The Essence of Lightweight Family Polymorphism

Chieri Saito and Atsushi Igarashi

Graduate School of Informatics, Kyoto University, Japan
{saito,igarashi}@kuis.kyoto-u.ac.jp

Abstract. The formal core calculus .FJ has been introduced to model lightweight family polymorphism, a programming style to support reusable yet type-safe mutually recursive classes. This paper clarifies the essence of the features of .FJ, by giving a formal translation from .FJ into a variant of Featherweight GJ with a little extension of F-bounded polymorphism. The extension, which allows self types to appear in mutually recursive constraints on type variables, is significant to achieve a clear correspondence between the two languages without losing type safety. We show that the extended version of Featherweight GJ is type sound and that 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 since the types of mutual references are the class names, which cannot be changed when they are inherited. As a result, type safety would usually be lost by typecasts inserted to sidestep the problem. We have proposed lightweight family polymorphism [7], a programming style in object-oriented programming to support reusable yet type-safe mutually recursive classes. Our proposal solved this problem by introducing new language features that can be easily applied to Java-like languages. Their semantics has been formalized as .FJ, an extension of Featherweight Java [5], and its type system has been proved sound. Actually, however, it had been known that similar programming is possible with existing features of generics [1] and F-bounded polymorphism [4] with which Java 5.0 and C# are equipped, although this programming requires a lot of boilerplate code. It makes us wonder if our proposal is merely a convenient syntax sugar for Java 5.0—in other words, are there any essential differences between them?

In this paper, we clarify the essence of the language features supporting lightweight family polymorphism by giving a translation from .FJ to Featherweight GJ [5] (with a small extension, as we will see) and answer the questions raised above—what works as a syntax sugar and what is a distinguishing characteristic. To clearly reveal correspondence between a program and its translation, we allow a translation to change only types and class names and do not to change

run-time behavior by, say, inserting casts or additional methods. This translation is given mainly by applying the technique of Torgersen [8], used to solve the expression problem, to the mutually recursive setting. Although, most translated programs are well typed with the original typing rules, program fragments of a certain form, unfortunately, are *not* well typed since the typing schema for self references of the two languages are different. There are workarounds, found in Torgersen [8] and Bruce et. al. [3], to make them type-check, but they would change run-time behavior.

To make all translations type-check without changing run-time behavior, we propose a little extension of F-bounded polymorphism in Java 5.0 and translate to a variant of Java 5.0 with this extension. This extension allows self types to appear in mutually recursive constraints on type variables in a generic class. On the other hand, subclassing and instantiation of such classes will be limited for safety. As a natural result, the additional typing schema for the extension resembles that of .FJ involving self references.

To rigorously show that the extension and translation are correct, we give

- a variant of Featherweight GJ (FGJ) [5], a formal core calculus of GJ [1] or Java 5.0, with extended F-bounded polymorphism,
- a formal translation from .FJ to the extended version of FGJ,
- a type soundness theorem of the extended version of FGJ, and
- theorems of correctness of the formal translation.

The correctness result of the translation states that the proposed extension of F-bounded polymorphism captures the essential difference between .FJ and (the original) FGJ.

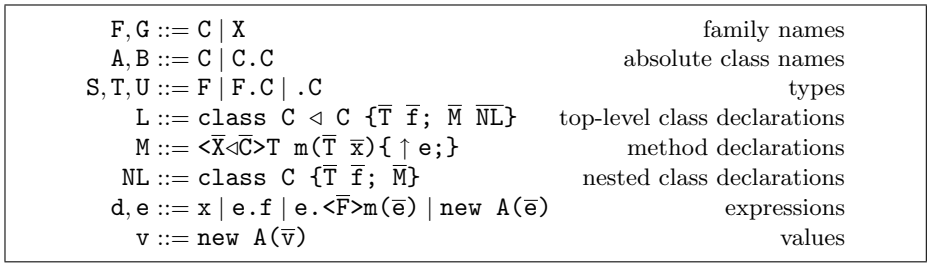
Besides the theoretical interest, the translation can be used for an implementation of lightweight family polymorphism, although there has been another possibility [7] using erasure [1]. The advantage of the present one is that the translation preserves the original type information without using typecasts.

Rest of This Paper. Section 2 reviews .FJ. Section 3 shows the outline of the translation. Section 4 formalizes the translation and proves its correctness as well as type soundness of the extended version of FGJ. Section 5 discusses related work. Section 6 concludes. A full version of this paper with proofs of the theorems will be available at `http://www.sato.kuis.kyoto-u.ac.jp/~saito/elfp/`.

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

In this section, we review the key features of .FJ to support lightweight family polymorphism quite briefly. We would like to refer readers to the original paper [7] for details and examples.

Figure 1 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` ranges over variables. The symbols



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

Fig. 1. .FJ: Syntax.

$\lhd$ 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 "$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$" for "$\mathtt{T}_1\ \mathtt{f}_1\mathtt{,}\ldots\mathtt{,T}_n\ \mathtt{f}_n$", where n is the length of $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$ 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.

Let us introduce the main judgments of .FJ. The subtyping relation "$\Delta \vdash \mathtt{S<:T}$" is read `S` is a subtype of `T` under the bound environment Δ. The typing judgment "$\Delta; \Gamma; \mathtt{A} \vdash \mathtt{e:T}$" of an expression `e` is read an expression `e` has a type `T` under the bound environment Δ and the type environment Γ in the enclosing class `A`. Here, a type environment Γ is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$; a bound environment Δ is a finite mapping from type variables to their bounds (top-level classes), written $\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}$. The reduction relation is written $\mathtt{e} \longrightarrow \mathtt{e}'$. The type system of .FJ has been proved sound.

The rest of this section shows how to program extensible mutually recursive classes in lightweight family polymorphism.

Mutually recursive classes are declared as *nested classes* $\overline{\mathtt{NL}}$ in a top-level class `L`. The notion "family" refers to nested classes and its enclosing (top-level) class. 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 nested classes, *relative path types* `.C` are used for mutual references instead of *absolute nested class names* `C.C`, which are called fully qualified names in the Java terminology. A relative path type `.C`, read "dot `C`", refers to a nested class `C` in the same top-level class and its meaning will be changed when it is inherited to nested classes in a derived top-level class—relative path types in inherited members refer to one another among the derived family, as desired.

A simplified and informal .FJ example from [7] implementing graphs composed by nodes and edges, and graphs composed by colored nodes and weighted edges is:

```
class Graph {
  class Node { void add(.Edge e){ .. }
  class Edge { void connect(.Node n){ n.add(this); ..}}
class CWGraph ◁ Graph {
  class Node { Color color; }
  class Edge { int weight; void connect(.Node n){ n.color;..}}}
```

where `ColorWeightGraph` is abbreviated to `CWGraph`. Since the argument `n` of the overriding `connect()` is of relative path type `.Node`, the field `color` of `CWGraph.Node` can be accessed on `n` in `CWGraph.Edge` without typecasts. They would be necessary if these classes were written in Java (without generics) since, for type safety, method signatures cannot be changed covariantly when they are overridden, i.e., once a method has a parameter of `Graph.Node`, it cannot be overridden by `CWGraph.Node`. The self reference `this` of a nested class is given a relative path type, for example, `this` is of type `.Edge` in `Graph.Edge`, so it can be used as an argument to the method invocation `n.add()` in `connect()`. The reason why `this` has a relative path type is that the meaning of `this` changes in subclasses. The translation will reveal that this is an essential difference from the original FGJ.

Once variables are declared or objects are created with absolute nested class names $\mathtt{C}_1.\mathtt{C}_2$, relative path types in the member signatures are resolved to be absolute with the family names $\mathtt{C}_1$. For example, an object `new Graph.Edge(..)` has method `connect()` whose argument type is `Graph.Edge`. Subtyping between nested classes is not allowed for safety, for example, `CWGraph.Edge` cannot be used as `Graph.Edge`, even though there is an inheritance relation.

Family-polymorphic methods `M` overcome the inconvenience associated with the absence of subtyping. They have family parameters with their bounds $\mathtt{<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{C}}\mathtt{>}$, for example:

```
<G ◁ Graph>void m(G.Node n, G.Edge e){ e.connect(n); }.
```

This method can work uniformly over families that can instantiate `G`, such as `Graph` and `CWGraph`. For example, if `G` is instantiated with `Graph`, the method will accept objects of `Graph.Node` and `Graph.Edge`.

3 An Outline of the Translation

In this section, we describe the outline of the translation from .FJ to FGJ. On the way, we show that some translated programs are not well typed because of the different typing schema of .FJ and FGJ, and then we propose an extension of F-bounded polymorphism.

3.1 Basic Ideas of the Translation

Nested classes are translated to generic (top-level) classes and all inheritance relations are made explicit. Nothing except for types and class names will be changed. For instance, the nested classes `Graph.Node` and `CWGraph.Node` are translated to:

```
class Graph$Node<Node ◁ Graph$Node<Node,Edge>,
                 Edge ◁ Graph$Edge<Node,Edge>> {
  void add(Edge e){ .. }
}
class CWGraph$Node<Node ◁ CWGraph$Node<Node,Edge>,
                   Edge ◁ CWGraph$Edge<Node,Edge>>
          extends Graph$Node<Node,Edge>{ Color color; }
```

Here, `$` is used to make atomic names for generic classes. `Graph$Edge` and `CWGraph$Edge` are defined with the same parameterization as `Graph$Node` and `CWGraph$Node`, respectively. Relative path types are translated to their corresponding type variables (without "."). Note that the upper bounds of type variables are changed when moving to subfamilies so that the functionalities of mutual references are characterized correctly. For example, `color` can be accessed on `n`, of type `Node`, in `CWGraph$Edge` since the upper bound of `Node` is `CWGraph$Node`, which has `color`. Although classes are no longer nested, generic classes from a family form a group and they will work together since the F-bounded constraints in a generic class require other generic classes.

Since these generic classes cannot be instantiated, their non-generic subclasses, *fixed point classes*, need to be declared. They correspond to absolute class names in .FJ. We add a suffix `Fix` to generic class names to make the names for their fixed point classes. A fixed point class `Graph$NodeFix` extends `Graph$Node` and instantiates the type variables with itself and another fixed point class `Graph$EdgeFix`, defined simultaneously:

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

(The class body contains only constructors.) 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, `CWGraph$NodeFix` $\not<:$ `Graph$NodeFix`. This fact corresponds that in .FJ there is no subtyping between nested classes.

To translate a family-polymorphic method, each type `X.C` in the method signature is translated to a type variable `X$C`, in which `$` is also used to make an atomic name. Then, each of them will be F-bounded in the parameterization clause by using translated nested classes in `X`'s upper bound. For example, `m()` is translated to:

```
<G$Node◁Graph$Node<G$Node,G$Edge>,
 G$Edge◁Graph$Edge<G$Node,G$Edge>, G◁Graph>
void m(G$Node n, G$Edge e){ .. }
```

where `G.Node` and `G.Edge` are translated to `G$Node` and `G$Edge`, respectively. Note that `X$C` must be made for all nested classes `C` in `X`'s upper bound even if some of them do not appear in the method signature. For example, even if `m()` did not have `G.Edge` in the signature, the translated `m()` would have the same parameterization as above since `G$Edge` is required to instantiate the upper bound `Graph$Node` of `G$Node`. The actual type argument to a method invocation is translated similarly: if it is a top-level class, it will be translated to a set of fixed point classes from the top-level class; if it is a type variable `X`, it will be translated to a set of type variables of form `X$C`.

3.2 An Extension of F-bounded Polymorphism

Although most translated programs are well typed, unfortunately program fragments of a certain form will not be well typed after translation. This is because the way self references are typed in generic classes is different from that in the type system of .FJ. In this subsection, we examine this problem and propose an extension of F-bounded polymorphism in Java 5.0.

This problem occurs when translating a program in which `this` is passed to method parameters of relative path types. For instance, `n.add(this)` in `connect()` of `Graph$Edge` is not well typed, because the typing rules of Java generics give `this` type `Graph$Edge<N,E>`, which does not agree with the argument type `Edge`, which is a type variable. However, `this` is of relative path type in .FJ, so its type should translate to a type variable so that such translated method invocations type-check.

In this paper, we modify this design of typing by extending F-bounded polymorphism a little. Our proposal, on the one hand, allows the type of `this` in a generic class to be a type variable, found from the F-bounded constraints, just as the type of other mutual references. On the other hand, subclassing and instantiation of such a generic class will be limited 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 of `this`. For example, `this` has type `Edge` in `Graph$Edge`. As a result, `n.add(this)` in `connect()` is well typed. We call such a type variable *a self type variable*. For type safety, we allow a class with a self type variable to be either fixed or extended by another class with a self type variable, and prohibit creating its objects with any type instantiation. For example, `Graph$Edge<Graph$NodeFix,Graph$EdgeFix>` cannot be instantiated since we can think its self type is `Graph$EdgeFix`, which instantiates the self type variable `Graph$Edge`. If it were allowed to create its objects, invoking methods like `connect()` on them would result in an ill-typed expression, invoking `add()` expecting `Graph$EdgeFix`, the type argument passed to the self type, with an incompatible argument of type `Graph$Edge<Graph$NodeFix,Graph$EdgeFix>`.

4 Formalization

In this section, we formalize the extension of F-bounded polymorphism on top of Featherweight GJ [5] and prove its type system sound. Then, we formalize the translation from .FJ to the extended version of FGJ. Finally, we prove that the translation is correct with respect to typing and reduction.

4.1 Featherweight GJ with Self Type Variables

We give the formal definition of Featherweight GJ extended with self type variables.

Syntax:

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

Method Typing:

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

Class Typing:

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

Well-formedness:

$$
\frac{\mathtt{S}_i = \begin{cases}\mathtt{X}_j & \text{if } \mathtt{class}\ j\ \mathtt{C{<}\overline{X}{\lhd}\overline{N}{>}{\lhd}N\{..\}} \\ \mathtt{C{<}\overline{X}{>}} & \text{otherwise}\end{cases} \qquad \overline{\mathtt{X}}{<:}\overline{\mathtt{N}} \vdash \mathtt{D{<}\overline{S}{>}}\ \text{ok}}{\mathtt{C{<}\overline{X}{\lhd}\overline{N}{>}} \vdash \mathtt{D{<}\overline{S}{>}}\ \text{ok-superclass}}
$$

$$
\frac{\mathtt{class}\ i\ \mathtt{C{<}\overline{X}{\lhd}\overline{N}{>}{\lhd}N\{..\}} \qquad \mathtt{T}_i = \mathtt{X} \qquad \mathtt{T}_i{<:}\mathtt{C{<}\overline{T}{>}} \in \Delta \qquad \Delta \vdash \mathtt{C{<}\overline{T}{>}}\ \text{ok}}{\Delta;\mathtt{X} \vdash \mathtt{C{<}\overline{T}{>}}\ \text{ok-bound}}
$$

$$
\frac{\mathtt{class\ C{<}\overline{X}{\lhd}\overline{N}{>}{\lhd}N\{..\}} \qquad \Delta \vdash \mathtt{C{<}\overline{T}{>}}\ \text{ok}}{\Delta \vdash \mathtt{C{<}\overline{T}{>}}\ \text{ok-type}}
$$

Fig. 2. FGJ with self type variables: Syntax and selected well-formedness rules and typing rules for methods and classes.

Syntax. Figure 2 shows the syntax of FGJ with self type variables. The syntax is extended so that an optional number i is introduced after `class` in a class declaration. This i is used to indicate that the i-th type variable in the F-bounded constraints of a generic class is the self type variable. There are no other changes from the original 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.

Typing. Since the extension is small, most typing rules and judgments are the same as those in the original FGJ and we show only the important changes. The typing judgments "$\vdash$ `L` ok" for classes and "`C<`$\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}$`>;U` $\vdash$ `M` ok" for methods are slightly extended, and the well-formedness judgment for types are extended to three kinds: one "$\Delta \vdash$ `T` ok-type" for types, one "$\Delta;$`X` $\vdash$ `N` ok-bound" for upper bounds and one "`C<`$\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}$`>` $\vdash$ `N` ok-superclass" for superclasses. The typing judgment $\Delta;\Gamma \vdash$ `e:T` for expressions and the subtyping relation $\Delta \vdash$ `S<:T` are

the same as the original ones. We abbreviate a sequence of judgments in the following way: $\Delta \vdash \mathtt{T}_1$ ok-type, ..., $\Delta \vdash \mathtt{T}_n$ ok-type to $\Delta \vdash \overline{\mathtt{T}}$ ok-type; $\Delta; \mathtt{X}_1 \vdash \mathtt{T}_1$ ok-bound, ..., $\Delta; \mathtt{X}_n \vdash \mathtt{T}_n$ ok-bound to $\Delta; \overline{\mathtt{X}} \vdash \overline{\mathtt{T}}$ ok-bound; and $\mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>}; \mathtt{U} \vdash \mathtt{M}_1$ ok, ..., $\mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>}; \mathtt{U} \vdash \mathtt{M}_n$ ok to $\mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>}; \mathtt{U} \vdash \overline{\mathtt{M}}$ ok.

Figure 2 shows selected typing and well-formedness rules. The judgment "$\mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>}; \mathtt{U} \vdash \mathtt{M}$ ok" is read "a method declaration M in `class` i_{opt} $\mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>}\mathtt{\triangleleft N\{..\}}$ is *ok* provided that U is the self type." The type U is given by the typing rules for classes.

The judgment "$\vdash \mathtt{L}$ ok" is read "a class L is *ok*." The typing rule in Figure 2 is for a class with a self type variable. The type variable $\mathtt{X}_i$, which is indicated by i after `class` as the self type, appears the method typing judgment. For example, this typing rule can be applied to the class declaration of `Graph$Node`, with now the number `1` after `class`, in Section 3:

$$\frac{\begin{array}{r}\ldots \\ \mathtt{Graph\$Node<Node,\ Edge>} = \mathtt{Graph\$Node<Node,\ Edge>} \\ \mathtt{Graph\$Node<Node} \triangleleft \mathtt{Graph\$Node<Node,Edge>}, \\ \mathtt{Edge} \triangleleft \mathtt{Graph\$Edge<Node,Edge>>; Node} \vdash \overline{\mathtt{M}} \text{ ok}\end{array}}{\begin{array}{l}\vdash \mathtt{class\ 1\ Graph\$Node<Node} \triangleleft \mathtt{Graph\$Node<Node,Edge>}, \\ \qquad\qquad \mathtt{Edge} \triangleleft \mathtt{Graph\$Edge<Node,Edge>>\ \{..}\overline{\mathtt{M}}\mathtt{\}} \text{ ok}\end{array}}$$

It is checked if the first type variable `Node` is truly a self type variable as indicated by the number `1`. Then, the method declarations $\overline{\mathtt{M}}$ are checked with `Node` as the self type.

The difference of the well-formedness rules for types, upper bounds and superclasses can be seen in the rules for class types $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$— each rule has its own requirement to $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ besides the common requirement of "$\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ ok", whose rule is omitted from Figure 2, meaning that the type instantiation is correct. A superclass $\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}$ of $\mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>}$, where $\mathtt{class\ D<}\overline{\mathtt{Y}}\mathtt{\triangleleft}\overline{\mathtt{P}}\mathtt{>}$ is with a self type variable $\mathtt{Y}_i$, is well formed if $\mathtt{Y}_i$ is instantiated with either a self type variable $\mathtt{X}_j$ or the extending class $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$ itself. An upper bound $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ of X, where $\mathtt{class}\ i\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>\{..\}}$ has a self type variable $\mathtt{X}_i$, is well-formed under bound environment Δ if $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ is F-bounded on X, that is, $\mathtt{T}_i$ is X. A type $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ is well formed under bound environment Δ if $\mathtt{class\ C<}\overline{\mathtt{X}}\mathtt{\triangleleft}\overline{\mathtt{N}}\mathtt{>\{..\}}$ does not have a self type variable.

Other rules, omitted from Figure 2, for well-formedness are trivial: `Object` is always a well-formed type, upper bound and superclass. A type variable X is a well-formed type under bound environment Δ if X is in the domain of Δ.

Type Soundness of Featherweight GJ with Self Type Variables. The type system of FGJ with self type variables is sound with respect to the operational semantics, as usual. Type soundness is proved in the standard manner via subject reduction and progress [9, 5]. The reduction relation is written $\mathtt{e} \longrightarrow \mathtt{e}'$.

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

Theorem 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* $\mathtt{e}'$*.*

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

4.2 A Formal Translation from .FJ to FGJ with Self Type Variables

Figure 3 shows the definition of the formal translation.

Translation of Types. Translation $|\mathtt{T}|$ of type $\mathtt{T}$ 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 three rules in the second row show that the result is a type variable. Recall that `$` is another character used to make an atomic name.

Translation of Expressions. We define an auxiliary function $\mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}}$ to collect all names $\overline{\mathtt{E}}$ of nested classes in a family $\mathtt{C}$ including those of inherited ones. Translation $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}}$ of an expression $\mathtt{e}$ is defined with respect to bound environment Δ, type environment Γ and enclosing class $\mathtt{A}$. The interesting case is one for method invocations. It relies on translation $|\mathtt{F}|_{\mathtt{C}}$ of a family argument $\mathtt{F}$ with respect to the upper bound $\mathtt{C}$ of the corresponding formal, defined by using *nestedclasses*. For example, $|\mathtt{CWGraph}|_{\mathtt{Graph}}$ is `CWGraph$NodeFix`, `CWGraph$EdgeFix`, `CWGraph`. In translating type arguments $\overline{\mathtt{F}}$ in a method invocation $\mathtt{e}_0.\mathtt{<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}$, their upper bounds $\overline{\mathtt{C}}$ of the correspoding formal $\overline{\mathtt{X}}$ are obtained by using the function *mtype* to lookup the method signature of $\mathtt{m}$ in the type $\mathtt{T}_0$ of the receiver $\mathtt{e}_0$. The translation of an object creation requires the translation of the class name. The translations of variables and field accesses are straightforward.

Translation of Methods and Classes. The translation of methods and classes is a little involved due to the fact that inherited nested classes need not be redefined in a subfamily yet it is legal to mention an absolute type corresponding to such implicit classes. One way to deal with implicit classes in the translation is to generate dummy generic classes and their fixed point. To save the generation of these generic classes, we take another approach by introducing the notion of *ceilings* of absolute types. The formal definition of the ceiling $\lceil \mathtt{C.E} \rceil$ of $\mathtt{C.E}$ can be found at the bottom of the left column in Figure 3: the ceiling of $\mathtt{C.E}$ is the first superclass that appears explicitly. Ceilings are used where generic class names are required in the translated program, such as the translation of upper bounds. We cannot save, however, fixed point classes for these implicit classes since they are not substitutable for ones from another family as mentioned before. That is, `CWGraph$NodeFix` will be defined whether `CWGraph` has `Node` or not.

Type parameterization $|\mathtt{X} \triangleleft \mathtt{C}|$ in a method declaration is translated as follows. The family parameter and its upper bound are translated to a set of type variables whose upper bounds are generic classes, whose names are obtained by ceiling. Translation $|\mathtt{M}|_{\mathtt{A}}$ of a method declaration $\mathtt{M}$ in a class $\mathtt{A}$ is defined so

Translation of Types:

$$|\mathtt{C}| = \mathtt{C} \quad |\mathtt{C.E}| = \mathtt{C\$EFix}$$
$$|\mathtt{X}| = \mathtt{X} \quad |\mathtt{X.E}| = \mathtt{X\$E} \quad |\mathtt{.E}| = \mathtt{E}$$

Definition of *nestedclasses*

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

$$\frac{\mathtt{class\ C}\lhd\mathtt{D\{\ ..\ }\overline{\mathtt{NL}}\ \mathtt{\}} \quad nestedclasses(\mathtt{D}) = \overline{\mathtt{E}}'}{nestedclasses(\mathtt{C}) = \overline{\mathtt{E}}', \{\mathtt{E} \mid \mathtt{E} \notin \overline{\mathtt{E}}', \mathtt{class\ E\{..\}} \in \overline{\mathtt{NL}}\}}$$

Translation of Type Arguments:

$$\frac{nestedclasses(\mathtt{C}) = \overline{\mathtt{E}}}{|\mathtt{F}|_{\mathtt{C}} = |\mathtt{F.}\overline{\mathtt{E}}|, |\mathtt{F}|}$$

Translation of Expressions:

$$|\mathtt{x}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{x}$$

$$|\mathtt{e}_0\mathtt{.f}_i|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}\mathtt{.f}_i$$

$$\frac{\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad mtype_{.\mathrm{FJ}}(\mathtt{m}, bound_\Delta(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\to\mathtt{U}_0}{|\mathtt{e}_0\mathtt{.<}\overline{\mathtt{F}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}\mathtt{.<}|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}\mathtt{>m(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}}$$

$$|\mathtt{new\ A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{new}\ |\mathtt{A}_0|\mathtt{(}|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}}\mathtt{)}$$

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 $\mathtt{class\ C}\lhd\mathtt{D\{..}\overline{\mathtt{NL}}\mathtt{\}}$

Translation of Methods:

$$\frac{nestedclasses(\mathtt{C}) = \overline{\mathtt{E}}}{|\mathtt{X}\lhd\mathtt{C}| = |\mathtt{X.E}_1|\lhd\lceil\mathtt{C.E}_1\rceil\mathtt{<}|\mathtt{X.}\overline{\mathtt{E}}|\mathtt{>}, \ldots, |\mathtt{X.E}_n|\lhd\lceil\mathtt{C.E}_n\rceil\mathtt{<}|\mathtt{X.}\overline{\mathtt{E}}|\mathtt{>,X}\lhd\mathtt{C}}$$

$$\frac{\Gamma = \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}, \mathtt{this:}\mathit{thistype}(\mathtt{A}) \quad \Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}}{|\mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{C}}\mathtt{>}\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e}_0\mathtt{;\ \}}|_{\mathtt{A}} = \mathtt{<}|\overline{\mathtt{X}}\lhd\overline{\mathtt{C}}|\mathtt{>}|\mathtt{T}_0|\ \mathtt{m(}|\overline{\mathtt{T}}|\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow|\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}\mathtt{;\}}}$$

Translation of Classes:

$$\frac{nestedclasses(\mathtt{C}) = \overline{\mathtt{E}}}{|\lhd\mathtt{C}| = |\mathtt{.E}_1|\lhd\lceil\mathtt{C.E}_1\rceil\mathtt{<}|\mathtt{.}\overline{\mathtt{E}}|\mathtt{>}, \ldots, |\mathtt{.E}_n|\lhd\lceil\mathtt{C.E}_n\rceil\mathtt{<}|\mathtt{.}\overline{\mathtt{E}}|\mathtt{>}}$$

$$\frac{\mathtt{class\ C}\lhd\mathtt{D\{..\}} \quad nestedclasses(\mathtt{C}) = \overline{\mathtt{E}} \quad nestedclasses(\mathtt{D}) = \overline{\mathtt{E}}'}{|\mathtt{class\ E}_i\ \mathtt{\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}|_{\mathtt{C}} = \mathtt{class}\ i\ \mathtt{C\$E}_i\mathtt{<}|\lhd\mathtt{C}|\mathtt{>}\lhd\lceil\mathtt{D.E}_i\rceil\mathtt{<}|\mathtt{.}\overline{\mathtt{E}}'|\mathtt{>\{}|\overline{\mathtt{T}}|\ \overline{\mathtt{f}}\mathtt{;}|\overline{\mathtt{M}}|_{\mathtt{C.E}}\mathtt{\}}}$$

$$\frac{\mathtt{class\ C}\lhd\mathtt{D\{..\}} \quad nestedclasses(\mathtt{C}) = \overline{\mathtt{E}} \quad nestedclasses(\mathtt{D}) = \overline{\mathtt{E}}' \quad \mathtt{E} \notin \overline{\mathtt{E}}'}{|\mathtt{class\ E}_i\ \mathtt{\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}}|_{\mathtt{C}} = \mathtt{class}\ i\ \mathtt{C\$E}_i\mathtt{<}|\lhd\mathtt{C}|\mathtt{>}\lhd\mathtt{Object\{}|\overline{\mathtt{T}}|\ \overline{\mathtt{f}}\mathtt{;}|\overline{\mathtt{M}}|_{\mathtt{C.E}}\mathtt{\}}}$$

$$\frac{nestedclasses(\mathtt{C}) = \overline{\mathtt{E}}}{\mathit{fix}(\mathtt{C},\mathtt{E}) = \mathtt{class}\ |\mathtt{C.E}|\lhd\lceil\mathtt{C.E}\rceil\mathtt{<}|\mathtt{C.}\overline{\mathtt{E}}|\mathtt{>\{\}}}$$

$$|\mathtt{class\ C}\lhd\mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \overline{\mathtt{NL}}\mathtt{\}}| = \mathtt{class\ C}\lhd\mathtt{D}\ \mathtt{\{}|\overline{\mathtt{T}}|\ \overline{\mathtt{f}}\mathtt{;}|\overline{\mathtt{M}}|_{\mathtt{C}}\mathtt{\}}\ |\overline{\mathtt{NL}}|_{\mathtt{C}}\ \mathit{fix}(\mathtt{C}, nestedclasses(\mathtt{C}))$$

Fig. 3. Translation of types, expressions, methods and classes.

that the method body $\mathtt{e}_0$ is translated with respect to bound environment from the type parameterization, type environment from the formal parameters and enclosing class A. Note that *thistype*(C.E) = .E and *thistype*(C) = C in .FJ.

A set of F-bounded constraints $|\lhd\mathtt{C}|$ for nested classes in a top-level class C is defined similarly to $|\mathtt{X}\lhd\mathtt{C}|$. The difference is in how to make the names of type variables. A nested class NL in a top-level class C is translated to a class $|\mathtt{NL}|_{\mathtt{C}}$ with a self type variable, consisting of the translated fields and methods, by using $|\lhd\mathtt{C}|$. The number i indicates the position of the self type variable. If a nested class does not have a superclass, the superclass of the translated class will be Object, otherwise it will be a class with a self type variable.

A fixed point class *fix*(C, E) for an absolute class name C.E is an empty class that is assumed to have only a constructor declaration. The translation $|\mathtt{C}|$ of a top-level class C is a set of translated nested classes, the fixed point classes given by using *fix* and translated rests. Recall that fixed point classes are given for all nested classes whether they are implicitly inherited or explicitly redefined in subfamilies.

Properties of the Translation. Now, we prove that the translation preserves typing and reduction. The translation $|\Delta|$ of a bound environment Δ is defined similarly to $|\mathtt{X}\lhd\mathtt{C}|$.

Theorem 4 (Translation Preserves Typing). *If a .FJ class table CT is ok then, by using the typing rules of FGJ with self type variables $|CT|$ is ok and*

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

Theorem 5 (Translation Preserves Reduction). *If* $\Delta;\Gamma;\mathtt{A}\vdash\mathtt{e}:\mathtt{T}$ *and* $\mathtt{e}\longrightarrow\mathtt{e}'$ *then,* $|\mathtt{e}|_{\Delta,\Gamma,\mathtt{A}}\longrightarrow_{\mathrm{FGJ}}|\mathtt{e}'|_{\Delta,\Gamma,\mathtt{A}}$.

5 Related Work

Although we extended F-bounded polymorphism, there are workarounds to make all translated programs type-check without extending the target language. One is to use typecasts, but the type safety would be lost. The others are to introduce an expression that will refer to the same object as this, but whose type is a type variable. One presented in [8] is to introduce an extra argument which is assumed to accept the receiver object, for example:

```
void connect(Node n, Edge self) { n.add(self); }
e.connect(n, e);
```

Another one presented in [3] is to declare abstract methods which will be implemented in fixed point classes so that they simply return this, as follows:

```
class Graph$Edge<Edge ◁ ...>{
    abstract Edge getThis();
    void connect(Node n){ n.add(getThis()); }
```

```
}
class Graph$EdgeFix ◁ Graph$Edge<...>{
    Graph$EdgeFix getThis(){ return this; }
}
```

However, both change run-time behaivior because the former requires an evaluation of the extra argument and the latter requires an extra method invocation.

LOOJ [2] is another variant of Java 5.0 extended with a special 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[1], one may 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 it cannot appear in the upper bounds of type variables as the following code reveals:

```
class Graph$Node<Edge◁Graph$Edge<ThisClass>> {..} // Not allowed
class Graph$Edge<Node◁Graph$Node<ThisClass>> {..} // Not allowed
```

In Scala [6], we can give self references arbitrary types explicitly. When objects are created, it is checked if the classes being instantiated are subtypes of their explicit self types. This mechanism is much more powerful than our extension of F-bounded polymorphism, in which self types are limited to type variables.

6 Conclusion

We have shown that the formal translation from .FJ to a variant of FGJ with the extension of F-bounded polymorphism, namely self type variables. The type system of FGJ with self type variables has been proved sound and the correctness of the translation has been proved, too.

The translation has clarified that nested classes, relative path types and family-polymorphic methods can be considered a convenient syntax sugar for Java 5.0 in which a lot of complicated parameterizations would be required, and that absolute class names let us to create objects without defining another set of classes, namely fixed point classes.

The extension of F-bounded polymorphism captures the essential difference between lightweight family polymorphism and Java generics. By allowing the self type in a generic class to be a type variable found in the F-bounded constraints, it is possible to treat `this` as one of mutual references with a suitable type without using any extra workarounds. On the other hand, its subclassing and instantiation are limited for safety.

We conclude that lightweight family polymorphism provides not only a set of convenient notations but also a bit more suitable type system than Java 5.0 for extensible mutually recursive classes.

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

Acknowledgements. We thank the anonymous referees of FTfJP 2007 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.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. Martin Odersky and Matthias Zenger. Scalable component abstraction. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57, San Diego, CA, October 2005.
7. Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 2007. To appear. 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.
8. 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.
9. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A The Definition of Featherweight GJ with Self Type Variables

In this section, we show the changes from the original definition of Featherweight GJ [5] in the typing rules for methods and classes, and type well-formedness. We replace type well-formedness "$\Delta \vdash \mathtt{T}$ ok" by "$\Delta \vdash \mathtt{T}$ ok-type" in the typing rules for expressions. Other rules and relations are the same.

Well-formed Instantiation:

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

Well-formed Superclasses:

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

$$\frac{\mathtt{class}\ i\ \mathtt{D<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>\lhd P\{..\}} \quad \mathtt{S}_i = \begin{cases} \mathtt{X}_j & \text{if } \mathtt{class}\ j\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd N\{..\}} \\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} & \text{otherwise} \end{cases} \quad \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}} \vdash \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok}}{\mathtt{C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-superclass}} \quad \text{(WFS-ClassSelf)}$$

$$\frac{\mathtt{class\ D<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>\lhd P\{..\}} \quad \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}} \vdash \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok}}{\mathtt{C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok-superclass}} \quad \text{(WFS-Class)}$$

Well-formed Upper bounds:

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

$$\frac{\mathtt{class}\ i\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd N\{..\}} \quad \mathtt{T}_i = \mathtt{X} \quad \mathtt{T}_i\mathtt{<:C<}\overline{\mathtt{T}}\mathtt{>} \in \Delta \quad \Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}{\Delta; \mathtt{X} \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-bound}} \quad \text{(WFB-ClassSelf)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd N\{..\}} \quad \Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}{\Delta; \mathtt{X} \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok-bound}} \quad \text{(WFB-Class)}$$

Well-formed Types:

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

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

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>}\lhd\texttt{N\{..\}} \qquad \Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{ok}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{ok-type}} \tag{WFT-Class}$$

Method Typing:

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

Class Typing:

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

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