Lightweight Dependent Classes

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

Extensive research efforts have been devoted to implement a group of type-safe mutually recursive classes; recently, proposals for separating each member of the group as a reusable and composable programming unit have also been presented. One problem of these proposals is verbosity of the source programs; we have to declare a recursive type parameter to parameterize each mutually recursive class within each class declaration, and we have to declare a fixed-point class with empty class body for each parameterized class. Therefore, even though the underlying type system is simple, programs written in these languages tend to be rather complex and hard to understand. In this paper, we propose a language with lightweight dependent classes that forms a simple type system built on top of generic Java. In this language, we can implement each member of type-safe mutually recursive classes in a separate source file without writing a lot of complex boilerplate code. To carefully investigate type soundness of our proposal, we develop X.FGJ, a simple extension of FGJ supporting lightweight dependent classes. This type system is proved to be sound.

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 Mutually recursive extensions, Class-based languages, Generics, Type safety, Dependent classes

1. Introduction

Recently, extensive research efforts have been devoted to implement a group of type-safe mutually recursive classes and to separate each member of the group as a reusable and composable programming unit. These pieces of work address the following problem. In most modern object-oriented languages with simple name-based type system such as Java and C#, each class is referred by its *name*, thus different sets of mutually recursive classes necessarily have different names, even though their structures are similar. The idea of grouping the related classes and nesting them inside a class that represents the group has many success stories for addressing this problem; it have been proved useful to form families of collaborating objects [11, 22], to develop scalable and extensible components [23, 28, 25], to address the "expression problem"[13, 14, 5], and so on. However, such nesting requires that each family becomes a large monolithic program, thus implementations of all the members of one family are placed in a single source file. Each member's implementation cannot be reused in the different context, and cannot be developed in parallel. Furthermore, the implementation of an enclosing class also depends on its enclosed classes.

To address this problem, a language with *dependent classes* [15] that is a generalization of virtual classes [21] that expresses similar semantics by parameterization rather than by nesting is proposed. In this language, an instance of the enclosing class becomes a formal parameter of a constructor of the enclosed class. However, since the families are represented by objects, a dependent type system has to be introduced, resulting in a rather complex type system.

On the other hand, there are much simpler approaches [20, 30] based not on virtual classes (attributes of objects) but on *lightweight family polymorphism* (attributes of classes) [31]. In these approaches, the modularization is achieved by parameterizing each member of family using type parameters, and their type systems are built on top of FGJ[17]. The resulting type systems are significantly simpler without losing much expressive power of the languages. One problem of these approaches is verbosity of the source programs; we have to declare a recursive type parameter to parameterize each mutually recursive class within each class declaration, and we have to declare a fixed-point class with empty class body for each parameterized class. Therefore, even though the underlying type system is simple, programs written in these languages tend to be rather complex and hard to understand.

In this paper, we propose a language with lightweight dependent classes, a class based simple solution without requiring a lot of boilerplate code for type parameter lists and fixed-point classes. The idea is to parameterize an enclosing class by using a type parameter. In Java, we cannot select a nested class member on a type parameter `X`; i.e., if there is a fully qualified name `X.C`, `X` cannot be a type parameter. We show that by allowing `X` to be a type parameter, the aforementioned problem is solved. The resulting type system is a quite simple extension of generic Java [3, 17] with significant expressive power. By using examples, we show that the problem of verbosity is solved by adding quite *lightweight* extension to Java 5.0 (syntactically, only the difference from the original Java 5.0 is that we can access a nested class member on a type parameter).

The same problem that the previous work faced also arises in this work; in Java, the type of `this` is "hard-linked" to the enclosing class [20, 30]. As discussed later, this hard-linking produces compile errors which can be avoided if we give more precise type to `this`. To tackle this problem, we define a type inference algo-

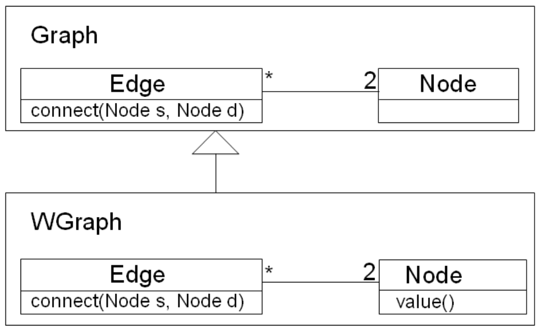

Figure 1. Overview of our example

rithm that infers the possible extension of type of `this`. With this feature, the restriction imposed on the use of the expression `this` is eased without losing the property of type safety. To carefully investigate type soundness of our proposal, we develop X.FGJ, a core calculus of lightweight dependent classes. This formalization is built on top of FGJ and its type system is proved to be sound.

So far, our contributions are summarized as follows:

- We identify a minimum lightweight set of language constructs that address the aforementioned problem of nesting without introducing a lot of boilerplate code. The resulting language is very similar to Java, one of the most popular mainstream languages.
- We present a programming style to implement modular and type-safe mutually recursive classes using the proposed language, and discuss its abilities and limitations.
- We rigorously formalize the idea and prove its type soundness.

The rest of this paper is structured as follows. Section 2 describes the features of lightweight dependent classes, and how the aforementioned problem is solved using this proposal. Section 3 explains some technical hot spots which have to be considered to construct a safe type system. In section 4, we formalize the proposed language features on top of FGJ, and show the proposal is type sound. In section 5, we discuss how this work is related to other researches. Finally, section 6 concludes this paper.

2. Programming Lightweight Dependent Classes

2.1 A Graph Example

We start by informally describing the main aspect of lightweight dependent classes, the language construct we study in this paper. To show how this construct is used to support modular and type-safe mutually recursive extensions, we consider an example originally presented in [11].

We illustrate this example in Fig. 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 incident 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` adding the feature of setting weight on each edge. In `WGraph`, the member `Edge` is refined to store the weight of each edge. The member `Node` is also refined to store a new property. This property may be color, or label etc.; in Fig. 1, `Node` declares an abstract method `value()` to return an integer value of this property. In our application, the weight of an edge is calculated by using this property of the pair of nodes

```
class Graph<E extends Edge<E#G>,
            N extends Node<E#G>> {
}

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 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 SimpleGraph
      extends Graph<Edge<SimpleGraph>,
                    Node<SimpleGraph>> { }
```

Figure 2. A lengthy graph definition in Scalable Java [20]

connected by the edge. To do so, the `connect` method declared in `Edge` is to be appropriately overridden.

2.2 Motivations

Even though this example is simple, there are many challenges. 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 have been recently devoted to solve this problem [8, 11, 23, 32, 31, 5, 26]. In family polymorphism, for example, like virtual methods, a reference to a nested class member is resolved at run-time and thus the meaning of mutual references to class names will change when a subclass of the enclosing class is created 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.

Although the family polymorphism approach is very powerful, 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 definition of `Edge` and `Node` is placed in the same source file. Therefore, it is hard to say the family polymorphism approach is based on components. When there are many kinds of `Edge` and `Node`, it will be convenient if we can compose them freely in arbitrary combinations, but the family polymorphism approach does not provide such a flexible way for it. Therefore, we identify the following requirement:

Separation of members of family: Each member of family may be placed in the separate source files from that of the family.

There are pieces of literature that address this requirement. For example, dependent classes are generalizations of virtual classes that expresses similar semantics by parameterization rather than by nesting. On the other hand, much simpler solutions are independently proposed [20, 30]. In these approaches, the decoupling of members and families is achieved by parameterizing each member of family by using a type parameter.

However, in these approaches there is a problem that a program written in the language becomes verbose. For example, Figure 2 shows an implementation of the graph example written in Scalable Java [20]. In this paper we do not intend the reader to understand details of Scalable Java (`G#E` is the type parameter `E` declared in (the upper bound of) type `G`). In Figure 2 we can see that the declaration of `Graph` is parametrized by type parameters, and these type parameters are recursively used in the constraints on these type parameters. Recursive type parameters are also used in the declarations of `Edge` and `Node`. This recursion makes the program complex and hard to understand. Furthermore, to “fix” the recursion, we have to create the fixed-point class `SimpleGraph`. Note that `SimpleGraph`’s body is empty; it is created only for fixing the recursion and provides no behaviors. Nevertheless, the programmer have to provide rather complex `extends` clause to complete its declaration. Thus, declaring the recursive type parameters and fixed-point classes with empty class body should be avoided.

Therefore, we also identify the following requirements:

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

Readability: We do not require a lot of boilerplate code. Declaring a lot of recursive type parameters and fixed-point classes is undesirable.

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

Type safety: Extensions do not create run-time errors.

Modularity: Extensions do not require modification or recompilation of the existing systems.

Non-destructive extension: Different extensions may co-exist in the same system.

2.3 Solution

We show how our proposal, lightweight dependent classes, fulfills the aforementioned requirements. The idea is to parameterize an enclosing class by using a type parameter. This is a quite simple extension of generic Java with significant expressive power. In the following subsections, we show how this construct solves the problems by examples.

2.3.1 Simple Graph Family

Figure 3 shows the definition of family `Graph` using lightweight dependent classes. It declares two nested classes, `Edge` and `Node`, whose implementations are separately provided in their superclasses, `EdgeI` and `NodeI`, respectively[1].

A class `NodeI` is a concrete implementation of `Graph.Node`, and a class `EdgeI` is a concrete implementation of `Graph.Edge`. Both of them are parameterized over its belonging family by type parameter `G`, whose upper bound is `Graph`. `NodeI` declares an instance variable `es`, which represents a set of incident edges with which this node is connected. For the future extensibility, the type of edge is not hard-linked to `Graph.Edge`; instead, it is declared as `G.Edge`. Similarly, the type of instance variables `src` and `dst` in `EdgeI` that represent source and destination nodes respectively, and formal parameter types in `connect` are also declared as `G.Node`.

Each type parameter `G` is instantiated by `Graph` inside the body of `Graph` itself. Note that, unlike Figure 2, we do not have to declare any recursive type parameters (the class `Graph` is not parametrized) and fixed-point classes with empty body.

[1] In this paper, we omit any modifier lists such as `public`, `static`, and so on for simplicity.

```
class Graph {
    class Edge extends EdgeI<Graph> { }
    class Node extends NodeI<Graph> { }
}

class EdgeI<G extends Graph> {
    G.Node src, dst;
    void connect(G.Node s, G.Node d) {
        s.add(this); d.add(this);
        src = s; dst = d;
    }
}

class NodeI<G extends Graph> {
    Vector<G.Edge> es = new Vector<G.Edge>();
    void add(G.Edge e) {
        es.add(e);
    }
}
```

Figure 3. Simple graph definitions

```
class WGraph extends Graph {
    class Edge extends WEdgeI<WGraph> { }
    class Node extends RichNode<WGraph> { }
}

class WEdgeI<G extends WGraph>
      extends EdgeI<G> {
    int weight;
    int f(G.Node s, G.Node d) {
        int sv = s.value(); int dv = d.value();
        ... }
    void connect(G.Node s, G.Node d) {
        weight = f(s,d);
        super.connect(s,d);
    }
}

abstract class RichNode<G extends Graph>
      extends NodeI<G> {
    abstract int value();
}
```

Figure 4. Weighted graph extension

2.3.2 Extending the Base Family

Figure 4 shows the definition of family `WGraph` that is a subclass of `Graph`. As in `Graph`, it also declares nested classes `Edge` and `Node`. In this case, nested classes declared in a subclass override nested classes declared in the superclass. Their superclasses are covariantly refined to `WEdgeI` and `RichNode`, respectively.

`WEdgeI` is a concrete implementation of `WGraph.Edge` that refines the definition of `EdgeI`. It declares an instance variable `weight` that stores the weight of edges. Similarly, `RichNode` is an abstract definition of `WGraph.Node`. It provides a method `value()` that returns the property of the node. Both of them declare a type parameter `G` whose upper bound is refined to `WGraph`. Since nested classes of `Graph` are accessed through the type parameter in the base classes (in the form of `G.Node` and `G.Edge`), in the subclasses, we can also refer to each member of the *extended* family even when declarations in the base classes are evaluated. For example, a method `connect` declared in `WEdgeI` overrides `connect` declared

```
class ColorNode<G extends Graph>
      extends RichNode<G> {
    Color color;
    int value() { ... }
}

class CWGraph extends WGraph {
    class Edge extends WEdgeI<CWGraph> {}
    class Node extends ColorNode<CWGraph> {}
}
```

Figure 5. Colored weighted graph

```
class LabelNode<G extends Graph>
      extends RichNode<G> {
    Label label;
    int value() { ... }
}

class LWGraph extends WGraph {
    class Edge extends WEdgeI<LWGraph> {}
    class Node extends LabelNode<LWGraph> {}
}
```

Figure 6. Labeled weighted graph

in `EdgeI`, because each of formal parameter types is declared as `G.Node`, and `Node` is appropriately overridden in `WGraph`.

To provide a complete implementation of `WGraph`, we have to implement the abstract class `RichNode`. A colored weighted graph implementation is shown in Figure 5. As in the case of `Graph`, we do not have to introduce any other fixed-point classes.

Note that the upper bound of `G` declared in `RichNode` and `ColorNode` is unchanged from that of `G` in `NodeI`, since both of definitions do not use the properties introduced in `WEdgeI`. To enhance reusability of each component, we do not unnecessarily restrict their upper bounds, as discussed in the next section.

2.3.3 Another Combination of Extensions

In this section we show that our approach enables much flexible composition as in [20]. 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 6 shows that there may be another implementation of nodes in `WGraph` other than `ColorNode`, namely `LabelNode`, which is also declared as a subclass of `RichNode`. The new family `LWGraph` 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 with 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 combining `ColorNode` with `Graph`:

```
class CGraph extends Graph {
    class Edge extends EdgeI<CGraph> {}
    class Node extends ColorNode<CGraph> {}
}
```

3. Technical Hot Spots

So far, we have shown how lightweight dependent classes support separation of members of family with quite simple extension of generic Java. We do not have to declare any recursive type parameters and fixed-point classes with empty class body, thus the resulting code is reasonably readable. The remaining important issue is type safety. There are some challenges in making type system of lightweight dependent classes sound. In this section, we overview these challenges and how we tackle the problems.

3.1 Notes on Inheritance and Subtyping

In Figure 3 and 4, both classes `Graph` and `WGraph` declare nested classes `Node` and `Edge`. How do the base class and the extended class with the same name relate to each other? There are two possibilities: `WGraph.Node` is a subclass (and thus a subtype) of `Graph.Node`, or `WGraph.Node` is not a subclass (and not a subtype) of `Graph.Node`. If the former approach is taken, the type system will not be safe. For example, consider the following demonstration code:

```
Graph.Node n1 = new Graph.Node();
Graph.Node n2 = new Graph.Node();
CWGraph.Edge e1 = new CWGraph.Edge();
Graph.Edge e2 = e1;
e2.connect(n1,n2); // error!
```

In this example, we connect two instances of `Graph.Node`, `n1` and `n2`. The type-checker accepts this code, since the static type of `e2` is `Graph.Edge` and the formal parameter types of `connect` on `Graph.Edge` are compatible to `Graph.Node`. However, the actual type of `e2` is `CWGraph.Edge`, where the `connect` method is overridden to call the `value()` method that is not provided by `n1` and `n2`.

Our language does not allow this subclassing; as in Java, subclassing is a reflexive and transitive closure induced by the `extends` clause, and no implicit subclassing are provided. However, this decision produces another subtlety. Consider that the situation where `Graph.Edge` has a field `Object f;`. In this case, a field access expression `e.f`, where `e`'s type is `G.Edge` and `G` is a type parameter whose upper bound is `Graph`, is not always safe.

To tackle this problem, we take somewhat a drastic approach; the body of nested classes must always be empty. We may distinguish a conventional nested or inner class from a nested class whose implementation is provided in a separate class, by using an annotation or a modifier indicating the annotated class is a conventional nested class. However, to focus on the features we would like to study in this paper, we only include nested classes whose body is empty in the core calculus explained in section 4.

To ensure type safety, we also impose another restriction on nested class declarations: when a nested class is overridden in a subclass, the superclass of overriding nested class have to be a subclass of that of overridden nested class; i.e., the definition of nested classes must be covariantly refined.

3.2 Type Inference for `this`

As also discussed in [20] and [30], another subtlety exists in the type of `this`. In Figure 3, for example, 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 3 is therefore `EdgeI<G>`, where `G` is some subtype of `Graph`. On the other hand, the formal parameter type of the `add` method is declared as `G.Edge`, which is a subtype of `EdgeI<G>`; therefore, the type of `this` in the method call `s.add(this)` is not compatible to the declared type.

There are some well-known solutions to address such a problem. For example, we may introduce a new kind of types like *MyType* (also known as `ThisType`) [4, 7] to the language. 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 [28]. However, both of these approaches require significant language extensions.

In this paper, we take another approach that is also taken in [20] and [30]; using upper bounds of type parameter `G`, the actual type of `this` may be *inferred*. For example, `EdgeI` is playing the role of `G.Edge` in the class `EdgeI`, thus we can treat `this` as having the type `G.Edge`. We informally describes type inference rules for `this` as follows:

- If one of the type parameters, namely `X`, has an upper bound that is a class that declares a nested class, namely `E`, whose superclass is the enclosing class of `X` where `X` is instantiated by the enclosing class of `E`, the type of `this` is `X.E`; i.e., in the following class declaration,

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

  where

  ```
  class D { .. class E extends C<D> {} ..}
  ```

  the type of `this` inside `C` is `X.E`.
- Otherwise, the type of `this` is its enclosing class.

In the case of program shown in Figure 3, `EdgeI` declares a type parameter `G` whose upper bound is `Graph`, and `Graph` declares a nested class `Edge` whose superclass is `EdgeI<Graph>`. Therefore, the type of `this` used inside `EdgeI` is inferred as `G.Edge`, which is the formal parameter type of `add` method declared in `NodeI` [2]. Thus, the program shown in Figure 3 is safely type-checked.

There may be ambiguity in the above algorithm, since it is possible that more than one type parameter can have the same upper bound and thus multiple interpretations of type of `this` may be possible; i.e., in Figure 3, there may be another type parameter in `EdgeI` whose upper bound is also `Graph`. In this case, the type inference algorithm does not proceed and type of `this` is always interpreted as the enclosing class.

Note that if the superclass of the nested class is instantiated by the *subclass* of the enclosing class, the first case of the type inference rule is *not* applied. For example, if `Graph` is declared as follows,

```
class Graph {
    class Edge extends EdgeI<CWGraph> {}
    class Node extends NodeI<CWGraph> {}
}
```

the type of `this` inside `EdgeI` is `EdgeI<G>`. If this property is not held and the first case of the type inference is applied in the case of the above declaration, the property of type safety is not ensured. Actually, we can write the following unsafe program:

```
Graph.Edge e1 = new Graph.Edge();
Graph.Node n1 = new Graph.Node();
Graph.Node n2 = new Graph.Node();
e1.connect(n1,n2);
```

[2] More precisely, we have to ensure that type parameter `G` used in `EdgeI` and in `NodeI` are identical. For this purpose, we model the method lookup on `G.Node` to search its superclass instantiated with type parameter `G` (i.e. `NodeI<G>`), even though `Node`'s superclass in `Graph` is declared as `NodeI<Graph>`. We develop this feature by appropriately defining the upper bound of `G.Node`, which is discussed in section 4.

In this program, the expression `e1.connect(n1,n2)` reduces to an expression `n1.add(this)` by substituting formal parameter `s` with `n1`. Since type of `n1` is `Graph.Node` and its superclass is `EdgeI<CWGraph>`, the formal parameter type of `add` is `CWGraph.Edge`, while the type of `this` is inferred `Graph.Edge`, and as explained in section 3.1, the two types are incompatible. Therefore, we require that to apply the first rule of the inference, the superclass of the nested class is instantiated *exactly* the same class as the enclosing class.

3.3 Limitations

In our proposal, there exists one limitation. Since a graph conceptually consists of a set of nodes and a set of edges, someone would like to store a set of nodes in a graph instance. For example, one may change the definition of `Graph` as follows:

```
class Graph {
    class Edge extends EdgeI<Graph> {}
    class Node extends NodeI<Graph> {}
    Vector<Node> ns = new Vector<Node>();
    void add(Node n) { ns.add(n); }
}
```

Unfortunately, field `ns` and method `add` in the above code cannot be used for the refined `Node` definition in the subclass of `Graph`, since any accesses to type `Node` inside `Graph` is hard-linked to `Graph.Node`:

```
CWGraph g = new CWGraph();
CWGraph.Node n1 = new CWGraph.Node();
g.add(n1);  // compile error!
```

In the above code, the type of expression `n1` is `CWGraph.Node`, while `g.add` expects `Graph.Node`, and as explained in the previous section, these types are not compatible.

The same limitation also exists in lightweight family polymorphism [31], where an access to a relative path type is prohibited in a top-level class. Interestingly, FGJ# [20] does not suffer from this limitation, since in FGJ#, any accesses to `Node` (and `Edge`) may be parameterized by using type parameters, which are finally fixed in a fixed-point class.

Nevertheless, it is still possible to create another class in the family that maintains the list of `Node`s. For example, we may modify the declaration of `Graph` so that it includes a wrapper for the container of `Node` whose implementation is separately provided:

```
class Graph {
  class Edge extends EdgeI<Graph> {}
  class Node extends NodeI<Graph> {}
  class Nodes extends NodesI<Graph> {}
}
class NodesI<G extends Graph> {
  Vector<G.Node> ns = new Vector<G.Node>();
  void add(G.Node n) { ns.add(n); }
}
```

In this setting, the following pieces of code can safely be compiled and raise no run-time errors:

```
CWGraph g = new CWGraph();
CWGraph.Node n1 = new CWGraph.Node();
CWGraph.Nodes ns = new CWGraph.Nodes();
ns.add(n1);
```

4. X.FGJ: A Tiny Core of Lightweight Dependent Classes

In this section, we formalize the ideas described in the previous sections as a small calculus named X.FGJ, built on top of Feath-

Syntax:

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

Subclassing:

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

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

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

Figure 7. X.FGJ syntax and subclassing

erweight GJ (FGJ) [17], a functional core of class-based object-oriented languages with the feature of generics.

4.1 Syntax

The abstract syntax of X.FGJ is given in Figure 7. The metavariables T, S, V, and U range over types; X, Y, Z, and W range over type variables; N, P, and Q range over nonvariable types; A ranges over instantiatable types; C, D, E and F range over class names; L ranges over class declarations; K ranges over constructor declarations; M ranges over method declarations; I ranges over nested class 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.

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. X.FGJ supports polymorphic methods, and type parameters for generic method invocation is 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, X.FGJ is a purely functional calculus.

Subclassing in X.FGJ, also shown in Figure 7, 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 X.FGJ program is a pair (CT,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. We assume that a class Object has no members and its definition does not appear in the class table. 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 ....

4.2 Auxiliary definitions

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 8 and 9. The function *fields*(N) is a sequence $\bar{\texttt{T}}\ \bar{\texttt{f}}$ of field types and names declared in N. Application of type substitution $[\bar{\texttt{T}}/\bar{\texttt{X}}]$ is defined in the customary manner. For example, we write $[\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{T}_0$ for the type obtained from $\texttt{T}_0$ by replacing $\texttt{X}_1$ with $\texttt{T}_1, \cdots$, and $\texttt{X}_n$ with $\texttt{T}_n$. We write $\texttt{C} \notin \bar{\texttt{I}}$ to mean that the nested class definition of the name C is not included in $\bar{\texttt{I}}$. The type of the method invocation m at N, written *mtype*(m,N), is a type of the form $\texttt{<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\bar{\texttt{U}} \rightarrow \texttt{U}$. We write $\texttt{m} \notin \bar{\texttt{M}}$ to mean that the method definition of the name m is not included in $\bar{\texttt{M}}$. The body of the method invocation m at N, written *mbody*(m,N), is a pair, written $\bar{\texttt{x}}\texttt{.e}$, of a sequence of parameters $\bar{\texttt{x}}$ and an expression e.

Type inference rules for this is shown in Figure 9. The function *thistype* returns the inferred type of this using the upper bounds of type parameters. We assume that the second rule is applied only when the first rule does not hold.

4.3 Typing

An environment Γ is a finite mapping from variables to types, written $\bar{\texttt{x}} : \bar{\texttt{T}}$. A type environment Δ is a finite mapping from type variables to nonvariable types, written $\bar{\texttt{X}}\texttt{<:}\bar{\texttt{N}}$. As defined in Figure 10, we write $\mathit{bound}_\Delta(\texttt{T})$ for an upper bound of T in Δ. Besides type parameters and nonvariable types, we also need to define bound_Δ of nested class type T.D. Note that the upper bound of T.D is its superclass, since the body of nested class is always empty. Furthermore, there is a special case for nested class type D on type variable X: X.D. In this case, if the upper bound of X appears in the type argument list of D's superclass, it is replaced with X. This replacement is necessary to ensure that type substitution preserves typing, because variance subtyping is not allowed for generic classes. We assume that the fourth rule is applied only when the third rule does not hold.

The subtyping relation $\Delta \vdash \texttt{S <: T}$, read "S is a subtype of T in Δ," is also defined in Figure 10. As in FGJ, subtyping is the reflexive and transitive closure of the extends relation. Note that these subtyping rules ensure that a nested class type T.D is not a super type of any types other than itself.

We write $\Delta \vdash \texttt{T}\ \mathit{ok}$ if a type T is well formed in context Δ. The rules for well-formed types appear in Figure 11. A type $\texttt{C<}\bar{\texttt{T}}\texttt{>}$ is well formed if a class declaration that begins with $\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}$ exists in CT, substituting $\bar{\texttt{T}}$ for $\bar{\texttt{X}}$ respects the bounds $\bar{\texttt{N}}$, and all of $\bar{\texttt{T}}$ are ok. A nested class type T.D is well-formed if the path type T is ok and the nested class D is declared in some super class of upper bound of T.

We say that a type environment Δ is well-formed if $\Delta \vdash \Delta(\texttt{X})$ 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 $\Delta \vdash \Gamma(\texttt{x})$ ok for all x in $dom(\Gamma)$.

Figure 11 also shows a rule that indicates the condition of valid overriding of nested classes, *override*$(\texttt{D}, \texttt{N}, \texttt{P}, \Delta)$. This condition is true if there exists a super type of N.D in environment Δ, it must be a super type of P. Another rule is for covariant overriding on the method result type, which is ensured by *override*$(\texttt{m}, \texttt{N}, \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{T}} \rightarrow \texttt{T}_0)$.

Field lookup:

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

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

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ class\ D{\triangleleft}P\ \{\}\ ..\}}}{\mathit{fields}(\mathtt{C{<}\overline{T}{>}.D}) = \mathit{fields}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P})} \qquad \text{(F-NEST)}$$

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ \overline{I}\ \}} \qquad \mathtt{D} \notin \overline{\mathtt{I}}}{\mathit{fields}(\mathtt{C{<}\overline{T}{>}.D}) = \mathit{fields}(\mathtt{N.D})} \qquad \text{(F-NEST-SUPER)}$$

Method type lookup:

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

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

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ class\ D{\triangleleft}P\ \{\}\ ..\}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C{<}\overline{T}{>}.D}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P})} \qquad \text{(MT-NEST)}$$

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ \overline{I}\ \}} \qquad \mathtt{D} \notin \overline{\mathtt{I}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C{<}\overline{T}{>}.D}) = \mathit{mtype}(\mathtt{m}, \mathtt{N.D})} \qquad \text{(MT-NEST-SUPER)}$$

Method body lookup:

$$\frac{\begin{array}{c}\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{\overline{S}\ \overline{f};\ K\ \overline{M}\ \overline{I}\}} \\ \mathtt{{<}\overline{Y} \triangleleft \overline{P}{>}\ U\ m(\overline{U}\ \overline{x})\ \{\ return\ e_0;\ \}} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m{<}\overline{V}{>}}, \mathtt{C{<}\overline{T}{>}}) = \overline{\mathtt{x}}.[\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e}_0} \qquad \text{(MB-CLASS)}$$

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

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ class\ D{\triangleleft}P\ \{\}\ ..\}}}{\mathit{mbody}(\mathtt{m{<}\overline{V}{>}}, \mathtt{C{<}\overline{T}{>}.D}) = \mathit{mbody}(\mathtt{m{<}\overline{V}{>}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P})} \qquad \text{(MB-NEST)}$$

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ \overline{I}\ \}} \qquad \mathtt{D} \notin \overline{\mathtt{I}}}{\mathit{mbody}(\mathtt{m{<}\overline{V}{>}}, \mathtt{C{<}\overline{T}{>}.D}) = \mathit{mbody}(\mathtt{m{<}\overline{V}{>}}, \mathtt{N.D})} \qquad \text{(MB-NEST-SUPER)}$$

Figure 8. X.FGJ lookup functions

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

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\{..\}}}{\mathit{thistype}(\mathtt{C{<}\overline{T}{>}}) = \mathtt{C{<}\overline{T}{>}}}$$

Figure 9. Type inference for `this`

Bound of type:

$$\mathit{bound}_\Delta(\mathtt{X}) = \Delta(\mathtt{X})$$

$$\mathit{bound}_\Delta(\mathtt{C{<}\overline{T}{>}}) = \mathtt{C{<}\overline{T}{>}}$$

$$\frac{\begin{array}{c}\mathit{bound}_\Delta(\mathtt{X}) = \mathtt{C{<}\overline{T}{>}} \\ \mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\{..\ class\ D{\triangleleft}E{<}\overline{S}, C{<}\overline{T}{>}, \overline{U}{>}\{\}..\}}\end{array}}{\mathit{bound}_\Delta(\mathtt{X.D}) = \mathtt{E{<}\overline{S}, X, \overline{U}{>}}}$$

$$\frac{\begin{array}{c}\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{C{<}\overline{T}{>}} \\ \mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\{..\ class\ D{\triangleleft}P\{\}\ ..\}}\end{array}}{\mathit{bound}_\Delta(\mathtt{T.D}) = \mathtt{P}}$$

$$\frac{\begin{array}{c}\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{C{<}\overline{T}{>}} \\ \mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\{..\ \overline{I}\}} \qquad \mathtt{D} \notin \overline{\mathtt{I}}\end{array}}{\mathit{bound}_\Delta(\mathtt{T.D}) = \mathit{bound}_\Delta(\mathtt{N.D})}$$

Subtyping:

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

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

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

$$\frac{\mathtt{class\ C{<}\overline{X} \triangleleft \overline{N}{>}\triangleleft N\ \{..\ class\ D{\triangleleft}P\ \{\}\ ..\}} \qquad \mathtt{E} \trianglelefteq \mathtt{C}}{\Delta \vdash \mathtt{E{<}\overline{T}{>}.D} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{P}} \qquad \text{(S-DOT)}$$

Figure 10. X.FGJ subtyping rules

Typing require another additional auxiliary definition. The reduction $\mathcal{R}(\mathtt{T})$ of `T` is defined in Figure 12. The only interesting case is for `N.C` (the first and the second rules), which determines which class provides the implementation of the (empty) nested class. Otherwise, $\mathcal{R}$ behaves as an identity function.

Typing rules for expressions, methods, and classes are defined in Figure 13. The typing judgment for expressions is of the form $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$, read as "under 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,

Well-formed types:

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

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

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

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

Valid downcast:

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

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

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

Valid nested class overriding:

$$\frac{\forall\mathtt{C}, \Delta \vdash \mathtt{N.D} \unlhd \mathtt{C} \text{ implies } \Delta \vdash \mathtt{P} \unlhd \mathtt{C}}{override(\mathtt{D},\mathtt{N},\mathtt{P},\Delta)}$$

Valid method overriding:

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

Figure 11. X.FGJ type well-formedness rules

$$\frac{\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\{\mathtt{..}\ \mathtt{class\ D}\lhd\mathtt{P}\ \{\}\ \mathtt{..}\}}{\mathcal{R}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>.D}) = \mathtt{P}}$$

$$\frac{\mathtt{class\ E<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\{\mathtt{..}\ \bar{\mathtt{I}}\ \} \qquad \mathtt{D} \notin \bar{\mathtt{I}}}{\mathcal{R}(\mathtt{E<}\bar{\mathtt{T}}\mathtt{>.D}) = \mathcal{R}(\mathtt{N.D})}$$

$$\mathcal{R}(\mathtt{X.C}) = \mathtt{X.C}$$

$$\mathcal{R}(\mathtt{N}) = \mathtt{N}$$

$$\mathcal{R}(\mathtt{X}) = \mathtt{X}$$

Figure 12. Reduction of types

Expression typing:

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

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

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

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{A}\ ok \qquad fields(\mathtt{A}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}\\ \Delta;\Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{S}} \qquad \Delta \vdash \bar{\mathtt{S}} \mathrel{\mathtt{<:}} \mathcal{R}(\bar{\mathtt{T}})\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ A(}\bar{\mathtt{e}}\mathtt{)} : \mathtt{A}} \qquad (\text{T-NEW})$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash bound_\Delta(\mathtt{T}_0) \mathrel{\mathtt{<:}} \mathtt{N}}{\Delta;\Gamma \vdash \mathtt{(N)}\mathtt{e}_0 : \mathtt{N}} \qquad (\text{T-UCAST})$$

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

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ ok\\ \mathtt{N} = \mathtt{C<}\bar{\mathtt{T}}\mathtt{>} \qquad bound_\Delta(\mathtt{T}_0) = \mathtt{D<}\bar{\mathtt{U}}\mathtt{>}\\ \mathtt{C} \ntrianglelefteq \mathtt{D} \qquad \mathtt{D} \ntrianglelefteq \mathtt{C} \qquad stupid\ warning\end{array}}{\Delta;\Gamma \vdash \mathtt{(N)}\mathtt{e}_0 : \mathtt{N}} \qquad (\text{T-SCAST})$$

Method typing:

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

Nested class typing:

$$\frac{\begin{array}{c}\mathtt{class\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\ \{\mathtt{..}\ \mathtt{class\ D}\lhd\mathtt{P}\ \{\}\ \mathtt{..}\}\\ \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}} \vdash \mathtt{P}\ ok \qquad override(\mathtt{D}, \mathtt{N}, \mathtt{P}, \bar{\mathtt{X}}\mathtt{<:}\bar{\mathtt{N}})\end{array}}{\mathtt{class\ D}\lhd\mathtt{P}\ \{\}\ \mathtt{OK\ IN\ C<}\bar{\mathtt{X}} \lhd \bar{\mathtt{N}}\mathtt{>}} \qquad (\text{T-NEST})$$

Class typing:

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

Figure 13. X.FGJ typing rules

Computation:

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

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

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

Congruence:

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

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

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

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ A(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{new\ A(}\ldots\mathtt{,e}_i'\mathtt{,}\ldots\mathtt{)}} \quad \text{(RC-NEW-ARG)}$$

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

Figure 14. X.FGJ reduction rules

dcast(C,D) defined in Figure 11 ensures that the result of the cast will be the same at runtime.

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 X.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, each method declaration in the class is ok, and each nested class declaration is ok.

A class table CT is `OK` if all its definitions are `OK`.

4.4 Reduction

The operational semantics of X.FGJ is defined with the reduction relation that is of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces 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. 14[3]. 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 $[\bar{\mathtt{d}}/\bar{\mathtt{x}}, \bar{\mathtt{e}}/\bar{\mathtt{y}}]\mathtt{e}_0$ 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 `y` with `e`.

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

4.5 Properties

We show that X.FGJ's type system is sound with respect to the operational semantics. Type soundness is proved in the standard manner via subject reduction and progress [33, 17].

THEOREM 4.1 (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* $\Delta \vdash \mathtt{T}'\mathtt{<:T}$.

Proof sketch. We can prove the theorem by firstly proving the facts that type substitution preserves typing and term substitution preserves typing. The former's proof is almost identical to that of the corresponding lemma appearing in [17] using the fact that the definition of nested classes are covariantly refined. The latter's proof is also similar to those of the corresponding lemmas appearing in [17] and [31]. Then, we can prove that if $\mathit{mtype}(\mathtt{m}, \mathtt{A}) = \mathtt{<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\bar{\mathtt{U}} \rightarrow \mathtt{U}$ and $\mathit{mbody}(\mathtt{m<}\bar{\mathtt{V}}\mathtt{>}, \mathtt{A}) = \bar{\mathtt{x}}.\mathtt{e}_0$ where $\Delta \vdash \mathtt{A}\ \mathit{ok}$ and $\Delta \vdash \bar{\mathtt{V}}\ \mathit{ok}$ and $\Delta \vdash \bar{\mathtt{V}}\mathtt{<:}[\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{P}}$, then there exist some $\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}$ and `S` such that $\mathtt{A} \trianglelefteq \mathtt{C}$ and $\Delta \vdash \mathtt{C<}\bar{\mathtt{T}}\mathtt{>}\ \mathit{ok}$ and $\Delta \vdash \mathtt{S<:}[\bar{\mathtt{V}}/\bar{\mathtt{S}}]\mathtt{U}$ and $\Delta \vdash \mathtt{S}\ \mathit{ok}$ and $\Delta; \bar{\mathtt{x}} : [\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{U}}, \mathtt{this} : \mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}) \vdash \mathtt{e}_0 : \mathtt{S}$ by induction on the derivation of $\mathit{mbody}(\mathtt{m<}\bar{\mathtt{V}}\mathtt{>}, \mathtt{A}) = \bar{\mathtt{x}}.\mathtt{e}_0$, and if $\Delta; \bar{\mathtt{x}} : \bar{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$, then $\Delta; \bar{\mathtt{x}} : \mathcal{R}(\bar{\mathtt{T}}) \vdash \mathtt{e} : \mathcal{R}(\mathtt{T})$ by induction on the derivation of $\Delta; \Gamma, \bar{\mathtt{x}} : \bar{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$.

With these facts, we can prove the subject reduction by induction on the derivation of $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ with case analysis on the last rule used. We only show the case of T-INVK.

Case T-INVK: $\mathtt{e} = \mathtt{new\ A}_0\mathtt{(}\bar{\mathtt{e}}\mathtt{).m<}\bar{\mathtt{V}}\mathtt{>(}\bar{\mathtt{d}}\mathtt{)}$
$\mathtt{e}' = [\bar{\mathtt{d}}/\bar{\mathtt{x}}, \mathtt{new\ A}_0\mathtt{(}\bar{\mathtt{e}}\mathtt{)}/\mathtt{this}]\mathtt{e}_0$
$\mathit{mbody}(\mathtt{m<}\bar{\mathtt{V}}\mathtt{>}, \mathtt{A}_0) = \bar{\mathtt{x}}\mathtt{.e}_0$

By T-INVK and T-NEW, we have

$$\begin{array}{ll}
\Delta; \Gamma \vdash \mathtt{new\ A}_0\mathtt{(}\bar{\mathtt{e}}\mathtt{)} : \mathtt{A}_0 & \Delta; \Gamma \vdash \mathtt{e} : \mathtt{U} \\
\mathit{mtype}(\mathtt{m}, \mathtt{A}_0) = \mathtt{<}\bar{\mathtt{Y}} \lhd \bar{\mathtt{P}}\mathtt{>}\bar{\mathtt{U}} \rightarrow \mathtt{U} & \Delta \vdash \bar{\mathtt{V}}\ \mathit{ok} \\
\Delta \vdash \bar{\mathtt{V}}\mathtt{<:}[\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{P}} & \Delta; \Gamma \vdash \bar{\mathtt{d}} : \bar{\mathtt{S}} \\
\Delta \vdash \bar{\mathtt{S}}\mathtt{<:}[\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{U}}. &
\end{array}$$

There exists some $\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}$ and `S` such that

$$\begin{array}{ll}
\mathtt{A}_0 \trianglelefteq \mathtt{C} & \Delta \vdash \mathtt{C<}\bar{\mathtt{T}}\mathtt{>}\ \mathit{ok} \\
\Delta \vdash \mathtt{S}\ \mathit{ok} & \Delta \vdash \mathtt{S<:}[\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\mathtt{U} \\
\multicolumn{2}{l}{\Delta; \bar{\mathtt{x}} : [\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{U}}, \mathtt{this} : \mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}) \vdash \mathtt{e}_0 : \mathtt{S}.}
\end{array}$$

Then, we have

$$\Delta; \bar{\mathtt{x}} : \mathcal{R}([\bar{\mathtt{V}}/\bar{\mathtt{Y}}]\bar{\mathtt{U}}), \mathtt{this} : \mathcal{R}(\mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>})) \vdash \mathtt{e}_0 : \mathcal{R}(\mathtt{S}).$$

Finally, it is easy to show that $\Delta \vdash \mathtt{A}_0\mathtt{<:}\mathcal{R}(\mathit{thistype}(\mathtt{C<}\bar{\mathtt{T}}\mathtt{>}))$ provided that $\mathtt{A}_0 \trianglelefteq \mathtt{C}$, and $\Delta \vdash \mathcal{R}(\mathtt{S})\mathtt{<:S}$, finishing the case. □

THEOREM 4.2 (Progress). *Suppose* `e` *is a well-typed expression.*

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

Proof. Easy. □

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

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

THEOREM 4.3 (X.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 X.FGJ* *value* $\mathtt{v}$ *with* $\emptyset;\emptyset \vdash \mathtt{v} : \mathtt{S}$ *and* $\emptyset \vdash \mathtt{S}{<:}\mathtt{T}$ *or (2) an expression containing* $\mathtt{(P)new\ A(\bar{e})}$ *where* $\emptyset \vdash \mathtt{A}{<:}\mathtt{P}$*.*

Proof. Immediate from theorem 4.1 and 4.2. □

We may also show that if an expression $\mathtt{e}$ 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 4.4. *If* $\mathtt{e}$ *is cast-safe in* $\emptyset;\emptyset$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is a value* $\mathtt{v}$*.*

Proof. Straightforward. □

5. Related Work

Lightweight dependent classes partially support family polymorphism [11, 31]. 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 superclass of members in the extended family. On the other hand, in our approach, each member of family may be placed in a 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[31]. 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: [23, 25, 29, 16, 8, 5].

Virtual classes[21, 18, 14], also known as path-dependent types [12, 28], are also closely related to this direction of research. In virtual classes, classes are declared as fields of (enclosing) classes and they are referred from the outside of class declarations as instance variables. Our idea of parameterizing the enclosing class using a type parameter is analog of the dependent classes' idea of parameterizing the enclosing object using a constructor's parameter.

Besides path-dependent types, Scala[28] 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 separate source files is possible in Scala. Actually, Scala has much expressive power that lightweight dependent classes do not support. On the other hand, lightweight dependent classes are quite simple extension of Java 5.0 that has reasonable expressive power.

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 pieces of work, covariant subtyping of members is allowed. To ensure type safety, they introduce the notion of *exact types* and allow to invoke a method that takes an argument of the same family only when the receiver's family is *exactly* known.

Our approach is based on generic Java (and thus on Java 5.0), but there are other extensive researches on adding genericity to Java [27, 9, 2, 1]. How the result of this paper is integrated with these languages is not considered in this paper and worth pursuing. For example, [2] supports mixin inheritance by parameterization of superclass, which is not studied in this paper.

6. Concluding Remarks

We have proposed a language with lightweight dependent classes, an extension of generic Java where access to nested class members on type parameters is allowed. With this feature, we can parameterize the enclosing class by using type parameters, and separation of members and families is achieved. With the mechanism of type inference of `this`, mutually recursive classes can be safely extended without modifying existing source code. The resulting language is a quite simple extension of generic Java, and we do not have to declare a log of recursive type parameters and fixed-point classes with empty class body, which were required in some pieces of previous work. The core language X.FGJ ensures type soundness of the proposed language.

One remaining important issue is implementation. We believe that the idea can be implemented by adopting (an extension of) *erasure* of type parameters that is used in the implementation of generic Java. We are planning to implement the language using extensible Java compilers [24, 10]. Empirical studies on how the language can be used to refactor some industrial systems are also worth pursuing.

Acknowledgments. We thank Atsushi Igarashi and all the members of Kumiki2.0 project for helpful comments and fruitful discussion on language design in this paper. This work is supported in part by Grant-in-Aid for Scientific Research No.18200001 and Grant-in-Aid for Young Scientists (B) No.20700022 from NEXT of Japan.

References

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

[2] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *OOPSLA'03*, 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 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'04*, 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] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA'98*, pages 201–215, 1998.

[10] Torbjorn Ekman and Gorel Hedin. The JastAdd extensible Java compiler. In *OOPSLA'07*, pages 1–18, 2007.

[11] Eric Ernst. Family polymorphism. In *ECOOP'01*, volume 2072 of *LNCS*, pages 303–327, 2001.

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

[13] Erik Ernst. Higher-order hierarchies. In *ECOOP'01*, volume 2743 of *LNCS*, pages 303–326, 2003.

[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] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA'07*, pages 133–151, 2007.

[16] Stephan Hermann. Object Teams – improving modularity for crosscutting collaborations. In *Net Object Days 2002*, volume 2591 of *LNCS*, 2002.

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

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

[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] Tetsuo Kamina and Tetsuo Tamai. Lightweight scalable components. In *GPCE'07*, pages 145–154, 2007.

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

[22] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA'02*, pages 52–67, 2002.

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

[24] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of 12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, 2003.

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

[26] 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.

[27] 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.

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

[29] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP'02*, volume 2374 of *LNCS*, pages 89–110, 2002.

[30] Chieri Saito and Atsushi Igarashi. The essence of lightweight family polymorphism. In *FTfJP*, pages 27–41, 2007.

[31] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(3):285–331, 2008.

[32] 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.

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