Covariant-FGJ and its type-safety proof

by

Bahman Sistany

A thesis

submitted to

the School of Computer Science at
Carleton University

in partial fulfilment of the
requirements for the degree of
Master of Computer Science

Thesis Supervisor: Douglas J. Howe

© Copyright by Bahman Sistany, 2005

December 30, 2005
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Canada

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Abstract

In statically typed class-based object-oriented languages such as Java, user-defined types are created by defining classes. A hallmark of such languages, is the way a class ‘A’ can inherit attributes and/or behavior from another class ‘B’ and extend them by adding new attributes and behavior. Through this inheritance a subtyping relationship between class types ‘A’ and ‘B’ is created and we write $A <: B$, to mean ‘A’ is a subtype of ‘B’. Since values (objects) of type ‘A’, can now be used anywhere a type ‘B’ is expected, we say these objects are polymorphic in their types, or put differently, we say we have “subtype polymorphism”. Another kind of polymorphism called “parametric polymorphism” has recently been added to some statically typed object-oriented languages, notably Java. In this case, a class is parameterized with formal type parameters. To create an instance of the parameterized class, actual type parameters need to be supplied.

In this thesis we first review various ways in which “parametric polymorphism” and “subtype polymorphism” can be integrated and discuss why such integration is useful in the first place. We then present “Covariant Featherweight Generic Java” (Covariant-FGJ), a
formal language that supports a specific style of integration between the two kinds of polymorphism. Covariant-FGJ is formalized as an extension of Featherweight Generic Java (FGJ).

The main contribution of this thesis is a detailed type-safety proof that shows Covariant-FGJ’s type system to be safe. Type-safety proofs in general are notoriously hard to get right, even when such proofs don’t need to be constructed from the ground up, as was the case for Covariant-FGJ’s type-safety proof; the type-safety proof for Covariant-FGJ is based on the type-safety proof of FGJ, however the type-safety proof for Covariant-FGJ is more than a straightforward extension of the proof for FGJ.
Acknowledgments

The research presented in this thesis could not have been possible if not for the help, understanding, and support of many people.

First and foremost I would like to thank Douglas J. Howe. His guidance and support helped me with the most difficult aspects of my research. He challenged many of my original ideas, forcing me to review and to reexamine them with a critical eye, eventually leading to the current research found in this thesis.

I would like to thank Peter Selinger for introducing me to type systems and also for his help and encouragement with the initial ideas behind Covariant-FGJ.

Thanks also to my thesis defence committee members Amy Felty, Dwight Deugo and Pat Morin for their valuable comments that greatly improved the quality of this thesis.

I’d like to thank Dwight Deugo in particular, for everything he taught me about object-oriented programming and about software design patterns. Dwight guided, helped and encouraged me to explore several different areas as I was trying to settle on a research topic.

Last but not least I’d like to thank the loves of my life, Sanam and Roitan
for their continuous support and encouragement and for the many sacrifices they’ve made to make this thesis possible.

Bahman Sistany

Carleton University
19 December 2005
## Contents

1 Introduction ................................................................. 1
   1.1 Overview of the Thesis ............................................ 7

2 Background ................................................................. 7
   2.1 Polymorphism ......................................................... 7
   2.2 Parametric Polymorphism ....................................... 9
   2.3 Subtyping or Subtype Polymorphism ......................... 11

3 Research Question ...................................................... 12

4 Review of the State of the Art ....................................... 14
   4.1 Definition-site (co)variance .................................... 14
   4.2 Variant parametric types ....................................... 22
   4.3 Wildcards ............................................................ 26
   4.4 Definition-site variance v.s. use-site variance .......... 32

5 Featherweight Generic Java (FGJ) ............................... 53
   5.1 Run-time semantics in FGJ ...................................... 56
   5.2 dcast predicate in FGJ ........................................... 58
   5.3 dcast and Covariant-FGJ ....................................... 60
   5.4 Stupid casts in FGJ ................................................ 62
   5.5 Covariant-FGJ cast typing rules ............................... 63

6 Covariant-FGJ: Main definitions .................................. 63
   6.1 Syntax of Covariant-FGJ ......................................... 65

7 Auxiliary Definitions ................................................... 69
   7.1 Field lookup ........................................................ 69
   7.2 Method type lookup .............................................. 69
   7.3 Method body lookup ............................................. 70
   7.4 Cov ................................................................. 71
      7.4.1 Motivation for Cov ......................................... 71

8 Typing ........................................................................ 77
   8.1 Bound of type ...................................................... 78
   8.2 Well-Formed Types ................................................ 78
   8.3 Subtyping ............................................................. 79
9 Computation 80
  9.1 Congruence 81

10 Typing Rules 81

11 Properties of Reduction Semantics 83
  11.1 Type Soundness 83

12 Conclusion 112
  12.1 Summary of Contributions 112
  12.2 Future Work 114
Listings

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>generic type examples</td>
</tr>
<tr>
<td>2</td>
<td>unsafe covariance of Java arrays</td>
</tr>
<tr>
<td>3</td>
<td>access restrictions for covariant type parameters</td>
</tr>
<tr>
<td>4</td>
<td>unsafe covariant subtyping</td>
</tr>
<tr>
<td>5</td>
<td>read-only and read/write versions of class C</td>
</tr>
<tr>
<td>6</td>
<td>VPT example: class C</td>
</tr>
<tr>
<td>7</td>
<td>VPTs restrictions</td>
</tr>
<tr>
<td>8</td>
<td>dummy type argument</td>
</tr>
<tr>
<td>9</td>
<td>un-needed information passed in</td>
</tr>
<tr>
<td>10</td>
<td>generic methods</td>
</tr>
<tr>
<td>11</td>
<td>wildcard restrictions</td>
</tr>
<tr>
<td>12</td>
<td>lower bounds in Scala</td>
</tr>
<tr>
<td>13</td>
<td>class C in Covariant-FGJ</td>
</tr>
<tr>
<td>14</td>
<td>class Vector from VPT paper</td>
</tr>
<tr>
<td>15</td>
<td>class Vector in Covariant-FGJ</td>
</tr>
<tr>
<td>16</td>
<td>fillFromVector method from VPT paper</td>
</tr>
<tr>
<td>17</td>
<td>fillFromVector method in Covariant-FGJ</td>
</tr>
<tr>
<td>18</td>
<td>criticism of definition-site variance</td>
</tr>
<tr>
<td>19</td>
<td>read-only and write-only classes</td>
</tr>
<tr>
<td>20</td>
<td>class C with two type parameters</td>
</tr>
<tr>
<td>21</td>
<td>too many combinations</td>
</tr>
<tr>
<td>22</td>
<td>dcast predicate disallows downcast</td>
</tr>
<tr>
<td>23</td>
<td>dcast predicate allows downcast</td>
</tr>
<tr>
<td>24</td>
<td>allowing updates to instance variables</td>
</tr>
<tr>
<td>25</td>
<td>unsafe update to instance variables</td>
</tr>
<tr>
<td>26</td>
<td>instantiated definition of method set</td>
</tr>
<tr>
<td>27</td>
<td>allowing updatable instance variables with conditions</td>
</tr>
<tr>
<td>28</td>
<td>allowing updatable instance variables with conditions is safe</td>
</tr>
</tbody>
</table>
1 Introduction

Statically-typed class-based object-oriented languages such as Java are the main focus of this thesis. The main facility for creating user-defined types in such languages is the "Class" mechanism. By defining a class, a programmer describes a template from which objects of the class can be created. The class definition is also used to provide method bodies and assign initial values to fields. By defining a class, a programmer also defines a type for all the objects that will be created from the class definition.

In statically-typed programming languages, when a value of type 'S' can be used anywhere a value of type 'T' is expected, we say type 'S' is a subtype of type 'T' and we write $S <: T$. In most of today's statically typed object-oriented languages, subtyping is implemented through inheritance. When a class 'A' inherits from class 'B', written in Java as "class A extends B...", a subtyping relationship is created between the two class types. Values of type 'A' are also of type 'B', and therefore can be used anywhere a type 'B' is expected. This property is referred to as "subtype polymorphism". Another kind of polymorphism called "parametric polymorphism" is also possible.

Parametric polymorphism in general refers to type or code that is parameterized with formal type parameters. In the case of statically-typed
object-oriented languages, the type or code takes the shape of classes or methods. The definition of a class ‘A’ parameterized with the formal type parameter ‘X’ in Java would look like: “public class A(X) {}”.

The parameterized class needs to be supplied with actual type parameters when being instantiated: “new A<Integer>(5);”. A<Integer> is called a parametric type or a generic type and is an example of how generic classes induce generic types. In general, generic types are produced by referring to a class name followed by another generic type; syntactically in Java, the type variable or the other generic type is surrounded by angle brackets, as shown in Listing[1] (see section[6.1] for more detail on the syntax of generic types):

Listing 1: generic type examples

```java
class C<X> {} 

// a variable c is declared to have a generic type C<Integer>, 
// which is an instantiation of the generic class C 

C<Integer> c;
```

When parametric polymorphism is added to languages that support subtype polymorphism, the two kinds of polymorphism must coexist. One way
such coexistence could be achieved is to keep the two features independent of each other; we can do this by keeping the subtyping rule intact, even for parameterized types. That is when \textit{“class A\langle X \rangle extends B\langle X \rangle...”}, the only subtype relation created is $A\langle S \rangle <: B\langle S \rangle$, where $S$ is any type. Notice the fact that the actual type argument (e.g. $S$) on the left hand side of the $<$ is the same as the one on the right hand side. In other words, the type variable doesn’t vary with the direction of the subtyping relationship and the instantiated types $A\langle S \rangle$ and $B\langle S \rangle$ are said to be in a point-wise subtyping relationship \cite{1}.

Limiting the type system to point-wise subtyping means in particular that not only different instantiations of the same generic class e.g. $A\langle S \rangle$ and $A\langle T \rangle$, or $B\langle S \rangle$ and $B\langle T \rangle$ have no subtyping relationship, but also the more general $A\langle S \rangle$ and $B\langle T \rangle$ have no relationship between them. This is true no matter what the relationship of $S$ and $T$ (where $S \neq T$).

The most intuitive and useful of the subtype relationships that can be defined between different instantiations of the same generic class, is called “covariant subtyping”. If $C\langle S \rangle <: C\langle T \rangle$ whenever $S <: T$, we say $C\langle S \rangle$ and $C\langle T \rangle$ are in a covariant subtyping relationship. For example, it is intuitive to think that a $\text{List}\langle \text{Circle} \rangle$ should be a subtype of $\text{List}\langle \text{Shape} \rangle$. 

3

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
However covariant subtyping is not in general, type-safe. Consider the example of Java arrays. A Java array can be considered a generic class with a single type parameter: \textit{class Array\{X\}}. As it turns out, Java arrays are covariant in their type parameters e.g., an array of strings \textit{String[]} is a subtype of an array of objects \textit{Object[]}]. This is true since \textit{String} is a subtype of \textit{Object}.

\begin{verbatim}
Listing 2: unsafe covariance of Java arrays
Object [] objects = new String[2];
objects[0] = new String("hello");
objects[1] = new Integer(5);
\end{verbatim}

In Listing[ 2] above, an array of strings of length 2 is created first and assigned to a variable \textit{objects} of type \textit{Object[]}]. This is type-checked successfully by the compiler because \textit{String[]} is a subtype of \textit{Object[]} and an instance of \textit{String[]} can be assigned to variable \textit{objects}.

An instance of type \textit{String}, namely the string literal "hello" is then assigned to the first element of the array variable \textit{objects}. This assignment is allowed as well, because \textit{String} is a subtype of \textit{Object}.

The last statement is also compiled without any problem. In this case, the assignment of an instance of type \textit{Integer} to the second element of the
array variable objects is permitted, because Integer is a subtype of Object. The array variable objects having run-time type of String[] has as its second element, an instance of a type different from String; this can lead to run-time errors such as calling a String specific method on the second element of the array and eventually a run-time error. To prevent such run-time errors from happening, Java’s run-time system checks each assignment to an array element, insuring an object of the right type is assigned to the array element and if not, it will throw an exception (as is the case in the last statement).

Java as the prototypical programming language with support for both subtype polymorphism and parametric polymorphism is an ideal setting for the study of various ways the two kinds of polymorphism can be integrated. However the full Java with advanced features such as concurrency and reflection is not well suited for formal study. The emphasis should be on Java’s type system and its formal properties and on how the language (and its type-safety) might be affected by additions and/or modifications to it.

FGJ [2], is a formal language that meets the above criteria. FGJ is minimal in that it only models essential Java features such as classes, methods, fields and inheritance. The FGJ language definition specifies its syntax, typing rules and operational semantics. A formal type-safety proof of FGJ is
also presented in [2].

In this thesis, FGJ's syntax, typing rules and operational semantic rules were modified to add support for a particular implementation of covariant subtyping. These modifications included a new covariant subtyping rule, new syntax to allow declaration of covariant type parameters in generic class definitions and other required restrictions, resulting in Covariant-FGJ.

The type-safety proof of FGJ as presented in [2] is extremely sensitive to modifications to the language definition: The main type-safety theorems use simpler lemmas that are proved in sequence, each building on the preceding one and all of them using specifics of the language definition (syntax, subtyping and typing rules, etc). Therefore, simple modification of the type-safety proof of FGJ to show that Covariant-FGJ is type-safe, was not an option in this thesis. To prove type-safety of Covariant-FGJ and hence to show that the above modifications made sense, a detailed type-safety proof of Covariant-FGJ from first principles is presented below. Although where appropriate, we have used the original FGJ type-safety proofs; clarifications and alternate proofs are presented for some of the other lemmas and/or theorems present in the original paper and finally where applicable, most of the proofs that are missing from the original paper are presented in this thesis.
1.1 Overview of the Thesis

In the following sections, we motivate why integrating subtype and parametric polymorphism—in particular covariant subtyping—is useful, by examining some background of the general notion of polymorphism and its accepted classifications in the current literature. We then examine the main contenders in this field which are variant parametric types [1], wildcards [4], Scala [5], NextGen [6] and Covariant-FGJ. Finally, after a detailed introduction to FGJ, the definition of Covariant-FGJ and its type-safety proof is presented.

2 Background

2.1 Polymorphism

Merriam-Webster dictionary defines polymorphism as "the quality or state of being able to assume different forms". In statically typed class-based object-oriented programming languages which is the main focus of this thesis, polymorphism refers to the ability to treat instances of a class ‘A’ as if they were instances of another class ‘B’; this is possible when class ‘A’ inherits from class ‘B’.

More generally, polymorphism is classified into universal polymorphism
and ad-hoc polymorphism [7], although only universal polymorphism is considered true polymorphism since only this kind of polymorphism is applied uniformly on similarly structured types. Ad-hoc polymorphism on the other hand, works on a finite set of unrelated types. Furthermore, typically, implementations of universal polymorphism, use the same code over the infinite set of types, whereas ad-hoc polymorphism uses different code for each polymorphic function/type.

Ad-hoc polymorphism is similarly divided into two different categories: overloading and coercion. Overloading refers to defining different functions with the same name where typically the right function is resolved by the implementation, depending on the number and types of the actual parameters. Overloading is simply syntactic sugar where usually the functions with the same names go through a mangling phase. This phase renames the functions and the calls to them differently based on context. Coercion on the other hand, refers to a semantic operation that converts one type to another. Coercion is either indicated by explicit syntax or done implicitly by compilers when a type different from what was expected is seen.

Universal polymorphism is further divided into parametric and inclusion polymorphism categories. Although inclusion polymorphism for statically
typed class-based languages is the same as subtype polymorphism (we will therefore only deal with subtype polymorphism in the rest of this document), it does not need to be so. Inclusion polymorphism refers to a value being viewed as belonging to many different and possibly unrelated types [7]. For example a record type \{x : INT, y : INT\} “includes” another record type \{x : INT\} and under inclusion polymorphism any function that takes values of the first type can take values of the second type. Subtype polymorphism is a specific instance of the more general term “inclusion polymorphism”. If \(S <: T\) then we say \(T\) includes \(S\).

Parametric polymorphism achieves its uniformity of type structure by abstracting the types as parameters. Subtype polymorphism on the other hand, works by means of subsumption. The subsumption rule states that if an expression has type \(S\), then the expression has also type \(T\) where \(T\) is any supertype of \(S\).

2.2 Parametric Polymorphism

Lambda calculus could be considered the smallest and the simplest programming language there is. In the simply typed version of lambda calculus we can define identity functions for two different types: \(id1\) and \(id2\) as
\[ id1 = \lambda x : INT. x \text{ and } id2 = \lambda x : BOOL. x \text{ (assuming } INT \text{ and } BOOL \text{ are basic types of the language)} \ [8]. \]

Comparing the two functions we notice that the only difference in the two functions is in the type of the input \( x \). The observation that the code of the two functions is nearly identical naturally leads to the idea of type abstraction (also called type generalization). We abstract out the type in the code of the functions by adding a type variable to obtain the following:

\[ id3 = \lambda X. \lambda x : X. x. \] In this version of the identity function, \( X \) is the type variable that needs to be given a type value before the function \( id3 \) can be applied to actual arguments.

The single function \( id3 \) could now be applied (instantiated) to concrete types \( INT \) and \( BOOL \) to get back the original functions \( id1 \) and \( id2 \): \( id1 = id3[INT] \) and \( id2 = id3[BOOL]. \)

Adding these notions of type abstraction and application, along with new polymorphic types and typing rules extends the simply typed lambda calculus giving us a richer calculus called the polymorphic lambda calculus. Lambda calculus was therefore enhanced in expressive power and abstraction facilities by adding parametric polymorphism.

Parametric Polymorphism was influenced by a formal system in logic
called System F originally developed by Girard [9]. System F based type systems represent the most powerful polymorphic type systems in terms of their expressive power, however this expressive power comes at a cost: to actually use some of the more complex polymorphic types in implementations of such type systems, type inference is highly desirable but, type inference for System F based type systems is known to be undecidable [8]. In addition to undecidability of type inference, as soon as general recursion is added to a programming language (which is the case for most real programming languages), type checking becomes undecidable. In practice therefore, more limited versions of System F based type systems are used.

### 2.3 Subtyping or Subtype Polymorphism

Subtyping is an important feature of statically typed class-based object-oriented languages such as Java. In such languages, if class $A$ inherits from another class $B$, then type $A$, is a subtype of type $B$. When one class is declared to inherit from another class in this fashion, the resulting subtype relationship is referred to as “Nominal Subtyping”. Subtyping could also be “Structural”. In this case a class $A$, having at least the same interface as another class $B$, could be considered to be inducing a subtyping relationship.
between \( A \) and \( B \). An example of structural subtyping is the record types example given above: a record type \( \{ x : \text{INT} \} \) is a subtype of another record type \( \{ x : \text{INT}, y : \text{INT} \} \). In any case, subtype polymorphism insures that if an object \( v \) has type \( A \), then \( v \) can be used in any context where an object of type \( B \) is expected.

### 3 Research Question

The idea of adding generics or parameterized types to Java has been behind much recent research into how to best integrate the two kinds of polymorphism [1, 4, 10].

In Java when "\texttt{class A}(X) \texttt{extends B}(X)\{\}\)”, the only subtype relation created is \( A(S) <: B(S) \), where \( S \) is any type. In other words, \( A(S) \) is a subtype of \( B(S) \) if and only if \( A(X) \) is declared to be a subclass of \( A(X) \). The type \( S \) doesn’t vary with the direction of the subtyping relationship and the instantiated types \( A(S) \) and \( B(S) \) are said to be in a point-wise relationship; the left and right hand side parameterized types are in a subtype relationship but the parametrization has been kept invariant. Put differently, subtyping is one-dimensional.
Point-wise subtyping is simple and intuitive but it is not expressive enough. Is it possible to vary the type variable over the $<$: to have two dimensional subtyping? For example could we allow $A(S) <: A(T)$ when $S <: T$ (covariant subtyping)?

If the answer to the research questions above is in the affirmative, even subtyping relationships such as $A(S) <: B(T)$ could be allowed (by using transitivity from $A(S) <: B(S)$ and $B(S) <: B(T)$). This would mean that subtype polymorphism can be applied to all parametric types, whether the actual type parameter on both sides of the $<$: is the same (as in the point-wise case) or not. We would have full integration of the two kinds of polymorphism and the resulting type system would be more powerful and more expressive than before.

Although the addition of subtyping schemes such as covariant subtyping to a language results in a type system that is more expressive, the type system still needs to be type-safe. How could this type-safety be achieved concretely and what metrics do we use to insure our type-safety implementation meets its goals?

In this thesis we answer the type-safety question by imposing certain restrictions on field accesses and method invocations that are not safe. Al-
though this general solution to recovering type-safety, when covariant-subtyping is added to a language that only supports point-wise subtyping, is well-known in the literature [3, 1], to our knowledge, a formal proof of type-safety of the resulting calculus—which we present in this thesis—has not previously been done.

4 Review of the State of the Art

4.1 Definition-site (co)variance

Subtype relationships that can be defined between different instantiations of the same generic class could be made statically type-safe. This type-safety can be achieved by effectively avoiding problematic fields and methods (ones with certain signatures) of the generic class. One way of avoiding such fields and methods is to not even allow their definitions. Another way would be to let them be defined but to restrict access to them, once they are defined.

In this section we consider applying type-safety restrictions on generic class definitions. We disallow certain fields and methods from being defined if the class is indicated to be covariant. This way of providing type-safe covariance is referred to as “Definition-site covariance”. Definition-site covari-
ance has been the traditional approach and until recently the only approach to type-safe covariant subtyping.

In definition-site covariance we impose restrictions on how a type variable can appear in a class definition, if the generic class is to be covariant in that type variable. If a class is defined to be covariant in the type parameter X for example, then X cannot appear in the type of a public/writable field nor in the type of any of the arguments of any public method. For example, in the following class definition written in Java (see Listing 3), in which the type parameter X is to be treated as covariant, X does not appear in the type of the public and writable field g and nor in the type of the argument y of method set:

Listing 3: access restrictions for covariant type parameters

```java
// class C is defined with two type parameters X and Y
class C<X,Y> {
    private X f;
    Y g;

    // constructor through which fields f and g are initialized
    C(X f, Y g)
```
It is instructive to see what could happen if an attempt is made to use covariant subtyping without such restrictions. See the class definitions in Listing 4:

Listing 4: unsafe covariant subtyping

```java
class C<X, Y> {
    X f;
    Y g;

    C(X f, Y g)
    {
        this.f = f; this.g = g;
    }
}
```
void set(X x)
{
  f = x;
}

class CIntStr extends C<Integer, String> {

  // constructor
  CIntStr(Integer f, String g)
  {
    super(f,g);
  }

  void set(Integer x)
  {
    f = f.add(x);
  }
}
void func1(C<Object, String> c) {
    c.set(new String("hi");
}

void func2() {
    func1(new CIntStr(1, "hello");
}

In function `func2` of the above listing (Listing[4]), an instance of `CIntStr` is created and passed to function `func1`. This expression will be type-checked since `CIntStr <: C<Integer, String>` and by covariance `C<Integer, String> <: C<Object, String>` (assuming `Integer <: Object`).

The call to method `set` is also type-checked successfully because the static type of `set`'s formal parameter is seen to be `Object` (c is of type `C<Object, String>`), therefore parameter x of function `set` has static type `Object`). At run-time, when `set` is called with an actual argument of type `String`, an attempt to call an `Integer` specific method `add`, on an object of type `String` will result in a run-time failure.

The class `C` defined with the restrictions mentioned, results in what is effectively a read-only class where the element of type X cannot be written to.
Having to make classes read-only to obtain covariant-subtyping is perhaps too steep a price to pay for the resulting expressiveness. However, [1] suggests having writable versions of read-only classes to remedy this situation (see Listing[5]):

Listing 5: read-only and read/write versions of class C

```java
class ReadOnlyC<X,Y> {
    private X f;
    Y g;

    C(X f, Y g)
    {
        super(); this.f = f; this.g = g;
    }

    void set(Y y)
    {
        g = y;
    }
}

class C<X,Y> extends ReadOnlyC<X,Y> {
```
C(X f, Y g)
{
    super(f,g);
}

void set(X x)
{
    f = x;
}

class CIntStr extends C<Integer, String> {
    CIntStr(Integer f, String g)
    {
        super(f,g);
    }
    void set(Integer x)
    {
        f = f.add(x);
    }
}
Here, the class designer declares a read-only class called `ReadOnlyC`. This is equivalent to declaring the class to be covariant in X and in fact, in later sections we will introduce and use special syntax (the '+ ' annotation in front of the covariant type parameters) to do just that.

Class C is a writable class which extends class `ReadOnlyC` and has the method `set` through which the element of type X namely f can be written to. Class C is therefore not a covariant class.

Since class C is not covariant in X, when `func1` is called with an instance of `CIntStr`, type-checking will fail. The type-checker will not type the call to `func1` because although `CIntStr <: C(Integer, String), C(Integer, String)` is no longer a subtype of `C(Object, String).

To obtain covariant subtyping C can be replaced with the read-only ver-
sion (the covariant version) namely with \textit{ReadOnlyC} in the call to \textit{func1} and this would allow the call expression to be typed successfully. However, the compilation would still fail at the point of calling the method \textit{set}, because no \textit{set} method is defined for class \textit{ReadOnlyC}.

Note that although \textit{ReadOnlyC} does not allow updates to the element of the designated covariant type parameter \textit{X}, updates to the elements of invariant type parameters such as \textit{Y} are allowed.

The above example shows that by imposing certain restrictions such as designing classes to be read-only (in a specific type parameter), type-safe covariant subtyping could be achieved. Furthermore, supplying writable versions of the read-only classes compensates for the loss functionality that is lost when classes are made read-only.

### 4.2 Variant parametric types

In this section and next, we consider two proposed ways of restricting access to problematic fields and methods (ones with certain signatures) of the generic class used in covariant subtyping. This approach of restricting access at use-site of a generic type, is in contrast to “Definition-site covariance” covered in section[ 4.1] and is appropriately called “Use-site variance”.

22
Igarashi and Viroli [1] proposed “Variant parametric types” (VPTs) as a specific implementation of Use-site variance to type-safe variant subtyping.

In the VPT scheme, we use special syntax – called variance annotations—to annotate generic types. These so-called variance annotations, for example + for covariance and – for contra-variance are associated with the actual type parameters of a generic type e.g. Vector(+Number).

The variance annotations indicate the intended use of the parametric type and what kind of subtyping the generic type induces. For example, all instantiations of the form C(S) where $S <: Number$ are subtypes of the variant generic type $C(+Number)$. We can therefore assign instances of $C(Number), C(Integer)$, as well as $C(Float)$ to a variable of type $C(+Number)$ (assuming of course that $Integer <: Number$ and $Float <: Number$ as they are in Java). This subtyping relationship is called covariant subtyping.

We can also have contra-variant subtyping where all instantiations of the form $C(S)$ where $Number <: S$ are subtypes of $C(–Number)$. In the context of VPTs, invariant subtyping in a generic type is specified by not having any variance annotations at all; this is just the plain old parametric type with no special subtyping relation being defined: $C(Number)$ is a subtype of itself.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In addition to specifying what kind of subtyping the generic type induces, the different variance annotations of VPT indicate how field accesses/method invocations on objects of variant parametric types are to be restricted. For example, if class $C$ is defined as:

```java
class C <Z> {
    Z f;
    void add(Z x) { f = x; }
}
```

Then assuming a variable $c$ has static type $C$ <+Object>, the + variance annotation indicates that methods that take arguments of the type parameter $Z$ cannot be called through variable $c$. For example the method `add` of class $C$ above, will not be allowed to be called through variable $c$. This means that the expression $c.add(new String("hi"))$ will not be type-checked (regardless of what the type of the actual argument is). By declaring the type of $c$ as $C$ <+Object>, the method `add`'s argument type is seen by the compiler to be `Object`. The call to `add` then cannot be allowed, since for example two different calls to `add` can each pass a different subtype of `Object` to the
method and the compiler cannot distinguish one from the other.

For a more specific example see Listing[7]: \textit{CInt} class inherits from a specific instantiation of generic class \textit{C} (see the definition of class \textit{C} in Listing[6] above). An object of type \textit{CInt} is then instantiated and assigned to a variable \textit{c} of type \textit{C} (+\textit{Object}). Now if the call to method \textit{add} in the last line of the listing was allowed to go through static type checking, a run-time failure would happen. The run-time failure happens because the static type of \textit{c} is declared to be \textit{C} (+\textit{Object}), so passing a \textit{String} to \textit{add} is typed, even though the specific \textit{add} that is being called has an argument type of \textit{Integer}. This type mismatch leads (at run-time) to the method \textit{someIntSpecificMethod} being called on an instance of \textit{String} which obviously will result in a run-time failure.

Listing 7: VPTs restrictions

```java
class CInt extends C<Integer> {

    CInt(Integer f) {
        super(f);
    }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
```java
void add(Integer x) { x.someIntSpecificMethod(); }
}

C<Object> c = new CInt(1);
c.add(new String("hi"));
```

### 4.3 Wildcards

As Java designers started considering the integration of subtype polymorphism and parametric polymorphism, they experimented with variant parametric types and in fact used variant parametric types in their early implementations of Java generics. As more experience and insight was gained into the use of variant parametric types, certain shortcomings and areas for improvement were detected [10, 4].

It had also been separately realized that with or without VPTs, in generic Java, there was no way of abstracting over different instantiations of the same generic class without using a specific type parameter. For example, the only way to abstract over the actual type parameter in `List<Integer>` and `List<String>` types was to use the generic type `List<Z>` where the type
parameter $Z$ is known and can be referred to by name. The type parameter $Z$ although abstract, can still be assumed to have at least type $\text{Object}$ (the supertype of all types in Java) and therefore methods of $\text{Object}$ can be called on values of type $Z$. It would be useful to be able to express the type of all lists, while at the same time not have to specify a type parameter at all.

Consider for example the method $\text{length}$ of Listing 8 that takes as argument a generic list and returns the size of the list. Calculating the size of a list, is of course independent of what the actual argument type is in the list. We need to be able to express this fact somehow.

Listing 8: dummy type argument

```java
class C <X> { 

    int length(List<X> lst) { }

}
```

In the pre-generic Java days, we would have used the non-generic type $\text{List}$ for the type of all lists. However, the main point of adding generics was to use parameterized types to obtain their known benefits (abstraction, expressiveness, etc). Wildcards are a way of abstracting out the type variable of generic types like $\text{List}(X)$ without going all the way back to non-generic
types. Syntactically a wildcard ‘?’ replaces the actual type argument in a
generic type; for example List(?) is the type of all generic lists.

In the example of Listing[8] where wildcards are not used, the only reason
that the method signature contains a type parameter (a type parameter that
is not used in the body of the method), is to take as input a generic list
(which is more useful than a non-generic one after all). The type variable is
then forgotten in the body of method length.

Forcing clients of methods like length to pass in information that is not
needed, is clearly not very sound software engineering practice (this is true
in the general case and is not specific to this particular context). For ex­
ample, forcing the clients of a method printAccountNumber to pass in a
ClientAccount instead of a AccountNumber leaves the door open to accessing
the account balance and/or other information that is not needed by the
method (the method could by mistake print this information for example):

Listing 9: un-needed information passed in

```java
void printAccountNumber(ClientAccount acc)
{
    print (acc.getAccountNumber());
    print (acc.getAccountBalance());
    print (acc.getAccountInfo());
}
```

28

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Even if methods like `length` could be defined using dummy type parameters (a type parameter that is not used in the body of the method), the same work-around cannot be used for fields, since fields are not parameterized in Java the way methods are. Method definitions in Java can be generic just like class definitions and can take type parameters which are different from the type parameters of the method’s generic class. In the worst case each method that needs a dummy type parameter could be declared to be generic in that type parameter, as shown in Listing[10]:

```java
class C <X> {
    <Y> D<Y> f; // NOT ALLOWED
    <Z> int length(List<Z> lst) { } // OK
}
```

A general solution namely wildcards that could express generic types where the actual type parameter is unknown and is also applicable to both fields and methods is therefore needed.

Syntactically, wildcards can be seen to unify variant parametric types’
different variance annotations into the single ‘?’ with optional bounds. The optional bounds are types that are specified by keywords extends (corresponding to the ‘+’ variance annotation) or super (corresponding to the ‘-’ variance annotation). For example, \(C(\text{? extends Integer})\) indicates that the unknown actual type parameter can vary within a set of subtypes of type \text{Integer} and \(C(\text{? super String})\) indicates that the unknown actual type parameter can vary within a set of supertypes of type \text{String}.

The ‘?’ syntax emphasizes that the actual type parameter of the generic type is unknown. It is important to note that ‘?’ does not stand for a specific type; for example in type \text{Pair{?, ?}}, ‘?’ does not mean that the first parameter and the second are the same:

\[
\text{Pair{?, ?}} \text{pair} = \text{new Pair{Integer, String}().}
\]

Wildcards enforce some access restrictions which are similar to restrictions imposed by variant parametric types. For example, in Listing[11]:

```java
class Pair <X extends B> {
    X first;
    X second;

    X getFirst() { return first;}
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
X getSecond() { return second; }

void setFirst(X x) { first = x; }
void setSecond(X x) { second = x; }

// pair has type Pair<?,?>

B b = pair.getFirst(); // OK

// NOT OK
pair.setSecond(new Integer(2));

If pair had static type Pair<?,?>, then calling a method like setSecond (or setFirst) that takes a parameter of type X (the type parameter of the generic class Pair) through pair cannot be permitted by the type checker, since the actual type parameters of pair’s type are unknown and could be any type. It would not be safe in general to let an Integer to be written to the second field (or to the first field), if pair’s actual type was for example Pair(String, String); we’d end up with a Pair instance where the types of the two fields would not match.
When methods like `setSecond` or `setFirst` of class `Pair` in Listing[11] are called, it is possible that some fields are updated e.g. `first` and `second` in this case. This is why calling such methods are referred to as “writing into the element type of a generic class”. On the other hand when we call methods that return the type parameter of the generic class e.g. `getFirst` or `getSecond`, we say we are “reading from the element type of a generic class”.

Continuing with class `Pair` of Listing[11], reading through variable `pair` (as in `pair.getFirst()`) is ok in contrast to writing, which was not ok.

4.4 Definition-site variance v.s. use-site variance

Compared to Covariant-FGJ or any other definition-site variance based system such as Scala or NextGen, use-site variance based type systems support much richer types. They therefore offer better abstraction facilities and are more flexible. This flexibility and power comes at a price however. Use-site variance type systems are extremely complicated, and because of this complexity, they tend to lead to type errors.

Definition-site variance based type systems, on the other hand, are more inline with the traditional object-oriented style of programming with all its benefits and drawbacks. The class designer has to indicate explicitly the
variance of a type parameter of a generic class to create the intended subtyping relationship. This is very similar to the way inheritance is implemented in Java-like object-oriented languages; the class designer explicitly subclasses another class and induces the corresponding subtype relationship. When use-site variance is used, this explicit control is no longer available to the class designer. Depending on one’s philosophy, it could also be argued that this lack of control is actually a good thing. The class designer is not burdened with having to declare variance. Whereas definition-site variance implies “intentional subtyping”, use-site variance implies “unintentional subtyping”, although still safe subtyping. Since the class designer can not predict the exact ways the class will be used, traditional notions of object-oriented modeling and design will need to be revisited in this context.

As a concrete implementation of definition-site variance, consider the language Scala. “Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.” [5] What’s more, Scala supports generics and definition-site variance annotations. Use-site variance annotations were however seriously considered to be added to the language at one point.
In the end Scala designers decided against using use-site variance, despite its greater flexibility, mostly because “We found that in practice it was quite difficult to achieve consistency of usage-site type annotations, so that type errors were not uncommon” [11].

In the case of a class where a type variable is annotated with a variance, Scala imposes the familiar restrictions on how the type variable can appear in the class definition.

If a type variable is annotated with a ‘+’ and it appears in the type of any of the method’s argument list, it is in a *contra-variant position* [11] and if it appears as part of the return type of a method, the type variable is in a *covariant position*. *Invariant position* refers to the situation where the annotated type variable appears as part of the type of a mutable field. The type variable that is annotated with a ‘+’ cannot appear in a contra-variant position and vice-versa, if the type variable is annotated with a ‘-‘, it cannot appear in a covariant position.

The restrictions above are quite similar to Covariant-FGJ’s restrictions. Although in contrast to Covariant-FGJ, when a generic class is declared to be covariant in a type variable, Scala does not prevent definition of methods where the covariant type parameter appears in the type of method’s argu-
ments (with a caveat). To accommodate such methods Scala uses a novel approach and allows syntax to specify lower-bounds: a new type parameter is added to methods (to make them polymorphic) and the covariant type variable of the class is declared to be a lower-bound for the new type variable. The new type variable is then used in the method’s signature. Such polymorphic methods with lower-bounds are said to simulate contra-variance in a covariant class [12]. See the example shown in Listing[12].

A lower bound in the definition of method set of class D, catches unsafe operations, while allowing the ones that are safe to occur: passing a B or a supertype of B like A to set is safe but passing in a subtype of B like C is unsafe (see class Pair in Listing[11] or Java arrays in Listing[2] for examples of why passing subtypes of the argument type in this way is unsafe).

Listing 12: lower bounds in Scala

```scala
class A{}

class B() extends A{}

class C() extends B{}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
class D[S]() {
    def set[T <: S](x: T): Unit = Console.println(x);
}

object HelloWorld with Application {
    // $S$ in the definition of class $D$ is instantiated with $B$
    val d = new D[B]();

    // $T$ is the upper-bound in the definition of method set of class $D$
    d.set(new B()); // $T = B$ and $B <: B$ so this is OK
    d.set(new A()); // $T = A$ still OK, since $A <: B$
    d.set(new C()); // $T = C$ is not OK, since $C$ not $ <: B$
}

We will next examine definition-site variance, specifically Covariant-FGJ, in terms of its expressiveness and compare and contrast it with use-site variance based type systems, namely variant parametric types and wildcards. In the rest of this section, Covariant-FGJ will be used to type certain examples cited for variant parametric types [1] and wildcards [4].

First, we define a generic class $ROC$, shown in Listing[ 13] in pseudo
Covariant-FGJ syntax where the type variable $X$ is declared to be covariant by annotating it with '+'.

Listing 13: class C in Covariant-FGJ

```java
class ROC <+X> {
    Pair<X,X> asPair(){}
    X get() {}
    ROC <ROC <X>> nest() {}
}

class C <X> extends ROC <X> {
    void add(X x){}
}
```

Given the definitions given in Listing[13] we’ll examine the typing of some expressions involving a variable `box` in Covariant-FGJ, assuming the static type of `box` is `ROC<T>` (this is equivalent to types $C(+T)$ and $C(? extends T)$ in variant parametric types and wildcards respectively).

In Covariant-FGJ, we type `box.get()` as $T$, `box.asPair()` as $Pair<T,T>$
and box.nest() as $ROC(ROC(T))$. In VPT, box.asPair() is typed as $Pair(+T,+T)$ (box having type $C(+T)$) and box.nest() is typed as $ROC(+ROC(+T))$.

Notice, in the case of box.asPair(), that in Covariant-FGJ the information that both type arguments to $Pair$ are the same is not lost. In variant parametric types this information is lost, because type $Pair(+T,+T)$ only expresses that the first and second parameters are some subtypes of $T$ and not that they are the same.

Similarly, in the case of box.nest(), the fact that the return type of nest() is of form $ROC(ROC(S))$ (where $S <: T$) is expressed in Covariant-FGJ, whereas in VPT, it is not: the equivalent type $ROC(+ROC(+T))$ is weakened because of the '+' on the inner $ROC$ allows any subtype of $ROC$ in that position.

In wildcards, box.asPair() is typed as $Pair(Z,Z)$, where $Z <: T$. We could for example return a $Pair(S,S)$ inside the body of asPair as long as $S <: T$. In Covariant-FGJ, however, we could only return a $Pair(S,S)$ (inside the body of asPair), if class $Pair$ had been declared to be covariant in both type arguments, otherwise we could only return an invariant subtype of $Pair(T,T)$. Again, this is either a benefit or an unintentional consequence,
depending on how one looks at it.

We next examine a few examples from the variant parametric types paper [1] in the Covariant-FGJ context with emphasis on covariance. A running example from [1] uses the following definition for a class Vector in a pseudo-GJ syntax:

Listing 14: class Vector from VPT paper

```java
class Vector <X> {
    private X[] ar;

    Vector(int size){ ar=new X[size]; }

    int size(){ return ar.length; }

    X getElementAt(int i){ return ar[i]; }

    void setElementAt(X t, int i){ ar[i]=t; }

    // gets elements from v
    void fillFrom(Vector <+X> v, int start)
}
```
Vector <Number> vn =
    new Vector <Number>(10);
Vector <Integer> vi =
    new Vector <Integer>(10);
Vector <Float> vf =
    new Vector <Float>(10);
vn.fillFrom(vi,0);
    // Ok since Vector <Integer> <: Vector <+Number>
vn.fillFrom(vf,20);
    // Ok since Vector <Float> <: Vector <+Number>

As shown in the above listing (Listing[14]), in VPT we can declare the
type of the first formal argument of method fillFrom to be Vector<+X>
which would allow typing of the last two calls to \textit{fillFrom} in the listing.

Replacing the definition shown in Listing[14] with the equivalent one in Covariant-FGJ, shown in Listing[15]:

\begin{lstlisting}[language=Java]
Listing 15: class Vector in Covariant-FGJ

class ROVector < +X extends Object> {
    private X[] ar;

    ROVector(int size){ ar=new X[size]; }

    int size(){ return ar.length; }

    X getElementAt(int i){ return ar[i]; }
}

class Vector <X> extends ROVector <X> {

    void setElementAt(X t, int i){ ar[i]=t; }

    // gets elements from v
    void fillFrom(ROVector <X> v, int start)
    {
    }
\end{lstlisting}
Vector <Number> vn =
    new Vector <Number>(10);
Vector <Integer> vi =
    new Vector <Integer>(10);
Vector <Float> vf =
    new Vector <Float>(10);
vn.fillFrom(vi,0);
// Ok since Vector <Integer> <: ROVector <Integer> <: ROVector <Number>
vn.fillFrom(vf,20);
// Ok since Vector <Float> <: ROVector <Float> <: ROVector <Number>

As we see in the example above (Listing[15]), the expressions in vari-
ant parametric types examples are also type-able in Covariant-FGJ without
loss of expressiveness. Here we assigned type \( ROVector(X) \) (equivalent to \( Vector(+X) \) of VPT) to the first formal argument of method \( fillFrom \) and as in the case of VPT, this allowed us to type the last two lines of the listing. This typing was possible because we could use point-wise subtyping and covariant subtyping and transitivity to conclude \( Vector(Integer) <: ROVector(Number) \) and similarly \( Vector(Float) <: ROVector(Number) \).

We will examine one more example, shown in Listing[16]. This is an example of a nested parametric type taken from the variant parametric types paper [1]. We modify the method \( fillFromVector \) in class \( Vector \) to the equivalent one in Covariant-FGJ (shown in Listing[17]):

Listing 16: fillFromVector method from VPT paper

```java
class Vector <X> {

    void fillFromVector(Vector <+Vector <+X>> vv, int pos)
    {
    }

}

Vector <Vector <Integer>> vvi =
    new Vector <Vector <Integer>> >(1);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Vector <Vector <Float>> vvf = 
   new Vector <Vector <Float>>(1);

Vector <Number> vn =
   new Vector <Number>(10);

vn.fillFromVector(vvi,0);
// Ok since
// Vector <Vector <Integer>> <: 
// Vector <+Vector <+Number>>

vn.fillFromVector(vvf,0);
// Ok since
// Vector <Vector <Float>> <: 
// Vector <+Vector <+Number>>

}

Listing 17: fillFromVector method in Covariant-FGJ

class Vector <X> extends ROVector <X> {

   void fillFromVector(ROVector <ROVector <X>> vv,

   }

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
int pos)
{
}
}

Vector <Vector <Integer>> vvi =
    new Vector <Vector <Integer>>(1);
Vector <Vector <Float>> vvf =
    new Vector <Vector <Float>>(1);
Vector <Number> vn =
    new Vector <Number>(10);

vn.fillFromVector(vvi,0);
// Ok since
// Vector <Vector <Integer>>
// <: ROVector <Vector <Integer>>
// <: ROVector <ROVector <Integer>>
// <: ROVector <ROVector <Number>>
// and that is true since
// Vector <Integer>
// <: ROVector <Integer>
A major criticism of definition-site variance, is that its applicability is limited [1]. The limitation is due to the fact that in practice, type variables in class definitions occur naturally in positions that renders the class unsuitable for both covariance and contra-variance. One such class is shown in Listing[18]:

Listing 18: criticism of definition-site variance

class C <X> {  
    X x;
}
C (X x) { super(); this.x = x; }

X get() { return x; }
void set(X x) { this.x = x; }

The type parameter X of class C of Listing[18] above cannot be declared covariant. If X were allowed to be covariant, field x and method set could not be allowed, because X appears as type of x, a non-private field and as type of method set’s argument. X cannot be declared contra-variant either because it appears as type of x, a non-private field and as the return type of method get.

The solution suggested by [1], which incidently is used by Covariant-FGJ, involves re-writing such classes as read-only and write-only classes and then extending both versions in yet another class.

In the case of class C of Listing[18], we would need to divide C into ROC (for read-only C) class and WOC (for write-only C). We would then define class C as extending both of these classes. Classes ROC and WOC each extend a common base class NonRWC (for non read nor write C). The Listing[19] shows all the details (the example is in pseudo-Java that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
supports multiple inheritance and also allows variance annotations similar to Covariant-FGJ):

Listing 19: read-only and write-only classes

class NonRWC <X> {
    protected X x;

    NonRWC (X x) { this.x = x; }
}

class ROC <+X> extend NonRWC<X>{ // covariant in X

    ROC (X x) { super(x); }

    X get() { return x; }
}

class WOC <–X> extend NonRWC<X>{ // contra-variant in X

    WOC (X x) { super(x); }

    void set(X x) { this.x = x; }
}
It’s clear that the suggested solution does not scale up very well. First of all, in order to support both covariance and contra-variance, we would need some sort of multiple inheritance as the example above shows. Secondly, as the number of type arguments in a class definition increases, the number of classes needed is also increased; since we need to cover all different combinations of read-only and write-only type variables. As an example of this explosion in the number of combinations, see below how a class $C$ with two type parameters may need as many as 16 classes. Class $C$ is shown in Listing[20] and a number of different combinations of read-only and write-only classes for class $C$ is shown in the following listing (Listing[21]):

Listing 20: class C with two type parameters

```java
class C <X, Y> {
    // ...
}
```
X f;
Y g;

C (X f, Y g) { super(); this.f = f; this.g = g; }

X getF() { return f; }
Y getG() { return g; }

void setF(X f) { this.f = f; }
void setG(Y g) { this.g = g; }

Listing 21: too many combinations

class NonRWC <X, Y> {
    protected X f;
    protected X g;
    NonRWC (X f, Y g) { super(); this.f = f; this.g = g; }
}

class RoFAndWoGInC <+X, −Y> extend NonRWC<X,Y>{
```java
RoFAndWoGInC (X f, Y g) { super(f, g); }

X getF() { return f; }
void setG(Y g) { this.g = g; }
}

class RoFAndRoGInC <+X, +Y> extend NonRWC<X,Y>{
    RoFAndRoGInC (X f, Y g) { super(f, g); }
    X getF() { return f; }
    Y getG() { return g; }
}

class WoFAndWoGInC <-X, -Y> extend NonRWC<X,Y>{
    WoFAndWoGInC (X f, Y g) { super(f, g); }
    void setF(X f) { this.f = f; }
    void setG(Y g) { this.g = g; }
}

class WoFAndRoGInC <-X, +Y> extend NonRWC<X,Y>{
```

51
Since covariance is considered more widely applicable and useful, Covariant-FGJ does not deal with contra-variance and therefore not only it doesn’t need
to support multiple inheritance, it does not need write-only classes either. Additionally, we believe that in practice, we rarely need to support all combinations of read-only and write-only classes, hence the exponential explosion in the number of classes needed [1], is only a theoretical worst-case scenario. Our assumption is that since definition-site variance supports “intentional subtyping”, a class designer typically will not want to provide a class to cover all cases, just in case they are needed. Instead a class designer, will provide for one or two specific cases based on the high-level design and modeling of the application; a class that is needed to be both covariant in all its type arguments and contra-variant in all its type arguments, should indeed be very rare.

5 Featherweight Generic Java (FGJ)

Parametric polymorphism or generics was recently added to the Java programming language after years in the making. Addition of parametric polymorphism which for an statically-typed object-Oriented language usually means parameterized classes/methods, resulted in a much more expressive and powerful type-system. As part of the Java Community Process, com-
peting proposals to add generics to the Java programming language were considered and the winner was finally declared to be GJ [13]; current JDK (Java Development Kit) from Sun Microsystems is based on GJ.

GJ’s type system has been modeled and studied by a core calculus called Featherweight Generic Java (FGJ) [2]. FGJ extended another formal language called Featherweight Java (FJ) [2] mainly by adding parametric types to FJ. FJ is a non-imperative and extremely simplified version of Java where not only most of the advanced features of Java such as reflection, packages and concurrency are not considered, but also some of the less advanced futures such as assignment are removed.

Since assignment is not an option in FJ, object’s fields are initialized through constructors once and for all; they cannot be reset at any other time. Although not including assignment in FJ, restricts FJ programs to be “functional” (as opposed to Java programs in full Java that are “imperative”), this does not affect the type-safety proofs for both FJ and FGJ in any significant way [2]. To this end, features such as type inference for generic method invocations were left out of FGJ because of the way they interact with assignment and make the type-safety proofs significantly harder. Type inference for generic method invocations involves inferring the actual type
arguments of a call to a polymorphic method when they are not explicitly specified (this kind of type inference was possible in GJ and now in full Java).

These features were eliminated to make a type-safety proof of the resulting type-system simple enough to do and to understand, “while still capturing the essence of the soundness argument for the full Java language” [2]. The simplicity of FJ and FGJ is also the reason a multitude of extensions to it, are seen in recent years, including modified type-safety proofs to account for the extensions of either language. It is remarkable to note that FJ and FGJ, while being compact and small (e.g. assignment is not supported), do support the following non-trivial features of object-oriented languages: mutually recursive class definitions, object creation via “new”, field access and method invocations, method overriding, subtyping and casting and finally method recursion through “this” [2]. Furthermore, these features are supported naturally as language primitives in contrast to other more traditional formal models where object-oriented features are encoded as a typed lambda calculus or even a denotational model [14].
5.1 Run-time semantics in FGJ

The run-time semantics of FGJ are formalized using “small-step operational semantics”. This style of semantics describes the behavior of the language at run-time by an abstract machine. The abstract machine defines states and transitions from states to other states. The states of an abstract machine for a language correspond to the different terms of the language and the transitions correspond to evaluation or computation relations defined on terms. The computation relations for FGJ (see section 9) are of the form $t \rightarrow t'$ and mean that the term $t$ can take a step of computation and become $t'$. As is the case for FGJ, computation relations are sometimes divided into the computation proper relations and “congruence relations”. Congruence relations describe where the evaluation is to be done while computation relations do the actual work of evaluating a term.

The computation relations for FGJ are defined directly on the terms of the calculus. During the computation, actual type parameters are carried along inside the expressions, even though they are not used during computation. For example in the “GR-Field” computation rule:
the generic types $\overline{T}$ and $N$ could be made non-generic and the computation
relation would not change at all.

This makes it possible for implementations to remove the type parameters
without any adverse effect on the computation rules. When type information
related to type parameters are removed in an implementation, erasure se-
mantics are being used in that implementation. Implementations could also
decide to represent parameterized types that are present in the computation
relations for FGJ without removing their type parameters first. This style
of implementation is said to use the type-passing semantics. For example,
current Java compiler implementations use the erasure-semantics style and
compile generic code to an erased and non-generic version, that is compatible
with current implementations of Java virtual machines.

Computation relations for Covariant-FGJ are basically the same as those
for FGJ; however although FGJ is meant to support two different imple-
mentation styles, Covariant-FGJ is only meant to support the type-passing
semantics – in fact as we will see below, Covariant-FGJ cannot support era-
sure semantics.
Although the run-time semantics (the computation relations) of FGJ allow for two different implementation styles, we expect the result of evaluating an expression directly, to be the same as erasing the expression first and then evaluating it. In other words, running a program should compute the same result, whether we use one implementation style or the other. This and similar expectations are formalized in theorems and corollaries in [2], although we will be referring to a specific theorem called the "commutativity with erasure" theorem in this paper. This theorem states that "The erasure of a reduction of an expression e will, after some further reduction, produce an expression identical to some reduction of the erasure of e" [2].

5.2 dcast predicate in FGJ

To insure that commutativity with erasure result of FGJ holds for downcasts, FGJ uses a predicate dcast in its typing rule for downcasts. This predicate insures that the result of the cast will be the same at run time, whether we use the type-passing style or the type-erasure style. Since type annotations are removed in erasure semantics, a downcast of an expression e of (static) type T to a generic type C(S), written (C(S))e is allowed only if all type arguments S 'contribute' to decide whether C(S) is a subtype of T [15].
As shown in Listing[22] below, since the actual type parameter $C$ of type $Cell(C)$ does not contribute to $Cell(C)$ being a subtype of $Object$ (put differently the subtyping relation is not point-wise), the downcast cannot be allowed. If it were allowed the downcast could have failed in the type-passing style but would have succeeded in the erasure style.

Listing 22: dcast predicate disallows downcast

```java
class Cell<X> {}

// type—passing style i.e. no erasure
Object obj = new Cell<D>();

(Cell<C>)obj // fails

// erasure style (after erasure)
Object obj = new Cell();

(Cell)obj // succeeds
```

As shown in Listing[23] below however, since the actual type parameter $C$ of type $Pair(C,C)$ contributes to $Pair(C,C)$ being a subtype of $Cell(C)$.
(the subtyping relation is point-wise), the downcast is allowed. As seen in
the example the evaluation of the downcast in both the type-passing style
and the erasure style succeeds.

Listing 23: dcast predicate allows downcast

class Pair<X,X> extends Cell<X> {}  

// type—passing style i.e. no erasure
Cell<C> cell = new Pair<C,C>();

(Pair<C,C>)cell // succeeds

// erasure style (after erasure)
Cell cell = new Pair();

(Pair)cell // succeeds

5.3 dcast and Covariant-FGJ

Having established that dcast is a necessity, is Covariant-FGJ compatible
with dcast? FGJ designers came up with the dcast rule to insure that the
erasure semantics and the type-carrying semantics would either both fail or both succeed at run-time when a downcast is attempted. Given \( e \) has static type \( D(T) \) and we want to downcast as in \( (C(S))e \), dcast insured that \( C(S) \) is the only subtype of \( D(T) \) of the form \( C(\ldots) \). Clearly, then if we want to allow covariant subtyping, dcast rule is a problem e.g. Assuming a class \( C \) is defined to be covariant in \( X \) (\texttt{class C(X) \{\}}\) and \( U <: V \), then by the covariant subtyping rule, we have \( C(U) <: C(V) \). Now in a downcast expression \( (C(U))e \), where \( e \) has static type \( C(V) \), dcast says that \( C(U) \) must be the only subtype of \( C(V) \) of form \( C(\ldots) \) which is equivalent to saying that \( U \) must be the unique subtype of \( V \). Even if it could be checked that \( U \) is the unique subtype of \( V \), this puts severe limitations on downcasting between covariant subtypes and makes covariant subtyping of little use. In short, in Covariant-FGJ, we had to either support erasure semantics in addition to type-carrying semantics and as a consequence have a very restricted form of covariant subtyping or to support full covariant subtyping (without any artificial restrictions) and drop erasure semantics. We chose the latter for the obvious reason that support for full covariant subtyping was the whole purpose of covariant-FGJ.
5.4 Stupid casts in FGJ

There are three different casting type rules in FGJ. In an upcast, the static type of an expression is a subtype of the type it is being cast to; in a downcast, the type that an expression is being cast to, is a subtype of the type of that expression. The third typing rule for casts is for casts that are called stupid casts since in such casts, the static type of an expression and the type it is being cast to are unrelated.

In FGJ such casts are not allowed before any evaluation rules have been applied but they do arise as intermediate terms (resulting from evaluating other terms) nonetheless and therefore they need to be typed. To emphasize the special nature of stupid casts, the hypothesis “stupid warning” is added to the premisses of GT-SCast, the typing rule for stupid casts in FGJ.

The reason stupid casts are included in FGJ has to do with the way type soundness is proved for FGJ. Type soundness proof for FGJ is done by means of subject reduction (see section 11.1 below) for a small-step semantics and the proof for the case of GT-DCast specifically needs to consider expressions containing stupid casts since a downcast expression may evaluate to a stupid cast (see proof of this case in [2]). This specific typing case for downcasts – GT-DCast – is added to FGJ because of the need for the dcast predicate.
and the fact that FGJ strives to keep its commutativity with erasure result for casts through it.

5.5 Covariant-FGJ cast typing rules

Since Covariant-FGJ is not concerned with erasure semantics and $dcast$ specifically, stupid casts do not need to be considered at all. Therefore there is no need to keep three different casting type rules as in FGJ. Covariant-FGJ has a single typing rule for casts.

6 Covariant-FGJ: Main definitions

To make Covariant-FGJ's syntax and semantics self-contained and complete, we include not only the modifications made to FGJ to obtain Covariant-FGJ, but also definitions from FGJ that don't change at all in the new extended calculus (Covariant-FGJ).

We mostly use the same notational conventions as in FGJ with minor modifications. The metavariables $A, B, C, D, \text{ and } E$ range over class names; $f$ and $g$ over field names; $m$ over method names; $x$ over variables; $d$ and $e$ range over expressions; $L$ ranges over class declarations; $K$ ranges over
constructor declarations; and $M$ ranges over method declarations; $X$, $Y$ and $Z$ range over type variables; $S$, $T$, $U$ and $V$ range over types; and $N$, $P$, and $Q$ over nonvariable types (any type that is not a type variables); We write $\bar{f}$ as shorthand for a possibly empty sequence $f_1, \ldots, f_n$ (and similarly for $\bar{c}$, $\bar{e}$, etc.) and write $\bar{M}$ as shorthand for $M_1, \ldots, M_n$. We write $\bar{X}$ as shorthand for $X_1, \ldots, X_n$ (and similarly for $\bar{T}$, $\bar{N}$, etc.) and write $\bar{X} \triangleleft \bar{N}$ for $X_1 \triangleleft N_1, \ldots, X_n \triangleleft N_n$ (the symbol $\triangleleft$ is abbreviation for keyword extends).

We assume sequences of type variables contain no duplicate names. We also assume that the set of variables includes the special variable this, which cannot be used as the name of an argument to a method.

We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\bar{x}$ is written $\#(x)$, the keyword extends is written $\triangleleft$ and the keyword return is written $\uparrow$.

We abbreviate operations on pairs of sequences in the obvious way, writing for example "$\bar{T} \bar{f}$" for "$T_1 f_1, \ldots, T_n f_n$", where $n$ is the length of $\bar{T}$ and $\bar{f}$.

We assume a fixed class table $CT$, which is a mapping from class names $C$ to class declarations $L$. A program is a pair $(CT, e)$ of a class table and an expression. The class table satisfies a set of sanity rules: (1) $CT(C) = \text{class } C \ldots$ for every $C \in \text{dom}(CT)$; (2) $\text{Object} \notin \text{dom}(CT)$; (3) for every
class name $C$ (except $\text{Object}$) appearing anywhere in $CT$, we have $C \in \text{dom}(CT)$; (4) subtyping is the reflexive and transitive closure of the relation between two class names in the $\text{extends}$ relation induced by $C(X \triangleleft N) \triangleleft N$ in class definitions and of the relation between different instantiations of class $C$, if $C$ is defined to be covariant in at least one type parameter (see section[ 6.1] below).

6.1 Syntax of Covariant-FGJ

$$L ::= \text{class } C(X;Y \triangleleft N) \triangleleft N \{ \overline{Vf}; \ K \ M \}$$

$$N ::= C(S; T)$$

$$S ::= X | N$$

$$T ::= X | N$$

$$M ::= (X \triangleleft N) \ T \ m \ (\overline{T} \ x) \ (\uparrow e; \}$$

$$K ::= C(\overline{T} \ f) \ \{ \ \text{super}(\overline{f}); \ this.\overline{f} = \overline{f}; \}$$

$$e ::= x \ | \ e.f \ | \ e.m(\overline{T})(\overline{e}) \ | \ new \ N(\overline{e}) \ | \ (N)e$$

A class declaration $L$ has its name $C$, formal type arguments $X;Y$, their bounds $\overline{N}$, supertype $N$, field declaration $\overline{Vf}$, a constructor declaration $K$,
and method declarations $\overline{M}$. As in FGJ, fields of a superclass may not be redeclared. In other words, shadowing of the fields which such redeclaration creates is not allowed.

The semicolon in the formal type arguments is significant as it delimits syntactically, the covariant type variables $\overline{X}$ and non-covariant type variables $\overline{Y}$.

Whenever we don't care to distinguish covariant type variables from non-covariant type variables, we write as in FGJ, $\overline{X}$ as shorthand for $X_1, \ldots, X_n$ (and similarly for $\overline{T}$, $\overline{N}$, etc.) and write $\overline{X} \triangleleft \overline{N}$ for $X_1 \triangleleft N_1, \ldots, X_n \triangleleft N_n$. We write $\overline{X}; \overline{Y} \triangleleft \overline{N}$ for $X_1 \triangleleft N_1, \ldots, X_m \triangleleft N_m, Y_1 \triangleleft N_{m+1}, \ldots, Y_n \triangleleft N_{m+n}$ for when we do care.

The use of the semicolon in this way, forces covariant type parameters to appear first in the type parameter list of a class declaration statement, and the non-covariant type parameters to appear second. This is arguably restrictive and makes for very un-intuitive syntax. However, it was preferred to the alternative of using annotations and defining a new mapping from type parameter lists (per class definition) to covariant/non-covariant type parameters (FGJ already defines two finite mappings called “environments”: $\Gamma$ from variables to types and $\Delta$ from type variables to nonvariable types).
Annotations have been the traditional syntax for this purpose and a ‘+’ annotation typically is prepended to the covariant type parameter e.g. \( C(+X_1 \triangleleft N_1, X_2 \triangleleft N_2) \) to specify \( X_1 \) as being the first (and only) covariant parameter. In this way, no artificial ordering need be imposed on the type parameters at the expense of having to define additional mappings e.g. \( \Omega(\overline{X}) = X_1 \) for covariant type parameters and \( \Pi(\overline{X}) = X_2 \) for non-covariant type parameters. As mentioned before we occasionally drop the semicolon and use a single parameter list, when we don’t need to distinguish between covariant type variables and non-covariant ones. Although to be formal the semicolon is not optional unless both covariant and non-covariant type argument lists are empty.

A non-variable type is a class name \( C \) followed by type parameters \( \overline{S};\overline{T} \) enclosed by \(<>\). A type is either a non-variable type or a type variable. The semicolon has the same meaning as in the class definition syntax above and we again drop the semicolon when we do not care to distinguish between covariant and non-covariant actual type parameters.

A method declaration \( M \) can also have formal type arguments \( \overline{X} \), with their bounds \( \overline{N} \).

A constructor declaration shows how to initialize the fields of an instance
of a class $C$. Its form is completely determined by the instance variable declarations of $C$ and its superclasses: it must take exactly as many parameters as there are instance variables, and its body must consist of a call to the superclass constructor to initialize its fields from the parameters $\tilde{g}$, followed by an assignment of the parameters $\tilde{f}$ to the new fields of the same names declared by $C$. These constraints are actually enforced by the typing rules.

An expression is a variable, field access, method invocation, object constructor, or typecast. As in FGJ, method invocation needs explicit type parameters for the polymorphic method.

In Covariant-FGJ as in FGJ, type variables cannot be bounds, the instantiated type of an object constructor, and the target of typecast expression. These restrictions were necessary in FGJ, to ensure the language was independent of which of two styles of implementation were used: erasure semantics where type parameter information is erased and is not available at run-time and type-carrying semantics where type parameters are available at run-time.

Covariant-FGJ as we’ll see below assumes a type-carrying semantics only and therefore does not need to have the same restrictions as FGJ; however to keep the changes in Covariant-FGJ to a minimum and concentrate on
Covariant-FGJ's main objective i.e. allowing covariant subtyping in a type-safe manner, these restrictions are kept.

7 Auxiliary Definitions

7.1 Field lookup

\[ \text{field}(\text{Object}) = \bullet \]  \hfill (F-OBJECT)

\[ \text{CT}(C) = \text{class } C(\overline{X} \triangleleft \overline{N}) \triangleleft N \{ \overline{V} \overline{f}; \ K \overline{M} \} \]

\[ \text{fields}((\overline{T} \overline{X})N) = \overline{U} \overline{g} \]

\[ \text{fields}(C(\overline{T})) = \overline{U} \overline{g}, (\overline{T} \overline{X})V \overline{f} \]  \hfill (F-CLASS)

7.2 Method type lookup

\[ \text{CT}(C) = \text{class } C(\overline{X} \triangleleft \overline{N}) \triangleleft N \{ \overline{V} \overline{f}; \ K \overline{M} \} \]

\[ (\overline{Z} \triangleleft \overline{P}) U \ m (\overline{U} \overline{x}) \{ \uparrow e_0; \} \in \overline{M} \]

\[ \text{mtype}(m, C(\overline{T})) = [\overline{T}/\overline{X}][(\overline{Z} \triangleleft \overline{P})\overline{U} \rightarrow U) \]  \hfill (MT-CLASS)

\[ \text{CT}(C) = \text{class } C(\overline{X} \triangleleft \overline{N}) \triangleleft N \{ \overline{V} \overline{f}; \ K \overline{M} \} \]

\[ m \text{ is not defined in } \overline{M} \]

\[ \text{mtype}(m, C(\overline{T})) = \text{mtype}(m, [\overline{T}/\overline{X}]N) \]  \hfill (MT-SUPER)
7.3 Method body lookup

\[
\text{CT}(C) = \text{class } C(X \triangleleft N) \triangleleft N \ldots
\]

\[
\frac{(\overline{Z} \triangleleft \overline{P}) U m \overline{(U \overline{x})} \{ \uparrow e_0; \} \in \overline{M}}{\text{mbody}(m(\overline{V}), C(\overline{T})) = (\overline{x}, [\overline{T}/\overline{X}, \overline{V}/\overline{Z}]e_0)} \quad \text{(MB-CLASS)}
\]

\[
\text{CT}(C) = \text{class } C(X \triangleleft N) \triangleleft N \ldots
\]

\[
\frac{m \text{ is not defined in } \overline{M}}{\text{mbody}(m(\overline{V}), C(\overline{T})) = \text{mbody}(m(\overline{V}), [\overline{T}/\overline{X}]N)} \quad \text{(MB-SUPER)}
\]

We use most of the same auxiliary definitions as in FJG for the typing and evaluation rules: \textit{fields}, \textit{mtype}, \textit{mbody} and additionally \textit{cov}.

The fields of a nonvariable type \( N \), written \textit{fields}(\( N \)), are a sequence of corresponding types and field names, \( \overline{T} \overline{f} \). The type of the method invocation \( m \) at nonvariable type \( N \), written \textit{mtype}(\( m, N \)), is a type of the form \( (\overline{Z} \triangleleft \overline{P})U \rightarrow U \). In this form, the variables \( \overline{X} \) are bound in \( \overline{N}, \overline{U} \), and \( U \), and we regard \( \alpha \)-convertible ones as equivalent; application of type substitution \([\overline{T}/\overline{X}]\) is defined in the customary manner. The body of the method invocation \( m \) at nonvariable type \( N \) with type parameters \( \overline{V} \), written \textit{mbody}(\( m(\overline{V}), N \)), is a pair, written \( \overline{x}.e \), of a sequence of parameters \( \overline{x} \) and an expression \( e \).
7.4 Cov

\[ \Delta \vdash \text{cov}(\overline{X}, X) \quad \text{(Cov-Var)} \]

\[ \overline{X} \notin \text{FV}(\overline{T}) \]

\[ \Delta \vdash \text{cov}(\overline{X}, \overline{S}) \]

\[ \Delta \vdash \text{cov}(\overline{X}, C(\overline{S}; \overline{T})) \quad \text{(Cov-Main)} \]

The \textit{cov} predicate is used in the typing rule GT-Class and judges if the type of a field of a class and its superclass is “Ok” (see below) with respect to the covariant type parameters of the class. Field types of the form \( X \), are trivially covariant in \( \overline{X} \). Field types of the form \( C(\overline{S}; \overline{T}) \), are covariant in \( \overline{X} \), if \( \overline{T} \) contain no \( X \), and if \( \overline{S} \) are covariant in \( \overline{X} \).

7.4.1 Motivation for Cov

Having established that to insure type-safety of covariant-subtyping certain restrictions on class definitions are needed in general, are they needed even for specific cases like FGJ?

First of all, notions of access control implemented via keywords such as \textit{private} and \textit{public} don’t exist in FGJ. Such features, seen in most of today’s
popular object-Oriented languages have been removed from FGJ to make it as small and simple as it is. Therefore all fields in FGJ are public and writable. This fact forces us to alter the first restriction above to read: For a class declared to be covariant in the type parameter \( X \), \( X \) should not appear as type of any fields. However since in FGJ, instance variables can only be assigned to, via constructors i.e. no field assignments are permitted, could this restriction be removed all-together? After all the sort of type-unsafe problems seen in the examples above, could no longer occur, if updates to fields were only done through monomorphic constructors. Although, this assertion is correct, however, examining the proposal from first principles, points out some other problems as shown in Listing 24.

Listing 24: allowing updates to instance variables

```java
class C<X;Y> {
    B<X> f;
    Y g;

    C(B<X> f, Y g)
    {
        super(); this.f = f; this.g = g;
    }
}
```

72
void set(Y y)
{
}

class CIntStr extends C<Integer, String> {
    CIntStr(B<Integer> f, String g)
    {
        super(f, g);
    }

    void set(String y)
    {
    }
}

class B<X> {
    X f1;
    B(X f1) { super(); this.f1 = f1; }

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In the preceding example—using syntax of Covariant-FGJ—a covariant subtyping relationship is induced: \( \text{CIntStr} <: \text{C(Object, String)} \); however, \( \text{fields(CIntStr)} = \text{B(Integer)} \) and \\
\( \text{fields(C(Object, String))} = \text{B(Object)} \) and note that \\
\( \text{B(Integer)} <: \text{B(Object)} \) does not hold. So the type of field of the subtype has no relation to the type of field of supertype. Since a subtype should be able to always masquerade as the supertype, let’s see if we can get an inconsistency in the example above:

Listing 25: unsafe update to instance variables

```java
((C<Object;String>)
    new CIntStr(new B<Integer>(5))).f.set("hello");
```

In the last expression, \( f \) has static type \( \text{B(Object)} \), so \( \text{set("hello")} \) types ok, but \( f \)'s run-type is \( \text{B(Integer)} \), so the method

Listing 26: instantiated definition of method set

```java
B<Integer> set(Integer i) { return new B<Integer>(i); }
```

74
of \( B(Integer) \) will be invoked and a \( String \) will be passed to the constructor that expects an \( Integer \).

This shows that simply allowing the covariant type variable as the type of fields with no restrictions is not type-safe. We need to ensure substitutability of a subtype for its supertype is achieved in a type-safe manner, at the same time as being able to use the fact FGJ is a functional language (no assignments).

We allow the covariant type variable to appear as the type of the fields only if we can insure the field type of the subtype is a subtype of the field type of the super-type. In the above example, we would allow a field declaration such as \( X f \) in class \( C \) and also \( B(X) f \), if \( B \) is a class that is covariant in \( X \) as well. In the first case: \( fields(CIntStr) = Integer f \), \( fields(C(Object, String)) = Object f \) and \( Integer <: Object \), so this will be allowed.

In the second case if \( B \) is covariant in \( X \), for example:

Listing 27: allowing updatable instance variables with conditions

```java
class B<X> { 
    X f; // This is ok

    B(X f) { super(); this.f = f; }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
// set method is NOT allowed anymore

// B<X> set(X f) { return new B<X>(f); }

We have: $\text{fields(CIntStr)} = B\langle\text{Integer}\rangle f$ and

$\text{fields(C(Object, String)}) = B\langle\text{Object}\rangle f$ and note that

$B\langle\text{Integer}\rangle <: B\langle\text{Object}\rangle$ does hold now. So this is allowed as well. Now
the unsafe invocation of method set cannot happen anymore because, set
cannot be declared a method of $B$ if it has arguments of the covariant type
parameter $X$.

How about extending the read-only class $B$ and adding the method set
in the writable version:

Listing 28: allowing updatable instance variables with conditions is safe

class BInt extends B<X> { 

BInt(Integer f)
{
    super(f);
}

B<Integer> set(Integer f)

76
8 Typing

As in FGJ, an environment $\Gamma$ is a finite mapping from variables to types, written $\bar{x} : \bar{T}$; a type environment $\Delta$ is a finite mapping from type variables to nonvariable types, written $\bar{X} : \bar{N}$, which takes each type variable to its bound. The main judgments of the type system consist of one for subtyping $\Delta \vdash S <: T$, one for type wellformedness $\Delta \vdash T ok$, and one for typing $\Delta ; \Gamma \vdash e \in T$. We abbreviate a sequence of judgments in the obvious way: $\Delta \vdash S_1 <: T_1, \ldots, \Delta \vdash S_n <: T_n$ to $\Delta \vdash \bar{S} <: \bar{T}$; $\Delta \vdash T_1 ok, \ldots, \Delta \vdash T_n ok$ to $\Delta \vdash \bar{T} ok$; and $\Delta; \Gamma \vdash e_1 \in T_1, \ldots, \Delta; \Gamma \vdash e_n \in T_n$ to $\Delta; \Gamma \vdash \bar{e} \in \bar{N}$.
8.1 Bound of type

\[ \text{bound}_\Delta(X) = \Delta(X) \]

\[ \text{bound}_\Delta(N) = N \]

The bound of a type variable is the non-variable type in the type environment \( \Delta \) for that type variable. The bound for a non-variable type is itself. These definitions insure that the bound of a type variable is always a nonvariable type.

8.2 Well-Formed Types

\[ \Delta \vdash \text{Object ok} \quad \text{(WF-OBJECT)} \]

\[ X \in \text{dom}(\Delta) \]
\[ \Delta \vdash X \text{ ok} \quad \text{(WF-VAR)} \]

\[ CT(C) = \text{class } C(\overline{X} \bowtie \overline{N}) \bowtie N \{ \overline{V} \overline{f}; \overline{K} \overline{M} \} \]

\[ \Delta \vdash \overline{T} \text{ ok} \quad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{N} \quad \text{(WF-CLASS)} \]
\[ \Delta \vdash C(\overline{T}) \text{ ok} \]

78
In the context of a type environment $\Delta$, $\text{Object}$ is trivially defined to be well-formed (this is irrespective of the specific $\Delta$ and it is always true), a type variable $X$ is also trivially well-formed, if it is declared to be in the domain of $\Delta$, and finally if a class $C$ is defined in the class table, with type parameters $\overline{X}$ and upper bounds $\overline{N}$, then instantiations of the class $C$ with well-formed types $\overline{T}$ are also well-formed provided that $\Delta \vdash \overline{T} : \langle \overline{T}/\overline{X} \rangle \overline{N}$.

### 8.3 Subtyping

\[
\Delta \vdash T <: T \quad \text{(S-REFL)}
\]

\[
\frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U} \quad \text{(S-TRANS)}
\]

\[
\Delta \vdash X <: \Delta(X) \quad \text{(S-VAR)}
\]

\[
CT(C) = \text{class } C(\overline{X} \triangleleft \overline{N}) \triangleleft N \{\overline{V} \overline{f}; \overline{K} \overline{M}\} \quad \text{(S-CLASS)}
\]

\[
\Delta \vdash C(\overline{T}) : \langle \overline{T}/\overline{X} \rangle \overline{N}
\]

\[
CT(C) = \text{class } C(\overline{X}; \overline{V} \triangleleft \overline{N}) \triangleleft N \{\overline{V} \overline{f}; \overline{K} \overline{M}\}
\]

\[
\Delta \vdash \overline{S} <: \overline{T} \quad \text{(S-Cov)}
\]

\[
\Delta \vdash C(\overline{S}; \overline{U}) : C(\overline{T}; \overline{U})
\]
The subtype relation is defined to be reflexive and transitive. A type variable is always a subtype of the bound defined for it, in its type environment namely $\Delta$. A well-formed instantiation of a class $C$ that is defined in the class table, with types $\overline{T}$ is a subtype of $C$’s supertype $N$, where all free variables of $N$, are replaced by $\overline{T}$. If class $C$ is in the domain of the class table, and declared to be covariant in type parameters $\overline{X}$, then one well-formed instantiation of it with types $\overline{S}; \overline{U}$ is a subtype of another well-formed instantiation of it with types $\overline{T}; \overline{U}$, provided that $\overline{S} <: \overline{T}$ in the type environment $\Delta$.

9 Computation

\[
\text{fields}(N) = \overline{T} \overline{f} \quad \text{(GR-FIELD)}
\]

\[
\begin{align*}
(new\ N(\overline{e})).f_i & \rightarrow e_i \\
\end{align*}
\]

\[
\text{mbody}(m(\overline{V}), N) = (\overline{x}, e_0) \quad \text{(GR-INVK)}
\]

\[
\begin{align*}
(new\ N(\overline{e})).m(\overline{V})(\overline{d}) & \rightarrow [\overline{d}/\overline{x}, new\ N(\overline{e})/\text{this}]e_0 \\
\end{align*}
\]

\[
0 \vdash N <: P \quad \text{(GR-CAST)}
\]

\[
(P)(new\ N(\overline{e})) \rightarrow new\ N(\overline{e})
\]
9.1 Congruence

\[
\begin{align*}
e_0 & \rightarrow e'_0 \\
e_0.f & \rightarrow e'_0.f
\end{align*}
\]
(GRC-FIELD)

\[
\begin{align*}
e_0 & \rightarrow e'_0 \\
e_0.m(\overline{T})(\overline{e}) & \rightarrow e'_0.m(\overline{T})(\overline{e})
\end{align*}
\]
(GRC-INV-RECV)

\[
\begin{align*}
e_i & \rightarrow e'_i \\
e_0.m(\overline{V})(\ldots,e_i,\ldots) & \rightarrow e'_0.m(\overline{V})(\ldots,e'_i,\ldots)
\end{align*}
\]
(GRC-INV-ARG)

\[
\begin{align*}
e_i & \rightarrow e'_i \\
{\text{new}} \ N(\ldots,e_i,\ldots) & \rightarrow {\text{new}} \ N(\ldots,e'_i,\ldots)
\end{align*}
\]
(GRC-NEW-ARG)

\[
\begin{align*}
e_0 & \rightarrow e'_0 \\
(N)e_0 & \rightarrow (N)e'_0
\end{align*}
\]
(GRC-CAST)

10 Typing Rules

\[
\begin{align*}
\Delta;\Gamma \vdash x \in \Gamma(x) \\
\Delta;\Gamma \vdash e_0 \in T_0 \\
{\text{fields}}(\text{bound}_{\Delta}(T_0)) = \overline{T} \overline{f} \\
\Delta;\Gamma \vdash e_0.f_i \in T_i
\end{align*}
\]
(GT-VAR)

(GT-FIELD)

81
\[
\Delta; \Gamma \vdash e_0 \in T_0
\]

\[
mtype(m, \text{bound}_\Delta(T_0)) = (\overline{V} \circ \overline{P})\overline{U} \rightarrow U
\]

\[
\Delta \vdash \overline{V} \ \text{ok}
\]

\[
\Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P}
\]

\[
\Delta; \Gamma \vdash \overline{e} \in \overline{S}
\]

\[
\Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}
\]

\[
\Delta; \Gamma \vdash e_0.m(\overline{V})(\overline{e}) \in [\overline{V}/\overline{Y}]\overline{U}
\]

\[
\Delta \vdash N \ \text{ok} \quad \text{fields}(N) = \overline{T} \overline{f}
\]

\[
\Delta; \Gamma \vdash \overline{e} \in \overline{S} \quad \Delta \vdash \overline{S} <: \overline{T}
\]

\[
\Delta; \Gamma \vdash \text{new } N(\overline{e}) \in N
\]

\[
\Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash N \ \text{ok}
\]

\[
\Delta; \Gamma \vdash (N)e_0 \in N
\]
\[ \Delta = \overline{X} : \overline{N}, \overline{Z} : \overline{P} \quad \Delta \vdash T, T, \overline{P} \text{ ok} \]
\[ \Delta; \overline{x} : T, \text{this} : C(\overline{X}) \vdash \epsilon_0 \in S \quad \Delta \vdash S : < : T \]

\[ \text{CT}(C) = \text{class } C(\overline{X} \triangleleft \overline{N}) \triangleleft N \{ \overline{V} \overline{f}; \ K M \} \]

\[ \text{if } \text{mtype}(m, N) = (\overline{Z} \triangleleft \overline{Q}) \overline{U} \rightarrow U, \text{ then} \]
\[ \overline{P}, T = [\overline{Z}/\overline{Z}](\overline{Q}, \overline{U}) \text{ and } \Delta \vdash T : [\overline{Z}/\overline{Z}]U \]

\[ (\overline{Z} \triangleleft \overline{P}) T m (\overline{T} \overline{x}) \{ \uparrow \epsilon_0; \} \text{ OK IN } C(\overline{X} \triangleleft \overline{N}) \]

\[ \overline{X}; \overline{Y} : \overline{N}, \vdash \overline{N}, \overline{N}, \overline{V} \text{ ok} \quad \text{fields}(N) = \overline{U} \overline{g} \]
\[ \overline{M} \text{ OK IN } C(\overline{X}; \overline{Y} \triangleleft \overline{N}) \]

\[ K = C(\overline{U} \overline{g}, \overline{V} \overline{f}) \{ \text{super}(\overline{g}); \ \text{this.}\overline{f} = \overline{f}; \} \]
\[ \Delta \vdash \text{cov}(\overline{X}, \overline{U}) \quad \Delta \vdash \text{cov}(\overline{X}, \overline{V}) \]

\[ \text{if } \text{mtype}(m, C(\overline{X}; \overline{Y})) = (\langle \overline{Z} \triangleleft \overline{P} \overline{W} \rightarrow W_0 \rangle) \text{ then} \]
\[ X_i \notin \text{FV}(\overline{W}) \text{ and } X_i \notin \text{FV}(\overline{P}) \text{ and } \Delta \vdash \text{cov}(\overline{X}, W_0) \]

\[ \text{CT}(C) = \text{class } C(\overline{X}; \overline{Y} \triangleleft \overline{N}) \triangleleft N \{ \overline{V} \overline{f}; \ K M \} \text{ OK} \]

### 11 Properties of Reduction Semantics

#### 11.1 Type Soundness

Type-soundness or type-safety is a relation between the static and dynamic semantics of a language. Static type safety guarantees that the dynamic se-
mantics of the language and the static semantics “match”. In other words, the “static guarantees” that the type system can provide, will not be broken at run-time. A type-soundness proof is usually expressed through two theorems, typically called the preservation and the progress theorems. The preservation theorem also known as “subject reduction property”, states that a well-typed term $e$, preserves its type, even after taking a step of evaluation. In languages where subtyping exists the preserved type of the term is allowed to be a subtype of the original type. The progress theorem expresses that if a well-typed term $e$ goes through a step of evaluation – according to the reductions rules – it won’t be “stuck”. The result of the evaluation will be either a value, meaning no more evaluation rules apply, or an evaluation step can be taken [16].

Covariant-FGJ programs enjoy subject reduction and progress properties exactly like programs in FGJ.

**Theorem 11.1** [Subject reduction] If $\Delta; \Gamma \vdash e \in T$ and $e \rightarrow e'$, then $\Delta; \Gamma \vdash e' \in T'$, for some $T'$ such that $\Delta \vdash T' <: T$.

**Proof:** See below.
Theorem 11.2  [Progress] Suppose e is a well-typed expression.

(1) If e includes new $N_0(\bar{e}).f$ as a sub-expression, then $\text{fields}(N_0) = \overline{T_f}$
and $f \in \overline{f}$.

Proof: By induction on the derivation of $\Delta; \Gamma \vdash e \in T$. We only show the directly related case.

Case GT-Field:

$$\Delta; \Gamma \vdash e_0 \in T_0$$

$$\text{fields(bound}_\Delta(T_0)) = \overline{T_f}$$

$$\Delta; \Gamma \vdash e_0.f_i \in T_i$$

$e = e_0.f_i$ and $T = T_i$

If $e_0$ is an object constructor, it must be of the form “new $T_0(\bar{e})$” since $\Delta; \Gamma \vdash e_0 \in T_0$. The conclusion follows from $\text{fields(bound}_\Delta(T_0)) = \overline{S_f}$. If $e_0$ is not an object constructor, the case is trivial by applying the induction hypothesis to subexpressions of $e_0$.

(2) If e includes new $N_0(\bar{e}). m(\overline{V})(\overline{d})$ as a sub-expression, then $\text{mbody}(m(\overline{V}), N_0) = (\overline{x}, e'_0)$ and $\#(\overline{x}) = \#(\overline{d})$. 

85

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Proof: By induction on the derivation of $\Delta; \Gamma \vdash e \in T$. We only show the directly related case.

Case GT-Invk:

$$\Delta; \Gamma \vdash e_0 \in T_0$$

$$\text{mtype}(m, \text{bound}_\Delta(T_0)) = (Y \triangleleft P)U \rightarrow U$$

$$\Delta \vdash \bar{V} \text{ ok}$$

$$\Delta \vdash \bar{V} <: [\bar{V}/Y]P$$

$$\Delta; \Gamma \vdash \bar{e} \in \bar{S}$$

$$\Delta \vdash \bar{S} <: [\bar{V}/Y]U$$

(Provable by induction on the derivation of $\Delta; \Gamma \vdash e_0 \in T_0$)

$$\Delta; \Gamma \vdash e_0.m(\bar{V})(\bar{e}) \in [\bar{V}/Y]U$$

$$e = e_0.m(\bar{V})(\bar{e}) \text{ and } T = [\bar{V}/Y]U$$

If $e_0$ is an object constructor, it must be of the form “new $T_0(\bar{e})$” since $\Delta; \Gamma \vdash e_0 \in T_0$. Since $\text{bound}_\Delta(T_0) = T_0$, we have

$$\text{mtype}(m, T_0) = (Y \triangleleft P)U \rightarrow U \text{ and } T_0 \text{ must be of form } C(T).$$

Now depending on whether $m \in \overline{M}$ where

$$CT(C) = \text{class } C(X \triangleleft \overline{N}) \triangleleft N \{ \overline{V}; f; \overline{K}; \overline{M} \}, \text{ either MT-Class or MT-Super is the last case used in the derivation.}$$
**Sub-Case MT-Class:**

\[ CT(C) = \text{class } C(\overline{\mathcal{X}} \triangleleft \overline{N}) \triangleleft N \{ \overline{V}; \overline{K}; \overline{M} \} \]

\[
\frac{\langle \overline{Y}; \overline{P} \rangle U' \, m \, (\overline{U}; \overline{x}) \{ \uparrow e_0^* ; \} \in \overline{M}}{mtype(m, C(\overline{T})) = ((\overline{Y} \triangleleft \overline{P}) U \rightarrow U)} \quad \text{(MT-Class)}
\]

where \([\overline{T}/\overline{X}](\langle \overline{Y}; \overline{P} \rangle U \rightarrow U^* ) = (\overline{Y} \triangleleft \overline{P}) \overline{U} \rightarrow U.

Clearly, \#(\overline{x}) = \#(\overline{U}) = \#(\overline{U}). From GT-Invk, \#(\overline{S}) = \#(\overline{U}) and 
\#(\overline{a}) = \#(\overline{S}), so \#(\overline{a}) = \#(\overline{U}) therefore \#(\overline{x}) = \#(\overline{a}). Since premises
of MT-Class also imply MB-Class, we have that

\[ mbody(m, C(\overline{T})) = (\overline{x}, [\overline{T}/\overline{X}][\overline{V}/\overline{Y}^*])e_0^* \]

Letting \( e_0' = [\overline{T}/\overline{X}][\overline{V}/\overline{Y}^*]e_0^* \) finishes the case.

**Sub-Case MT-Super:** In this case we have \( C \trianglelefteq D \) for some \( D \).

Then by MT-Super,

\[ mtype(m, C(\overline{T})) = mtype(m, D(\overline{T}^*)) \]

By MT-Class, we have

\[ CT(D) = \text{class } D(\overline{\mathcal{X}} \triangleleft \overline{N}^*) \triangleleft N^* \{ \overline{V}; \overline{F}; \overline{K}^* \} \]

\[
\frac{\langle \overline{Y}; \overline{P}^* \rangle U^* \, m \, (\overline{U}; \overline{x}) \{ \uparrow e_0^* ; \} \in \overline{M}^*}{mtype(m, D(\overline{T})) = ((\overline{Y} \triangleleft \overline{P}) U \rightarrow U^*)} \quad \text{(MT-Class)}
\]

where \([\overline{T}/\overline{X}](\langle \overline{Y}; \overline{P}^* \rangle U^* \rightarrow U^* ) = (\overline{Y} \triangleleft \overline{P}) \overline{U} \rightarrow U^*.

Clearly, \#(\overline{x}) = \#(\overline{U}) = \#(\overline{U}). Also from GT-Invk, \#(\overline{a}) = \#(\overline{U}).
so \( \#(\bar{x}) = \#(\bar{d}) \). Since premises of MT-Class also imply MB-Class, we have that \( \text{mbody}(m(\bar{V}), D(\bar{T}^e)) = (\bar{x}, [\bar{T^*}/\bar{X}]/\bar{V}/\bar{Y}^*|e_1^*) \).

Letting \( e'_00 = [\bar{T^*}/\bar{X}]/\bar{V}/\bar{Y}^*|e_1^* \) finishes the case.

Lemma 11.3 [Weakening] Suppose \( \Delta, \bar{X}; \bar{Y} \vdash \bar{N} \vdash \bar{N} \text{ ok} \) and \( \Delta \vdash U \text{ ok} \).

1. If \( \Delta \vdash S :< T \), then \( \Delta, \bar{X}; \bar{Y} \vdash \bar{N} \vdash S :< T \).

2. If \( \Delta \vdash S \text{ ok} \), then \( \Delta, \bar{X}; \bar{Y} \vdash \bar{N} \vdash S \text{ ok} \).

3. If \( \Delta; \Gamma \vdash e \in T \), then \( \Delta; \Gamma, x : U \vdash e \in T \), and \( \Delta, \bar{X}; \bar{Y} \vdash \bar{N}; \Gamma \vdash e \in T \).

Proof: As for FGJ, each of them is proved by straightforward induction on the derivation of \( \Delta \vdash S :< T \) and \( \Delta \vdash S \text{ ok} \) and \( \Delta; \Gamma \vdash e \in T \). We only need to add the case of S-Cov to the proof of the first sub-lemma.

Case S-Cov: We need to show if \( \Delta \vdash C(\bar{S}; \bar{U}) :< C(\bar{T}; \bar{U}) \), then \( \Delta, \bar{X}; \bar{Y} \vdash \bar{N} \vdash C(\bar{S}; \bar{U}) :< C(\bar{T}; \bar{U}) \). However it’s easy to see that adding the environment \( \bar{X}; \bar{Y} \vdash \bar{N} \) to the assumptions, has no bearing on the fact that \( C(\bar{S}; \bar{U}) :< C(\bar{T}; \bar{U}) \).
Lemma 11.4 If $\Delta \vdash E(\tilde{S}; \tilde{U}) \ll D(\tilde{T}; \tilde{V})$ and $D \not\preceq C$ and $C \not\preceq D$, then $E \not\preceq C$ and $C \not\preceq E$.

Proof: There are two cases to consider:

Case ($E \neq D$) : It is easy to see that $\Delta \vdash E(\tilde{S}; \tilde{U}) \ll D(\tilde{T}; \tilde{V})$ implies $E \leq D$. Proof by contradiction:

Sub-Case ($E \leq C$) : We also have $E \leq D$. There are only two cases since for every class $F$, there is only one class $G$ such that $F \preceq G$.

Sub-Case ($C \leq D$) : Which contradicts the assumption.

Sub-Case ($D \leq C$) : Which contradicts the assumption.

Sub-Case ($C \leq E$) : We also have $E \leq D$. But this implies $C \leq D$ which is a contradiction.

Case ($E = D$) : Need to show If $\Delta \vdash E(\tilde{S}; \tilde{U}) \ll E(\tilde{T}; \tilde{V})$ and $D \not\preceq C$ and $C \not\preceq D$, then $D \not\preceq C$ and $C \not\preceq D$, but that is easily seen to be true.
Lemma 11.5 [Type substitution preserves subtyping]

If \( \Delta_1, \overline{X} \vdash \overline{N}, \Delta_2 \vdash S \vdash T \) and \( \Delta_1 \vdash \overline{U} \vdash [\overline{U}/\overline{X}]\overline{N} \) with \( \Delta_1 \vdash \overline{U} \) ok and none of \( \overline{X} \) appearing in \( \Delta_1 \), then \( \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S \vdash [\overline{U}/\overline{X}]T \).

Proof: By induction on the derivation of \( \Delta_1, \overline{X}; \overline{Y} \vdash \overline{N}, \Delta_2 \vdash S \vdash T \). All cases except S-Cov are similar to FGJ; we show this additional case only.

Case S-Cov:

\[
CT(C) = \text{class } C(\overline{X}; \overline{Y} \triangleleft \overline{N}) \triangleleft N \{\overline{V}; \overline{f}; \overline{K}; \overline{M}\}
\]

\[
\Delta \vdash \overline{S} \vdash \overline{T} \quad \text{(S-Cov)}
\]

Need to show \( \Delta_1, [\overline{U}/\overline{X}; \overline{Y}]\Delta_2 \vdash [\overline{U}/\overline{X}; \overline{Y}]C(\overline{S}; \overline{W}) \vdash [\overline{U}/\overline{X}; \overline{Y}]C(\overline{T}; \overline{W}) \) or

\[
\Delta_1, [\overline{U}/\overline{X}; \overline{Y}]\Delta_2 \vdash C([\overline{U}/\overline{X}; \overline{Y}]\overline{S}; \overline{W}) \vdash C([\overline{U}/\overline{X}; \overline{Y}]\overline{T}; \overline{W}) . \quad \text{By induction hypothesis,}
\]

\[
\Delta_1, [\overline{U}/\overline{X}; \overline{Y}]\Delta_2 \vdash [\overline{U}/\overline{X}; \overline{Y}]\overline{S}; \overline{W} \vdash [\overline{U}/\overline{X}; \overline{Y}]\overline{T}; \overline{W} . \quad \text{Letting}
\]

\[
[\overline{U}/\overline{X}; \overline{Y}]\overline{S}; \overline{W} = \overline{S'}; \overline{W'} \text{ and } [\overline{U}/\overline{X}; \overline{Y}]\overline{T}; \overline{W} = \overline{T'}; \overline{W'}, \text{ we get}
\]

\[
\Delta_1, [\overline{U}/\overline{X}; \overline{Y}]\Delta_2 \vdash \overline{S'}; \overline{W'} \vdash \overline{T'}; \overline{W'} . \quad \text{Applying S-Cov, we get}
\]

\[
\Delta_1, [\overline{U}/\overline{X}; \overline{Y}]\Delta_2 \vdash C(\overline{S'}; \overline{W'}) \vdash C(\overline{T'}; \overline{W'}) \quad \text{which is equivalent to}
\]
\( \Delta_1, [U/X; Y] \Delta_2 \vdash C([U/X; Y]S; [U/X; Y]W) <: C([U/X; Y]T)[U/X; Y]W \)

or finally \( \Delta_1, [U/X; Y] \Delta_2 \vdash [U/X; Y]C(S; W) <: [U/X; Y]C(T; W) . \)

Lemma 11.6 [Type substitution preserves type well-formedness]

If \( \Delta_1, X <: \overline{N}, \Delta_2 \vdash T \) ok and \( \Delta_1 \vdash \overline{U} <: [U/X]\overline{N} \) with \( \Delta_1 \vdash \overline{U} \) ok and none of \( X \) appearing in \( \Delta_1 \), then \( \Delta_1, [U/X] \Delta_2 \vdash [U/X]T \) ok.

Proof: By induction on the derivation of \( \Delta_1, X <: \overline{N}, \Delta_2 \vdash T \) ok, with a case analysis on the last rule used. The proof is the same as the proof for FGJ.

Lemma 11.7 Suppose \( \Delta_1, X <: \overline{N}, \Delta_2 \vdash T \) ok and \( \Delta_1 \vdash \overline{U} <: [U/X]\overline{N} \) with \( \Delta_1 \vdash \overline{U} \) ok and none of \( X \) appearing in \( \Delta_1 \), then

\( \Delta_1, [U/X] \Delta_2 \vdash \text{bound}_{\Delta_1, [U/X] \Delta_2}([U/X]T) <: [U/X](\text{bound}_{\Delta_1, [U/X] \Delta_2} (T)) . \)

Proof: The proof is the same as the proof for FGJ.

91
Lemma 11.8 If $\Delta \vdash \text{cov}(X, V)$ and $\Delta \vdash \overline{S} <: \overline{T}$, then

$$\Delta \vdash \overline{S/X}V <: \overline{T/X}V.$$ 

Proof: By straightforward induction on the derivation of $\Delta \vdash \text{cov}(X, V)$.

Case Cov-Var:

$$\Delta \vdash \text{cov}(X, X)$$  \hspace{1cm} (Cov-Var)

We need to show that if $\Delta \vdash \text{cov}(X, X)$ and $\Delta \vdash \overline{S} <: \overline{T}$,
then $\Delta \vdash \overline{S/X}X <: \overline{T/X}X$.

Sub-Case $(X \in \overline{X})$: Since in this case $\overline{S/X}X = S_i$ and $\overline{T/X}X = T_i$,
we only need to show $\Delta \vdash S <: T$, which is already known.

Sub-Case $(X \notin \overline{X})$: Since in this case $\overline{S/X}X = X$ and $\overline{T/X}X = X$,
we only need to show $\Delta \vdash X <: X$, which is already known by S-Refl.

Case Cov-Main:

$$\overline{X} \notin FV(\overline{T})$$

$$\Delta \vdash \text{cov}(\overline{X}, \overline{S})$$  \hspace{1cm} (Cov-Main)

$$\Delta \vdash \text{cov}(\overline{X}, C(\overline{S}; \overline{T}))$$

92
We need to show that If $\Delta \vdash \text{cov}(\overline{X}, C(S; T))$ and $\Delta \vdash \overline{V} \triangleleft \overline{W}$, then
$\Delta \vdash [\overline{V}/X]C(S; T) \triangleleft [\overline{W}/X]C(S; T)$. Since $\overline{X} \notin \text{FV}(\overline{T})$, this becomes
$\Delta \vdash C([\overline{V}/X]S; T) \triangleleft C([\overline{W}/X]S; T)$. By induction hypothesis, since
$\Delta \vdash \text{cov}(\overline{X}, S)$ and $\Delta \vdash \overline{V} \triangleleft \overline{W}$, then
$\Delta \vdash [\overline{V}/X]S \triangleleft [\overline{W}/X]S$. By $S$-Cov, we have that
$\Delta \vdash C([\overline{V}/X]S; T) \triangleleft C([\overline{W}/X]S; T)$.

Lemma 11.9 If $\Delta \vdash C(S; W) \triangleleft C(T; W)$, then $\text{fields}(C(S; W)) = STf$, $\text{fields}(C(T; W)) = TFt$ and
$\Delta \vdash \overline{S} \triangleleft \overline{T}$.

Proof: By F-Class,
$\text{fields}(C(S; W)) = \text{fields}([S/X; W/Y]N), [S/X; W/Y]Vf$, if $CT(C) =$ class $C(X; Y \triangleleft N) \triangleleft N \{V f; \ldots\}$.

By straight forward induction on the derivation of $\text{fields}([S/X; W/Y]N)$, it can be shown that
$\text{fields}([S/X; W/Y]N) = [S/X; W/Y]\text{fields}(N)$, if N is a non-variable type. So if $\text{fields}(N) = \overline{Ug}$, we have
$\text{fields}(C(S; W)) = [S/X; W/Y]\overline{Ug}, [S/X; W/Y]Vf = [S/X; W/Y](\overline{Ug}, Vf)$.

Since by GT-Class, $\Delta \vdash \text{cov}(\overline{X}, U)$, $\Delta \vdash \text{cov}(\overline{X}, V)$ and also that
$\Delta \vdash \overline{S}; \overline{W} \triangleleft \overline{T}; \overline{W}$ by lemma 11.8, we have

93
Lemma 11.10  If $\Delta \vdash S <: T$ and $\text{fields}(\text{bound}_\Delta(T)) = \overline{T} \overline{f}$, then $\text{fields}(\text{bound}_\Delta(S)) = \overline{S} \overline{g}$ and $S_i <: T_i$ and $g_i = f_i$ for all $i \leq \#(\overline{f})$.

Proof: By straightforward induction on the derivation of $\Delta \vdash S <: T$.

Case S-Trans:

$$\frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U} \quad \text{(S-Trans)}$$

Need to show if $\Delta \vdash S <: U$ and $\text{fields}(\text{bound}_\Delta(U)) = \overline{U} \overline{u}$, then $\text{fields}(\text{bound}_\Delta(S)) = \overline{S} \overline{g}$ and $\Delta \vdash S_i <: U_{t_i}$ and $s_i = u_i$ for all $i \leq \#(\overline{u})$.

By induction hypothesis, if $\text{fields}(\text{bound}_\Delta(S)) = \overline{S} \overline{g}$ and $\text{fields}(\text{bound}_\Delta(T)) = \overline{T} \overline{t}$ and $\text{fields}(\text{bound}_\Delta(U)) = \overline{U} \overline{u}$, then $\Delta \vdash S_i <: T_{t_i}$ and $s_i = t_i$ for all $i \leq \#(\overline{t})$ and also $\Delta \vdash T_{t_i} <: U_{t_i}$ and $t_i = u_i$ for all $i \leq \#(\overline{u})$.

By S-Trans, we have $\Delta \vdash S_i <: U_{t_i}$ and $s_i = u_i$ for all $i \leq \#(\overline{u})$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Case S-Var:

\[ \Delta \vdash X <: \Delta(X) \quad \text{(S-Var)} \]

Trivial as in FGJ, since \(\text{bound}_\Delta(X) = \text{bound}_\Delta(\Delta(X))\).

Case S-Refl:

\[ \Delta \vdash T <: T \quad \text{(S-REFL)} \]

Trivial since \(\text{fields}(\text{bound}_\Delta(T)) = \text{fields}(\text{bound}_\Delta(T))\).

Case S-Class:

\[
CT(C) = \text{class } C(\overline{X} \triangleleft \overline{N}) \triangleleft N \{ \overline{V} \; \overline{f} ; \; K \; \overline{M} \}
\]

\[ \Delta \vdash C(\overline{T}) <: [\overline{T}/\overline{X}]N \quad \text{(S-CLASS)} \]

Need to show if \(\text{fields}(\text{bound}_\Delta([\overline{T}/\overline{X}]N)) = \text{fields}([\overline{T}/\overline{X}]N) = \overline{U}/\overline{f},\)

then \(\text{fields}(\text{bound}_\Delta(C(\overline{T}))) = \text{fields}(C(\overline{T})) = \overline{S}/\overline{g}\) and \(\Delta \vdash S_i <: U_i\) and \(g_i = f_i\) for all \(i \leq \#(\overline{f})\).

By F-Class, \(\text{fields}(C(\overline{T})) = \text{fields}([\overline{T}/\overline{X}]N), [\overline{T}/\overline{X}]V\overline{v}.\)

So \(\overline{S}/\overline{g} = \overline{U}/\overline{f}, [\overline{T}/\overline{X}]V\overline{v}\), which means that \(S_i = U_i\) and \(g_i = f_i\) for all \(i \leq \#(\overline{f})\).

Case S-Cov:
\[ CT(C) = \text{class } C(X, Y \triangleleft N) \triangleleft N \{ V; \overline{f}; \overline{M} \} \]

\[
\Delta \vdash \overline{S} \triangleleft \overline{T} \quad \text{(S-Cov)}
\]

\[
\Delta \vdash C(S; U) \triangleleft C(T; U)
\]

Need to show if \( \Delta \vdash C(S; U) \triangleleft C(T; U) \) and \( \text{fields}(C(T; U)) = T \overline{f} \), then \( \text{fields}(C(S; U)) = \overline{S} \overline{g} \) and \( \Delta \vdash \overline{S} \triangleleft \overline{T} \), and \( g_i = f_i \) for all \( i \leq \#(\overline{f}) \).

By lemma 11.9 \( \Delta \vdash \overline{S} \triangleleft \overline{T} \). Furthermore in the case of S-Cov, \( \overline{f} = \overline{g} \) for all \( i \leq \#(\overline{f}) \).

**Lemma 11.11** If \( \Delta \vdash T \) ok and \( mtype(m, \text{bound}_\Delta(T)) = (Y \triangleleft \overline{P})U \rightarrow U_0 \), then for any \( S \) such that \( \Delta \vdash S \triangleleft T \) and \( \Delta \vdash S \) ok, we have

\( mtype(m, \text{bound}_\Delta(S)) = (Y \triangleleft \overline{P})U \rightarrow U'_0 \) and \( \Delta, Y \triangleleft \overline{P} \vdash U'_0 \triangleleft U_0 \).

**Proof:** By straightforward induction on the derivation of \( \Delta \vdash S \triangleleft T \) with a case analysis by the last rule used.

**Case S-Refl:**

\[
\Delta \vdash T \triangleleft T \quad \text{(S-Refl)}
\]
Trivial.

**Case S-Var:**

\[
\Delta \vdash X <: \Delta(X) \quad (\text{S-Var})
\]

Need to show if \( \Delta \vdash \Delta(X) \) ok and \( \text{mtype}(m, \text{bound}_\Delta(\Delta(X))) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W_0 \), then \( \text{mtype}(m, \text{bound}_\Delta(X)) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W'_0 \) and

\[
\Delta, \overline{Z} <: P \vdash W'_0 <: W_0.
\]

Since \( \text{bound}_\Delta(X) = \Delta(X) \) and \( \text{bound}_\Delta(\Delta(X)) = \Delta(X) \),

\[
\text{mtype}(m, \text{bound}_\Delta(X)) = \text{mtype}(m, \Delta(X)) = \text{mtype}(m, \text{bound}_\Delta(\Delta(X))) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W_0.
\]

Letting \( W_0 = W'_0 \) finishes the case.

**Case S-Trans:**

\[
\Delta \vdash S <: T \quad \Delta \vdash T <: U \quad (\text{S-Trans})\]

\[
\Delta \vdash S <: U
\]

Need to show if \( \Delta \vdash U \) ok and \( \text{mtype}(m, \text{bound}_\Delta(U)) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W_0 \),
then \( \text{mtype}(m, \text{bound}_\Delta(S)) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W'_0 \) and

\[
\Delta, \overline{Z} <: P \vdash W'_0 <: W_0.
\]

By induction hypothesis, \( \text{mtype}(m, \text{bound}_\Delta(T)) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W''_0 \) and

\[
\Delta, \overline{Z} <: P \vdash W''_0 <: W_0 \quad \text{and}
\]
\[ mtype(m, bound_\Delta(S)) = (\overline{Z} \triangleleft P)\overline{W} \rightarrow W'_0 \] and

\[ \Delta, \overline{Z} \triangleleft P \vdash W'_0 \triangleleft W_0. \]

Applying S-Trans, we get

\[ \Delta, \overline{Z} \triangleleft P \vdash W'_0 \triangleleft W_0. \]

**Case S-Class:**

\[
\begin{align*}
CT(C) &= \text{class } C(\overline{X} \triangleleft N) \triangleleft N \{\overline{V}, \overline{J} \triangleleft K \overline{M}\} \\
\Delta \vdash C(T) \triangleleft [T/X]N
\end{align*}
\]

(S-CLASS)

Need to show if \( \Delta \vdash [T/X]N \) ok and

\[ mtype(m, bound_\Delta([T/X]N)) = (\overline{Z} \triangleleft P)\overline{U} \rightarrow U_0, \]

then

\[ mtype(m, bound_\Delta(C(T))) = (\overline{Z} \triangleleft P)\overline{U} \rightarrow U'_0 \]

\[ \Delta, \overline{Z} \triangleleft P \vdash U'_0 \triangleleft U_0. \]

If \( m \notin \overline{M} \), then by MT-super,

\[ mtype(m, bound_\Delta(C(T))) = mtype(m, C(T)) = mtype(m, [T/X]N) = mtype(bound_\Delta(m, bound_\Delta([T/X]N)) = (\overline{Z} \triangleleft P)\overline{U} \rightarrow U_0. \]

Letting \( U'_0 = U_0 \) finishes the case.

If \( m \in \overline{M} \), by straight forward induction on the derivation of \( mtype \), we can show that \( mtype(m, [T/X]N) = [T/X]mtype(m, N) = [T/X](\overline{Z} \triangleleft P)\overline{U} \rightarrow U''_0 \), if \( mtype(m, N) = (\overline{Z} \triangleleft P)\overline{U} \rightarrow U''_0. \)

By GT-Method, it must be the case that \( (\overline{Z} \triangleleft P) W'_0 m (U'' \overline{x}) \{\ldots\} \in \overline{M} \)

and \( X \triangleleft \overline{N}, Z \triangleleft \overline{P} \vdash W'_0 \triangleleft U''_0. \)

By Lemmas 11.5 and 11.3,
and assuming that \( \overline{X} \) and \( \overline{Z} \) are distinct which means \( \overline{[T/X]Z} = \overline{Z} \) and letting \( \overline{[T/X]P^r} = \overline{P} \), we have \( \Delta, \overline{Z} <: \overline{P} \vdash \overline{[T/X]W'_0} <: \overline{[T/X]U''_0} \). Letting \( \overline{[T/X]U''_0} = U_0 \), we now have \( \Delta, \overline{Z} <: \overline{P} \vdash \overline{[T/X]W'_0} <: U_0 \) and by MT-Class, we have \( \text{mtype}(m, C(\overline{T})) = \overline{[T/X]}((\overline{Z} \triangleleft \overline{P})U^r \rightarrow W'_0) \),

since \( \langle \overline{Z} \triangleleft \overline{P} \rangle W'_0 \models (U^r \overline{x}) \{\ldots\} \in \overline{M} \). Letting \( \overline{U}'_0 = \overline{[T/X]W'_0} \) and \( \overline{U} = \overline{[T/X]U}' \), finishes the case.

**Case S-Cov:**

\[
\begin{align*}
\text{CT}(C) &= \text{class } C(\overline{X}; \overline{Y} \triangleleft \overline{N}) \triangleleft \overline{N} \{\overline{V} \overline{f}; K \overline{M}\} \\
\Delta &\vdash \overline{S} <: \overline{T} \\
\hline
\Delta &\vdash C(\overline{S}; \overline{U}) <: C(\overline{T}; \overline{U})
\end{align*}
\]

(S-Cov)

Need to show if \( \Delta \vdash C(\overline{T}; \overline{U}) \) ok and

\[
\text{mtype}(m, \text{bound}_\Delta(C(\overline{T}; \overline{U}))) = \langle \overline{Z} \triangleleft \overline{P} \rangle \overline{W} \rightarrow W'_0, \text{ then}
\]

\[
\text{mtype}(m, \text{bound}_\Delta(C(\overline{S}; \overline{U}))) = \langle \overline{Z} \triangleleft \overline{P} \rangle \overline{W} \rightarrow W''_0 \quad \text{and}
\]

\( \Delta, \overline{Z} <: \overline{P} \vdash W''_0 <: W_0 \). If \( m \in \overline{M} \), then by MT-Class,

\[
\text{mtype}(m, \text{bound}_\Delta(C(\overline{T}; \overline{U}))) = \overline{[T/X; U/Y]}((\overline{Z} \triangleleft \overline{P})\overline{W} \rightarrow W'_0) = \\
\langle \overline{Z} \triangleleft \overline{P} \rangle \overline{U/Y)W} \rightarrow \overline{[T/X; U/Y]W'}_0, \text{ assuming that } \overline{X}; \overline{Y} \text{ and } \overline{Z} \text{ are distinct which means } \overline{[T/X; U/Y]Z} = \overline{Z} \text{ and letting } \overline{[T/X; U/Y]P^r} = \overline{U/Y}P^r = \overline{P}
\]

(since \( X_i \notin \text{FV}(\overline{P}) \) and \( X_i \notin \text{FV}(\overline{W}) \) by GT-Class).

Also \( \text{mtype}(m, \text{bound}_\Delta(C(\overline{S}; \overline{U}))) = \overline{[S/X; U/Y]}((\overline{Z} \triangleleft \overline{P})\overline{W} \rightarrow W'_0) = 
\]

99
Since by GT-Class, \( \Delta \vdash \text{cov}(X, W'_0) \)
then \( \Delta \vdash [S/X; U/Y]W'_0 <: [T/X; U/Y]W'_0 \). Letting
\[ W_0 = [T/X; U/Y]W'_0 \] and \( W'_0 = [S/X; U/Y]W'_0 \) finishes the case.

If \( m \notin \overline{M} \), then by MT-Super \( \text{mtype}(m, C(T; U)) = \text{mtype}(m, [T/X; U/Y]N) \).

By straight forward induction on the derivation of \( \text{mtype} \), we can show that
\[ \text{mtype}(m, [T/X; U/Y]N) = [T/X; U/Y]\text{mtype}(m, N). \]

Letting \( \text{mtype}(m, N) = (Z \triangleleft P)W \rightarrow W'_0 \), we have \( \text{mtype}(m, C(T; U)) = [T/X; U/Y](\langle Z \triangleleft P \rangle W \rightarrow W'_0) \). We also have, using the same arguments,
\[ \text{mtype}(m, C(S; U)) = [S/X; U/Y](\langle Z \triangleleft P \rangle W \rightarrow W'_0) = \]
\[ (Z \triangleleft P)[U/Y]W \rightarrow [S/X; U/Y]W'_0, \] assuming that \( X; Y \) and \( Z \) are distinct
which means \( [S/X; U/Y]Z = Z \) and letting \([S/X; U/Y]\overline{P} = [U/Y]\overline{P} = \overline{P} \)
(since \( X \notin FV(\overline{P}) \) and \( X \notin FV(\overline{W}) \) by GT-Class). By the same assumptions, we have \( \text{mtype}(m, C(T; U)) = (Z \triangleleft P)[U/Y]W \rightarrow [T/X; U/Y]W'_0. \)

By GT-Class, since \( \Delta \vdash \text{cov}(X, W'_0) \) then
\[ \Delta \vdash [S/X; U/Y]W'_0 <: [T/X; U/Y]W'_0 . \] Letting \( W_0 = [T/X; U/Y]W'_0 \) and \( W'_0 = [S/X; U/Y]W'_0 \) finishes the case.

**Lemma 11.12 [Type substitution preserves typing]** If \( \Delta_1, \overline{X} <: \overline{N}, \Delta_2; \Gamma \vdash e \in T \) and \( \Delta_1 \vdash \overline{U} <: [U/X]\overline{N} \) with \( \Delta_1 \vdash \overline{U} \ ok \) and none of \( \overline{X} \) appearing
in $\Delta_1$, then $\Delta_1, [U/X] \Delta_2; [U/X]\Gamma \vdash [U/X] e \in S$ for some $S$ such that $\Delta_1, [U/X] \Delta_2 \vdash S \triangleleft [U/X]\Gamma$.

**Proof:** By induction on the derivation of $\Delta_1, \overline{\Delta} \triangleleft \overline{\Delta_2}; \Gamma \vdash e \in T$, with a case analysis on the last rule used.

**Case** GT-Var:

\[ \Delta; \Gamma \vdash x \in \Gamma(x) \] (GT-VAR)

Need to show if $\Delta_1, \overline{\Delta} \triangleleft \overline{\Delta_2}; \Gamma \vdash x \in \Gamma(x)$ and $\Delta_1 \vdash \overline{U} \triangleleft [U/X]\overline{\Delta}$ with $\Delta_1 \vdash \overline{U}$ ok and none of $\overline{\Delta}$ appearing in $\Delta_1$, then $\Delta_1, [U/X] \Delta_2; [U/X]\Gamma \vdash [U/X] x \in S$ for some $S$ such that $\Delta_1, [U/X] \Delta_2 \vdash S \triangleleft [U/X]\Gamma(x)$. Since $[U/X] x = x$ and $[U/X]\Gamma(x) = \Gamma(x)$, we have $\Delta_1, [U/X] \Delta_2; [U/X]\Gamma \vdash [U/X] x \in [U/X]\Gamma(x)$. Letting $S = [U/X]\Gamma(x)$ finishes the case.

**Case** GT-Field:

\[ \Delta; \Gamma \vdash e_0 \in T_0 \]

\[ \text{fields}(\text{bound}_\Delta(T_0)) = \overline{T} \overline{f} \] (GT-FIELD)

\[ \Delta; \Gamma \vdash e_0.f_i \in T_i \]

Need to show if $\Delta_1, \overline{\Delta} \triangleleft \overline{\Delta_2}; \Gamma \vdash e_0.f_i \in T_i$ and $\Delta_1 \vdash \overline{U} \triangleleft [U/X]\overline{\Delta}$, then $\Delta_1, [U/X] \Delta_2; [U/X]\Gamma \vdash [U/X] e_0.f_i \in S$ for some $S$ such that
\[\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S < S_0 : [\overline{U}/\overline{X}]T_i.\]

By induction hypothesis, If \(\Delta_1, \overline{X} < S_0, \Delta_2; \Gamma \vdash \epsilon_0 \in T_0\), then
\[\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]\epsilon_0 \in S_0\] for some \(S_0\) such that
\[\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 \subset [\overline{U}/\overline{X}]T_0\] which implies
\[\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(S_0) < bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T_0).\]

By Lemma 11.7,
\[\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T_0) < bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T_0).\]

By S-TRANS, we have
\[\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(S_0) < bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T_0).\] Furthermore, by straightforward induction, we can show that,
\[\operatorname{fields}([\overline{U}/\overline{X}](bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(T_0))) = [\overline{U}/\overline{X}]\overline{T} \overline{f}.\] Now by Lemma 11.10,
\[\operatorname{fields}(bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(S_0)) = \overline{S} \overline{g}\] and we have \(S_j \subset [\overline{U}/\overline{X}]T_j\) and \(f_j = g_j\) \(\forall j \leq \#(\overline{f})\). Applying GT-Field we have \(\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]\epsilon_0, f_i \in S_i\). Letting \(S = S_i(\subset [\overline{U}/\overline{X}]T_i)\) finishes the case.

Case GT-Invk: Proof is the same as in FGJ.

Case GT-New:
\[ \Delta \vdash N \text{ ok} \quad \text{fields}(N) = \overline{T} \overline{f} \]

\[ \Delta; \Gamma \vdash \overline{e} \in \overline{S} \quad \Delta \vdash \overline{S} <: \overline{T} \quad \text{(GT-New)} \]

\[ \Delta; \Gamma \vdash \text{new } N(\overline{e}) \in N \]

Need to show that if \( \Delta_1, \overline{X} <: \overline{N}, \Delta_2; \Gamma \vdash \text{new } N(\overline{e}) \in N \) and

\[ \Delta_1 \vdash \overline{U} <: \overline{[U/X]}N, \text{ then } \Delta_1, [U/X] \Delta_2; [U/X] \Gamma \vdash [U/X](\text{new } N(\overline{e})) \in S \]

for some \( S \) such that

\[ \Delta_1, [U/X] \Delta_2 \vdash \overline{S} <: [U/X]N. \]

By induction hypothesis: \( \Delta_1, [U/X] \Delta_2; [U/X] \Gamma \vdash [U/X] \overline{e} \in \overline{S} \) for \( S' \) such that

\[ \Delta_1, [U/X] \Delta_2 \vdash \overline{S'} <: [U/X]S. \]

By straightforward induction we can show that \( \text{fields}([U/X]N) = [U/X]\overline{T} \overline{f}. \)

We also have by lemma 11.6, \( \Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash [U/X]N \text{ ok}, \) and by lemma 11.5, \( \Delta_1, [U/X] \Delta_2 \vdash [U/X] \overline{S} <: [U/X] \overline{T}. \)

Finally by GT-New,

\[ \Delta_1, [U/X] \Delta_2; [U/X] \Gamma \vdash \text{new } [U/X]N([U/X] \overline{e}) \in [U/X]N \text{ or} \]

\[ \Delta_1, [U/X] \Delta_2; [U/X] \Gamma \vdash \text{new } N([U/X] \overline{e}) \in N. \]

Letting \( S = N \) finishes the case.

**Case** GT-Cast:
![](image)

\[
\Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash N \text{ ok}
\]

\[
\Delta; \Gamma \vdash (N)e_0 \in N
\]

Need to show that if \(\Delta_1, \overline{X} <: \overline{N}, \Delta_2; \Gamma \vdash (N)e_0 \in N\) and

\[
\Delta_1 \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{N}, \text{ then } \Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]((N)e_0) \in S \text{ for some } S \text{ such that } \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S <: [\overline{U}/\overline{X}]T_0.
\]

By induction hypothesis: \(\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0 \in S_0 \text{ for some } S_0 \text{ such that } \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0.\)

Also since \(\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{N} \text{ ok, by lemma 11.6, we also have } \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{N} \text{ ok.}\)

Applying GT-Cast, we get \(\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash ([\overline{U}/\overline{X}]\overline{N})([\overline{U}/\overline{X}]e_0) \in [\overline{U}/\overline{X}]N.\) Letting \(S = [\overline{U}/\overline{X}]N\) finishes the case.

**Lemma 11.13** [Term substitution preserves typing] If \(\Delta; \Gamma, \overline{x} : \overline{T} \vdash e \in T\) and, \(\Delta; \Gamma \vdash \overline{d} \in \overline{S}\) where \(\Delta \vdash \overline{S} <: \overline{T}, \text{ then } \Delta; \Gamma \vdash [\overline{d}/\overline{x}]e \in S \text{ for some } S \text{ such that } \Delta \vdash S <: T.\)

**Proof:** By induction on the derivation of \(\Delta; \Gamma, \overline{x} : \overline{T} \vdash e \in T, \text{ with a case} \)
analysis on the last rule used.

Case GT-Var:

\[ \Delta; \Gamma \vdash x \in \Gamma(x) \]  

(GT-Var)

Need to show if \( \Delta; \Gamma; \overline{x} : \overline{T} \vdash x \in \Gamma(x) \) and \( \Delta; \Gamma \vdash \overline{d} \in S \) and \( \Delta \vdash \overline{S} <: \overline{T} \) then, \( \Delta; \Gamma \vdash [\overline{d}/\overline{x}]x \in S \) such that \( \Delta \vdash S <: \Gamma(x) \).

If \( x \in dom(\Gamma) \) then, the proof is trivial. Since then, \( [\overline{d}/\overline{x}]x = x \) and since \( \Delta; \Gamma; \overline{x} : \overline{T} \vdash x \in \Gamma(x) \). Letting \( S = \Gamma(x) \) finishes the case. On the other hand, if \( x = x_i \) then, we have \( x_i \in T_i \). Letting \( S = S_i \) finishes the case.

Case GT-Field:

\[ \Delta; \Gamma \vdash e_0 \in T_0 \]

\[ \text{fields}(bound_{\Delta}(T_0)) = \overline{T} \overline{f} \]  

(GT-FIELD)

Need to show if \( \Delta; \Gamma; \overline{x} : \overline{W} \vdash e_0, f_i \in T_i \) and, \( \Delta; \Gamma \vdash \overline{d} \in S \) where \( \Delta \vdash \overline{S} <: \overline{W} \), then \( \Delta; \Gamma \vdash [\overline{d}/\overline{x}]e_0, f_i \in S \) for some \( S \) such that \( \Delta \vdash S <: T_i \). By induction hypothesis, \( \Delta; \Gamma \vdash [\overline{d}/\overline{x}]e_0 \in S_0 \) for some \( S_0 \) such that \( \Delta \vdash S_0 <: T_0 \).

By lemma 11.10, \( \text{fields}(bound_{\Delta}(S_0)) = \overline{S} \overline{g} \) such that \( S_j <: T_j \) and \( f_j = g_j \) for all \( j \leq \#(\overline{f}) \). Therefore by GT-Field, \( \Delta; \Gamma \vdash ([\overline{d}/\overline{x}]e_0) . g_j \in S_j \). Letting \( S = S_j \) finishes the case.
Case GT-Invk:

\[
\Delta; \Gamma \vdash e_0 \in T_0
\]

\[
\text{mtype} (m, \text{bound}_{\Delta} (T_0)) = (\overline{Y} < \overline{P}) \overline{U} \rightarrow U
\]

\[
\Delta \vdash \overline{V} \text{ ok}
\]

\[
\Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P}
\]

\[
\Delta; \Gamma \vdash \overline{e} \in \mathcal{S}
\]

\[
\Delta \vdash \mathcal{S} <: [\overline{V}/\overline{Y}]\overline{U}
\]

\[
\Delta; \Gamma \vdash e_0. m (\overline{V})(\overline{e}) \in [\overline{V}/\overline{Y}]U
\]

Need to show that if \(\Delta; \Gamma, \overline{x} : T \vdash e_0. m (\overline{V})(\overline{e}) \in [\overline{V}/\overline{Y}]U\) and, \(\Delta; \Gamma \vdash \overline{d} \in \overline{W}\) where \(\Delta \vdash \overline{W} <: \overline{T}\), then \(\Delta; \Gamma \vdash [\overline{d}/\overline{x}](e_0. m (\overline{V})(\overline{e})) \in W\) for some \(W\) such that \(\Delta \vdash \overline{W} <: [\overline{V}/\overline{Y}]U\).

By induction hypothesis, \(\Delta; \Gamma \vdash [\overline{d}/\overline{x}]e_0 \in S_0\) for some \(S_0\) such that

\(\Delta \vdash S_0 <: T_0\) and \(\Delta; \Gamma \vdash [\overline{d}/\overline{x}] \overline{e} \in \overline{W}\) for some \(\overline{W}\) such that \(\Delta \vdash \overline{W} <: \mathcal{S}\).

By lemma 11.11, \(\text{mtype} (m, \text{bound}_{\Delta} (S_0)) = (\overline{Y} < \overline{P}) \overline{U} \rightarrow U'\) and  

\(\Delta, \overline{Y} <: \overline{P} \vdash U' <: U\). Since \(\Delta \vdash \overline{W} <: \mathcal{S} <: [\overline{V}/\overline{Y}]U\), by GT-Invk, we have \(\Delta; \Gamma \vdash ([\overline{d}/\overline{x}]e_0). m (\overline{V})([\overline{d}/\overline{x}] \overline{e}) \in [\overline{V}/\overline{Y}]U'\), or equivalently

\(\Delta; \Gamma \vdash [\overline{d}/\overline{x}](e_0. m (\overline{V})(\overline{e})) \in [\overline{V}/\overline{Y}]U'.\) By lemma 11.5, since

\(\Delta, \overline{Y} <: \overline{P} \vdash U' <: U\), we have \(\Delta \vdash [\overline{V}/\overline{Y}]U' <: [\overline{V}/\overline{Y}]U\). Letting \(W = [\overline{V}/\overline{Y}]U'\) finishes the case.

106
Case GT-New:

\[
\begin{align*}
&\Delta \vdash N \text{ ok} \quad \text{fields}(N) = \overline{T} \overline{f} \\
&\Delta;\Gamma \vdash e \in \overline{S} \quad \Delta \vdash \overline{S} <: \overline{T} \\
&\Delta;\Gamma \vdash \text{new } N(e) \in N
\end{align*}
\]

(GT-New)

Need to show if \(\Delta;\Gamma, \overline{x} : \overline{W} \vdash \text{new } N(\overline{e}) \in N\) and, \(\Delta;\Gamma \vdash \overline{d} \in \overline{V}\) where \(\Delta \vdash \overline{V} <: \overline{W}\), then \(\Delta;\Gamma \vdash [\overline{d}/\overline{x}](\text{new } N(\overline{e})) \in V\) for some \(V\) such that \(\Delta \vdash V <: N\).

By induction hypothesis, \(\Delta;\Gamma \vdash [\overline{d}/\overline{x}]e \in \overline{S}'\) for some \(\overline{S}'\) such that \(\Delta \vdash \overline{S}' <: \overline{S}\) which implies \(\Delta \vdash \overline{S} <: \overline{T}\). Applying GT-New, we get \(\Delta;\Gamma \vdash [\overline{d}/\overline{x}](\text{new } N(\overline{e})) \in N\). Letting \(V = N\) finishes the case.

Case GT-Cast:

\[
\begin{align*}
&\Delta;\Gamma \vdash e_0 \in T_0 \quad \Delta \vdash N \text{ ok} \\
&\Delta;\Gamma \vdash (N)e_0 \in N
\end{align*}
\]

(GT-Cast)

Need to show if \(\Delta;\Gamma, \overline{x} : \overline{T} \vdash (N)e_0 \in N\) and \(\Delta;\Gamma \vdash \overline{d} \in \overline{S}\) where \(\Delta \vdash \overline{S} <: \overline{T}\) then, \(\Delta;\Gamma \vdash [\overline{d}/\overline{x}]((N)e_0) \in S\) for some \(S\) such that \(\Delta \vdash S <: N\). By induction hypothesis, \(\Delta;\Gamma \vdash [\overline{d}/\overline{x}]e_0 \in S_0\) for some \(S_0\) such that \(\Delta \vdash S_0 <: T_0\). Applying GT-Cast, we get \(\Delta;\Gamma \vdash (N)((\overline{d}/\overline{x})e_0) \in N\). Letting \(S = N\) finishes the case.
Lemma 11.14  If \( mtype(m, C(T)) = (\overline{Y} \triangle F)\overline{U} \rightarrow U \) and 

\[
mbody(m(\overline{V}), C(\overline{T})) = (\overline{x}, e_0) \quad \text{where} \quad \Delta \vdash C(\overline{T}) \text{ ok and } \Delta \vdash \overline{V} \text{ ok and } \\
\Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P} , \text{ then there exist some } N \text{ and } S \text{ such that } \Delta \vdash C(\overline{T}) <: N \\
\text{ and } \Delta \vdash N \text{ ok and } \Delta \vdash S <: [\overline{V}/\overline{Y}]U \text{ and } \Delta \vdash S \text{ ok and } \Delta; \overline{x} : [\overline{V}/\overline{Y}]\overline{U}, \text{ this : } \\
N \vdash e_0 \in S.
\]

Proof: By induction on the derivation of \( mbody(m(\overline{V}), C(\overline{T})) = (\overline{x}, e_0) \) using Lemmas 11.5 and 11.12. The proof is the same as the proof for FGJ.

Proof of Theorem 11.1: By induction on the derivation of \( e \rightarrow e' \). We only show interesting cases.

Case GR-Field:

\[
\begin{align*}
\text{fields}(N) &= \overline{T} \overline{f} \\
\text{(new N(\overline{e})).f}_i &\rightarrow e_i \\
\text{(GR-FIELD)}
\end{align*}
\]

Need to show that if \( \Delta; \Gamma \vdash (\text{new N(\overline{e}))}.f_i \in T_i \) and \( (\text{new N(\overline{e}))}.f_i \rightarrow e_i \), 
then \( \Delta; \Gamma \vdash e_i \in S_i \) for some \( S_i \) such that \( \Delta \vdash S_i <: T_i \).

By GT-Field, \( \Delta; \Gamma \vdash (\text{new N(\overline{e}))}.f_i \in T_i \) and \( \Delta; \Gamma \vdash (\text{new N(\overline{e}))} \in T_0 \) and 
\[
\text{fields(bound}_{\Delta}(T_0)) = \overline{T} \overline{f}.
\]
By GT-New $\Delta; \Gamma \vdash new\ N(\overline{e}) \in N$, so $T_0 = N$ and we also have
$\Delta; \Gamma \vdash \overline{e} \in \overline{S}$ and $\Delta \vdash \overline{S} <: \overline{T}$. In particular, $\Delta; \Gamma \vdash e_i \in S_i$. Letting $S_i = T_i$ finishes the case.

**Case GR-Invk:**

\[
\frac{mbody(m(\overline{V}), N) = (\overline{x}, e_0)}{(new\ N(\overline{e})).m(\overline{V})(\overline{d}) \rightarrow [\overline{d}/\overline{x}, new\ N(\overline{e})/this]e_0} \tag{GR-Invk}
\]

Need to show that if
$\Delta; \Gamma \vdash new\ N(\overline{e}).m(\overline{V})(\overline{d}) \in T$ then, $\Delta; \Gamma \vdash [\overline{d}/\overline{x}, new\ N(\overline{e})/this]e_0 \in T'$
such that $\Delta \vdash T' <: T$.

By GT-Invk we have $\Delta; \Gamma \vdash new\ N(\overline{e}).m(\overline{V})(\overline{d}) \in [\overline{V}/\overline{Y}]U$.

By GT-New $\Delta; \Gamma \vdash new\ N(\overline{e}) \in N$. We also know $\text{bound}_\Delta(N) = N$. By Lemma 11.14, $\Delta; \overline{x} : [\overline{V}/\overline{Y}]U, this : P \vdash e_0 \in S$ for some $P$ and $S$ such that $\Delta \vdash N <: P$ where $\Delta \vdash P \ ok$, and $\Delta \vdash S <: [\overline{V}/\overline{Y}]U$ where $\Delta \vdash S \ ok$.

Then by 11.3 and 11.13, $\Delta; \Gamma \vdash [\overline{d}/\overline{x}, new\ N(\overline{e})/this]e_0 \in T_0$ for some $T_0$ such that $T_0 <: S$. By S-Trans, we have $\Delta \vdash T_0 <: [\overline{V}/\overline{Y}]U$. Letting $T = [\overline{V}/\overline{Y}]U$ and $T' = T_0$ finishes the case.

**Case GR-Cast:**
\[ 0 \vdash N <: P \]

\[ (P)(\text{new } N(\vec{e})) \rightarrow \text{new } N(\vec{e}) \]  

(GR-Cast)

By GT-Cast \( \Delta; \Gamma \vdash (P)(\text{new } N(\vec{e})) \in P \) and by GT-New

\[ \Delta; \Gamma \vdash \text{new } N(\vec{e}) \in N \], and we have \( \emptyset \vdash N <: P \). Letting \( T' = N \) and \( T = P \) finishes the case.

**Case** GRC-Field:

\[ e_0 \rightarrow e'_0 \]  

(GRC-Field)

\[ e_0.f \rightarrow e'_0.f \]

By GT-Field, we have \( e_0.f \in T_i \) (let \( f = f_i \) and \( T = T_i \)), if \( \Delta; \Gamma \vdash e_0 \in T_0 \), and \( \text{fields}(\text{bound}_\Delta(T_0)) = \overline{T} \overline{f} \). By induction hypothesis, \( \Delta; \Gamma \vdash e'_0 \in T'_0 \) for some \( T'_0 \) such that \( \Delta \vdash T'_0 <: T_0 \).

Applying 11.10 we get \( \text{fields}(\text{bound}_\Delta(T'_0)) = \overline{T'} \overline{g} \), and for \( j \leq \#(\overline{f}) \), we have \( g_i = f_i \) and \( T'_i <: T_i \). Therefore by GT-Field, \( \Delta; \Gamma \vdash e'_0.f_i \in T'_0 \).

Letting \( T' = T'_i \) finishes the case.

**Case** GRC-Inv-Recv:

\[ e_0 \rightarrow e'_0 \]  

(GRC-Inv-Recv)

\[ e_0.m(\overline{T})(\vec{e}) \rightarrow e'_0.m(\overline{T})(\vec{e}) \]
By GT-Invk, \( \Delta; \Gamma \vdash e_0.m(\overline{V})(\overline{\varepsilon}) \in [\overline{V}/\overline{Y}]U \) and \( \Delta; \Gamma \vdash e_0 \in T_0 \).

By induction hypothesis, \( \Delta; \Gamma \vdash e_0' \in T_0' \) for some \( T_0' \) such that
\[
\Delta \vdash T_0' <: T_0 .
\]
By \ref{lem:11.14}, we have \( mtype(m, bound_\Delta(T_0')) = (\overline{Y} \triangleq \overline{P})U \rightarrow U' \) and \( \Delta, \overline{Y} <: \overline{P} \vdash U' <: U \) where \( mtype(m, bound_\Delta(T_0)) = (\overline{Y} \triangleq \overline{P})U \rightarrow U \) by GT-Invk. By Lemma \ref{lem:11.5}, we have \( \Delta \vdash [\overline{V}/\overline{Y}]U' <: [\overline{V}/\overline{Y}]U . \) By GT-Invk, \( \Delta; \Gamma \vdash e_0'.m(\overline{V})(\overline{\varepsilon}) \in [\overline{V}/\overline{Y}]U' \). Letting \( T' = [\overline{V}/\overline{Y}]U' \) and \( T = [\overline{V}/\overline{Y}]U \) finishes the case.

**Case** GRC-Cast:

\[
\frac{e_0 \rightarrow e_0'}{(N)e_0 \rightarrow (N)e_0'} \quad \text{(GRC-CAST)}
\]

\( \Delta; \Gamma \vdash (N)e_0' \in T' \) for some \( T' \) and \( \Delta \vdash T_0' <: T_0 \), where
\( \Delta; \Gamma \vdash (N)e_0 \in T \). By GT-Cast, \( T = T' = N \). Letting \( T = N \) and \( T' = N \) finishes the case.

**Case** GRC-Inv-Arg:

\[
\frac{e_i \rightarrow e_i'}{e_0.m(\overline{V})(\ldots, e_i, \ldots) \rightarrow e_0.m(\overline{V})(\ldots, e_i', \ldots)} \quad \text{(GRC-INV-ARG)}
\]

By GT-Invk, \( \Delta; \Gamma \vdash e_0.m(\overline{V})(\ldots, e_i, \ldots) \in [\overline{V}/\overline{Y}]U \) and \( \Delta; \Gamma \vdash \overline{\varepsilon} \in S \) where \( \Delta \vdash S <: [\overline{V}/\overline{Y}]U \). In particular, \( \Delta; \Gamma \vdash e_i \in S_i \) and \( \Delta \vdash S_i <: [V_i/Y_i]U_i \).
By induction hypothesis, $\Delta; \Gamma \vdash e_i' \in S'_i$ for some $S'_i$ such that

$\Delta \vdash S'_i \prec S_i (\prec [V_i/Y_i]U_i)$. By GT-Invk, $e_0.m(\overline{V})(\ldots, e'_i, \ldots) \in [\overline{V}/\overline{Y}]U$.

Letting $T' = [\overline{V}/\overline{Y}]U = T$ finishes the case.

Case GRC-New-Arg:

$e_i \rightarrow e'_i$  \hspace{6em} (GRC-New-Arg)

new $N(..., e_i, \ldots) \rightarrow$ new $N(..., e'_i, \ldots)$

By GT-New, new $N(..., e'_i, \ldots) \in N$. Letting $T' = T = N$ finishes the case.

12 Conclusion

In summary, this paper presents Covariant-FGJ (as an extension to FGJ) and shows a detailed type-safety proof of the resulting type system.

12.1 Summary of Contributions

Although, the ideas behind the access restrictions imposed by Covariant-FGJ to obtain type-safety are not novel, we believe a careful and formal study of how a cross-cutting feature like covariant-subtyping interacts with other features specially parametric polymorphism is important and novel. A detailed
type-safety proof, shows that such interactions implemented by additions and modifications to the language definition (to FGJ), were the right ones and "made sense". Scala, after all has already implemented covariant-subtyping and the necessary access restrictions, but Scala's type system has not yet shown to be sound. On the other hand Eiffel's type system was shown to be unsound [17], years after it was first introduced (Eiffel supported covariance by default). Such experiences show the importance of formal type-safety proofs, even for subsets of real-world languages and specially in the early stages of language development, when new features is proposed.

We have also shown that the known safety restrictions for field accesses may be relaxed for Covariant-FGJ. This weakening of the safety restrictions is achieved via the addition of the "Cov" predicate and the main reason this was possible is the lack of assignment in FGJ and Covariant-FGJ. Although the application of the above contribution is limited to functional settings (real object-oriented programming languages after all do have assignment), we have shown how to integrate "Cov" with other language features in such a way as to insure the type-safety property of the type system is preserved.
12.2 Future Work

With the burden of keeping the “commutativity with erasure” result gone, Covariant-FGJ could be released from other restrictions as well. Specifically, if run-time type information is to be kept, type-dependent additions such as “new X” could be added to the languages. Furthermore, parametric types could be used as first-class types in the style of NextGen [6].

Another rather obvious future work direction would be to add contravariance to Covariant-FGJ (perhaps calling it variant-FGJ) and prove the type safety of such extension. Further research is needed to come up with a satisfactory solution that avoids the difficulty of defining classes that need to be both covariant and contra-variant. One possibility is to use mixins to extend multiple read-only and write-only classes. Scala uses (and NextGen proposes to use) mixins as a way of implementing multiple-inheritance.

References


114


