPSLA'06*, pages 21–36, 2006.

[32] Martin Odersky. The Scala language specification, version 2.6. EPFL. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, November 2007.

[33] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *Proc. of ECOOP '03*, volume 2743 of *LNCS*, pages 201–224, Darmstadt, Germany, July 2003.

[34] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. of OOPSLA '05*, pages 41–57, 2005.

[35] Chieri Saito and Atsushi Igarashi. The essence of lightweight family polymorphism. *Journal of Object Technology*, 7(5):67–99, June 2008. Special Issue: Workshop on FTfJP 2007.

[36] Chieri Saito and Atsushi Igarashi. Matching *ThisType* to subtyping. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC2009)*, pages 1851–1858, Honolulu, HI, March 2009.

[37] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(3):285–331, 2008.

[38] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg, November 1998.

[39] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proc. of 13th ECOOP (ECOOP'99)*, volume 1628 of *LNCS*, pages 186–204, Lisbon, Portugal, June 1999.

[40] Mads Torgersen. The expression problem revisited: Four new solutions using generics. In *Proc. of ECOOP2004*, volume 3086 of *LNCS*, pages 123–146, Oslo, Norway, June 2004.

[41] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proc. of SAC'04*, pages 1289–1296, 2004.

[42] Philip Wadler. The expression problem., 1998. Discussion on the Java-Genericity mailing list. Also available at `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[43] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.