Lightweight Nested Inheritance in Layer Decomposition

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

Software is often constructed as a stack of layers where a sublayer extends its superlayer. Such extension of layers requires extension of mutually recursive classes that form a layer. Furthermore, a class within a layer often inherits from another class within the same layer, and this inheritance relation is preserved in the sublayer. Supporting these features in a type-safe object-oriented languages imposes many challenges to us, and this issue has attracted many researchers. One problem of constructing a layer is that it can be a large monolithic module, thus a mechanism of decomposition of a large layer is required. We propose a programming language that supports decomposition of layers, which works even when a class within a layer inherits from another class within the same layer and thus supports extension of inheritance relations. This language is a very small extension of our previous work "lightweight dependent classes," thus this is a lightweight extension of Java. The language is formalized as a small calculus.

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 Family polymorphism, Generics, Lightweight dependent classes, Type system, Featherweight Java

1. Introduction

Software is often constructed as a stack of layers where a sublayer extends its superlayer [4–6, 26, 36][1]. Such extension of layers requires extension of mutually recursive classes that form a layer, and to support this feature in a type-safe object-oriented languages imposes many challenges to us, thus considerable research efforts such as *family polymorphism* and similar technologies have been proposed [8, 9, 11, 19, 27–29, 35, 38]. Furthermore, a class within a layer often inherits from another class within the same layer. The *nested inheritance*, which extends the inheritance relation within a layer to its sublayer, has been known as useful in many applications such as programming language processors[19, 27, 28].

One problem of constructing a layer is that it can be a large monolithic module. Each class implementation within a layer cannot be reused in other layers, and a layer cannot be composed from reusable smaller modules. Many efforts have been devoted to address this problem [15, 22, 23]. For example, in *dependent classes* [15], an instance of the layer becomes a formal parameter of the constructor of the enclosed classes. Likewise, we have also proposed language constructs that allow parametrization of a layer, which is not achieved by the parametrization of the constructor of enclosed class but the parametrization of class declaration by using type parameters. In these pieces of work, each inner class can be decomposed from its enclosing layer but type-safe extension of mutually recursive classes is possible.

However, these mechanisms do not work in the presence of inheritance within a layer. For example, in our previous work *lightweight dependent classes* [23], in the extended layer, each inner class explicitly inherits from the corresponding class of the original layer. Since it is not allowed to inherits from multiple classes, the inner class cannot extend another class of the same layer.

In this paper, we propose a programming language that supports decomposition of layers, which works even when a class within a layer inherits from another class within the same layer and thus supports extension of inheritance relations. This language is a very small extension (or modification) of our previous work lightweight dependent classes. In this proposal, in the `extends` clause that specifies the superclass, we can use a type like `L.Expr`, where `L` is a type parameter and `Expr` is an inner class that can be accessed on `L`. Since the layer is parametrized, the actual superclass is resolved at the layer composition time. Thus the inheritance relation can be changed with respect to the layer composition so that extension of inheritance relation is supported. The language is formalized as a small calculus based on Featherweight Java (FJ) [17], but this formalization is almost identical to that of [23]. We summarize this formalization in appendix A.

Contributions of this paper are as follows:

- An extension of our previous work lightweight dependent classes that supports subclassing of an inner class whose enclosing layer is parametrized (and thus the actual implementation is provided at the instantiation time of that type parameter).
- A proposal of programming technique using the proposed language for the decomposition of layers enabling safe extension of mutually recursive classes including inheritance relations.

The rest of this paper is structured as follows. In section 2, we briefly introduce the earlier pieces of work and discuss their problems. We then informally describe our idea by using a example

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $10.00

[1] Throughout this paper, we use the term "layer" in a consistent way with that in mixin layers [36]. A layer is a group of closely related classes, and sublayers can extend the functionality of superlayers.

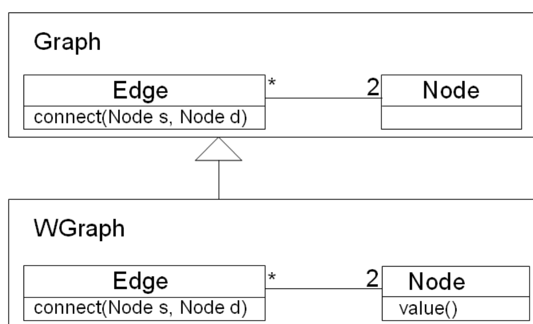

Figure 1. A layer extension

of simplified programming language processor taken from [19] in section 3. In section 4, we discuss other related work. Finally, section 5 concludes this paper.

2. Problem

2.1 An Overview of Our Previous Work

A layer consists of a set of (inner) classes that are mutually recursive. When we extend the layer, we define a new layer that consists of extensions of each inner classes. Figure 1 illustrates very simplified form of such a layered design. This example features a `Graph` that consists of mutually recursive classes, namely `Node` and `Edge`; each instance of `Node` holds references to its incident edges (instances of `Edge`), and each edge holds references to its source and destination nodes. Figure 1 also illustrates an extension of `Graph`, namely `WGraph`, which adds a feature of setting weight of each edge to `Graph`. In `WGraph`, the inner class `Edge` is refined to store the weight of edge, and the inner class `Node` is refined to store a property (e.g., a color or a label of the node). The extended `Node` declares an abstract method `value()` to return a normalized value of this property. In this program, the weight of edge is calculated by using this property of the pair of nodes connected by the edge. Thus, the `connect()` method have to be appropriately overridden.

Although this is a quite simple example, it explains a shortcoming of the existing object-oriented languages; it is very hard to extend such mutually recursive classes *at once*. 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.

Another problem of layered design is that, in many solutions to this problem, mutually recursive classes are programmed as inner classes of a top-level class, thus each layer can be a large monolithic program. There exist some pieces of work that address this problem. A common capability shared by them is to separate the "inner" classes from the enclosing layer. To illustrate this feature, we show an example program taken from [23] in Figure 2 and Figure 3, which implement the graph example shown in Figure 1.

Figure 2 shows a simple graph definition. A layer `Graph` is implemented by a class `Graph` that consists of two inner classes, namely `Edge` and `Node`, but each of their body is empty. The implementation is separately provided by their superclasses `EdgeI` and `NodeI`, respectively. `EdgeI` and `NodeI` are parametrized over its enclosing layer by using type parameter (namely `G`), whose upper bound is restricted by `Graph`. The notable feature is its ability to access the inner classes on the type parameter, in the form of `G.Node`. Thus, the corresponding inner class can be accessed through the type parameter, but the *actual* class is lately provided at the instantiation time of the parametrized class. This late binding of enclosing layer enables safe extension of the layer `Graph` to its

```
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 2. Simple graph definition

```
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 3. Weighted graph extension

extension `WGraph` shown in Figure 3. In `WGraph`, the upper bound of type parameter is refined to `WGraph`, thus the type `G.Node` is ensured to be compatible with `WGraph.Node`. For the type safety, some restrictions (e.g., the body of `Edge` and `Node` have to be empty) and type inference rules for `this` (originally developed in [22] and [34]) are proposed in [23].

2.2 Problem Description

An important limitation of this approach is that we cannot represent inheritance between classes *within* a layer. To show the problem, we use Figure 4 that is taken from [19]. This diagram shows a layered design of a language processor. The upper layer, namely `AST`, provides an abstraction of abstract syntax tree, which consists of three inner classes, namely `Expr`, `Const`, and `Plus`. Since both `Const` and `Plus` are kinds of `Expr`, they inherit from `Expr`. The layer below, namely `ASTeval`, is an extension of `AST` that implements a feature of evaluation of each node. Thus, `ASTeval.Expr`

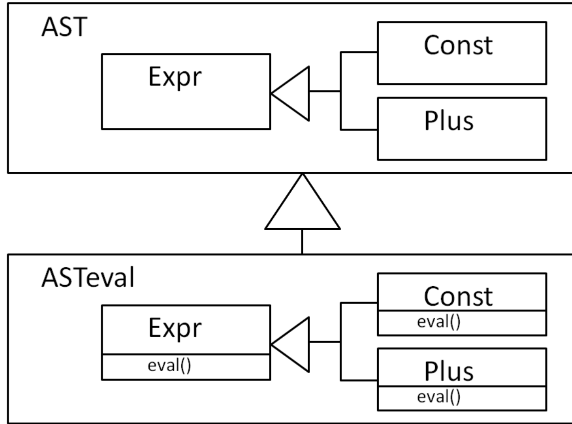

Figure 4. Layers for a language processor

provides an abstract eval method and both ASTeval.Const and ASTeval.Plus inherit and override it. Furthermore, both of them also inherit from AST.Const and AST.Plus, respectively, thus we need multiple inheritance in this case, which is not supported by [23].

Introducing multiple inheritance raises another problem. Since AST.Const and AST.Plus also inherit from AST.Expr, AST.Expr is inherited twice in ASTeval.Const and ASTeval.Plus. This is particularly problematic when AST.Expr has fields. Virtual inheritance in C++ is one solution to ensure that the subclass inherits only one copy of AST.Expr's field [37], but it also raises a problem in object initializers.

Another solution to this diamond problem is to linearize the order of method dispatch to ensure that the methods declared in AST.Expr are dispatched only once. Even though this linearization approach is taken by many languages[16, 27], it is not a best solution in our case; if the linearization approach is taken, the super call may result in a method call that is not declared in the declared super class (for example, ASTeval.Expr's super class is AST.Expr, but the super call can result in a method call declared in AST.Plus). This implicit change of the "actual" superclass sometimes results in behaviors that are not intended by the class implementer.

3. Informal Description of Our Proposal

To address the aforementioned problem, we propose a new programming language mechanism that supports decomposition of layers, which works even when a class within a layer inherits from another class within the same layer. This is a very small extension (or modification) of [23]. In [23], we can access an inner class on a type parameter, and the actual class referred by it is lately resolved at the instantiation time of the parametric class. In this paper, we apply this technique to the declaration of superclass; a class can inherits from an inner class on a type parameter whose actual type is lately provided at the instantiation time of the parametric class.

In this section, we explain the feature by using the example shown in Figure 4.

3.1 Declaring Members of a Layer

Figure 5 shows our implementation of inner classes of the layer AST, namely ExprT, ConstT and PlusT, which are actually superclasses of the inner classes (see Figure 6). Each of them corresponds to Expr, Const, and Plus in our layered design of language processor shown in Figure 4, respectively. In these classes, the enclosing layer is parametrized by the type parameter L, which makes each class reusable in another layer that is compatible with AST.

```
class ExprT<L extends AST> {
  String format() { return ""; }
}
class ConstT<L extends AST> extends L.Expr {
  int val;
  String format() { return ""+val; }
}
class PlusT<L extends AST> extends L.Expr {
  L.Expr op1, op2;
  String format() {
    return op1.format()+"+"+op2.format();
  }
  void replaceOp1(L.Expr e) { op1 = e; }
}
```

Figure 5. Members of the AST layer

```
class AST {
  class Expr extends ExprT<AST> {}
  class Const extends ConstT<AST> {}
  class Plus extends PlusT<AST> {}
}
```

Figure 6. AST class

PlusT declares two fields op1 and op2 corresponding to two operands for the plus expression. The type of them is parametrized over the enclosing layer L, like L.Expr. Thus the actual type of them is lately provided at the instantiation of the parametrized class PlusT. Likewise, it declares a method whose formal parameter type is L.Expr, which replaces its first operand with the provided expression.

Since both constant and plus expression are expressions, ConstT and PlusT inherit from L.Expr. This is our new proposal that is not provided by [23], since in [23], an actual superclass have to be determined at the class declaration time. In this paper, on the other hand, we can specify an inner class accessed on a type parameter as a superclass, thus the actual superclass will be determined at the class instantiation time. This flexibility provides an ability to extend inheritance relation when the layer is extended.

The classes shown in Figure 5 are composed into a concrete layer AST shown in Figure 6. Each type parameter L is instantiated by AST inside the body of AST itself.

3.2 Extending Members

Now, we extend the AST layer to provide a feature of evaluation of each AST node. For this purpose, we add the eval() method to the Expr class. Figure 7 shows this extension. In this extension, the upper bound of type parameter of EPlusT is now refined to an extension of AST layer, namely ASTeval, shown in Figure 8.

EExprT is a subclass of ExprT that provides a new method eval(). EConstT and EPlusT are also subclasses of ConstT and PlusT, respectively. EPlusT overrides the eval() method, and inside the body of it, the eval() method is called on the fields op1 and op2. These fields are inherited from PlusT, but these method invocations are safe, because the type of op1 and op2 is declared as L.Expr, and in the context of Figure 7, this type is compatible with ASTeval.Expr that provides the eval() method. This mechanism for extending mutually recursive classes is exactly the same as that of [23]. Now, these classes are composed into a layer shown in Figure 8.

```
class EExprT<L extends AST>
      extends ExprT<L> {
  int eval() { return 0; }
}
class EConstT<L extends AST>
      extends ConstT<L> {
  int eval() { return val; }
}
class EPlusT<L extends ASTeval>
      extends PlusT<L> {
  int eval() {
    return op1.eval() + op2.eval(); }
}
```

Figure 7. Members of the extended AST layer

```
class ASTeval extends AST {
  class Expr extends EExprT<ASTeval> {}
  class Const extends EConstT<ASTeval> {}
  class Plus extends EPlusT<ASTeval> {}
}
```

Figure 8. Extended AST class

```
class EConstT2 <L extends AST>
      extends EConstT<L> {
  ...
}

class ASTeval2 extends ASTeval {
  class Expr extends EExprT<ASTeval2> {}
  class Const extends EConstT2<ASTeval2> {}
  class Plus extends EPlusT<ASTeval2> {}
}
```

Figure 9. Another combination of extensions

Note that in Figure 8, both `ASTeval.Const` and `ASTeval.Plus` inherit from `ASTeval.Expr`. Therefore, the inheritance relation within `AST`, i.e., `Expr` is inherited by `Const` and `Plus`, is maintained in `ASTeval`, thus it is assured that some newly provided features by `ASTeval.Expr` are also provided by `ASTeval.Const` and `ASTeval.Plus`. The reason why this extension of inheritance relation is possible is that, in the superclass of `EPlusT` (and `EConstT`), the enclosing layer is parametrized so that the actual superclass is resolved as `ASTeval.Expr` when the parametrized class is instantiated by `ASTeval`.

Since this approach is not use multiple inheritance, the diamond problem is not occurred. This approach is actually somewhat similar to mixins [7] in the sense that the superclass is parametrized. Actually, in the definition of `ConstT` and `PlusT`, the implementer *expects* that some subclass of them will refine the upper bound of type parameter `L` so that some class that is more specific than `AST.Expr` is mixed-in the inheritance chain.

3.3 Constructing a Series of Layers

So far, we have presented that a mechanism of layer decomposition that works even when a class within a layer inherits from another class within a layer. Now, we briefly review how this mechanism can flexibly compose a series of layers from the decomposed components of layers.

Figure 9 shows that there may be another implementation of constant node in `ASTeval`, namely `EConstT2`. The new layer `ASTeval2` demonstrates that `EConstT2` is also composed with `EExprT` and `EPlusT`. This modification is local to implementation of constant node and does not affect development of other parts of the AST implementation. Furthermore, we can also compose the extended class with the original layer. For example, we can form a layer where only the constant node provides the `eval()` method:

```
class AST2 extends AST {
  class Expr extends ExprT<AST2> {}
  class Const extends EConstT<AST2> {}
  class Plus extends PlusT<AST2> {}
}
```

Even though this is just a toy example, such ability to flexibly compose a series of layers from the "modularized inner classes" will help the development of much larger program, especially when the modularization is systematically preplanned in a process such as software product lines [33].

3.4 Notes on Typing Rules

To close this section, we finally discuss some issues on typing rules.

Restriction on inner classes. In Figure 6 and 8, both `AST` and `ASTeval` declare inner classes `Expr`, `Const` and `Plus`. How does a class from the base layer and another class with the same name from the extended layer relate to each other? For example, is `ASTeval.Plus` a subtype of `AST.Plus`?

For type safety, this subtyping is prohibited. To show the reason, consider the following demonstration code:

```
AST.Const c1 = new AST.Const();
ASTeval.Plus p1 = new ASTeval.Plus();
AST.Plus p2 = p1;
p2.replaceOp1(c1);
p1.eval();  // error!
```

This code assigns the value of `p1` to the variable `p2`. If the type-checker accepts this code, i.e., `ASTeval.Plus` is a subtype of `AST.Plus` (thus the assignment is allowed), the type of `c1` is compatible with the formal parameter type of `replaceOp1` invoked on p2. However, the runtime type of `p1` is `ASTeval.Plus` and thus the invocation `p2.eval()` results in an invocation of `eval()` method on the instance referred by `c1`, which results in the "method not understood" error. Thus, the type-checker reports an error when analyzing the assignment `p2 = p1`. This restriction on subtyping can also be understood in another way; `ASTeval.Plus` is a subclass of `EPlusT<ASTeval>`, while `AST.Plus` is a subclass of `PlusT<AST>`, and both classes are not compatible in type system of Java 5.0.

However, this restriction raises another problem; if an inner class declares some fields and methods, the access to those fields and methods will not be safe. For example, if `AST.Expr` declares a field `Object f`, a field access expression `e.f`, where `e`'s static type is `L.Expr` and `L` is a type parameter whose upper bound is `AST`, is not always safe, because `L` can be instantiated by `ASTeval` and `ASTeval.Expr` no longer provides that field. To address this field, the type system requires that the body of inner class must always be empty. Although this restriction seems to be too strict, we may easily relax it. For example, we may introduce a new modifier that indicates the inner class can be accessed on type parameters and thus may not declare any members. We may still declare non empty inner classes that cannot accessed on type parameters. For simplicity, in this paper, we omit this annotations and treat that all the inner classes have an empty body.

Restriction on parametric superclass. In Figure 8, each inner class overrides the corresponding inner class of the superlayer (for example, `ASTeval.Expr` overrides `AST.Expr`). The superclass of the overriding inner class is a subclass of that of overridden inner

class (for example, EExprT is a subclass of ExprT). This subclassing is required by the type system, since without it, the type system cannot ensure the type safety. For example, consider the following declaration:

```
class ASTfoo extends AST {
  class Expr extends Foo {}
  ..
}
```

In this case, ConstT and PlusT declared in Figure 5 have no relations with ExprT when their type parameter is instantiated by ASTfoo. Especially, the type of PlusT's fields op1 and op2 become a subtype of Foo and thus the method invocation of format() on these fields raises an error. To avoid this problem, it is necessary to ensure that L.Expr is always a subclass of ExprT.

Type inference for this. Another issue on type system is the type of this. In the type system of Java, this can be treated as an expression whose type is its enclosing class. However, as discussed in [22, 23], in the support of extension of mutually recursive classes, the type of this has also to be extended. The detailed discussion is found in [23]; in this paper, we briefly resummarize the type inference rule for this as follows:

- If one of the type parameters, namely X, has an upper bound that is a class that declares an inner 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 the class D is declared as

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

  then the type of this within C is X.E.
- Otherwise, the type of this is its enclosing class.

Note that if the superclass of the inner class is instantiated by the *subclass* of the enclosing class, the first case of the type inference rule is not applied. To ensure the type safety, it is required that to apply the first rule of the inference, the superclass of the inner class is instantiated *exactly* the same class as the enclosing class.

4. Related Work

As stated earlier, our approach supports extension of mutually recursive classes like family polymorphism [11, 35], enabling decomposition of mutually recursive classes from the enclosing layer. This feature comes with a price that, in our approach, we have to invent new names for implementing class (i.e., the super class) of the inner classes when the layer is extended. Originally, inner 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 [20], Saito et al. propose a much simpler variant of family polymorphism, in which families are identified with classes[35]. 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.

Virtual classes[13, 18, 24], also known as path-dependent types [12, 31], and Delegation Layers [32], 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. Delegation Layers combine the mechanism of delegation and virtual classes where composition of layers is instance-based using the delegation mechanism. Our idea of parametrizing the enclosing class using a type parameter is analog of the dependent classes' idea of parametrizing the enclosing object using a constructor's parameter.

This paper only shows layers that contain inner classes containing no other deeply nested classes. Variant path types for arbitrarily deep nested classes are studied in [19]. Our work cannot scale well in this manner. The examples shown in this paper only allow nesting to be 1-level. If we encode such arbitrarily deep nesting structure in this work, we have to parametrize all the levels of outer classes in the declaration of deep inner classes. How to address this issue remains as future work.

Avoiding the diamond inheritance is also studied by Malayeri [25]. However, as mentioned in section 3.2, our approach of parametrization of superclass to avoid the diamond inheritance is much similar to mixins [7]. There are many extensions of Java that support mixins [3, 14, 21]. Instead, our approach is based on genericity. Even though many pieces of work have been devoted to add genericity to Java [1, 2, 10, 30], parametrization of superclass is not so common in them. A notable exception is NextGen [2] that supports parametrization of superclass and thus supports mixin-based inheritance. A similar programming technique using C++ template is also studied in [36, 39]. In our approach, a type parameter can also be used to represent an enclosing class that is not known at the implementation time of the inner class.

5. Concluding Remarks

A programming language that is a small extension of our previous work "lightweight dependent classes" is proposed, which supports subclassing of an inner class whose enclosing layer is parametrized. The proposed language is built as a very small extension of Java with generics. Using this language, a programming technique for decomposing mutually recursive classes from the layer, which works even when a class within a layer inherits from another class within the same layer, is proposed. This makes it possible to flexibly compose a series of layers from the decomposed components of layers. The core of language is formalized based on FJ. This formalization provides a solid information for compiler implementation and proof of type soundness, which indicate our direction of future work.

Acknowledgments. This work is supported in part by Grant-in-Aid for Young Scientists (B) No.20700022 and Grant-in-Aid for Scientific Research No.22240002 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] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.

[4] Sven Apel, Christian Kästner, and Christian Lengauer. Feature featherweight java: A calculus for feature-oriented programming and stepwise refinement. In *GPCE'08*, pages 101–111, 2008.

[5] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Feature C++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE'05*, volume 3676 of *LNCS*, pages 125–140, 2005.

[6] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: Aspects and features in concert. In *ICSE'06*, pages 122–131, 2006.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

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

[9] 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.

[10] 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.

[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, 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.

[14] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 98*, pages 171–183, 1998.

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

[16] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *ECOOP'02*, pages 62–88, 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] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *OOPSLA'07*, pages 113–132, 2007.

[20] 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.

[21] Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of Java with mixin-types. In *2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 398–414. Springer, 2004.

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

[23] Tetsuo Kamina and Tetsuo Tamai. Lightweight dependent classes. In *GPCE'08*, pages 113–124, 2008.

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

[25] Donna Malayeri and Jonathan Aldrich. CZ: Multiple inheritance without diamonds. In *OOPSLA'09*, pages 21–39, 2009.

[26] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-12)*, pages 127–136, 2004.

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

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

[29] 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.

[30] 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.

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

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

[33] Klaus Pohl, Gunter Bockle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005.

[34] Chieri Saito and Atsushi Igarashi. The essence of lightweight family polymorphism. *Journal of Object Technology*, 7(5):67–99, 2008.

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

[36] 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.

[37] Bjarne Stroustrap. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[38] 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.

[39] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.

Syntax:

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

Subclassing:

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

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

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

Figure 10. Syntax and subclassing

A. Formalization

In this section, we formalize the idea described in the previous section. This formalization is built on top of FJ [17].

A.1 Syntax

Abstract syntax is shown in Figure 10. The metavariables `T`, `S`, `V`, and `U` range over types; `X`, `Y`, `Z`, and `W` range over type variables; `N`, `O`, `P`, and `Q` range over nonvariable types; `A` ranges over instantiatable types; `C`, `D`, and `E` range over class names; `L` ranges over class declarations; `K` ranges over constructor declarations; `M` ranges over method declarations; `I` ranges over inner class declarations; `f` and `g` range over field names; `m` ranges over method names; `x` and `y` range over variables; `d` and `e` range over expressions.

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

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

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{S } \{\texttt{.. class D}\lhd\texttt{P } \{\}\ \texttt{..}\} \quad \text{fields}([\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{P}) = \bar{\texttt{U}}\ \bar{\texttt{g}} \quad \texttt{E} \trianglelefteq \texttt{C}}{\text{fields}(\texttt{E<}\bar{\texttt{T}}\texttt{>.D}) = \bar{\texttt{U}}\ \bar{\texttt{g}}} \qquad \text{(F-NEST)}$$

Figure 11. Field lookup function

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.

We abbreviate the keyword `extends` to the symbol $\lhd$. A class can extend any kinds of types, including type parameters and inner class accessed on type parameter. We assume that the set of variables includes the special variable `this`, which is considered to be implicitly bound in every method declaration. Our calculus 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, and a method body consists of single `return` statement.

Subclassing is also shown in Figure 10, which is represented by the relation $\texttt{C} \trianglelefteq \texttt{T}$ between a class and a type. This is a reflexive and transitive closure induced by the clause $\texttt{C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>} \lhd \texttt{D<}\bar{\texttt{T}}\texttt{>}$.

A 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 their corresponding class declarations. The expression `e` is considered as the `main` method of the real 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) $CT(\texttt{C}) =$`class C ...` for every $\texttt{C} \in \text{dom}(CT)$; (2) $\texttt{Object} \notin \text{dom}(CT)$; (3) $\texttt{C} \in \text{dom}(CT)$ for every class name appearing in $\text{ran}(CT)$; (4) there are no cycles in subclass relations induced by CT.

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

A.2 Auxiliary definitions

We show some auxiliary definitions that are required for typing rules. The function $\text{fields}(\texttt{N})$, shown in Figure 11, is a sequence $\bar{\texttt{T}}\ \bar{\texttt{f}}$ of field types and field names declared in `N`. Application of type substitution $[\bar{\texttt{T}}/\bar{\texttt{N}}]$ is defined in the customary manner. We write $\texttt{C} \notin \bar{\texttt{I}}$ to mean that the inner class definition of the name `C` is not included in $\bar{\texttt{I}}$. The function fields returns all the fields declared in the class (including the superclasses). In the case of inner class, the fields are searched even when the inner class is declared in the superclass of the top-level class.

The type of method `m` at `N`, written $\text{mtype}(\texttt{m}, \texttt{N})$, is defined in Figure 12. It is a type of the form $\texttt{<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}$. Just as in Java's

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

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{V } \{\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K } \bar{\texttt{M}}\} \quad \texttt{m} \notin \bar{\texttt{M}} \\ \text{mtype}(\texttt{m}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{V}) = \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}\end{array}}{\text{mtype}(\texttt{m}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}} \qquad \text{(MT-SUPER)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{V } \{\texttt{.. class D}\lhd\texttt{Q } \{\}\ \texttt{..}\} \\ \text{mtype}(\texttt{m}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{Q}) = \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}\end{array}}{\text{mtype}(\texttt{m}, \texttt{C<}\bar{\texttt{T}}\texttt{>.D}) = \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}} \qquad \text{(MT-NEST)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{V } \{\texttt{.. } \bar{\texttt{I}}\} \quad \texttt{D} \notin \bar{\texttt{I}} \\ \text{mtype}(\texttt{m}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{V.D}) = \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}\end{array}}{\text{mtype}(\texttt{m}, \texttt{C<}\bar{\texttt{T}}\texttt{>.D}) = \texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{U}} \to \texttt{U}} \qquad \text{(MT-NEST-SUPER)}$$

Figure 12. Method type lookup function

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

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{W } \{\bar{\texttt{S}}\ \bar{\texttt{f}}\texttt{; K } \bar{\texttt{M}}\} \quad \texttt{m} \notin \bar{\texttt{M}} \\ \text{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{W}) = \bar{\texttt{x}}.\texttt{e}\end{array}}{\text{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, \texttt{C<}\bar{\texttt{T}}\texttt{>}) = \bar{\texttt{x}}.\texttt{e}} \qquad \text{(MB-SUPER)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{W } \{\texttt{.. } \bar{\texttt{M}}\texttt{ class D}\lhd\texttt{Q } \{\}\ \texttt{..}\} \\ \texttt{m} \notin \bar{\texttt{M}} \quad \text{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{Q}) = \bar{\texttt{x}}.\texttt{e}\end{array}}{\text{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, \texttt{C<}\bar{\texttt{T}}\texttt{>.D}) = \bar{\texttt{x}}.\texttt{e}} \qquad \text{(MB-NEST)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{W } \{\texttt{.. } \bar{\texttt{I}}\ \} \quad \texttt{D} \notin \bar{\texttt{I}} \\ \text{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{W.D}) = \bar{\texttt{x}}.\texttt{e}\end{array}}{\text{mbody}(\texttt{m<}\bar{\texttt{V}}\texttt{>}, \texttt{C<}\bar{\texttt{T}}\texttt{>.D}) = \bar{\texttt{x}}.\texttt{e}} \qquad \text{(MB-NEST-SUPER)}$$

Figure 13. Method body lookup functions

type system, class declaration is searched for finding the type of method, and if the method is not declared in the current searched class, then the superclass is searched. We write $\texttt{m} \notin \bar{\texttt{M}}$ to mean that the method definition of the name `m` is not included in $\bar{\texttt{M}}$. Similarly the body of method `m` at `N`, written $\text{mbody}(\texttt{m}, \texttt{N})$, is defined in Figure 13. It is a pair, written $\bar{\texttt{x}}.\texttt{e}$, of a sequence of parameters $\bar{\texttt{x}}$ and an expression `e`.

Type reduction rules and type inference rules for `this` are defined in Figure 14. The reduction $\mathcal{R}(\texttt{T})$ returns the class that provides the implementation of the (empty) inner class. The only interesting case is for `N.C` (the first and the second rules). Otherwise, $\mathcal{R}$ behaves as an identity function. The function thistype returns the

Type reduction:

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

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

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

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

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

Type inference for `this`:

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{W}\{\texttt{..}\} \quad \texttt{N}_i = \texttt{D<}\bar{\texttt{S}}\texttt{>} \quad \texttt{U}_i = \texttt{D<}\bar{\texttt{S}}\texttt{>}\\ \texttt{class D<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\lhd\texttt{V}\{\texttt{.. class E}\lhd\texttt{C<}\bar{\texttt{U}}\texttt{>\{\} ..}\}\end{array}}{\text{thistype}(\texttt{C<}\bar{\texttt{T}}\texttt{>}) = \texttt{T}_i\texttt{.E}}$$

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{V}\{\texttt{..}\}}{\text{thistype}(\texttt{C<}\bar{\texttt{T}}\texttt{>}) = \texttt{C<}\bar{\texttt{T}}\texttt{>}}$$

Figure 14. Type reduction and type inference for `this`

$$\text{bound}_\Delta(\texttt{X}) = \Delta(\texttt{X})$$

$$\text{bound}_\Delta(\texttt{C<}\bar{\texttt{T}}\texttt{>}) = \texttt{C<}\bar{\texttt{T}}\texttt{>}$$

$$\frac{\begin{array}{c}\text{bound}_\Delta(\texttt{X}) = \texttt{C<}\bar{\texttt{T}}\texttt{>}\\ \texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{V}\{\texttt{.. class D}\lhd\texttt{E<}\bar{\texttt{S}},\texttt{C<}\bar{\texttt{T}}\texttt{>},\bar{\texttt{U}}\texttt{>\{\}..}\}\end{array}}{\text{bound}_\Delta(\texttt{X.D}) = \texttt{E<}\bar{\texttt{S}},\texttt{X},\bar{\texttt{U}}\texttt{>}}$$

$$\frac{\begin{array}{c}\text{bound}_\Delta(\texttt{T}) = \texttt{C<}\bar{\texttt{T}}\texttt{>}\\ \texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{S}\{\texttt{.. class D}\lhd\texttt{P\{\} ..}\}\end{array}}{\text{bound}_\Delta(\texttt{T.D}) = \texttt{P}}$$

$$\frac{\begin{array}{c}\text{bound}_\Delta(\texttt{T}) = \texttt{C<}\bar{\texttt{T}}\texttt{>}\\ \texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{S}\{\texttt{.. }\bar{\texttt{I}}\} \quad \texttt{D} \notin \bar{\texttt{I}}\end{array}}{\text{bound}_\Delta(\texttt{T.D}) = \text{bound}_\Delta(\texttt{S.D})}$$

Figure 15. Bound of type

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. To ensure the type safety, the first rule is applied only when the superclass of the inner class is instantiated *exactly* the same class as the enclosing class (further discussion can be found in [23]).

A.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 15, we write $\text{bound}_\Delta(\texttt{T})$ for an upper bound of `T` in Δ. Besides

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

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

$$\frac{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{S }\{\texttt{...}\}}{\Delta \vdash \texttt{C<}\bar{\texttt{T}}\texttt{> <: } [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{S}} \quad \text{(S-CLASS)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}}\texttt{>}\lhd\texttt{S }\{\texttt{.. class D}\lhd\texttt{P\{\}..}\} \qquad \texttt{E} \trianglelefteq \texttt{C}\\ \texttt{E} \trianglelefteq \texttt{C}_0 \text{ and } \texttt{C}_0 \trianglelefteq \texttt{C} \text{ implies } \texttt{class C}_0\texttt{<..>..}\{\texttt{..}\bar{\texttt{I}}\} \text{ and } \texttt{D} \notin \bar{\texttt{I}}\end{array}}{\Delta \vdash \texttt{E<}\bar{\texttt{T}}\texttt{>.D <: } [\bar{\texttt{T}}/\bar{\texttt{X}}]\texttt{P}} \quad \text{(S-DOT)}$$

Figure 16. Subtyping rules

type parameters and nonvariable types, we also need to define bound_Δ of inner class type `T.D`. Note that the upper bound of `T.D` is its superclass, since the body of inner class is always empty. Furthermore, there is a special case for inner class type `D` on type variable `X`, i.e., `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 defined in Figure 16. Subtyping is the reflexive and transitive closure of the `extends` relation. If a class overrides an inner class, the type of that inner class is subtype of the superclass of the *lower* most inner class.

We write $\Delta \vdash \texttt{T}\ ok$ if a type `T` is well formed in context Δ. The rules for well-formed types appear in Figure 17. 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. An inner class type `T.D` is well-formed if the path type `T` is ok and the inner 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 $dom(\Gamma)$. The function $\text{override}(\texttt{D},\texttt{N},\texttt{P},\Delta)$ indicates the condition of valid overriding of inner classes. This condition is true if there exists a super type of `N.D` in environment Δ, it must be a super type of `P`. The function $\text{override}(\texttt{m},\texttt{N},\texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\bar{\texttt{T}} \rightarrow \texttt{T}_0)$ ensures covariant overriding on the method result type.

Typing rules for expressions are shown in Figure 18. The typing judgment for expressions is of the form $\Delta;\Gamma \vdash \texttt{e} : \texttt{T}$, 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. These rules are straightforward. The typing rules for constructor and method invocations check that the type of each argument expression is a subtype of the corresponding formal parameter type.

Typing rules for method and class declarations are defined in Figure 19. 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 function thistype infers the type of `this`, and the predicate override ensures the covariant overriding on the method result type. The typing judgment for inner class declarations has the form `I OK`

Well-formed types:

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

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

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

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

Valid inner class overriding:

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

Valid method overriding:

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

Figure 17. Type well-formedness rules and valid method overriding

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

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

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

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

Figure 18. Expression typing

Method typing:

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

Inner class typing:

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

Class typing:

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

Figure 19. Method and class typing

IN C, read "inner class declaration I is ok when it occurs in class C." 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 inner class declaration is ok.

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

A.4 Reduction

The operational semantics 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 Figure 20. As in the original FJ, we use the non-deterministic reduction strategy. There are two computation rules, one for field access and another for method invocation. 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 y with e.

A.5 Property

We are now preparing the proof of type soundness of the calculus presented in this paper. This can be done by proving the following three theorems.

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

THEOREM A.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* $\text{fields}(\mathtt{A}_0) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}$ *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* $\text{mbody}(\mathtt{m<}\bar{\mathtt{V}}\mathtt{>}, \mathtt{A}_0) = \bar{\mathtt{x}}.\mathtt{e}_0$ *and* $\#(\bar{\mathtt{x}}) = \#(\bar{\mathtt{d}})$ *for some* $\bar{\mathtt{x}}$ *and* $\mathtt{e}_0$.

Computation:

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

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

Congruence:

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

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

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

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

Figure 20. Reduction rules

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

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

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