Lightweight Scalable Components

Tetsuo Kamina
The University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo,
113-0033, Japan
kamina@acm.org

Tetsuo Tamai
The University of Tokyo
3-8-1, Komaba, Meguro-ku, Tokyo,
153-8902, Japan
tamai@acm.org

Abstract

One limitation of the well-known family polymorphism approach is that each “family” will be a large monolithic program. In this paper, we introduce a minimal lightweight set of languages that treat each member of a family as a reusable programming unit, while preserving the important feature of scalability. The only language construct we propose in this paper is type parameter members, which allows type parameters to be referred from the outside of class declarations. To investigate properties of type parameter members in the real programming language settings, we develop a programming language Scalable Java (SJ), an extension of Java generics with type parameter members. To carefully investigate the type soundness ofSJ, we develop FGJ#, a core calculus of this extension based on FGJ, a functional core of Java with generics. Furthermore, to explore how to implement this proposal, we define the erasure of FGJ# programs as an extension of the erasure of FGJ programs, which compiles SJ to Java without generics.

Categories and Subject Descriptors D.1.5 [Programming Techniques]: Object-Oriented Programming; D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages

Keywords Scalable Java, Type parameter members, Parametric polymorphism, Family polymorphism, FGJ

1. Introduction

How to construct reusable software components has been one of the most significant challenges in the research field of programming languages. Most modern object-oriented languages with simple name-based type systems such as Java and C# have been equipped with simple linguistic means to define and extend/modify software components, e.g., classes and inheritance. Furthermore, there has been much work to improve modularity, reusability, and scalability of the existing languages. In this paper, we focus on two significant requirements for components: reusability and scalability.

An important requirement for components is that they be reusable. Objects sometimes have complicated interactions with one another and cannot be reused independently. To make software components reusable, some details of components should be abstracted to make them applicable in contexts other than the one in which they have been developed. For example, many research efforts have been devoted to investigate how to design parametric polymorphism in object-oriented languages[3, 24, 11, 2, 1]. In such languages, types may be abstracted from the definitions of classes and methods. However, some other important structures such as nested classes, inner classes, etc. cannot be abstracted, thus nested classes are not reusable programming units.

This restriction becomes a serious problem when we consider the collaboration based implementation such as family polymorphism and other approaches [8, 12, 21, 30, 18, 5, 23]. Family polymorphism satisfies the important requirement of scalability; i.e., in family polymorphism, each set of mutually recursive classes, namely a family, may be safely extended without modification of the existing source code. However, there is one shortcoming in this approach; since mutually recursive classes are programmed as nested class members of a top-level class, each family becomes a large monolithic program. Therefore, implementations of all the members of one family are placed in a single source file, thus this approach is not based on components; each member’s implementation cannot be reused in the different context, and cannot be developed in parallel.

In summary, we require a mechanism that treats each member of family as a reusable programming unit, while preserving the important feature of scalability.

In this paper, we introduce a minimal lightweight set of language features to solve this problem. The idea is to abstract nested classes from the definition of outer classes by using type parameters. Since nested classes are parameterized, the actual implementations for them are not given in the outer class definition. Syntactically, the only one new language construct we propose is type parameter members, that allows type parameters to be referred from the outside of class declarations, using the notation T#X, where T is a type and X is a type parameter declared in (possibly some super class of) T (if T is a type parameter, some super class of the type bound of T). In our approach, the mutually recursive classes are placed in separate source files, solving the problem that the family polymorphism approach faced.

We show that this approach is scalable. One problem is that the type of T#X is “hard-linked” to the enclosing class. To tackle this problem, we define a type inference algorithm that infers the possible extensions of type of T#X. With this feature, the safe extension of mutually recursive classes that is supported by family polymorphism is also possible in our approach.

To investigate properties of type parameter members in the real programming language settings, we develop a programming language Scalable Java (SJ), an extension of Java generics that is equipped with type parameter members. It is a quite simple extension but has remarkable expressive power. To carefully investigate...
have different names, even though their structures are similar.

Since the semantics of each nested class member is not hard-linked to the enclosing class, we may safely extend the mutually recursive classes.

The family polymorphism approach is very powerful; however, it has one shortcoming. Since mutually recursive classes are programmed as nested class members of a top-level class, each family becomes a large monolithic program. Both definitions of Edge and Node are placed in the same source file. Therefore, the family polymorphism approach is not based on components; when there are many kinds of Edge and Node, then it will be convenient if we may compose them as we want, but the family polymorphism approach does not provide any flexible ways for it. Therefore, we identify the following requirements:

Separation of members of family. Each member of family may be placed in a separate source file from that declares the family.

Furthermore, as in the most modern typed programming languages, we also require the following properties:

Type safety. Extensions cannot create run-time errors.

Modularity. Extensions cannot require modification or recompilation of the existing systems.

Non-destructive extension. Each extension may co-exist in the same system.

Lightweight extension. We would not require completely a new programming language. Furthermore, new language constructs that are added to the existing language should be very simple.

2.2 Solution Using SJ

2.2.1 Basic Strategy

We show how Scalable Java (SJ) fulfills the aforementioned requirements. The basic strategy is to abstract each member of family from the family by representing them using type parameters. However, we have to take into account that each member is a mutually recursive class. To refer to another member, it must be able to identify the partner class belonging to the same family. Since member’s implementation is separated from the implementation of family, the member has to know to which family it belongs. Therefore, each member also has to be parameterized over its belonging family. Furthermore, to identify the partner class, which is represented as a type parameter in the implementation of family, there needs to be a mechanism that allows type parameters to be referred from the outside of family declaration.

Current object-oriented languages does not provide such a mechanism. We propose a new language construct type parameter members that provides such a feature. We use the notation T#X, where T is a type and X is a type parameter declared in (possibly some super class, or type bound of) T. In the following subsections, we show how this construct solves the problems by examples.

2.2.2 Simple Graph Family

Figure 2 shows the definition of the family Graph using SJ. It declares type parameters E and N that correspond to the members representing edges and nodes, respectively. It also declares an instance variable na that represents a set of node instances in the graph. A class Node is a concrete implementation of a member of Graph that represents nodes in a graph, and a class Edge is a concrete implementation of a member of Graph that represents edges in a graph. Both of them are parameterized over its belonging family by type parameter G, whose upper bound is Graph. Node declares an instance variable ea, which represents a set of edges where this node is connected. For the future extensibility, the type of edge is not hard-linked to Edge; instead, it is declared as a type parameter member G#E, where E is a type parameter that is declared.
Figure 3 shows the definition of family

2.2.3 Extending the Base Family

Note that in this case we may instantiate type parameter members, of

Graph. As in Graph, it also declares type parameters E and N. In this case, the type parameters declared in a subclass override the parameters declared in superclass. Upper bounds for type parameters E and N are covariantly refined to WeightEdge and RichNode, respectively.

RichNode provides an interface for WeightEdge to provide the property of the node through the abstract method value(). A class ColorNode is one of the concrete implementations of RichNode. Similarly, a class WeightEdge is a concrete implementation of a member of WGraph that refines the definition of Edge. Both of them also override the type parameter G that are declared in their superclasses. Since type parameter members are used in the base classes to refer to each other, in the extensions, we can refer to each member of the extended family even when declarations in the base classes are evaluated. For example, a method connect declared in WeightEdge overrides connect declared in Edge, because each of formal parameter types is declared as G#N, and G and N are appropriately overridden.

Finally, as in the case of SimpleGraph, we introduce a fixed-point class ColorWeightGraph to complete the definition of WGraph. We may instantiate each member of WGraph as follows:

Figure 4 shows that there may be another implementation of nodes in WGraph other than ColorNode, namely LabelNode, which is also declared as subclass of RichNode. The fixed-point class LabelWeightGraph demonstrates that LabelNode is also safely composed with WGraph. This modification is local to implementation of nodes; since each implementation is provided in a separate class, it does not affect development of other parts of the graph application.
Furthermore, such extensions may also be used in the original base Graph. For example, someone may require that there needs a graph where each node is colored, but the weight feature on edges is not needed. We may obtain such a graph by composing ColorNode and Edge with Graph:

```java
class ColorGraph extends Graph<Edge<ColorGraph>,
    ColorNode<ColorGraph>> {
}
```

### 2.3 Rules Ensuring Type Safety

So far, we have shown the main aspects of SJ. The remaining important issue is its type safety. Actually, there are some challenges in making SJ type system sound. In this section, we overview these challenges and how we tackle the problems.

#### 2.3.1 Reduction of Type Parameter Members

The first subtlety is that, if type parameter members are introduced, there may be multiple representations for one type. To show this fact, let us consider Figure 2 again. In this example, a generic class Node declares a type parameter G whose upper bound is Graph<G,E,G,N>. On the other hand, in the declaration of Graph, Node is instantiated by assigning Graph<E,N> to the type parameter. Therefore, the following subtyping relation must be satisfied:

```
Graph<E,N> <: Graph<Graph<E,N>,Graph<E,N>,Graph<E,N>>
```

Actually, Graph<E,N> is the same type as E, thus the above subtype relation is satisfied in SJ. In general, a type parameter member is the same type as the corresponding argument for the type parameter. In the semantic analysis of SJ programs, every type parameter member is reduced to its argument. This reduction is performed whenever the subtype relation is checked in the type checking phase of compilation. The formal semantics for this type reduction is given in section 3.

#### 2.3.2 Type Inference for this

Some thoughtful readers may also wonder how the program shown in Figure 2 is type checked, because there seems to be a mismatch between the expected type of this and its actual type. In the semantics of Java, the type of this is its enclosing class. The type of an argument for the method call s.add(this) in Figure 2 is therefore Edge<Graph, where E is a subtype of Graph<G,E,G,N>>. On the other hand, the formal parameter type of the add method is declared as Graph<E,N>. The problem is that Edge<Graph,> and Edge<Graph,E,N> are actually not compatible.

Apparently, this program does not produce any run-time errors, because run-time type of this is compatible to Edge<Graph,E,N>. Therefore, we need to provide the programmers with some means to tell the compiler this fact. There are some ways to do so. For example, we may introduce a new kind of types like thisType[4, 7] to SJ. Another possible approach is to provide some means to programmers to explicitly declare the actual type of this, like the self type annotation of Scala[25]. However, both of those approaches require significant language extensions.

On the other hand, using upper bounds of type parameters, the actual type of this may be inferred. SJ takes this approach. We informally describes type inference rules for this as follows:

- If one of the type parameters, namely X, has an upper bound that is the enclosing class, the type of this inside this class is X; i.e., in the following class declaration,
  ```java
class C<.., X extends C<?,X,...>, ..> { .. }
```
  the type of this inside C is X.
- If one of the type parameters, namely X, has an upper bound that is a class that declares a type parameter, namely Y, whose upper bound is the enclosing class of X, the type of this inside this class is X#Y; i.e., in the following class declaration,
  ```java
class C<X extends C<X#Y,> { .. }
```
  where
  ```java
class D<X extends C<X#Y,> { .. }
```
  the type of this inside C is X#Y.
- Otherwise, the type of this is its enclosing class.

In the case of program shown in Figure 2, Edge declares a type parameter G whose upper bound is Graph<G,E,G,N>, and Graph declares a type parameter E whose upper bound is Edge<Graph<E,N>>. Therefore, the type of this used inside Edge is G#E, which is the same as the formal parameter type of add method. Thus, the program shown in Figure 2 is safely type checked (see lemma 3.4 in section 3).

There may be ambiguity in the above algorithm. For example, if there is the following class declaration,

```java
class C<X extends C<X,Y>,
    Y extends <C,YY> { .. }
```

the type of this may be both of X and Y. To avoid such ambiguity, the current version of SJ does not proceed type inference if there are multiple paths for type inference. Therefore, in the above case the type of this will be C<X, Y>.

#### 2.3.3 Some notes on subtyping

In Figure 3, ColorWeightNode is not a subtype of SimpleGraph, because the former is not a subclass of the latter. On the other hand, we could declare ColorWeightGraph to be a subtype of SimpleGraph as follows:

```java
class WGraph<E extends WeightEdge<WGraph<E,>>,
    N extends RichNode<WGraph<E,>>,
    ... class ColorWeightGraph extends
        WGraph<WeightEdge<ColorWeightGraph>,
        ColorNode<ColorWeightGraph>> { }
```

In this case, ColorWeightGraph is a subclass of SimpleGraph, so the former is a subtype of the latter.

Does our system allow covariant subtyping regarding type parameter members, i.e., is ColorWeightGraph#N a subtype of SimpleGraph#N? If this subtyping is allowed, the type system will not be safe. For example, consider the following demonstration code:

```java
ColorWeightGraph g1 = new ColorWeightGraph();
SimpleGraph g2 = g1;
g2.add(new FooGraph#N());
```

In this example, we add an instance of FooGraph#N, which is a subtype of SimpleGraph#N and does not provide a method named value(), to the instance variable ns of g1, which is actually expected to provide value().

SJ does not allow this subtyping, because type reduction is performed whenever subtype relation is checked; type reduction of ColorWeightGraph#N is RichNode<ColorWeightGraph>, while type reduction of SimpleGraph#N is Node<SimpleGraph>, and these types are not compatible. The same observation is found
3. FGJ#: A Tiny Core of Scalable Components

In this section, we formalize the ideas described in the previous section as a small calculus named FGJ# based on Featherweight GJ (FGJ) [16], a functional core of class-based object-oriented languages with the feature of generics.

3.1 Syntax

The abstract syntax of FGJ# is given in Figure 5. The metavariables T, S, V, U, and Q range over types; X, Y, Z, and W range over type variables; N and P range over nonvariable types; C, D, and E range over class names; L ranges over class declarations; M ranges over constructor declarations; R ranges over method declarations; f and g range over field names; m ranges over method names; x and y range over variables; e and d range over expressions.

We write f as a shorthand for a possibly empty sequence f₁, ..., fₙ, and N as a shorthand for N₁, ..., Nₙ. Furthermore, we abbreviate pairs of sequences in a similar way, writing “(f₁, ..., fₙ)” as a shorthand for “(f₁)”, “(f₁, f₂)” as a shorthand for “((f₁), f₂)”, “(f₁, ..., fₙ)” as a shorthand for “(f₁, ..., fₙ)”, and so on. We write the empty sequence as - and the length of sequence f as #(f). Sequences of type variables, field declarations, parameter names, and method declarations are assumed to contain no duplicate names.

In syntax, the only difference between FGJ# and FGJ is that FGJ# has type constructor T#X. As in FGJ, we abbreviate the keyword extends to the symbol <. We assume that the set of variables includes the special variable this, which is considered to be implicitly bound in every method declaration. FGJ# supports polymorphic methods, and type parameters of method invocation are explicitly provided with the form e. m<T>(x).

Thus, FGJ# is a purely functional calculus.

Subclassing in FGJ#, also shown in Figure 5, represented by the relation C ⊑ D between class names, is a reflexive and transitive closure induced by the clause C< X > < N > ⊑ C< N > < D >.< T >.< E >.

An FGJ# program is a pair (C<T>, e) of a class table C<T> and an expression e. A class table is a map from class names to class declarations. The expression e may be considered as the main method of the real functional program. The class table is assumed to satisfy the following conditions: (1) C<T>(C) = class C ... for every C ∈ dom(C<T>); (2) Object ̸∈ dom(C<T>); (3) C ∈ dom(C<T>) for every class name appearing in ran(C<T>); (4) there are no cycles in subclass relation induced by C<T>. In the induction hypothesis shown below, we abbreviate C<T>(C) = class C ... as class C ....

3.2 Auxiliary definitions

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 6 and 7. The function fields(N) is a sequence of field types and names declared in N. Application of type substitution [T/N] is defined in the customary manner. Type of the method invocation m at N, written mtype(m,N), is a type of the form <X ⊑ N>U. The body of the method invocation m at N, written mbody(m,N), is a pair, written x,e, of a sequence of parameters x and an expression e. Upper bound of type variable X in N, written
Type reduction:

\[
\begin{align*}
\text{reduce}(\texttt{C<T>}) &= \texttt{C<T>} \\
\text{reduce}(X) &= X \\
\text{reduce}(X\#Y) &= X\#Y \\
\text{reduce}(\texttt{C<T>\#X}) &= \text{reduce}(\texttt{T}_i)
\end{align*}
\]

Type inference for this:

\[
\begin{align*}
\text{class } \texttt{C<X <\#N> <\{ \cdots \} > N}_i = \texttt{C<T>}, \text{ } T_i = X_i \\
\forall j, \neg j \neq i, N_j \neq \texttt{C<T>}
\end{align*}
\]

\[
\text{thistype}(\texttt{C<X>}) = X_i
\]

\[
\begin{align*}
\text{class } \texttt{C<X <\#N> <\{ \cdots \} > N}_i = \texttt{D<T>}, \text{ } T_j = X_i\#Y_j \\
\forall k, \neg k \neq i, N_k \neq \texttt{D<T>}
\end{align*}
\]

\[
\text{thistype}(\texttt{C<X>}) = T_i
\]

\[
\begin{align*}
\text{class } \texttt{C<X <\#N> <\{ \cdots \} > N}_i = \texttt{C<T>}, \text{ } T_i = X_i \\
\forall j, \neg j \neq i, N_j \neq \texttt{C<T>}
\end{align*}
\]

\[
\text{thistype}(\texttt{C<T>}) = C<T>
\]

Figure 7. Type reduction and type inference for this.

\[
\text{bound}(X, \mathcal{N}), \text{ is a type that appears right hand side of } <\text{ whose left hand side is } X.
\]

Rules of type reduction and type inference of this are shown in Figure 7. The function reduce performs type reduction that reduces type parameter members to their actual (argument) types. The result of reduction is further reduced until it reaches a base case of reduction rules. Base cases are types that are not type parameter members, or type parameter members on type parameters.

The function thistype returns the inferred type of this using the upper bounds of type parameters. Note that, to prove the type soundness theorem, we have to define thistype of type instantiation besides three cases explained in section 2.3.2.

3.3 Typing

An environment \( \Gamma \) is a finite mapping from variables to types, written \( \bar{x} : \text{T}. \) A type environment \( \Delta \) is a finite mapping from type variables to nonvariable types, written \( \bar{X} : \mathcal{N}. \)

As defined in Figure 8, we write \( \text{bound}_\Delta(T) \) for an upper bound of \( T \) in \( \Delta. \) Besides type parameters and nonvariable types, we also need to define \( \text{bound}_\Delta \) of type parameter member \( T\#X. \) The auxiliary function \( \text{bound}, \) which is defined in Figure 6, is used in the definition of \( \text{bound}_\Delta. \)

The subtyping relation \( \Delta \vdash T < : S, \) read as \( \text{"T is a subtype of S in } \Delta, \)" is also defined in Figure 8. As in FGI, subtyping is the reflexive and transitive closure of the extenda relation, and type parameters are invariant with regard to subtyping. If the upper bound of type parameter member \( T\#X \) is \( X, \) then \( T\#X \) is a subtype of \( X. \)

We write \( \Delta \vdash T \text{ ok} if a type T is well formed in context \( \Delta. \) The rules for well-formed types appear in Figure 9. A type \( \texttt{C<T>}, \) is well formed if a class declaration that begins with \( \texttt{class } C<X <\#N> <\{ \cdots \} > N \) \) exists in \( CT, \) substituting \( \bar{T} \) for \( \bar{X} \) respects the bounds \( \mathcal{N} \), and all of \( T \) are ok. A type parameter member \( T\#X \) is well-formed if the upper bound of \( T \) is defined, and there is a super class, namely \( C, \) of it that declares the type parameter \( X. \) Before subtyping is judged, type reduction rules shown in Figure 7 are applied to both sides of \( <: \) (we write \( \Delta \vdash \text{reduce}(T) < : \text{reduce}(S) \) as a shorthand for \( \Delta \vdash \text{reduce}(T_1) < : \text{reduce}(S_1) \cdots ) \) if a type \( \Delta \vdash T \text{ ok } \) for all \( x \) in \( \text{dom} (\Delta) \).

We say that a type environment \( \Gamma \) is well-formed if \( \Delta \vdash \Delta \) ok for all \( X \) in \( \text{dom} (\Gamma). \) We also say that an environment \( \Gamma \) is well-formed with respect to \( \Delta. \) written \( \Delta \vdash \Gamma \text{ ok } \) if \( \Delta \vdash \Gamma (x) \text{ ok } \) for all \( x \) in \( \text{dom} (\Gamma). \)

Figure 9 also shows rules that allow covariant overriding on the method result type, which is ensured by \( \text{override}(\mathcal{N}, \mathcal{N}, \bar{X} < \bar{X} : \text{T} \rightarrow T_\text{0}). \) Note also that covariant overriding of type parameters is allowed, which is ensured by \( \text{override}(\bar{X}, \bar{L}). \)

Typing rules for expressions, methods, and classes are defined in Figure 10. The typing judgment for expressions is of the form \( \Delta ; \Gamma \vdash e : \text{T}, \) read as “in the type environment \( \Delta \) and the environment \( \Gamma, \) the expression \( e \) has type \( \text{T}. \)” The typing rules are syntax directed, with one rule for each form of expression, save that there are three rules for typecasts. As in FGI, in the rule T-DCAST, dcast(C,C,D) defined in Figure 9 ensures that the result of the cast will be the same at runtime. Note that before subtyping is judged, type reduction is applied to both sides of \( <. \) Note also that we may instantiate a nonvariable type \( T \) providing the type reduction of \( T \) is a nonvariable type.

1To employ the algorithmic typing style[27], we explicitly apply the function reduce to the subtype judgments appearing on typing rules, instead of introducing one extra subtyping rule for type reduction. This approach simply yields a type checking algorithm.
Well-formed types:

\[
\begin{align*}
\Delta \vdash \text{Object } ok & \quad \text{(WF-OBJECT)} \\
\Delta \vdash x ok & \quad \text{(WF-VAR)} \\
\Delta \vdash \text{reduce}(T) \langle : \text{reduce}([T/X][N]) & \quad \text{(WF-CLASS)} \\
\Delta \vdash\text{new } T \langle \text{\(\bar{x}\)} & \quad \text{T-DOT)}
\end{align*}
\]

Valid downcast:

\[
\begin{align*}
dcast(C, D) & \quad \text{dcast}(D, E) \quad dcast(C, E) \\
dcast(C, D) & \quad \text{dcast}(D, E) \quad dcast(C, E)
\end{align*}
\]

Valid method overriding:

\[
\begin{align*}
\text{override}(m, N, \langle Y < P \rangle T) & \quad \text{reduce}(T) \langle : \text{reduce}(\Delta) \langle T_0 \rangle \\
\text{override}(m, N, \langle Y < P \rangle T) & \quad \text{reduce}(T) \langle : \text{reduce}(\Delta) \langle T_0 \rangle
\end{align*}
\]

Valid type overriding:

\[
\begin{align*}
\text{override}(X, C\langle X < N > \langle N \ldots \rangle) & \quad \text{reduce}(P) \\
\text{override}(X, C\langle X < N > \langle N \ldots \rangle) & \quad \text{reduce}(P)
\end{align*}
\]

Expression typing:

\[
\begin{align*}
\Delta; \Gamma \vdash x : \Gamma(x) & \quad \text{(T-VAR)} \\
\Delta; \Gamma \vdash e_0 : T_0 & \quad \text{fields(bound}_{\Delta}(T_0)) = \bar{T} \bar{x} \\
\Delta; \Gamma \vdash e_0.f ; T_i & \quad \text{(T-FIELD)} \\
\Delta; \Gamma \vdash e_0 : T_0 & \quad \text{mtype}(\bar{e}, \text{bound}_{\Delta}(T_0)) = \langle \bar{Y} < P \rangle U \rightarrow U \\
\Delta; \Gamma \vdash \bar{Y} ok & \quad \text{reduce}(\bar{Y}) \langle : \text{reduce}([\bar{Y}/Y][P]) \\
\Delta; \Gamma \vdash e ; S & \quad \Delta; \Gamma \vdash \text{reduce}(\bar{S}) \langle : \text{reduce}([\bar{Y}/Y][U]) \\
\Delta; \Gamma \vdash e_0.m < P > (\bar{e}) : [\bar{Y}/Y][U] & \quad \text{(T-INVK)} \\
\Delta; \Gamma \vdash T ok & \quad \text{fields(reduce}(T)) = \bar{T} \bar{x} \\
\Delta; \Gamma \vdash \text{new } T \langle \bar{e} \rangle & \quad \text{T-DOT)}
\end{align*}
\]

Method typing:

\[
\begin{align*}
\Delta = \bar{x} \langle: \bar{Y} \langle: \bar{P} & \quad \Delta; \Gamma \vdash T, P ok \\
\Delta; \bar{x} : T, \text{this}: \text{thistype}(C\langle X \rangle) & \quad e_0 : S \\
\Delta; \Gamma \vdash \text{reduce}(S) \langle : \text{reduce}(T) \\
\text{class } C\langle X < N > \langle N \ldots \rangle & \quad \text{override}(m, N, \langle Y < P \rangle T) \rightarrow T) \\
\langle Y < P \rangle & \quad \text{T-METHOD)
\end{align*}
\]

Class typing:

\[
\begin{align*}
\bar{x} \langle: \bar{Y} \langle: \bar{P} & \quad \langle Y < P \rangle T m(\bar{t} \bar{x}) \{ \text{return } e_0; \} \quad \text{OK IN } C\langle X < N \rangle \rightarrow T) \\
\text{class } C\langle X < N > \langle N \ldots \rangle & \quad \text{T-CLASS)
\end{align*}
\]

Figure 9. FGJ# type well-formedness rules

The typing judgment for method declarations, which has the form \(N \text{ OK IN } C\), read “method declaration \(N\) is ok when it occurs in class \(C\),” uses the expression typing judgment on the body of the method, where the free variables are the parameters and the method with their declared types and the special variable \(\text{this}\). The type of \(\text{this}\) is inferred by using function \(\text{thistype}\). Covariant overriding of methods on the method result type is also allowed in FGJ#.

The typing judgment for class declarations, which has the form \(C \text{ OK},\) read “class declaration \(C\) is ok,” checks that the constructor is well-defined and that each method declaration in the class is ok. Furthermore, it allows covariant overriding of type parameters.

A class table \(CT\) is ok if all its definitions are ok.

3.4 Reduction

The operational semantics of FGJ# is defined with the reduction relation that is of the form \(e \rightarrow e'\), read “expression \(e\) reduces
Computation:

\[ \text{fields(reduce}(T)) = \bar{T} \bar{f} \]  
(R-FIELD)

\[ \text{mbody}(\text{m}<\bar{V}>), \text{reduce}(T)) = \bar{x}.e_0 \]  
(R-INVK)

\[ (\text{new } \bar{T}(\bar{a})).m<\bar{V}> \bar{d} \rightarrow [\bar{d}/\bar{x}] \text{new } \bar{T}(\bar{a})/\text{this}/e_0 \]  
(R-CAST)

Congruence:

\[ e_0 \rightarrow e_0' \]  
(RC-FIELD)

\[ e_0.\bar{f} \rightarrow e_0'.\bar{f} \]  
(RC-ARG)

\[ e_0.\bar{m}<\bar{T}>\bar{e}(\bar{a}) \rightarrow e_0.\bar{m}<\bar{T}>\bar{e}(\bar{a}) \]  
(RC-RECV)

\[ e_i \rightarrow e_i' \]  
(RC-ARG)

\[ e_i.\bar{m}<\bar{T}>\bar{e}(\bar{a}) \rightarrow e_i.\bar{m}<\bar{T}>\bar{e}(\bar{a}) \]  
(RC-ARG)

\[ \bar{T}e_0 \rightarrow \bar{T}e_0' \]  
(RC-CAST)

Lemma 3.4. If class \( C<\bar{x} < \bar{N}> < \bar{R} \{ \cdots \} \), then \( \bar{x}.<\bar{N}> \vdash \text{thistype}(C<\bar{x}>)<\bar{N}>. \)

With these lemmas, we can prove the following important lemmas:

Lemma 3.5 (Type Substitution Preserves Subtyping). If \( \Delta_1, \bar{x}.<\bar{N}> \vdash \bar{S} < \bar{T} \) and \( \Delta_1 \vdash \bar{U} < [\bar{U}]/\bar{X}]/\bar{U} \) with \( \Delta_1 \vdash \bar{U} \ ) ok, and none of \( \bar{x} \) appearing in \( \Delta_1 \), then \( \Delta_1, [\bar{U}]/\bar{X}/\Delta_2 \vdash [\bar{U}]/\bar{X}/\bar{S} < [\bar{U}]/\bar{X}/\bar{T} \).

Proof. By induction on the derivation of \( \Delta_1, \bar{x}.<\bar{N}> ; \Delta_2 \vdash \bar{S} < \bar{T} \) using Lemma 3.1.

Lemma 3.6 (Type Substitution Preserves Type Well-Formedness). If \( \Delta_1, \bar{x}.<\bar{N}> \vdash \bar{T} \ ) ok and \( \Delta_1 \vdash \bar{U} < [\bar{U}]/\bar{X}]/\bar{U} \) with \( \Delta_1 \vdash \bar{U} \ ) ok and none of \( \bar{x} \) appearing in \( \Delta_1 \), then \( \Delta_1, [\bar{U}]/\bar{X}/\Delta_2 \vdash [\bar{U}]/\bar{X}/\bar{T} \ ) ok.

Proof. By induction on the derivation of \( \Delta_1, \bar{x}.<\bar{N}> ; \Delta_2 \vdash \bar{T} \ ) ok using Lemma 3.2.

Lemma 3.7 (Type Substitution Preserves Typing). If \( \Delta_1, \bar{x}.<\bar{N}> \vdash \bar{T} \ ) ok and \( \Delta_1 \vdash \bar{U} < [\bar{U}]/\bar{X}]/\bar{U} \) with \( \Delta_1 \vdash \bar{U} \ ) ok and none of \( \bar{x} \) appearing in \( \Delta_1 \), then \( \Delta_1, [\bar{U}]/\bar{X}/\Delta_2 \vdash [\bar{U}]/\bar{X}/\bar{T} \).

Proof. By induction on the derivation of \( \Delta_1, \bar{x}.<\bar{N}> ; \Delta_2 \vdash \bar{T} \ ) ok using Lemmas 3.5 and 3.6.

Lemma 3.8 (Term Substitution Preserves Typing). If \( \Delta; \Gamma, \bar{x} : \bar{T} \vdash \bar{e} ; \Delta; \Gamma; \bar{d} : \bar{S} \ ) where \( \Delta \vdash \bar{S} < \bar{T} \), then \( \Delta; \Gamma; \bar{d} : \bar{S} \ ) for some \( \bar{S} \) such that \( \bar{S} < \bar{T} \).

Proof. By induction on the derivation of \( \Delta; \Gamma; \bar{x} : \bar{T} \vdash \bar{e} \) using Lemma 3.5.

Lemma 3.9. If \( \text{mtype}(\bar{m}, \bar{C}<\bar{T}>; \bar{m}<\bar{V}>), \bar{C}<\bar{T}>; \bar{m}<\bar{V}> \) = \( \bar{x}.\bar{e}_0 \) where \( \Delta \vdash \bar{C}<\bar{T}> \) ok and \( \Delta \vdash \bar{V} \ ) ok and \( \Delta \vdash \bar{V} < [\bar{V}]/\bar{Y}]/\bar{V} \), then there exist some \( \bar{V} \) and \( \bar{U} \) such that \( \Delta \vdash \bar{C}<\bar{T}> < \bar{R} \) and \( \Delta \vdash \bar{S} < \bar{T} \) and \( \Delta \vdash \bar{S} < \bar{T} \) for some \( \bar{T} \) such that \( \bar{T} < \bar{S} \).

Proof. By induction on the derivation of \( \bar{m} < \bar{V}> \) using Lemmas 3.8 and 3.9.

The proof of the following progress theorem is easy.

Theorem 3.1 (Subject Reduction). If \( \Delta; \Gamma; \bar{x} : \bar{T} \vdash \bar{e} ; \Delta; \Gamma; \bar{d} : \bar{S} \ ) for some \( \bar{T} \) such that \( \bar{T} < \bar{S} \).

Proof. By induction on the derivation of \( \bar{e} \vdash \bar{e}' \) using Lemmas 3.4, 3.5, and 3.7.

Then, we can prove the following subject reduction theorem.

Theorem 3.2 (Progress). Suppose \( \bar{e} \) is a well-typed expression.

1. If \( \bar{e} \) includes \( \text{new } \bar{N}_0(\bar{a}) \) as a subexpression, then fields(\( \bar{N}_0 \)) = \( \bar{T} \ ) and \( \bar{f} \in \bar{f} \) for some \( \bar{T} \) and \( \bar{f} \).

2. If \( \bar{e} \) includes \( \text{new } \bar{N}_0(\bar{a}) \) as a subexpression, then \( \text{mbody}(\bar{m}<\bar{V}>), \bar{C}<\bar{T}>; \bar{m}<\bar{V}> \) = \( \bar{x}.\bar{e}_0 \) and \( \#(\bar{x}) = \#(\bar{a}) \) for some \( \bar{x} \) and \( \bar{e}_0 \).

To state FGI# type soundness formally, we give the definition of FGI# value below:

\[ \bar{v} := \text{new } \bar{N}(\bar{r}) \]

Finally, we get the FGI# type soundness theorem.

Theorem 3.3 (FGI# Type Soundness). If \( \emptyset \vdash : \bar{e} : \bar{T} \) and \( \bar{e} \vdash \bar{e}' \) with \( \bar{e}' \) a normal form, then \( \bar{e}' \) is either (1) an FGI# value with \( \emptyset ; \emptyset \vdash \bar{v} ; \bar{S} \) and \( \emptyset \vdash \bar{S} < \bar{T} \) or (2) an expression containing \( \bar{F} \) new \( \bar{N}(\bar{e}) \) where \( \emptyset \vdash \bar{N} < \bar{F} \).

Proof. Immediate from Theorems 3.1 and 3.2.
We may also show that if an expression \( e \) is cast-safe in \( \Delta; \Gamma \) (i.e., the type derivations of the underlying \( CT \) and \( \Delta; \Gamma \vdash e : T \) do not use \text{T-DCAST} and \text{T-SCAST} rules), it does not produce any typecast errors.

**Theorem 3.4.** If \( e \) is cast-safe in \( \emptyset; \emptyset \) and \( e \longrightarrow^* e' \) with \( e' \) a normal form, then \( e' \) is a value \( v \).

### 4. Implementation

As in GJ, our SJ compiler translates source code of SJ to Java (without generics), which maintains no information about type parameters at runtime. We model the implementation of SJ by extending the existing framework of implementation of erasure translation from FGJ to FJ[16]. Due to limited space, we briefly describe how SJ programs are erased to Java programs by examples. Formalization and studies on properties of this translation are remained in future work.

Like the erasure algorithm of FGJ, an SJ program is erased by replacing types with their erasures, inserting downcasts where required. In the erasure algorithm of FGJ, a type is erased by removing type parameters, and replacing type parameters with the erasures of their bounds. However, there are two difficulties when applying this mechanism to the erasure of SJ: (1) we need to change the definition of erasure of types; (2) we need more synthetic casts.

To show how our erasure semantics is changed from that of FGJ, let us consider the following example. In the erasure algorithm of FGJ, the erasure of class \( \text{Edge} \) shown in Figure 2 will be as follows,

```java
class Edge {
    Node src, dst;
    void connect(Node s, Node d) {
        src = s;
        dst = d;
        s.add(this);
        d.add(this);
    }
}
```

and the erasure of class \( \text{WeightEdge} \) shown in Figure 3 will be as follows, because type parameters are replaced with the erasures of their bounds:

```java
class WeightEdge extends Edge {
    int weight;
    void connect(Node s, Node d) {
        int sv = ((ColorNode)s).value();
        int dv = ((ColorNode)d).value();
        super.connect(s, d);
    }
}
```

This erasure is a well-typed Java program that preserves the behavior of the original SJ program.

### 5. Related Work

Virtual classes[20, 17, 14], also known as path-dependent types [13, 25], are similar to type parameter members in that types are declared as fields of classes and they are reffered from the outside of class declarations as instance variables. In our proposal, however, types are referred as static members of classes, which is necessary to express F-bounded polymorphism such as \text{class Node<G extends Graph<G#E,G#N>}} \{ ... \}.

Type parameter members partially supports family polymorphism [12, 18]. The difference is, while the original family polymorphism uses the same name as each member of the original family to that of the extended family, in our approach, we have to invent new names for members in the extended family. Furthermore, programs written in the pure family polymorphism approaches tend to be less verbose than ours, where we need to provide type bounds to type parameter members and define fixed point classes. On the other hand, in our approach, each member of family may be placed in the separate source file.

Originally, nested classes in family polymorphism are members of an object, thus the language essentially involves a dependent type system. On the other hand, based on the observation given by [19], Igarashi et al. propose a much simpler variant of family polymorphism, in which families are identified with classes[18]. Even though this approach sacrifices some important features of the original family polymorphism (e.g. each member of family may not access the instance of the enclosing class), the resulting calculus is quite compact and reasonably expressive. Our approach is similar to this approach, in that type parameter members are also static members of enclosing class. There are also some other related work on family polymorphism: [21, 22, 26, 15, 8, 5].

Besides path-dependent types, Scala[25] also supports symmetric mixin compositions and self-type annotations. Self-type annotations allow us to explicitly declare the type of \text{this}. With these features, our programming style where members of families may be placed in the separate source files is also possible in Scala. Actually, Scala has much expressive power that SJ does not support. On the other hand, SJ is a quite simple extension of Java 5.0 that has reasonable expressive power. Since its difference from Java is very small, it will be easy to develop tools and refactor the existing programs.

\text{MyType} (or sometimes called \text{ThisType}) is also studied to solve the mismaching problem of recursive class definitions [6, 7]. \text{MyType} is the type of \text{this} and changes its meaning along the inheri-
tance chain. Lately, this idea is extended to mutually recursive class definitions [8, 5] by introducing constructs to group mutually recursive definitions. These approaches are similar to us in the sense that dependent types are not used. However, like other approaches, the resulting group will be a large monolithic program. Another important difference is, in these work, covariant subtyping of members is allowed. To ensure type safety, they introduce the notion of exact types and allow us to invoke a method that take an argument of the same family only when the receiver’s family is exactly known.

The type inference rules for this are also independently invented in [28], where translation from lightweight family polymorphism [18] to Featherweight GJ with a little extension of F-bounded polymorphism is considered.

6. Concluding Remarks

We have proposed a new language construct named type parameter members and the language SJ, an extension of Java generics that is equipped with type parameter members. With this construct, we may abstract nested classes from the definition of outer class. With the mechanism of type inference of this, mutually recursive classes can be safely scaled without modification of existing source code, and these classes may be placed in the separate source files, solving the problem of family polymorphism approach in which each family becomes a large monolithic program. The core language FGJ# ensures type soundness of the proposed language, and each family becomes a large monolithic program. The core language FGJ# ensures type soundness of the proposed language, and the description of the erasure algorithm provides useful information for language implementation.

This approach cannot abstract inner classes (not nested classes) from the definition of outer class. In the body of Node, e.g., we may not access the enclosing instance. We feel that how to support this feature is worth pursuing. Another direction for future work will be applying the proposed mechanism to mixin layers approaches [29, 10]. Since mixins provides convenient means for abstracting super classes from class declarations, such extension will provide much stronger support for component based programming.

References