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 language features that treat each member of a family as a reusable programming unit, while preserving the important feature of scalability. The only one 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 of SJ, 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 parametrized, 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 `this` 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 `this`. 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

GPCE'07, October 1–3, 2007, Salzburg, Austria.
Copyright © 2007 ACM 978-1-59593-855-8/07/0010…$5.00

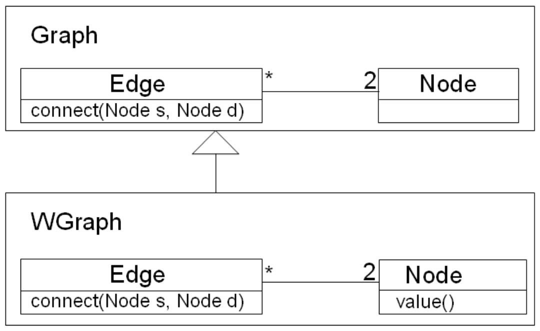

Figure 1. Overview of our example

type soundness of SJ, we develop FGJ#, a core calculus of this extension based on FGJ, a functional core of Java with generics. Moreover, 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.

The rest of this paper is structured as follows. Section 2 describes the features of SJ, and how the aforementioned problems are solved using SJ. In section 3, we formalize the proposed language features by extending FGJ, and show the proposal is type sound. In section 4, we define the erasure of FGJ# to show how the proposed language is implemented. In section 5, we discuss how this work is related to other researches. Finally, section 6 concludes this paper.

2. Programming Lightweight Scalable Components

2.1 A Graph Example

We start by informally describing the main aspects of type parameter members, the language construct we study in this paper. To show how this construct is used to support family polymorphic extension, we consider an example originally presented in [12].

We illustrate this example in Figure 1. This example features a *family* (or group) `Graph`, containing the *members* of the family, namely `Node` and `Edge`. In our example, each instance of `Node` holds a reference to connected edges (instances of `Edge`), and each edge holds references to its source and destination nodes. Thus, the definitions of `Node` and `Edge` are mutually recursive.

Then, we extend `Graph` to `WGraph` that adds the feature of setting weight of each edge. In `WGraph`, the member `Edge` is refined to store the weight of this edge. The member `Node` is also refined to store its property. This property may be color, or label etc.; in Figure 1, `Node` declares an abstract method `value()` to return an integer value of this property. In our application, the weight of edge is calculated by using this property. To do so, the `connect` method declared in `Edge` is appropriately overridden.

Even though this example is simple, there exist many challenges in it. At first, extending a set of classes that are mutually recursive is very difficult in the existing object-oriented languages. In such languages, mutually recursive classes refer to each other by their *names*, thus different sets of mutually recursive classes necessarily have different names, even though their structures are similar.

Considerable research efforts including family polymorphism have been recently devoted to solve this problem [8, 12, 21, 30, 18, 5, 23]. In family polymorphism, like virtual methods, a reference to a nested class member is resolved at run-time, thus the meaning of mutual references to class names will change when a subclass of the enclosing class is derived and those member classes are inherited. 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 parametrized 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 `ns` 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 parametrized over its belonging family by type parameter `G`, whose upper bound is `Graph`. `Node` declares an instance variable `es`, 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

```
class Graph<E extends Edge<Graph<E,N>>,
            N extends Node<Graph<E,N>>> {
  Vector<N> ns = new Vector<N>();
  void add(N n) { ns.add(n); }
}
class Node<G extends Graph<G#E,G#N>> {
  Vector<G#E> es = new Vector<G#E>();
  void add(G#E e) { es.add(e); }
}
class Edge<G extends Graph<G#E,G#N>> {
  G#N src, dst;
  void connect(G#N s, G#N d) {
    src = s;
    dst = d;
    s.add(this);
    d.add(this); }
}
class SimpleGraph
      extends Graph<Edge<SimpleGraph>,
                    Node<SimpleGraph>> { }
```

Figure 2. Simple graph definitions

in the upper bound of `G`. Similarly, the type of instance variables `src` and `dst` declared in `Edge` that represent source and destination nodes respectively, and formal parameter types in `connect` are also declared as a type parameter member `G#N`.

We do not have to invent a new language construct for composition of family members. Since each type parameter `E`, `N`, and `G` appears on both sides of `extends` clause, i.e. we use F-bounded polymorphism[9], we have to *fix* the definition of `Graph`, `Node`, and `Edge` by introducing the fixed-point class `SimpleGraph`, which will be used to build graph instances. The following is a demonstration program:

```
SimpleGraph g = new SimpleGraph();
SimpleGraph#N n1 = new SimpleGraph#N();
SimpleGraph#N n2 = new SimpleGraph#N();
SimpleGraph#E e1 = new SimpleGraph#E();
g.add(n1); g.add(n2); e1.connect(n1,n2);
```

Note that in this case we may instantiate type parameter members, because their actual types are exactly known (see section 2.3.1).

2.2.3 Extending the Base Family

Figure 3 shows the definition of family `WGraph` that is an extension 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 superclasses. 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.

```
// Colored and weighted graph extensions
class WGraph<E extends WeightEdge<WGraph<E,N>>,
             N extends RichNode<WGraph<E,N>>>
      extends Graph<E,N> { }
abstract
class RichNode<G extends Graph<G#E,G#N>>
      extends Node<G> {
  abstract int value();
}
class ColorNode<G extends Graph<G#E,G#N>>
      extends RichNode<G> {
  Color color;
  int value() { ... }
}
class WeightEdge<G extends WGraph<G#E,G#N>>
      extends Edge<G> {
  int weight;
  int f(G#N s, G#N d) {
    int sv = s.value(); int dv = d.value();
    ... }
  void connect(G#N s, G#N d) {
    weight = f(s, d);
    super.connect(s, d); }
}
class ColorWeightGraph extends
      WGraph<WeightEdge<ColorWeightGraph>,
             ColorNode<ColorWeightGraph>> { }
```

Figure 3. Weighted graph extension

```
class LabelNode<G extends Graph<G#E,G#N>>
      extends RichNode<G> {
  Label label;
  int value() { ... }
}
class LabelWeightGraph extends
      WGraph<WeightEdge<LabelWeightGraph>,
             LabelNode<LabelWeightGraph>> { }
```

Figure 4. Another weighted graph extension

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:

```
ColorWeightGraph g = new ColorWeightGraph();
ColorWeightGraph#E e1 = new ColorWeightGraph#E();
ColorWeightGraph#N n1 = new ColorWeightGraph#N();
ColorWeightGraph#N n2 = new ColorWeightGraph#N();
g.add(n1); g.add(n2); e1.connect(n1,n2);
```

2.2.4 Replacing Implementations of Members

We then show that our approach enables much flexible composition that is not supported by the family polymorphism approach. Since each member is placed in a separate class, we may implement many kinds of members, and we may combine them as we want.

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`:

```
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>#E,Graph<E,N>#N>
```

Actually, `Graph<E,N>#E` 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<G>`, where `G` is some subtype of `Graph<G#E,G#N>`. On the other hand, the formal parameter type of the `add` method is declared as `G#E`, where `E`'s upper bound is `Edge<Graph<E,N>>`. The problem is that `Edge<G>` 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 *MyType* (also known as `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,

  ```
  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,

  ```
  class C<X extends D<X#Y>> { .. }
  ```

  where

  ```
  class D<Y extends C<D<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,

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

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, `ColorWeightGraph` 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:

```
class WGraph<E extends WeightEdge<WGraph<E,N>>,
             N extends RichNode<WGraph<E,N>>>
      extends SimpleGraph { }
...
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:

```
ColorWeightGraph g1 = new ColorWeightGraph();
SimpleGraph g2 = g1;
g2.add(new FooGraph#N());
// FooGraph is another subtype of SimpleGraph
g1.ns.elementAt(0).value(); // error!
```

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

Syntax:

$$\begin{array}{lcl}
\texttt{T} & ::= & \texttt{X} \mid \texttt{N} \mid \texttt{T\#X} \\
\texttt{N} & ::= & \texttt{C<}\bar{\texttt{T}}\texttt{>} \\
\texttt{L} & ::= & \texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{ }\bar{\texttt{T}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{ \}} \\
\texttt{K} & ::= & \texttt{C(}\bar{\texttt{T}}\ \bar{\texttt{f}}\texttt{) \{ super(}\bar{\texttt{f}}\texttt{); this.}\bar{\texttt{f}}\texttt{=}\bar{\texttt{f}}\texttt{; \}} \\
\texttt{M} & ::= & \texttt{<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{> T m(}\bar{\texttt{T}}\ \bar{\texttt{x}}\texttt{) \{ return e; \}} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m<}\bar{\texttt{T}}\texttt{>(}\bar{\texttt{e}}\texttt{)} \mid \texttt{new T(}\bar{\texttt{e}}\texttt{)} \mid \texttt{(T)e}
\end{array}$$

Subclassing:

$$\texttt{C} \trianglelefteq \texttt{C} \qquad \frac{\texttt{C} \trianglelefteq \texttt{D} \quad \texttt{D} \trianglelefteq \texttt{E}}{\texttt{C} \trianglelefteq \texttt{E}}$$

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{D<}\bar{\texttt{T}}\texttt{>\{...\}}}{\texttt{C} \trianglelefteq \texttt{D}}$$

Figure 5. FGJ# syntax and subclassing

in [18], in which the policy that states inheritance of family members is not subtyping is taken. There are other approaches that allow such kind of subtyping but introduce a notion of *exact types* to ensure type safety (e.g., [8]), which is discussed in section 5.

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; `K` ranges over constructor declarations; `M` 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 $\bar{\texttt{f}}$ as a shorthand for a possibly empty sequence $\texttt{f}_1 \cdots \texttt{f}_n$, and $\bar{\texttt{M}}$ as a shorthand for $\texttt{M}_1 \cdots \texttt{M}_n$. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\bar{\texttt{T}}\ \bar{\texttt{f}}$" as a shorthand for "$\texttt{T}_1\ \texttt{f}_1, \ldots, \texttt{T}_n\ \texttt{f}_n$," "$\texttt{this.}\bar{\texttt{f}}\texttt{=}\bar{\texttt{f}}\texttt{;}$" as a shorthand for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\ldots\texttt{;this.f}_n\texttt{=f}_n\texttt{;}$", "$\bar{\texttt{X}} \lhd \bar{\texttt{N}}$" as a shorthand for "$\texttt{X}_1 \lhd \texttt{N}_1, \cdots, \texttt{X}_n \lhd \texttt{N}_n$", and so on. We write the empty sequence as $\cdot$ and the length of sequence $\bar{\texttt{f}}$ as $\#(\bar{\texttt{f}})$. 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 $\lhd$. 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 for generic method invocation are explicitly provided with the form $\texttt{e.m<}\bar{\texttt{T}}\texttt{>(}\bar{\texttt{e}}\texttt{)}$. A class must declare only one constructor that initializes all the fields of that class. A constructor declaration is only the place where assignment operator is allowed; once initialized, an instance never change its state. Method body consists of single `return` statement. Thus, FGJ# is a purely functional calculus.

Subclassing in FGJ#, also shown in Figure 5, represented by the relation $\texttt{C} \trianglelefteq \texttt{D}$ between class names, is a reflexive and transitive closure induced by the clause $\texttt{C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>} \lhd \texttt{D<}\bar{\texttt{T}}\texttt{>}$.

An FGJ# program is a pair $(CT,\texttt{e})$ of a class table CT 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 SJ program. The class table is assumed to satisfy the following conditions: (1) $CT(\texttt{C}) = \texttt{class C ...}$ for every $\texttt{C} \in \mathit{dom}(CT)$; (2) $\texttt{Object} \notin \mathit{dom}(CT)$; (3) $\texttt{C} \in \mathit{dom}(CT)$ for every class name appearing in $\mathit{ran}(CT)$; (4) there are no cycles in subclass relation induced by CT.

In the induction hypothesis shown below, we abbreviate $CT(\texttt{C}) = \texttt{class C ...}$ as `class C ...`.

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \cdot \qquad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \mathit{fields}([\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{N}) = \bar{\texttt{U}}\ \bar{\texttt{g}}}{\mathit{fields}(\texttt{C<}\bar{\texttt{T}}\texttt{>}) = \bar{\texttt{U}}\ \bar{\texttt{g}}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\bar{\texttt{S}}\ \bar{\texttt{f}}} \qquad \text{(F-CLASS)}$$

Method type lookup:

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \\ \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{> U m(}\bar{\texttt{U}}\ \bar{\texttt{x}}\texttt{) \{ return e; \}} \in \bar{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = [\bar{\texttt{T}}/\bar{\texttt{X}}](\texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \rightarrow \texttt{U})} \qquad \text{(MT-CLASS)}$$

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{m} \notin \bar{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{N})} \qquad \text{(MT-SUPER)}$$

Method body lookup:

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \\ \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{> U m(}\bar{\texttt{U}}\ \bar{\texttt{x}}\texttt{) \{ return e}_0\texttt{; \}} \in \bar{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = \bar{\texttt{x}}.[\bar{\texttt{T}}/\bar{\texttt{X}}, \bar{\texttt{V}}/\bar{\texttt{Y}}]\texttt{e}_0} \qquad \text{(MB-CLASS)}$$

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{m} \notin \bar{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = \mathit{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{N})} \qquad \text{(MB-SUPER)}$$

Type bound lookup:

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}}}{\mathit{bound}(\texttt{X}_i, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{N}_i}$$

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{N \{}\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K }\bar{\texttt{M}}\texttt{\}} \qquad \texttt{Y} \notin \bar{\texttt{X}}}{\mathit{bound}(\texttt{Y}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = \mathit{bound}(\texttt{Y}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{N})}$$

Figure 6. FGJ# lookup functions

3.2 Auxiliary definitions

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

Type reduction:

$$\mathit{reduce}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}) = \mathtt{C<}\bar{\mathtt{T}}\mathtt{>}$$

$$\mathit{reduce}(\mathtt{X}) = \mathtt{X} \qquad\qquad \mathit{reduce}(\mathtt{X\#Y}) = \mathtt{X\#Y}$$

$$\frac{\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\ \{\ \cdots\ \}}{\mathit{reduce}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>\#X}_i) = \mathit{reduce}(\mathtt{T}_i)}$$

$$\frac{\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\ \{\ \cdots\ \} \qquad \mathtt{Y} \notin \bar{\mathtt{X}}}{\mathit{reduce}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>\#Y}) = \mathit{reduce}([\bar{\mathtt{T}}/\bar{\mathtt{X}}]\mathtt{N\#Y})}$$

Type inference for `this`:

$$\frac{\begin{array}{c}\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\ \{\ \cdots\ \} \qquad \mathtt{N}_i = \mathtt{C<}\bar{\mathtt{T}}\mathtt{>} \qquad \mathtt{T}_i = \mathtt{X}_i \\ \forall j, j \neq i, \mathtt{N}_j \neq \mathtt{C<}\bar{\mathtt{T}}\mathtt{>}\end{array}}{\mathit{thistype}(\mathtt{C<}\bar{\mathtt{X}}\mathtt{>}) = \mathtt{X}_i}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\{\ \cdots\ \} \qquad \mathtt{N}_i = \mathtt{D<}\bar{\mathtt{T}}\mathtt{>} \qquad \mathtt{T}_j = \mathtt{X}_i\mathtt{\#Y}_j \\ \forall k, k \neq i, \mathtt{N}_k \neq \mathtt{D<}\bar{\mathtt{T}}\mathtt{>} \\ \mathtt{class\ D<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\lhd\{\ \cdots\ \} \qquad \mathtt{P}_j = \mathtt{C<}\bar{\mathtt{S}}\mathtt{>} \qquad \mathtt{S}_i = \mathtt{D<}\bar{\mathtt{U}}\mathtt{>} \\ \mathtt{U}_j = \mathtt{Y}_j \qquad \forall k, k \neq j, \mathtt{P}_k \neq \mathtt{C<}\bar{\mathtt{S}}\mathtt{>}\end{array}}{\mathit{thistype}(\mathtt{C<}\bar{\mathtt{X}}\mathtt{>}) = \mathtt{X}_i\mathtt{\#Y}_j}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\ \{\ \cdots\ \} \qquad \mathtt{N}_i = \mathtt{C<}\bar{\mathtt{T}}\mathtt{>} \qquad \mathtt{T}_i = \mathtt{X}_i \\ \forall j, j \neq i, \mathtt{N}_j \neq \mathtt{C<}\bar{\mathtt{T}}\mathtt{>}\end{array}}{\mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}) = \mathtt{T}_i}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\{\ \cdots\ \} \qquad \mathtt{N}_i = \mathtt{D<}\bar{\mathtt{T}}\mathtt{>} \qquad \mathtt{T}_j = \mathtt{X}_i\mathtt{\#Y}_j \\ \forall k, k \neq i, \mathtt{N}_k \neq \mathtt{D<}\bar{\mathtt{T}}\mathtt{>} \\ \mathtt{class\ D<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\lhd\{\ \cdots\ \} \qquad \mathtt{P}_j = \mathtt{C<}\bar{\mathtt{S}}\mathtt{>} \qquad \mathtt{S}_i = \mathtt{D<}\bar{\mathtt{U}}\mathtt{>} \\ \mathtt{U}_j = \mathtt{Y}_j \qquad \forall k, k \neq j, \mathtt{P}_k \neq \mathtt{C<}\bar{\mathtt{S}}\mathtt{>}\end{array}}{\mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}) = \mathtt{T}_i\mathtt{\#Y}_j}$$

$$\frac{\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\ \{\ \cdots\ \}}{\mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}) = \mathtt{C<}\bar{\mathtt{T}}\mathtt{>}}$$

Figure 7. Type reduction and type inference for `this`

$\mathit{bound}(\mathtt{X}, \mathtt{N})$, is a type that appears right hand side of $\lhd$ 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 Γ is a finite mapping from variables to types, written $\bar{\mathtt{x}} : \bar{\mathtt{T}}$. A type environment Δ is a finite mapping from type variables to nonvariable types, written $\bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}}$.

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

Bound of type:

$$\begin{array}{rcl}\mathit{bound}_\Delta(\mathtt{X}) & = & \Delta(\mathtt{X}) \\ \mathit{bound}_\Delta(\mathtt{N}) & = & \mathtt{N} \\ \mathit{bound}_\Delta(\mathtt{T\#X}) & = & \mathit{bound}(\mathtt{X}, \mathit{bound}_\Delta(\mathtt{T}))\end{array}$$

Subtyping:

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

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

$$\frac{\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C<}\bar{\mathtt{T}}\mathtt{>} \mathrel{\mathtt{<:}} [\bar{\mathtt{T}}/\bar{\mathtt{X}}]\mathtt{N}} \qquad \text{(S-CLASS)}$$

$$\frac{\mathit{bound}_\Delta(\mathtt{T\#X}) = \mathtt{N}}{\Delta \vdash \mathtt{T\#X} \mathrel{\mathtt{<:}} \mathtt{N}} \qquad \text{(S-DOT)}$$

Figure 8. FGJ# subtyping rules

The subtyping relation $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$, read "`S` is a subtype of `T` in Δ," is also defined in Figure 8. As in FGJ, subtyping is the reflexive and transitive closure of the `extends` relation, and type parameters are *invariant* with regard to subtyping. If the upper bound of type parameter member `T#X` is `N`, then `T#X` is a subtype of `N`.

We write $\Delta \vdash \mathtt{T}\ ok$ if a type `T` is well formed in context Δ. The rules for well-formed types appear in Figure 9. A type $\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}$ is well formed if a class declaration that begins with $\mathtt{class\ C<}\bar{\mathtt{X}}\lhd\bar{\mathtt{N}}\mathtt{>}$ exists in CT, substituting $\bar{\mathtt{T}}$ for $\bar{\mathtt{X}}$ respects the bounds $\bar{\mathtt{N}}$, and all of $\bar{\mathtt{T}}$ 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 $\Delta \vdash \mathit{reduce}(\bar{\mathtt{T}})\mathtt{<:}\mathit{reduce}(\bar{\mathtt{S}})$ as a shorthand for $\Delta \vdash \mathit{reduce}(\mathtt{T}_1)\mathtt{<:}\mathit{reduce}(\mathtt{S}_1) \cdots \Delta \vdash \mathit{reduce}(\mathtt{T}_n)\mathtt{<:}\mathit{reduce}(\mathtt{S}_n)$).

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

Figure 9 also shows rules that allow covariant overriding on the method result type, which is ensured by $\mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\bar{\mathtt{T}} \rightarrow \mathtt{T}_0)$. Note also that covariant overriding of type parameters is allowed, which is ensured by $\mathit{override}(\mathtt{X}, \mathtt{L})$.

Typing rules for expressions, methods, and classes are defined in Figure 10. The typing judgment for expressions is of the form $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$, read as "in the type environment Δ and the environment Γ, the expression `e` has type `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 FGJ, in the rule T-DCAST, *dcast*(`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 `<:`[1]. Note also that we may instantiate a nonvariable type `T` providing the type reduction of `T` is a nonvariable type.

[1] To 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:

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

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

$$\frac{\begin{array}{c}\mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}N\ \{\ldots\}}\\ \Delta \vdash \bar{\mathtt{T}}\ ok \qquad \Delta \vdash reduce(\bar{\mathtt{T}})\ \mathtt{<:}\ reduce([\bar{\mathtt{T}}/\bar{\mathtt{X}}]\bar{\mathtt{N}})\end{array}}{\Delta \vdash \mathtt{C{<}\bar{T}{>}}\ ok} \qquad \text{(WF-CLASS)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}N\ \{\ldots\}}\\ bound_{\Delta}(\mathtt{T}) = \mathtt{D{<}\bar{T}{>}} \qquad \mathtt{D} \trianglelefteq \mathtt{C} \qquad \Delta \vdash \mathtt{T}\ ok\end{array}}{\Delta \vdash \mathtt{T\#X}_i\ ok} \qquad \text{(WF-DOT)}$$

Valid downcast:

$$\frac{dcast(\mathtt{C},\mathtt{D}) \qquad dcast(\mathtt{D},\mathtt{E})}{dcast(\mathtt{C},\mathtt{E})}$$

$$\frac{\begin{array}{c}\mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}D{<}\bar{T}{>}\ \{\ldots\}}\\ \bar{\mathtt{X}} = FV(\bar{\mathtt{T}})\end{array}}{dcast(\mathtt{C},\mathtt{D})}$$

($FV(\bar{\mathtt{T}})$ denotes the set of type variables in $\bar{\mathtt{T}}$)

Valid method overriding:

$$\frac{\begin{array}{l}mtype(\mathtt{m},\mathtt{N}) = \mathtt{{<}\bar{Z} \lhd \bar{Q}{>}\bar{U}} \rightarrow \mathtt{U}_0 \text{ implies}\\ \bar{\mathtt{P}},\bar{\mathtt{T}} = [\bar{\mathtt{Y}}/\bar{\mathtt{Z}}](\bar{\mathtt{Q}},\bar{\mathtt{U}}) \text{ and } \bar{\mathtt{Y}}\mathtt{<:}\bar{\mathtt{P}} \vdash reduce(\mathtt{T}_0)\ \mathtt{<:}\ reduce([\bar{\mathtt{Y}}/\bar{\mathtt{Z}}]\mathtt{U}_0)\end{array}}{override(\mathtt{m},\mathtt{N},\mathtt{{<}\bar{Y} \lhd \bar{P}{>}\bar{T}} \rightarrow \mathtt{T}_0)}$$

Valid type overriding:

$$\frac{\begin{array}{l}bound(\mathtt{X},\mathtt{N}) = \mathtt{P} \text{ implies}\\ \mathtt{X} \in \bar{\mathtt{X}} \text{ and } \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}} \vdash reduce(bound(\mathtt{X},\mathtt{C{<}\bar{T}{>}}))\ \mathtt{<:}\ reduce(\mathtt{P})\end{array}}{override(\mathtt{X},\mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}N\ \{\ldots\}})}$$

Figure 9. FGJ# type well-formedness rules

The typing judgment for method declarations, which has the form `M OK IN C`, read "method declaration `M` 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 of the method with their declared types and the special variable `this`. The type of `this` is inferred by using function *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 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 $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces

Expression typing:

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

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

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

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{T}\ ok \qquad fields(reduce(\mathtt{T})) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}\\ reduce(\mathtt{T}) \text{ is a nonvariable type}\\ \Delta;\Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{S}} \qquad \Delta \vdash reduce(\bar{\mathtt{S}})\ \mathtt{<:}\ reduce(\bar{\mathtt{T}})\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ T(\bar{e})} : \mathtt{T}} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash reduce(bound_{\Delta}(\mathtt{T}_0))\ \mathtt{<:}\ reduce(\mathtt{T})\\ reduce(\mathtt{T}) \text{ is a nonvariable type}\end{array}}{\Delta;\Gamma \vdash \mathtt{(T)e}_0 : \mathtt{T}} \qquad \text{(T-UCAST)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ ok\\ \Delta \vdash reduce(\mathtt{T})\ \mathtt{<:}\ reduce(bound_{\Delta}(\mathtt{T}_0))\\ \mathtt{N} = \mathtt{C{<}\bar{T}{>}} \qquad bound_{\Delta}(\mathtt{T}_0) = \mathtt{D{<}\bar{U}{>}} \qquad dcast(\mathtt{C},\mathtt{D})\\ reduce(\mathtt{T}) \text{ is a nonvariable type}\end{array}}{\Delta;\Gamma \vdash \mathtt{(T)e}_0 : \mathtt{T}} \qquad \text{(T-DCAST)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ ok\\ reduce(\mathtt{T}) = \mathtt{C{<}\bar{T}{>}} \qquad bound_{\Delta}(\mathtt{T}_0) = \mathtt{D{<}\bar{U}{>}}\\ \mathtt{C} \not\trianglelefteq \mathtt{D} \qquad \mathtt{D} \not\trianglelefteq \mathtt{C} \qquad stupid\ warning\\ reduce(\mathtt{T}) \text{ is a nonvariable type}\end{array}}{\Delta;\Gamma \vdash \mathtt{(T)e}_0 : \mathtt{T}} \qquad \text{(T-SCAST)}$$

Method typing:

$$\frac{\begin{array}{c}\Delta = \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}},\bar{\mathtt{Y}}\mathtt{<:}\bar{\mathtt{P}} \qquad \Delta \vdash \bar{\mathtt{T}},\mathtt{T},\bar{\mathtt{P}}\ ok\\ \Delta;\bar{\mathtt{x}} : \bar{\mathtt{T}},\mathtt{this} : thistype(\mathtt{C{<}\bar{X}{>}}) \vdash \mathtt{e}_0 : \mathtt{S}\\ \Delta \vdash reduce(\mathtt{S})\ \mathtt{<:}\ reduce(\mathtt{T})\\ \mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}N\ \{\ldots\}} \qquad override(\mathtt{m},\mathtt{N},\mathtt{{<}\bar{Y} \lhd \bar{P}{>}\bar{T}} \rightarrow \mathtt{T})\end{array}}{\mathtt{{<}\bar{Y} \lhd \bar{P}{>}\ T\ m(\bar{T}\ \bar{x})\ \{\ return\ e_0;\ \}\ OK\ IN\ C{<}\bar{X} \lhd \bar{N}{>}}} \qquad \text{(T-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}\bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}} \vdash \bar{\mathtt{N}},\mathtt{N},\bar{\mathtt{T}}\ ok \qquad fields(\mathtt{N}) = \bar{\mathtt{U}}\ \bar{\mathtt{g}} \qquad \bar{\mathtt{M}}\ \mathtt{OK\ IN\ C{<}\bar{X} \lhd \bar{N}{>}}\\ \mathtt{K} = \mathtt{C(\bar{U}\ \bar{g},\ \bar{T}\ \bar{f})\ \{super(\bar{g});\ this.\bar{f}{=}\bar{f};\}}\\ override(\bar{\mathtt{X}},\mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}N\{\ldots\}})\end{array}}{\mathtt{class\ C{<}\bar{X} \lhd \bar{N}{>}{\lhd}N\ \{\bar{T}\ \bar{f};\ K\ \bar{M}\}\ OK}} \qquad \text{(T-CLASS)}$$

Figure 10. FGJ# typing rules

Computation:

$$\frac{\mathit{fields}(\mathit{reduce}(\mathtt{T})) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{\mathtt{(new\ T(}\bar{\mathtt{e}}\mathtt{)).f}_i \longrightarrow \mathtt{e}_i} \quad \text{(R-FIELD)}$$

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

$$\frac{\emptyset \vdash \mathit{reduce}(\mathtt{T}) \texttt{<:}\ \mathtt{P}}{\mathtt{(P)(new\ T(}\bar{\mathtt{e}}\mathtt{))} \longrightarrow \mathtt{new\ T(}\bar{\mathtt{e}}\mathtt{)}} \quad \text{(R-CAST)}$$

Congruence:

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

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0\mathtt{.m}\texttt{<}\bar{\mathtt{T}}\texttt{>}\mathtt{(}\bar{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0'\mathtt{.m}\texttt{<}\bar{\mathtt{T}}\texttt{>}\mathtt{(}\bar{\mathtt{e}}\mathtt{)}} \quad \text{(RC-INV-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0\mathtt{.m}\texttt{<}\bar{\mathtt{T}}\texttt{>}\mathtt{(\ldots,e}_i\mathtt{,\ldots)} \longrightarrow \mathtt{e}_0\mathtt{.m}\texttt{<}\bar{\mathtt{T}}\texttt{>}\mathtt{(\ldots,e}_i'\mathtt{\ldots)}} \quad \text{(RC-INV-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ T(\ldots,e}_i\mathtt{,\ldots)} \longrightarrow \mathtt{new\ T(\ldots,e}_i'\mathtt{,\ldots)}} \quad \text{(RC-NEW-ARG)}$$

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

Figure 11. FGJ# reduction rules

to expression $\mathtt{e}'$ in one step." We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The reduction rules are given in Fig. 11[2]. There are three reduction rules, one for field access, one for method invocation, and one for casting. The field access reduces to the corresponding argument for the constructor. The method invocation reduces to the expression of the method body, substituting all the parameter $\bar{\mathtt{x}}$ with the argument expression $\bar{\mathtt{d}}$ and the special variable `this` with the receiver. We write $[\bar{\mathtt{d}}/\bar{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ for the expression obtained from $\mathtt{e}_0$ by replacing $\mathtt{x}_1$ with $\mathtt{d}_1$,...,$\mathtt{x}_n$ with $\mathtt{d}_n$, and $\mathtt{y}$ with $\mathtt{e}$.

3.5 Properties

We show that FGJ#'s type system is sound with respect to the operational semantics. The basic structures of the proofs are similar to those of FGJ. The outline of proof is as follows. At first, we require some lemmas.

LEMMA 3.1. *If* $\Delta \vdash \mathtt{S}\texttt{<:}\mathtt{T}$, *then* $\Delta \vdash \mathit{bound}(\mathtt{X},\mathtt{S})\texttt{<:}\mathit{bound}(\mathtt{X},\mathtt{T})$.

LEMMA 3.2. *Suppose* $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ *ok and* $\Delta_1 \vdash \bar{\mathtt{U}}\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\bar{\mathtt{N}}$ *with* $\Delta_1 \vdash \bar{\mathtt{U}}$ *ok and none of* $\bar{\mathtt{X}}$ *appearing in* Δ_1. *Then,* $\Delta_1, [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Delta_2}([\bar{\mathtt{U}}/\bar{\mathtt{X}}]\mathtt{T})\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}](\mathit{bound}_{\Delta_1,\bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}},\Delta_2}(\mathtt{T}))$.

LEMMA 3.3. *If* $\Delta \vdash \mathtt{S}\texttt{<:}\mathtt{T}$, *then* $\Delta \vdash \mathit{reduce}(\mathtt{S})\texttt{<:}\mathit{reduce}(\mathtt{T})$, *and if* $\Delta \vdash \mathit{reduce}(\mathtt{S})\texttt{<:}\mathit{reduce}(\mathtt{T})$, *then* $\Delta \vdash \mathtt{S}\texttt{<:}\mathtt{T}$.

[2] As in the original FJ, FGJ# uses the non-deterministic reduction strategy.

LEMMA 3.4. *If* $\mathtt{class\ C}\texttt{<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\texttt{>}\lhd\mathtt{N}\{\cdots\}$, *then* $\bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}} \vdash \mathit{thistype}(\mathtt{C}\texttt{<}\bar{\mathtt{X}}\texttt{>})\texttt{<:}\mathtt{N}$.

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

LEMMA 3.5 (Type Substitution Preserves Subtyping). *If* $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2 \vdash \mathtt{S}\texttt{<:}\mathtt{T}$ *and* $\Delta_1 \vdash \bar{\mathtt{U}}\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\bar{\mathtt{U}}$ *with* $\Delta_1 \vdash \bar{\mathtt{U}}$ *ok, and none of* $\bar{\mathtt{X}}$ *appearing in* Δ_1, *then* $\Delta_1, [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Delta_2 \vdash [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\mathtt{S}\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2 \vdash \mathtt{S}\texttt{<:}\mathtt{T}$ using Lemma 3.1. □

LEMMA 3.6 (Type Substitution Preserves Type Well-Formedness). *If* $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ *ok and* $\Delta_1 \vdash \bar{\mathtt{U}}\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\bar{\mathtt{N}}$ *with* $\Delta_1 \vdash \bar{\mathtt{U}}$ *ok and none of* $\bar{\mathtt{X}}$ *appearing in* Δ_1, *then* $\Delta_1, [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Delta_2 \vdash [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\mathtt{T}$ *ok.*

Proof. By induction on the derivation of $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ *ok* using Lemma 3.2. □

LEMMA 3.7 (Type Substitution Preserves Typing). *If* $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta_1 \vdash \bar{\mathtt{U}}\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\bar{\mathtt{N}}$ *where* $\Delta_1 \vdash \bar{\mathtt{U}}$ *ok and none of* $\bar{\mathtt{X}}$ *appearing in* Δ_1, *then* $\Delta_1, [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Delta_2; [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Gamma \vdash [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\mathtt{e} : \mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\Delta_1, [\bar{\mathtt{U}}/\bar{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}\texttt{<:}[\bar{\mathtt{U}}/\bar{\mathtt{X}}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1, \bar{\mathtt{X}}\texttt{<:}\bar{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e} : \mathtt{T}$ using Lemmas 3.5 and 3.6. □.

LEMMA 3.8 (Term Substitution Preserves Typing). *If* $\Delta; \Gamma, \bar{\mathtt{x}} : \bar{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta; \Gamma \vdash \bar{\mathtt{d}} : \bar{\mathtt{S}}$ *where* $\Delta \vdash \bar{\mathtt{S}}\texttt{<:}\bar{\mathtt{T}}$, *then* $\Delta; \Gamma \vdash [\bar{\mathtt{d}}/\bar{\mathtt{x}}]\mathtt{e} : \mathtt{S}$ *for some* $\mathtt{S}$ *such that* $\Delta \vdash \mathtt{S}\texttt{<:}\mathtt{T}$.

Proof. By induction on the derivation of $\Delta; \Gamma, \bar{\mathtt{x}} : \bar{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$ using Lemma 3.5. □

LEMMA 3.9. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\bar{\mathtt{T}}\texttt{>}) = \texttt{<}\bar{\mathtt{Y}}\lhd\bar{\mathtt{P}}\texttt{>}\bar{\mathtt{U}} \to \mathtt{U}$ *and* $\mathit{mbody}(\mathtt{m}\texttt{<}\bar{\mathtt{V}}\texttt{>}, \mathtt{C}\texttt{<}\bar{\mathtt{T}}\texttt{>}) = \bar{\mathtt{x}}.\mathtt{e}_0$ *where* $\Delta \vdash \mathtt{C}\texttt{<}\bar{\mathtt{T}}\texttt{>}$ *ok and* $\Delta \vdash \bar{\mathtt{V}}$ *ok and* $\Delta \vdash \bar{\mathtt{V}}\texttt{<:}[\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{P}}$, *then there exist some* $\mathtt{N}$ *and* $\mathtt{S}$ *such that* $\Delta \vdash \mathtt{C}\texttt{<}\bar{\mathtt{T}}\texttt{>}\texttt{<:}\mathtt{N}$ *and* $\Delta \vdash \mathtt{N}$ *ok and* $\Delta \vdash \mathtt{S}\texttt{<:}[\bar{\mathtt{V}}/\bar{\mathtt{S}}]\mathtt{U}$ *and* $\Delta \vdash \mathtt{S}$ *ok and* $\Delta; \bar{\mathtt{x}} : [\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{U}}, \mathtt{this} : \mathit{thistype}(\mathtt{N}) \vdash \mathtt{e}_0 : \mathtt{S}$.

Proof. By induction on the derivation of $\mathit{mbody}(\mathtt{m}\texttt{<}\bar{\mathtt{V}}\texttt{>}, \mathtt{C}\texttt{<}\bar{\mathtt{T}}\texttt{>}) = \bar{\mathtt{x}}.\mathtt{e}$ using Lemmas 3.4, 3.5, and 3.7. □

Then, we can prove the following subject reduction theorem.

THEOREM 3.1 (Subject Reduction). *If* $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$, *then* $\Delta; \Gamma \vdash \mathtt{e}' : \mathtt{T}'$ *for some* $\mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}'\texttt{<:}\mathtt{T}$.

Proof. By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ using Lemmas 3.8 and 3.9. □

The proof of the following progress theorem is easy.

THEOREM 3.2 (Progress). *Suppose* $\mathtt{e}$ *is a well-typed expression.*

1. *If* $\mathtt{e}$ *includes* $\mathtt{new\ N}_0\mathtt{(}\bar{\mathtt{e}}\mathtt{).f}$ *as a subexpression, then* $\mathit{fields}(\mathtt{N}_0) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$ *and* $\mathtt{f} \in \bar{\mathtt{f}}$ *for some* $\bar{\mathtt{T}}$ *and* $\bar{\mathtt{f}}$.
2. *If* $\mathtt{e}$ *includes* $\mathtt{new\ N}_0\mathtt{(}\bar{\mathtt{e}}\mathtt{).m}\texttt{<}\bar{\mathtt{V}}\texttt{>}\mathtt{(}\bar{\mathtt{d}}\mathtt{)}$ *as a subexpression, then* $\mathit{mbody}(\mathtt{m}\texttt{<}\bar{\mathtt{V}}\texttt{>}, \mathtt{N}_0) = \bar{\mathtt{x}}.\mathtt{e}_0$ *and* $\#(\bar{\mathtt{x}}) = \#(\bar{\mathtt{d}})$ *for some* $\bar{\mathtt{x}}$ *and* $\mathtt{e}_0$.

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

$$\mathtt{v} ::= \mathtt{new\ N(}\bar{\mathtt{v}}\mathtt{)}$$

Finally, we get the FGJ# type soundness theorem.

THEOREM 3.3 (FGJ# 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 either (1) an FGJ# value* $\mathtt{v}$ *with* $\emptyset; \emptyset \vdash \mathtt{v} : \mathtt{S}$ *and* $\emptyset \vdash \mathtt{S}\texttt{<:}\mathtt{T}$ *or (2) an expression containing* $\mathtt{(P)new\ N(}\bar{\mathtt{e}}\mathtt{)}$ *where* $\emptyset \vdash \mathtt{N}\texttt{<:}\mathtt{P}$.

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 $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ do not use T-DCAST and T-SCAST rules), it does not produce any typecast errors.

THEOREM 3.4. *If* e *is cast-safe in* $\emptyset; \emptyset$ *and* $\mathtt{e} \longrightarrow^{*} \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *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 *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 Edge<G> shown in Figure 2 will be as follows,

```
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 WeightEdge<G> shown in Figure 3 will be as follows, because type parameters are replaced with the erasures of their bounds:

```
class WeightEdge extends Edge {
    int weight;
    int f(RichNode s, RichNode d) {
        int sv = s.value(); int dv = d.value();
        ... }
    void connect(RichNode s, RichNode d) {
        weight = f(s, d);
        super.connect(s, d); }
    int f(int i1, int i2) { return 0; } }
```

Unfortunately, the erased connect method declared in WeightEdge does not override the connect method declared in Edge, while the original one does. Therefore, behavior of the erased program is changed from that of the original program. Thus, we have to modify the definition of erasures of type parameters; while an erasure of a type parameter is defined as its bound in FGJ, in SJ, it is defined as the bound of a type parameter declared in the *highest* superclass. In this definition, the erasure of formal parameter type G#E of the connect method in WeightEdge<G> is Node, thus it appropriately overrides the superclass' method.

Then, a second difficulty arises. Suppose the modified erasure of WeightEdge<G> shown below:

```
class WeightEdge extends Edge {
    int weight;
    int f(Node s, Node d) {
        // type error!
        int sv = s.value(); int dv = d.value();
        ... }
    void connect(Node s, Node d) {
        weight = f(s, d);
        super.connect(s, d); } }
```

This erased program actually cannot be typed, because the erased class Node does not declare a method value(). To recover the erased type information, we have to insert synthetic casts as follows:

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

MyType (or sometimes called *ThisType*) is also studied to solve the mismatching problem of recursive class definitions [6, 7]. MyType is the type of this and changes its meaning along the inheritance 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 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

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Conference Proceedings of OOPSLA '97, Atlanta*, pages 49–65. ACM, 1997.

[2] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of OOPSLA2003*, pages 96–114, 2003.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA'98*, pages 183–200, 1998.

[4] Kim Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(2):225–290, 2003.

[5] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82(8), 2003.

[6] Kim B. Bruce, Adrian Fiech, and Leaf Peterson. Subtyping is not a good "match" for object-oriented languages. In *ECOOP'97*, volume 1241 of *LNCS*, pages 104–127, 1997.

[7] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 389–413, 2004.

[8] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, volume 1445 of *LNCS*, pages 523–549, 1998.

[9] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[10] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *ICSE 2001*, pages 285–294, 2001.

[11] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA 1998*, pages 201–215, 1998.

[12] Eric Ernst. Family polymorphism. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 303–327, 2001.

[13] Erik Ernst. Propagating class and method combination. In *ECOOP'99*, volume 1628 of *LNCS*, pages 67–91. Springer-Verlag, 1999.

[14] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of 33th ACM Symposium on Principles of Programming Languages (POPL)*, pages 270–282, 2006.

[15] Stephan Hermann. Object Teams: Improving modularity for crosscutting collaborations. In *Net Object Days 2002*, volume 2591 of *LNCS*, 2002.

[16] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[17] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2003.

[18] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005*, volume 3780 of *LNCS*, pages 161–177, 2005.

[19] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP 2004)*, 2004.

[20] Ole Lehrmann Madsen and Birger Moller-Pdersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA'89*, pages 397–406, 1989.

[21] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA'04*, pages 99–115, 2004.

[22] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. *J&*: Nested intersection for scalable software composition. In *OOPSLA'06*, pages 21–35, 2006.

[23] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP'03*, volume 2743 of *LNCS*, pages 201–224, 2003.

[24] Martin Odersky and Philip Wadler. Pizza into Java: Translation theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 146–159, 1997.

[25] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA'05*, pages 41–57, 2005.

[26] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002*, volume 2374 of *LNCS*, pages 89–110, 2002.

[27] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[28] Chieri Saito and Atsushi Igarashi. The essence of lightweight family polymorphism. In *FTfJP*, 2007.

[29] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based design. *ACM TOSEM*, 11(2):215–255, 2002.

[30] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *ECOOP'99*, volume 1628 of *LNCS*, pages 186–204, 1999.