Lightweight Family Polymorphism

Atsushi Igarashi[1], Chieri Saito[1], and Mirko Viroli[2]

[1] Kyoto University, Japan, {igarashi,saito}@kuis.kyoto-u.ac.jp
[2] Alma Mater Studiorum, Università di Bologna a Cesena, Italy,
mviroli@deis.unibo.it

Abstract. Family polymorphism has been proposed for object-oriented languages as a solution to supporting reusable yet type-safe mutually recursive classes. A key idea of family polymorphism is the notion of families, which are used to group mutually recursive classes. In the original proposal, due to the design decision that families are represented by objects, dependent types had to be introduced, resulting in a rather complex type system. In this paper, we propose a simpler solution of *lightweight* family polymorphism, based on the idea that families are represented by classes rather than objects. This change makes the type system significantly simpler without losing much expressibility of the language. Moreover, "family-polymorphic" methods now take a form of parametric methods; thus it is easy to apply the Java-style type inference. To rigorously show that our approach is safe, we formalize the set of language features on top of Featherweight Java and prove the type system is sound. An algorithm of type inference for family-polymorphic method invocations is also formalized and proved to be correct.

1 Introduction

Mismatch between Mutually Recursive Classes and Simple Inheritance. It is fairly well-known that, in object-oriented languages with simple name-based type systems such as C++ or Java, mutually recursive class definitions and extension by inheritance do not fit very well. Since classes are usually closed entities in a program, mutually recursive classes here really mean a set of classes whose method *signatures* refer to each other by their *names*. Thus, different sets of mutually recursive classes necessarily have different signatures, even though their structures are similar. On the other hand, in C++ or Java, it is not allowed to inherit a method from the superclass with a different signature (in fact, it is not safe in general to allow covariant change of method parameter types). As a result, deriving subclasses of mutually recursive classes yields another set of classes that do *not* refer to each other and, worse, this mismatch is often resolved by typecasting, which is a potentially unsafe operation (not to say unsafe, an exception may be raised). A lot of studies [6, 8, 11, 14, 16, 19, 3, 18] have been recently done to develop a language mechanism with a static type system that allows "right" extension of mutually recursive classes without resorting to typecasting or other unsafe features.

Family Polymorphism. Erik Ernst [11] has recently coined the term "family polymorphism" for a particular programming style using virtual classes [15] of `gbeta` [10] and applied it to solve the above-mentioned problem of mutually recursive classes.

In his proposal, mutually recursive classes are programmed as nested class members of another (top-level) class. Those member classes are virtual in the same sense as virtual methods—a reference to a class member is resolved at run-time. Thus, the meaning of mutual references to class names will change when a subclass of the enclosing class is derived and those member classes are inherited. This late-binding of class names makes it possible to reuse implementation without the mismatch described above. The term family refers to such a set of mutually recursive classes grouped inside another class. He has also shown how a method that can uniformly work for different families can be written in a safe way: such "family-polymorphic" methods take as arguments not only instances of mutually recursive classes but also the family that they belong to, so that semantical analysis (or a static type checker) can check if those instances really belong to the same family.

Although family polymorphism seems very powerful, we feel that there may be a simpler solution to the present problem. In particular, in `gbeta`, nested classes really are members (or, more precisely, attributes) of an *object*, so types for mutually recursive classes include as part object references, which serve as identifiers of families. As a result, the semantical analysis of `gbeta` essentially involves a dependent type system [1, 18], which is rather complex (especially in the presence of side effects).

Contributions of the Paper. We identify a minimal, *lightweight* set of language features to solve the problem of typing mutually recursive classes, rather than introduce a new advanced mechanism. As done in elsewhere [14], we adopt what we call the "classes-as-families" principle, in which families are identified with classes, which are static entities, rather than objects, which are dynamic. Although it loses some expressibility, a similar style of programming is still possible. Moreover, we take the approach that inheritance is not subtyping, for type safety reasons, and also avoid exact types [6], which are often deemed important in this context. These decisions simplify the type system a lot, making much easier a type soundness argument and application to practical languages such as Java. As a byproduct, we can view family polymorphic methods as a kind of parametric methods found e.g. in Java generics and find that the technique of type argument synthesis as in GJ and Java 5.0 [2, 17] can be extended to our proposal as well.

Other technical contributions of the present paper can be summarized as follows:

- Simplification of the type system for family polymorphism with the support for family-polymorphic methods;
- A rigorous discussion of the safety issues by developing a formal model called .FJ (read "dot FJ") of lightweight polymorphism, on top of Featherweight

Java [13] by Igarashi, Pierce, and Wadler, and a correctness theorem of the type system; and
– An algorithm of type argument synthesis for family-polymorphic methods and its correctness theorem.

The Rest of This Paper. After Section 2 presents the overview of our language constructs through the standard example of graphs, in Section 3, we formalize those mechanisms as the calculus .FJ and discuss its type safety. Then, Section 4 presents a type inference algorithm for family-polymorphic method invocations and discuss its correctness. Section 5 discusses related work, and Section 6 concludes. For brevity, we omit proofs of theorems; they will appear in a full version, which will be available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`.

2 Programming Lightweight Family Polymorphism

We start by informally describing the main aspects of the language constructs we study in this paper, used to support lightweight family polymorphism. To this end, we consider as a reference the example in [11], properly adapted to fit our "classes-as-families" principle.

This example features a *family* (or group) `Graph`, containing the classes `Node` and `Edge`, which are the *members* of the family, and are used as components to build graph instances. As typically happens, members of the same family can mutually refer to each other: in our example for instance, each node holds a reference to connected edges, while each edge holds a reference to its source and destination nodes. Now suppose we are interested in defining a new family `ColorWeightGraph`, used to define graphs with colored nodes and weighted edges—nodes and edges with the new properties called `color` and `weight`, respectively—such that the weight of an edge depends on the color of its source and destination nodes. Note that in this way the the members of the family `Graph` are not compatible with those of family `ColorWeightGraph`. To achieve reuse, we would like to define the family `ColorWeightGraph` as an extension of the family `Graph`, and declare a member `Node` which automatically inherits all the properties (fields and methods) of `Node` in `Graph`, and similarly for member `Edge`. Moreover, as advocated by the family polymorphism idea, we would like classes `Node` and `Edge` in `ColorWeightGraph` to mutually refer to each other automatically, as opposed to those solutions exploiting single class-inheritance where e.g. class `Node` of `ColorWeightGraph` would simply refer to `Edge` of `Graph`—thus requiring an extensive use of type-casts.

2.1 Nested Classes, Relative Path Types, and Extension of Families

This graph example can be programmed using our lightweight family polymorphism solution as reported at the top of Figure 1, whose code adheres to a

```
class Graph {
    static class Node {
        .Edge[] es=new .Edge[10]; int i=0;
        void add(.Edge e) { es[i++] = e; }}
    static class Edge {
        .Node src, dst;
        void connect(.Node s, .Node d) {
            src = s; dst = d; s.add(this); d.add(this); }}}
```

```
class ColorWeightGraph extends Graph {
    static class Node {
        Color color; }
    static class Edge {
        int weight;
        void connect(.Node s, .Node d) {
            weight = f(s.color, d.color); super.connect(s, d); }}}
```

```
Graph.Edge e; Graph.Node n1, n2;
ColorWeightGraph.Edge we; ColorWeightGraph.Node cn1, cn2;
e.connect(n1, n2);    // 1: OK
we.connect(cn1, cn2); // 2: OK
we.connect(n1, cn2);  // 3: compile-time error
e.connect(n1, cn2);   // 4: compile-time error
```

```
<G extends Graph>
  void connectAll(G.Edge[] es, G.Node n1, G.Node n2){
    for (int i: es) es[i].connect(n1,n2); }

Graph.Edge[] ges;            Graph.Node gn1,gn2;
ColorWeightGraph.Edge[] ces; ColorWeightGraph.Node cn1,cn2;
connectAll(ges, gn1, gn2);   // G as Graph
connectAll(ces, cn1, cn2);   // G as ColorWeightGraph
connectAll(ces, gn1, gn2);   // compile-time error
```

Fig. 1. `Graph` and `ColorWeightGraph` Classes

Java-like syntax—which is also the basis for the syntax of the calculus .FJ we introduce in Section 3.

The first idea is to represent families as (top-level) classes, and their members as nested classes. Note that in particular we relied on the syntax of Java `static` member classes, which provide a grouping mechanism suitable to define a family—in spite of this similarity, we shall in the following give a different semantics to that member classes, in order to support family polymorphism. The types of nodes and edges of class(-family) `Graph` are denoted by notations `Graph.Node` and `Graph.Edge` which we call *fully qualified types*. Whereas such types are useful outside the family to declare variables and to create instances of such member classes, we do not use them to specify mutual references of family members. The notations `.Node` and `.Graph` are instead introduced to this purpose, meaning "member `Node` in the current family" and "member `Edge` in the current family". We call such types *relative path types*: this terminology is justified by noting that while the notation for a type $\mathtt{C}_1.\mathtt{C}_2$ resembles an absolute directory path $/\mathtt{d}_1/\mathtt{d}_2$, notation $.\mathtt{C}_2$ resembles the relative directory path $../\mathtt{d}_2$.

The importance of relative path types becomes clear when introducing the concept of family extension. To define the new family `ColorWeightGraph`, a new class `ColorWeightGraph` is declared to extend `Graph` and providing the member classes `Node` and `Edge`. Such new members, identified outside their family by the fully qualified types `ColorWeightGraph.Node` and `ColorWeightGraph.Edge`, will inherit all the properties of classes `Graph.Node` and `Graph.Edge`, respectively. In particular, `ColorWeightGraph.Edge` will inherit method `connect()` from `Graph.Edge`, and can therefore override it as shown in the reference code, and even redirect calls by the `super.connect()` invocation. However, `connect` is declared to accept two arguments of type `.Node`, and for the particular semantics we give to relative path types, it will accept a `Graph.Node` when invoked through a `Graph.Edge`, and a `ColorWeightGraph.Node` when invoked through a `ColorWeightGraph.Edge`. Relative path types are necessary to realize family polymorphism, as they guarantee members of the extended family to mutually refer to each other, and not to refer to a different family.

2.2 Inheritance is not Subtyping for Member Classes

This covariance schema for relative path types—they change as we move from a family to a subfamily—resembles and extends the `ThisType` construct [5], used to make classes self-referencing themselves covariantly through inheritance hierarchies. As well known, however, such a covariance schema prevents inheritance and substitutability from correctly working together as happens instead in single class-inheritance of most common object-oriented languages. In particular, when relative path types are used as argument type to a method in a family member, as in method `connect()` of class `Edge`, they prevent its instances from being substituted for those in the superfamily, even though the proper inheritance relation is supported. The following code fragment reveals this problem:

```
// If ColoredWeightGraph.Edge were substitutable for Graph.Edge
Graph.Edge e=new ColoredWeightGraph.Edge();
Graph.Node n1=new Graph.Node();
Graph.Node n2=new Graph.Node();
e.connect(n1,n2);  // Unsafe!!
```

If class `ColorWeightGraph.Edge` could be substituted for `Graph.Edge`, then it could happen to invoke `connect()` on a `ColorWeightGraph.Edge` passing some `Graph.Node` as elements. Such an invocation would lead to the attempt of accessing field `color` on an object of class `Graph.Node`, which does *not* have a definition for it!

To prevent this form of unsoundness, our lightweight family polymorphism solution proposes to prevent such substitutability by adopting an "inheritance without subtyping" approach for family members. Applied to our graph example, it means that while `ColorWeightGraph.Node` inherits all the properties of `Graph.Node` (for `ColorWeightGraph` extends `Graph`), we do not have that `ColorWeightGraph.Node` is a subtype of `Graph.Node`. As a result of this choice, we are able to correctly check for the invocation of methods in members. In the client code of Figure 1 (third box), the first two invocations are correct for node

arguments belong to the same family of the receiving edge, the third and fourth are (statically) incorrect, as we are passing as argument a node belonging to a different family than the receiving edge, thus incompatible with the expected node—as `Graph.Node` and `ColorWeightGraph.Node` are not in the subtype relation.

2.3 Family-Polymorphic Methods as Parametric Methods

To fully exploit the benefits of family polymorphism it should be possible to write so-called *family-polymorphic methods*, that is, methods that can work over different families, and where a single invocation is specific to the family of the elements passed as argument. As an example, it should be possible to write a method `connectAll` taking as input an array of edges and two nodes, connecting each edge to the two nodes, and ensuring the edges and nodes of the same family are used. In our language this is realized through parametric methods as shown in the bottom of Figure 1. Method `connectAll` is defined as parametric in a type `G` with upper-bound `Graph`: `G` represents the family used for a specific invocation, and correspondingly the arguments are of type `G.Edge[]`, `G.Node` and `G.Node` respectively. As a result, in the first invocation of the example code, by passing edges and nodes of family `Graph` the compiler would infer for `G` the type `Graph`, and similarly in the second invocation infers `ColorGraphWeight`. Finally, in the third invocation no type can be inferred for `G`, since for no `G` we have that `G.Edge` and `G.Node` match types `ColorWeightGraph.Edge` and `Graph.Node`.

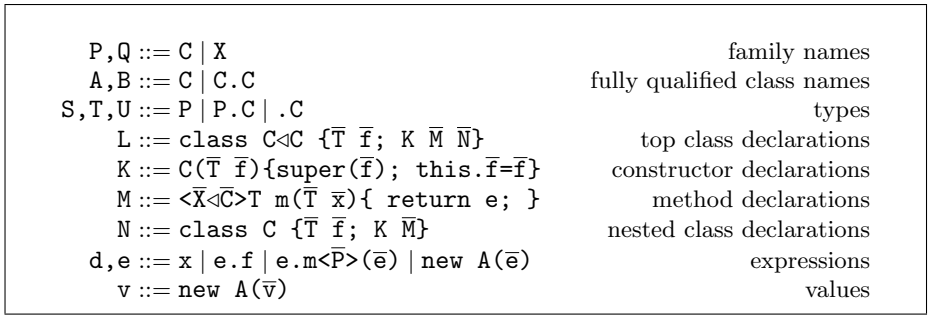
3 .FJ: A Formal Model of Lightweight Family Polymorphism

In this section, we formalize the ideas described in the previous section, namely, nested classes with simultaneous extension, relative path types and family-polymorphic methods as a small calculus named .FJ based on Featherweight Java [13], a functional core of class-based object-oriented languages. After formally defining the syntax (Section 3.1), type system (Sections 3.2 and 3.3), and operational semantics (Section 3.4) of .FJ, we show a type soundness result (Section 3.5).

For simplicity, we deal with a single level of nesting, as opposed to Java, which allows arbitrary levels of nesting. Although they are easy to add, typecasts—which appear in Featherweight Java—are dropped since one of our aims here is to show programming in the previous section is possible without typecasts. In .FJ, every parametric method invocation has to provide its type arguments—type inference will be discussed in Section 4.

3.1 Syntax

The abstract syntax of top-level/nested class declarations, constructor declarations, method declarations, and expressions of the extended system is given in



P,Q	::= C | X	family names
A,B	::= C | C.C	fully qualified class names
S,T,U	::= P | P.C | .C	types
L	::= class C◁C {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$ $\overline{\texttt{N}}$}	top class declarations
K	::= C($\overline{\texttt{T}}$ $\overline{\texttt{f}}$){super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$}	constructor declarations
M	::= <$\overline{\texttt{X}}$◁$\overline{\texttt{C}}$>T m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){ return e; }	method declarations
N	::= class C {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$}	nested class declarations
d,e	::= x | e.f | e.m<$\overline{\texttt{P}}$>($\overline{\texttt{e}}$) | new A($\overline{\texttt{e}}$)	expressions
v	::= new A($\overline{\texttt{v}}$)	values

Fig. 2. .FJ: Syntax

Figure 2. Here, the metavariables C, D, and E range over (simple) class names; X and Y range over type variable names; f and g range over field names; m ranges over method names; x ranges over variables.

We put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\texttt{C}}$ $\overline{\texttt{f}}$" for "$\texttt{C}_1$ $\texttt{f}_1$,...,$\texttt{C}_n$ $\texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and "this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;" as shorthand for "this.$\texttt{f}_1$=$\texttt{f}_1$;...;this.$\texttt{f}_n$=$\texttt{f}_n$;" and so on. Sequences of type variables, field declarations, parameter names, and method declarations are assumed to contain no duplicate names. We write the empty sequence as • and denote concatenation of sequences using a comma.

A family name, used as a type argument to family-polymorphic methods, is either a top-level class name or a type variable. Fully qualified class names can be used to instantiate objects, so they play the role of run-time types of objects. A type can be a family name, a fully qualified class name, X.C, or a relative path type .C. A top-level class declaration consists of its name, its superclass, field declarations, a constructor, methods, and nested classes. The symbol ◁ is read extends. On the other hand, nested classes does not have an extends clause since the class from which it inherits is implicitly determined. We have dropped the keyword static, used in the previous section, for conciseness. As in Featherweight Java, a constructor is given in a stylized syntax and just takes initial (and final) values for the fields and assigns them to corresponding fields. A method declaration can be parameterized by type variables, whose bounds are top-level class (i.e., family) names. Since the language is functional, the body of a method is a single return statement. An expression is either a variable, field access, method invocation, or object creation. We assume that the set of variables includes the special variable this, which cannot be used as the name of a parameter to a method.

A class table CT is a mapping from fully qualified class names A to (top-level or nested) class declarations. A program is a pair $(CT, \texttt{e})$ of a class table and an expression. To lighten the notation in what follows, we always assume a

Field Lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;} \ldots\} \quad \mathit{fields}(\texttt{D}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

$$\mathit{fields}(\texttt{Object.C}) = \bullet$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{N}}\} \quad \texttt{class E } \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;} \ldots\} \in \overline{\texttt{N}} \quad \mathit{fields}(\texttt{D.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{N}}\} \quad \texttt{E} \notin \overline{\texttt{N}} \quad \mathit{fields}(\texttt{D.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C.E}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}$$

Method Type Lookup:

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{M}}\} \quad \texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{C}}\texttt{>T}_0\ \texttt{m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0}$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{M}} \ldots\} \quad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{N}}\} \quad \texttt{class E } \{\ldots \overline{\texttt{M}}\} \in \overline{\texttt{N}} \quad \texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{C}}\texttt{>T}_0\ \texttt{m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C.E}) = \texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0}$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{N}}\} \quad \texttt{class E } \{\ldots \overline{\texttt{M}}\} \in \overline{\texttt{N}} \quad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C.E}) = \mathit{mtype}(\texttt{m}, \texttt{D.E})}$$

$$\frac{\texttt{class C} \lhd \texttt{D } \{\ldots \overline{\texttt{N}}\} \quad \texttt{E} \notin \overline{\texttt{N}}}{\mathit{mtype}(\texttt{m}, \texttt{C.E}) = \mathit{mtype}(\texttt{m}, \texttt{D.E})}$$

Fig. 3. .FJ: Auxiliary lookup functions

fixed class table CT. As in Featherweight Java, we assume that `Object` has no members and its definition does *not* appear in the class table.

3.2 Lookup Functions

Before proceeding to the type system, we give functions to lookup field or method definitions. The function $\mathit{fields}(\texttt{A})$ returns a sequence $\overline{\texttt{T}}\ \overline{\texttt{f}}$ of field names of the class `A` with their types. The function $\mathit{mtype}(\texttt{m}, \texttt{A})$ takes a method name and a class name as input and returns the corresponding method signature of the form $\texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{C}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0$, in which $\overline{\texttt{X}}$ are bound. They are defined by the rules in Figure 3. Here, $\texttt{m} \notin \overline{\texttt{M}}$ (and $\texttt{E} \notin \overline{\texttt{N}}$) mean the method of name `m` (and the nested class of name `E`, respectively) do not exist in $\overline{\texttt{M}}$ (and $\overline{\texttt{N}}$, respectively).

As mentioned before, `Object` do not have any fields, methods, or nested classes, so $\mathit{fields}(\texttt{Object}) = \mathit{fields}(\texttt{Object.C}) = \bullet$ for any `C`, and $\mathit{mtype}(\texttt{m}, \texttt{Object})$ and $\mathit{mtype}(\texttt{m}, \texttt{Object.C})$ are undefined. The definitions are straightforward extensions of the ones in Featherweight Java. Interesting rules are the last rules: when a nested class `C.E` does not exist, it looks up the nested class of the same name `E` in the superclass of the enclosing class `C`.

Subtyping:

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$$

$$\Delta \vdash \mathtt{T} <: \mathtt{Object}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{...\}}}{\Delta \vdash \mathtt{C} <: \mathtt{D}}$$

Type Well-formedness:

$$\Delta \vdash \mathtt{Object}\ \text{ok in}\ \mathtt{A}$$

$$\frac{\mathtt{A} \in dom(CT)}{\Delta \vdash \mathtt{A}\ \text{ok in}\ \mathtt{B}}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\{...\overline{\mathtt{N}}\} \qquad \mathtt{E} \notin \overline{\mathtt{N}} \qquad \Delta \vdash \mathtt{D.E}\ \text{ok in}\ \mathtt{A}}{\Delta \vdash \mathtt{C.E}\ \text{ok in}\ \mathtt{A}}$$

$$\frac{\Delta \vdash \Delta(\mathtt{X})\ \text{ok in}\ \mathtt{A}}{\Delta \vdash \mathtt{X}\ \text{ok in}\ \mathtt{A}}$$

$$\frac{\Delta \vdash \Delta(\mathtt{X})\mathtt{.C}\ \text{ok in}\ \mathtt{A}}{\Delta \vdash \mathtt{X.C}\ \text{ok in}\ \mathtt{A}}$$

$$\frac{\Delta \vdash \mathtt{C.E}\ \text{ok in}\ \mathtt{C.D}}{\Delta \vdash \mathtt{.E}\ \text{ok in}\ \mathtt{C.D}}$$

Fig. 4. .FJ: Subtyping and type well-formedness

3.3 Type System

The main judgments of the type system consist of one for subtyping $\Delta \vdash \mathtt{S} <: \mathtt{T}$, one for type well-formedness $\Delta \vdash \mathtt{T}$ ok in $\mathtt{A}$, and one for typing $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ in $\mathtt{A}$. Here, Γ is a *type environment*, which is a finite mapping from variables to types, written $\overline{\mathtt{x}}{:}\overline{\mathtt{T}}$; Δ is a *bound environment*, which is a finite mapping from type variables to their bounds, written $\overline{\mathtt{X}}{<:}\overline{\mathtt{C}}$. Since we are not concerned with more general forms of bounded polymorphism, upper bounds are always top-level class names. We abbreviate a sequence of judgments in the obvious way: $\Delta \vdash \mathtt{S}_1 <: \mathtt{T}_1$, $\ldots$, $\Delta \vdash \mathtt{S}_n <: \mathtt{T}_n$ to $\Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$; $\Delta \vdash \mathtt{T}_1$ ok in $\mathtt{A}$, $\ldots$, $\Delta \vdash \mathtt{T}_n$ ok in $\mathtt{A}$ to $\Delta \vdash \overline{\mathtt{T}}$ ok in $\mathtt{A}$; and $\Delta; \Gamma \vdash \mathtt{e}_1{:}\mathtt{T}_1$ in $\mathtt{A}$, $\ldots$, $\Delta; \Gamma \vdash \mathtt{e}_n{:}\mathtt{T}_n$ in $\mathtt{A}$ to $\Delta; \Gamma \vdash \overline{\mathtt{e}}{:}\overline{\mathtt{T}}$ in $\mathtt{A}$.

Subtyping. The subtyping judgment $\Delta \vdash \mathtt{S} <: \mathtt{T}$, read as "$\mathtt{S}$ is subtype of $\mathtt{T}$ under Δ," is defined in Figure 4. This relation is the reflexive and transitive closure of the `extends` relation with `Object` being the top type. Note that a nested class, which does not have the `extends` clause, has only a trivial super/subtype, which is itself, even if some members are inherited from another (nested) class.

Type Well-formedness. The type well-formedness judgment $\Delta \vdash \mathtt{T}$ ok in $\mathtt{A}$, read as "$\mathtt{T}$ is a well formed type in (the body of) class $\mathtt{A}$ under Δ." The rules for well-formed types appear also in Figure 4. `Object` and class names in the domain of the class table are well formed. Moreover, a nested class `C.E` is well formed if `E` is inherited from `C`'s superclass `D`. Type `X` (possibly with a suffix) is well formed if its upper bound (with the suffix) is well formed. Finally, a relative path type `.E` is well formed in a nested class `C.D` if `C.E` is well formed.

Typing. We first introduce a few more auxiliary definitions, required for expression typing. We write $\mathit{bound}_\Delta(\mathtt{T})$ for the upper bound of $\mathtt{T}$ in Δ, defined by:

$$\mathit{bound}_\Delta(\mathtt{A}) = \mathtt{A} \qquad \mathit{bound}_\Delta(\mathtt{X}) = \Delta(\mathtt{X}) \qquad \mathit{bound}_\Delta(\mathtt{X.C}) = \Delta(\mathtt{X})\mathtt{.C}.$$

This notation is used to replace a type variable with its upper bound. We never ask the upper bound of a relative path type, so $\mathit{bound}_\Delta(\mathtt{.C})$ is undefined. The *resolution* $\mathtt{T@S}$ of $\mathtt{T}$ in $\mathtt{S}$, which intuitively denotes the class name that $\mathtt{T}$ refers to in a given class $\mathtt{S}$, is defined by:

$$\mathtt{.D@P.C} = \mathtt{P.D} \qquad \mathtt{.D@.C} = \mathtt{.D} \qquad \mathtt{P@T} = \mathtt{P} \qquad \mathtt{P.C@T} = \mathtt{P.C}.$$

The only interesting case is the first clause: It means that a relative path type `.D` in `P.C` refers to `P.D`—it resembles the command `cd` of UNIX shells: `cd ../D` changes the current directory from `P/C` to `P/D`. For example, $\mathtt{.Edge@Graph.Node} = \mathtt{Graph.Edge}$. Note that `.D@C` is undefined since a relative path type is not allowed to appear in top-level classes.

The typing judgment for expressions is of the form $\Delta; \Gamma \vdash \mathtt{e:T}$ in $\mathtt{A}$, read as "under bound environment Δ and type environment Γ, expression $\mathtt{e}$ has type $\mathtt{T}$ in class $\mathtt{A}$." Typing rules are given in Figure 5. Interesting rules are T-Field and T-Invk, although the basic idea is as usual—for example, in T-Field, the field types are retrieved from the receiver's type $\mathtt{T}_0$, and the corresponding type of the accessed field is the type of the whole expression. We need some tricks to deal with relative path types (and type variables): if the receiver's type $\mathtt{T}_0$ is a relative path type, it has to be resolved in $\mathtt{A}$, the class in which $\mathtt{e}$ appears; a type variable is taken to its upper bound by $\mathit{bound}_\Delta()$. Moreover, if the field type is a relative path type, it is resolved in the *receiver's type*. For example, if $\mathit{fields}(\mathtt{CWGraph.Node}) = \mathtt{.Edge\ edg}$ and $\Gamma = \mathtt{x:CWGraph.Node,\ y:.Node}$, then

$$\Delta; \Gamma \vdash \mathtt{x.edg\ :\ CWGraph.Edge} \text{ in } \mathtt{CWGraph.Node} \text{ and}$$
$$\Delta; \Gamma \vdash \mathtt{y.edg\ :\ .Edge} \text{ in } \mathtt{CWGraph.Node}.$$

In this way, accessing a field of relative path type gives a relative path type only when the receiver is also given a relative path type. Similarly, in T-Invk, the method type is retrieved from the receiver's type; then, it is checked whether the given type arguments are subtypes of bounds $\overline{\mathtt{C}}$ of formal type parameters and the types of actual value arguments are subtypes of those of formal parameters, where type arguments are substituted for variables. For example, if $\mathit{mtype}(\mathtt{connectAll}, \mathtt{C}) = \mathtt{<G\triangleleft Graph>(G.Node,G.Edge)}\rightarrow\mathtt{void}$, then

$$\Delta; \mathtt{x:C}, \mathtt{n:CWGraph.Node}, \mathtt{e:CWGraph.Edge} \vdash$$
$$\mathtt{x.connectAll<CWGraph>(n,e)} : \mathtt{void} \text{ in } \mathtt{A}.$$

Judgments for method typing are written $\mathtt{M}$ ok in $\mathtt{A}$, and derived by T-Method. Here, $\mathit{thistype}(\mathtt{A})$ and $\mathit{superclass}(\mathtt{A})$ are defined by:

$$\mathit{thistype}(\mathtt{C}) = \mathtt{C} \qquad \mathit{superclass}(\mathtt{C}) = \mathtt{D}$$
$$\mathit{thistype}(\mathtt{C.E}) = \mathtt{.E} \qquad \mathit{superclass}(\mathtt{C.E}) = \mathtt{D.E}$$

Expression Typing:

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0:\mathtt{T}_0 \text{ in } \mathtt{A} \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0@\mathtt{A})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i:\mathtt{T}_i@\mathtt{T}_0 \text{ in } \mathtt{A}} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \text{ in } \mathtt{A} \qquad \mathit{mtype}(\mathtt{m},\, \mathit{bound}_\Delta(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{X}} \mathtt{\triangleleft}\, \overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U} \\ \Delta \vdash \overline{\mathtt{P}}\ \mathtt{<:}\ \overline{\mathtt{C}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}}:\overline{\mathtt{T}} \text{ in } \mathtt{A} \qquad \Delta \vdash \overline{\mathtt{T}}\ \mathtt{<:} ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\mathtt{U})@\mathtt{T}_0 \text{ in } \mathtt{A}} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \text{ in } \mathtt{A} \qquad \Delta \vdash \overline{\mathtt{U}}\ \mathtt{<:}\overline{\mathtt{T}}@\mathtt{A}_0}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{A}_0\mathtt{(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{A}_0 \text{ in } \mathtt{A}} \qquad \text{(T-New)}$$

Method and Class Typing:

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\ \mathtt{<:}\ \overline{\mathtt{C}} \qquad \Delta;\overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{this}:\mathit{thistype}(\mathtt{A}) \vdash \mathtt{e}_0:\mathtt{U}_0 \text{ in } \mathtt{A} \\ \Delta \vdash \mathtt{U}_0\ \mathtt{<:}\ \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0\mathtt{,}\overline{\mathtt{T}}\mathtt{,}\overline{\mathtt{C}} \text{ ok in } \mathtt{A} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathit{superclass}(\mathtt{A})) = \mathtt{<}\overline{\mathtt{Y}} \mathtt{\triangleleft}\, \overline{\mathtt{D}}\mathtt{>}\overline{\mathtt{S}}\mathtt{\to}\mathtt{S}_0 \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \overline{\mathtt{T}}\mathtt{,}\mathtt{T}_0 = [\overline{\mathtt{X}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\mathtt{,}\mathtt{S}_0)\end{array}}{\mathtt{<}\overline{\mathtt{X}} \mathtt{\triangleleft}\, \overline{\mathtt{C}}\mathtt{>}\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{ return\ e}_0\mathtt{;\ \}} \text{ ok in } \mathtt{A}} \qquad \text{(T-Method)}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{E(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{\}} \\ \mathit{fields}(\mathit{superclass}(\mathtt{C})\mathtt{.E}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}} \text{ ok in } \mathtt{C.E} \qquad \emptyset \vdash \overline{\mathtt{T}} \text{ ok in } \mathtt{C.E}\end{array}}{\mathtt{class\ E\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \text{ ok in } \mathtt{C}} \qquad \text{(T-NClass)}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}} \text{ ok in } \mathtt{C} \qquad \overline{\mathtt{N}} \text{ ok in } \mathtt{C} \qquad \emptyset \vdash \overline{\mathtt{T}}\mathtt{,D} \text{ ok in } \mathtt{C}\end{array}}{\mathtt{class\ C} \mathtt{\triangleleft}\, \mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\ \overline{\mathtt{N}}\mathtt{\}} \text{ ok}} \qquad \text{(T-TClass)}$$

Fig. 5. .FJ: Typing

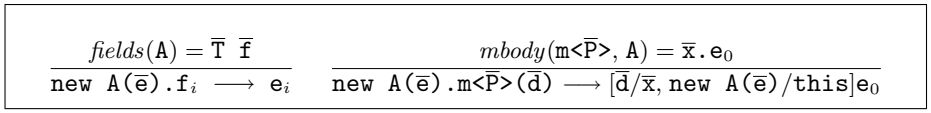


Fig. 6. .FJ: Reduction

where `class C◁D{...}`. It checks that the body of the method is well typed under the bound and type environments obtained from the definition. Note that `this` of a nested class is given a relative path type, as the meaning of `this` changes in subclasses. The last conditional premise checks that `m` correctly overrides (if it does) the method of the same name in the superclass with the same signature (modulo renaming of type variables). Note that a relative path type is, literally, not changed by overriding but its meaning is changing.

There are two class typing rules, one for top-level classes and one for nested classes. Both of them are essentially the same: they check that field types and constructor argument types are the same, and that methods are ok in the class. The rule T-TC<sc>lass</sc> for top-level classes also checks that nested classes are ok.

3.4 Operational Semantics

The operational semantics is given by the reduction relation of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to `e′` in one step." We require another lookup function *mbody*(`m`, `A`), of which we omitted its obvious definition, for the method body with formal parameters, written $\overline{\mathtt{x}}.\mathtt{e}$, of given method and class names.

The reduction rules are given in Figure 6. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ 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`. There are two reduction rules, one for field access and one for method invocation, which are straightforward. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), omitted here. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

3.5 Type Soundness

The type system is sound with respect to the operational semantics, as expected. Type soundness is proved in the standard manner via subject reduction and progress [21, 13].

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

Lemma 2 (Progress). *If* $\emptyset; \emptyset \vdash \mathtt{e:A}$ in `B` *and* `e` *is not a value, then* $\mathtt{e} \longrightarrow \mathtt{e}'$, *for some* `e′`.

Theorem 1 (Type Soundness). *If* $\emptyset; \emptyset \vdash \mathtt{e:A}$ in `B` *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *with* `e′` *a normal form, then* `e′` *is a value* `v` *with* $\emptyset; \emptyset \vdash \mathtt{v:A}'$ in `B` *and* $\emptyset \vdash \mathtt{A}'\mathtt{<:A}$.

$$
\begin{array}{lcl}
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(\emptyset) & = & [\,] \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{P.C} <: \mathtt{X}_i\mathtt{.C}\}) & = & [\mathtt{X}_i \mapsto \mathtt{P}] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}([\mathtt{P}/\mathtt{X}_i]S') \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1 <: \mathtt{X}_i\} \uplus \{\mathtt{T}_2 <: \mathtt{X}_i\}) & = & \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{(\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2) <: \mathtt{X}_i\}) \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T} <: \mathtt{X}_i, \mathtt{X}_i <: \mathtt{C}_i\}) & = & \begin{array}{ll} [\mathtt{X}_i \mapsto \mathtt{T}] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & \text{if } \Delta \vdash \mathtt{T} <: \mathtt{C}_i \text{ and } \mathtt{X}_i \notin S' \\ \mathit{fail} & \text{otherwise} \end{array} \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{X}_i <: \mathtt{C}_i\}) & = & \begin{array}{ll} [\mathtt{X}_i \mapsto \mathtt{C}_i] \circ \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & \text{if } \mathtt{X}_i \notin S' \\ \mathit{fail} & \text{otherwise} \end{array} \\
\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S' \uplus \{\mathtt{T}_1 <: \mathtt{T}_2\}) & = & \begin{array}{ll} \mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S') & \text{if } \Delta \vdash \mathtt{T}_1 <: \mathtt{T}_2 \\ \mathit{fail} & \text{otherwise} \end{array}
\end{array}
$$

Fig. 7. Algorithm for Type Argument Synthesis

4 Type Inference for Parametric Method Invocations

The language .FJ in the previous section is considered an intermediate language in which every type argument to parametric methods is made explicit. In this section, we briefly discuss how type arguments can be recovered, give an algorithm for type argument inference, and show its correctness theorem.

The basic idea of type inference is the same as Java 5.0[1]: Given a method invocation expression $\mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}})$ that appears in class `A` without specifying type arguments, we can at least compute the type $\mathtt{T}_0$ of $\mathtt{e}_0$, the signature $\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ of the method `m`, and the types $\overline{\mathtt{T}}$ of (value) arguments. Then, it is easy to see from the rule T-Invk that it suffices to find $\overline{\mathtt{P}}$ that satisfies $\overline{\mathtt{P}} <: \overline{\mathtt{C}}$ and $\overline{\mathtt{T}} <: ([\overline{\mathtt{P}}/\overline{\mathtt{X}}]\overline{\mathtt{U}})@\mathtt{T}_0$. In other words, the goal of type inference is to solve the set $\{\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}\mathtt{,}\overline{\mathtt{T}}\mathtt{<:}(\overline{\mathtt{U}}@\mathtt{T}_0)\}$ of inequalities with respect to $\overline{\mathtt{X}}$.

We formalize this constraint solving process as function $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S)$. It takes as input a set of inequalities of the form either $\mathtt{X}\mathtt{<:}\mathtt{C}$ or $\mathtt{T}_1\mathtt{<:}\mathtt{T}_2$ where $\mathtt{T}_1$ does not contain $\mathtt{X}_i$, and returns a mapping from $\overline{\mathtt{X}}$ to types. Δ records other variables' bounds, so $\overline{\mathtt{X}}$ and the domain of Δ are assumed to be disjoint. The definition of $\mathit{Infer}^{\Delta}_{\overline{\mathtt{X}}}(S)$ is shown in Figure 7. Here, $\mathtt{T}_1 \sqcup_{\Delta} \mathtt{T}_2$ is the least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$ (the least upper bound of given two types always exist since we do not have interfaces, which can extend more than one interface.) We assume that each clause is applied in the order shown—thus, for example, the fourth clause will not be applied until there is only one inequation of the form $\mathtt{T}\mathtt{<:}\mathtt{X}_i$.

The algorithm is explained as follows. The second clause is the case where a formal argument type is $\mathtt{X}_i\mathtt{.C}$ and the corresponding actual is `P.C`: since `P.C` has only a trivial supertype (namely, itself), $\mathtt{X}_i$ must be `P`. The third clause is the case where a type variable has more than one lower bound: we replace two inequalities by one using the least upper bound. The following two clauses are

[1] We should note that the flaw, found by Alan Jeffrey, in the original GJ type inference does not apply since our setting is much simpler.

applied when no other constraints on $\mathtt{X}_i$ appear elsewhere; it checks whether the constraint is satisfiable.

Now, we state the theorem of correctness of type inference. It means that, if type inference succeeds, it gives the least type arguments.

Theorem 2 (Type Inference Correctness). *If* $\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0$ in $\mathtt{A}$ *and* $mtype(\mathtt{m}, bound_\Delta(\mathtt{T}_0)@\mathtt{A}) = \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0$ *and* $\Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ in $\mathtt{A}$ *and* $Infer^{\Delta}_{\overline{\mathtt{X}}}(\{\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}, \overline{\mathtt{T}}\mathtt{<:}(\overline{\mathtt{U}}@\mathtt{T}_0)\})$ *returns* $\sigma = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{P}}]$, *then* $\Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.m<}\sigma\overline{\mathtt{X}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} : (\sigma\mathtt{U}_0)@\mathtt{T}_0$ in $\mathtt{A}$. *Moreover,* σ *is the least solution if every* $\mathtt{X}_i$ *occurs in* $\overline{\mathtt{U}}$ *in the sense that for any* σ' *such that* $\Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.m<}\sigma'\overline{\mathtt{X}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} : (\sigma'\mathtt{U}_0)@\mathtt{T}_0$ in $\mathtt{A}$, *it holds that* $\Delta \vdash \sigma(\mathtt{X}_i) \mathtt{<:} \sigma'(\mathtt{X}_i)$ *for any* $\mathtt{X}_i$.

5 Related Work

As we have already mentioned, in the original formulation of family polymorphism [11] nested classes are members (or attributes) of an object of their enclosing class. Thus, to create node or edge objects, one first has to instantiate `Graph` and then to invoke `new` on that object. It would be written as

```
Graph g = new Graph();
g.Node n = new g.Node(...);  g.Edge e = new g.Edge(...);
```

Notice that the types of nodes and edges would include a reference `g` to the `Graph` object. Relative path types `.Node` and `.Edge` would respectively become `this.Node` and `this.Edge`, where the meaning of types changes as the meaning of `this` changes due to usual late-binding. Finally, `connectAll` would take four *value* arguments instead of one type and three value arguments as:

```
void connectAll(Graph g,
                g.Edge[] es, g.Node n1, g.Node n2){ ... }
```

Notice that the first argument appears as part of types of the following arguments; it is required for a type system to guarantee that `es`, `n1`, and `n2` belong to the same graph. As a result, a type system is equipped with dependent types, such as `g.Edge`, which can be quite tricky (especially in the presence of side-effects). We deliberately avoid such types by identifying families with classes. As a byproduct, as shown in the previous section, we have discovered that GJ-style type inference is easy to extend to this setting. Although complex, the original approach has one apparent advantage: one can instantiate arbitrary number of `Graph` objects and distinguish nodes and edges of different graphs by static types. Scala [18] and JX [16] also support family polymorphism, based on dependent types.

Historically, the mismatching problem of recursive class definitions has been studied in the context of binary methods [4], which take an object of the same class as the receiver, hence the interface is recursive. In particular, Bruce's series of work [7, 5] introduced the notion of *MyType* (or sometimes called *ThisType*), which is the type of `this` and changes its meaning along the inheritance chain, just as our relative path types. Later, he extended the idea to mutually recursive type/class definitions [6, 8, 3] by introducing constructs to group mutually

recursive definitions, and the notion of *MyGroup*, which is a straightforward extension of *MyType* to the mutually recursive setting. Jolly et al. [14] has designed the language called Concord by following this approach and has applied to a Java-like language with a name-based type system. The core type system has been proven sound. Our approach is similar to them in the sense that dependent types are not used. However, in these work, family-polymorphic methods are not taken into account very seriously, although a similar idea is mentioned in Bruce et al. [6] and it can be considered a generalization of match-bound polymorphic methods in the language $\mathcal{LOOM}$ [7]. In Bruce et al. [6], inheritance is considered subtyping, so `ColorWeightGraph.Node <: Graph.Node`, for example. To ensure type safety, they introduced the notion of exact types and allow to invoke a method that take an argument of the same family only when the receiver's family is *exactly* known. We have avoided them by viewing every (nested-class) type as exact.

In Concord, `gbeta`, Scala, and JX, an inheritance relation between nested classes can be introduced. For example, `C.F` can be a subclass of `C.E` and, in a subclass `D` of `C`, the relationship is preserved while members can be added to both `E` and `F`. Although useful, we have carefully avoided this feature, too, which is not strongly required by family polymorphism, since there is a semantic complication as in languages with multiple inheritance: `D.F` may inherit conflicting members of the same name from `C.F` and `D.E`.

Finally, we should note that programming described in Section 2 could be carried out in Java 5.0 proper, which is equipped with generics [2] and F-bounded polymorphism [9], by using the technique [20] used to solve the "expression problem". It requires, however, a lot of boilerplate code for type parameterization, which makes programs less easy to grasp.

6 Concluding Remarks

We have identified a minimal set of language features to solve the problem of mismatching between mutually recursive classes and inheritance. Our proposal is lightweight in the sense that the type system, which avoids (value) dependent types, is much simpler than the original formulation of family polymorphism and easy to apply to mainstream languages such as Java and C$^{\#}$. We have shown type safety of the language mechanism by proving a type soundness theorem for the formalized core language .FJ. We have also formalized an algorithm for type argument inference for family polymorphic methods with its correctness theorem. Although .FJ is not equipped with generics, we believe it can be easily integrated into the ordinary type argument inference algorithm.

We feel that the principle of classes-as-families is worth pursuing. There have been much work based on object-based families, such as higher-order hierarchies [12], nested inheritance [16], and so on. It is interesting to investigate whether the principle can be applied to those advanced ideas.

Acknowledgements

The first author would like to thank members of the Kumiki project for fruitful discussions on this subject. This work was supported in part by Grant-in-Aid for Scientific Research on Priority Areas Research No. 13224013 from MEXT of Japan (Igarashi), and from the Italian PRIN 2004 Project "Extensible Object Systems" (Viroli).

References

[1] David Aspinall and Martin Hofmann. Dependent types. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 2, pages 45–86. The MIT Press, 2005.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 183–200, Vancouver, BC, October 1998.

[3] Kim B. Bruce. Some challenging typing issues in object-oriented languages. In *Proceedings of Workshop on Object-Oriented Development (WOOD'03)*, volume 82 of *Electronic Notes in Theoretical Computer Science*, 2003.

[4] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary method. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[5] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2004)*, volume 3086 of *Lecture Notes on Computer Science*, Oslo, Norway, June 2004. Springer Verlag.

[6] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes on Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer Verlag.

[7] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes on Computer Science*, pages 104–127, Jyväskylä, Finland, June 1997. Springer Verlag.

[8] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, volume 20 of *Electronic Notes in Theoretical Computer Science*, New Orleans, LA, April 1999. Elsevier. Available through `http://www.elsevier.nl/locate/entcs/volume20.html`.

[9] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of ACM Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 273–280, London, England, September 1989. ACM Press.

[10] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[11] Erik Ernst. Family polymorphism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2001)*, volume 2072 of *Lecture Notes on Computer Science*, pages 303–326, 2001.

[12] Erik Ernst. Higher-order hierarchies. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2003)*, volume 2743 of *Lecture Notes on Computer Science*, pages 303–328, Darmstadt, Germany, July 2003. Springer Verlag.

[13] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. A preliminary summary appeared in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146, Denver, CO, October 1999.

[14] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *Proceedings of 6th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP2004)*, June 2004.

[15] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pages 397–406, October 1989.

[16] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, Vancouver, BC, October 2004.

[17] Martin Odersky. Inferred type instantiation for GJ. Available at `http://lampwww.epfl.ch/~odersky/papers/localti02.html`, January 2002.

[18] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes on Computer Science*, pages 201–224, Darmstadt, Germany, July 2003. Springer Verlag.

[19] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings of 13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes on Computer Science*, pages 186–204, Lisbon, Portugal, June 1999. Springer Verlag.

[20] Mads Torgersen. The expression problem revisited: Four new solutions using generics. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP2004)*, volume 3086 of *Lecture Notes on Computer Science*, pages 123–146, Oslo, Norway, June 2004.

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