JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 7, No. 5, Special Issue: Workshop on FTfJP 07, June 2008

# The Essence of Lightweight Family Polymorphism

**Chieri Saito**, Graduate School of Informatics, Kyoto University, Japan
**Atsushi Igarashi**, Graduate School of Informatics, Kyoto University, Japan

We have proposed lightweight family polymorphism, a programming style to support reusable yet type-safe mutually recursive classes, and introduced its formal core calculus .FJ. In this paper, we give a formal translation, which changes *only* type declarations, from .FJ into $\text{FGJ}_{\text{self}}$, an extension of Featherweight GJ with *self type variables*. They improve self typing and are required for the translation to preserve typing. Therefore we claim that self type variables represent the essential difference between .FJ and Featherweight GJ, namely, lightweight family polymorphism provides better self typing for mutually recursive classes than Java generics. To support this claim rigorously, we show that $\text{FGJ}_{\text{self}}$ enjoys type soundness and the formal translation preserves typing and reduction.

## 1 INTRODUCTION

Simple inheritance with which C++ and Java (without generics) are equipped is not suitable for extending mutually recursive classes—their subclasses do not refer to each other but refer to the superclasses since it is not allowed to inherit superclass members with different signatures (in fact, it is not safe in general to allow covariant change of member signatures.) This "signature mismatching" problem is often resolved by typecasting, but it is a potentially unsafe operation. Ideally, extension of mutually recursive classes should yield another set of mutually recursive classes without losing type safety. There have been many proposals [3, 5, 6, 18, 19, 9, 20, 15, 11] to solve the above-mentioned problem. Ernst [11] has coined the term "family polymorphism" for a particular programming style using virtual classes [17] of `gbeta` [10] and applied it to the problem. The term "family" roughly means a set of mutually recursive classes, which are extensible without the mismatching.

We have proposed *lightweight* family polymorphism [22], a programming style in object-oriented programming to support reusable yet type-safe mutually recursive classes, as a solution. Our proposal is, as its name suggests, a lightweight version of family polymorphism. We identified a minimal set of language features for programming extensible mutually recursive classes in Java-like languages. We formalized the proposed features as .FJ, an extension of Featherweight Java [12], and proved the soundness of its type system.

Actually, however, it had been known that similar programming [5, 23] is possible in Java 5.0 and C# 2.0 with generics [2] and F-bounded polymorphism [7], although this programming requires a lot of boilerplate code for complicated parameterizations. It makes us wonder if our proposal is merely a convenient syntactic sugar for programming using Java-style generics—in other words, does our proposal have any essential advantages?

Our answer to the question is that the proposed features can be considered mostly as a syntactic sugar for the boilerplate code which would be required with Java generics, but lightweight family polymorphism has an essential advantage in *self typing* over Java generics—we can give more precise types to `this` (the self reference).

In this paper, we rigorously derive the answer above. Furthermore, we wish to propose a language feature for the target language to remedy the disadvantage in self typing. For these purposes, we give a formal translation, which changes *only* type declarations, from .FJ into an extension of Featherweight GJ [12] with *self type variables*, which we propose in this paper. The reason why we restrict the translation to change only type declarations—that is, no typecasts or executable code such as method declarations can be added—is that we aim to expose the similarity and difference of typing schema between the two languages. Self type variables, which allow the type of `this` to be a type variable, are essential for the translation to preserve typing. Without them, program fragments that have a certain form involving `this` would be ill typed after translation.[1] Therefore, we claim that the features in .FJ can be considered as a syntactic sugar and that self type variables are a language mechanism to bridge the essential gap between .FJ and Featherweight GJ in self typing. To support this claim rigorously, we show that Featherweight GJ with self type variables enjoys type soundness and the formal translation preserves typing and reduction.

We summarize our technical contributions as follows:

- an extension of Featherweight GJ with self type variables, called $\text{FGJ}_{\text{self}}$,
- a formal translation from .FJ into $\text{FGJ}_{\text{self}}$,
- a type soundness theorem of $\text{FGJ}_{\text{self}}$, and
- theorems of correctness of the formal translation.

Besides theoretical interest, the translation can be used for an implementation of lightweight family polymorphism, although there has been another possibility [22] using erasure [2]. The advantage of the present one is that the translation preserves the original type information without using typecasts.

**Rest of This Article.** Section 2 reviews lightweight family polymorphism and .FJ. Section 3 shows the outline of the translation and proposes self type variables.

[1]In fact, they could be well typed by using some workarounds [23, 5], but additional executable code would be inserted in the translation. See Section 5 for details.

Section 4 gives $\mathrm{FGJ_{self}}$ and the formal translation, and proves their properties. Section 5 discusses related work. Section 6 concludes.

This article is a revised version of the workshop paper [21]. We simplify the proof of subject reduction for $\mathrm{FGJ_{self}}$ (in Appendix C) by referring to Kamina and Tamai [16], who have proposed a mechanism similar to our self type variables.

## 2 LIGHTWEIGHT FAMILY POLYMORPHISM

In this section, we briefly review the key features supporting lightweight family polymorphism and its formal core calculus .FJ through an example of extensible mutually recursive classes. We would like to refer interested readers to the previous work [22] for more details.

### Programming in Lightweight Family Polymorphism.

Lightweight family polymorphism is realized by nested inheritance[2], relative path types and so on. These features are crucial to support type-safe and extensible mutually recursive classes so that extension of a family (set of mutually recursive classes) can yield another family.

Figure 1 shows the running example from [22]. This example features a family for graphs composed of nodes and edges, and its extending family for graphs in which each node has a color and each edge has a weight. The weight of an edge depends on the colors of the two nodes connected to the edge. Type declarations peculiar to lightweight family polymorphism are colored in red.

First, mutually recursive classes are declared as *nested classes* in a top-level class. In Figure 1, class `Graph` has two nested classes `Node` and `Edge`. Hereafter, we use the notion "family" to refer to nested classes and its enclosing (top-level) class. For example, family `Graph` consists of classes `Graph`, `Graph.Node` and `Graph.Edge`. (We use absolute nested class names[3], such as `Graph.Node`, to specify nested classes in a particular family.) This class nesting has a special inheritance mechanism. Mutually recursive classes in a family are extended *simultaneously* when their enclosing class is extended. Members in a nested class in a top-level class are inherited by a nested class of the same name in the derived top-level class. In Figure 1, class `ColorWeightGraph` is declared as an extension of class `Graph`. Therefore, classes `Node` and `Edge` in class `ColorWeightGraph` inherit members from classes `Node` and `Edge` in class `Graph`, respectively.

In nested classes, *relative path types* such as `.Node` and `.Edge` are used for mutual references instead of absolute class names. A relative path type refers to a nested class in the *same* top-level class and its meaning will change when it is

[2]Originally coined by Nystrom et al. [18].

[3]They are called fully qualified names in the Java terminology.

```
class Graph {
  class Node {
    Vector<.Edge> es=new Vector<.Edge>();
    void add(.Edge e){ es.add(e); }}
  class Edge {
    .Node src, dst;
    void connect(.Node s, .Node d){
      src = s; dst = d; s.add(this); d.add(this);
} } }
class ColorWeightGraph extends Graph {
  class Node { Color color; }
  class Edge {
    int weight;
    void connect(.Node s, .Node d){
      weight = colorToWeight(s.color, d.color);
      super.connect(s, d);
} } }
```

---

```
Graph.Edge e;              Graph.Node n1, n2;
ColorWeightGraph.Edge we;  ColorWeightGraph.Node cn1, cn2;
e.connect(n1, n2);        // 1: OK
we.connect(cn1, cn2);     // 2: OK
we.connect(n1, cn2);      // 3: compile-time error
e.connect(n1, cn2);       // 4: compile-time error
```

---

```
<G extends Graph>
  void connectAll(Vector<G.Edge> es, G.Node n1, G.Node n2){
    for (G.Edge e: es) e.connect(n1, n2); }

Vector<Graph.Edge> ges;             Graph.Node gn1, gn2;
Vector<ColorWeightGraph.Edge> ces;  ColorWeightGraph.Node cn1, cn2;
connectAll(ges, gn1, gn2);  // 5: OK (G as Graph)
connectAll(ces, cn1, cn2);  // 6: OK (G as ColorWeightGraph)
connectAll(ces, gn1, gn2);  // 7: compile-time error
```

Figure 1: Graph and ColorWeightGraph classes and a family-polymorphic method.

inherited to nested classes in a derived top-level class—relative path types in inherited members refer to one another in the derived family, as desired. For example,

.Edge in class Graph.Node refers to Graph.Edge. However, when it is inherited to ColorWeightGraph.Node, it will refer to ColorWeightGraph.Edge. The use of relative path types benefits the overriding connect() in ColorWeightGraph.Edge. The parameters s and d refer to ColorWeightGraph.Node since their types are given .Node. Thus, the field color can be accessed on them without typecasts. In summary, thanks to relative path types we can prevent mutual references from being hard-linked to classes in a particular family and can keep their mutual relationship through family extension. If we wrote the method signature by the absolute class names Graph.Node, typecasts would be necessary since the signature is not allowed to change covariantly when the method is overridden in subclasses for type safety. The languages without a mechanism like relative path types force us to do such possibly unsafe programming using typecasts.

*Absolute class names* such as Graph.Edge are used to declare variables or create objects outside families. When absolute class names are used, relative path types in the member signatures are *resolved* to be absolute with the family names. For example, in the middle of Figure 1 the variable e of type Graph.Edge has the method connect() with the parameter type Graph.Node, whereas the variable we of type ColorWeightGraph.Edge has the method connect() with ColorWeightGraph.Node. So, the method invocations 1 and 2 are well typed. The method invocation 3 is not allowed since n1 does not agree with the formal parameter type ColorWeightGraph.Node. A little surprisingly, the method invocation 4 is not allowed, either, since subtyping between nested classes is not allowed for safety, that is, ColorWeightGraph.Node is not substitutable for Graph.Node, even though there is an inheritance relation. Therefore, the objects from different families cannot be mixed.

*Family-polymorphic methods* overcome inconvenience associated with the absence of subtyping. They can work uniformly over different families and are realized as parametric methods which have family parameters with their upper bounds. An example is shown at the bottom of Figure 1. The family-polymorphic method connectAll() takes as input a vector of edges and two nodes of any family that extends family Graph such as Graph and ColorWeightGraph. The family names for es, n1 and n2 are given by the family parameter G, which is (implicitly) instantiated in order for method invocations to be well typed. For example, in 5 and 6, G is instantiated to Graph and ColorWeightGraph, respectively. However, there is no instantiation to make 7 well typed, so it results in a compile-time error.

The essence of lightweight family polymorphism is *self typing* for mutually recursive classes, as mentioned in Introduction. The self reference this of a nested class is given a relative path type since the meaning of this changes in subclasses. For example, this is of type .Edge in Graph.Edge, so it can be used as arguments to the method invocations s.add() and d.add() in connect(). (If this had type Graph.Edge in Graph.Edge, then these method invocations would be ill typed since there is no subtyping between Graph.Edge and .Edge in either direction. In fact, subtyping between them should not be allowed since the latter can change its meaning through family extension.) The translation in the next section will reveal that

$$
\begin{array}{rcll}
\texttt{F},\texttt{G} & ::= & \texttt{C} \mid \texttt{X} & \text{family names} \\
\texttt{A},\texttt{B} & ::= & \texttt{C} \mid \texttt{C.C} & \text{absolute class names} \\
\texttt{S},\texttt{T},\texttt{U} & ::= & \texttt{F} \mid \texttt{F.C} \mid \texttt{.C} & \text{types} \\
\texttt{L} & ::= & \texttt{class C} \triangleleft \texttt{C \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\ \overline{\texttt{NL}}\texttt{\}} & \text{top-level class declarations} \\
\texttt{M} & ::= & \texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{C}}\texttt{>T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{}\uparrow\texttt{e;\}} & \text{method declarations} \\
\texttt{NL} & ::= & \texttt{class C \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}} & \text{nested class declarations} \\
\texttt{d},\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new A(}\overline{\texttt{e}}\texttt{)} & \text{expressions} \\
\texttt{v} & ::= & \texttt{new A(}\overline{\texttt{v}}\texttt{)} & \text{values}
\end{array}
$$

Figure 2: .FJ: Syntax.

self typing is an essential difference from Java generics.

In summary, it is important to declare the signatures of members with relative path types for type-safe reuse and extension. If nested classes in a family type-check, it is always safe to resolve their member signatures with any family extending that family. In other words, members with relative path types are polymorphic over families.

## .FJ: A Formal Core Calculus of Lightweight Family Polymorphism

We formalized the features reviewed above as .FJ, an extension of Featherweight Java [12]. We defined its syntax, type system and operational semantics, and proved its properties. Here, we review only the syntax, judgments and properties. See Appendix A for the typing and reduction rules.

Figure 2 shows the .FJ syntax. The metavariables `C`, `D` and `E` range over (simple) class names; `X` and `Y` range over type variable names; `f` and `g` range over field names; `m` ranges over method names; `x` and `y` range over variables. (We choose the metavariables `F` and `G` for family names instead of `P` and `Q`, which are used in the original syntax [22], to avoid confusion since `P` and `Q` will be used in Featherweight GJ for different meanings. For the same reason, we choose `NL` for nested class declarations instead of `N`.) The symbols $\triangleleft$ and $\uparrow$ are read `extends` and `return`, respectively. Although constructor declarations are omitted for simplicity, we assume that every class has an obvious constructor that takes initial values of all the fields and assigns them. We put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}$" for "$\texttt{T}_1\ \texttt{f}_1\texttt{;}\ldots\texttt{T}_n\ \texttt{f}_n\texttt{;}$", where $n$ is the length of $\overline{\texttt{T}}$ and $\overline{\texttt{f}}$ and so on. Sequences of type variables, field declarations, parameter names, method declarations, and nested class declarations are assumed to contain no duplicate names. We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. Note that the invocation of a family-polymorphic method requires the actual arguments $\overline{\texttt{F}}$ in the formal language, although they are implicitly inferred in Figure 1. In fact, an inference algorithm for them has been developed [22]. A class table $CT$ is a mapping from absolute class names `A` to (top-level or nested) class declarations. A .FJ program is a pair $(CT$, `e`$)$ of a class table and an expression.

The main judgments of the .FJ type system are as follows: the subtyping judgment "$\Delta \vdash$ `S<:T`" is read "`S` is a subtype of `T` under the bound environment $\Delta$"; the type well-formedness judgment "$\Delta;$ `A` $\vdash$ `T` ok" is read "`T` is a well-formed type in (the body of) class `A` under the bound environment $\Delta$"; the typing judgment for expressions "$\Delta; \Gamma;$ `A` $\vdash$ `e:T`" is read "an expression `e` has a type `T` under the bound environment $\Delta$ and the type environment $\Gamma$ in the enclosing class `A`." Here, a type environment $\Gamma$ is a finite mapping from variables to types, written $\overline{\mathtt{x}}{:}\overline{\mathtt{T}}$; a bound environment $\Delta$ is a finite mapping from type variables to their bounds (top-level classes), written $\overline{\mathtt{X}}{<:}\overline{\mathtt{C}}$.

The operational semantics is defined by the reduction relation "$\mathtt{e} \longrightarrow \mathtt{e}'$", read "an expression `e` reduces to an expression $\mathtt{e}'$ in one step". We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The type system of .FJ has been proved sound with respect to the operational semantics, as the following theorems show.

**Theorem 2.1 (.FJ Subject Reduction)** *If* $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}{:}\mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}'{:}\mathtt{T}'$*, for some* $\mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}'{<:}\mathtt{T}$*.*

**Theorem 2.2 (.FJ Progress)** *If* $\emptyset; \emptyset; \mathtt{B} \vdash \mathtt{e}{:}\mathtt{A}$ *and* `e` *is not a value, then* $\mathtt{e} \longrightarrow \mathtt{e}'$*, for some* $\mathtt{e}'$*.*

**Theorem 2.3 (.FJ Type Soundness)** *If* $\emptyset; \emptyset; \mathtt{B} \vdash \mathtt{e} : \mathtt{A}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* `v` *with* $\emptyset; \emptyset; \mathtt{B} \vdash \mathtt{v} : \mathtt{A}'$ *and* $\emptyset \vdash \mathtt{A}' <: \mathtt{A}$*.*

## 3 AN OUTLINE OF THE TRANSLATION

In this section, we outline the translation from lightweight family polymorphism to Java generics. First, we describe the basic ideas of the translation with the example in Section 2. Although most parts of the example will be well typed after translation in this approach, we find that some program fragments involving `this` will *not* be well typed. To make *all* translations well typed, we then propose self type variables for Java generics. Recall that this translation changes only type declarations, but does not add any executable code.

### Basic Ideas of the Translation

The point of the translation is in how to simulate relative path types with Java generics. In particular, we need types that represent mutually recursive classes

```
class Graph { }
class Graph$Node<Node ◁ Graph$Node<Node,Edge>,
                 Edge ◁ Graph$Edge<Node,Edge>> {
  Vector<Edge> es=new Vector<Edge>();
  void add(Edge e){ es.add(e); }
}
class Graph$Edge<Node ◁ Graph$Node<Node,Edge>,
                 Edge ◁ Graph$Edge<Node,Edge>> {
  Node src, dst;
  void connect(Node s, Node d){
    src = s; dst = d; s.add(this); d.add(this);
} }
class ColorWeightGraph ◁ Graph { }
class ColorWeightGraph$Node<Node ◁ ColorWeightGraph$Node<Node,Edge>,
                            Edge ◁ ColorWeightGraph$Edge<Node,Edge>>
           ◁ Graph$Node<Node,Edge>{ Color color; }
class ColorWeightGraph$Edge<Node ◁ ColorWeightGraph$Node<Node,Edge>,
                            Edge ◁ ColorWeightGraph$Edge<Node,Edge>>
           ◁ Graph$Edge<Node,Edge>{
  int weight;
  void connect(Node s, Node d){
    weight = colorToWeight(s.color, d.color);  super.connect(s, d);
} }
```

```
class Graph$NodeFix ◁ Graph$Node<Graph$NodeFix, Graph$EdgeFix>{..}
class Graph$EdgeFix ◁ Graph$Edge<Graph$NodeFix, Graph$EdgeFix>{..}
```

```
<G$Node ◁ Graph$Node<G$Node,G$Edge>,
 G$Edge ◁ Graph$Edge<G$Node,G$Edge>, G ◁ Graph>
void connectAll(Vector<G$Edge> es, G$Node n1, G$Node n2){
  for (G$Edge e: es) e.connect(n1, n2); }
```

Figure 3: Translation of the classes and method in Figure 1

and that will change their meanings as classes are extended. Such types can be realized by type variables in parameterized classes, as pointed by Bruce et al. [5] and Torgersen [23]. We explain the basic ideas of the translation, by using the example in the last section.

The generic classes at the top of Figure 3 are the translation of the nested classes in Figure 1. Red parts show how types are translated. The classes `Node` and `Edge` in `Graph` are translated to top-level generic classes `Graph$Node` and `Graph$Edge`, in which the character `$` is used to make atomic names from qualified names. These classes are parameterized by type variables `Node` and `Edge`, which substitute for relative path types `.Node` and `.Edge`, respectively. To express extensible mutual recursion, these type variables are bounded by the classes in which they appear and the arguments to the bounds are the type variables themselves—in other words, type variables are F-bounded [7]. Then, a method invocation on such a type variable is given an expected type: for example, the method invocation of `s.add()` in `connect()` of `Graph$Edge` has the signature `Edge`$\longrightarrow$`void`, in which the type variable `Edge` represents the relative path type `.Edge`.

In the translation of a subfamily, the inheritance relation is made explicit: for example, `ColorWeightGraph$Node<Node,Edge>` extends `Graph$Node<Node,Edge>`. However, the upper bounds of type variables are refined covariantly. So, the derived generic classes inherit the members with the same signatures but the type variables in them have refined upper bounds. As a result, for example, it is legal to access the fields `s.color` and `d.color` in the overriding `connect()` in `ColorWeightGraph$Edge`, without using typecasts, since `s` and `d` are of type `Node` whose upper bound is `ColorWeightGraph$Node`, which has `color`. Although classes are no longer nested, generic classes from one family will always work together due to the given F-bounded constraints.

*Fixed point classes*, non-generic subclasses of these generic classes, have to be declared for object creation—the generic classes cannot be instantiated since their type parameters cannot be instantiated with any types that can be composed from existing class definitions. Fixed point classes correspond to absolute class names in lightweight family polymorphism. We make their names by adding a suffix `Fix` to the generic class names. A fixed point class `Graph$NodeFix` extends `Graph$Node` and instantiates the type variables with itself and another fixed point class `Graph$EdgeFix`, defined in the middle of Figure 3. (The omitted class bodies contain only constructors.) Type variables in types of members inherited from a generic class are instantiated with the names of the fixed point classes, just as relative path types are resolved to be absolute when an absolute path type is used. The fixed point classes `ColorWeightGraph$NodeFix` and `ColorWeightGraph$EdgeFix` for the derived family `ColorWeightGraph` are defined similarly. Note that fixed point classes in a subfamily are not substitutable for ones in its super family since they are not in inheritance relations. For example, `ColorWeightGraph$NodeFix` $\not<:$ `Graph$NodeFix`. This fact corresponds to that in lightweight family polymorphism there is no subtyping between nested classes.

To translate a family-polymorphic method, which has a family parameter `X` whose upper bound is `C`, we have to simulate types of form `X.D`. We need types that can represent the generic classes translated from the upper bound family `C` and its subfamilies. Such types can be realized by parameterizing the method with all generic classes in the translation of family `C`. Each type `X.D` is translated to a type variable X$D, which is F-bounded. For example, the translation of connectAll() in Figure 1 is found at the bottom of Figure 3. Types G.Node and G.Edge translate to G$Node and G$Edge, respectively. The type variables and their upper bounds that the translation introduces are colored in red. A careful reader may notice that the type variable G has no connection with G$Node and G$Edge. Nevertheless, the method body is well typed (and would be, even when it took an argument of type G). It is because relative path types in .FJ can refer only to sibling classes and cannot to, say, the enclosing class as is possible in other languages [14, 9]. So, it is sufficient to represent the connection among nested classes by F-bounded constraints. Note that X$D must be introduced to the parameterization clause for *all* nested classes in the upper bound of X even if some of them do not appear in the method signature. For example, even if G.Edge did not appear in the signature of connectAll(), the translated connectAll() would have the same parameterization as that in the figure since G$Node requires G$Edge to appear in its upper bound.

The actual type argument to a method invocation is translated similarly: if it is a top-level class name, it will translate to a sequence of the names of the fixed point classes generated from this top-level class, followed by the top-level class name itself; if it is a type variable X, it will translate to a sequence of type variables of form X$C, followed by the type variable itself.

## Self Type Variables

Most parts of the program are well typed after the translation described above. Unfortunately, however, program fragments that have a certain form will not be well typed since how this is typed in generic classes is different from that in nested classes in lightweight family polymorphism. In this subsection, we examine this problem and propose self type variables for Java generics to improve its self typing.

This problem occurs in the translation of a method invocation in which this is passed to the parameters of relative path types as the arguments. For instance, s.add(this) and d.add(this) in connect() of Graph$Edge in Figure 3 are, actually, not well typed, because this has type Graph$Edge<Node,Edge> with the typing rules of Java generics, but it does not agree with the parameter type Edge, which is a type variable. However, the type of this should translate to a type variable so that such translated method invocations will be well typed since this is given a relative path type.

In this paper, we propose self type variables to modify self typing of Java generics as a solution to the problem. In the proposal, on the one hand, we allow the type of this in a generic class to be a type variable, chosen from the F-bounded constraints, just as the types of other mutual references. On the other hand, we limit subclassing and instantiation of such a class for safety.

More precisely, if the upper bound of a type variable is the same as the name of the class, the type variable can be given as the type for this. We call such a type variable *a self type variable* since the self type (the type for `this`) is represented by a type variable. For example, `Edge` is the self type variable in `Graph$Edge` since, as the following code shows, the class name (colored in magenta) and the upper bound (in green) of `Edge` are the same `Graph$Edge<Node,Edge>`:

```
class Graph$Edge<Node ◁ Graph$Node<Node,Edge>,
                 Edge ◁ Graph$Edge<Node,Edge>> { ..
  void connect(Node s, Node d){
    ... s.add(this); d.add(this); // well typed
} }
```

As a result, `s.add(this)` and `d.add(this)` in `connect()` become well typed.

For type safety, we limit subclassing and instantiation of such a generic class that has a self type variable. Otherwise, type safety would be lost. For example, consider the following class declaration:

```
class Graph$EdgeFake ◁ Graph$Edge<Graph$NodeFix, Graph$EdgeFix>{..}
```

which at first looks fine because both arguments to the superclass `Graph$Edge` satisfy the constraints. However, if this class were allowed, invoking method `connect()` on an object of the class would result in an ill-typed expression. For example, assume that `fake` has type `Graph$EdgeFake`, and `n1` and `n2` have type `Graph$NodeFix`. Then, the method invocation `fake.connect(n1, n2)` would produce the method invocations `n1.add(fake)` and `n2.add(fake)`, which are ill typed since the argument type `Graph$EdgeFake` is incompatible with the parameter type `Graph$EdgeFix`. (Note that `Graph$EdgeFake` is not a subtype of `Graph$EdgeFix` since they are not in inheritance relation.)

There are two cases when subclassing of a class that has a self type variable is allowed. The first case is that the subclass is a fixed point class such as `Graph$EdgeFix`—the extending class instantiates the self type variable of the extended class with itself. The second case is that the subclass has a self type variable and instantiates the self type variable of the superclass with it as `ColorWeightGraph$Node` does. In other words, the self type variable is inherited. Instantiation of a class that has a self type variable is not allowed regardless of type instantiation. For example, `Graph$Edge<Graph$NodeFix,Graph$EdgeFix>` cannot be instantiated since `Graph$Edge` has self type variable `Edge`.

Subclassing and instantiation of a class that does not have a self type variable are always allowed as before.

Self type variables offer more flexible self typing than Java generics; even more flexible self typing is supported in Scala by self type annotations [20]. Although self types can be arbitrary in Scala, type safety is still guaranteed because of two

$$
\begin{array}{rcll}
\mathtt{S},\mathtt{T},\mathtt{U} & ::= & \mathtt{X} \mid \mathtt{N} & \text{types} \\
\mathtt{N},\mathtt{P},\mathtt{Q} & ::= & \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} & \text{non-variable types} \\
\mathtt{L} & ::= & \mathtt{class}\ i_{opt}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\texttt{>}\triangleleft\mathtt{N}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} & \text{class declarations} \\
\mathtt{M} & ::= & \texttt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\texttt{>}\mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\uparrow\mathtt{e};\ \} & \text{method declarations} \\
\mathtt{d},\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.}\texttt{<}\overline{\mathtt{T}}\texttt{>}\mathtt{m}(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) & \text{expressions} \\
\mathtt{v} & ::= & \mathtt{new}\ \mathtt{N}(\overline{\mathtt{v}}) & \text{values}
\end{array}
$$

Figure 4: FGJ$_{\text{self}}$: Syntax.

requirements: (1) the self type of a class must be a subtype of the self type of its superclass, (2) a class being instantiated in a `new` expression must be a subtype of the self type of the class. Our rules described above correspond to these requirements.

## 4 FORMALIZATION

In this section, we formalize self type variables as FGJ$_{\text{self}}$ based on Featherweight GJ (FGJ) [12] and prove the soundness of its type system. Then, we formalize the translation from .FJ into FGJ$_{\text{self}}$. Finally, we prove that the translation is correct with respect to typing and reduction.

### FGJ$_{\text{self}}$: An Extension of Featherweight GJ with Self Type Variables

We give the formal definition of FGJ$_{\text{self}}$. The differences from that of FGJ are found in the syntax and type system, but not in the operational semantics.

#### Syntax

Figure 4 shows the syntax of FGJ$_{\text{self}}$. The difference from the FGJ syntax is found in a class declaration, in which an optional number $i$ is introduced after keyword `class`. This $i$ is used to indicate that the $i$-th type variable of a generic class is the self type variable. If $i$ does not appear in a class declaration, we consider the class to be a usual class, which does not have a self type variable. Our formalization requires self type variables to be explicitly specified as opposed to the example in Section 3. Formalizing an inference rule for $i$ would be easy as done in Kamina and Tamai [16]. There are no other differences from the FGJ syntax except that typecasts are omitted for simplicity. Note that the meaning of the metavariables is the same as that in the .FJ syntax. The new metavariables `N`, `P` and `Q` are introduced to range over non-variable types.

A class table $CT$ is a mapping from class names `C` to class declarations `L`. An FGJ$_{\text{self}}$ program is a pair ($CT$, `e`). In what follows, we assume a *fixed* class table $CT$ to simplify the notation.

The following auxiliary functions are used as in FGJ [12]. A function $\mathit{bound}_\Delta(\mathtt{T})$ returns the upper bound of type $\mathtt{T}$. A lookup function $\mathit{fields}(\mathtt{N})$ returns a sequence $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ of field names of class $\mathtt{N}$ with their types. A lookup function $\mathit{mtype}(\mathtt{m}, \mathtt{N})$ returns the signature, written $\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}$, of given method and class names. A lookup function $\mathit{mbody}(\mathtt{m<}\overline{\mathtt{U}}\mathtt{>}, \mathtt{N})$ returns the method body with formal parameters, written $\overline{\mathtt{x}}\mathtt{.e}$, of given method and class names. They are defined essentially the same as those of FGJ; we refer readers to Figure 10 in Appendix B for their definitions.

## Type System

A type environment $\Gamma$ is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$. A bound environment $\Delta$ is a finite mapping from type variables to non-variable types, written $\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}$. Application of type substitution $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$ is defined in the customary manner. The main judgments of FGJ$_{\text{self}}$ consist of one for subtyping "$\Delta \vdash \mathtt{S<:T}$", ones for well-formedness (mentioned later), one for expression typing "$\Delta;\Gamma \vdash \mathtt{e:T}$", one for method typing "$\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{M}$ ok", and one for class typing "$\vdash \mathtt{L}$ ok". Although FGJ has a single well-formedness judgment "$\Delta \vdash \mathtt{T}$ ok" for all types, we distinguish the well-formedness judgments for upper bounds "$\Delta; \mathtt{X} \vdash \mathtt{N}$ ok-bound" and for superclasses "$\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{N}$ ok-superclass" from "$\Delta \vdash \mathtt{T}$ ok" for other types, namely the field types, the return and parameter types of methods, the type arguments to method invocations, and the class names for object creations. The judgment for correct class instantiation "$\Delta \vdash \mathtt{N}$ ok-inst" is given to avoid defining the duplicate rules for the three well-formedness judgments above. We abbreviate a sequence of judgments in the following way: $\Delta \vdash \mathtt{T}_1$ ok, ..., $\Delta \vdash \mathtt{T}_n$ ok to $\Delta \vdash \overline{\mathtt{T}}$ ok; $\Delta; \mathtt{X}_1 \vdash \mathtt{N}_1$ ok-bound, ..., $\Delta; \mathtt{X}_n \vdash \mathtt{N}_n$ ok-bound to $\Delta; \overline{\mathtt{X}} \vdash \overline{\mathtt{N}}$ ok-bound; $\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{M}_1$ ok, ..., $\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{M}_n$ ok to $\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>} \vdash \overline{\mathtt{M}}$ ok. The rules for subtyping relation and expression typing, which are essentially the same as those of FGJ [12], are shown in Appendix B.

Figure 5 shows well-formedness rules and typing rules for methods and classes. In the figure, we use an auxiliary function $\mathit{selftype}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})$, which returns the self type of non-variable type $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$, defined as follows:

$$\begin{array}{llll}
\mathit{selftype}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} & \mathit{if} & \mathtt{class} & \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{..\}} \\
\mathit{selftype}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{T}_i & \mathit{if} & \mathtt{class}\ i & \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{..\}}
\end{array}$$

**Subtyping.** The subtyping relation is the reflexive and transitive closure of the inheritance relation $\triangleleft$ (`extends`), as in FGJ [12].

**Well-formedness.** The well-formedness rules for types, upper bounds and superclasses are similar in that `Object` is always a well-formed type, upper bound and superclass. However, their difference can be found in the case of class types $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$. All rules require that the class type is correctly instantiated, derived by (WF-Inst), but each has a characteristic premise about self type variables.

**Correct Class Instantiation:**

$$\frac{\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\ \lhd\ \mathtt{N}\ \mathtt{\{...\}} \qquad \Delta\vdash\overline{\mathtt{T}}\ \text{ok} \qquad \Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}}{\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-inst}}\ \text{(WF-Inst)}$$

**Well-formed Types:**

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

$$\frac{\mathtt{X}\in dom(\Delta)}{\Delta\vdash\mathtt{X}\ \text{ok}}\ \text{(WFT-Var)}$$

$$\frac{selftype(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-inst}}{\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}\ \text{(WFT-Class)}$$

**Well-formed Upper Bounds:**

$$\Delta;\mathtt{X}\vdash\mathtt{Object}\ \text{ok-bound}\qquad\text{(WFB-Object)}$$

$$\frac{selftype(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\mathtt{T}_i=\mathtt{X} \qquad \Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-inst}}{\Delta;\mathtt{X}\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-bound}}\ \text{(WFB-ClassSelf)}$$

$$\frac{selftype(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-inst}}{\Delta;\mathtt{X}\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-bound}}\ \text{(WFB-Class)}$$

**Well-formed Superclasses:**

$$\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{Object}\ \text{ok-superclass}\qquad\text{(WFS-Object)}$$

$$\frac{selftype(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>})=\mathtt{S}_i=\mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \qquad \overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-inst}}{\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-superclass}}\ \text{(WFS-ClassFix)}$$

$$\frac{selftype(\mathtt{C<}\overline{\mathtt{X}}\mathtt{>})=\mathtt{X}_i=selftype(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}) \qquad \overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-inst}}{\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-superclass}}\ \text{(WFS-ClassSelf)}$$

$$\frac{selftype(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>})=\mathtt{D<}\overline{\mathtt{S}}\mathtt{>} \qquad \overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-inst}}{\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-superclass}}\ \text{(WFS-Class)}$$

**Method Typing:**

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}} \qquad \Delta\vdash\overline{\mathtt{T}},\mathtt{T}\ \text{ok}\\ \Delta;\overline{\mathtt{Y}}\vdash\overline{\mathtt{P}}\ \text{ok-bound}\\ \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}},\mathtt{this}:selftype(\mathtt{C<}\overline{\mathtt{X}}\mathtt{>})\vdash\mathtt{e}_0:\mathtt{S}\\ \Delta\vdash\mathtt{S}<:\mathtt{T} \qquad \mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\lhd\mathtt{N\{..\}}\\ override(\mathtt{m},\mathtt{N},\mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T})\end{array}}{\mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e}_0\mathtt{;\}}\ \text{ok}}\ \text{(GT-Method)}$$

**Class Typing:**

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \qquad \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{N}\ \text{ok-superclass}\\ \Delta\vdash\overline{\mathtt{T}}\ \text{ok} \qquad \Delta;\overline{\mathtt{X}}\vdash\overline{\mathtt{N}}\ \text{ok-bound}\\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{>}=\mathtt{N}_i \qquad \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\overline{\mathtt{M}}\ \text{ok}\end{array}}{\vdash\mathtt{class}\ \ i\ \ \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\lhd\mathtt{N\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}\ \text{ok}}\ \text{(GT-ClassSelf)}$$

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \qquad \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\mathtt{N}\ \text{ok-superclass}\\ \Delta\vdash\overline{\mathtt{T}}\ \text{ok} \qquad \Delta;\overline{\mathtt{X}}\vdash\overline{\mathtt{N}}\ \text{ok-bound}\\ \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\vdash\overline{\mathtt{M}}\ \text{ok}\end{array}}{\vdash\mathtt{class}\ \ \mathtt{C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\lhd\mathtt{N\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}\ \text{ok}}\ \text{(GT-Class)}$$

Figure 5: FGJ$_{\text{self}}$: Well-formedness rules and typing rules for methods and classes.

A class type $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ is a well-formed type if class $\mathtt{C}$ does not have a self type variable (WFT-Class). The upper bound $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ of a type variable $\mathtt{X}$, where $\mathtt{T}_i$ is the self type of $\texttt{C<}\overline{\texttt{T}}\texttt{>}$, is well-formed under bound environment $\Delta$ if $\texttt{X}$ is $\texttt{T}_i$ (WFB-ClassSelf). That is, $\texttt{X}$ is F-bounded by $\texttt{C<}\overline{\texttt{T}}\texttt{>}$. There are two cases (WFS-ClassFix, WFS-ClassSelf) for a superclass $\texttt{D<}\overline{\texttt{S}}\texttt{>}$ of $\texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}$, where $\texttt{S}_i$ is the self type of $\texttt{D<}\overline{\texttt{S}}\texttt{>}$, to be well formed. The former is when $\texttt{S}_i$ is the extending class $\texttt{C<}\overline{\texttt{X}}\texttt{>}$, meaning that $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ is a fixed point class of $\texttt{D<}\overline{\texttt{S}}\texttt{>}$. The latter is when $\texttt{S}_i$ is the self type variable $\texttt{X}_j$ of $\texttt{C<}\overline{\texttt{X}}\texttt{>}$, meaning that the self type variable is inherited correctly.

**Expression Typing.** As mentioned before, the typing rules for expressions are the same as those of FGJ [12]. See Figure 10 in Appendix B for their definitions. Since the rules for well-formed types are slightly changed, it may be worthwhile noting that the rule (GT-New) for object creations $\texttt{new N(}\overline{\texttt{e}}\texttt{)}$ requires that $\texttt{N}$ is a well-formed type ($\Delta \vdash \texttt{N}$ ok), meaning that classes that have self type variables cannot be instantiated as mentioned in Section 3.

**Method Typing.** The judgment "$\texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>} \vdash \texttt{M}$ ok" is read "a method declaration $\texttt{M}$ in $\texttt{class}$ $i_{opt}$ $\texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N\{..\}}$ is *ok*." The rule (GT-Method) checks that the method body is well typed under the bound environment and type environment in which the type of $\texttt{this}$ is *selftype*($\texttt{C<}\overline{\texttt{X}}\texttt{>}$). The premise using *override* checks that the method correctly overrides (if it does) the method of the same name in the superclass with the same signature (modulo renaming of type variables) in which covariant overriding of the return type is allowed.

**Class Typing.** The judgment "$\vdash \texttt{L}$ ok" is read "a class declaration $\texttt{L}$ is *ok*." There are two rules, one for classes with self type variables (GT-ClassSelf) and one for classes without them (GT-Class). Both rules require that superclasses are *ok*. In the rule GT-ClassSelf, where the class declaration is $\texttt{class}\ i\ \texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\texttt{\}}$, the self type variable is $\texttt{X}_i$ as indicated by $i$. The rule checks that $\texttt{X}_i$ is a true self type variable for class $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ i.e., the upper bound $\texttt{N}_i$ of $\texttt{X}_i$ equals the class name $\texttt{C<}\overline{\texttt{X}}\texttt{>}$. For example, this typing rule can be applied to class declaration of $\texttt{Graph\$Node}$, with now the number $\texttt{1}$ after $\texttt{class}$, in Section 3:

$$\dfrac{\begin{array}{c}\ldots\\ \texttt{Graph\$Node<Node,Edge>} = \texttt{Graph\$Node<Node,Edge>}\end{array}}{\begin{array}{l}\vdash \texttt{class 1 Graph\$Node<Node} \ \triangleleft\ \texttt{Graph\$Node<Node,Edge>,}\\ \qquad\qquad\qquad\qquad\qquad\ \ \texttt{Edge}\ \triangleleft\ \texttt{Graph\$Edge<Node,Edge>> \{ .. \}}\ \text{ok}\end{array}}$$

The equality between the class name (colored in magenta) and the upper bound (in green) of the *first* type variable $\texttt{Node}$ means that $\texttt{Node}$ is truly the self type variable of class $\texttt{Graph\$Node}$.

A class table $CT$ is *ok* if all its definitions are ok.

$$\frac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i} \qquad \text{(GR-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{U}}\mathtt{>}, \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).<}\overline{\mathtt{U}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0} \qquad \text{(GR-INVK)}$$

Figure 6: $\text{FGJ}_{\text{self}}$: Reduction.

### Operational Semantics

The operational semantics is given by the reduction relation $\mathtt{e}\longrightarrow\mathtt{e}'$. The rules are the same as those of FGJ [12], defined in Figure 6. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1$, ..., $\mathtt{x}_n$ with $\mathtt{d}_n$, and $\mathtt{y}$ with $\mathtt{e}$. Two reduction rules, one for field access and one for method invocation, are defined straightforwardly. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), omitted here. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

### Type Soundness

The type system of $\text{FGJ}_{\text{self}}$ is sound with respect to the operational semantics. Type soundness is proved in the standard manner via subject reduction and progress [26, 12].

**Theorem 4.1 (Subject Reduction)** *If $\Delta;\Gamma \vdash \mathtt{e}:\mathtt{T}$ and $\mathtt{e} \longrightarrow \mathtt{e}'$, then $\Delta \vdash \mathtt{T}' \mathtt{<:} \mathtt{T}$, for some $\mathtt{T}'$ such that $\Delta;\Gamma \vdash \mathtt{e}':\mathtt{T}'$.*

*Proof.* See Appendix C. □

**Theorem 4.2 (Progress)** *If $\emptyset;\emptyset \vdash \mathtt{e}:\mathtt{T}$ and $\mathtt{e}$ is not a value, then $\mathtt{e}\longrightarrow\mathtt{e}'$, for some $\mathtt{e}'$.*

*Proof.* Similar to that for FGJ [12]. □

**Theorem 4.3 (Type Soundness)** *If $\emptyset;\emptyset \vdash \mathtt{e}:\mathtt{T}$ and $\mathtt{e}\longrightarrow^*\mathtt{e}'$ with $\mathtt{e}'$ a normal form, then $\mathtt{e}'$ is a value $\mathtt{v}$ with $\emptyset;\emptyset \vdash \mathtt{v}:\mathtt{T}'$ and $\emptyset \vdash \mathtt{T}'\mathtt{<:}\mathtt{T}$.*

*Proof.* Immediate from Theorems 4.1 and 4.2. □

## A Formal Translation from .FJ into $\text{FGJ}_{\text{self}}$

Figure 7 shows the definition of the formal translation. As mentioned before, this translation changes only type declarations. The translation of sequences is abbreviated in a straightforward way. For example, "$|\mathtt{F}.\overline{\mathtt{E}}|$" stands for "$|\mathtt{F}.\mathtt{E}_1|, \ldots, |\mathtt{F}.\mathtt{E}_n|$", where $n$ is the length of $\overline{\mathtt{E}}$; "$|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}$" for "$|\mathtt{F}_1|_{\mathtt{C}_1}, \ldots, |\mathtt{F}_n|_{\mathtt{C}_n}$", and so on.

**Translation of Types:**

$$\begin{array}{lcl@{\qquad}lcl}
|\mathtt{C}| & = & \mathtt{C} & |\mathtt{C.E}| & = & \mathtt{C\$EFix} \\
|\mathtt{X}| & = & \mathtt{X} & |\mathtt{X.E}| & = & \mathtt{X\$E} \\
|\mathtt{.E}| & = & \mathtt{E} & & &
\end{array}$$

**Definition of *classes*:**

$$classes(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C}\triangleleft\mathtt{D\{\ ..\ \ \overline{NL}\ \}} \qquad classes(\mathtt{D}) = \overline{\mathtt{E}}'}{classes(\mathtt{C}) = \overline{\mathtt{E}}', \{\mathtt{E} \mid \mathtt{E} \notin \overline{\mathtt{E}}', \mathtt{class\ E\{..\}} \in \overline{\mathtt{NL}}\}}$$

**Translation of Type Arguments:**

$$\frac{classes(\mathtt{C}) = \overline{\mathtt{E}}}{|\mathtt{F}|_{\mathtt{C}} = |\mathtt{F}.\overline{\mathtt{E}}|, |\mathtt{F}|} \quad (\textsc{Tr-Arg})$$

**Translation of Expressions:**

$$|\mathtt{x}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{x} \qquad (\textsc{Tr-Var})$$

$$|\mathtt{e}_0\mathtt{.f}_i|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}\mathtt{.f}_i \qquad (\textsc{Tr-Field})$$

$$\frac{\Delta;\Gamma;\mathtt{A} \vdash_{.\mathrm{FJ}} \mathtt{e}_0\mathtt{:T}_0 \qquad mtype_{.\mathrm{FJ}}(\mathtt{m}, bound_\Delta(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0}{|\mathtt{e}_0\mathtt{.<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}\mathtt{.<}|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}\mathtt{>m(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}} \qquad (\textsc{Tr-Invk})$$

$$|\mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)} \qquad (\textsc{Tr-New})$$

**Definition of Ceiling:**

$$\lceil\mathtt{C.E}\rceil = \begin{cases} \mathtt{C\$E} & \text{if } \mathtt{class\ E\{..\}} \in \overline{\mathtt{NL}} \\ \lceil\mathtt{D.E}\rceil & \text{otherwise} \end{cases}$$

where $\mathtt{class\ C}\triangleleft\mathtt{D\{..}\overline{\mathtt{NL}}\mathtt{\}}$

**Translation of Methods:**

$$\frac{classes(\mathtt{C}) = \overline{\mathtt{E}}}{|\mathtt{X}\triangleleft\mathtt{C}| = |\mathtt{X.E}_1|\triangleleft\lceil\mathtt{C.E}_1\rceil\mathtt{<}|\mathtt{X}.\overline{\mathtt{E}}|\mathtt{>}, \ldots, |\mathtt{X.E}_n|\triangleleft\lceil\mathtt{C.E}_n\rceil\mathtt{<}|\mathtt{X}.\overline{\mathtt{E}}|\mathtt{>}, \mathtt{X}\triangleleft\mathtt{C}} \qquad (\textsc{Tr-ParaMethod})$$

$$\frac{\Gamma = \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}, \mathtt{this:}thistype(\mathtt{A}) \qquad \Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}}{|\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e}_0\mathtt{;\ \}}|_{\mathtt{A}} = \mathtt{<}|\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}|\mathtt{>}|\mathtt{T}_0|\ \mathtt{m(}|\overline{\mathtt{T}}|\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}\mathtt{;\}}} \qquad (\textsc{Tr-Method})$$

**Translation of Classes:**

$$\frac{classes(\mathtt{C}) = \overline{\mathtt{E}}}{|\triangleleft\mathtt{C}| = |\mathtt{.E}_1|\triangleleft\lceil\mathtt{C.E}_1\rceil\mathtt{<}|.\overline{\mathtt{E}}|\mathtt{>}, \ldots, |\mathtt{.E}_n|\triangleleft\lceil\mathtt{C.E}_n\rceil\mathtt{<}|.\overline{\mathtt{E}}|\mathtt{>}} \qquad (\textsc{Tr-ParaClass})$$

$$\frac{\mathtt{class\ C}\triangleleft\mathtt{D\{..\}} \quad classes(\mathtt{C}) = \overline{\mathtt{E}} \quad classes(\mathtt{D}) = \overline{\mathtt{E}}' \quad \mathtt{E}_i \in \overline{\mathtt{E}}'}{|\mathtt{class\ E}_i\ \mathtt{\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}|_{\mathtt{C}} = \mathtt{class}\ i\ \mathtt{C\$E}_i\mathtt{<}|\triangleleft\mathtt{C}|\mathtt{>}\triangleleft\lceil\mathtt{D.E}_i\rceil\mathtt{<}|.\overline{\mathtt{E}}'|\mathtt{>\{}\ |\overline{\mathtt{T}}|\ \overline{\mathtt{f}}\mathtt{;}|\overline{\mathtt{M}}|_{\mathtt{C.E}_i}\mathtt{\}}} \qquad (\textsc{Tr-NClass1})$$

$$\frac{\mathtt{class\ C}\triangleleft\mathtt{D\{..\}} \quad classes(\mathtt{C}) = \overline{\mathtt{E}} \quad classes(\mathtt{D}) = \overline{\mathtt{E}}' \quad \mathtt{E}_i \notin \overline{\mathtt{E}}'}{|\mathtt{class\ E}_i\ \mathtt{\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}|_{\mathtt{C}} = \mathtt{class}\ i\ \mathtt{C\$E}_i\mathtt{<}|\triangleleft\mathtt{C}|\mathtt{>}\triangleleft\mathtt{Object\{}\ |\overline{\mathtt{T}}|\ \overline{\mathtt{f}}\mathtt{;}|\overline{\mathtt{M}}|_{\mathtt{C.E}_i}\mathtt{\}}} \qquad (\textsc{Tr-NClass2})$$

$$\frac{classes(\mathtt{C}) = \overline{\mathtt{E}}}{fix(\mathtt{C}) = \mathtt{class}\ |\mathtt{C.E}_1|\triangleleft\lceil\mathtt{C.E}_1\rceil\mathtt{<}|\mathtt{C}.\overline{\mathtt{E}}|\mathtt{>\{\}}\ \ldots\ \mathtt{class}\ |\mathtt{C.E}_n|\triangleleft\lceil\mathtt{C.E}_n\rceil\mathtt{<}|\mathtt{C}.\overline{\mathtt{E}}|\mathtt{>\{\}}} \qquad (\textsc{Fix})$$

$$|\mathtt{class\ C}\triangleleft\mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \overline{\mathtt{NL}}\mathtt{\}}| = \mathtt{class\ C}\triangleleft\mathtt{D}\ \mathtt{\{}|\overline{\mathtt{T}}|\ \overline{\mathtt{f}}\mathtt{;}|\overline{\mathtt{M}}|_{\mathtt{C}}\mathtt{\}}\ |\overline{\mathtt{NL}}|_{\mathtt{C}}\ fix(\mathtt{C}) \qquad (\textsc{Tr-TClass})$$

Figure 7: Translation of types, expressions, methods and classes.

### Translation of Types

Translation $|\mathtt{T}|$ of type `T` is defined at the top of the left column. Note that the two rules in the first row show that the result is a class name, whereas the other three rules show that the result is a type variable. Recall that `$` is a character used to make an atomic name.

### Translation of Expressions

We define an auxiliary function $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$ to collect all names $\overline{\mathtt{E}}$ of nested classes in a family `C` including those of inherited ones.

Translation of an expression `e` requires bound environment $\Delta$, type environment $\Gamma$ and enclosing class `A` as auxiliary information and is written $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}}$. The translation of a variable reference and field access is straightforward. The translation of an object creation requires the translation of the class name.

Translation of a method invocation $\mathtt{e}_0\mathtt{.<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}$ requires the translations $|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}$ of the type arguments $\overline{\mathtt{F}}$ besides the translations of subexpressions $\mathtt{e}_0$ and $\overline{\mathtt{e}}$. As mentioned in the previous section, the translation $|\mathtt{F}_i|_{\mathtt{C}_i}$ of each type argument $\mathtt{F}_i$ with respect to a top-level class $\mathtt{C}_i$ is a sequence of fixed point classes if $\mathtt{F}_i$ is a top-level class or one of types of form `X$D` if $\mathtt{F}_i$ is a type variable. Both cases require the nested class names in class $\mathtt{C}_i$ to make the sequence. Here, class $\mathtt{C}_i$ is the upper bound of $\mathtt{X}_i$, the formal parameter corresponding to the argument $\mathtt{F}_i$, and is obtained from the signature of `m()` in .FJ by using the lookup function $\mathit{mtype}_{\text{.FJ}}$ on the type $\mathtt{T}_0$ of the receiver $\mathtt{e}_0$. ($\mathit{bound}_\Delta(\mathtt{T}_0@\mathtt{A})$ stands for the class name to look up. When $\mathtt{T}_0$ is a relative path type, it is resolved with `A` (by $\mathtt{T}_0@\mathtt{A}$) and, when $\mathtt{T}_0$ contains a type variable, the type variable is replaced with its upper bound, to obtain an absolute class name. See Appendix A for their definitions.) The sequence is derived by the rule (TR-ARG), in which *classes* is used. For example, $|\mathtt{CWGraph}|_{\mathtt{Graph}}$ is a sequence of `CWGraph$NodeFix`, `CWGraph$EdgeFix`, `CWGraph`. (`CWGraph` stands for `ColorWeightGraph`.)

### Ceiling of Absolute Class Names

The translation of methods and classes is a little involved since there can exist nested classes that do not appear explicitly as class declarations but it is legal in .FJ to mention an absolute type corresponding to such implicit classes. This situation arises when we do not redefine a nested class in a subfamily. The type system will assume that the subfamily has an empty nested class, which lacks field and method declarations. For example, consider the following code:

```
class C     { class E1 {...}   class E2 {...} }
class D ◁ C { class E1 {...} }
```

Class E2 is not redefined in class D, but we can create the object of type D.E2.

Such implicit classes raise a problem in the translation. If we simply applied the translation strategy in Section 3 to the code above, the translation would be:

```
class 1 C$E1<E1 ◁ C$E1<E1,E2>, E2 ◁ C$E2<E1,E2>>{...}
class 2 C$E2<E1 ◁ C$E1<E1,E2>, E2 ◁ C$E2<E1,E2>>{...}
class 1 D$E1<E1 ◁ D$E1<E1,E2>, E2 ◁ D$E2<E1,E2>> ◁ C$E1<E1,E2> {...}
```

Note that the F-bounded constraints of class D$E1<E1,E2> requires the presence of class D$E2<E1,E2>, which is *absent*, however. So, class D$E1<E1,E2> is not a legal declaration, meaning that the translation fails.

One way to deal with such references to missing classes is to generate empty generic classes corresponding to the implicit classes. In this approach, an empty class D$E2<E1,E2> would be added to the translation above.

We instead take another approach to save such generation. We replace the missing class names caused by implicit classes with the names of the generic classes translated from the nearest explicit superclasses of the implicit classes. For example, in this approach, the class declaration for D$E1<E1,E2> will be:

```
class 1 D$E1<E1 ◁ D$E1<E1,E2>, E2 ◁ C$E2<E1,E2>> ◁ C$E1<E1,E2> {...}
```

Note that the upper bound of E2 is here C$E2<E1,E2> (colored in red above) instead of D$E2<E1,E2>. We introduce the notion of the *ceiling* of an absolute class name to refer to such a nearest explicit superclass name. The formal definition of the ceiling $\lceil \texttt{C.E} \rceil$ of C.E can be found at the bottom of the left column in Figure 7. Ceilings will be used to determine upper bounds and superclass names in the translation of methods and classes.

We cannot, however, save generating the fixed point classes corresponding to implicit classes since they are not substitutable for ones from another family as mentioned before. That is, D$E2Fix is always defined regardless of the presence of the class declaration for D.E2 in the source program.

### Translation of Methods

Translation $|\texttt{M}|_{\texttt{A}}$ of a method declaration M (= $\texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{C}}\texttt{>T}_0\ \texttt{m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{}\uparrow\texttt{e}_0\texttt{;\}}$) in a class A consists of the translation of the signature and that of the body. Translation $|\texttt{X}_i\triangleleft\texttt{C}_i|$ of each type parameterization $\texttt{X}_i\triangleleft\texttt{C}_i$ is a sequence of pairs of a type variable and its upper bound, whose class name is obtained by ceiling. Translation of $|\texttt{e}_0|_{\Delta,\Gamma,\texttt{A}}$ of a method body $\texttt{e}_0$ is the one derived under bound environment from the type parameterization, type environment from the formal parameters, and enclosing class A. Note that *thistype*(A) is a .FJ function, which returns the type for this in class A, defined as follows: *thistype*(C.E) = .E; *thistype*(C) = C.

### Translation of Classes

Translation $|\texttt{NL}|_{\texttt{C}}$ of a nested class declaration `NL` in a top-level class `C` is a generic class with a self type variable consisting of a set of F-bounded constraints $|\triangleleft\texttt{C}|$ and the translation of the field and method declarations. A set of F-bounded constraints $|\triangleleft\texttt{C}|$ is defined similarly to $|\texttt{X}\triangleleft\texttt{C}|$. The difference is in how to make the names of type variables: a relative path type translates to a type variable of the same name in which "." is removed. If a nested class does not have a superclass, the superclass of the translation will be `Object`. Otherwise it will be another generic class with a self type variable.

Translation $|\texttt{L}|$ of a top-level class declaration `L` consists of the translations of its nested classes, the fixed point classes, and the class declaration in which the field and method declarations are translated and the nested classes are removed. The function $\mathit{fix}(\texttt{C})$ is defined to generate a sequence of fixed point classes $\texttt{C\$E}_1\texttt{Fix}$ ... $\texttt{C\$E}_n\texttt{Fix}$ for nested classes $\overline{\texttt{E}}$ in top-level class `C`. The generated classes have bodies that are empty except constructor declarations. Recall that fixed point classes are generated for all nested classes whether they are implicitly inherited or explicitly redefined in subfamilies.

### Correctness

Now, we prove that the translation preserves typing and reduction. Note that one-step reduction in .FJ corresponds to also one-step reduction in FGJ$_{\text{self}}$, due to the way the translation is defined. The translation $|\Delta|$ of a bound environment $\Delta$ is defined similarly to $|\texttt{X}\triangleleft\texttt{C}|$. We write $\vdash_{\text{FGJ}}$ and $\longrightarrow_{\text{FGJ}}$ for the judgment and reduction relation of FGJ$_{\text{self}}$, respectively.

**Theorem 4.4 (Translation Preserves Typing)** *If a .FJ class table CT is ok, then by using the typing rules of FGJ$_{self}$ $|CT|$ is ok and*

1. *if* $\Delta;\Gamma;\texttt{C}\vdash\texttt{e}:\texttt{T}$, *then* $|\Delta|;|\Gamma|\vdash_{\text{FGJ}}|\texttt{e}|_{\Delta,\Gamma,\texttt{C}}:|\texttt{T}|$.
2. *if* $\Delta;\Gamma;\texttt{C.E}\vdash\texttt{e}:\texttt{T}$, *then* $|\Delta|,|\triangleleft\texttt{C}|;|\Gamma|\vdash_{\text{FGJ}}|\texttt{e}|_{\Delta,\Gamma,\texttt{C.E}}:|\texttt{T}|$.

**Theorem 4.5 (Translation Preserves Reduction)** *If* $\Delta;\Gamma;\texttt{A}\vdash\texttt{e}:\texttt{T}$ *and* $\texttt{e}\longrightarrow\texttt{e}'$, *then* $|\texttt{e}|_{\Delta,\Gamma,\texttt{A}}\longrightarrow_{\text{FGJ}}|\texttt{e}'|_{\Delta,\Gamma,\texttt{A}}$.

## 5 RELATED WORK

### Extensible Mutually Recursive Classes with Generics

Encoding extensible mutually recursive classes with generics has been an active topic for a long time. When Wadler discussed the expression problem [25], his original solution using generics turned out to be untypable due to the same problem of self typing as ours. He pointed out as the reason that fixed point classes are not really fixed points: for instance, Graph$EdgeFix is not equivalent to Graph$Edge< Graph$NodeFix, Graph$EdgeFix> since the latter is not a subtype of the former. Self type variables amount to considering these two types to be equivalent in type-checking: we can consider that the type system implicilty inserts a downcast from the ordinary self type Graph$Edge<Node, Edge> to a subtype Edge before this. This implicit downcasting is safe thanks to the restriction on instantiation and subclassing of classes with self type variables.

There is a workaround to make all translated programs well typed with the typing rules of FGJ without any extension like self type variables. The basic idea is to introduce an expression that will refer to the same object as this, but whose type is a type variable. We describe two approaches. One by Torgersen [23] is to introduce an extra argument which is assumed to accept the receiver object, for example:

```
void connect(Node s, Node d, Edge self) { ...  s.add(self); ... }
e.connect(n1, n2, e);
```

Another one by Bruce, Odersky, and Wadler [5] is to declare abstract methods which will be implemented in fixed point classes so that they simply return this, as follows:

```
class Graph$Edge<Node ◁ Graph$Node<Node,Edge>,
                 Edge ◁ Graph$Edge<Node,Edge>>{
    abstract Edge getThis();
    void connect(Node s, Node d){ .. s.add(getThis());  ... }
}
class Graph$EdgeFix ◁ Graph$Edge<Graph$NodeFix, Graph$EdgeFix>{
    Graph$EdgeFix getThis(){ return this; }
}
```

Both approaches insert additional code (colored in red). Since the addition contains executable code, the run-time behavior changes: the former requires an evaluation of the extra argument and the latter requires an extra method invocation.

## Other Mechanisms for Flexible Self Typing

LOOJ [4] is another variant of Java 5.0 extended with a built-in type ThisClass, which represents self types. Its meaning changes when moving to subclasses as that of a relative path type changes. Since this is of type ThisClass in LOOJ[4], one may

[4]More precisely, this is of type @ThisClass representing a narrower type, but it does not matter in this argument.

expect that a translation to LOOJ will be successful. However, although useful in a self-recursive class, ThisClass will not help us, in the mutually recursive setting, since the syntax does not allow ThisClass to appear in the upper bounds of type variables. So the following code is not allowed.

```
class Graph$Node<Edge ◁ Graph$Edge<ThisClass>> {..}
class Graph$Edge<Node ◁ Graph$Node<ThisClass>> {..}
```

In Scala [20], we can give self references arbitrary types explicitly as mentioned before. This mechanism is much more flexible than our extension, in which only type variables can be chosen for explicit self types. Chin et al. [8] have proposed a similar mechanism in an extension of variant parametric types [13].

## Integration with Other Typing Mechanisms

Although self type variables are not as powerful as arbitrary self types in Scala, we feel that they are a lightweight yet useful extension of generics. Here, we give a short discussion how they can be integrated into other typing mechanisms.

Self type variables are easily integrated with Java's interfaces and wildcards [24]. An interface type can be the upper bound of a self type variable in a generic class if the class implements the interface. When the class is fixed, the fixed point class will have to implement a fixed point interface of the interface.

Self type variables are applicable to variants of Featherweight GJ, provided that F-bounded polymorphism is supported. A good example is FGJ# [16], which supports a similar mechanism, namely type inference for this, although it is invented independently. We find that the type system has a flaw—it allows unsafe subclassing and instantiation of a generic class whose self type is a type variable—resulting in failure of type soundness. It can be easily modified by adding a few restrictions.

Self type variables can be adapted to FGJ$_\omega$ [1], an extension of FGJ with type constructor parameterization. The example for simulating generic data-types with binary methods is:

```
class Collection<Self<Z> ◁ Collection<Self, Z>, X>{
      Collection<Self, X> append(Self<X> that){ .. }
  <Y> Collection<Self, Y> flatMap(Function<X, Self<Y>> f){ .. }
}
```

Self is a type constructor variable. Its application to (any) Z yields the type Self<Z>, which has an upper bound Collection<Self,Z>. The type for this is straightforwardly of type Collection<Self,X>. However, Self<X> is more appropriate.

## 6 CONCLUSION

We have shown that the formal translation from .FJ into $\mathrm{FGJ}_{\mathrm{self}}$. We have proved that the type system of $\mathrm{FGJ}_{\mathrm{self}}$ is sound and the translation is correct.

The translation has clarified that the features of lightweight family polymorphism, namely nested classes, relative/absolute path types and family-polymorphic methods, can be considered as a syntactic sugar for a lot of complicated parameterizations and fixed point classes, which would be required in Java generics. The translation has also clarified that lightweight family polymorphism provides better self typing for mutually recursive classes than Java generics, for which we have proposed self type variables to remedy the disadvantage of self typing.

We conclude that lightweight family polymorphism provides not only a set of convenient notations but also a more suitable type system, in particular for self typing, than Java generics for extensible mutually recursive classes.

**Acknowledgements.** We thank the anonymous referees for helping us improve the presentation of this paper. This work is supported in part by Grant-in-Aid for Scientific Research No. 18200001 and Grant-in-Aid for Young Scientists (B) No. 18700026 from MEXT of Japan. Saito is a research fellow of the Japan Society for the Promotion of Science for Young Scientists.

## REFERENCES

[1] Philippe Altherr and Vincent Cremet. Adding type constructor parameterization to Java. In *Proceedings of 9th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2007)*, July 2007.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 183–200, Vancouver, BC, October 1998.

[3] Kim B. Bruce. Some challenging typing issues in object-oriented languages. In *Proceedings of Workshop on Object-Oriented Development (WOOD'03)*, volume 82 of *Electronic Notes in Theoretical Computer Science*, 2003.

[4] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2004)*, volume 3086 of *Lecture Notes on Computer Science*, pages 390–414, Oslo, Norway, June 2004. Springer Verlag.

[5] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of 12th European Conference on Object-Oriented*

*Programming (ECOOP'98)*, volume 1445 of *Lecture Notes on Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer Verlag.

[6] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, volume 20 of *Electronic Notes in Theoretical Computer Science*, New Orleans, LA, April 1999. Elsevier. Available through http://www.elsevier.nl/locate/entcs/volume20.html.

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

[8] Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. A flow-based approach for variant parameteric types. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 273–290, Portland, OR, October 2006.

[9] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *Proceedings of International Conference on Aspect Oriented Software Development (AOSD'07)*, pages 121–134, Vancouver, BC, March 2007.

[10] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[11] Erik Ernst. Family polymorphism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2001)*, volume 2072 of *Lecture Notes on Computer Science*, pages 303–326, Budapest, Hungary, June 2001. Springer Verlag.

[12] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. A preliminary summary appeared in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146, Denver, CO, October 1999.

[13] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, September 2006. A preliminary version appeared under the title "On Variance-Based Subtyping for Parametric Types" in *Proceedings of 16th European Conference on Object-Oriented Programming (ECOOP2002)*, Springer LNCS vol. 2374, pages 441–469, June 2002.

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

[15] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proceedings of 6th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2004)*, June 2004.

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

[17] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pages 397–406, October 1989.

[18] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 99–115, Vancouver, BC, October 2004.

[19] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 21–36, Portland, OR, October 2006.

[20] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57, San Diego, CA, October 2005.

[21] Chieri Saito and Atsushi Igarashi. The essence of lightweight family polymorphism. In *Proceedings of 9th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2007)*, July 2007.

[22] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(03):285–331, May 2008. A preliminary summary appeared in *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS2005)*, Springer LNCS vol. 3780, pages 161–177, November, 2005.

[23] Mads Torgersen. The expression problem revisited: Four new solutions using generics. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2004)*, volume 3086 of *Lecture Notes on Computer Science*, pages 123–146, Oslo, Norway, June 2004.

[24] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, December 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, http://www.jot.fm/issues/issue_2004_12/article5.

[25] Philip Wadler. The expression problem. Discussion on the Java-Genericity mailing list, 1998.

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

## A .FJ: DEFINITIONS

In this appendix, we show the definitions of .FJ. Figure 8 shows the definitions of subtyping, type well-formedness and expression typing. Figure 9 shows the definitions of method and class typing, and reduction. (The obvious congruence rules for reduction are omitted.) In the figures, $\mathtt{m} \notin \overline{\mathtt{M}}$ (and $\mathtt{E} \notin \overline{\mathtt{NL}}$) means the method of name $\mathtt{m}$ (and the nested class of name $\mathtt{E}$, respectively) does not exist in $\overline{\mathtt{M}}$ (and $\overline{\mathtt{NL}}$, respectively). Application of type substitution $[\overline{\mathtt{F}}/\overline{\mathtt{X}}]$ is defined in the customary manner. We abbreviate a sequence of judgments in the following way: $\Delta;\mathtt{A} \vdash \mathtt{T}_1$ ok,.., $\Delta;\mathtt{A} \vdash \mathtt{T}_n$ ok to $\Delta;\mathtt{A} \vdash \overline{\mathtt{T}}$ ok; $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_1\mathtt{:T}_1$,.., $\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_n\mathtt{:T}_n$ to $\Delta;\Gamma;\mathtt{A} \vdash \overline{\mathtt{e}}\mathtt{:}\overline{\mathtt{T}}$; $\mathtt{A} \vdash \mathtt{M}_1$ ok,.., $\mathtt{A} \vdash \mathtt{M}_n$ ok to $\mathtt{A} \vdash \overline{\mathtt{M}}$ ok; $\Delta \vdash \mathtt{S}_1\mathtt{<:}\mathtt{T}_1$,.., $\Delta \vdash \mathtt{S}_n\mathtt{<:}\mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}}\mathtt{<:}\overline{\mathtt{T}}$; $\mathtt{C} \vdash \mathtt{NL}_1$ ok,.., $\mathtt{C} \vdash \mathtt{NL}_n$ ok to $\mathtt{C} \vdash \overline{\mathtt{NL}}$ ok.

We need four more definitions. We write $\mathit{bound}_\Delta(\mathtt{T})$ for the upper bound of $\mathtt{T}$ with respect to $\Delta$, defined by: $\mathit{bound}_\Delta(\mathtt{A}) = \mathtt{A}$, $\mathit{bound}_\Delta(\mathtt{X}) = \Delta(\mathtt{X})$ and $\mathit{bound}_\Delta(\mathtt{X.C}) = \Delta(\mathtt{X})\mathtt{.C}$. The *resolution* $\mathtt{T@S}$ of $\mathtt{T}$ at $\mathtt{S}$, required for expression typing, intuitively denotes the class name that $\mathtt{T}$ refers to in a given class $\mathtt{S}$. For exmpale, it is used to determine which class to look up for fields or methods when a receiver's type is relative: *fields* and *mtype* require absolute path types as arguments. The definition is: $\mathtt{.D@F.C} = \mathtt{F.D}$; $\mathtt{.D@.C} = \mathtt{.D}$; $\mathtt{F@T} = \mathtt{F}$; $\mathtt{F.C@T} = \mathtt{F.C}$. The only interesting case is the first clause: it means that a relative path type $\mathtt{.D}$ in $\mathtt{F.C}$ refers to $\mathtt{F.D}$. The functions $\mathit{thistype}(\mathtt{A})$ and $\mathit{superclass}(\mathtt{A})$ are defined as follows: $\mathit{thistype}(\mathtt{C}) = \mathtt{C}$; $\mathit{thistype}(\mathtt{C.E}) = \mathtt{.E}$; $\mathit{superclass}(\mathtt{C}) = \mathtt{D}$; $\mathit{superclass}(\mathtt{C.E}) = \mathtt{D.E}$ where $\mathtt{class\ C{\lhd}D\{..\}}$.

## B FGJ$_{\mathrm{self}}$: OMITTED DEFINITIONS

Figure 10 shows the definitions of the auxiliary functions, subtyping and expression typing of FGJ$_{\mathrm{self}}$. We abbreviate a sequence of judgments in the following way: $\Delta;\Gamma \vdash \mathtt{e}_1\mathtt{:T}_1$,.., $\Delta;\Gamma \vdash \mathtt{e}_n\mathtt{:T}_n$ to $\Delta;\Gamma \vdash \overline{\mathtt{e}}\mathtt{:}\overline{\mathtt{T}}$; $\Delta \vdash \mathtt{S}_1\mathtt{<:}\mathtt{T}_1$,.., $\Delta \vdash \mathtt{S}_n\mathtt{<:}\mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}}\mathtt{<:}\overline{\mathtt{T}}$.

**Subtyping:**

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{Object}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \qquad \frac{\mathtt{class\ C \triangleleft D\ \{...\}}}{\Delta \vdash \mathtt{C} <: \mathtt{D}}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}$$

**Type Well-formedness:**

$$\Delta; \mathtt{A} \vdash \mathtt{Object}\ \text{ok}$$

$$\frac{\mathit{bound}_\Delta(\mathtt{F}) \in \mathit{dom}(CT)}{\Delta; \mathtt{A} \vdash \mathtt{F}\ \text{ok}}$$

$$\frac{\begin{array}{c}\mathtt{C} = \mathit{bound}_\Delta(\mathtt{F}) \\ \mathtt{class\ C \triangleleft D\{..class\ E\ \{..\}\ ..\}}\end{array}}{\Delta; \mathtt{A} \vdash \mathtt{F.E}\ \text{ok}}$$

$$\frac{\begin{array}{cc}\mathtt{C} = \mathit{bound}_\Delta(\mathtt{F}) & \mathtt{class\ C \triangleleft D\{..}\overline{\mathtt{NL}}\mathtt{\}} \\ \mathtt{E} \notin \overline{\mathtt{NL}} & \Delta; \mathtt{A} \vdash \mathtt{D.E}\ \text{ok}\end{array}}{\Delta; \mathtt{A} \vdash \mathtt{F.E}\ \text{ok}}$$

$$\frac{\Delta; \mathtt{C.D} \vdash \mathtt{C.E}\ \text{ok}}{\Delta; \mathtt{C.D} \vdash \mathtt{.E}\ \text{ok}}$$

**Field Lookup:**

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C \triangleleft D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;..\}} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$$\mathit{fields}(\mathtt{Object.C}) = \bullet$$

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\{..class\ E\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;..\}..\}} \\ \mathit{fields}(\mathtt{D.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}\end{array}}{\mathit{fields}(\mathtt{C.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\ \{...}\overline{\mathtt{NL}}\mathtt{\}} \qquad \mathtt{E} \notin \overline{\mathtt{NL}} \\ \mathit{fields}(\mathtt{D.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}\end{array}}{\mathit{fields}(\mathtt{C.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}$$

**Method Type Lookup:**

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\ \{...}\overline{\mathtt{M}}\mathtt{...\}} \\ \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T_0\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{} \uparrow \mathtt{e;\ \}} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}}$$

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\ \{...}\overline{\mathtt{M}}\mathtt{...\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}}$$

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\ \{..class\ E\ \{..}\overline{\mathtt{M}}\mathtt{\}..\}} \\ \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T_0\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{} \uparrow \mathtt{e;\ \}} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C.E}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}}$$

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\ \{..class\ E\ \{..}\overline{\mathtt{M}}\mathtt{\}..\}} \\ \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D.E}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C.E}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}}$$

$$\frac{\begin{array}{c}\mathtt{class\ C \triangleleft D\ \{...}\overline{\mathtt{NL}}\mathtt{\}} \qquad \mathtt{E} \notin \overline{\mathtt{NL}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D.E}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}\end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C.E}) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T_0}}$$

**Expression Typing:**

$$\Delta; \Gamma; \mathtt{A} \vdash \mathtt{x} : \Gamma(\mathtt{x})$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T_0@A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}\end{array}}{\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e_0.f}_i : \mathtt{T}_i\mathtt{@T_0}}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T_0@A})) = \mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U_0} \\ \Delta; \mathtt{A} \vdash \overline{\mathtt{F}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{F}} <: \overline{\mathtt{C}} \\ \Delta; \Gamma; \mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \Delta \vdash \overline{\mathtt{T}} <: ([\overline{\mathtt{F}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})\mathtt{@T_0}\end{array}}{\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e_0.<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : ([\overline{\mathtt{F}}/\overline{\mathtt{X}}]\mathtt{U_0})\mathtt{@T_0}}$$

$$\frac{\begin{array}{c}\Delta; \mathtt{A} \vdash \mathtt{A_0}\ \text{ok} \qquad \mathit{fields}(\mathtt{A_0}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta; \Gamma; \mathtt{A} \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \Delta \vdash \overline{\mathtt{U}} <: (\overline{\mathtt{T}}\mathtt{@A_0})\end{array}}{\Delta; \Gamma; \mathtt{A} \vdash \mathtt{new\ A_0(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{A_0}}$$

Figure 8: .FJ: Subtyping, type well-formedness and expression typing.

**Method Typing:**

$$\frac{\begin{array}{c}\Delta = \overline{X}{<:}\overline{C} \\ \Delta;\overline{x}:\overline{T},\ \mathtt{this}:\mathit{thistype}(A);A\vdash e_0{:}U_0 \\ \Delta\vdash U_0 <: T_0 \qquad \Delta;A\vdash T_0,\overline{T},\overline{C}\ \mathrm{ok} \\ \mathit{mtype}(m,\mathit{superclass}(A)) = {<}\overline{Y}\triangleleft\overline{D}{>}\overline{S}{\rightarrow}S_0 \\ \text{implies } (\overline{X},\overline{C},\overline{T},T_0) = (\overline{Y},\overline{D},\overline{S},S_0)\end{array}}{A\vdash {<}\overline{X}\triangleleft\overline{C}{>}T_0\ m(\overline{T}\ \overline{x})\{\uparrow e_0;\}\ \mathrm{ok}}$$

**Class Typing:**

$$\frac{C.E\vdash\overline{M}\ \mathrm{ok} \qquad \emptyset;C.E\vdash\overline{T}\ \mathrm{ok}}{C\vdash \mathtt{class}\ E\ \{\overline{T}\ \overline{f};\ \overline{M}\}\ \mathrm{ok}}$$

$$\frac{C\vdash\overline{M}\ \mathrm{ok} \qquad C\vdash\overline{NL}\ \mathrm{ok} \qquad \emptyset;C\vdash\overline{T},D\ \mathrm{ok}}{\vdash \mathtt{class}\ C\triangleleft D\{\overline{T}\ \overline{f};\ \overline{M}\ \overline{NL}\}\ \mathrm{ok}}$$

**Method Body Lookup:**

$$\frac{\begin{array}{c}\mathtt{class}\ C\triangleleft D\{\ldots\overline{M}\ldots\} \\ {<}\overline{X}\triangleleft\overline{C}{>}T\ m(\overline{T}\ \overline{x})\{\uparrow e_0;\ \}\in\overline{M}\end{array}}{\mathit{mbody}(m{<}\overline{F}{>},C) = \overline{x}.[\overline{F}/\overline{X}]e_0}$$

$$\frac{\mathtt{class}\ C\triangleleft D\ \{\ldots\overline{M}\ldots\} \qquad m\notin\overline{M}}{\mathit{mbody}(m{<}\overline{F}{>},C) = \mathit{mbody}(m{<}\overline{F}{>},D)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ C\triangleleft D\ \{\ldots\overline{NL}\} \\ \mathtt{class}\ E\ \{\ldots\overline{M}\}\in\overline{NL} \\ {<}\overline{X}\triangleleft\overline{C}{>}T\ m(\overline{T}\ \overline{x})\{\uparrow e_0;\ \}\in\overline{M}\end{array}}{\mathit{mbody}(m{<}\overline{F}{>},C.E) = \overline{x}.[\overline{F}/\overline{X}]e_0}$$

$$\frac{\mathtt{class}\ C\triangleleft D\ \{\ldots\overline{NL}\} \qquad E\notin\overline{NL}}{\mathit{mbody}(m{<}\overline{F}{>},C.E) = \mathit{mbody}(m{<}\overline{F}{>},D.E)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ C\triangleleft D\ \{\ldots\overline{NL}\} \\ \mathtt{class}\ E\ \{\ldots\overline{M}\}\in\overline{NL} \qquad m\notin\overline{M}\end{array}}{\mathit{mbody}(m{<}\overline{F}{>},C.E) = \mathit{mbody}(m{<}\overline{F}{>},D.E)}$$

**Computation:**

$$\frac{\mathit{fields}(A) = \overline{T}\ \overline{f}}{\mathtt{new}\ A(\overline{e}).f_i \longrightarrow e_i}$$

$$\frac{\mathit{mbody}(m{<}\overline{F}{>},A) = \overline{x}.e_0}{\begin{array}{l}\mathtt{new}\ A(\overline{e}).{<}\overline{F}{>}m(\overline{d}) \\ \longrightarrow [\overline{d}/\overline{x},\mathtt{new}\ A(\overline{e})/\mathtt{this}]e_0\end{array}}$$

Figure 9: .FJ: Method and class typing, and reduction.

## C PROOF OF THEOREM 4.1

In this appendix, we prove the subject reduction theorem of FGJ$_{\text{self}}$. The structure of the proof is similar to that of FGJ [12], beginning with weakening lemmas, followed by various substitution lemmas, showing the properties of the lookup functions. Since self type variables are a small extension, it is sufficient to update a lemma to Lemma C.9 and add Lemma C.10. Other lemmas are unchaged and their proofs are omitted. Refer to the proofs of FGJ [12] for the omitted proofs. In what follows, the metavariable Z (and V) ranges over type variables (and types, respectively).

**Lemma C.1 (Weakening)** *Suppose* $\Delta,\overline{X}{<:}\overline{N}\vdash\overline{N}\ \mathrm{ok}$ *and* $\Delta\vdash U\ \mathrm{ok}$. *(1) If* $\Delta\vdash S{<:}T$, *then* $\Delta,\overline{X}{<:}\overline{N}\vdash S{<:}T$. *(2) If* $\Delta\vdash S$ ok-inst, *then* $\Delta,\overline{X}{<:}\overline{N}\vdash S$ ok-inst. *(3) If* $\Delta;\Gamma\vdash e{:}T$, *then* $\Delta;\Gamma,x{:}U\vdash e{:}T$ *and* $\Delta,\overline{X}{<:}\overline{N};\Gamma\vdash e{:}T$.

**Lemma C.2 (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* $\Delta_1\vdash\overline{U}\ \mathrm{ok}$ *and none of* $\overline{X}$ *appearing in* $\Delta_1$, *then* $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash[\overline{U}/\overline{X}]S <: [\overline{U}/\overline{X}]T$.

**Bound of type:**

$$\begin{aligned} bound_\Delta(\mathtt{X}) &= \Delta(\mathtt{X}) \\ bound_\Delta(\mathtt{N}) &= \mathtt{N} \end{aligned}$$

**Subtyping:**

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

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

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

$$\frac{\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{...\}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \qquad \text{(S-Class)}$$

**Field Lookup:**

$$fields(\mathtt{Object}) = \bullet \qquad \text{(F-Object)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \\ fields([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}\end{array}}{fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \qquad \text{(F-Class)}$$

**Method Type Lookup:**

$$\frac{\begin{array}{c}\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>U\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow \mathtt{e;\}} \in \overline{\mathtt{M}}\end{array}}{mtype(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U})} \qquad \text{(MT-Class)}$$

$$\frac{\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtype(\mathtt{m},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})} \qquad \text{(MT-Super)}$$

**Method Body Lookup:**

$$\frac{\begin{array}{c}\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>U\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow \mathtt{e}_0\mathtt{;\}} \in \overline{\mathtt{M}}\end{array}}{mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{x}}.[\overline{\mathtt{T}}/\overline{\mathtt{X}},\ \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e}_0} \qquad \text{(MB-Class)}$$

$$\frac{\mathtt{class}\ \ i_{opt}\ \ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N\{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\begin{aligned} &mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) \\ = \ &mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})\end{aligned}} \qquad \text{(MB-Super)}$$

**Valid Method Overriding:**

$$\frac{\begin{array}{c} mtype(\mathtt{m},\ \mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0 \text{ implies} \\ (\overline{\mathtt{Y}}\mathtt{,}\overline{\mathtt{Q}}\mathtt{,}\overline{\mathtt{T}}) = (\overline{\mathtt{Z}},\overline{\mathtt{P}},\overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}} <: \overline{\mathtt{Q}} \vdash \mathtt{T}_0 <: \mathtt{U}_0\end{array}}{override(\mathtt{m},\ \mathtt{N},\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0)}$$

**Expression Typing:**

$$\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(GT-Var)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad fields(bound_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i : \mathtt{T}_i} \qquad \text{(GT-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ mtype(\mathtt{m},\ bound_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}} \text{ ok} \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \qquad \text{(GT-Invk)}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{N} \text{ ok} \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ \ N(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{N}} \qquad \text{(GT-New)}$$

Figure 10: FGJ$_{\text{self}}$: Subtyping, auxiliary functions, and expression typing.

**Lemma C.3 (Type Substitution Preserves Correct Class Instantiation)** *If* $\Delta_1,\ \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ ok-inst *and* $\Delta_1 \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1 \vdash \overline{\mathtt{U}}$ ok-inst *and none of* $\overline{\mathtt{X}}$ *appearing in* $\Delta_1$*, then* $\Delta_1,\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$ ok-inst*.*

**Lemma C.4** *Suppose* $\Delta_1, \overline{X}\texttt{<:}\overline{N}, \Delta_2 \vdash T$ 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* $\Delta_1$. *Then,* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \mathit{bound}_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T) <: [\overline{U}/\overline{X}](\mathit{bound}_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T))$.

**Lemma C.5** *If* $\Delta \vdash S <: T$ *and* $\mathit{fields}(\mathit{bound}_\Delta(T)) = \overline{T}\ \overline{f}$, *then* $\mathit{fields}(\mathit{bound}_\Delta(S)) = \overline{S}\ \overline{g}$ *and* $S_i = T_i$ *and* $g_i = f_i$ *for all* $i \leq |\overline{f}|$.

**Lemma C.6** *If* $\Delta \vdash T$ ok *and* $\mathit{mtype}(m, \mathit{bound}_\Delta(T)) = \texttt{<}\overline{Y}\triangleleft\overline{P}\texttt{>}\overline{U}\rightarrow U_0$, *then for any* $S$ *such that* $\Delta \vdash S <: T$ *and* $\Delta \vdash S$ ok, *we have* $\mathit{mtype}(m, \mathit{bound}_\Delta(S)) = \texttt{<}\overline{Y}\triangleleft\overline{P}\texttt{>}\overline{U}\rightarrow U_0{}'$ *and* $\Delta, \overline{Y}<:\overline{P} \vdash U_0{}' <: U_0$.

**Lemma C.7 (Type Substitution Preserves Typing)** *If* $\Delta_1, \overline{X}<:\overline{N}, \Delta_2; \Gamma \vdash e\texttt{:}T$ *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* $\Delta_1$, *then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e\texttt{:}S$ *for some* $S$ *such that* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S <: [\overline{U}/\overline{X}]T$.

**Lemma C.8 (Term Substitution Preserves Typing)** *If* $\Delta; \Gamma, \overline{x} : \overline{T} \vdash e\texttt{:}T$ *and* $\Delta; \Gamma \vdash \overline{d}\texttt{:}\overline{S}$ *where* $\Delta \vdash \overline{S} <: \overline{T}$, *then* $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]e\texttt{:}S$ *for some* $S$ *such that* $\Delta \vdash S <: T$.

**Lemma C.9** *If* $\mathit{mtype}(m, C\texttt{<}\overline{T}\texttt{>}) = \texttt{<}\overline{Y}\triangleleft\overline{P}\texttt{>}\overline{U}\rightarrow U$ *and* $\mathit{mbody}(m\texttt{<}\overline{V}\texttt{>}, C\texttt{<}\overline{T}\texttt{>}) = \overline{x}.e_0$ *where* $\Delta \vdash \overline{V}$ ok *and* $\Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P}$ *and* $\Delta \vdash C\texttt{<}\overline{T}\texttt{>}$ ok-inst, *then there exist some* $N$ *and* $S$ *such that* $\Delta \vdash C\texttt{<}\overline{T}\texttt{>} <: N$ *and* $\Delta \vdash N$ ok-inst *and* $\Delta \vdash S <: [\overline{V}/\overline{Y}]U$ *and* $\Delta \vdash S$ ok-inst *and* $\Delta; \overline{x} : [\overline{V}/\overline{Y}]\overline{U}, \mathtt{this} : \mathit{selftype}(N) \vdash e_0\texttt{:}S$.

*Proof.* By induction on the derivation of $\mathit{mbody}(m\texttt{<}\overline{V}\texttt{>}, C\texttt{<}\overline{T}\texttt{>})$ using Lemma C.8 with a case analysis on the last rule used. □

**Lemma C.10** *If* $\Delta \vdash N <: P$, *then* $\Delta \vdash \mathit{selftype}(N) <: \mathit{selftype}(P)$

*Proof.* By induction on the derivation of $\Delta \vdash N <: P$. Note that if $C\texttt{<}\overline{X}\triangleleft\overline{N}\texttt{>} \vdash N$ ok-superclass, then $\overline{X}<:\overline{N} \vdash \mathit{selftype}(C\texttt{<}\overline{X}\texttt{>}) <: \mathit{selftype}(N)$. □

*Proof.* (Theorem 4.1) By induction on the derivation of $e \longrightarrow e'$ with a case analysis on the reduction rule used. We show only the case of GR-Invk. Other cases can be proved as described in [12].

**Case** GR-Invk: $e = \mathtt{new}\ N(\overline{e}).\texttt{<}\overline{V}\texttt{>}m(\overline{d})$ $\qquad$ $\mathit{mbody}(m\texttt{<}\overline{V}\texttt{>}, N) = \overline{x}.e_0$

$e' = [\overline{d}/\overline{x}, \mathtt{new}\ N(\overline{e})/\mathtt{this}]e_0$

By the rules GT-Invk and GT-New, we have

$$\Delta; \Gamma \vdash \mathtt{new}\ N(\overline{e})\texttt{:}N \qquad \mathit{mtype}(m, \mathit{bound}_\Delta(N)) = \texttt{<}\overline{Y}\triangleleft\overline{P}\texttt{>}\overline{U}\rightarrow U \qquad \Delta \vdash \overline{V} \text{ ok}$$
$$\Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P} \qquad \Delta; \Gamma \vdash \overline{d}\texttt{:}\overline{S} \qquad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U} \qquad T = [\overline{V}/\overline{Y}]U \qquad \Delta \vdash N \text{ ok}$$

By Lemma C.9, $\Delta; \overline{x} : [\overline{V}/\overline{Y}]\overline{U}, \mathtt{this} : \mathit{selftype}(P) \vdash e_0\texttt{:}S$ for some $P$ and $S$ such that $\Delta \vdash P$ ok-inst and $\Delta \vdash N <: P$ and $\Delta \vdash S <: [\overline{V}/\overline{Y}]U$ and $\Delta \vdash S$ ok-inst. Since $N = \mathit{selftype}(N)$, by Lemma C.10 $\Delta \vdash N <: \mathit{selftype}(P)$. Then, by Lemmas C.1 and C.8, $\Delta; \Gamma \vdash [\overline{d}/\overline{x}, \mathtt{new}\ N(\overline{e})/\mathtt{this}]e_0\texttt{:}T_0$ for some $T_0$ such that $\Delta \vdash T_0 <: S$. By S-Trans, we have $\Delta \vdash T_0 <: T$. Finally, letting $T' = T_0$ finishes the case. □

## D PROOF OF THEOREMS 4.4 AND 4.5

First, we develop a number of the required lemmas.

**Lemma D.1** *1. If* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$*, then* $\mathit{fields}_{\mathrm{FGJ}}(|\mathtt{C}|) = |\overline{\mathtt{T}}|\ \overline{\mathtt{f}}$*.*

*2. If* $\mathit{fields}(\mathtt{C.E}_i) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *where* $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$*, then* $\mathit{fields}_{\mathrm{FGJ}}(\lceil \mathtt{C.E}_i \rceil \mathtt{<}|\overline{\mathtt{U}}|\mathtt{>}) = |\overline{\mathtt{T}}@\mathtt{U}_i|\ \overline{\mathtt{f}}$ *for* $\overline{\mathtt{U}}$ *where* $\overline{\mathtt{U}} = \mathtt{F}.\overline{\mathtt{E}}$ *or* $\overline{\mathtt{U}} = .\overline{\mathtt{E}}$*.*

*3. If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0$*, then* $\mathit{mtype}_{\mathrm{FGJ}}(\mathtt{m}, \mathtt{C}) = \mathtt{<}|\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}|\mathtt{>}|\overline{\mathtt{T}}|\rightarrow|\mathtt{T}_0|$*.*

*4. If* $\mathit{mtype}(\mathtt{m}, \mathtt{C.E}_i) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0$ *where* $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$*, then* $\mathit{mtype}_{\mathrm{FGJ}}(\mathtt{m}, \lceil \mathtt{C.E}_i \rceil \mathtt{<}|\overline{\mathtt{U}}|\mathtt{>}) = \mathtt{<}|\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}|\mathtt{>}|\overline{\mathtt{T}}@\mathtt{U}_i|\rightarrow|\mathtt{T}_0@\mathtt{U}_i|$ *for* $\overline{\mathtt{U}}$ *where* $\overline{\mathtt{U}} = \mathtt{F}.\overline{\mathtt{E}}$ *or* $\overline{\mathtt{U}} = .\overline{\mathtt{E}}$*.*

*Proof.* Each can be proved by induction on the derivation of $\mathit{fields}(\mathtt{C})$, $\mathit{fields}(\mathtt{C.E}_i)$, $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ and $\mathit{mtype}(\mathtt{m}, \mathtt{C.E}_i)$, respectively. Note that if $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$ and $\Delta; \mathtt{C.E}_j \vdash \mathtt{T}$ ok for some $\Delta$, then $[|\overline{\mathtt{U}}|/|.\overline{\mathtt{E}}|]|\mathtt{T}| = |\mathtt{T}@\mathtt{U}_i|$ for $\overline{\mathtt{U}}$ where $\overline{\mathtt{U}} = \mathtt{F}.\overline{\mathtt{E}}$ or $\overline{\mathtt{U}} = .\overline{\mathtt{E}}$. □

**Lemma D.2** *If* $\Delta \vdash \mathtt{S<:T}$*, then* $|\Delta| \vdash_{\mathrm{FGJ}} |\mathtt{S}|\mathtt{<:}|\mathtt{T}|$*.*

*Proof.* By induction on the derivation of $\Delta \vdash \mathtt{S<:T}$. □

**Lemma D.3** *If* $\Delta \vdash \mathtt{F<:C}$ *and* $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$*, then* $|\Delta| \vdash_{\mathrm{FGJ}} |\mathtt{F}|_{\mathtt{C}}\mathtt{<:}\ ([|\mathtt{F}|_{\mathtt{C}}/|\mathtt{X}|_{\mathtt{C}}]\lceil \mathtt{C.E}_i \rceil \mathtt{<}|\mathtt{X}.\overline{\mathtt{E}}|\mathtt{>}), \mathtt{C}$*.*

*Proof.* By case analysis on $\mathtt{F}$ using the fact that $|[\mathtt{F}/\mathtt{X}]\mathtt{T}| = [|\mathtt{F}|_{\mathtt{C}}/|\mathtt{X}|_{\mathtt{C}}]|\mathtt{T}|$. □

**Lemma D.4** *1. if* $\Delta; \mathtt{C} \vdash \mathtt{F}, \mathtt{C}$ ok *and* $\Delta \vdash \mathtt{F<:C}$*, then* $|\Delta| \vdash_{\mathrm{FGJ}} |\mathtt{F}|_{\mathtt{C}}$ ok.

*2. if* $\Delta; \mathtt{C} \vdash \mathtt{T}$ ok*, then* $|\Delta| \vdash_{\mathrm{FGJ}} \mathtt{T}$ ok.

*3. if* $\Delta; \mathtt{C.E}_i \vdash \mathtt{T}$ ok *where* $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$*, then* $|\Delta|, |\triangleleft\mathtt{C}| \vdash_{\mathrm{FGJ}} |\mathtt{T}|$ ok.

*4. if* $\mathit{classes}(\mathtt{C}) = \overline{\mathtt{E}}$*, then* $|\mathtt{X<:C}|; |\mathtt{X.E}_i| \vdash_{\mathrm{FGJ}} \lceil \mathtt{C.E}_i \rceil \mathtt{<}|\mathtt{X}.\overline{\mathtt{E}}|\mathtt{>}$ ok-bound.

*5. if* $\emptyset; \mathtt{A} \vdash \mathtt{C}$ ok*, then* $\vdash_{\mathrm{FGJ}} \mathit{fix}(\mathtt{C})$ ok

*Proof.* Each can be easily proved. □

*Proof.* (Theorem 4.4) We prove the theorem in two steps: first, it is shown that if $\Delta; \Gamma; \mathtt{C} \vdash \mathtt{e:T}$ then $|\Delta|; |\Gamma| \vdash_{\mathrm{FGJ}} |\mathtt{e}|_{\Delta,\Gamma,\mathtt{C}}\mathtt{:}|\mathtt{T}|$, and if $\Delta; \Gamma; \mathtt{C.E} \vdash \mathtt{e:T}$ then $|\Delta|, |\triangleleft\mathtt{C}|; |\Gamma| \vdash_{\mathrm{FGJ}} |\mathtt{e}|_{\Delta,\Gamma,\mathtt{C.E}}\mathtt{:}|\mathtt{T}|$; and second, we show $CT$ is ok.

The first part is proved by induction on the derivation of $\Delta; \Gamma; \mathtt{C} \vdash \mathtt{e:T}$ and $\Delta; \Gamma; \mathtt{C.E} \vdash \mathtt{e:T}$. We show only the latter case since the former can be proved similarly to the latter. The proof below shows only the case for field accesses. Other cases for method invocations and object creations can be proved similarly.

**Case:** $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i$ $\quad$ $\Delta; \Gamma; \mathtt{C.E} \vdash \mathtt{e}_0 : \mathtt{T}_0$
$\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0@\mathtt{C.E})) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$ $\quad$ $\mathtt{T} = \mathtt{T}_i@\mathtt{T}_0$

By induction hypothesis, $|\Delta|, |\triangleleft\mathtt{C}|; |\Gamma| \vdash_{\mathrm{FGJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{C.E}} : |\mathtt{T}_0|$. Case analysis on $\mathtt{T}_0$.

**Subcase:** $\mathtt{T}_0 = \mathtt{F}$

By Lemma D.1(1) and the fact that $|\Delta(\mathtt{F})| = |\Delta|(|\mathtt{F}|)$, $\mathit{fields}_{\mathrm{FGJ}}(\mathit{bound}_{(|\Delta|,|\triangleleft\mathtt{C}|)}(|\mathtt{F}|)) = |\bar{\mathtt{T}}|\ \bar{\mathtt{f}}$. Since $\mathtt{T}_i@\mathtt{T}_0 = \mathtt{T}_i$, by GT-Field $|\Delta|, |\triangleleft\mathtt{C}|; |\Gamma| \vdash_{\mathrm{FGJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{C.E}}\mathtt{.f}_i : |\mathtt{T}_i|$.

**Subcase:** $\mathtt{T}_0 = \mathtt{C}'\mathtt{.D}_j$ where $\mathit{classes}(\mathtt{C}') = \bar{\mathtt{D}}$

Since there exists $\mathtt{class}\ |\mathtt{C}'\mathtt{.D}_j| \triangleleft \lceil\mathtt{C}'\mathtt{.D}_j\rceil\mathtt{<}|\mathtt{C}'\mathtt{.}\bar{\mathtt{D}}|\mathtt{>\{\}}$, by Lemma D.1(2) $\mathit{fields}_{\mathrm{FGJ}}(|\mathtt{C}'\mathtt{.D}_j|) = |\bar{\mathtt{T}}@\mathtt{C}'\mathtt{.D}_j|\ \bar{\mathtt{f}}$. By GT-Field, $|\Delta|, |\triangleleft\mathtt{C}|; |\Gamma| \vdash_{\mathrm{FGJ}} |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{C.E}}\ \mathtt{.f}_i\mathtt{:}\ |\mathtt{T}_i@\mathtt{C}'\mathtt{.D}_j|$.

**Subcase:** $\mathtt{T}_0 = \mathtt{X.D}_j$ where $\mathtt{X}\mathtt{<:}\mathtt{C}' \in \Delta$ and $\mathit{classes}(\mathtt{C}') = \bar{\mathtt{D}}$

Since $\mathit{bound}_\Delta(\mathtt{X.D}_j) = \mathtt{C}'\mathtt{.D}_j$, by Lemma D.1(2) $\mathit{fields}_{\mathrm{FGJ}}(\mathit{bound}_{(|\Delta|,|\triangleleft\mathtt{C}|)}(|\mathtt{X.D}_j|)) = \mathit{fields}_{\mathrm{FGJ}}(\lceil\mathtt{C}'\mathtt{.D}_j\rceil\mathtt{<}|\mathtt{X.}\bar{\mathtt{D}}|\mathtt{>}) = |\bar{\mathtt{T}}@\mathtt{X.D}_j|$. By GT-Field, $|\mathtt{T}| = |\mathtt{T}_i@\mathtt{X.D}_j|$.

**Subcase:** $\mathtt{T}_0 = \mathtt{.E}_j$ where $\mathit{classes}(\mathtt{C}) = \bar{\mathtt{E}}$

Since $\mathtt{.E}_j@\mathtt{C.E} = \mathtt{C.E}_j$, $\mathit{fields}_{\mathrm{FGJ}}(\mathit{bound}_{(|\Delta|,|\triangleleft\mathtt{C}|)}(|\mathtt{.E}_j|)) = \mathit{fields}_{\mathrm{FGJ}}(\lceil\mathtt{C.E}_j\rceil\mathtt{<}|\mathtt{.}\bar{\mathtt{E}}|\mathtt{>}) = |\bar{\mathtt{T}}@\mathtt{.E}_j|\ \bar{\mathtt{f}}$, by Lemma D.1(2). By GT-Field, $|\mathtt{T}| = |\mathtt{T}_i@\mathtt{.E}_j|$.

The second part ($|CT|$ ok) follows from the first part with examination of the rules GT-Method, GT-Class and GT-ClassSelf. It is easy to show that: if $\mathtt{C} \vdash \mathtt{M}$ ok, then $\mathtt{C} \vdash_{\mathrm{FGJ}} |\mathtt{M}|_{\mathtt{C}}$ ok; if $\mathtt{C.E}_i \vdash \mathtt{M}$ ok where $\mathit{classes}(\mathtt{C}) = \bar{\mathtt{E}}$, then $\mathtt{C\$E}_i\mathtt{<}|\triangleleft\mathtt{C}|\mathtt{>}\vdash_{\mathrm{FGJ}} |\mathtt{M}|_{\mathtt{C.E}_i}$; if $\mathtt{C} \vdash \mathtt{NL}$ ok, then $\vdash_{\mathrm{FGJ}} |\mathtt{NL}|_{\mathtt{C}}$ ok; if $\vdash \mathtt{L}$ ok, then $\vdash_{\mathrm{FGJ}} |\mathtt{L}|$ ok. □

We need the following three lemmas from [22]. We write $\mathtt{A}\ \mathtt{<\#}\ \mathtt{B}$ if either (1) $\mathtt{A} = \mathtt{C}$, $\mathtt{B} = \mathtt{D}$, and $\vdash \mathtt{C}\ \mathtt{<:}\ \mathtt{D}$, or (2) $\mathtt{A} = \mathtt{C.E}$, $\mathtt{B} = \mathtt{D.E}$, and $\vdash \mathtt{C}\ \mathtt{<:}\ \mathtt{D}$.

**Lemma D.5** *If* $\mathtt{A}\ \mathtt{<\#}\ \mathtt{B}$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{B}) = \mathtt{<}\bar{\mathtt{X}}\triangleleft\bar{\mathtt{C}}\mathtt{>}\bar{\mathtt{U}}\mathtt{\rightarrow}\mathtt{U}_0$, *then* $\mathit{mtype}(\mathtt{m}, \mathtt{A}) = \mathtt{<}\bar{\mathtt{X}}\triangleleft\bar{\mathtt{C}}\mathtt{>}\bar{\mathtt{U}}\mathtt{\rightarrow}\mathtt{U}_0$.

**Lemma D.6** *If* $\Delta, \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{C}}; \Gamma; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta \vdash \bar{\mathtt{F}}$ ok *and* $\Delta \vdash \bar{\mathtt{F}}\ \mathtt{<:}\ \bar{\mathtt{C}}$, *then there exists some* $\mathtt{S}$ *such that* $\Delta; [\bar{\mathtt{F}}/\bar{\mathtt{X}}]\Gamma; \mathtt{A} \vdash [\bar{\mathtt{F}}/\bar{\mathtt{X}}]\mathtt{e} : \mathtt{S}$ *and* $\Delta \vdash \mathtt{S}\ \mathtt{<:}\ [\bar{\mathtt{F}}/\bar{\mathtt{X}}]\mathtt{T}$.

**Lemma D.7** *If* $\Delta; \Gamma, \bar{\mathtt{x}}\mathtt{:}\bar{\mathtt{T}}; \mathtt{A} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta; \Gamma; \mathtt{A} \vdash \bar{\mathtt{d}} : \bar{\mathtt{S}}$ *and* $\Delta \vdash \bar{\mathtt{S}}\ \mathtt{<:}\ \bar{\mathtt{T}}$, *then there exists some* $\mathtt{S}$ *such that* $\Delta; \Gamma; \mathtt{A} \vdash [\bar{\mathtt{d}}/\bar{\mathtt{x}}]\mathtt{e} : \mathtt{S}$ *and* $\Delta \vdash \mathtt{S}\ \mathtt{<:}\ \mathtt{T}$.

**Lemma D.8** *If* $\mathtt{X}\mathtt{<:}\mathtt{C} \in \Delta$ *and* $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}\mathtt{:}\mathtt{T}$ *and* $\Delta \vdash \mathtt{F}\mathtt{<:}\mathtt{C}$, *then* $|[\mathtt{F}/\mathtt{X}]\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}} = [|\mathtt{F}|_{\mathtt{C}}/|\mathtt{X}|_{\mathtt{C}}]|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}}$

*Proof.* By induction on the derivation of $\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e}\mathtt{:}\mathtt{T}$. Note that if $\emptyset; \mathtt{B} \vdash \mathtt{C}\mathtt{<:}\mathtt{D}$, then $|\mathtt{F}|_{\mathtt{D}} = [|\mathtt{F}|_{\mathtt{C}}/|\mathtt{X}|_{\mathtt{C}}]|\mathtt{X}|_{\mathtt{D}}$. □

**Lemma D.9** *If* $\Delta; \Gamma; \mathtt{A} \vdash \bar{\mathtt{e}}\mathtt{:}\bar{\mathtt{T}}$ *and* $\Delta'; \Gamma'; \mathtt{A}_0 \vdash \mathtt{e}_0\mathtt{:}\mathtt{T}_0$, *then* $|[\bar{\mathtt{e}}/\bar{\mathtt{x}}]\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}} = [|\bar{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}/\bar{\mathtt{x}}]\ |\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}_0}$.

*Proof.* By induction on the derivation of $\Delta'; \Gamma'; \mathtt{A}_0 \vdash \mathtt{e}_0 \mathtt{:} \mathtt{T}_0$. □

*Proof.*(Theorem 4.5) By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with a case analysis on the last reduction rule used. We show only the cases for computation. Other cases for the congruence rules can be proved immediately by the induction hypothesis.

**Case:** $\mathtt{e} = \mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{).f}_i \qquad \mathtt{e}' = \mathtt{e}_i \qquad \mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

Since $|\mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{).f}_i|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{).f}_i$, $\mathit{fields}_{\mathrm{FGJ}}(|\mathtt{A}_0|) = |\overline{\mathtt{T}}@\mathtt{A}_0|\ \overline{\mathtt{f}}$ by Lemmas D.1(1, 2). By GR-Field, $\mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{).f}_i \longrightarrow |\mathtt{e}_i|$

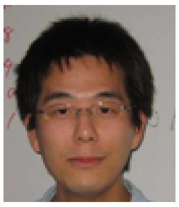
**Case:** $\mathtt{e} = \mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{).<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)} \qquad \mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)}/\mathtt{this}]\mathtt{e}'_0$
$\mathit{mbody}(\mathtt{m<}\overline{\mathtt{F}}\mathtt{>}, \mathtt{A}_0) = \overline{\mathtt{x}}\mathtt{.e}'_0$

Assume that $\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\uparrow e}_0\mathtt{;\}}$. By the definition of *mbody*, $\mathtt{e}'_0 = [\overline{\mathtt{F}}/\overline{\mathtt{X}}]\mathtt{e}_0$. By Tr-New and Tr-Invk, $|\mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{).<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{new}\ |\mathtt{A}_0|\ \mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{).}\ \mathtt{<}|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}\mathtt{>m(}$ $|\overline{\mathtt{d}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}$. Let $\Delta' = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}$, $\Gamma' = \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}, \mathtt{this:A}_0$. Since $\mathit{mbody}_{\mathrm{FGJ}}\ (\mathtt{m<}|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}\mathtt{>}, |\mathtt{A}_0|) = \overline{\mathtt{x}}.\mathtt{e}''_0$ where $\mathtt{e}''_0 = [|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}/|\overline{\mathtt{X}}|_{\overline{\mathtt{C}}}]|\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}_0}$, we must show that
$|[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)}/\mathtt{this}]\mathtt{e}'_0|_{\Delta,\Gamma,\mathtt{A}} = [|\overline{\mathtt{d}}|_{\Delta,\Gamma,\mathtt{A}}/\overline{\mathtt{x}}, \mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}/\mathtt{this}]\mathtt{e}''_0$.

$$\begin{array}{rll} & [|\overline{\mathtt{d}}|_{\Delta,\Gamma,\mathtt{A}}/\overline{\mathtt{x}}, \mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}/\mathtt{this}]\mathtt{e}''_0 & \text{(by Lemma D.8)} \\ = & [|\overline{\mathtt{d}}|_{\Delta,\Gamma,\mathtt{A}}/\overline{\mathtt{x}}, \mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}/\mathtt{this}]|[\overline{\mathtt{F}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta',\Gamma',\mathtt{A}_0} & \text{(by Lemma D.9)} \\ = & |[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)}/\mathtt{this}][\overline{\mathtt{F}}/\overline{\mathtt{X}}]\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}. & \end{array}$$

□

## ABOUT THE AUTHORS



**Chieri Saito** is a PhD student at Graduate School of Informatics, Kyoto University, Japan. See http://www.sato.kuis.kyoto-u.ac.jp/~saito/.

**Atsushi Igarashi** is an associate professor at Kyoto University. His home page is at http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/.