cJ: Enhancing Java with Safe Type Conditions

Shan Shan Huang David Zook Yannis Smaragdakis

College of Computing, Georgia Institute of Technology

ssh,dzook,yannis@cc.gatech.edu

http://www.cc.gatech.edu/∼ssh/cj

Abstract

cJ is an extension of Java that allows supertypes, fields, and methods of a class or interface to be provided only under some static subtyping condition. For instance, a cJ generic class, `C<P>`, may provide a member method `m` only when the type provided for parameter `P` is a subtype of a specific type `Q`.

From a practical standpoint, cJ adds to generic Java classes and interfaces the ability to express case-specific code. Unlike conditional compilation techniques (e.g., the C/C++ "`#ifdef`" construct) cJ is statically type safe and maintains the modular type-checking properties of Java generic classes: a cJ generic class can be checked independently of the code that uses it. Just like regular Java, checking a cJ class implies that all uses are safe, under the contract for type parameters specified in the class's signature.

As a specific application, cJ addresses the well-known shortcomings of the Java Collections Framework (JCF). JCF data structures often throw run-time errors when an "optional" method is called upon an object that does not support it. Within the constraints of standard Java, the authors of the JCF had to either sacrifice static type safety or suffer a combinatorial explosion of the number of types involved. cJ avoids both problems, maintaining both static safety and conciseness.

1. Introduction

Generic types increase the expressiveness and safety of a programming language. Since the introduction of Java and C#, researchers have worked on adding genericity mechanisms that were subsequently integrated into the base languages themselves [3, 17, 33]. From a language design standpoint, modern genericity mechanisms offer a good tradeoff between expressiveness and separate checkability. For instance, Java generics have limited expressiveness compared to undisciplined mechanisms, such as C++ template classes, yet offer the ability to detect static errors (e.g., type errors) without having to provide a specific type parameter that triggers the error.

This paper proposes cJ: an extension of Java that adds more expressiveness to its genericity mechanism without sacrificing any of the Java type-checking guarantees. Specifically, we add to Java the ability to place type-conditions on methods, fields, or supertypes. This is best illustrated with a small example. Consider the following generic cJ class:


```
class C<X> {
  X xRef;
  ...
  <X extends DataSource>?
  void store() { ... xRef.getConnection() ... }
}
```

In this example, the member method `store` is declared in a type-instantiation of generic class `C`, *only* when the type argument for `X` is a class (or interface) that implements (resp., extends) interface `DataSource`. The `<...>?` syntax is cJ's type-conditional construct. One can read this syntax as "static-if", or just "if". The call `xRef.getConnection()` is well-typed only because type `X` is guaranteed to be a subtype of `DataSource` and, consequently, to provide the `getConnection` method.

cJ is translated by erasure, reducing to regular Java in a backward compatible manner. This allows us to solve a well-recognized problem in the Java Collections Framework (JCF), the standard Java data structures library. Currently, JCF data structures support two main common interfaces (`Collection` and `Map`), regardless of optional behavior, such as whether the data structure is modifiable or not, and whether the data structure has variable size or not. Classes that do not support the corresponding operations throw `UnsupportedOperationExceptions` when the operations are called at run-time. The design of the JCF is an instance of sacrificing static type safety in favor of conciseness. cJ solves this problem, maintaining both type safety and conciseness of expression.

Interestingly, cJ can be thought of as a language that allows cross-cutting [19] at the level of types. cJ type conditions are used to define many implicit types from a single class definition. In the above example, the single definition of class `C` can be thought of as defining the implicit types `C<subtype-of-DataSource>` and `C<not-DataSource>`. Thus, with cJ type conditionals, one can add orthogonal "aspects" or "dimensions" to an existing type hierarchy. Our re-implementation of the JCF provides a vivid demonstration of this feature. Beginning from a simple subtyping hierarchy, we introduce variation based on whether a data structure supports content modification or size variability. The definitions of the various collections (e.g., the `List`, `Collection`, `Map`, and `Set` interfaces) are as simple as in plain Java, yet a much richer type hierarchy is produced by modifying each type with attributes chosen from a separate type hierarchy (with types such as `Modifiable`, `DeleteOnly`, and `Resizable`). Thus, whereas traditional AOP tools allow the expression of cross-cutting features at the level of methods, cJ supports separation of concerns at the type level. Type hierarchies can be specified separately to represent orthogonal concerns, and cJ allows their composition to form richer, derived hierarchies. Writing general code that exploits these derived hierarchies can be done through a natural extension of the Java variance/wildcards mechanism (e.g., "`? extends T`" clauses).

Although Java was chosen as the platform for our ideas, the cJ approach is far from Java-specific. The same programming and type-checking framework can be applied to other languages. Yet Java is a good representative of modern OO languages and integrating with it demonstrates clearly both the benefits and the intricacies of our approach.

Concretely, our work makes the following contributions:

- cJ allows expressing highly variable generic classes concisely. Compared to standard OO mechanisms, cJ allows a single generic class to express the functionality of an exponential number of regular Java classes.
- cJ offers full type safety, analogous to that of the base Java language. A cJ generic class is checked separately from its uses. The type system ensures that the class is type-correct under any consistent combination of outcomes of the type-conditionals.
- Other research work [7, 21, 22, 24] has targeted the problem of type-safe conditional declarations. Nevertheless, many of the past mechanisms are in a simpler context (e.g., no subtyping) or do not allow some of the cJ features (e.g., conditional subtyping). None of the past research dealt with the integration of conditional members and subtypes with (use-site) variance, or provided a backward compatible, erasure-based translation. Overall, cJ is distinguished by its power and its smooth integration in a modern language.
- cJ solves the static type safety issues of the JCF. We are not aware of other language proposals that address this well-publicized need without sacrificing conciseness.

The rest of the paper is organized as follows. We first give an informal introduction to the cJ language extensions. This serves as background for the motivating examples of Section 3 and the JCF case study. In Section 4, we present interesting ways in which the cJ extensions interact with variance in Java. We then analyze the cJ implementation in Section 5. Section 6 formalizes cJ and subsequently we discuss related work (Section 7) and our conclusions.

2. cJ Language Introduction

We next give an informal overview of cJ's syntax and semantics, to prepare the ground for our motivating examples. A formal description of the language is laid out in Section 6.

2.1 cJ Basics and Examples

cJ is a conservative extension of Java—we assume Java 5, with support for generics and variance ("wildcards") [3, 31] in the base language. cJ adds to Java the ability to change a type's structure depending on static type conditions. The language provides a *type-conditional* construct. The following is a simple example showing the use of a type-conditional:

```
class Foo<T> {
  <T extends Bar>?
  int i;
}
```

In the above example, <T extends Bar>? is a type-conditional. The declaration immediately following it, "int i;", exists only if Foo is parameterized by a subtype of Bar.

Type-conditionals can be used for the declaration of class- or interface-level methods and fields, as well as for the declaration of conditional supertypes. For instance, we can have:

```
class Foo<T>
<T extends Serializable>? implements Serializable
{ ... }
```

The above class Foo implements interface Serializable only when it is parameterized by a type that also implements (or extends) Serializable.

Multiple declarations can exist in the same conditional block by surrounding them in < ... >. For instance:

```
class Foo<T> {
  <T extends Bar>?
  < int i;
    void meth(T t) { }
  >
}
```

The above is equivalent to preceding each declaration individually with the type-conditional <T extends Bar>?.

There are two required components to each type-conditional block. The first is the type condition, defined inside "< ... >?". Any syntax that is valid for defining the type parameters of a Java class is valid here as a type condition: the type conditions have the standard F-bounded polymorphism form [3], where a type parameter can be referenced by its own bound, e.g., "T extends I<T>". Note especially that "extends" is used to express all kinds of subtyping constraints (including interface conformance) and that the syntax admits conjunctions of subtyping bounds (e.g., "T extends I<T> & J<T>"), as well as bounding multiple parameters (e.g., "S extends I<S>, T extends J<S>"). Similarly to Java, it is *not* valid for a type parameter to appear by itself on the right hand side of extends (i.e. we cannot place lower bounds on a type parameter). Also note that only type parameters declared at the class/interface level are allowed in type conditions. Polymorphic method type parameters are not allowed. The second required component, the consequent block, immediately follows the type condition. Declarations within this block exist for the enclosing type if and only if the type condition is true, after all type parameters are instantiated. A type-conditioned declaration is syntactically a declaration, hence, type-conditionals can nest.

cJ ensures that all uses of type-conditionals are statically safe. All code should be well-typed under its enclosing type conditions. Furthermore, all uses of class or interface members should be under equivalent or stronger conditions than those employed in the member's declaration. For instance, the following use is legal only if J is a subtype of I:

```
class Foo<T> {
  <T extends I>?
  int i;

  <T extends J>?
  void incI() { i++; } // legal iff J subtypes I
}
```

The following code is also legal, as the type conditions are strengthened by adding conjunctions:

```
class Foo<T,U> {
  <T extends I>?
  int i;

  <T extends I, U extends K>?
  void incI() { i++; } // legal: stronger condition

  <T extends I & J>?
  void decI() { i--; } // legal: stronger condition
}
```

2.2 Restrictions

There are some restrictions that cJ imposes on conditional declarations. These restrictions significantly simplify the translation and interfacing with existing Java code, as we will discuss in Section 5. The rule of thumb is that a cJ class (or interface) should be a legal Java class (interface) if all type conditions are removed.

A cJ class can have at most one extends clause, regardless of whether it is under a type-conditional. Of course, a cJ class can implement multiple interfaces and any of the `implements` clauses can be conditional.

Declarations that are conflicting per the standard Java rules are not allowed, even if their type-conditional conditions are exclusive. For instance, the following is illegal in cJ, even when neither `Baz` nor `Bar` are a subtype of the other:

```
interface IFoo<T> {
  <T extends Bar>?
  void foo(int i);

  <T extends Baz>?
  int foo(int i);  // duplicate definition
}
```

Furthermore, subtypes are required to define conditional methods under equivalent or weaker conditions than conflicting methods in their (possibly conditional) supertypes. For example:

```
interface ISuper<T,U> {
  <T extends Bar>?
  void meth1(int i);

  <T extends Bar>?
  void meth2(Object o);
}

interface ISub<T,U> extends ISuper<T,U> {
  <T extends Bar & Baz>?
  void meth1(int i); // illegal unless Bar subtypes Baz

  void meth2(Object o); // legal: weaker (no condition)
}
```

The above rules extend to the members of conditional supertypes. Their type conditions from the perspective of the subtype is the conjunction of the subtyping and the membership conditions. (E.g., a member under a static condition P in an interface implemented under condition Q should be thought of as being under a condition $P\&Q$ for the purposes of the above discussion.) Our formalism in Section 6 makes this definition precise.

3. cJ Benefits

Having introduced the cJ language, we can now examine some motivating examples. We first discuss a small example that demonstrates how a type-conditional avoids a combinatorial blowup of the number of classes required in a Java application. Then, we examine a specific case study: the Java Collections Framework and its well-known shortcomings with respect to static type safety.

3.1 The Argument for Safety and Conciseness

There are two ways to view the benefits of cJ over regular Java. In Java, when the contents of a class can vary with respect to multiple orthogonal concerns, the programmer can either choose to maintain static type safety and suffer a combinatorial explosion of the number of classes involved, or sacrifice static type safety in order to keep the number of classes manageable. cJ achieves both benefits simultaneously.

The conciseness benefits of cJ are relatively easy to see. When multiple conditionals capture different axes of variability, a cJ generic class corresponds to a hierarchy of many different regular classes. Consider a simple example class `C`:

```
class C<X> {
  <X extends Serializable>?
  public void store() { ... }
  ...
  <X extends Comparable<X>>?
  public X getMin() { ... }
}
```

That is, class `C` supports method `store` only when type parameter `X` is a serializable type. Similarly, `C` supports method `getMin` only when type parameter `X` is a comparable type.

To achieve the same effect with regular Java, the programmer needs to create separate classes that capture all possible combinations. One possibility would be the following class hierarchy:

```
class CommonC<X> {
  ... // the common parts of C
}

class CSer<X extends Serializable> extends CommonC<X> {
  public void store() { ... }
}

class CComp<X extends Comparable<X>> extends CommonC<X> {
  public X getMin() { ... }
}

class CCompSer<X extends Comparable<X> & Serializable>
 extends CSer<X>
{
  public X getMin() { ... }
}
```

The result is four different classes, capturing the same content as the original cJ class. Method code is replicated: `CCompSer` cannot inherit `getMin` from `CComp` because it already has a superclass, `CSer`. Furthermore, `CCompSer` is not a subtype of `CComp`, hence, a `CCompSer` object cannot be used where a `CComp` is expected, even though it supports the required methods of a `CComp`. Such code replication and subtyping problems can be alleviated by using delegation techniques and interfaces, but this may require significant code reorganization, weakening of encapsulation, and explicitly maintaining object identity. For instance, to minimize code length with delegation, the programmer often needs to enable access to members of another class, as well as manually ensure a one-to-one mapping among different sub-objects.

Note that this example deals with only two axes of variability: whether `X` is `Comparable` and whether `X` is `Serializable`. Still, the result is undesirable. In the general case, the number of Java classes required for a faithful emulation is exponential to the number of distinct type-conditionals in the cJ class, assuming a straightforward mapping. Overall code length will also be exponentially greater, unless delegation, with its aforementioned disadvantages, is used.

In practice, it is unlikely that Java developers would want to deal with this kind of combinatorial complexity. Instead, they will likely prefer to provide a single type that captures the union of all possible members. In that case, when an "unsupported" method is called, a run-time error can be signaled in the form of an exception. For instance, following Java conventions, our earlier example is likely to be written in standard Java as follows:

```
class C<X> {
  public void store()
   throws UnsupportedOperationException
  { ... }
  ...
  public X getMin()
   throws UnsupportedOperationException
  { ... }
}
```

This addresses the code size and number-of-types explosion problem at the expense of sacrificing static type safety. The type checker is no longer able to tell under what conditions the `store` and `getMin` operations would be illegal. A run-time type error is produced instead, when illegal operations get called.[1] Relative to plain Java, cJ combines the advantages of static type safety and code conciseness.

It is worth noting that the cJ compiler translates its input into plain Java by following an approach similar to that of the example above (i.e., a single class is produced, containing all possible members). Yet, the cJ type system statically ensures that no exceptions for unsupported methods are thrown at run-time. We describe the cJ implementation in Section 5.

Finally, an interesting question on the power of type-conditionals concerns their value under multiple inheritance. Multiple inheritance can address the problems of delegation, in that it allows composing a class modularly without violating object identity or encapsulation. If Java had multiple inheritance, in addition to its bounded generics, the above example could be expressed in the same amount of code as in cJ. Nevertheless, the main benefit of type-conditionals is not in minimizing the code length, but in minimizing the number of explicit types that users need to manage. Consider the case of a type hierarchy among types `I1`, `I2`, ..., `IN`. Type conditionals allow the programmer to create implicitly a virtual isomorphic hierarchy by using a single class `C<X>` with member and/or supertype declarations conditional on `X` extending `I1`, `I2`, etc. The language will automatically ensure that the two hierarchies have consistent structure. If, for instance, `I1` is a subtype of `I5`, all methods in `C` declared conditionally under `X extends I1` will be able to access methods declared conditionally under `X extends I5`. With traditional subtyping mechanisms, the user would need to create explicit types `C1`, `C2`, ..., `CN` with a subtyping hierarchy reflecting the one of `I1`, `I2`, ..., `IN`. Relieving the programmer from explicitly managing these types is the greatest advantage of type-conditionals in *any* language setting. As we discuss in the next section, the stated motivation of Java developers for choosing a type-unsafe solution for the JCF was not avoiding code size explosion but avoiding an explosion in the number of explicit types that users would need to deal with.

3.2 Case Study: Java Collections Framework

A striking demonstration of the problems presented above can be found in the Java Collections Framework: the standard Java data structures library. The JCF supplies types such as `Collection`, `Set`, `Map`, and `List`. However, there are other cross-cutting concerns along which to organize these basic data structures. One such concern is that of "modifiability": is a data structure modifiable through its public interface or not? This concept is not captured via the Java type system in the design of the JCF. Instead, any attempt to modify an "unmodifiable" collection results in the throwing of an `UnsupportedOperationException` at run-time. Another similar concern is that of size variability. Some data structures are modifiable, yet their size cannot change—arrays are a standard example. An array supports the operations of the `List` interface with the exception of `add` or `remove`, which throw `UnsupportedOperationException`. This is a case of circumventing the static type system in order to avoid a combinatorial explosion in the number of types specified in the library. In fact, six out of the fifteen methods of interface `Collection` in JDK 1.5 are optional and may result in run-time errors.

The above is a well-known issue. The very first "frequently asked question" in the Java Collections API Design FAQ[2] is:

> Why don't you support immutability directly in the core collection interfaces so that you can do away with optional operations (and `UnsupportedOperationException`)?

The design rationale reflected in the answer to this FAQ indirectly offers a compelling argument for cJ. The developers note:

> Clearly, static (compile time) type checking is highly desirable, and is the norm in Java. We would have supported it if we believed it were feasible. Unfortunately, attempts to achieve this goal cause an explosion in the size of the interface hierarchy ...

Subsequently, the Java Collections API developers proceed to give an illustration of the kinds of "explosion in size" problems that a type-safe design would encounter, if cross-cutting concerns such as "modifiable", "variable-size", "append-only", etc., are expressed in the type system. The Java Collections Design FAQ concludes:

> Now we're up to twenty or so interfaces and five iterators, and it is almost certain that there are still collections arising in practice that don't fit cleanly into any of the interfaces.

The above issue is not specific to the Java Collections Framework. Other developers of Java data structure libraries have identified the same shortcomings. Doug Lea (quoted in the JCF FAQ) authored a popular Java collections package and remarks:

> Much as it pains me to say it, strong static typing does not work for collection interfaces in Java.

(We invite the reader to consult online the informative FAQ answer, which we cannot reproduce here in its entirety.)

cJ addresses fully and cleanly the above problem with the JCF. Interfaces `Collection`, `List`, etc. are implemented modularly using type-conditionals. Specifically, there are three interesting properties that we capture: whether a collection is modifiable, whether it supports only deletions, and whether it supports both deletions and additions (i.e., all size change operations). These cross-cutting concerns are expressed using (marker) interfaces `Modifiable`, `DeleteOnly` and `Resizable`. The `Resizable` interface is a subtype of `DeleteOnly`—a resizable collection supports operations such as `clear` and `remove`, but also `add` and `addAll`. By combining these interfaces one can specify different flavors of each collection. This is done through a type parameter `M` passed to each collection generic class. For instance, interfaces `Collection` and `List` are implemented as follows:[3]

```
interface Collection<E,M> extends Iterable<E,M> {
  <M extends Resizable>?
  <
  boolean add(E o);
  boolean addAll(Collection<? extends E, ?> c);
  >
  <M extends DeleteOnly>?
  <
  boolean removeAll(Collection<?, ?> c);
  void clear();
  ...
  >
  boolean contains(Object o);
  boolean isEmpty();
  ... // other methods common to all collections
}

interface List<E,M> extends Collection<E,M> {
  <M extends Resizable>?
  <
  void add(int index, E element);
```

[1] The `UnsupportedOperationException` is a run-time exception (i.e., the compiler does not check that it is always caught or declared) and a member of the JCF. For the purposes of this paper, we use this exception type even for code outside the JCF. Any different exception could assume the same general role.

[2] http://java.sun.com/j2se/1.5.0/docs/guide/collections/designfaq.html

[3] Our re-implementation of the JCF can be found on the cJ website: http://www.cc.gatech.edu/∼ssh/cj.

```
  boolean addAll(int index, Collection<? extends E,?> c);
  >
  <M extends DeleteOnly>?
  <
  E remove(int index);
  ...
  >
  <M extends Modifiable>?
  E set(int index, E element);
  ... // other methods common to all lists
}
```

(Note that the question-mark symbol is used both in our type-conditional syntax, and as a wildcard in order to specify variance in generic operations, per the standard Java syntax.) Concrete classes that implement these interfaces (e.g., `ArrayList`) have similarly structured type-conditionals. This implementation is concise without sacrificing static type safety. The user of the `List` interface explicitly selects the desired flavor of the collection. For instance, a possible type instantiation of `List` is `List<Integer,Modifiable>`, signifying a modifiable (but not resizable) list of integers. Another possible instantiation is `List<Integer,Object>` (or any type that is not a subtype of `Modifiable` in place of `Object`) to signify a non-modifiable and non-resizable list. The programmer cannot accidentally call a `set` method on a collection that is statically specified to be unmodifiable. The need for an `UnsupportedOperationException` is eliminated.

The JCF case study serves well as a motivating example for the more powerful cJ features described in later sections. Specifically, the major question we have not yet addressed is how to write general code that abstracts over multiple cJ types. There are two ways to safely abstract over types in the Java type system. One way is to use interfaces—e.g., we may want to write code that works with all `Comparable` objects regardless of whether they are of type `Integer`, `String`, `Array`, etc. The other way is to use variance—e.g., we can write code that works with all `List<X>` objects, as long as the element type, `X`, is a subtype of a given type, say, `Number`. Both of these valuable mechanisms are straightforwardly extended and enhanced in cJ.

cJ conditional supertypes enable abstraction using interfaces even for types that support the corresponding operations only conditionally. For instance, we can have definitions such as:

```
class ArrayList<X,M>
<X extends Comparable<X>>? implements Comparable<List<X>>
{
  <X extends Comparable<X>>?
  public int compareTo(List<X> that) { ... }
  ...
}
```

The above `ArrayList` class implements interface `Comparable` and provides the appropriate `compareTo` method only if its parameter type is also a `Comparable`.[4] Thus, we can use such `ArrayList` objects with code accepting any `Comparable` object—unlike the original Java `ArrayList` class.

The second kind of abstraction is quite interesting and practically valuable in the cJ setting. For instance, how can we write code that deals uniformly with `List` objects that support at least a `remove` operation, regardless of whether the objects are of type `ArrayList<E,DeleteOnly>` or `ArrayList<E,Resizable>` or any other compatible subtype and "flavor" combination? This is precisely the role of the question-mark wildcard types that appeared in our above Java Collections code—e.g., for method `addAll`. The general approach follows a natural extension of the standard Java variance mechanism. We discuss this topic in the next section.

[4] Our thanks to Phil Wadler for this motivating example.

4. Subtyping and Variance

cJ type-conditionals turn out to fit very well in the Java type checking framework. In particular, the relationships among different instantiations of the same generic cJ class fall out very simply from the standard rules for variance, with only a small addition. We next give a bird's eye view of wildcards and variance in the Java type system (readers familiar with variance can skip Section 4.1) and then discuss how these relate to cJ.

4.1 Variance and Wildcards

Here we only give a brief (and simplified) summary of Java wildcards as used to implement variance. A thorough treatment can be found in past literature [14, 31].

Java allows using generic types with a non-specific instantiation, through the wildcard syntax "`? extends T`", "`? super T`" and "`?`". For a generic type `C`, the meaning of a `C<? extends T>` is "`C` instantiated with any subtype of `T`". For instance, the JCF `Collection` interface supports a method:

```
interface Collection<E> extends Iterable<E> {
  ...
  addAll(Collection<? extends E> c);
}
```

The wildcard means that if, for instance, we have an object of type `Collection<Number>`, we can pass as an argument to its `addAll` method an object of type `Collection`-of-some-subtype-of-`Number`. For instance (assuming `Integer` subtypes `Number`):

```
Collection<Number> c = new ArrayList<Number>();
Collection<Integer> ci = new ArrayList<Integer>();
... // populate ci
c.addAll(ci);
```

Similarly, the wildcard syntax "`C<? super T>`" means "`C` instantiated with any supertype of `T`", and the syntax "`C<?>`" means "`C` instantiated with anything".

Wildcards form an elegant way to write highly general code that can apply to multiple instantiations of generic types. Nevertheless, to statically ensure that the result is safe (i.e., that the object can indeed support all the operations that the code wants to perform on it) several restrictions need to be imposed.

- An object `c` of type `C<? extends T>` can only be used to call methods where the type parameter of `C` is in a *co-variant* position, i.e., appears only as the return type of a method, if at all. Also, fields of `c` typed as the type parameter of `C` can only be read from, not written to. For instance, given an object `c` of type `Collection<? extends E>`, we can never invoke a method such as "`boolean add(E o)`" on `c`, because this method is declared in interface `Collection<E>`, and the type parameter `E` appears as an argument type to `add`.
- Similarly, an object `c` of type `C<? super T>` can only be used to call methods with the type parameter of `C` in a *contra-variant* position, i.e., it appears only as an argument type to a method, if at all. Fields of `c` typed as the type parameter of `C` can be written to with values typed `T`, but only read as values of type `Object`.
- An object `c` of type `C<?>` can only be used to call methods where the type parameter of `C` does not appear at all (*bi-variance*). Similarly, the fields of `c` typed as the type parameter of `C` can only be read as `Object`s, and not written to.

Next we discuss how a slight extension of the Java variance rules makes them apply transparently to cJ.

4.2 Variance and Type-Conditionals

We return to the original question regarding type-conditionals and subtyping. Consider a cJ class `C<X>`. Can we write code that is general enough to work type-safely with multiple instantiations of C<X> (i.e., for multiple values of X)? Consider the simple example from Section 3.1:

```
class C<X> {
  <X extends Serializable>?
  public void store() { ... }
  ...
  <X extends Comparable<X>>?
  public X getMin() { ... }
}
```

Intuitively, X is used in this example only in order to add more members to generic class C. Thus, a "stronger" X (i.e., one that will satisfy more "extends" type conditions) will only result in more members being added. In other words, if type S is a subtype of T then C<S> could safely be a subtype of C<T>—generic class C can be co-variant in its type parameter.

cJ, just like regular Java, does not automatically relate different instantiations of a generic class via subtyping. That is, in the Java and cJ type systems, an instantiation C<A> is never a subtype of C<B> for two distinct classes A and B, regardless of the contents of C or how A and B are related. However, if A is a subtype of B, then C<A> is a subtype of C<? extends B> and C<B> is a subtype of C<? super A>. The programmer can use such subtyping relations to write code that applies to multiple instantiations of a generic class and the language statically checks that the code is safe, based on the rules outlined earlier.

cJ enhances the variance rules to deal with type conditions. For instance, we can have the following method, accepting an argument of the above type C:

```
  void export(C<? extends Serializable> c) {
   ... c.store(); ...
  }
```

That is, the export method accepts objects of type C-of-some-subtype-of-Serializable. The language ensures that the body of export uses its argument c correctly. In this case, the call to store is statically type safe, since for any subtype X of Serializable, type C<X> will support store.

The general rule for interactions between type parameters and variance is simple:

> An occurrence of type parameter X in an <X extends ...>? type-condition (on either a supertype declaration or a member declaration) constitutes a co-variant use.

Enhanced with the above rule, all other rules of the standard variance framework of Java apply and enable general type safety.

Consider, for instance, a Queue that supports averaging of its elements if they are Numbers. (This is an artificial example—the functionality is not part of the Java Collections Framework.):

```
interface Queue<X> {
  <X extends Number>?
  X average();
  ... // other methods
}
```

Both appearances of type parameter X are in co-variant positions: either in a type condition, or as a return type. In this case, a method can accept objects of type Queue-of-some-subtype-of-Number and call average on them safely. For instance, we can have:

```
   void covariant(Queue<? extends Number> q) {
     Number a = q.average();
   }
```

We already saw uses of variance in our Java Collections API case study. Consider the following excerpt from the definition of Collection<E,M> in Section 3.2:

```
interface Collection<E,M> extends Iterable<E,M> {
  ...
  boolean addAll(Collection<? extends E, ?> c);
  ...
  boolean removeAll(Collection<?, ?> c);
}
```

Methods addAll and removeAll in the above use arguments bi-variant with respect to the second type parameter of Collection. That is, these methods can accept any collection, regardless of whether it is modifiable or not, delete-only or not, etc. Note that the above type signatures statically prevent the implementation of methods addAll and removeAll from calling methods such as add, clear, or set on their argument c: all these methods are declared conditionally and c may not support them. Intuitively, this reflects the intent of the interface for methods addAll and removeAll: they modify the object from which they are invoked, but not their argument object, from which they only read values to add or remove.

Overall, cJ type-conditionals are an excellent match for Java variance. Not only does variance offer a natural abstraction mechanism for conditional types, but also variance and type-conditionals offer the same kind of benefit in a programming language. Both mechanisms allow specifying a single class C<X> and having the type system automatically compute several useful derivative types. In the case of variance the derivative types are C<? extends T>, C<? super T> and C<?>, which contain only the co-variant, contra-variant, and bi-variant methods of the class, with respect to some type T. In the case of cJ the derivative types correspond to all possible outcomes of type-conditionals. For instance, Modifiable-and-DeleteOnly-List is an implicit type produced from the List<E,M> definition. Each cJ implicit type contains only the members that exist for this combination of conditions.

5. Implementation

The design of cJ was carefully planned to admit a simple *erasure-based* translation that is backward compatible with Java code. Each cJ generic class can be translated to a single Java generic class (which in turn can be translated to a single non-generic Java class, per the standard erasure translation of Java generics). This and other implementation topics are discussed next.

Erasure. The current cJ compiler is a source-to-source translator into Java. Nevertheless, exactly the same techniques could be used in a direct-to-bytecode translation. Indeed, the source-to-source translation has even more transparency requirements and demonstrates how well cJ fits the Java model.

cJ translates a class (or interface) with type-conditionals into a Java class (resp., interface) by removing all conditional statements. This enables a single class to play the role of all possible instantiations. Consider our earlier example:

```
class C<X> {
  <X extends Serializable>?
  public void store() { ... }
  ...
  <X extends Comparable<X>>?
  public X getMin() { ... }
}
```

cJ translates C into a class:

```
class C<X> {
  public void store() { ... }
  ...
  public X getMin() { ... }
}
```

Note that there is no need for a run-time exception. The cJ type system ensures statically that unsupported methods can never be called. (If client code is not compiled with the cJ compiler, there is no such guarantee. We later discuss how the user can explicitly request dynamic checks to ensure that these methods are not called.)

Erasure Intricacies. The cJ translation requires a few more steps than simply removing the type-conditionals. First, the cJ compiler translates the bodies of conditional methods using type casts that ensure the appropriate type conditions. Second, it supplies dummy method bodies to classes implementing (or extending) an interface (class) with unsupported methods. Lastly, it translates certain type instantiations into their "raw type" forms, and performs the same code generation that a plain Java compiler performs in translating generic code into non-generic code. We demonstrate these translations via examples.

When translating conditional code, the cJ compiler needs to maintain known type bounds for each expression. If this is different from the type the expression would have when conditionals are eliminated, then casts need to be output. Consider the example from the Introduction:

```
class C<X> {
  X xRef;
  ...
  <X extends DataSource>?
  void store() { ... xRef.getConnection() ... }
}
```

The call to `getConnection` is only valid because the type `xRef` is known to be a subtype of `DataSource`. Thus, the compiler needs to emit a cast that will ensure this type constraint when the type-conditional is removed. The cast cannot fail at run-time, as the cJ static type checker ensures the `store` method is only called when `X` is indeed a subtype of `DataSource`. The translated code is:

```
class C<X> {
  X xRef;
  ...
  void store()
  { ...((DataSource) xRef).getConnection()... }
}
```

In the case of interfaces (or abstract classes), our erasure translation means that classes implementing (extending) an interface (abstract class) may need to be automatically enhanced. Consider a conditional interface method. Erasure removes the type-conditional and the method will be declared for all instantiations of the interface. Yet, classes implementing some of these instantiations will not provide implementations of the method, as the method is undeclared for the given type parameters. For instance:

```
interface List<E,M> extends Collection<E,M> {
  <M extends DeleteOnly>?
  E remove(int index);
  ...
}

class FixedList<E> implements List<E,Object> {
  ... // no remove: Object is not subtype of DeleteOnly
}
```

The translation adds a dummy `remove` public method in `FixedList`. The translated version of the above example is as follows:

```
interface List<E,M> extends Collection<E,M> {
  E remove(int index);
  ...
}

class FixedList<E> implements List<E,Object> {
  ...
  public E remove(int index)
   throws UnsupportedOperationException
  { throw new UnsupportedOperationException(); }
}
```

The same translation technique is used for safety: in a naive translation scheme, subclasses of a class that has conditional methods would inherit those methods because of the erasure translation in the superclass, allowing code not compiled with the cJ compiler to gain access to those methods. Instead, we ensure that the subclass overrides the method with a dummy implementation to avoid such accidental exposure of the superclass functionality. Note that this problem is similar to that faced by the designers of GJ [3], and the solution we adopt is also similar to theirs. For instance, consider the following class:

```
class Channel<T> {
  <T extends Trusted>?
  void disconnect() { ... }
}
```

Erasure will remove the type-conditional and, thus, expose the `disconnect` method. If the user wants to ensure security he/she can export only specialized subclasses that explicitly do not implement the `Trusted` interface:

```
class NonsecureChannel extends Channel<Object> { }
```

The cJ compiler will translate the latter into a class that is safe to use in an insecure environment, avoiding accidental exposure of the superclass method:

```
class NonsecureChannel extends Channel<Object> {
  void disconnect()
   throws UnsupportedOperationException
  { throw new UnsupportedOperationException(); }
}
```

This translation technique does not help avoid the accidental exposure of fields, however. To protect a conditional field against unauthorized access, a programmer could designate the field `private`, and define getter/setter methods for it. The above translation technique for methods can then be used to protect the getter/setter methods from unauthorized uses.

In certain situations, a type instantiation considered legal by the cJ compiler might not be considered legal by a regular Java compiler. For example,

```
class C<X> {
  <X extends Enum<X>>?
  EnumSet<X> es = null;
}

class EnumSet<X extends Enum<X>> {
  ...
}
```

A simple erasure applied to class `C<X>` would erase the type condition `<X extends Enum<X>>?`. However, type instantiation `EnumSet<X>` is not compilable using a Java compiler, because `X` is nowhere declared to be a subtype of `Enum<X>`. In these situations, cJ translates type `EnumSet<X>` all the way down to its "raw type" form, `EnumSet`. Thus, the translation of `C<X>` would be:

```
class C<X> {
  EnumSet es = null;
}
```

The cJ compiler then needs to perform all the translations that a regular Java compiler does for expressions of type `EnumSet`, e.g., generating casts of return types of methods called on this type.

Translation and Backward Compatibility. The interesting aspect of the cJ translation, as described above, is that it is remarkably simple and fits very well the existing Java object model. The restrictions of the cJ language outlined in Section 2.2 are in place explicitly so that an elegant erasure-based translation can be supported. For instance, ensuring that methods do not conflict, even when they are under disjoint type conditions, means that we can employ the erasure-based translation without the need for method renamings.

Similarly, ensuring that overriding methods (in a subclass) are declared under weaker type conditions than the overridden methods (in the superclass) enables a clean erasure by just removing the type-conditionals. It means that a subclass method does not “accidentally” override a valid superclass method when the subclass method should not really exist based on its type condition. Translating all cJ classes and methods one-to-one into Java classes and methods ensures good interfacing with client code, and even unsuspecting legacy (i.e., standard Java) code.

The cJ translation also includes some transparent special case handling purely for strong backward compatibility, even at the source level. This was motivated by our study of the Java Collections Framework. The special handling occurs when the cJ compiler is invoked in “compatibility mode” and when a type parameter is used *only* in type-conditionals (and not, for instance, to declare references). In that case, the cJ compiler treats the parameter as optional. For instance, the cJ compiler can compile legacy Java code using the `Collection<E>` interface (and any of the classes implementing it) against the cJ library, which defines `Collection` as `Collection<E,M>`. (Either all optional parameters or none need to be omitted.) When a type parameter is omitted, the instantiation is assumed to satisfy all the type-conditionals, and any instantiation with full type parameters is a subtype of it, and vice versa. Our treatment is directly analogous to “raw types” in the translation of Java generics [3].

Furthermore, when a type parameter to a class is used only in type-conditionals (or transitively as a type argument to another class that uses this parameter only in type conditionals), then the cJ compiler removes it from the translated code. This means that the code generated from the cJ compiler can be used as a regular Java library, under plain Java compilers. This is best illustrated with an example. Consider the form of our standard `List` interface from the Java Collections API:

```
interface List<E,M> extends Collection<E,M> {
  <M extends Modifiable>?
  E set(int index, E element);

  <M extends Resizable>?
  void add(int index, E element);
  ...
}
```

`List` uses its type parameter `M` only in type conditions and to instantiate another type, `Collection`, where it is also used only in type conditions. `M` is never used as an argument or return type of a method. Therefore, the cJ compiler accepts code that refers to `List` with only one type parameter. At the same time, the translation of the above cJ interface into a plain Java interface eliminates the second type parameter:

```
interface List<E> extends Collection<E> {
  E set(int index, E element);

  void add(int index, E element);
  ...
}
```

In short, the cJ compiler compiles old-style Java code even against new-style (cJ) libraries that use extra type variables for type conditions. Furthermore, the cJ compiler translates new-style (type conditional) library code into Java code that is source-compatible with existing Java client code, under standard Java compilers.

Clearly, our erasure translation has the same requirements as other erasure translations—e.g., that of GJ [3]—for the purpose of full integration in Java. For instance, the reflection mechanism needs to change to support cJ-translated code. This is not part of our current implementation.

6. Formalization

We present the formal syntax and typing rules for a subset of cJ. Our formalism is an extension of the formalism for Featherweight GJ (FGJ) with variance, by Igarashi and Viroli [14]. We call our calculus Featherweight cJ (FCJ). FCJ captures a core subset of cJ functionality that allows us to explore the type-safety issues introduced by type-conditionals, with minimum extra baggage and duplication of work that has already been done for FGJ [11] and variance [14]. Our formalism requires that all type parameters declare upper bounds, which may be `Object`. Each class must declare a superclass, which may also be `Object`. Additionally, all superclass declarations must be guarded by type-conditionals, though unconditional superclasses can be expressed by having the type-conditional be `<X extends N>?`, where `N` is `X`’s declared upper bound. Similarly, all method declarations must be guarded by type-conditionals, as well. Conditional fields are not supported in the formalism, since the issues involving conditional member declarations are thoroughly represented by conditional methods. Interfaces are not part of either the original FGJ, or our formalism. Thus, we only support conditional superclasses. A class declaration includes a sequence of fields and method declarations. We assume an implicit constructor for each class, which takes as arguments expressions that can be used to initialize field values. The method body is simply an expression.

Note that the variance formalization by Igarashi and Viroli does not strictly model the wildcard implementation in Java. Some notable differences include the inability in the Igarashi and Viroli system to access a co-variantly typed field from a contra-variantly instantiated type, yielding `Object` as the field’s type. An attempt to formalize the wildcard mechanism as it is implemented in Java is presented by Torgersen et al. [30]. However, the Torgersen et al. formalism has not been proven sound. Thus, we choose to work with the Igarashi and Viroli formalization here, as a solid basis for proving the soundness of our type system.[5]

Notation. For readers unfamiliar with FGJ [11] and the variance formalism [14], we briefly introduce the notational conventions used. The meta-variables `C` and `D` range over class names; `X`, `Y`, and `Z` range over distinct type variables; `S`, `T`, `U`, `V`, and `W` range over types; `H`, `N`, `O`, `P`, `Q`, and `R` range over nonvariable types (fully instantiated types); `f` and `g` range over field names; `m` ranges over method names; `x` ranges over parameter names; `d` and `e` range over expressions; and `M` ranges over method declarations. Meta-variable `v` represents variance annotations `o`,`+`, `-`, and `*`, for in-variant, co-variant, contra-variant, and bi-variant, respectively—e.g., `+T` corresponds to `? extends T` in the full Java syntax. Variance annotations can be placed in front of any non-variable type. In-variant is the assumed default, and thus, `C<oT>` is abbreviated to `C<T>`. A partial order $\leq$ on variance annotations can be defined as: $\mathtt{o} \leq \mathtt{+} \leq \mathtt{*}$, $\mathtt{o} \leq \mathtt{-} \leq \mathtt{*}$. $\mathtt{v}_1 \vee \mathtt{v}_2$ represents the least upper bound of $\mathtt{v}_1$ and $\mathtt{v}_2$.

In addition, we use a few shorthand conventions for conciseness. $\overline{\mathtt{X}}$ is a shorthand for $\mathtt{X}_1, \ldots, \mathtt{X}_n$, and similarly, $\overline{\mathtt{T}}\,\overline{\mathtt{x}}$ is a shorthand for $\mathtt{T}_1 \mathtt{x}_1, \ldots, \mathtt{T}_n \mathtt{x}_n$. When this shorthand is applied to a type variable or a regular variable (i.e., fields, method arguments), it represents a sequence with no duplication. We use $\bullet$ to denote an empty sequence. The notation $\triangleleft$ is the shorthand for keyword `extends`, and $\uparrow$ is the shorthand for keyword `return` in method bodies.

[5] After the completion of the work presented in this paper, an even more recent formalization of Java wildcards has been published [4]. This formalization does have a proof of soundness, and should reflect more accurately the wildcard mechanism in Java. We intend to explore using this formalization in the cJ type system in our future work.

We also assume a class table CT, which maps class names C to their declarations. A *program* is a pair $(CT, \texttt{e})$ of a class table, and an expression.

6.1 Syntax

We present the FCJ syntax in Figure 1. The syntax follows closely the abstract syntax for FGJ with variance [14]. The main difference is the addition of a type-conditional construct in front of superclass and method declarations. The type-conditional construct, $<\overline{X} \lhd \overline{R}>?$, evaluates to true if, after type parameter instantiation, the types for $\overline{X}$ are subtypes of $\overline{R}$. A fully instantiated class has the declared superclass if and only if the type-conditional guarding the superclass declaration evaluates to true. Otherwise, it extends Object. Similarly, a method exists for a fully instantiated class if and only if its type condition evaluates to true.

Syntax:

T	::=	X | N
N	::=	$C<\overline{vT}>$
v	::=	o | + | - | *
CL	::=	class $C<\overline{X}\lhd\overline{N}>$ S-if $\lhd D<\overline{S}>$ $\{\overline{T}\ \overline{f};\ \overline{M}\}$
M	::=	S-if $<\overline{Y}\lhd\overline{P}>$ T m $(\overline{T}\ \overline{x})$ $\{\uparrow e;\}$
S-if	::=	$<\overline{X}\lhd\overline{R}>?$
e	::=	x
	|	e.f
	|	$e.<\overline{T}>m(\overline{e})$
	|	new $C<\overline{T}>(\overline{e})$
	|	(T)e

Figure 1. Syntax

6.2 Type System

The main typing rules for FCJ are presented in Figure 2. Δ and Γ are the two environments used in typing judgments. Δ is a type environment that ranges over subtyping assumptions of the form $T<:S$. When $X<:N\in\ \Delta$ and N is a non-variable type, for all X, we say that Δ has non-variable bounds. Γ is a variable environment that maps a variable x to its type T.

To support the typing rules, we present some auxiliary definitions in Figure 3, and the definition of "open"($\Uparrow^{\Delta}$) and "close"($\Downarrow_{\Delta}$) of variant types in Figure 4. These rules and definitions follow closely the format of those in variance-based FGJ. We assume the reader has a certain familiarity with the FGJ formalization, though not necessarily with the Igarashi and Viroli variance formalism. To enhance the understanding of our type system, we first highlight some important additions to FGJ made by Igarashi and Viroli regarding variance. We then delve into the rules and definitions specifically changed for the inclusion of type-conditionals in cJ.

Background on Variance Formalism. The two most important additions of the Igarashi and Viroli system over FGJ are the concepts of "open" and "close" (Figure 4). Before any type is used (i.e., for field or method invocation, or in a subtyping judgment), it must be "opened" first. Opening a type means that we introduce a fresh type variable for each co- or contra-variantly defined type. For example, before we can check the validity of invoking method m in type C<+T>, we must open this type by introducing a fresh type variable X into Δ, where $\Delta \vdash X <: T$. To look for method m in C<+T> now means to look for m in C<X>. If T occurs anywhere in m's type, it is replaced by X, as well.

This "opening" conveniently disallows illegal accesses of methods or fields that we informally described in Section 4. For example, suppose that in the definition of class C<X>, we have method D m (X x) { ... }. The type parameter X appears in a contravariant position—as method m's argument type. This means that any co-variantly instantiated type C<+T> should not be able to invoke method m. This is indeed the case in this formalism: we first open C<+T> to C<Y>, where $\Delta \vdash Y<:T$. We then check that any invocation of m passes in an argument of some subtype of Y. However, Y is simply a type variable with an *upper* bound of T. According to the subtyping rules in Figure 2, no type can be deemed a subtype of Y (except Y itself, which is not available before the opening, and thus cannot be the type of any argument passed to m). Thus, no invocation of m on an expression of type C<+T> can be well-typed. Similarly, had the type parameter X appeared in a co-variant position in C<X>, e.g., as the return type of a method, an expression with the contra-variantly instantiated type C<-T> would not have been able to invoke that method.

Since "open" introduces new type variables into the type environment, it is always paired with a "close" operation, where the newly introduced type variable is closed down to its bound and removed from the type environment. Closing also re-introduces variance annotations, using a conservative combination of the variance annotations of the current use context (i.e., the type being closed) and the surrounding definition context used for the preceding "open".

Auxiliary Definitions. Function $mtype(\Delta,\ m,\ C<\overline{T}>)$ returns the signature of method m, in type $C<\overline{T}>$, in the form of $<\overline{Y}\lhd\overline{P}>\overline{U}\rightarrow U_0$. $mtype(\Delta,\ m,\ C<\overline{T}>)$ is defined under two rules:

- MT-CLASS says that if method m is declared in class $C<\overline{X}>$ with type-conditional $<\overline{X}\lhd\overline{R}>?$, and the type-conditional is satisfied by substituting types $\overline{T}$ for type parameters $\overline{X}$, i.e., $\Delta \vdash \overline{T}<:[\overline{T}/\overline{X}]\overline{R}$, then $mtype(\Delta,\ m,\ C<\overline{T}>)$ is defined.
- MT-SUPER covers the condition when method m is not declared in class $C<\overline{X}>$ at all. In this case, if the type-conditional for superclass $D<\overline{S}>$ is satisfied by the substitution $[\overline{T}/\overline{X}]$, then $mtype(\Delta,\ m,\ C<\overline{T}>)$ is defined as $mtype(\Delta,\ m,\ [\overline{T}/\overline{X}](D<\overline{S}>))$.

Note that we do not need a case for when m is declared in $C<\overline{X}>$, but the type-conditional guarding it is not satisfied by the substitution $[\overline{T}/\overline{X}]$. As explained in Section 2.2, the type conditions on a subclass method must be weaker than the type conditions guarding the method it overrides in the superclass (this restriction is formalized in the *override* rule, which we explain later in this section). Thus, if method m's type conditions in $C<\overline{X}>$ cannot be satisfied by the assumptions in Δ, then its type conditions in the superclass of $C<\overline{X}>$ cannot possibly be satisfied. There is no need to invoke MT-SUPER in this case.

$mbody(\Delta,\ m<\overline{W}>,\ C<\overline{T}>)$ returns a pair, $(\overline{x},\ e)$. $\overline{x}$ are the parameters of m, and e is m's body. $\overline{W}$ are the actual types inferred for a polymorphic method m. Note $mbody$ is similarly defined under the same two conditions that $mtype$ is.

$fields(\Delta,\ C<\overline{T}>)$ returns a sequence of fields in class $C<\overline{T}>$. Object has no fields. For all other types $C<\overline{T}>$, $fields(\Delta,\ C<\overline{T}>)$ returns the sequence of fields declared in $C<\overline{T}>$, and, if the type-conditional guarding $C<\overline{T}>$'s superclass, $D<\overline{S}>$, is satisfied by the substitution $[\overline{T}/\overline{X}]$, the value of $fields(\Delta,\ [\overline{T}/\overline{X}](D<\overline{S}>))$ is returned, as well.

The predicate $override(\Delta,\ m,\ <\overline{X}\lhd\overline{R}>?D<\overline{S}>,\ <\overline{X}\lhd\overline{H}>?<\overline{Y}\lhd\overline{P}>\overline{U}\rightarrow U_0)$ judges if a method m, with signature $<\overline{X}\lhd\overline{H}>?<\overline{Y}\lhd\overline{P}>\overline{U}\rightarrow U_0$ may be defined in a class that has a conditional superclass $D<\overline{S}>$, guarded by condition $<\overline{X}\lhd\overline{R}>$. The extra complication in this rule reflects the restrictions described in Section 2.2, and is used in proving the correctness of our erasure-based translation (Section 6.4). There are two aspects of our translation to consider. Firstly, recall that our translation scheme erases all type conditionals. After

Expression typing:

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>} \qquad \mathit{fields}(\Delta, \mathit{bound}_\Delta(\mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>})) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \quad \mathtt{S}_i \Downarrow_{\Delta'} \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{T}} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \\ \mathit{mtype}(\Delta,\ \mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{U}_0 \\ \overline{\mathtt{Y}} <: \overline{\mathtt{Q}} \notin \Delta' \text{ for any } \overline{\mathtt{Q}} \quad \Delta \vdash \overline{\mathtt{V}}\ ok \\ \Delta, \Delta' \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \Delta, \Delta' \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\texttt{<}\overline{\mathtt{V}}\texttt{>}\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{T}} \qquad \text{(T-INVK)}$$

$$\frac{\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ ok \quad \mathit{fields}(\Delta, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \quad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}}) \in \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathtt{T}\ ok \\ \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathit{bound}_\Delta(\mathtt{T}) \\ \text{or } \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}) <: \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma \vdash (\mathtt{T})\mathtt{e}_0 \in \mathtt{T}} \qquad \text{(T-CAST)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathtt{T}\ ok \\ \Delta \nvdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathit{bound}_\Delta(\mathtt{T}) \\ \text{and } \Delta \nvdash \mathit{bound}_\Delta(\mathtt{T}) <: \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma \vdash (\mathtt{T})\mathtt{e}_0 \in \mathtt{T}} \qquad \text{(T-SCAST)}$$

Method typing:

$$\frac{\begin{array}{c}\overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{R}} <: \overline{\mathtt{N}} \quad \Delta = \overline{\mathtt{X}} <: \overline{\mathtt{R}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \quad \Delta \vdash \overline{\mathtt{R}}, \overline{\mathtt{P}}, \overline{\mathtt{T}}, \mathtt{T}_0\ ok \\ \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this}:\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>} \vdash \mathtt{e}_0 \in \mathtt{S}_0 \quad \Delta \vdash \mathtt{S}_0 <: \mathtt{T}_0 \\ CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{V}}\texttt{>?} \triangleleft \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}\ \{\ldots\} \\ \mathit{override}(\Delta,\ \mathtt{m},\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{V}}\texttt{>?}\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>},\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\texttt{>?<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}_0)\end{array}}{\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\texttt{>?<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\mathtt{T}_0\ \mathtt{m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \uparrow \mathtt{e}_0;\ \}\ \mathtt{OK\ IN}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}} \qquad \text{(T-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}\Delta_1 = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \quad \Delta_1 \vdash \overline{\mathtt{R}} <: \overline{\mathtt{N}} \quad \Delta_1 \vdash \overline{\mathtt{N}}\ ok \\ \Delta_2 = \overline{\mathtt{X}} <: \overline{\mathtt{R}} \quad \Delta_2 \vdash \overline{\mathtt{R}},\ \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}\ ok \\ \Delta_1 \vdash \overline{\mathtt{T}}\ ok \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\end{array}}{\mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\texttt{>?} \triangleleft \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-CLASS)}$$

Subtyping:

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

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

$$\frac{\mathtt{X} <: \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X} <: \mathtt{T}} \qquad \text{(S-UBOUND)}$$

$$\frac{\mathtt{T} <: \mathtt{X} \in \Delta}{\Delta \vdash \mathtt{T} <: \mathtt{X}} \qquad \text{(S-LBOUND)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\texttt{>?} \triangleleft \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}\ \{\ldots\} \\ \Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>} \\ \Delta, \Delta' \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{R}} \quad ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}) \Downarrow_{\Delta'} \mathtt{T}\end{array}}{\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} <: \mathtt{T}} \qquad \text{(S-CLASS)}$$

$$\frac{\begin{array}{c}\overline{\mathtt{v}} \le \overline{\mathtt{w}} \quad \text{if } \mathtt{w}_i \le \mathtt{-},\ \text{then } \Delta \vdash \mathtt{T}_i <: \mathtt{S}_i \\ \text{if } \mathtt{w}_i \le \mathtt{+},\ \text{then } \Delta \vdash \mathtt{S}_i <: \mathtt{T}_i\end{array}}{\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{S}}\texttt{>} <: \mathtt{C}\texttt{<}\overline{\mathtt{w}}\overline{\mathtt{T}}\texttt{>}} \qquad \text{(S-VAR)}$$

Well-formed types:

$$\Delta \vdash \mathtt{Object}\ ok \qquad \text{(WF-OBJECT)}$$

$$\frac{\mathtt{X} <: \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X}\ ok} \qquad \text{(WF-VAR)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\texttt{>?} \triangleleft \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}\ \{\ \ldots\} \\ \Delta \vdash \overline{\mathtt{T}}\ ok \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\end{array}}{\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ ok} \qquad \text{(WF-CLASS)}$$

Figure 2. Typing Rules

translation, a class $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$ extends its superclass $\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ unconditionally. Consequently, even if a method `m` in $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$ is declared under a type-conditional that precludes the condition for the superclass, the type of `m` in $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$ still cannot conflict with the type of `m` in $\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$. Secondly, also recall that if a method in a subclass has the same type signature as a method in a superclass, we require the subclass method to *always* override the superclass method. This means the type-conditional on the subclass method must be implied by the type-conditional on the superclass method. This requirement ensures that we do not have to dynamically decide whether a class's own method implementation should be invoked or it should call `super.m(...)`.

In order for *override* to reflect these restrictions, it uses the function mtype_{uc}, which *unconditionally* recurses up the chain of superclasses to find a method's signature. $\mathit{mtype}_{uc}(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$ returns a pair, $(\Delta',\ \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{U}_0)$. Δ' contains subtyping assumptions that must be satisfied for method `m` to have type $\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{U}_0$. The second part of the pair is `m`'s signature as defined in the closest superclass up the unconditional chain of inheritance.

The *override* rule uses the *implies* notation (as in FGJ [11]) to indicate that the restrictions represented by the consequent of the *implies* need to be satisfied only if the antecedent is true. In this case, the antecedent is: $\mathit{mtype}_{uc}(\mathtt{m},\ \overline{\mathtt{X}},\ \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}) = (\Delta', \texttt{<}\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{Q}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}_0)$. This means that method `m` is defined in either $\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$, or some conditional superclass of $\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$. If this antecedent is true, then the parameter, return types, and bounds on the inferred types must be the same in the subclass as they are in the conditional superclass. It must also be true that given all conditions guarding the chain of conditional superclasses mtype_{uc} recursed through to find `m`, and the condition guarding `m` itself, the condition guarding the subclass method is true, as well. This is checked by augmenting Δ with Δ' and $\overline{\mathtt{X}} <: \overline{\mathtt{R}}$, and requiring that these are sufficient to show $\overline{\mathtt{X}} <: \overline{\mathtt{H}}$.

Note that the definition of *override* is dependent on the subtyping rules defined in Figure 2. Depending on the specific algorithm implementing our declarative subtyping rules, it is possible that the subtyping condition guarding the overriding method, $\overline{\mathtt{X}} <: \overline{\mathtt{H}}$, cannot be shown to be true using the assumptions in Δ, Δ', and $\overline{\mathtt{X}} <: \overline{\mathtt{R}}$. Consequently, certain valid overriding methods cannot be proven so. To see this concretely, let $\Delta = \mathtt{Foo}\texttt{<X>} <: \mathtt{Baz}$, where `Foo` is defined as: `class Foo<X> <X◁Bar>?◁Baz {...}`. If we want to show that $\Delta \vdash \mathtt{X} <: \mathtt{Bar}$, we need to deconstruct type `Foo<X>`, and infer from $\mathtt{Foo}\texttt{<X>} <: \mathtt{Baz}$ that $\mathtt{X} <: \mathtt{Bar}$ must be true, as well. Our current implementation does not deconstruct types to do such inference. Subtyping assumptions thrown into Δ' by $\mathit{mtype}_{\mathrm{uc}}$ are only effective if the type variables $\overline{\mathtt{X}}$ are not buried inside of constructed types, such as `Foo<X>`. We are currently working on a decidable algorithm for deconstructing types to get more precise subtyping assumptions. Note that this is a standard point of trade-off: a too powerful reasoning procedure may well end up being undecidable. A conservative algorithm, on the other hand, will reject some programs because of its inability to establish the conditions for their soundness. The latter is typically preferable in practice, since, in this setting, troublesome programs tend to be highly contrived.

Method type lookup:

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{Q}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{S}}\texttt{>\{...}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>U}_0\ \texttt{m}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\ \{\uparrow\texttt{e;}\}\in\overline{\texttt{M}} \qquad \Delta\vdash\overline{\texttt{T}}<:[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{R}} \end{array}}{mtype(\Delta,\ \texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0)}$$

(MT-CLASS)

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{Q}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{S}}\texttt{>\{...}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{m is not defined in } \overline{\texttt{M}} \qquad \Delta\vdash\overline{\texttt{T}}<:[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{Q}} \end{array}}{mtype(\Delta,\ \texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mtype(\Delta,\ \texttt{m},\ [\overline{\texttt{T}}/\overline{\texttt{Q}}]\texttt{D<}\overline{\texttt{S}}\texttt{>})}$$

(MT-SUPER)

Method body lookup:

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{Q}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{S}}\texttt{>\{...}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>U}_0\ \texttt{m}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\ \{\uparrow\texttt{e;}\}\in\overline{\texttt{M}} \qquad \Delta\vdash\overline{\texttt{T}}<:[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{R}} \end{array}}{mbody(\Delta,\ \texttt{m<}\overline{\texttt{W}}\texttt{>},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}},\ [\overline{\texttt{W}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{e})}$$

(MB-CLASS)

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{Q}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{S}}\texttt{>\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{m is not defined in } \overline{\texttt{M}} \qquad \Delta\vdash\overline{\texttt{T}}<:[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{Q}} \end{array}}{mbody(\Delta,\ \texttt{m<}\overline{\texttt{W}}\texttt{>},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mbody(\Delta,\ \texttt{m<}\overline{\texttt{W}}\texttt{>},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{S}}\texttt{>})}$$

(MB-SUPER)

Field lookup:

$$fields(\Delta, \texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{U}}\texttt{>}\ \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} \\ fields(\Delta,\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{U}}\texttt{>}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \Delta\vdash\overline{\texttt{T}}<:[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{R}} \end{array}}{fields(\Delta,\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{U}}\texttt{>}\ \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} \\ \Delta\vdash\overline{\texttt{T}}\not<:[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{R}} \end{array}}{fields(\Delta,\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}}$$

Unconditional field lookup:

$$fields_{uc}(\texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{U}}\texttt{>}\ \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} \\ fields_{uc}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{U}}\texttt{>}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \end{array}}{fields_{uc}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}}$$

Unconditional method type lookup:

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{Q}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{S}}\texttt{>\{...}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>U}_0\ \texttt{m}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\ \{\uparrow\texttt{e;}\}\in\overline{\texttt{M}} \qquad \Delta' = [\overline{\texttt{T}}/\overline{\texttt{X}}](\overline{\texttt{X}}<:\overline{\texttt{R}}) \end{array}}{mtype_{uc}(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\Delta',\ [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0))}$$

(MT_{UC}-CLASS)

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}}\lhd\overline{\texttt{Q}}\texttt{>?}\lhd\texttt{D<}\overline{\texttt{S}}\texttt{>\{...}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{m is not defined in } \overline{\texttt{M}} \\ mtype_{uc}(\texttt{m},\ \texttt{D<}\overline{\texttt{S}}\texttt{>}) = (\Delta, \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0) \qquad \Delta' = \Delta, [\overline{\texttt{T}}/\overline{\texttt{X}}](\overline{\texttt{X}}<:\overline{\texttt{Q}}) \end{array}}{mtype_{uc}(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\Delta',\ [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0))}$$

(MT_{UC}-SUPER)

Valid method overriding:

$$\frac{\begin{array}{c} mtype_{uc}(\texttt{m},\ \texttt{D<}\overline{\texttt{S}}\texttt{>}) = (\Delta',\ \texttt{<}\overline{\texttt{Z}}\lhd\overline{\texttt{Q}}\texttt{>}\overline{\texttt{T}}\to\texttt{T}_0) \text{ implies} \\ [\overline{\texttt{Y}}/\overline{\texttt{Z}}](\overline{\texttt{T}}, \texttt{T}_0, \overline{\texttt{Q}}) = (\overline{\texttt{U}}, \texttt{U}_0, \overline{\texttt{P}}) \text{ and } \Delta, \Delta', \overline{\texttt{X}} <: \overline{\texttt{R}} \vdash \overline{\texttt{X}} <: \overline{\texttt{H}} \end{array}}{override(\Delta,\ \texttt{m},\ \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?D<}\overline{\texttt{S}}\texttt{>},\ \texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{H}}\texttt{>?<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\to\texttt{U}_0)}$$

Bound of type:

$$bound_\Delta(\texttt{N}) = \texttt{N} \qquad \frac{\Delta(\texttt{X}) = (+, \texttt{S})}{bound_\Delta(\texttt{X}) = bound_\Delta(\texttt{S})}$$

Figure 3. Auxiliary definitions

The two rules for unconditional field lookup are only used in proving the correctness of erasure, using the erasure rules detailed in the accompanying technical report [10]. They are included for completeness of the auxiliary functions.

Type Rules. Most of the rules presented in Figure 2 are the same as their variance-based FGJ counterparts. We now go through the ones particular to the type-conditional extensions in FCJ.

T-FIELD and T-INVK: these rules define when a field reference or a method invocation, respectively, is well-typed. Even though they look identical to their variance-based FGJ counterparts, they use functions *fields* and *mtype*, which fully encapsulate the lookup of conditional supertypes and conditional methods, as previously explained.

T-METHOD: The interesting change to the T-METHOD rule from its counterpart in variance-based FGJ is that the environment Δ (under which the return and parameter types, as well as the body of the method, expression $\texttt{e}_0$, must be well-typed) is augmented with the type bound $\overline{\texttt{X}}<:\overline{\texttt{R}}$, which is the type-conditional under which the method is declared. Intuitively, this says that if a method is declared under type-conditional $\texttt{<}\overline{\texttt{X}}\lhd\overline{\texttt{R}}\texttt{>?}$, then in the scope of the body of the method it can be assumed that the type environment Δ supports this bound.

T-CLASS: Note that the conditional superclass $\texttt{D<}\overline{\texttt{S}}\texttt{>}$ needs to be proved well-typed under Δ augmented with the type-conditional condition guarding it.

6.3 Proof of Soundness

We prove the soundness of our type system by proving subject reduction and progress properties [32]. We state the reductions rules and theorems here. Interested readers can find the full version of the proofs in the appendix.

Open:

$$\Delta \vdash \texttt{T} \Uparrow^{\emptyset} \texttt{T} \qquad \text{(O-REFL)}$$

$$\frac{\Delta \vdash \texttt{S} \Uparrow^{\Delta_1} \texttt{T} \quad \Delta, \Delta_1 \vdash \texttt{T} \Uparrow^{\Delta_2} \texttt{U}}{\Delta \vdash \texttt{S} \Uparrow^{\Delta_1, \Delta_2} \texttt{U}} \qquad \text{(O-TRANS)}$$

$$\frac{\texttt{X} \text{ fresh for } \Delta, \texttt{C<}\overline{\texttt{v}}_1\overline{\texttt{T}}_1\texttt{,vT,}\overline{\texttt{v}}_2\overline{\texttt{T}}_2\texttt{>} \qquad \texttt{v} \neq \texttt{o}}{\Delta \vdash \texttt{C<}\overline{\texttt{v}}_1\overline{\texttt{T}}_1\texttt{,vT,}\overline{\texttt{v}}_2\overline{\texttt{T}}_2\texttt{>} \Uparrow^{\texttt{X}:(\texttt{v},\texttt{T})} \texttt{C<}\overline{\texttt{v}}_1\overline{\texttt{T}}_1\texttt{,oX,}\overline{\texttt{v}}_2\overline{\texttt{T}}_2\texttt{>}} \qquad \text{(O-CLASS)}$$

Close:

$$\frac{\Delta(\texttt{X}) = (\texttt{+},\texttt{T})}{\texttt{X} \Downarrow_\Delta \texttt{T}} \qquad \text{(C-PROM)}$$

$$\frac{\texttt{X} \notin dom(\Delta)}{\texttt{X} \Downarrow_\Delta \texttt{X}} \qquad \text{(C-TVAR)}$$

$$\frac{(\texttt{w}_i\texttt{,T}'_i) = \begin{cases} (\texttt{v}_i, \texttt{T}_i) & \text{if } \texttt{T}_i \Downarrow_\Delta \texttt{T}_i \\ (\texttt{v}_i \vee +, \texttt{U}_i) & \text{if } \texttt{T}_i \Downarrow_\Delta \texttt{U}_i \text{ and } \texttt{T}_i \neq \texttt{U}_i \\ (\texttt{v}_i \vee \texttt{v}'_i, \texttt{U}_i) & \text{if } \texttt{T}_i = \texttt{X} \text{ and } \Delta(\texttt{X}) = (\texttt{v}'_i, \texttt{U}_i) \end{cases}}{\texttt{C<}\overline{\texttt{v}}\overline{\texttt{T}}\texttt{>} \Downarrow_\Delta \texttt{C<}\overline{\texttt{w}}\overline{\texttt{T}}'\texttt{>}}$$

(C-CLASS)

Figure 4. Open and Close

Theorem 1 [Subject Reduction]: If $\Delta;\Gamma \vdash \texttt{e}\in\texttt{T}$ and $\texttt{e} \to \texttt{e}'$, then $\Delta;\Gamma \vdash \texttt{e}' \in \texttt{S}$ and $\Delta \vdash \texttt{S}<:\texttt{T}$ for some $\texttt{S}$.

Theorem 2 [Progress]: Let $\texttt{e}$ be a well-typed expression.

1. If $\texttt{e}$ has $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}$ as a subexpression, then $fields(\emptyset, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{f}}$, and $\texttt{f} = \texttt{f}_i$.

2. If $\texttt{e}$ has $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$ as a subexpression, then $mbody(\emptyset,\ \texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, \texttt{e}_0)$ and $|\overline{\texttt{x}}| = |\overline{\texttt{d}}|$.

Theorem 3 [Type Soundness]: If $\emptyset;\emptyset \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \to^* \texttt{e}'$ being a normal form, then $\texttt{e}'$ is either a value v such that

$\emptyset; \emptyset \vdash v \in \texttt{S}$ and $\emptyset \vdash \texttt{S} <: \texttt{T}$ for some S, or an expression that includes $\texttt{(T)new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$ where $\emptyset \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \not<: \texttt{T}$.

Reduction Rules:

$$\frac{\mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{f}}}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i \longrightarrow \texttt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>,C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, \texttt{e}_0)}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{d}}\texttt{)} \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}},\ \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0} \quad \text{(R-INVK)}$$

$$\frac{\emptyset \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{T}}{\texttt{(T)new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}} \quad \text{(R-CAST)}$$

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

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}'_0}{\texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{e}'_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}} \quad \text{(RC-INV-RECV)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}'_i}{\texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\ldots\texttt{,e}_i\texttt{,}\ldots\texttt{)} \longrightarrow \texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\ldots\texttt{,e}'_i\texttt{,}\ldots\texttt{)}} \quad \text{(RC-INV-ARG)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}'_i}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\ldots\texttt{,e}_i\texttt{,}\ldots\texttt{)} \longrightarrow \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\ldots\texttt{,e}'_i\texttt{,}\ldots\texttt{)}} \quad \text{(RC-NEW-ARG)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}'_0}{\texttt{(T)e}_0 \longrightarrow \texttt{(T)e}'_0} \quad \text{(RC-CAST)}$$

Figure 5. Reduction Rules

6.4 Proof of Correctness of Erasure

We wish to prove that our erasure-based translation preserves both the types and semantics of the FCJ program being translated. In order to do so, we formalized our erasure implementation by defining an erasure function, $|\ldots|_{\Delta,\Gamma}$, that transforms FCJ expressions and types into FGJ_vexpressions and types. $|\texttt{e}|_{\Delta,\Gamma}$ represents the erasure of a FCJ expression e, under type enviroment Δ, and variable environment Γ. $|\texttt{T}|_{\Delta}$ represents the erasure of a FCJ type T under the type environment Δ. These rules are listed in Figure 6.

One rule requiring more explaination is E-NEW-FIELDS. Notice that extra expressions, $\overline{\texttt{e}}'$, are created and added as arguments to the expression $\texttt{new}\ |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_{\Delta}\texttt{(...)}$. This is a way to get around an idiosyncrasy in our formalism: a $\texttt{new C<}\overline{\texttt{T}}\texttt{>(...)}$ expression must take as arguments expressions that initialize C's fields, as well as C's superclass's fields. When $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ does not have a superclass in the FCJ type system, i.e. $\overline{\texttt{T}}$ do not satisfy the type condition placed on $\texttt{C<}\overline{\texttt{X}}\texttt{>}$'s conditional superclass, then in a well-formed expression $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$, $\overline{\texttt{e}}$ are only used for initializing C's fields. However, after erasure, C *unconditionally* extends a superclass. Thus, $\overline{\texttt{e}}$ are no longer sufficient as the arguments to the erased new expression. We overcome this issue by retrieving all the superclass's fields, unconditionally. We use the function fields_{uc}, which recurses up the chain of conditional superclasses and retrieves all of these superclasses' fields regardless of the state of the type conditions. We then create a new vector of expressions for these fields, by simply creating an expression `new Object()` for each field, and casting each expression to its required type. This is certainly a downcast that would fail, *if* that field is ever accessed. However, our type system ensures that a conditional superclass's fields are never accessed unless the conditional superclass's type-conditions are actually satisfied. And if the type-conditional conditions for a superclass is satisfied by a particular instantiation, these dummy expressions are never created during erasure. This makes our erasure safe. Note that this is merely a trick to get around the formal system's requirement that fields be initialized at the time of object creation. The actual implementation of cJ does not change arguments to method calls.

Certain erasure techniques mentioned in Section 5 are not formalized by our erasure rules. The lack of interfaces in the FCJ formalization means that we are not formalizing the insertion of dummy methods (i.e. methods that throw `UnsupportedOperationException`) into a class implementing a particular interface, where some of the interface's methods have type conditions that can never be satisfied by the implementing class. Similarly, we are not formalizing the insertion of dummy methods into a class extending another class, where some of the superclass's methods are never accessible through the subclass. The insertion of dummy methods in this case is purely to protect cJ code against code not compiled with a cJ compiler, and we feel it is acceptable to leave it out of our formalization at this point. We also do not formalize the translation of certain types down to raw types. This is due to the fact that raw types are not part of the FGJ_vformalization, which is what we are erasing FCJ down to[6]. Therefore, we divide type enviroment Δ into Δ_d Δ_{tc}, where Δ_d contains all type bounds declared at the declaration site of the type parameters, and Δ_{tc} contains the type bounds added by type-conditionals. The erasure function, then, is a partial function that is only well-defined for expressions and types in which a type instantiation is *ok* in Δ_d only (not Δ_d, Δ_{tc}). We refer to this restriction as the *raw type restriction* in the text that follows. The raw type restriction is evident in the type erasure rules: E-OBJECT, E-TYPE, and E-VAR. If we were to formalize the erasure down to raw types, as well, we would have the additional rule:

$$\frac{\Delta_d \nvdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}} \quad \Delta_d, \Delta_{tc} \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}} \qquad \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}} \triangleleft \overline{\texttt{R}}\texttt{>?} \triangleleft \texttt{D<}\overline{\texttt{S}}\texttt{>}\ \{\ldots\}}{|\texttt{C<}\overline{\texttt{T}}\texttt{>}|_{\Delta} = \texttt{C}} \quad \text{E-TYPE-RAW}$$

We now present the two theorems for the correctness of erasure. To distinguish between the typing rules and semantics of FCJ and FGJ_v, we use the subscript $_{FGJ_v}$ to indicate rules in the FGJ formalization. We use the subscript $_{FCJ}$, or no subscript, to indicate rules in the FCJ formalization. Our first theorem proves that erasure preserves typing:

Theorem 4 [Erasure Preserves Typing] For a program $(CT, \texttt{e})$, if CT is ok, and $\Delta; \Gamma \vdash_{FCJ} \texttt{e} \in \texttt{T}$ under the raw type restriction, then $|\Delta|_{\Delta}; |\Gamma|_{\Delta} \vdash_{FGJ_v} |\texttt{e}|_{\Delta,\Gamma} \in |\texttt{T}|_{\Delta}$.

To further prove that erasure preserves the semantics of the original FCJ program, we must prove that the execution of the program is preserved through erasure. One might expect that if a FCJ expression $\texttt{e} \longrightarrow_{FCJ} \texttt{e}'$, then $|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{FGJ_v} |\texttt{e}'|_{\Delta,\Gamma}$. However, because the erasure of FCJ to FGJ involves insertion of casts, preservation of execution in this sense is not true. Authors of FGJ [11] faced the same problem in proving that erasure from FGJ to FJ without generics preserves execution semantics. To overcome this problem, they defined an expansion function, $\stackrel{exp}{\Longrightarrow}$, as:

> Let us call a well-typed expression d an *expansion* of a well-typed expression e, written $\texttt{e} \stackrel{exp}{\Longrightarrow} \texttt{d}$, if d is obtained from e by some combination of (1) addition of zero or more synthetic upcasts, (2) replacement of some synthetic casts (D) with (C), where C is a supertype of D, or (3) removal of some sythetic casts.

Sythetic casts are those inserted by the erasure functions. We borrow this definition of $\stackrel{exp}{\Longrightarrow}$, and prove the following theorem:

[6] A formalization of FGJ with raw types was presented by Igarashi et. al.[12]. However, this formalization does not include variance, which we deem a very important feature in our language and a key differenciating factor. Thus, we use the formalization of FGJ_vinstead. Future work might involve enhancing [12] with variance, or [13] with raw types, and thus providing a more complete proof for our erasure.

Erasure Rules

$$|\mathtt{Object}|_\Delta = \mathtt{Object} \qquad \text{E-OBJECT}$$

$$\frac{\begin{array}{c}\Delta_d \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}} \\ \mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\mathtt{>?}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \{\ldots\}\end{array}}{|\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta = \mathtt{C<}|\overline{\mathtt{T}}|_\Delta\mathtt{>}} \qquad \text{E-TYPE}$$

$$\frac{\mathtt{X} \in dom(\Delta)}{|\mathtt{X}|_\Delta = \mathtt{X}} \qquad \text{E-TYPE-VAR}$$

$$|\mathtt{v}|_\Delta = \mathtt{v} \qquad \text{E-VARIANCE}$$

$$|\mathtt{x}|_{\Delta,\Gamma} = \mathtt{x} \qquad \text{E-VAR}$$

$$|\mathtt{(N)e}_0|_{\Delta;\Gamma} = \mathtt{(}|\mathtt{N}|_\Delta\mathtt{)}|\mathtt{e}_0|_{\Delta;\Gamma} \qquad \text{E-CAST}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f} \in \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{fields}_{FGJ_v}(|\mathtt{T}_0|_\Delta) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \quad \mathtt{S}_i = |\mathtt{T}|_\Delta\end{array}}{|\mathtt{e}_0\mathtt{.f}|_{\Delta,\Gamma} = |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}} \qquad \text{E-FIELD}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f} \in \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T} \\ \mathit{fields}_{FGJ_v}(|\mathtt{T}_0|_\Delta) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \quad \mathtt{S}_i \neq |\mathtt{T}|_\Delta\end{array}}{|\mathtt{e}_0\mathtt{.f}|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{T}|_\Delta\mathtt{)}|\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.f}} \qquad \text{E-FIELD-CAST}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{mtype}_{FGJ_v}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0 \quad \mathtt{U}_0 = |\mathtt{T}|_\Delta\end{array}}{|\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = |\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m<}|\overline{\mathtt{V}}|_\Delta\mathtt{>(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}} \qquad \text{E-INVK}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{mtype}_{FGJ_v}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0 \quad \mathtt{U}_0 \neq |\mathtt{T}|_\Delta\end{array}}{|\mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = \mathtt{(}|\mathtt{T}|_\Delta\mathtt{)}|\mathtt{e}_0|_{\Delta,\Gamma}\mathtt{.m<}|\overline{\mathtt{V}}|_\Delta\mathtt{>(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}} \qquad \text{E-INVK-CAST}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\mathtt{>?}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ldots\} \\ \mathtt{class\ D<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\mathtt{>\ <}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}'}\mathtt{>?}\triangleleft\mathtt{E<}\overline{\mathtt{U}}\mathtt{>}\{\ldots\} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}\end{array}}{|\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = \mathtt{new}\ |\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}} \qquad \text{E-NEW}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\mathtt{>?}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ldots\} \\ \mathtt{class\ D<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\mathtt{>\ <}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}'}\mathtt{>?}\triangleleft\mathtt{E<}\overline{\mathtt{U}}\mathtt{>}\{\ldots\} \quad \Delta \not\vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}} \\ \mathit{fields}_{uc}(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \quad \sharp(\overline{\mathtt{e}'}) = \sharp(\overline{\mathtt{D}}) \quad \mathtt{e}'_i = \mathtt{(D}_i\mathtt{)new\ Object()}\end{array}}{|\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma} = \mathtt{new}\ |\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta\mathtt{(}|\overline{\mathtt{e}'}|_{\Delta,\Gamma}\mathtt{,}\ |\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}} \qquad \text{E-NEW-FIELDS}$$

$$\frac{\begin{array}{c}\Gamma = \overline{\mathtt{x}}:\overline{\mathtt{T}}\mathtt{,this:C<}\overline{\mathtt{X}}\mathtt{>} \quad \Delta_d = \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{Q}} \\ \Delta_{tc} = \overline{\mathtt{X}} <: \overline{\mathtt{R}} \quad \Delta = \Delta_d, \Delta_{tc}\end{array}}{\begin{array}{c}|\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\mathtt{>?<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{Q}}\mathtt{>\ T}_0\ \mathtt{m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)}\ |_{\Delta,\Gamma} = \\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft|\overline{\mathtt{Q}}|_\Delta\mathtt{>}\ |\mathtt{T}_0|_\Delta\ \mathtt{m\ (}|\overline{\mathtt{T}}|_\Delta\ \overline{\mathtt{x}'}\mathtt{)}\ \{\ \uparrow[\mathtt{(}|\overline{\mathtt{T}}|_\Delta\mathtt{)}\ \overline{\mathtt{x}'}/\overline{\mathtt{x}}]\ |\mathtt{e}|_{\Delta,\Gamma}\ \}\end{array}} \qquad \text{E-METHOD}$$

$$\frac{\Delta_d = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \quad \Delta_{tc} = \overline{\mathtt{X}} <: \overline{\mathtt{R}} \quad \Delta = \Delta_d, \Delta_{tc}}{\begin{array}{c}|\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\mathtt{>?}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}| = \\ \mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft|\overline{\mathtt{N}}|_{\Delta_d}\mathtt{>}\triangleleft|\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}|_\Delta\ \{\ |\overline{\mathtt{S}}|_{\Delta_d}\ \overline{\mathtt{f}};\ |\overline{\mathtt{M}}|_{\Delta_d,\Gamma}\ \}\end{array}} \qquad \text{E-CLASS}$$

Figure 6. Erasure Rules

Theorem 5 [Erasure Preserves Execution Modulo Expansion]: If $\Delta;\Gamma \vdash_{FCJ} \mathtt{e} \in \mathtt{T}$, $\mathtt{e} \longrightarrow_{FCJ}{}^* \mathtt{e}'$, and e and e′ are well-typed under the raw type restriction, then there exists some FGJ$_v$expression d′such that $|\mathtt{e}'|_{\Delta,\Gamma} \stackrel{exp}{\Longrightarrow} \mathtt{d}'$ and $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{FGJ_v}{}^* \mathtt{d}'$.

We refer readers to the appendix for the detailed proofs of the above two theorems. The proofs refer to FGJ$_v$typing and reduction rules. We refer users to [13] for the complete listing of these rules.

7. Related Work

cJ is related to several programming language and software engineering concepts. These range from mainstream modularization techniques to meta-programming and conditional compilation approaches.

Clearly the idea of a type-conditional is closely related to conditional compilation, as with the C/C++ preprocessor "`#ifdef`" construct. Although `#ifdef` is valuable for configuring large projects, it addresses very different needs from cJ. Conditional compilation gives low-level manual control for software configuration. In the context of a portable language, like Java, an `#ifdef` statement becomes less useful. At the same time, conditional compilation suffers from the lack of any form of safety control. The use of conditional flags may be inconsistent, resulting in invalid configurations that are not detected until one attempts to select them. There has been work on adding some safety to conditional compilation by analyzing all configurations of a C program, and there is evidence that such a heuristic approach may work in several contexts—especially for refactoring [8]. Nevertheless, cJ offers full static safety guarantees, eliminating the problem altogether. Furthermore, the type-conditions of cJ are structured, richer than mere propositions, and well-integrated with the Java type system.

Conditional methods have been explored in OO language in work at least as early as CLU [21]. Nevertheless, CLU does not support subtyping, so the language context of this work is notably different. It is, thus, difficult to compare CLU to cJ, where our main goal is to maintain static type safety, yet, at the same time, maintain a clean subtyping hierarchy used for abstraction. Past work on optional methods in Java was also presented by Myers et al. [24]. This was in the context of a proposal for adding genericity to Java, and it includes the feature of attaching `where` clauses to individual methods. The conditions on the `where` clauses, however, can only be "structural" constraints—i.e., does type parameter `T` provide method `void foo();`. This mechanism does not support conditional subtyping—e.g., it is not possible to express that a `Collection` is `Comparable`, if the elements it holds are `Comparable`. Even more importantly, the work by Myers et al. does not support type-safe abstraction over classes with conditional methods, as in the interaction of cJ with variance.

More recently, Emir et al. presented an extension to C# to support generalized type constraints on methods [7]. This extension allows both upper and lower bound type conditions on methods. This is similar to cJ in that methods exist conditionally based on the instantiation of parametric types. Nevertheless, there are significant differences, and in future work we plan to pursue combining the two approaches. cJ currently does not allow using the type parameter of a polymorphic method inside a type conditional—a crucial feature in Emir et al.'s work. At the same time, cJ has several features not found in the generalized type constraints approach. First, cJ supports conditional definitions of fields, as well as conditional subtyping. Furthermore, the cJ (and Java) form of variance we examined earlier is a "use-site variance" mechanism as opposed to the "definition-site variance" supported in Emir et al.'s work. In addition to being part of standard Java, we believe that use-site variance is a mechanism better suited for imperative programming languages in general. In this setting, a single class definition is unlikely to produce types that are purely co-variant, purely contra-variant, or purely bi-variant. Instead defining a class will likely implicitly yield a co-variant part, a contra-variant part, etc. In use-site variance these subsets of the class functionality are derived automatically from a single definition. In definition-site variance, they have to be explicitly separated out into distinct interfaces by the programmer. Thus, we believe use-site variance to be a more user-friendly system and a natural fit for Java.

In languages with (multi-)methods outside classes, the work on constraint-based polymorphism in Cecil [22] is related to cJ. Cecil provides users the ability to add constraints to both methods and supertypes. The constraints can be subtyping constraints, as well as structural constraints, requiring a type to provide a particular method. This is a very different context from that of our work, however. Furthermore, the Cecil type system does not have an analogue of our variance approach to abstracting over all objects with or without some of the conditionally defined members.

Our type-conditional is also related to traditional meta-programming techniques, which offer mechanisms for programs to generate other programs. Recent approaches, such as SafeGen [9] and Genoupe [5] attempt to add safety guarantees to meta-programming, yet maintain expressiveness. Nevertheless, these approaches either fail to achieve full safety, or reject programs in a way that is not transparent to the programmer. Neither mechanism integrates seamlessly with a programming language, as cJ does. For instance, SafeGen uses an automatic theorem prover in order to prove well-formedness properties of every produced program. Yet this approach is not guaranteed to always produce accurate results, as the theorem prover may not terminate. Similarly, Genoupe suffers from potential unsafeties, as its reasoning on the safety of generated code relies on the equivalence of arbitrarily complex expressions from the generator source code, which is undecidable to determine. (Based on the published description, it seems that Genoupe unsoundly estimates the run-time equivalence of expressions based on syntactic similarity.)

It is tempting to find parallels between cJ and advanced OO modularization mechanisms such as traits [6, 27], mixins [2], or mixin layers [29]. These approaches vary in expressiveness and several of them are insufficient for solving the combinatorial explosion problems identified in Section 3. For instance, C++-based mixins or mixin layers would still require a large number of compositions, with explicit subtyping links added among them, in order to express the required functionality of the Java Collections Framework. It is possible that a traits-based mechanism could serve to alleviate many of the problems in the Java Collections Framework (albeit with a complete rewrite). Nevertheless, there is no such mechanism currently for Java that would tie well with the rest of the language's type system (e.g., variance) and execution model. Furthermore, no mixin or traits mechanism offers capabilities similar to those shown in Section 3.1, i.e., the ability to add extra members only when a type parameter that is already used for other purposes has a certain subtyping property.

Type-conditionals in cJ can be viewed as being similar to type-safe variant records work—e.g., [26]. Nevertheless, variant records mechanisms typically try to address the problem of *run-time* variability with static type-safety. cJ is not concerned with changes to the type of a variable during run-time. Instead, cJ focuses on the static configurability of components. The techniques used to ensure static type safety in the case of variant records and in the case of cJ show this difference clearly: statically safe variant records typically require the programmer to specify what code will get executed for any possible type. Indeed, this is the best one can hope when the object can indeed vary at run-time. In contrast, cJ statically ensures that the legal operations on an object are fully known.

Configuring generic code is also reminiscent of techniques in C++ template programming [1, 15]. Fundamentally, C++ templates offer a powerful (Turing-complete) but unsafe language for configuring types: there is little static checking capability beyond the checking of templates after instantiation. Furthermore, there is no way to guarantee that a template computation will even terminate. The C++ community has developed ideas on statically validating the input to a template [23, 28] and the general idea of *concepts* has emerged and even developed as a language-independent notion [16]. Nevertheless, concept-based techniques concentrate on validating the type parameters of a generic class, rather than configuring it under static conditions. cJ offers the ability to configure classes based on subtyping conditions, without sacrificing static type safety and with a smooth integration in the base language.

cJ can be viewed as an instance of the aspect-oriented programming paradigm [19], because of its ability to allow a class to be configured based on the structure of a different type hierarchy (representing a cross-cutting concern). As we demonstrated with the cJ implementation of the JCF, the cross-cutting concern "modifiability" is separated in the type system from the intrinsic form of a data structure (e.g., whether it is a list, or a set, or a map). The type system does maintain concepts such as "modifiable list", "unmodifiable map", etc., but these are derived from their component types. In fact, the cJ reimplementation of the JCF can be compared to previous work that uses AOP to enforce consistency in data structure and behavior [20, 25]—in the JCF, consistency in the usage of data structures along cross-cutting dimensions is enforced by the cJ type system. Nevertheless, cJ differs from common aspect-oriented languages like AspectJ [18] in that separation of concerns in cJ is confined to the type level. cJ does not offer any cross-cutting features at the level of code or method definitions: these are interspersed throughout traditional Java language components (i.e., classes) and not collected in a single entity. Thus, what cJ has to offer is orthogonal to traditional aspect languages and it is interesting to consider integrating their advantages in future work.

8. Conclusions

We presented cJ: an extension of Java that allows declaring class and interface members and supertypes provisionally, under subtyping conditions on parameter types. cJ's power lies in that it allows the composition of orthogonal type hierarchies concisely (avoiding a combinatorial blowup of the number of declared types) yet without sacrificing static type safety. Thus, cJ has a cross-cutting flavor at the level of type hierarchies: the user can define separate aspects of a type hierarchy independently and combine them using cJ type-conditionals to form the complete set of expressible types.

We believe that cJ offers an interesting combination of expressiveness and safety, together with a smooth integration with a representative mainstream OO language. cJ's ability to solve real problems is demonstrated by applying it to the Java Collections Framework. cJ addresses the Collection Framework's well-known shortcomings, eliminating the possibility of run-time errors for unsupported operations without sacrificing conciseness.

Acknowledgments

Phil Wadler and Martin Odersky suggested the need for conditional subtyping and offered many excellent comments that helped strengthen the paper. Multiple anonymous reviewers also made very helpful suggestions. We gratefully acknowledge support by the NSF under Grant CCR-0238289.

References

[1] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[4] W.-N. Chin, F. Craciun, S.-C. Khoo, and C. Popeea. A flow-based approach for variant parametric types. In *Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'06)*, volume 41, pages 273–290, New York, NY, USA, 2006. ACM Press.

[5] D. Draheim, C. Lutteroth, and G. Weber. A type system for reflective program generators. In *Generative Programming and Component Engineering (GPCE)*, 2005.

[6] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[7] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2006.

[8] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *21st International Conference of Software Maintenance*. IEEE, 2005.

[9] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Generative Programming and Component Engineering (GPCE)*, pages 309–326, 2005.

[10] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. Technical report, 2006. `http://www.cc.gatech.edu/~ssh/cj/cjfull.pdf`.

[11] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[13] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *European Conference on Object-oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 441–469. Springer-Verlag, 2002.

[14] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006.

[15] ISO Standards Committee. ISO/IEC standard 14882: Programming languages – C++, 1998.

[16] J. Järvi, J. Willcock, and A. Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 2005.

[17] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2001.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[19] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[20] P. Lam, V. Kuncak, and M. Rinard. Crosscutting techniques in program specification and analysis. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 169–180, New York, NY, USA, 2005. ACM Press.

[21] B. Liskov. *CLU Reference Manual*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

[22] V. Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *OOPSLA '98 Conference Proceedings*, volume 33(10), pages 388–411, 1998.

[23] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *C++ Template Programming Workshop*, Oct. 2000.

[24] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, New York, NY, 1997.

[25] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD'06: Proceedings of the 5th International Conference on Aspect-oriented Software Development*, pages 75–86, New York, NY, USA, 2006. ACM Press.

[26] D. Rémy. Type checking records and variants in a natural extension of ML. In *Symposium on Principles of Programming Languages (POPL)*, pages 77–88. ACM Press, 1989.

[27] N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *European Conference on Object Oriented Programming (ECOOP)*. Springer LNCS 2743, 2003.

[28] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *C++ Template Programming Workshop*, Oct. 2000.

[29] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[30] M. Torgersen, E. Ernst, and C. P. Hansen. Wild fj. In *FOOL 2005: Foundations of Object-Oriented Languages*, Long Beach, California, 2005. ACM Press.

[31] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *OOPS track of the Symposium on Applied Computing (SAC)*, 2004.

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

[33] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *31st Symposium on Principles of Programming Languages (POPL)*. ACM, 2004.

Appendix

We provide here proofs for the theorems presented in Section 6, as well as their supporting lemmas. Readers familiar with the proofs presented in [11] and [13] will find these proofs similar.

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

Proof. We prove using induction on the derivation of $\mathtt{e}\longrightarrow\mathtt{e}'$.

Case R-FIELD:

$\mathtt{e} = \mathtt{new\ C<\overline{T}>(\overline{e}).f}_i \quad \mathtt{e}'=\mathtt{e}_i$

$\mathit{fields}(\mathtt{C<\overline{T}>})=\mathtt{\overline{U}\ \overline{f}} \qquad \Delta;\Gamma\vdash\mathtt{new\ C<\overline{T}>(\overline{e}).f}_i\in\mathtt{T}$

By T-FIELD, T-NEW, and O-REFL:

$\Delta;\Gamma\vdash\mathtt{new\ C<\overline{T}>(\overline{e})}\in\mathtt{C<\overline{T}>} \qquad \Delta\vdash\mathit{bound}_\Delta(\mathtt{C<\overline{T}>})\Uparrow^{\emptyset}\mathtt{C<\overline{T}>}$

$\mathit{fields}(\mathtt{C<\overline{T}>})=\mathtt{\overline{U}\ \overline{f}} \qquad \mathtt{U}_i\Downarrow_\emptyset\mathtt{U}_i$, where $\mathtt{U}_i=\mathtt{T}$

$\Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \qquad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

Let $\mathtt{S}=\mathtt{S}_i$

Case R-INVK:

$\mathtt{e}=\mathtt{new\ C<\overline{T}>(\overline{e}).<\overline{V}>m(\overline{d})}$

$\mathtt{e}'=[\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{new\ C<\overline{T}>(\overline{e})/this}]\mathtt{e}_0$

$\mathit{mbody}(\mathtt{m<\overline{V}>,\ C<\overline{T}>}) = (\overline{\mathtt{x}},\ \mathtt{e}_0)$

By T-INVK, T-NEW, O-REFL:

$\Delta;\Gamma\vdash\mathtt{new\ C<\overline{T}>(\overline{e})}\in\mathtt{C<\overline{T}>}$

$\Delta\vdash\mathit{bound}_\Delta(\mathtt{C<\overline{T}>})\Uparrow^{\emptyset}\mathtt{C<\overline{T}>}$

$\mathit{mtype}(\Delta,\ \mathtt{m},\ \mathtt{C<\overline{T}>})=\mathtt{<\overline{Y}\lhd\overline{P}>\overline{U}}\rightarrow\mathtt{U}_0 \qquad \Delta\vdash\mathtt{C<\overline{T}>}\ \mathit{ok}$

$\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\notin\Delta'$ for any $\overline{\mathtt{Q}}$ $\qquad \Delta\vdash\overline{\mathtt{V}}\ \mathit{ok}$

$\Delta\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}}$

$\Delta\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \qquad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_\emptyset\mathtt{T}$

By Lemma 3.1, for some N, $\mathtt{S}_0$,

$\Delta\vdash\mathtt{C<\overline{T}>}<:\mathtt{N} \qquad \Delta\vdash\mathtt{N}\ \mathit{ok}$

$\Delta\vdash\mathtt{S}_0<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \qquad \Delta;\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{N}\vdash\mathtt{e}_0\in\mathtt{S}_0$

By Lemma 3.4, there is some $\mathtt{S'}_0$ such that:

$\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}},\mathtt{new\ C<\overline{T}>/this}]\mathtt{e}_0\in\mathtt{S}_0' \qquad \Delta\vdash\mathtt{S}_0'<:\mathtt{S}_0$

By S-TRANS, $\Delta\vdash\mathtt{S}_0'<:\mathtt{T}$. Let $\mathtt{S}=\mathtt{S'}_0$.

Case R-CAST:

$\mathtt{e}=\mathtt{(T)new\ C<\overline{T}>(\overline{e})} \qquad \mathtt{e}'=\mathtt{new\ C<\overline{T}>(\overline{e})} \qquad \emptyset\vdash\mathtt{C<\overline{T}>}<:\mathtt{T}$

By T-CAST, T-NEW:

$\Delta\vdash\mathtt{new\ C<\overline{T}>(\overline{e})}\in\mathtt{C<\overline{T}>}$

Let $\mathtt{S}=\mathtt{C<\overline{T}>}$.

Case RC-FIELD:

$\mathtt{e}=\mathtt{e}_0\mathtt{.f} \quad \mathtt{e}'=\mathtt{e}_0'\mathtt{.f} \quad \mathtt{e}_0\longrightarrow\mathtt{e}_0'$

By T-FIELD:

$\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0 \qquad \Delta\vdash\mathit{bound}_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C<\overline{U}>}$

$\mathit{fields}(\mathit{bound}_\Delta(\mathtt{C<\overline{U}>}))=\overline{\mathtt{S}}\ \overline{\mathtt{f}} \qquad \mathtt{S}_i\Downarrow_{\Delta'}\mathtt{T}$

By induction, there exists $\mathtt{S}_0$ such that:

$\Delta;\Gamma\vdash\mathtt{e}_0'\in\mathtt{S}_0 \qquad \Delta\vdash\mathtt{S}_0<:\mathtt{T}_0$

By Lemma 3.2, for some Δ'', $\mathtt{D<\overline{S}>}$, $\overline{\mathtt{V}}$, $\mathtt{V}_0'$,

$\Delta\vdash\mathit{bound}_\Delta(\mathtt{S}_0)\Uparrow^{\Delta''}\mathtt{D<\overline{S}>} \qquad \mathit{fields}(\mathtt{D<\overline{S}>})=\ldots,\overline{\mathtt{V}}\ \overline{\mathtt{f}}$

$\mathtt{V}_i\Downarrow_{\Delta''}\mathtt{V}_0' \qquad \Delta\vdash\mathtt{V}_0'<:\mathtt{T}$

Let $\mathtt{S}=\mathtt{V}_0'$.

Case RC-INV-RECV:

$\mathtt{e}=\mathtt{e}_0\mathtt{.<\overline{V}>m(\overline{e})} \qquad \mathtt{e}'=\mathtt{e}_0'\mathtt{.<\overline{V}>m(\overline{e})} \qquad \mathtt{e}_0\longrightarrow\mathtt{e}_0'$

By T-INVK, and induction, for some $\mathtt{T}_0'$:

$\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0 \qquad \Delta;\Gamma\vdash\mathtt{e}_0'\in\mathtt{T}_0'$

$\Delta\vdash\mathit{bound}_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C<\overline{T}>} \qquad \Delta\vdash\mathtt{T}_0'<:\mathtt{T}_0$

$\mathit{mtype}(\Delta,\Delta',\ \mathtt{m},\ \mathtt{C<\overline{T}>})=\mathtt{<\overline{Y}\lhd\overline{P}>\overline{U}}\rightarrow\mathtt{U}_0 \qquad \Delta\vdash\mathtt{C<\overline{T}>}\ \mathit{ok}$

$\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\notin\Delta'$ for any $\overline{\mathtt{Q}}$ $\qquad \Delta\vdash\overline{\mathtt{V}}\ \mathit{ok}$

$\Delta,\Delta'\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$

$\Delta,\Delta'\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \qquad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta'}\mathtt{T}$

By Lemma 3.3,

$\Delta\vdash\mathit{bound}_\Delta(\mathtt{T}_0')\Uparrow^{\Delta''}\mathtt{D<\overline{S}'>}$

$\mathit{mtype}(\Delta,\Delta'',\ \mathtt{m},\ \mathtt{D<\overline{S}'>})=\mathtt{<\overline{Y}\lhd\overline{P}'>\overline{U}'}\rightarrow\mathtt{U}_0'$

$\Delta,\Delta''\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}' \qquad \Delta,\Delta''\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0'\Downarrow_{\Delta''}\mathtt{V}_0' \qquad \Delta\vdash\mathtt{V}_0'<:\mathtt{T}$

By T-INVK, let $\mathtt{S}=\mathtt{V}_0'$.

Case RC-INV-ARG:

$\mathtt{e}=\mathtt{e}_0\mathtt{.<\overline{V}>m(\ldots,e}_i\mathtt{,\ldots)} \qquad \mathtt{e}'=\mathtt{e}_0\mathtt{.<\overline{V}>m(\ldots,e}_i'\mathtt{,\ldots)}$

$\mathtt{e}_i\rightarrow\mathtt{e}_i'$

By T-INVK:

$\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$

$\Delta\vdash\mathit{bound}_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C<\overline{T}>}$

$\mathit{mtype}(\Delta,\Delta',\ \mathtt{m},\ \mathtt{C<\overline{T}>})=\mathtt{<\overline{Y}\lhd\overline{P}>\overline{U}}\rightarrow\mathtt{U}_0 \qquad \Delta\vdash\mathtt{C<\overline{T}>}\ \mathit{ok}$

$\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\notin\Delta'$ for any $\overline{\mathtt{Q}}$ $\qquad \Delta\vdash\overline{\mathtt{V}}\ \mathit{ok}$

$\Delta,\Delta'\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \qquad \Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$

$\Delta,\Delta'\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \qquad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta'}\mathtt{T}$

$\Delta;\Gamma\vdash\mathtt{e}_i\in\mathtt{S}_i$. By induction, for some $\mathtt{S}_i'$,

$\Delta;\Gamma\vdash\mathtt{e}_i'\in\mathtt{S}_i' \qquad \Delta\vdash\mathtt{S}_i'<:\mathtt{S}_i$

By S-TRANS and T-INVK,

$\Delta;\Gamma\vdash\mathtt{e}_0\mathtt{.<\overline{V}>m(\ldots,e}_i'\mathtt{,\ldots)}\in\mathtt{T}$

Case RC-NEW-ARG:

$\mathtt{e}=\mathtt{new\ C<\overline{T}>(\ldots,e}_i\mathtt{,\ldots)} \qquad \mathtt{e}'=\mathtt{new\ C<\overline{T}>(\ldots,e}_i'\mathtt{,\ldots)}$

$\mathtt{e}_i\longrightarrow\mathtt{e}_i'$

By T-NEW: $\mathtt{T}=\mathtt{C<\overline{T}>}$

$\Delta\vdash\mathtt{C<\overline{T}>}\ \mathit{ok} \qquad \mathit{fields}(\mathtt{C<\overline{T}>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$

$\Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \qquad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

$\Delta;\Gamma\vdash\mathtt{e}_i\in\mathtt{S}_i$. By induction, for some $\mathtt{S}_i'$,

$\Delta;\Gamma\vdash\mathtt{e}_i'\in\mathtt{S}_i' \qquad \Delta\vdash\mathtt{S}_i'<:\mathtt{S}_i$

By S-TRANS and T-NEW,

$\Delta;\Gamma\vdash\mathtt{new\ C<\overline{T}>(\ldots,e}_i'\mathtt{,\ldots)}\in\mathtt{C<\overline{T}>}$

Case RC-CAST:

$\mathtt{e}=\mathtt{(T)e}_0 \qquad \mathtt{e}'=\mathtt{(T)e}_0' \qquad \mathtt{e}_0\rightarrow\mathtt{e}_0'$

There are two cases when $\Delta;\Gamma\vdash\mathtt{(T)e}_0\in\mathtt{T}$:

1) T-CAST:

$\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0 \qquad \Delta\vdash\mathtt{T}\ \mathit{ok}$

$\Delta\vdash\mathtt{T}_0<:\mathit{bound}_\Delta(\mathtt{T})$ or $\Delta\vdash\mathit{bound}_\Delta(\mathtt{T})<:\mathtt{T}$

By induction and S-TRANS, there is a $\mathtt{T}_0'$ such that

$\Delta;\Gamma\vdash\mathtt{e}_0'\in\mathtt{T}_0' \qquad \Delta\vdash\mathtt{T}_0'<:\mathtt{T}_0$

$\Delta\vdash\mathtt{T}_0'<:\mathit{bound}_\Delta(\mathtt{T})$

By T-CAST, $\Delta;\Gamma\vdash\mathtt{(T)e}_0'\in\mathtt{T}$.

2) T-SCAST: similar to the case by T-CAST.

□

Theorem 2 (Progress). *Let* e *be a well-typed expression.*

1. If e *has* $\mathtt{new\ C<\overline{T}>(\overline{e}).f}$ *as a subexpression, then* $\mathit{fields}(\emptyset,\ \mathtt{C<\overline{T}>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$, *and* $\mathtt{f}=\mathtt{f}_i$.

2. If e *has* $\mathtt{new\ C<\overline{T}>(\overline{e}).m(\overline{d})}$ *as a subexpression, then* $\mathit{mbody}(\emptyset,\ \mathtt{m},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$ *and* $|\overline{\mathtt{x}}|=|\overline{\mathtt{d}}|$.

Proof. 1. Immediate from T-FIELD, T-NEW. 2. Immediate from T-INVK, T-NEW, and the definition of *mbody*. □

Theorem 3 (Type Soundness). *If* $\emptyset;\emptyset\vdash\mathtt{e}\in\mathtt{T}$ *and* $\mathtt{e}\rightarrow^*\mathtt{e'}$ *being a normal form, then* e' *is either a value* v *such that* $\emptyset;\emptyset\vdash v\in\mathtt{S}$ *and* $\emptyset\vdash\mathtt{S}<:\mathtt{T}$ *for some* S, *or an expression that includes* $\mathtt{(T)new\ C<\overline{T}>(\overline{e})}$ *where* $\emptyset\not\vdash\mathtt{C<\overline{T}>}<:\mathtt{T}$.

Proof. Proof is immediate from Theorem 1 and Theorem 2. □

Lemma 3.1. *If* $\mathit{mtype}(\Delta,\ \mathtt{m},\ \mathtt{C<\overline{T}>})=\mathtt{<\overline{Y}\lhd\overline{P}>\overline{U}}\rightarrow\mathtt{U}_0$, $\mathit{mbody}(\Delta,\ \mathtt{m<\overline{V}>},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\ \mathtt{e}_0)$, $\Delta\vdash\mathtt{C<\overline{T}>}\ \mathit{ok}$, $\Delta\vdash\overline{\mathtt{V}}\ \mathit{ok}$, *and* $\Delta\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$, *then there exist some* N *and* S *such that*

$\Delta \vdash \mathtt{C<\overline{T}>} <: \mathtt{N}$ *and* $\Delta \vdash \mathtt{N}$ *ok and* $\Delta \vdash \mathtt{S} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0$ *and* $\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this}:\mathtt{N} \vdash \mathtt{e}_0 \in \mathtt{S}$.

Proof. Prove by induction on the derivation of $mtype$.

$Case$ MT-CLASS:

$\mathtt{<\overline{Y} \lhd \overline{P}>\overline{U}} \rightarrow \mathtt{U}_0 = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<\overline{Y} \lhd \overline{P}'>\overline{U}'} \rightarrow \mathtt{U}'_0)$

$\overline{\mathtt{P}} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}' \quad \overline{\mathtt{U}} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}' \quad \mathtt{U}_0 = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}'_0$

$CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{H}>?\lhd D<\overline{S}>\ \{\ldots\overline{M}\}}$

$\mathtt{<\overline{X} \lhd \overline{R}>?<\overline{Y} \lhd \overline{P}'>U'_0\ m\ (\overline{U}'\ \overline{x})\ \{\uparrow e'_0;\}} \in \overline{\mathtt{M}}$

$\Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$

By MB-CLASS:

$mbody(\Delta, \mathtt{m<\overline{V}>}, \mathtt{C<\overline{T}>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}'_0)$

$\mathtt{e}_0 = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}'_0$

By T-METHOD:

$\overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{X}} <: \overline{\mathtt{R}} \quad \Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{H}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}'$

$\Delta' \vdash \overline{\mathtt{P}}', \overline{\mathtt{H}}, \overline{\mathtt{U}}', \mathtt{U}'_0\ ok \quad \Delta'; \overline{\mathtt{x}} : \overline{\mathtt{U}}', \mathtt{this}:\mathtt{C<\overline{X}>} \vdash \mathtt{e}'_0 \in \mathtt{S}_0$

$\Delta \vdash \mathtt{S}_0 <: \mathtt{U}'_0$

$CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{H}>?\lhd D<\overline{S}>\ \{\ldots\overline{M}\}}$

$override(\mathtt{m}, \mathtt{<\overline{X} \lhd \overline{H}>?D<\overline{S}>}, \mathtt{<\overline{X} \lhd \overline{R}>?\overline{U}'} \rightarrow \mathtt{U}'_0)$

By WF-CLASS:

$\Delta \vdash \overline{\mathtt{T}}\ ok \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By Lemma 3.4.1, 3.4.2, 3.4.3, 3.4.6,

$\Delta \vdash [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0 <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}'_0$

$\Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this}:\mathtt{C<\overline{T}>} \vdash \mathtt{e}'_0 \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0$

Let $\mathtt{N} = \mathtt{C<\overline{T}>}$, $\mathtt{S} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0$.

$Case$ MT-SUPER:

$mtype(\Delta, \mathtt{m}, \mathtt{C<\overline{T}>}) = mtype(\Delta, \mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{S}>})$

$CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{H}>?\lhd D<\overline{S}>\ \{\ldots\overline{M}\}}$

m is not defined in $\overline{\mathtt{M}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{H}}$

By MB-SUPER:

$mbody(\Delta, \mathtt{m<\overline{V}>}, \mathtt{C<\overline{T}>}) = mbody(\Delta, \mathtt{m<\overline{V}>}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{S}>})$

By WF-CLASS and $\Delta \vdash \mathtt{C<\overline{T}>}\ ok$, we have

$\Delta \vdash \overline{\mathtt{T}}\ ok \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By T-CLASS, $\overline{\mathtt{X}} <: \overline{\mathtt{H}} \vdash \mathtt{D<\overline{S}>}\ ok$

By Lemma 3.4.5, $\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{S}>}\ ok$

By induction, there is N, S such that

$\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{S}>} <: \mathtt{N} \quad \Delta \vdash \mathtt{N}\ ok$

$\Delta \vdash \mathtt{S} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \quad \Delta; \overline{\mathtt{x}} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}, \mathtt{this}:\mathtt{N} \vdash \mathtt{e}_0 \in \mathtt{S}$

By S-CLASS, $\Delta \vdash \mathtt{C<\overline{T}>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<\overline{S}>}$

By S-TRANS, $\Delta \vdash \mathtt{C<\overline{T}>} <: \mathtt{N}$.

□

Lemma 3.2. *If* Δ *has non-variable bounds and* $\Delta \vdash bound_\Delta(\mathtt{T}) \Uparrow^{\Delta_1} \mathtt{C<\overline{T}>}$ *and* $fields(\Delta, \mathtt{C<\overline{T}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ *and* $\mathtt{U}_i \Downarrow_{\Delta_1} \mathtt{U}'$, *then for any* S *such that* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \mathtt{S}$ *ok, it holds that* $\Delta \vdash bound_\Delta(\mathtt{S}) \Uparrow^{\Delta_2} \mathtt{D<\overline{S}>}$ *and* $fields(\Delta, \mathtt{D<\overline{S}>}) = \ldots, \overline{\mathtt{V}}\ \overline{\mathtt{f}}$ *and* $\mathtt{V}_i \Downarrow_{\Delta_2} \mathtt{V}'$ *and* $\Delta \vdash \mathtt{V}_0 <: \mathtt{U}_0$ *for some* Δ_2, $\overline{\mathtt{f}}$, $\mathtt{D<\overline{S}>}$, $\overline{\mathtt{V}}$, *and* $\mathtt{V}'$.

Proof. We prove by induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

$Case$ S-REFL:

Let $\Delta_2 = \Delta_1$, $\mathtt{D<\overline{S}>} = \mathtt{C<\overline{T}>}$, $\overline{\mathtt{V}} = \overline{\mathtt{U}}$, and $\mathtt{V}_0 = \mathtt{U}_0$.

$Case$ S-TRANS:

Let there be a type U such that $\Delta \vdash \mathtt{S} <: \mathtt{U}$, $\Delta \vdash \mathtt{U} <: \mathtt{T}$, and $\Delta \vdash \mathtt{U}$ ok.

By induction, there exists Δ_u, $\overline{\mathtt{f}}_u$, $\mathtt{D}_u\mathtt{<\overline{S}}_u\mathtt{>}$, $\overline{\mathtt{V}}_u$, and $\mathtt{V}_{u_0}$ such that:

$\Delta \vdash bound_\Delta(\mathtt{U}) \Uparrow^{\Delta_u} \mathtt{D}_u\mathtt{<\overline{S}}_u\mathtt{>} \quad \mathtt{V}_{u_i} \Downarrow_{\Delta_u} \mathtt{V}_{u_0}$

$fields(\mathtt{D}_u\mathtt{<\overline{S}}_u\mathtt{>}) = \ldots, \overline{\mathtt{V}}_u \overline{\mathtt{f}}_u \quad \Delta \vdash \mathtt{V}_{u_0} <: \mathtt{U}_0$

Also by induction, there exists Δ_s, $\overline{\mathtt{f}}_s$, $\mathtt{D}_s\mathtt{<\overline{S}}_s\mathtt{>}$, $\overline{\mathtt{V}}_s$, and $\mathtt{V}_{s_0}$ such that:

$\Delta \vdash bound_\Delta(\mathtt{U}) \Uparrow^{\Delta_s} \mathtt{D}_s\mathtt{<\overline{S}}_s\mathtt{>} \quad \mathtt{V}_{u_i} \Downarrow_{\Delta_s} \mathtt{V}_{s_0}$

$fields(\mathtt{D}_s\mathtt{<\overline{S}}_s\mathtt{>}) = \ldots, \overline{\mathtt{V}}_s \overline{\mathtt{f}}_s \quad \Delta \vdash \mathtt{V}_{s_0} <: \mathtt{V}_{u_0}$

Let $\Delta_2 = \Delta_u, \Delta_s$, $\overline{\mathtt{f}} = \overline{\mathtt{f}}_s$, $\mathtt{D<\overline{S}>} = \mathtt{D}_s\mathtt{<\overline{S}}_s\mathtt{>}$, $\overline{\mathtt{V}} = \overline{\mathtt{V}}_s$, and $\mathtt{V}_0 = \mathtt{V}_{s_0}$.

$Case$ S-UBOUND:

$bound(\mathtt{S}) = bound(\mathtt{T})$.

Let $\Delta_1 = \Delta_2$, $\mathtt{D<\overline{S}>} = \mathtt{C<\overline{T}>}$, $\overline{\mathtt{V}} = \overline{\mathtt{U}}$, and $\mathtt{V}_0 = \mathtt{U}_0$.

$Case$ S-LBOUND:

Cannot happen: X cannot have a $bound$ if it has a - for variance notation, and thus preconditions of this lemma cannot be satisfied.

$Case$ S-CLASS: $\mathtt{S} = \mathtt{D<\overline{W}>}$

$CT(\mathtt{D}) = \mathtt{class\ D<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{R}>?\lhd C<\overline{S}'>\ \{\ \overline{G}\ \overline{g};\ \overline{M}\ \}}$

$\Delta \vdash \mathtt{D<\overline{W}>} \Uparrow^{\Delta_2} \mathtt{D<\overline{W}'>} \quad \Delta \vdash \overline{\mathtt{W}}' <: [\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$

$([\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C<\overline{S}'>}) \Downarrow_{\Delta_2} \mathtt{T}$

By definition of $fields$,

$fields(\mathtt{D<\overline{W}'>}) = fields([\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C<\overline{S}'>}), \overline{\mathtt{G}}\ \overline{\mathtt{g}}$

Let $fields(([\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C<\overline{S}'>}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}}$.

By Lemma 3.4.9, $\mathtt{V}_i \Downarrow_{\Delta_2} \mathtt{U}_0$.

$Case$ S-VAR:

$\mathtt{S} = \mathtt{C<\overline{S}>} \quad \mathtt{T} = \mathtt{C<\overline{T}>} \quad \overline{\mathtt{v}} \leq \overline{\mathtt{w}}$

We analyze the four possible values of $\mathtt{w}_i$ separately. Without loss of generality, we assume the rest of $\overline{\mathtt{w}}$ remain invariant, e.g. $\mathtt{w}_j = \mathtt{o}, j \neq i$.

$Subcase$ $\mathtt{w}_i = \mathtt{o}$:

$\mathtt{v}_i = \mathtt{o}$, and $\mathtt{S}_i = \mathtt{T}_i$. Thus, $\mathtt{S} = \mathtt{T}$.

$Subcase$ $\mathtt{w}_i = +$:

$\mathtt{v}_i = \mathtt{o}$ *or* $+$, $\Delta \vdash \mathtt{S}_i <: \mathtt{T}_i$.

If $\mathtt{v}_i = +$, let $\Delta_1 = \mathtt{T}'_i <: \mathtt{T}_i$, $\Delta_2 = \mathtt{S}'_i <: \mathtt{S}_i$,

$bound(\mathtt{C<\overline{T}_1,+T_i,\overline{T}_2>}) \Uparrow^{\Delta_1} \mathtt{C<\overline{T}_1,T'_i,\overline{T}_2>}$

$field(\mathtt{C<\overline{T}_1,T'_i,\overline{T}_2>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}} \quad \mathtt{U}_i \Downarrow_{\Delta_1} \mathtt{U}_0$

$bound(\mathtt{C<\overline{S}_1,+S_i,\overline{S}_2>}) \Uparrow^{\Delta_2} \mathtt{C<\overline{S}_1,S'_i,\overline{S}_2>}$

$field(\mathtt{C<\overline{S}_1,S'_i,\overline{S}_2>}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \quad \mathtt{V}_i \Downarrow_{\Delta_2} \mathtt{V}_0$

By Lemma 3.4.10(2), $\Delta \vdash \mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{V}_0$, and $\Delta \vdash \mathtt{V}_0 <: \mathtt{U}_0$.

If $\mathtt{v}_i = \mathtt{o}$, let $\Delta_1 = \mathtt{T}'_i <: \mathtt{T}_i$, $\Delta_2 = \emptyset$. Similarly to the above case, by Lemma 3.4.10(1), $\Delta \vdash [\mathtt{T}_i/\mathtt{T}'_i]\mathtt{U}_i \Downarrow_\emptyset \mathtt{V}_0$, $\Delta \vdash \mathtt{V}_0 <: \mathtt{U}_0$.

$Subcase$ $\mathtt{w}_i = -$:

The proof is similar to the above cases.

□

Lemma 3.3. *If* Δ *has non-variable bounds and* $\Delta \vdash \mathtt{T}$ *ok and* $\Delta \vdash bound_\Delta(\mathtt{T}) \Uparrow^{\Delta_1} \mathtt{C<\overline{T}>}$ *and* $mtype(\mathtt{m}, \mathtt{C<\overline{T}>}) = \mathtt{<\overline{Y} \lhd \overline{P}>\overline{U}} \rightarrow \mathtt{U}_0$ *and* $\Delta \vdash \overline{\mathtt{V}}, \overline{\mathtt{W}}$ *ok and* $\Delta, \Delta_1 \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$ *and* $\Delta, \Delta_1 \vdash \overline{\mathtt{W}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ *and* $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_1} \mathtt{V}_0$, *then for any* S *such that* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \mathtt{S}$ *ok, we have* $\Delta \vdash bound_\Delta(\mathtt{S}) \Uparrow^{\Delta_2} \mathtt{D<\overline{S}>}$ *and* $mtype(\mathtt{m}, \mathtt{D<\overline{S}>}) = \mathtt{<\overline{Y} \lhd \overline{P}'>\overline{U}'} \rightarrow \mathtt{U}'_0$ *and* $\Delta, \Delta_2 \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{P}'$ *and* $\Delta, \Delta_2 \vdash \overline{\mathtt{W}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$ *and* $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_2} \mathtt{V}'_0$ *and* $\Delta \vdash \mathtt{V}'_0 <: \mathtt{V}_0$.

Proof. Prove by induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

$Case$ S-REFL: trivial.

$Case$ S-TRANS:

Let U be such that $\Delta \vdash \mathtt{S} <: \mathtt{U}$, $\Delta \vdash \mathtt{U} <: \mathtt{T}$.

By induction:

$\Delta \vdash bound_\Delta(\mathtt{U}) \Uparrow^{\Delta_u} \mathtt{D}_u\mathtt{<\overline{S}}_u\mathtt{>}$

$mtype(\mathtt{m}, \mathtt{D}_u\mathtt{<\overline{S}}_u\mathtt{>}) = \mathtt{<\overline{Y} \lhd \overline{P}'_u>\overline{U}'_u} \rightarrow \mathtt{U}'_{u_0}$

$\Delta, \Delta_u \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'_u \quad \Delta, \Delta_u \vdash \overline{\mathtt{W}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'_u$

$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_u} \mathtt{V}'_{u_0} \quad \Delta \vdash \mathtt{V}'_{u_0} <: \mathtt{V}_0$

Also by induction:

$\Delta \vdash bound_\Delta(\mathtt{S}) \Uparrow^{\Delta_s} \mathtt{D}_s\texttt{<}\overline{\mathtt{S}}_s\texttt{>}$
$mtype(\mathtt{m},\ \mathtt{D}_s\texttt{<}\overline{\mathtt{S}}_s\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'_s\texttt{>}\overline{\mathtt{U}}'_s \rightarrow \mathtt{U}'_{s_0}$
$\Delta, \Delta_s \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'_s \qquad \Delta, \Delta_s \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'_s$
$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_{u_0} \Downarrow_{\Delta_s} \mathtt{V}'_{s_0} \qquad \Delta \vdash \mathtt{V}'_{s_0} \texttt{<:} \mathtt{V}_{u_0}$

$Case$ S-UBOUND: easy, since $bound_\Delta(\mathtt{S}) = bound_\Delta(\mathtt{T})$.

$Case$ S-LBOUND: antecedent of the lemma is impossible.

$Case$ S-CLASS: $\mathtt{S} = \mathtt{D}\texttt{<}\overline{\mathtt{W}}\texttt{>}$

$CT(\mathtt{D}) = \mathtt{class}\ \mathtt{D}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}\ \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\texttt{>?}\triangleleft \mathtt{C}\texttt{<}\overline{\mathtt{S}}'\texttt{>}\ \{\ldots\}$
$\Delta \vdash \mathtt{D}\texttt{<}\overline{\mathtt{W}}\texttt{>} \Uparrow^{\Delta_2} \mathtt{D}\texttt{<}\overline{\mathtt{W}}'\texttt{>} \qquad \Delta \vdash \overline{\mathtt{W}}' \texttt{<:} [\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{R}}$
$([\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{S}}'\texttt{>}) \Downarrow_{\Delta_2} \mathtt{T}$

There are three rules for the value of $mtype(\mathtt{m},\ \mathtt{D}\texttt{<}\overline{\mathtt{W}}'\texttt{>})$. We analyze them case by case:

$Subcase$ MT-CLASS:

$mtype(\mathtt{m},\ \mathtt{D}\texttt{<}\overline{\mathtt{W}}'\texttt{>}) = [\overline{\mathtt{W}}'/\overline{\mathtt{X}}](\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}''\texttt{>}\overline{\mathtt{U}}'' \rightarrow \mathtt{U}''_0)$
$\overline{\mathtt{P}}' = [\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{P}}'' \quad \overline{\mathtt{U}}' = [\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{U}}'' \quad \mathtt{U}'_0 = [\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{U}''_0$
$mtype(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}''\texttt{>}\overline{\mathtt{U}}'' \rightarrow \mathtt{U}''_0)$
$\overline{\mathtt{P}} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}'' \quad \overline{\mathtt{U}} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}'' \quad \mathtt{U}_0 = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}''_0$

By Lemma 3.4.9(2),
$\Delta, \Delta_2 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{P}}'' \Rightarrow \Delta, \Delta_2 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'$
$\Delta, \Delta_2 \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{U}}'' \Rightarrow \Delta, \Delta_2 \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

By Lemma 3.4.9(1),
$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{U}''_0 \Downarrow_{\Delta_2} \mathtt{V}'_0 \Rightarrow [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}'_0 \Downarrow_{\Delta_2} \mathtt{V}'_0$

$Subcase$ MT-SUPER:

The proof is obvious for the cases where $mtype(\mathtt{m},\ \mathtt{D}\texttt{<}\overline{\mathtt{W}}'\texttt{>}) = mtype(\mathtt{m},\ [\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{S}}'\texttt{>})$. Use Lemma 3.4.9.

$Case$ S-VAR:

We analyze the four possible values of $\mathtt{w}_i$ separately. Without loss of generality, we assume the rest of $\overline{\mathtt{w}}$ remain invariant, e.g. $\mathtt{w}_j = \mathtt{o}, j \neq i$.

$Subcase$ $\mathtt{w}_i = \mathtt{o}$: $\mathtt{S}_i = \mathtt{T}_i$. Easy.

$Subcase$ $\mathtt{w}_i = \mathtt{+}$: $\mathtt{S}_i \texttt{<:} \mathtt{T}_i$

$\mathtt{C}\texttt{<}\overline{\mathtt{T}}_1\texttt{,+}\mathtt{T}_i\texttt{,}\overline{\mathtt{T}}_2\texttt{>} \Uparrow^{\mathtt{X}' \texttt{<:} \mathtt{T}_i} \mathtt{C}\texttt{<}\overline{\mathtt{T}}_1\texttt{,}\mathtt{X}'\texttt{,}\overline{\mathtt{T}}_2\texttt{>} \quad \Delta_1 = \mathtt{X}' \texttt{<:} \mathtt{T}_i$
$mtype(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}_1\texttt{,}\mathtt{X}'\texttt{,}\overline{\mathtt{T}}_2\texttt{>}) = [\mathtt{X}'/\mathtt{T}_i](\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}'' \rightarrow \mathtt{U}''_0)$
$\overline{\mathtt{P}} = [\mathtt{X}'/\mathtt{X}_i]\overline{\mathtt{P}}'' \quad \overline{\mathtt{U}} = [\mathtt{X}'/\mathtt{X}_i]\overline{\mathtt{U}}'' \quad \mathtt{U}_0 = [\mathtt{X}'/\mathtt{X}_i]\mathtt{U}''_0$

$\mathtt{v}_i$ could be either + or o.

If $\mathtt{v}_i = \mathtt{+}$,
$\mathtt{C}\texttt{<}\overline{\mathtt{S}}_1\texttt{,+}\mathtt{S}_i\texttt{,}\overline{\mathtt{S}}_2\texttt{>} \Uparrow^{\mathtt{X}' \texttt{<:} \mathtt{T}_i} \mathtt{C}\texttt{<}\overline{\mathtt{S}}_1\texttt{,}\mathtt{X}'\texttt{,}\overline{\mathtt{S}}_2\texttt{>} \quad \Delta_2 = \mathtt{X}' \texttt{<:} \mathtt{S}_i$
$mtype(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{S}}_1\texttt{,}\mathtt{X}'\texttt{,}\overline{\mathtt{S}}_2\texttt{>}) = [\mathtt{X}'/\mathtt{S}_i](\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}'' \rightarrow \mathtt{U}''_0)$
$\overline{\mathtt{P}}' = [\mathtt{X}'/\mathtt{X}_i]\overline{\mathtt{P}}'' \quad \overline{\mathtt{U}}' = [\mathtt{X}'/\mathtt{X}_i]\overline{\mathtt{U}}'' \quad \mathtt{U}'_0 = [\mathtt{X}'/\mathtt{X}_i]\mathtt{U}''_0$

By Lemma 3.4.2(1),
$\Delta, \Delta_2 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}' \quad \Delta, \Delta_2 \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

By Lemma 3.4.10(2),
$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_2} \mathtt{V}'_0,\ \Delta \vdash \mathtt{V}'_0 \texttt{<:} \mathtt{V}_0$

If $\mathtt{v}_i = \mathtt{o}$,
$\mathtt{C}\texttt{<}\overline{\mathtt{S}}_1\texttt{,}\mathtt{S}_i\texttt{,}\overline{\mathtt{S}}_2\texttt{>} \Uparrow^{\emptyset} \mathtt{C}\texttt{<}\overline{\mathtt{S}}_1\texttt{,}\mathtt{S}_i\texttt{,}\overline{\mathtt{S}}_2\texttt{>} \qquad \Delta_2 = \emptyset$
$mtype(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{S}}_1\texttt{,}\mathtt{S}_i\texttt{,}\overline{\mathtt{S}}_2\texttt{>}) = [\mathtt{S}_i/\mathtt{X}_i](\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}}'' \rightarrow \mathtt{U}''_0)$
$\overline{\mathtt{P}}' = [\mathtt{S}_i/\mathtt{X}_i]\overline{\mathtt{P}}'' \quad \overline{\mathtt{U}}' = [\mathtt{S}_i/\mathtt{X}_i]\overline{\mathtt{U}}'' \quad \mathtt{U}'_0 = [\mathtt{S}_i/\mathtt{X}_i]\mathtt{U}''_0$

By Lemma 3.4.3,
$\Delta, \Delta_2 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}' \quad \Delta, \Delta_2 \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

By Lemma 3.4.10(1), $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_2} \mathtt{V}'_0,\ \Delta \vdash \mathtt{V}'_0 \texttt{<:} \mathtt{V}_0$

$Subcase$ $\mathtt{w}_i = \mathtt{-}$ or $\mathtt{*}$: Similar to above.

□

Lemma 3.4 (TERM SUBSTITUTION PRESERVES TYPING)**.** *For any Δ that has non-variable bounds, if $\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} \in \mathtt{T}_0$ and $\Delta; \Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}}$ where $\Delta \vdash \overline{\mathtt{S}} \texttt{<:} \overline{\mathtt{T}}$, then $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{S}_0$ for some $\mathtt{S}_0$ such that $\Delta \vdash \mathtt{S}_0 \texttt{<:} \mathtt{T}_0$.*

Proof. Prove by induction on the derivaion of $\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} \in \mathtt{T}_0$.

$Case$ T-VAR:

$\Delta; \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{x} \in \mathtt{T}_i$
$\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x} \in \mathtt{S}_i$
$\Delta; \Gamma \vdash \mathtt{S}_i \texttt{<:} \mathtt{T}_i$

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

$\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash bound_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>}$
$fields(bound_\Delta(\mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>})) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \quad \mathtt{S}_i \Downarrow_{\Delta'} \mathtt{T}$

By induction, there exists an $\mathtt{S}_0$ such that $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{S}_0$, $\Delta \vdash \mathtt{S}_0 \texttt{<:} \mathtt{T}_0$.

By Lemma 3.2,
$\Delta \vdash bound_\Delta(\mathtt{S}_0) \Uparrow^{\Delta'} \mathtt{D}\texttt{<}\overline{\mathtt{V}}\texttt{>} \quad fields(\mathtt{D}\texttt{<}\overline{\mathtt{V}}\texttt{>}) = \overline{\mathtt{W}}\ \overline{\mathtt{f}}\ldots$
$\mathtt{W}_i \Downarrow_{\Delta''} \mathtt{W}'_i \qquad \Delta \vdash \mathtt{W}'_i \texttt{<:} \mathtt{T}$

By T-FIELD, $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\mathtt{.f}_i \in \mathtt{W}'_i$.

$Case$ T-INVK: $\mathtt{e} = \mathtt{e}_0\texttt{.<}\overline{\mathtt{V}}\texttt{>m(}\overline{\mathtt{e}}\texttt{)}$

$\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash bound_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$
$mtype(\mathtt{m},\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0$
$\overline{\mathtt{Y}} \cap dom(\Delta') = \emptyset \qquad \Delta \vdash \overline{\mathtt{V}}\ ok$
$\Delta, \Delta' \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}$
$\Delta, \Delta' \vdash \overline{\mathtt{S}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}$

By the induction hypothesis, there exists a $\mathtt{T}'_0$ such that
$\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 \in \mathtt{T}'_0 \quad \Delta \vdash \mathtt{T}'_0 \texttt{<:} \mathtt{T}_0$

There exists $\overline{\mathtt{S}}'$ such that
$\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} \in \overline{\mathtt{S}}' \quad \Delta \vdash \overline{\mathtt{S}}' \texttt{<:} \overline{\mathtt{S}}$

By Lemma 3.3,
$\Delta \vdash bound_\Delta(\mathtt{T}'_0) \Uparrow^{\Delta_2} \mathtt{D}\texttt{<}\overline{\mathtt{T}}'\texttt{>}$
$mtype(\mathtt{m},\ \mathtt{D}\texttt{<}\overline{\mathtt{T}}'\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'\texttt{>}\overline{\mathtt{U}}' \rightarrow \mathtt{U}'_0$
$\Delta, \Delta_2 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}' \qquad \Delta, \Delta_2 \vdash \overline{\mathtt{S}}' \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$
$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{S}'_0 \Downarrow_{\Delta_2} \mathtt{T}' \qquad \Delta \vdash \mathtt{T}' \texttt{<:} \mathtt{T}$

By T-METHOD, $\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{T}'$.

$Case$ T-NEW: $\mathtt{e} = \mathtt{new}\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>(}\overline{\mathtt{e}}\texttt{)}, \mathtt{T} = \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$.

$\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ ok \quad fields(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$
$\Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \quad \Delta \vdash \overline{\mathtt{S}} \texttt{<:} \overline{\mathtt{U}}$

By induction hypothesis, there exists $\overline{\mathtt{S}}'$ such that
$\Delta; \Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}} \in \overline{\mathtt{S}}' \quad \Delta \vdash \overline{\mathtt{S}}' \texttt{<:} \overline{\mathtt{S}}$

By S-TRANS and T-NEW,
$\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>(}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\texttt{)} \in \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$.

$Case$ T-CAST and T-SCAST:
easy using induction hypothesis. □

Lemma 3.4.1 (WEAKENING)**.** *Suppose $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}$ ok and $\Delta \vdash \mathtt{U}$ ok.*
1) If $\Delta \vdash \mathtt{S} \texttt{<:} \mathtt{T}$, then $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \mathtt{S} \texttt{<:} \mathtt{T}$.
2) If $\Delta \vdash \mathtt{S}$ ok, then $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \mathtt{S}$ ok.
3) If $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, then $\Delta; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e} \in \mathtt{T}$ and $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}}; \Gamma \vdash \mathtt{e} \in \mathtt{T}$.

Proof. 1)Prove by induction on the derivation of $\Delta \vdash \mathtt{S} \texttt{<:} \mathtt{T}$
$Case$ S-REFL, S-TRANS: easy.
$Case$ S-UBOUND: $\Delta(\mathtt{S}) = (\mathtt{+},\ \mathtt{T})$.
If $\mathtt{S} \in \overline{\mathtt{X}}$, let $\mathtt{S} = \mathtt{X}_i$.
If $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \mathtt{N}_i \texttt{<:} \mathtt{T}$, then by S-TRANS, $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \mathtt{S} \texttt{<:} \mathtt{T}$.
If $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \mathtt{N}_i \not\texttt{<:} \mathtt{T}$, we still have $\Delta, \overline{\mathtt{X}} \texttt{<:} \overline{\mathtt{N}} \vdash \mathtt{S} \texttt{<:} \mathtt{T}$.

If $S \not\in \overline{X}$, this is trially true.

Case S-LBOUND: $\Delta(T) = (-, S)$.

If $T \not\in \overline{X}$, this case is trivially true.

If $T \in \overline{X}$, let $T = X_i$.

If $\Delta \vdash N_i <: S$, then it conflicts with $\Delta \vdash S <: T$, and thus Δ is inconsistent, and we can prove anything from it.

The only valid case is when $\Delta \vdash S <: N_i$. And from this, we can easily see that $\Delta, \overline{X} <: \overline{N} \vdash S <: T$.

Case S-CLASS: $\underline{S}$=C<$\overline{T}$>
class C<$\overline{X} \lhd \overline{N}$> <$\overline{X} \lhd \overline{R}$>?$\lhd$D<$\overline{S}$> { ... }
$\Delta \vdash$C<$\overline{T}$>$\Uparrow^{\Delta'}$C<$\overline{U}$> $\quad \Delta \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{R} \quad ([\overline{U}/\overline{X}]$D<$\overline{S}$>$) \Downarrow_{\Delta'}$T

By induction, all conditions still hold when Δ becomes $\Delta, \overline{X} <: \overline{N}$. Thus, we have $\Delta, \overline{X} <: \overline{N} \vdash$C<$\overline{T}$><:T.

Case S-VAR: S=C<$\overline{S}$>, T=C<$\overline{T}$>.

By induction, all conditions for S-VAR still hold when Δ becomes $\Delta, \overline{X} <: \overline{N}$. Thus, we have $\Delta, \overline{X} <: \overline{N} \vdash$C<$\overline{S}$><:C<$\overline{T}$>.

2) Prove by induction on the derivation of well-formed type rules.
Case WF-OBJECT: Trivial.
Case WF-VAR:
$S \in dom(\Delta)$ implies that $S \in dom(\Delta, \overline{X} <: \overline{N})$. Done.
Case WF-CLASS: S=C<$\overline{T}$>
CT(C) =class C<$\overline{Y} \lhd \overline{Q}$> <$\overline{Y} \lhd \overline{R}$>?$\lhd$D<$\overline{S}$> {...}
$\Delta \vdash \overline{T}\ ok \quad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{Q}$

By induction hypothesis and Lemma 3.4.1(1),
$\Delta, \overline{X} <: \overline{N} \vdash \overline{T}\ ok \quad \Delta, \overline{X} <: \overline{N} \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{Q}$

Thus, $\Delta, \overline{X} <: \overline{N} \vdash$C<$\overline{T}$> ok.

3) Prove by induction on the derivation of $\Delta; \Gamma \vdash e \in T$ rules.
Case T-VAR: Trivial.
Case T-FIELD: $e = e_0.f_i$, $\Delta;\Gamma \vdash e_0.f_i \in T$.
$\Delta;\Gamma \vdash e_0 \in T_0 \qquad \Delta \vdash bound_\Delta(T_0) \Uparrow^{\Delta'}$C<$\overline{U}$>
$field(bound$(C< $\overline{U}$>$)) = \overline{Sf} \quad S_i \Downarrow_{\Delta'}T$

By induction hypothesis and Lemma 3.4.11, the above conditions hold when Δ is changed to $\Delta, \overline{X} <: \overline{N}$, and Γ to $\Gamma, x : U$, which means $\Delta, \overline{X} <: \overline{N}; \Gamma \vdash e_0.f_i \in T$, and $\Delta; \Gamma, x{:}U \vdash e_0.f_i \in T$

Case T-INVK: e=e$_0$.<$\overline{V}$>m($\overline{e}$)$\in$T.

Proof is similar to the above, using induction hypothesis, Lemma 3.4.1(1), 3.4.1(2), and 3.4.11.

Case T-NEW, T-CAST, T-SCAST:
Easy using induction hypotheses and Lemma 3.4.1(1). □

Lemma 3.4.2 (NARROWING). *1) If* Δ,X<:S$\vdash T_1 <: T_2$ *and* $\Delta \vdash$U<:S*, then* Δ,X<:U$\vdash T_1 <: T_2$.
2) If Δ,X:>S$\vdash T_1 <: T_2$ *and* $\Delta \vdash$S<:U*, then* Δ,X:>U$\vdash T_1 <: T_2$.
3) If Δ,X: (*,S) $\vdash T_1 <: T_2$*, then* Δ, (X:(v,U) $\vdash T_1 <: T_2$.

Proof. Straight forward using induction on the derivation of Δ,X<:S$\vdash T_1 <: T_2$, Δ,X:>S$\vdash T_1 <: T_2$, and Δ,X: (*,S) $\vdash T_1 <: T_2$, respectively. □

Lemma 3.4.3 (TYPE SUBSTITUTION PRESERVES SUBTYPING)**.** *If* $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash S <: T$ *and* $\Delta_1 \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{N}$*, with* $\Delta_1 \vdash \overline{U}$ *ok and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S <: [\overline{U}/\overline{X}]T$.

Proof. Prove by induction on the derivation of $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash S <: T$
Case S-REFL: Trivial.
Case S-TRANS:

Let U be a type such that: $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash S <: V$ $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash V <: T$

Then, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S <: [\overline{U}/\overline{X}]V$, and $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]V <: [\overline{U}/\overline{X}]T$

By S-TRANS, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S <: [\overline{U}/\overline{X}]T$.

Case S-UBOUND: S=X, $(\Delta_1, \overline{X} <: \overline{N}, \Delta_2)(X) = (+, T)$.

If $S \in dom(\Delta_1)$, then the result is obvious, since none of $\overline{X}$ appears in Δ_1.

If $S \in dom(\Delta_2)$, result is also obvious since substitution is applied to both Δ_2, S, and T.

If $S = X_i$, $\Delta_1 \vdash U_i <: [\overline{U}/\overline{X}]N_i$, $[\overline{U}/\overline{X}]S = U_i$. Then by Lemma 3.4.1, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S <: [\overline{U}/\overline{X}]T$.

Case S-LBOUND: $T \in dom(\Delta_1) \cup dom(\Delta_2)$. The result is obvious.
Case S-CLASS: $\underline{S}$=C<$\overline{S}$>
Let $\Delta = \Delta_1, \overline{X} <: \overline{N}, \Delta_2$,
CT(C) =class C<$\overline{X} \lhd \overline{N}$> <$\overline{X} \lhd \overline{R}$>?$\lhd$D<$\overline{S}$> {...}
$\Delta \vdash$C<$\overline{T}$>$\Uparrow^{\Delta'}$C<$\overline{V}$> $\quad \Delta \vdash \overline{V} <: [\overline{V}/\overline{X}]\overline{R} \quad ([\overline{V}/\overline{X}]$D<$\overline{S}$>$\Downarrow_{\Delta'}$T

By Lemma 3.4.4(1) and (2), and the induction hypothesis, the result is obvious.

Case S-VAR: easily follows from induction hypothesis. □

Lemma 3.4.4. *1) If* $\Delta_1 \vdash \overline{S} <: [\overline{S}/\overline{X}]\overline{N}$ *and* $\Delta_1 \vdash \overline{S}$ *ok,* $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash$C<$\overline{T}$>$\Uparrow^{\Delta'}$C<$\overline{U}$> *with none of* $\overline{X}$ *appearing in* Δ_1 *and none of the type variables is* $dom(\Delta')$ *appearing in* $\overline{S}$*, then* $\Delta_1, [\overline{S}/\overline{X}]\Delta_2 \vdash [\overline{S}/\overline{X}]$C<$\overline{T}$>$\Uparrow^{[\overline{S}/\overline{X}]\Delta'}$ $[\overline{S}/\overline{X}]$C<$\overline{U}$>.
2) If $S \Downarrow_\Delta T$ *where* $dom(\Delta)$ *and* $\overline{X}$ *are distinct, then* $[\overline{S}/\overline{X}]S \Downarrow_{[\overline{S}/\overline{X}]\Delta} [\overline{S}/\overline{X}]T$.
3) $[\overline{S}/\overline{X}]fields$(C<$\overline{T}$>$) = fields([\overline{S}/\overline{X}]$C<$\overline{T}$>)
4) $[\overline{S}/\overline{X}]mtype$(m, C<$\overline{T}$>$) = mtype$(m, $[\overline{S}/\overline{X}]$C<$\overline{T}$>)

Proof. Result easily follows from induction on the derivation of $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash$C<$\overline{T}$>$\Uparrow^{\Delta'}$C<$\overline{U}$>, $[\overline{S}/\overline{X}]S \Downarrow_{[\overline{S}/\overline{X}]\Delta} [\overline{S}/\overline{X}]T$, $fields([\overline{S}/\overline{X}]$C<$\overline{T}$>), and $[\overline{S}/\overline{X}]mtype$(m,C<$\overline{T}$>), respectively. □

Lemma 3.4.5 (TYPE SUBSTITUTION PRESERVES TYPE WELL-FORMEDNESS)**.** *If* $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash T$ *ok and* $\Delta_1 \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{N}$ *with* $\Delta_1 \vdash \overline{U}$ *ok and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]T$ *ok.*

Proof. We prove by induction on the derivation of $\Delta_1, \overline{X} <: \overline{N}, \Delta_2 \vdash T\ ok$.
Case WF-OBJECT: Trivial.
Case WF-VAR: T=X, $X \in dom(\Delta_1, \overline{X} <: \overline{N})$.

If $X \in dom(\Delta_1) \cup dom(\Delta_2)$, result is obvious.

If $X = X_i$, $[\overline{U}/\overline{X}]X = U_i$. By assumptiong, $\Delta_1 \vdash U_i\ ok$. By Lemma 3.4.1(2), $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash U_i$

Case WF-CLASS: T=C<$\overline{T}$>
CT(C) =class C<$\overline{Y} \lhd \overline{N}$> <$\overline{Y} \lhd \overline{R}$>?D<$\overline{S}$> {...}
$\Delta_1, \overline{Y} <: \overline{N}, \Delta_2 \vdash \overline{T}\ ok \quad \Delta_1, \overline{Y} <: \overline{N}, \Delta_2 \vdash \overline{T} <: [\overline{T}/\overline{Y}]\overline{N}$

By induction hypothesis, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{T}\ ok$.

By Lemma 3.4.3, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{T} <: [\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{N}$. Since $\overline{X}$ are not free in $\overline{N}$, $[\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{N} = [[\overline{U}/\overline{X}]\overline{T}/\overline{Y}]\overline{N}$. The result follows from WF-CLASS. □

Lemma 3.4.6 (TYPE SUBSTITUTION PRESERVES TYPING)**.** *If both* Δ_1 *and* Δ_2 *have non-variable bounds and* $\Delta_1, \overline{X} <: \overline{N}, \Delta_2; \Gamma \vdash e \in T$ *and* $\Delta_1 \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{N}$ *where* $\Delta_1 \vdash \overline{U}$ *ok and none of* $\overline{X}$ *appears in* Δ_1*, then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e \in S$ *for some* S *such that* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S <: [\overline{U}/\overline{X}]T$.

Proof. Prove by induction on the derivation of $\Delta_1, \overline{X} <: \overline{N}, \Delta_2; \Gamma \vdash e \in T$. Let $\Delta = \Delta_1, \overline{X} <: \overline{N}, \Delta_2$.

Case T-VAR: Trivial.
Case T-FIELD: $e{=}e_0.f_i$

$\Delta;\Gamma \vdash e_0 \in T_0$ $\quad \Delta \vdash bound_\Delta(T_0) \Uparrow^{\Delta'} C\text{<}\overline{T}\text{>}$
$fields(bound_\Delta(C\text{<}\overline{T}\text{>}) = \overline{S}\ \overline{f}$ $\quad S_i \Downarrow_{\Delta'} T$

By induction hypothesis, there is a S_0 such that
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash e_0 \in S_0$ $\quad \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0$

By Lemma 3.4.7,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(T_0)$
$<: [\overline{U}/\overline{X}](bound_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T_0)).$

By S-TRANS,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)$
$<: [\overline{U}/\overline{X}](bound_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T_0)).$

By Lemma 3.2,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_\Delta(S_0) \Uparrow^{\Delta''} D\text{<}\overline{S}'\text{>}$
$fields(D\text{<}\overline{S}'\text{>}) = \overline{S}''\overline{f}$ $\quad S_i'' \Downarrow_{\Delta''} T'$
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash T' <: [\overline{U}/\overline{X}]T$

By T-FIELD, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0.f_i \in T'$.

Case T-INVK: $e{=}e_0.\text{<}\overline{V}\text{>}m(\overline{e}){\in}T$.

$\Delta;\Gamma \vdash e_0 \in T_0$ $\quad \Delta \vdash bound_\Delta(T_0) \Uparrow^{\Delta'} C\text{<}\overline{T}\text{>}$
$mtype(m,\ C\text{<}\overline{T}\text{>}) = \text{<}\overline{Y} \lhd \overline{P}\text{>}\overline{W} \rightarrow W_0$
$\overline{Y} \cap dom(\Delta') = \emptyset$ $\quad \Delta \vdash \overline{V}\ ok$
$\Delta, \Delta' \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P}$ $\quad \Delta;\Gamma \vdash \overline{e} \in \overline{S}$
$\Delta, \Delta' \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}$ $\quad [\overline{V}/\overline{Y}]W_0 \Downarrow_{\Delta'} T$

By induction hypothesis, there is a S_0 such that
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash e_0 \in S_0$ $\quad \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0$

Also by the induction hypothesis, there exists $\overline{S}'$ such that
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash \overline{e} \in \overline{S}'$ $\quad \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{S}' <: [\overline{U}/\overline{X}]\overline{S}$

By Lemma 3.4.7 and S-TRANS,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)$
$<: [\overline{U}/\overline{X}](bound_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T_0)).$

By Lemma 3.4.5,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{V}\ ok$

By Lemma 3.4.3,
$\Delta_1, [\overline{U}/\overline{X}](\Delta_2, \Delta') \vdash [\overline{U}/\overline{X}]\overline{V} <: [\overline{U}/\overline{X}][\overline{V}/\overline{Y}]\overline{P}$
$\Delta_1, [\overline{U}/\overline{X}](\Delta_2, \Delta') \vdash [\overline{U}/\overline{X}]\overline{S} <: [\overline{U}/\overline{X}][\overline{V}/\overline{Y}]\overline{W}$

By Lemma 3.4.4(2) and (4),
$mtype(m,\ [\overline{U}/\overline{X}]C\text{<}\overline{T}\text{>}) = \text{<}\overline{Y} \lhd [\overline{U}/\overline{X}]\overline{P}\text{>}[\overline{U}/\overline{X}]\overline{W} \rightarrow [\overline{U}/\overline{X}]W_0$
$[\overline{U}/\overline{X}][\overline{V}/\overline{Y}]W_0 \Downarrow_{[\overline{U}/\overline{X}]\Delta'} [\overline{U}/\overline{X}]T$

Because $\overline{X}$ does not appear in Δ_1,
$[\overline{U}/\overline{X}][\overline{V}/\overline{Y}]T = [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]([\overline{U}/\overline{X}]T)$

By Lemma 3.3,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0) \Uparrow^{\Delta''} D\text{<}\overline{T}'\text{>}$
$mtype(m,\ D\text{<}\overline{T}'\text{>}) = \text{<}\overline{Y} \lhd \overline{P}\text{>}\overline{W}' \rightarrow W_0'$
$\Delta_1, ([\overline{U}/\overline{X}]\Delta_2), \Delta'' \vdash [\overline{U}/\overline{X}]\overline{V} <: [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]\overline{W}'$
$\Delta_1, ([\overline{U}/\overline{X}]\Delta_2), \Delta'' \vdash [\overline{U}/\overline{X}]\overline{S} <: [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]\overline{W}'$
$[[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]W_0' \Downarrow_{\Delta''} T'$ $\quad \Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash T' <: [\overline{U}/\overline{X}]T$

By Lemma 3.4.1 and S-TRANS,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2, \Delta'' \vdash \overline{S}' <: [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]\overline{W}'$

By T-METHOD,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]e_0.\text{<}\overline{V}\text{>}m(\overline{e}){\in}T'$

Case T-NEW: $e{=}\mathtt{new}\ C\text{<}\overline{T}\text{>}(\overline{e}), T{=}C\text{<}\overline{T}\text{>}$.

$\Delta \vdash C\text{<}\overline{T}\text{>}\ \ ok$ $\quad fields(C\text{<}\overline{T}\text{>}) = \overline{U}'\ \overline{f}$
$\Delta;\Gamma \vdash \overline{e} \in \overline{S}$ $\quad \Delta \vdash \overline{S} <: \overline{U}'$

By Lemma 3.4.5,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]C\text{<}\overline{T}\text{>}$

By Lemma 3.4.4(3),
$fields([\overline{U}/\overline{X}]C\text{<}\overline{T}\text{>}) = [\overline{U}/\overline{X}]\overline{U}'\ \overline{f}$

By induction hypothesis, there is $\overline{S}'$,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]\overline{e} \in \overline{S}'$
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{S}' <: \overline{S}$

By Lemma 3.4.3 and S-TRANS,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{S}' <: [\overline{U}/\overline{X}]\overline{U}'$

By T-NEW, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \quad \vdash C\text{<}[\overline{U}/\overline{X}]\overline{T}\text{>}([\overline{U}/\overline{X}]\overline{e}) \in [\overline{U}/\overline{X}]T$.

Case T-CAST, T-SCAST: Easy. ☐

Lemma 3.4.7. *Suppose Δ has non-variable bounds and is of the form $\Delta_1, \overline{X} <: \overline{N}, \Delta_2$. If $\Delta \vdash T$ ok and $\Delta_1 \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{N}$ with $\Delta_1 \vdash \overline{U}$ ok and none of the $\overline{X}$ appear in Δ_1, then, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T) <: [\overline{U}/\overline{X}](bound_\Delta(T))$.*

Proof. We prove for each case in the definition of $bound_\Delta$.
If T is a non-variable type, $bound_\Delta(T) = T$. The proof is trivial.
If T is a variable, $T{=}X$, $\Delta(X) = (+,S)$, then
$bound_\Delta(T) = bound_\Delta(S)$ $\quad \Delta \vdash T <: S$
The proof is trivial if $X{\in}\ dom(\Delta_1) \cup dom(\Delta_2)$. If $X{=}X_i$,
$bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T) = bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(U_i)$
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(N_i)$
$<: [\overline{U}/\overline{X}](bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S))$
By assumption and Lemma 3.4.1,
$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash U_i <: N_i$
And S-TRANS finishes the case. ☐

Lemma 3.4.8 (CLOSE YIELDS A SUPERTYPE WITHOUT LOCAL TYPE VARIABLES)**.** *If $\Delta, \Delta' \vdash S$ ok and $S{\Downarrow_{\Delta'}}T$, then $\Delta, \Delta' \vdash S <: T$ and $\Delta \vdash T$ ok.*

Proof. We prove by induction on the $\Downarrow_{\Delta'}$ rules.
Case C-PROM: $S{=}X$, $\Delta'(X) = (+,\ T)$
By S-VAR and Lemma 3.4.1, $\Delta, \Delta' \vdash X{<:}T$.
Case C-TVAR: Trivial.
Case C-CLASS: $S{=}C\text{<}\overline{T}\text{>}$, $T{=}C\text{<}\overline{T}'\text{>}$.
There are three subcases, and we analyze without loss of generality only one T_i.
Subcase 1: $T_i \Downarrow_{\Delta'} T_i$, $(w_i, T_i') = (v_i, T_i)$. Trivially true.
Subcase 2: $T_i \Downarrow_{\Delta'} U_i$, $T_i \neq U_i$, $(w_i, T_i') = (v_i{\vee}{+}, U_i)$.
By induction hypothesis, $\Delta, \Delta' \vdash T_i <: U_i$. w_i is either + or *. If $w_i = *$, then this case is trivially true. If $w_i = +$, by S-VAR, and $\Delta, \Delta' \vdash T_i <: U_i$, we still have $\Delta, \Delta' \vdash S <: T$.
Subcase 3: $T_i = X$, $\Delta(X) = (v_i', U_i)$,
Since $v_i{=}o$ because there cannot be any other variance annotation on X, $w_i = v_i'$.
If $v_i' = o$, $T_i = U_i$, and $w_i = o$. This case is then trivial.
If $v_i' = +, -, or *$, these cases easily falls out from the S-VAR rule. ☐

Lemma 3.4.9. *Suppose $\Delta, \Delta_1 \vdash C\text{<}\overline{S}\text{>}$ ok and $C\text{<}\overline{S}\text{>}{\Downarrow_{\Delta_1}}T$ and $\Delta \vdash T{\Uparrow^{\Delta_2}}C\text{<}\overline{U}\text{>}$ and $\Delta_3 \vdash S_0$ ok where $dom(\Delta_i)$ $(i = 1, 2, 3)$ are distinct from each other.*
1) If $[\overline{U}/\overline{X}]S_0 \ \Downarrow_{\Delta_2} S_0'$ and $\overline{X}$ do not appear in Δ_1 or Δ_2, then $[\overline{S}/\overline{X}]S_0 \Downarrow_{\Delta_1} S_0'$.
2) If $\Delta, \Delta_2 \vdash U_0{<:}[\overline{U}/\overline{X}]S_0$ and $\Delta \vdash U_0$ ok with $\overline{X}$ not appearing in Δ or Δ_1 or Δ_2, then $\Delta, \Delta_1 \vdash U_0 <: [\overline{S}/\overline{X}]S_0$.

Proof. By inspecting the given conditions, one of these three properties must hold:
1) $S_i \Downarrow_{\Delta_1} S_i$, $U_i = S_i$.
2) $S_i \Downarrow_{\Delta_1} S_i'$, $S_i \neq S_i'$. By Lemma 3.4.8, $\Delta, \Delta_1 \vdash S_i <: S_i'$. U_i is a type variable Y such that $\Delta_2(Y) = (+, S_i')$.
3) S_i is a type variable Z and $\Delta_1(Z) = (+,V)$, and U_i is a type variable Y and $\Delta_1(Y) = (+,V)$.

We prove by the first part of the lemma by induction on S_0.
Case $S_0 = X$, where X is a type variable.
There are two cases:
If $X{\in}\ \overline{X}$: $[\overline{U}/\overline{X}]S_0 = U_i$, $[\overline{S}/\overline{X}]S_0 = S_i$.

From the analysis of the three properties above,

1) $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}_i$, $\mathtt{U}_i = \mathtt{S}_i$, $\mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{U}_i$.
2) $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}'_i$, $\mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{S}'_i$.
3) $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{V}$, $\mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{V}$.

If $\mathtt{X} \not\in \overline{\mathtt{X}}$: $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0 = \mathtt{X}$. Proof is simple through induction hypothesis.

Case $\mathtt{S}{=}\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}$, $\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}\Downarrow_{\Delta_2}\mathtt{C}\texttt{<}\overline{\mathtt{w}}\overline{\mathtt{T}}'\texttt{>}$. The rest is easy through induction hypothesis.

The second part is proven by inspecting the three properties above. There exists $\Delta', \Delta'_1, \overline{\mathtt{Y}}, \overline{\mathtt{T}}, \overline{\mathtt{S}}''$, such that $\Delta_1 = \Delta', \Delta'_1$, $\Delta_2 = \Delta', \overline{\mathtt{Y}} <: \overline{\mathtt{T}}$, and $\overline{\mathtt{S}} = [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$, and $\Delta, \Delta_1 \vdash \overline{\mathtt{S}}'' <: \overline{\mathtt{T}}$. By Lemma 3.4.3, $\Delta, \Delta', \Delta'_1 \vdash [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\mathtt{U}_0 <: [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0$. $\overline{\mathtt{Y}}$ is fresh in $\mathtt{U}_0$ and $\mathtt{S}_0$, then $[\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\mathtt{U}_0 = \mathtt{U}_0$, and $[\overline{\mathtt{S}}''/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0$. □

Lemma 3.4.10. *1. If* $\mathtt{S}\Downarrow_{\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}}\mathtt{S}''$*, and* $\Delta \vdash \mathtt{T}$ *ok where* $dom(\Delta', \mathtt{X}:(\mathtt{v},\mathtt{T}))$ *are fresh w.r.t.* Δ*, then* $[\mathtt{T}/\mathtt{X}]\mathtt{S}\Downarrow_{\Delta'}\mathtt{S}'$ *and* $\Delta \vdash \mathtt{S}' <: \mathtt{S}''$.

2. If $\Delta \vdash \mathtt{T} <: \mathtt{U}$ *and* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}<:\mathtt{U}}\mathtt{S}''$ *where* $dom(\Delta', \mathtt{X}<:\mathtt{U})$ *are fresh for* Δ*, then* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}<:\mathtt{T}}\mathtt{S}'$ *and* $\Delta \vdash \mathtt{S}' <: \mathtt{S}''$.

3. If $\Delta \vdash \mathtt{U} <: \mathtt{T}$ *and* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:>\mathtt{U}}\mathtt{S}''$ *where* $dom(\Delta', \mathtt{X}:>\mathtt{U})$ *are fresh for* Δ*, then* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:>\mathtt{T}}\mathtt{S}'$ *and* $\Delta \vdash \mathtt{S}' <: \mathtt{S}''$.

4. If $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(*,\mathtt{U})}\mathtt{S}''$ *where* $dom(\Delta') \cup \{\mathtt{X}\}$ *are fresh for* Δ*, then* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{S}'$ *and* $\Delta \vdash \mathtt{S}' <: \mathtt{S}''$.

Proof. We prove by structural induction on $\mathtt{S}$.

1)

Case $\mathtt{S}{=}\mathtt{Y}$, where $\mathtt{Y}$ is a type variable. $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{S}''$.

If $\mathtt{Y} \neq \mathtt{X}$, $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{Y}$, and $\mathtt{Y}\Downarrow_{\Delta'}\mathtt{Y}$.

If $\mathtt{Y}{=}\mathtt{X}$, $\mathtt{v}{=}{+}$, $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{T}$. $[\mathtt{T}/\mathtt{X}]\mathtt{Y}{=}\mathtt{T}$, $\mathtt{T}\Downarrow_{\Delta'}\mathtt{T}$.

If $\mathtt{Y} \not\in dom(\Delta', \mathtt{X}:(\mathtt{v},\mathtt{T}))$, then $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{Y}$, $\mathtt{Y}\Downarrow_{\Delta'}\mathtt{Y}$.

Case $\mathtt{S}{=}\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}$, $\mathtt{S}''{=}\mathtt{C}\texttt{<}\overline{\mathtt{w}}\overline{\mathtt{T}}'\texttt{>}$.

By inspecting the three cases in the definition of $(\mathtt{w}_i, \mathtt{T}_i')$ and the induction hypothesis, it's easy to show that $[\mathtt{T}/\mathtt{X}]\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}\Downarrow_{\Delta'}\mathtt{C}\texttt{<}\overline{\mathtt{w}'\mathtt{T}''}\texttt{>}$ such that $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{w}'\mathtt{T}''}\texttt{>} <: \mathtt{C}\texttt{<}\overline{\mathtt{w}\mathtt{T}'}\texttt{>}$.

2), 3), and 4) can be proven similarly through induction. □

Lemma 3.4.11. *If* $\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}$ *ok,* $\Delta \vdash bound_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>}$*, then* $\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash bound_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{U}}\texttt{>}$.

Proof. By straight-forward induction on the derivation of open rules. □

Theorem 4. ***[Erasure Preserves Typing]*** *For a program* $(CT, \mathtt{e})$*, if* CT *is ok, and* $\Delta; \Gamma \vdash_{FCJ} \mathtt{e} \in \mathtt{T}$ *under the raw type restriction, then* $|\Delta|_\Delta; |\Gamma|_\Delta \vdash_{FGJ_v} |\mathtt{e}|_{\Delta,\Gamma} \in |T|_\Delta$.

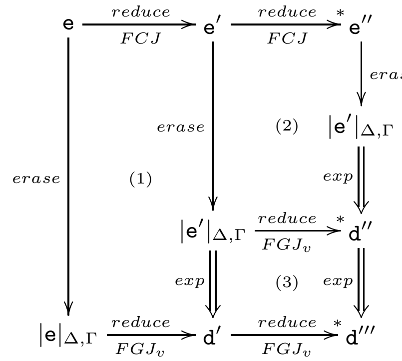
Proof. The theorem is proven by applying Lemma 5.1 and Lemma 5.2. □

Theorem 5. ***[Erasure Preserves Execution Modulo Expansion]*** *If* $\Delta; \Gamma \vdash_{FCJ} \mathtt{e} \in \mathtt{T}$, $\mathtt{e} \longrightarrow_{FCJ}{}^* \mathtt{e}'$*, and* $\mathtt{e}$ *and* $\mathtt{e}'$ *are well-typed under the raw type restriction, then there exists some FGJ$_v$expression* $\mathtt{d}'$*such that* $|\mathtt{e}'|_{\Delta,\Gamma} \overset{exp}{\Longrightarrow} \mathtt{d}'$ *and* $|\mathtt{e}|_{\Delta,\Gamma} \longrightarrow_{FGJ_v} \mathtt{d}'$.

Proof. We prove by induction on the number of steps in reduction $\longrightarrow$.

Base case $n = 0$: $\mathtt{e} = \mathtt{e}'$. Trivial.

Induction hypothesis: assume theorem is true for n. For $n + 1$, we have the following:



Commutation (1) is proved by Lemma 5.3, (2) by the induction hypothesis, and (3) by Lemma 5.4. □

Lemma 5.1. *If* $CT(\mathtt{C})$ = `class C<X̄◁N̄> <X̄◁R̄>?◁D<S̄> {T̄ f̄; M̄}`*, then* |`class C<X̄◁N̄> <X̄◁R̄>?◁D<S̄> {T̄ f̄; M̄}`| *is ok.*

Proof. |`class C<X̄◁N̄> <X̄◁R̄>?◁D<S̄> {T̄ f̄; M̄}`| = `class C<X̄◁`$|\overline{\mathtt{N}}|_{\Delta_d}$`>◁`$|$`D<S̄>`$|_\Delta$ { $|\overline{\mathtt{T}}|_{\Delta_d}$ $\overline{\mathtt{f}}$; $|\overline{\mathtt{M}}|_{\Delta_d,\Gamma}$ }

$\Delta_d = \overline{\mathtt{X}} <: \overline{\mathtt{N}}$ $\quad \Delta_{tc} = \overline{\mathtt{X}} <: \overline{\mathtt{R}}$ $\quad \Delta = \Delta_d, \Delta_{tc}$

By T-CLASS,

$\Delta_d \vdash \overline{\mathtt{N}}$ ok $\quad \Delta_d, \Delta_{tc} \vdash \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ ok $\quad \Delta_d \vdash \mathtt{T}$ ok

$\overline{\mathtt{M}}$ ok in $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\texttt{>}$

By Lemma 5.1.1 and 5.2.2(1),

$\overline{\mathtt{X}} <: |\overline{\mathtt{N}}|_{\Delta_d} \vdash |\overline{\mathtt{N}}|_{\Delta_d}$, $|\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}|_\Delta$, $|\overline{\mathtt{T}}|_{\Delta_d}$ ok.

By Lemma 5.1.4, $|\overline{\mathtt{M}}|_{\Delta_d,\Gamma}$ ok in $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\texttt{>}$

By T-CLASS$_{FGJ_v}$, the erased class $\mathtt{C}$ is ok. □

Lemma 5.1.1. *If* $\Delta \vdash \mathtt{N}$ *ok, then* $|\Delta|_\Delta \vdash |\mathtt{N}|_\Delta$ *ok.*

Proof. We prove by induction on the FCJ well-formed type rules.

Case WF-OBJECT: Trivial with the WF-OBJECT$_{FGJ_v}$ rule.

Case WF-VAR: $\mathtt{N}{=}\mathtt{X}$, $\mathtt{X} \in dom(\Delta)$

By E-TYPE-VAR: $|\mathtt{X}|_\Delta{=}\mathtt{X}$

By definition: $\mathtt{X} \in dom(|\Delta|_\Delta)$

By WF-TVAR$_{FGJ_v}$, $|\Delta|_\Delta \vdash \mathtt{X}$ ok.

Case WF-CLASS: $\mathtt{N}{=}\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}$.

$CT(\mathtt{C})$=`class C<X̄◁N̄> <X̄◁R̄>?◁D<S̄>` $\{\ldots\}$

$\Delta \vdash \overline{\mathtt{T}}$ ok $\quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By Lemma 5.1: `class C<X̄◁`$|\overline{\mathtt{N}}|_\Delta$`>◁`$|$`D<S̄>`$|_\Delta\{\ldots\}$ ok.

By induction hypothesis: $|\Delta|_\Delta \vdash |\overline{\mathtt{T}}|_\Delta$ ok.

By Lemma 5.1.2: $|\Delta|_\Delta \vdash |\overline{\mathtt{T}}|_\Delta <: [|\overline{\mathtt{T}}|_\Delta/\overline{\mathtt{X}}]|\overline{\mathtt{N}}|_\Delta$.

Then by WF-CLASS$_{FGJ_v}$, $|\Delta|_\Delta \vdash |\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}|_\Delta$ ok. □

Lemma 5.1.2. ***[Erasure Preserves Subtyping]****. If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$*, then* $|\Delta|_\Delta \vdash |\mathtt{S}|_\Delta <: |\mathtt{T}|_\Delta$.

Proof. We prove by induction on the FCJ subtyping rules.

Case S-REFL: Trivial by S-REFL$_{FGJ_v}$

Case S-TRANS: Trivial by induction hypothesis and S-TRANS$_{FGJ_v}$.

Case S-UBOUND: $\Delta(\mathtt{X}) = (+, \mathtt{T})$. By E-VARIANCE, $|\Delta|_\Delta(\mathtt{X}) = (+, |\mathtt{T}|_\Delta)$. And by S-UBOUND$_{FGJ_v}$, $|\Delta|_\Delta \vdash \mathtt{X} <: |\mathtt{T}|_\Delta$.

Case S-LBOUND: Similar to S-UBOUND case.

Case S-CLASS: $\mathtt{S}{=}\mathtt{C}\texttt{<}\overline{\mathtt{v}}\overline{\mathtt{T}}\texttt{>}$

$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\mathtt{>?D<}\overline{\mathtt{S}}\mathtt{>}\ \{\ldots\}$
$\Delta\vdash\mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}\Uparrow^{\Delta'}\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}\quad \Delta\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}\quad ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>})\Downarrow_{\Delta'}\mathtt{T}$

By E-CLASS and Lemma 5.1:
$\mathtt{class\ C<}\overline{\mathtt{X}}\lhd|\overline{\mathtt{N}}|_\Delta\mathtt{>}\lhd|\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}|_\Delta\{\ldots\}$ ok.
By Lemma 5.1.3(1): $|\Delta|_\Delta\vdash|\mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}|_\Delta\Uparrow^{|\Delta'|_\Delta}|\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}|_\Delta$.
By Lemma 5.1.3(2): $|([\overline{\mathtt{U}}/\overline{\mathtt{S}}])\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}|_\Delta\Downarrow_{|\Delta'|_\Delta}|\mathtt{T}|_\Delta$.
By S-CLASS$_{FGJ_v}$, $|\Delta|_\Delta\vdash|\mathtt{C<}\overline{\mathtt{vT}}|_\Delta<:|\mathtt{T}|_\Delta$.

□

Lemma 5.1.3. *1. If* $\Delta\vdash\mathtt{T}\Uparrow^{\Delta'}\mathtt{S}$ *then* $|\Delta|_\Delta\vdash|\mathtt{T}|_\Delta\Uparrow^{|\Delta'|_\Delta}|\mathtt{S}|_\Delta$
2. If $\mathtt{T}\Downarrow_{\Delta'}\mathtt{S}$*, then* $|\mathtt{T}|_\Delta\Downarrow_{|\Delta'|_\Delta}|\mathtt{S}|_\Delta$.

Proof. Proofs are simple by induction on rules for $\Uparrow^{|\Delta'|_\Delta}{}_{FGJ_v}$ and $\Downarrow_{|\Delta'|_\Delta FGJ_v}$.

□

Lemma 5.1.4. *If* $\mathtt{M}$ *ok in* $\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}$*, then* $|\mathtt{M}|_{\Delta,\Gamma}$ *ok in* $|\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}|_\Delta$.

Proof. Let $\mathtt{M}=\mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\mathtt{>?<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>\ T_0\ m\ (}\overline{\mathtt{T}}\mathtt{,\ }\overline{\mathtt{x}}\mathtt{)}\{\uparrow\mathtt{e};\}$

By T-METHOD,
$\Delta_d=\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\quad \Delta_{tc}=\overline{\mathtt{X}}<:\overline{\mathtt{R}}\quad \Delta_d,\Delta_{tc}\vdash\overline{\mathtt{P}},\ \overline{\mathtt{T}},\ \mathtt{T_0}\ ok$
$\Delta_d,\Delta_{tc};\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this:C<}\overline{\mathtt{X}}\mathtt{>}\vdash\mathtt{e_0}\in\mathtt{S_0}\quad \Delta_d,\Delta_{tc}\vdash\mathtt{S_0}<:\mathtt{T_0}$
$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\lhd\overline{\mathtt{V}}\mathtt{>?}\lhd\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\{\ldots\}$
$override(\mathtt{m},\ \mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{V}}\mathtt{>?D<}\overline{\mathtt{S}}\mathtt{>},\ \mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\mathtt{>?<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}}\to\mathtt{T_0})$

By E-METHOD,
$|\mathtt{M}|_{\Delta,\Gamma}=$
$\mathtt{<}\overline{\mathtt{Y}}\lhd|\overline{\mathtt{P}}|_\Delta\mathtt{>}\ |\mathtt{T_0}|_\Delta\ \mathtt{m}\ \mathtt{(}|\overline{\mathtt{T}}|_\Delta\ \overline{\mathtt{x}}\mathtt{')}\ \{\uparrow[(|\overline{\mathtt{T}}|_\Delta)\overline{\mathtt{x}}\mathtt{'}/\overline{\mathtt{x}}]|\mathtt{e_0}|_{\Delta,\Gamma}\ ;\}$

By definition,
$|\Delta_d|_\Delta=\overline{\mathtt{X}}<:|\overline{\mathtt{N}}|_\Delta,\overline{\mathtt{Y}}<:|\mathtt{P}|_\Delta\quad |\Delta_{tc}|_\Delta=\overline{\mathtt{X}}<:|\mathtt{R}|_\Delta$

By Lemma 5.1.1 and 5.2.2,
$\overline{\mathtt{X}}<:|\overline{\mathtt{N}}|_\Delta,\overline{\mathtt{Y}}<:|\overline{\mathtt{P}}|_\Delta\vdash|\overline{\mathtt{P}}|_\Delta,|\overline{\mathtt{T}}|_\Delta,|\mathtt{T_0}|_\Delta$ ok

By Lemma 5.2,
$|\Delta_d|_\Delta,|\Delta_{tc}|_\Delta;\overline{\mathtt{x}}:|\overline{\mathtt{T}}|_\Delta,\mathtt{this:}|\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}|_\Delta\vdash|\mathtt{e_0}|_{\Delta,\Gamma}\in|\mathtt{S_0}|_\Delta$

By Lemma 5.2.2(2),
$|\Delta_d|_\Delta;\overline{\mathtt{x}}:|\overline{\mathtt{T}}|_\Delta,\mathtt{this:}|\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}|_\Delta\vdash|\mathtt{e_0}|_{\Delta,\Gamma}\in|\mathtt{S_0}|_\Delta$

It is then obvious from substitution and type of casting rules that,
$|\Delta_d|_\Delta;\overline{\mathtt{x}}':|\overline{\mathtt{T}}|_\Delta,\mathtt{this:}|\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}|_\Delta\vdash[(|\overline{\mathtt{T}}|_\Delta)\overline{\mathtt{x}}'/\overline{\mathtt{x}}]|\mathtt{e_0}|_{\Delta,\Gamma}\in|\mathtt{S_0}|_\Delta$

By Lemma 5.2.2(3): $|\Delta_d|_{\Delta_d}\vdash|\mathtt{S_0}'|_\Delta<:|\mathtt{T_0}|_\Delta$

By E-CLASS: $\mathtt{class\ C<}\overline{\mathtt{X}}\lhd|\overline{\mathtt{N}}|_\Delta\mathtt{>}\lhd|\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}|_\Delta\{\ldots\}$

By Lemma 5.2.3:
$override_{FGJ_v}(\mathtt{m},\ |\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}|_\Delta,\ \mathtt{<}\overline{\mathtt{Y}}\lhd|\overline{\mathtt{P}}|_\Delta\mathtt{>}|\overline{\mathtt{T}}|_\Delta\to|\mathtt{T_0}|_\Delta)$

And finally, by T-METHOD$_{FGJ_v}$, $|\mathtt{M}|_{\Delta,\Gamma}$ ok in $|\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}|_\Delta$.

□

Lemma 5.2. *If* $\Delta;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*, then* $|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\mathtt{e}|_{\Delta,\Gamma}\in|\mathtt{T}|_\Delta$.

Proof. We prove by induction on the expression typing rules.
$Case$ T-VAR: Trivial.
$Case$ T-FIELD: $\mathtt{e}=\mathtt{e_0.f}_i$
$\Delta;\Gamma\vdash\mathtt{e_0}\in\mathtt{T_0}\quad \Delta\vdash bound_\Delta(\mathtt{T_0})\Uparrow^{\Delta'}\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}$
$fields(bound_\Delta(\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}))=\overline{\mathtt{S}}\ \overline{\mathtt{f}}\quad \mathtt{S}_i\Downarrow_{\Delta'}\mathtt{T}$

This expression can be erased via two rules, E-FIELD, and E-FIELD-CAST. The case of E-FIELD-CAST is trivial because of the T-CAST rule. We now show this lemma is true when, through E-FIELD, $|\mathtt{e}|_{\Delta,\Gamma}=|\mathtt{e_0}|_{\Delta,\Gamma}\mathtt{.f}$.

By induction hypothesis: $|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\mathtt{e_0}|_{\Delta,\Gamma}\in|\mathtt{T_0}|_\Delta$.

By Lemma 5.1.3(1): $|\Delta|_\Delta\vdash bound_{|\Delta|_\Delta}(\mathtt{T_0})\Uparrow^{|\Delta'|_\Delta}|\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}|_\Delta$.

By Lemma 5.2.1, if $fields(bound_{|\Delta|_\Delta}(|\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}|_\Delta))=\overline{\mathtt{S}}'\overline{\mathtt{f}}'$, then there exists $\mathtt{S}'_i$ such that $|\mathtt{S}_i|_\Delta=\mathtt{S}'_i$.

By Lemma 5.1.3(2), $\mathtt{S}'_i\Downarrow_{|\Delta'|_\Delta}|\mathtt{T}|_\Delta$.

By T-FIELD$_{FGJ_v}$, $|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\mathtt{e_0}|_{\Delta,\Gamma}\mathtt{.f}\in|\mathtt{T}|_\Delta$.

$Case$ T-INVK: $\mathtt{e}=\mathtt{e_0.<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}$.
$\Delta;\Gamma\vdash\mathtt{e_0}\in\mathtt{T_0}\quad \Delta\vdash bound_\Delta(\mathtt{T_0})\Uparrow^{\Delta'}\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$
$mtype(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U}'_0$
$\{\overline{\mathtt{Y}}\}\cap dom(\Delta')=\emptyset\quad \Delta\vdash\overline{\mathtt{V}}$ ok
$\Delta,\Delta'\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}\quad \Delta;\Gamma\vdash\mathtt{e}\in\overline{\mathtt{S}}$
$\Delta,\Delta'\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U_0}\Downarrow_{\Delta'}\mathtt{T}$

There are two rules we can use to erase this expression. The proof for the E-INVK-CAST is rivial because of the inserted cast and the T-CAST rule. We now show that the lemma is true for $|\mathtt{e}|_{\Delta,\Gamma}=|\mathtt{e_0}|_{\Delta,\Gamma}\mathtt{.m<}|\overline{\mathtt{V}}|_\Delta\mathtt{>(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}$.

By the induction hypothesis:
$|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\mathtt{e_0}|_{\Delta,\Gamma}\in|\mathtt{T_0}|_\Delta$

By Lemma 5.1.3(1)
$|\Delta|_\Delta\vdash bound_{|\Delta|_\Delta}(|\mathtt{T_0}|_\Delta)\Uparrow^{|\Delta'|_\Delta}\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$

By Lemma 5.2.5,
$mtype_{FGJ_v}(\mathtt{m},\ |\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta)=\mathtt{<}\overline{\mathtt{Y}}\lhd|\overline{\mathtt{P}}|_\Delta\mathtt{>}|\overline{\mathtt{U}}|_\Delta\to|\mathtt{U_0}|_\Delta$

By definition, $dom(\Delta')=dom(|\Delta'|_\Delta)$, and thus:
$\{\overline{\mathtt{Y}}\}\cap dom(|\Delta'|_\Delta)=\emptyset$

By Lemma 5.1.1,
$|\Delta|_\Delta\vdash|\overline{\mathtt{V}}|_\Delta$ ok

By Lemma 5.1.2,
$|\Delta|_\Delta,|\Delta'|_\Delta\vdash|\overline{\mathtt{V}}|_\Delta<:[|\overline{\mathtt{V}}|_\Delta/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$

By induction hypothesis,
$|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\mathtt{e}|_{\Delta,\Gamma}\in\overline{\mathtt{S}}$

By Lemma 5.1.2,
$|\Delta|_\Delta,|\Delta'|_\Delta\vdash\overline{\mathtt{S}}<:[|\overline{\mathtt{V}}|_\Delta/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$

By Lemma 5.1.3(2),
$[|\overline{\mathtt{V}}|_\Delta/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\Downarrow_{|\Delta'|_\Delta}|\mathtt{T}|_\Delta$

Finally, by T-INVK$_{FGJ_v}$, lemma holds.

$Case$ T-NEW: $\mathtt{e}=\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$, $\mathtt{T}=\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$.
$\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ ok $\quad fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$
$\Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}\quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

By Lemma 5.1.1,
$|\Delta|_\Delta\vdash|\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta$ ok

By Lemma 5.2,
$|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\overline{\mathtt{e}}|_{\Delta,\Gamma}\in|\overline{\mathtt{S}}|_{\Delta,\Gamma}$

By Lemma 5.1.2,
$|\Delta|_\Delta\vdash|\overline{\mathtt{S}}|_\Delta\in|\overline{\mathtt{U}}|_\Delta$

We show this lemma holds for each of the two erasure rules applicable.
$Subcase$ E-NEW:
$|\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma}=\mathtt{new}\ |\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}$
$\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\mathtt{>?}\lhd\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ldots\}$
$\mathtt{class\ D<}\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}\mathtt{>\ <}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}'\mathtt{>?}\lhd\mathtt{E<}\overline{\mathtt{U}}\mathtt{>}\{\ldots\}$
$\Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}$

By Lemma E-CLASS and T-FIELD$_{FGJ_v}$,
$fields(|\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta)=|\overline{\mathtt{U}}|_\Delta\ \overline{\mathtt{f}}$

And by T-NEW$_{FGJ_v}$, this lemma holds.

$Subcase$ E-NEW-FIELDS:
$|\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma}=\mathtt{new}\ |\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Delta\mathtt{(}|\overline{\mathtt{e}}'|_{\Delta,\Gamma}\mathtt{,}\ |\overline{\mathtt{e}}|_{\Delta,\Gamma}\mathtt{)}$, where:
$\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>\ <}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\mathtt{>?}\lhd\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ldots\}$
$\mathtt{class\ D<}\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}\mathtt{>\ <}\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}'\mathtt{>?}\lhd\mathtt{E<}\overline{\mathtt{U}}\mathtt{>}\{\ldots\}$
$\Delta\nvdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}\quad fields_{uc}(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>})=\overline{\mathtt{D}}\ \overline{\mathtt{g}}$
$\sharp(\overline{\mathtt{e}}')=\sharp(\overline{\mathtt{D}})\quad \mathtt{e}'_i=(\mathtt{D}_i)\mathtt{new\ Object()}$

Clearly, $\overline{\mathtt{e}}'\mathtt{,}\overline{\mathtt{e}}\in\overline{\mathtt{D}}$, $\overline{\mathtt{U}}$, where $\Delta;\Gamma\vdash\overline{\mathtt{e}}'\in\overline{\mathtt{D}}$. And by T-FIELD, $\Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$ where $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$. By the induction hypothesis and Lemma 5.1.2,
$|\Delta|_\Delta;|\Gamma|_\Delta\vdash|\overline{\mathtt{e}}'\mathtt{,}\overline{\mathtt{e}}|_{\Delta,\Gamma}\in|\overline{\mathtt{D}}|_\Delta\mathtt{,}|\overline{\mathtt{U}}|_\Delta$

And by T-NEW$_{FGJ_v}$, the lemma holds.

Case T-CAST, T-SCAST: $\mathtt{e}{=}(\mathtt{T})\mathtt{e}_0$

These are obvious from the E-CAST rule and Lemma 5.1.2.

□

Lemma 5.2.1. *If* $\mathit{fields}(\mathtt{T}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$, *then* $\mathit{fields}_{FGJ_v}(|\mathtt{T}|_\triangle) = \overline{\mathtt{S}}'\ \overline{\mathtt{f}}'$, *such that for each* $\mathtt{S}_i{\in}\overline{\mathtt{S}}$, *there exists a* $\mathtt{S}'_j{\in}\overline{\mathtt{S}}'$ *such that* $|\mathtt{S}_i|_\triangle{=}\mathtt{S}'_j$, *and* $\mathtt{f}_i{=}\mathtt{f}'_j$.

Proof. We prove by induction on $\mathtt{T}$.

Case $\mathtt{T}{=}\mathtt{Object}$:

$\mathit{fields}(\mathtt{Object}) = \bullet$, $|\mathtt{Object}|_\triangle{=}\mathtt{Object}$, and
$\mathit{fields}_{FGJ_v}(\mathtt{Object}) = \bullet$.

Case $\mathtt{T}{=}\mathtt{C}{<}\overline{\mathtt{T}}{>}$:

There are two rules defining $\mathit{fields}(\mathtt{C}{<}\overline{\mathtt{T}}{>})$. Both have as given:
$CT(\mathtt{C}){=}\mathtt{class\ C}{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}{>}\ {<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{R}}{>}?{\lhd}\mathtt{D}{<}\overline{\mathtt{U}}{>}\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \ldots\}$

Subcase $\Delta\vdash\overline{\mathtt{T}}{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$:

$\mathit{fields}(\mathtt{C}{<}\overline{\mathtt{T}}{>}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\quad [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}$
$\mathit{fields}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}{<}\overline{\mathtt{U}}{>}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}$

By E-CLASS and definition of fields_{FGJ_v}, $\mathit{fields}_{FGJ_v}(|\mathtt{C}{<}\overline{\mathtt{T}}{>}|_\triangle) = \overline{\mathtt{D}}'\ \overline{\mathtt{g}}',\ |\overline{\mathtt{S}}|_\triangle\ \overline{\mathtt{f}}$, where $\overline{\mathtt{D}}'\ \overline{\mathtt{g}}'{=}\ \mathit{fields}_{FGJ_v}(|\mathtt{D}{<}\overline{\mathtt{S}}{>}|_\triangle)$.

If $\mathtt{S}_i{\in}\overline{\mathtt{S}}$, proof is simple. If $\mathtt{S}_i{\in}\overline{\mathtt{D}}$, by induction hypothesis, there is some type $\mathtt{D}'_j{\in}\overline{\mathtt{D}}'$ such that $|\mathtt{S}_i|_\triangle{=}\mathtt{D}'_j$, and $\mathtt{g}'_j{=}\mathtt{f}_i$.

Subcase $\Delta\not\vdash\overline{\mathtt{T}}{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$:

In this case, $\mathit{fields}(\mathtt{C}{<}\overline{\mathtt{T}}{>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$. Then clearly the lemma holds by E-CLASS and definition of fields_{FGJ_v}.

□

Lemma 5.2.2. *1. If* $\Delta_d, \Delta_{tc} \vdash \mathtt{T}$ *ok, then* $|\Delta_d|_{\Delta_d} \vdash |\mathtt{T}|_{\Delta_d}$ *ok.*
2. If $\Delta_d, \Delta_{tc}; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, *then* $|\Delta_d|_{\Delta_d}; |\Gamma|_{\Delta_d} \vdash |\mathtt{e}|_{\Delta_d,\Gamma} \in |\mathtt{T}|_{\Delta_d}$.
3. If $\Delta_d, \Delta_{tc} \vdash \mathtt{S} <: \mathtt{T}$, *then* $|\Delta_d|_{\Delta_d} \vdash |\mathtt{S}|_{\Delta_d}{<:}|\mathtt{T}|_{\Delta_d}$.

Proof. 1. We prove by induction on the well-formed type rules.

Case WF-OBJECT: $\mathtt{T}{=}\mathtt{Object}$.
Proof is obvious from WF-OBJECT$_{FGJ_v}$.

Case WF-VAR:
Easy by definition $dom(\Delta_{tc}) \subseteq dom(\Delta_d)$ and E-TYPE-VAR.

Case WF-CLASS:
Base on our assumption that E-TYPE-RAW is never used, we know that $\Delta_d \vdash \overline{\mathtt{T}}{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$. Proof is then simple by Lemma 5.1, Lemma 5.1.2, and Lemma 5.1.1.

2. We prove by induction on expression typing rules.

Case T-VAR: trivial.

Case T-FIELD: $\mathtt{e}{=}\mathtt{e}_0.\mathtt{f}$

$\Delta;\Gamma\vdash\mathtt{e}_0{\in}\mathtt{T}_0 \qquad \Delta\vdash bound_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}{<}\overline{\mathtt{U}}{>}$
$\mathit{fields}(bound_\Delta(\mathtt{C}{<}\overline{\mathtt{U}}{>})) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \qquad \mathtt{S}_i{\Downarrow}_{\Delta'}\mathtt{T}$

By induction hypothesis,
$|\Delta_d|_{\Delta_d}; |\Gamma|_{\Delta_d} \vdash |\mathtt{e}_0|_{\Delta_d,\Gamma} \in |\mathtt{T}_0|_{\Delta_d}$.

By Lemma 5.1.3(1),
$|\Delta_d|_{\Delta_d} \vdash bound_{|\Delta|_\triangle}(|\mathtt{T}_0|_{\Delta_d})\Uparrow^{|\Delta'|_\triangle}|\mathtt{C}{<}\overline{\mathtt{U}}{>}|_{\Delta_d}$

By Lemma 5.2.1 and 5.1.3(2),
$\mathit{fields}(|\mathtt{C}{<}\overline{\mathtt{U}}{>}|_{\Delta_d}) = \overline{\mathtt{S}}'\ \overline{\mathtt{f}}$, and $\mathtt{S}'_j{\Downarrow}_{|\Delta'|_\triangle}|\mathtt{T}|_\triangle$ for some $\mathtt{S}'_j{=}|\mathtt{S}_i|_{\Delta_d}$.

Case T-INVK, T-NEW, T-CAST, and T-SCAST:
The proofs are straightforward from the conditions of T-INVK, induction hypothesis, (1) and (3) of this lemma, and Lemma 5.1.3.

3. We prove by induction on the subtyping rules.

Case S-REFL, S-TRANS:
Easy by induction hypothesis.

Case S-UBOUND, S-LBOUND:
Easy because $dom(\Delta_{tc}) \subseteq dom(\Delta_d)$.

Case S-CLASS:
Easy because the S-CLASS rule already uses Δ_d only in the $\Uparrow^{\Delta'}$.

Case S-VAR: Easy by induction hypothesis.

□

Lemma 5.2.3. *If* $ovrride(\mathtt{m},\ {<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{V}}{>}?\mathtt{D}{<}\overline{\mathtt{S}}{>},\ {<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{R}}{>}?{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{T}}{\to}\mathtt{T}_0)$, *then* $override_{FGJ_v}(\mathtt{m},\ |\mathtt{D}{<}\overline{\mathtt{S}}{>}|_\triangle,\ {<}\overline{\mathtt{Y}}{\lhd}|\overline{\mathtt{P}}|_\triangle{>}|\overline{\mathtt{T}}|_\triangle{\to}|\mathtt{T}_0|_\triangle)$.

Proof. From Lemma 5.2.4 and $mtype_{uc}(\mathtt{m},\ \mathtt{D}{<}\overline{\mathtt{S}}{>}) = (\Delta',{<}\overline{\mathtt{Z}}{\lhd}\overline{\mathtt{Q}}{>}\overline{\mathtt{T}}{\to}\mathtt{T}_0)$, we have:

$mtype(\mathtt{m},\ |\mathtt{D}{<}\overline{\mathtt{S}}{>}|_\triangle) = ({<}\overline{\mathtt{Z}}{\lhd}|\overline{\mathtt{Q}}|_\triangle{>}|\overline{\mathtt{T}}|_\triangle{\to}|\mathtt{T}_0|_\triangle)$.

By definition of erasure, from $(\overline{\mathtt{T}},\ \mathtt{T}_0,\ [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\overline{\mathtt{Q}}) = (\overline{\mathtt{U}},\ \mathtt{U}_0,\ \overline{\mathtt{P}})$, we obtain: $(|\overline{\mathtt{T}}|_\triangle,\ |\mathtt{T}_0|_\triangle,\ |[\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\overline{\mathtt{Q}}|_\triangle) = (|\overline{\mathtt{U}}|_\triangle,\ |\mathtt{U}_0|_\triangle,\ |\overline{\mathtt{P}}|_\triangle)$.

We've show both conditions for $override_{FGJ_v}$ hold.

□

Lemma 5.2.4. *If* $mtype_{uc}(\mathtt{m},\ \mathtt{C}{<}\overline{\mathtt{T}}{>}) = (\Delta',{<}\overline{\mathtt{Z}}{\lhd}\overline{\mathtt{Q}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0)$, *then* $mtype(\mathtt{m},\ |\mathtt{C}{<}\overline{\mathtt{T}}{>}|_\triangle) = {<}\overline{\mathtt{Z}}{\lhd}|\overline{\mathtt{Q}}|_\triangle{>}|\overline{\mathtt{U}}|_\triangle{\to}|\mathtt{U}_0|_\triangle$.

Proof. We prove by anaylizing each rule for $mtype_{uc}$ separately:

Case MT$_{\mathrm{UC}}$-CLASS, MT$_{\mathrm{UC}}$-SUPER-1:

We have:
$CT(\mathtt{C}){=}\mathtt{class\ C}{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}{>}\ {<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{Q}}{>}?{\lhd}\mathtt{D}{<}\overline{\mathtt{S}}{>}\{\ldots\overline{\mathtt{M}}\}$, and
${<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{R}}{>}?{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\mathtt{U}'_0\ \mathtt{m}\ (\overline{\mathtt{U}}'\ \overline{\mathtt{x}})\{\uparrow\mathtt{e};\}{\in}\overline{\mathtt{M}}$
where: $\overline{\mathtt{Z}}{=}\overline{\mathtt{Y}},\ \overline{\mathtt{Q}}{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{P}},\ \overline{\mathtt{U}}{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}'$, and $\mathtt{U}_0{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}'_0$.

It is clear from the erasure rule for E-CLASS and E-METHOD that MT-CLASS$_{FGJ_v}$'s conditions are satisfied.

Case MT$_{\mathrm{UC}}$-SUPER-2: Easy from the erasure rule for E-CLASS and the unduction hypothesis.

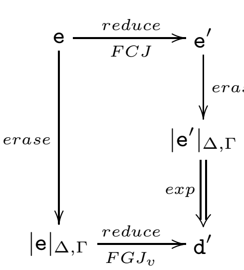
□

Lemma 5.2.5. *If* $mtype(\mathtt{m},\ \mathtt{C}{<}\overline{\mathtt{T}}{>}) = {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0$, *then* $mtype_{FGJ_v}(\mathtt{m},\ |\mathtt{C}{<}\overline{\mathtt{T}}{>}|_\triangle) = {<}\overline{\mathtt{Y}}{\lhd}|\overline{\mathtt{P}}|_\triangle{>}|\overline{\mathtt{U}}|_\triangle{\to}|\mathtt{U}_0|_\triangle$.

Proof. Proof is easy from applying erasure rules to the conditions in both MT-CLASS and MT-SUPER.

□

Lemma 5.3. *If* $\Delta;\Gamma\vdash\mathtt{e}{\in}\mathtt{T}$ *and* $\mathtt{e}{\longrightarrow}_{FCJ}\mathtt{e}'$, *then there exist some FGJ expression* $\mathtt{d}'$*such that* $|\mathtt{e}'|_{\Delta,\Gamma}\overset{exp}{\Longrightarrow}\mathtt{d}'$, *and* $|\mathtt{e}|_{\Delta,\Gamma}{\longrightarrow}_{FGJ_v}\mathtt{d}'$. *In another word, the following diagram commutes:*



Proof. We prove by induction on the reduction rules for $\mathtt{e}{\longrightarrow}_{FCJ}\mathtt{e}'$.

Case R-FIELD: $\mathtt{e}{=}\mathtt{new\ C}{<}\overline{\mathtt{T}}{>}(\overline{\mathtt{e}}).\mathtt{f}_i$, $\mathtt{e}'{=}\mathtt{e}_i$.

Erasing by E-NEW:

$|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i|_\Delta = \texttt{new } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta\texttt{(}|\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).f}_i$. By R-FIELD$_{FGJ_v}$, $\texttt{new } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta\texttt{(}|\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).f}_i \longrightarrow_{FGJ_v} |\texttt{e}_i|_{\Delta,\Gamma}$. Clearly $|\texttt{e}'|_{\Delta,\Gamma} \stackrel{exp}{\Longrightarrow} \texttt{d}'$.

Erasing by E-NEW-FIELDS:

$|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i|_\Delta = \texttt{new } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta\texttt{(}|\overline{\texttt{e}}'|_{\Delta,\Gamma}\texttt{, } |\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).f}_i$. We know by T-FIELD that $\texttt{f}_i$ cannot refer to $\overline{\texttt{e}}'$. By R-FIELD$_{FGJ_v}$, $\texttt{new } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta\texttt{(}|\overline{\texttt{e}}'|_{\Delta,\Gamma}\texttt{, } |\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{).f}_i \longrightarrow_{FGJ_v} |\texttt{e}_i|_{\Delta,\Gamma}$. Again, $|\texttt{e}'|_{\Delta,\Gamma} \stackrel{exp}{\Longrightarrow} \texttt{d}'$

Case R-INVK:

$\texttt{e}=\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{d}}\texttt{)}$, $\texttt{e}'=[\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{,new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0$.

where $mbody(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, \texttt{e}_0)$.

By Lemma 5.4.1, we have:

$mbody(\texttt{m<}|\overline{\texttt{V}}|_\Delta\texttt{>, } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta) = (\overline{\texttt{x}}, |\texttt{e}_0|_{\Delta,\Gamma})_{FGJ_v}$.

$|\texttt{e}'|_{\Delta,\Gamma} = [\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{,}|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_\Delta\texttt{/this}]|\texttt{e}_0|_{\Delta,\Gamma}$.

There are two ways to erase e:

Subcase E-INVK:

$|\texttt{e}|_{\Delta,\Gamma} = |\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_\Delta\texttt{.m<}|\overline{\texttt{V}}|_\Delta\texttt{>(}|\overline{\texttt{d}}|_{\Delta,\Gamma}\texttt{)}$ By R-INVK$_{FGJ_v}$,

$|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{FGJ_v} [\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{, } |\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}\texttt{/this}].|\texttt{e}_0|_{\Delta,\Gamma}$

Clearly $|\texttt{e}'|_{\Delta,\Gamma} \stackrel{exp}{\Longrightarrow} \texttt{d}'$

Subcase E-INVK-CAST:

$|\texttt{e}|_{\Delta,\Gamma} = \texttt{(}|\texttt{T}|_\Delta\texttt{)}|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_\Delta\texttt{.m<}|\overline{\texttt{V}}|_\Delta\texttt{>(}|\overline{\texttt{d}}|_{\Delta,\Gamma}\texttt{)}$

where $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$.

By RC-CAST$_{FGJ_v}$,

$|\texttt{e}|_{\Delta,\Gamma} \longrightarrow_{FGJ_v} \texttt{(}|\texttt{T}|_\Delta\texttt{)}[\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{,}|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_\Delta\texttt{/this}]|\texttt{e}_0|_{\Delta,\Gamma}$.

We obtain $\texttt{d}'$ from $|\texttt{e}'|_{\Delta,\Gamma}$ by adding a synthetic cast $\texttt{(}|\texttt{T}|_\Delta\texttt{)}$.

Case R-CAST: $\texttt{e}=\texttt{(T)new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$, $\texttt{e}'=\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$, where $\emptyset \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{T}$.

By E-CAST, $|\texttt{e}|_{\Delta,\Gamma} = \texttt{(}|\texttt{T}|_\Delta\texttt{)}|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}$.

By Lemma 5.1.2, we have $\emptyset \vdash |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta <: |\texttt{T}|_\Delta$. And by R-CAST$_{FGJ_v}$, we have $\texttt{d}' = |\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}|_{\Delta,\Gamma}$.

By E-NEW, $|\texttt{e}'|_{\Delta,\Gamma} = \texttt{new } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta\texttt{(}|\overline{\texttt{e}}|_{\Delta,\Gamma}\texttt{)} = \texttt{d}'$

Case RC-FIELD: $\texttt{e}=\texttt{e}_0\texttt{.f}$, $\texttt{e}'=\texttt{e}'_0\texttt{.f}$, where $\texttt{e}_0 \longrightarrow_{FCJ} \texttt{e}'_0$.

By the induction hypothesis, we know that there exists a $\texttt{d}''$ such that $|\texttt{e}_0|_{\Delta,\Gamma} \longrightarrow_{FGJ_v} \texttt{d}''$, and $|\texttt{e}'_0|_{\Delta,\Gamma} \stackrel{exp}{\Longrightarrow} \texttt{d}''$.

There are two ways to erase e:

Subcase E-FIELD: $|\texttt{e}|_{\Delta,\Gamma} = |\texttt{e}_0|_{\Delta,\Gamma}\texttt{.f}$

Similarly, there are two ways to erase $\texttt{e}'$:

$|\texttt{e}'|_{\Delta,\Gamma} = |\texttt{e}'_0|_{\Delta,\Gamma}\texttt{.f}$: in this case, let $\texttt{d}'=\texttt{d}''\texttt{.f}$.

$|\texttt{e}'|_{\Delta,\Gamma} = \texttt{(}|\texttt{T}'|_\Delta\texttt{)}|\texttt{e}'_0|_{\Delta,\Gamma}\texttt{.f}$: in this case, let $\texttt{d}'=\texttt{(T}'\texttt{)d}''\texttt{.f}$.

Subcase E-FIELD-CAST: $|\texttt{e}|_{\Delta,\Gamma} = \texttt{(}|\texttt{T}|_\Delta\texttt{)}|\texttt{e}_0|_{\Delta,\Gamma}\texttt{.f}$. Similar.

Case RC-INV-RECV: $\texttt{e}=\texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}$, $\texttt{e}'=\texttt{e}'_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}$, where $\texttt{e}_0 \longrightarrow_{FCJ} \texttt{e}'_0$.

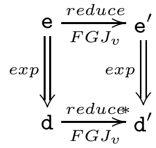
Similar to the case above.

Case RC-INV-ARG, RC-NEW-ARG, RC-CAST:

Easy by induction hypothesis.

□

Lemma 5.4. *If* e *is a FGJ expression,* $\texttt{e} \longrightarrow_{FGJ_v} \texttt{e}'$ *and* $\texttt{e} \stackrel{exp}{\Longrightarrow} \texttt{d}$, *then there exists some FGJ expression* $\texttt{d}'$ *such that* $\texttt{e}' \stackrel{exp}{\Longrightarrow} \texttt{d}'$ *and* $\texttt{d} \longrightarrow_{FGJ_v}{}^* \texttt{d}'$. *In other words, the following diagram commutes:*



Proof. We prove by induction on the FGJ reduction rules for $\texttt{e} \longrightarrow_{FGJ_v} \texttt{e}'$.

Case R-FIELD$_{FGJ_v}$: $\texttt{e}=\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i$, $\texttt{e}'=\texttt{e}_i$, where $fields_{FGJ_v}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{f}}$.

The expansions d can be of the form $\texttt{(D}_0\texttt{)...(D}_n\texttt{)new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i$, where for each $\texttt{D}_i$, $\texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{D}_i$, since we only introduce upcasts. It is then straightforward to see that

$\texttt{d} \longrightarrow_{FGJ_v}{}^* \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i \longrightarrow_{FGJ_v} \texttt{e}_i$.

Other cases are similarly straightforward.

□

Lemma 5.4.1. *If* $mbody(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, \texttt{e}_0)$, *then* $mbody_{FGJ_v}(\texttt{m<}|\overline{\texttt{V}}|_\Delta\texttt{>, } |\texttt{C<}\overline{\texttt{T}}\texttt{>}|_\Delta) = (\overline{\texttt{x}}, |\texttt{e}_0|_{\Delta,\Gamma})$.

Proof. We prove by induction on the rules for $mbody$:

Case MB-CLASS: $\texttt{e}_0 = [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{e}$, where:

$CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> <}\overline{\texttt{X}} \triangleleft \overline{\texttt{Q}}\texttt{>?} \triangleleft \texttt{D<}\overline{\texttt{S}}\texttt{> \{... }\overline{\texttt{M}}\texttt{\}}$

$\texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{R}}\texttt{>?<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>U}_0\texttt{ m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}}$

$\Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{R}}$.

From Lemma 5.1 and E-METHOD, the conditions for MB-CLASS$_{FGJ_v}$ are satisfied.

Case MB-SUPER: Simple by MB-SUPER, erasure rules for C, and the induction hypothesis.

□