Variant Path Types for Scalable Extensibility

Atsushi Igarashi
Kyoto University, Japan
igarashi@kuis.kyoto-u.ac.jp

Mirko Viroli
Alma Mater Studiorum – Università di Bologna a Cesena, Italy
mirko.viroli@unibo.it

Abstract

Much recent work in the design of object-oriented programming languages has been focusing on identifying suitable features to support so-called scalable extensibility, where the usual extension mechanism by inheritance works in different scales of software components—that is, classes, groups of classes, groups of groups and so on. Mostly, this issue has been addressed by means of dependent type systems, where nested types are seen as properties of objects. In this work, we seek instead for a different and possibly simpler solution, retaining the Java-like approach of nested types as properties of classes. We introduce the mechanism of *variant path types*, which provides a flexible means to intra-group relationship (among classes) that has to be preserved through extension. Featuring the new notions of *exact* and *inexact qualifications*, these types also provide rich abstractions to express various kinds of set of objects, thanks to a flexible subtyping mechanism. We formalize a safe type system for variant path types on top of Featherweight Java. Though a full study of applicability and expressiveness is ongoing work, our development currently results in a complete solution for scalable extensibility, similarly to previous attempts based on dependent type systems.

1. Introduction

Background Much recent work in the design of object-oriented programming languages has been focusing on identifying suitable features to support extensibility not just for individual classes, but also for groups of classes, groups of groups and so on. This research direction is meant to make object-oriented languages meet the requirements of *scalable* component-based applications: since a reusable piece of code (namely, a component) can be implemented as a group of cooperating classes, it would be useful to apply the traditional mechanism of inheritance to groups of classes. Researches on family polymorphism [11], higher-order structures [12], nested inheritance [23], and grouping mechanisms [3, 19], all share this common goal, which we shall refer to as *scalable extensibility*, the term coined in the work by Nystrom et al. [23]. In particular, for an object-oriented language supporting scalable extensibility, a number of features must be provided, namely: *(i)* a mechanism for nesting classes at an arbitrary level, *(ii)* an inheritance construct seemlessy working for both single classes and group of classes, *(iii)* a flexible enough subtyping relation for nested class-types, and *(iv)* a group-polymorphism mechanism.

In spite of a few attempts such as [3, 19], languages supporting scalable extensibility are currently based on dependent type (or class) systems, like JX [23], Scala [25], or `gbeta` [12]. There, nested types are accessed through (a restricted set of) expressions: as on one hand this schema is rather expressive, it forces the programmer to take into account somewhat orthogonal aspects such as immutability of fields and variables—see Section 5 for a more detailed discussion. Though current works are devoted to identify simple core calculi for languages with dependent types—such as for Scala and `gbeta` [9, 13]—such languages are typically more complex than the standard Java setting, and more difficult to manage. It is therefore interesting to evaluate whether (and to which extent) scalable extensibility can be achieved in a language without dependent types.

In [17], we started approaching this issue by seeking a minimal set of features for supporting family polymorphism as proposed in [11] in the context of the Beta-style virtual classes [20], that is, scalable extensibility at one level of nesting.

Our Contributions In this paper, we develop this approach a step further, supporting intra-group inheritance and arbitrary levels of group hierarchies. This is achieved through a new typing construct, which we name *variant path types* [1]. Starting from [17], this construct first extends the concept of *relative types* to work in a deeply nested structure. Generalizing the notion of *MyType* and *MyGroup* in [2, 3], such types can express self reference and mutual reference among classes in a group, which have to be preserved by group extension. In addition to them, we introduce two kinds of qualifications—the notation to access a nested class `D` (as a type) inside the class of a type `T`—which can be used in combination at any level of nesting: *exact* (`T@D`) and *inexact* qualifications (`T.D`). While exact qualification supports safe family polymorphism (or binary methods in a broad sense) by restricting subtyping, inexact qualification recovers subtyping by restricting possibly unsafe binary methods. Thereby, they provide rich abstractions to express various kinds of set of objects with flexible subtyping. The name "variant" comes from the facts that: *(i)* the two kinds of qualifications can be seen as operators that, given a path type `T`, take a (local) class name `C` and yield types `T@C` and `T.C` respectively; and *(ii)* such operators have variance properties concerning subtyping/subclassing similarly to variant parametric types [18] (a.k.a. wildcards [28] in Java 5.0 [14]). More specifically, exact qualifications act as invariant: `T@D` is a subtype of `T@E` only when `D = E`; and inexact qualifications act as covariant: `T.D` is a subtype of `T.E` when `D` extends `E` (inside the class of type `T`).

Our technical contributions can be summarized as follows:

[1] This name was derived from the metaphor of a nesting hierarchy of classes as a directory structure in a file system.

- introduction of the notion of variant path types for safe scalable extensibility; and
- formalization of a core language FJ_{path} (extending Featherweight Java [16], or simply FJ) with a sound type system of variant path types.

Full potential of the expressiveness of variant path types and applicability to mainstream languages like Java are to be fully explored, yet. Though, variant path types are interesting for they support safe extensions of groups in a rather simple setting, and can then be considered as starting mechanism to achieve a lightweight form of scalable extensibility.

Rest of This Paper After Section 2 describes the basic framework of classes with arbitrary level of nesting, Section 3 introduces the informal syntax and semantics of variant path types, mainly by means of examples. Then, Section 4 develops the formal core calculus FJ_{path}. Finally, Section 5 discusses related works, and Section 6 provides concluding remarks.

2. Class Nesting and Extension

In this section, we briefly review how the notion of groups and their extension provide scalable extensibility, considering a simplified setting without static types.

2.1 Grouping Classes by Nesting

Like in previous approaches such as JX [23], we see a class as both a mechanism to generate objects and one to group classes. Considering the "graph" example [11], by a class definition of the kind

```
class Graph{
  class Node{
    field edges;
  }
  class Edge{
    field src, dst;
    method connect(node1, node2) {
            src=node1; dst=node2;
    }
  }
  ..
  method createGraph(..){..}
}
```

we define a *group* of classes: classes `Node` and `Edge` are called *member* classes of the *group* class `Graph`. (In order to concentrate on the semantics of groups and their inheritance, in this section we will use keywords `field` and `method` for field/method declarations.) To denote a nested class, we rely on the familiar notation of $\mathtt{C}_1.\mathtt{C}_2.\cdots.\mathtt{C}_n$, which can be used e.g. to create instances out of members `Edge` and `Node` as in the following code:

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

(Again, we use the keyword `var` for variable declarations.) A new instance of member `Edge` (`Node`) inside class `Graph` is assigned to variable `e` (`n`).

A key idea of scalable extensibility is to extend the usual class extension mechanism to allow to inherit not only fields and methods but also member classes, which can be *further extended*. For example, by the definition of the new group class `CWGraph` (a class for graphs of colored nodes and weighted edges) below

```
class CWGraph extends Graph{
  class Node {
    field color;
  }
  class Edge {
    field weight;
    method connect(node1, node2) {
      weight = .. ;
      super.connect(node1, node2);
    }
  }
}
```

`CWGraph` inherits method `createGraph()` and member classes `Node` and `Edge`; furthermore, those member classes are extended simultaneously with new fields and methods such as `color`, `weight`, and overriding `connect()`. Hence, an instance of `CWGraph.Edge` has three fields:

```
var e = new CWGraph.Edge(..);
.. e.weight .. e.src .. e.dst ..
```

This extension mechanism is meant to work at any level of depth in the structure of nesting. If `Graph.Edge` itself defines member classes `A` and `B`, then `CWGraph.Edge.A` and `CWGraph.Edge.B` automatically inherit from the original versions of `A` and `B` inside `Graph.Edge`.

In standard single-inheritance languages such as Java and Smalltalk, the "complete" definition of a subclass is obtained by composing all of its superclasses by taking overriding into account. Here, the complete definition of a class is obtained by *recursively* composing enclosing classes from the top level down to the leaf of the nesting hierarchy [10]. For example, the complete definition of `CWGraph` is obtained by composing `Object`, `Graph` and `CWGraph` in this order; it composes `Node` and `Edge` in `Graph` with those in `CWGraph`, resulting in the expected group of classes.

```
class AST{
  field root;
  class Expr extends Object{
    method toString(){ return ""; }
    method equal(e) { return false; }
  }
  class Literal extends Expr {
    field val;
    method toString(){ return val; }
    method equal(e) { return this.val == e.val; }
  }
  class Plus extends Expr {
    field op1, op2;
    method toString(){
      return this.op1.toString()+
              "+"+this.op2.toString();
    }
    method equal(e) {
      return this.op1.equal(e.op1)
              && this.op2.equal(e.op2);
    }
    method replaceOp1(e) { this.op1 = e; }
  }
}
class ASTeval extends AST {
  class Expr extends Object{
    method eval(){ return 0; }
  }
  class Literal extends Expr{
    method eval(){ return val; }
  }
  class Plus extends Expr{
    method eval(){
      return this.op1.eval() + this.op2.eval();
    }
  }
}
```

Figure 1. Simple Expressions

2.2 Extension inside Group

As discussed elsewhere [12, 23], it is reasonable to expect members of a class to extend another class. In particular, it would be useful to allow a member class to extend from another in the same group to express the so-called expression example [23, 27], as in Figure 1.

The group class `AST` has classes `Literal` and `Plus` for concrete syntax tree nodes that extend a member of the same class

Expr. Each member class is equipped with method toString() to return a string representation of an abstract syntax tree. In an extension ASTeval of AST, each member class is extended with eval() for evaluation. As in the previous example, ASTeval.Plus inherits fields op1 and op2 from AST.Plus. This schema seems to naturally lead to a multiple inheritance scenario: ASTeval.Plus actually inherits from ASTeval.Expr and AST.Plus, and both of these inherit from AST.Expr—thus leading to a typical diamond structure. Notice that, while inheriting from ASTeval.Expr is explicit through the extends clause, inheriting from AST.Plus is implicit, as it is due to the enclosing group extension.

As argued also in Nystrom et al. [23], however, we can avoid problems that typically happen in ordinary multiple-inheritance languages by hierarchical, recursive composition described above. To obtain a complete definition of Plus in ASTeval, for example, the top-level ASTeval is first composed with AST, resulting in member classes each of which is composed with the member class of the same name in AST. Then, the complete definition of Plus is finally obtained by composing Expr and Plus in the composed ASTeval. As a result, priority is given to properties implicitly inherited rather than to explicitly inherited ones.

Note that in general, deeper nesting structures might lead a class to inherit from more than two classes, but the above discussion naturally extends to such cases, as formalized in Section 4.

3. Variant Path Types

Built on top of this language fragment with class nesting and hierarchical composition, we introduce variant path types that allow to flexibly express a number of interesting relationships between classes in a group.

3.1 Absolute vs. Relative Path Types

The ability to automatically inherit member classes (in general a whole structure of nesting) is not sufficient per se to provide a true scalable extensibility mechanism in a statically typed setting. If some relationship exists between members inside a group, e.g., in Graph we have that instances of member Edge should hold a reference to an instance of member Node, then we want it to be preserved through extension, that is, the same relation must automatically hold in class CWGraph as well. More concretely, we may require instances of Graph.Edge to hold references to instances of Graph.Node, and instances of CWGraph.Edge to hold references to instances of CWGraph.Node, as also argued in Ernst [11]: in other words, cross-group reference such as an instance of CWGraph.Node being a source node of Graph.Edge must be disallowed. However, a naive type system as in Java fails to express such an invariant: if we declare src and dst to have type Graph.Node, then those fields would be inherited with the same type, resulting in cross-group reference.

To express such relationship, we introduce a new kind of types called *relative path types* [17], which refer to other classes in a “relative” way from the class where that type appears (as in relative path expressions in the UNIX file system.) Examples of relative path types are This, This.A, This.A.B, ^This, ^^This, ^This.A. Type This means “the current class”—it is found in other languages [23, 4] with a different name such as *MyType* [2]. Analogously, type This.A means “member A inside the current class”, and This.A.B “member B inside member A inside current class”. Type ^This means “the group of the current class” (or “the enclosing class of the current class”), type ^^This “the group of the group of the current class”, and so on. Finally, ^This.A is “member A inside the group of the current class”, which is a type used by a class to denote a member of its same group. A general form $\hat{}\cdots\hat{}$This.C_1.C_2.$\cdots$.C_n of relative path types is hence understood as first going up k times in the nesting structure (k is the number of “^”), and then going down through path C_1.C_2.$\ldots$.C_n.

```
class Graph {
  class Node {
    ^This.Edge[] es=new ^This.Edge[10];
    int i=0;
    void add(^This.Edge e) { es[i++] = e; }
  }
  class Edge {
    ^This.Node src, dst;
    void connect(^This.Node s, ^This.Node d) {
      src = s; dst = d;
      s.add(this); d.add(this);
  } }
  ..
  This.Node startNode;
  boolean containsNode(This.Node n){..}
  boolean containsEdge(This.Edge n){..}
}
class CWGraph extends Graph {
  class Node {
    Color color;
  }
  class Edge {
    int weight;
    void connect(^This.Node s, ^This.Node d) {
      weight = f(s.color, d.color);
      super.connect(s, d);
  } }
}
```

Figure 2. Graph and CWGraph Classes

Going back to the graph example, the intra-group relationship between Edge and Node is expressed by making Edge using type ^This.Node, which means Graph.Node in the class of Graph.Edge, and CWGraph.Node in the class of CWGraph.Edge. Figure 2 shows a complete graph example written in our language. Here, nodes hold a reference to the array of edges of type ^This.Edge and edges hold two references to source and destination nodes of type ^This.Node to express they are from the same kind of graph. In the class CWGraph, types of those fields are inherited as written in the superclass and they now refer to Edge and Node in CWGraph. This example clarifies the need to disallow cross-group references: method connect() invoked through CWGraph must take two instances of CWGraph.Node, otherwise accessing field color on them would fail.

As seen in previous section, relative path types are coupled with types of the kind C_1.$\cdots$.C_n—which we call *absolute path types*, since they denote a certain class independently of the position where such a type is used.

A natural way to exploit the class structure seen above through absolute types is as follows:

```
Graph g = new Graph(..);
..
Graph.Node n = g.startNode;

CWGraph.Edge e;
CWGraph.Node n1,n2;
..
e.connect(n1, n2);
```

Notice that the type of startNode is declared to be This.Node and accessed through the absolute path type Graph yields type Graph.Node by substituting the receiver type Graph for This. Similarly, the argument types of e.connect() becomes CWGraph.Node by replacing ^This in the declared type ^This.Node with CWGraph, which is a prefix of the receiver type CWGraph.Edge.

3.2 Exactness for Type Safety

It is very well known that scalable extensibility suffers from the covariance problem: in the standard framework of “inheritance is subtyping" of mainstream object-oriented languages, it is not safe to use type `This` (and some other relative path type) in certain places such as a method argument type.

In our graph example, although class `CWGraph` inherits `Graph` and class `CWGraph.Node` implicitly inherits from `Graph.Node`, assuming naively `CWGraph` to be a subtype of `Graph` or similarly `CWGraph.Node` to be a subtype of `Graph.Node` will break soundness of the type system as the following code reveals:

```
Graph.Node n1 = new Graph.Node(..);
Graph.Node n2 = new Graph.Node(..);

Graph.Edge e = new CWGraph.Edge(..);
e.connect(n1,n2); // Unsafe call

Graph g = new CWGraph(..);
Graph.Edge e2 = g.startNode.es[0];
e2.connect(n1,n2); // Also unsafe
```

Since the code fragment above is trying to connect a `CWGraph.Edge` to two `Graph.Node`s, the call to `connect()` causes the attempt to access field `color` to a node of type `Graph.Node`, which does *not* have it! Actually, a similar situation occurs only by allowing subtyping between `CWGraph` and `Graph` as the last three lines show.

To solve this problem, some language mechanism is required to ensure that the classes of `e`, `n1`, and `n2` are members of the same group. The solution adopted in JX relies on what they call dependent classes and immutable variables—see Section 5 for a detailed discussion. We instead rely on a simpler solution of exact types [5, 3, 4], briefly reviewed below.

An exact type denotes instances of a single class, excluding any of its subclasses, thus also plays a role of run-time types of objects. We might use the tentative notation `@(A)` to mean an exact type corresponding to the class designated by the absolute path type `A`: for example, exact type `@(Graph.Node)` consists only of instances of class `Graph.Node`. On the other hand, a type `Graph.Node`, which is said to be *inexact*, includes instances of class `Graph.Node` and its subclasses, explicit or implicit.[2] A method taking a relative path type such as `connect()` cannot be invoked on inexact `Graph.Edge`, as we do not know whether an actual instance belongs to the group `Graph` or `CWGraph`. Thus, invocation of a method taking a relative path type is allowed only when the receiver type is exact; the argument type obtained by replacing `This` (or `^···^This`) will also be considered exact. In this sense, `This` (possibly with `^`) is always exact.

By using exact types, the type system can reject the example above: invocation of `connect()` on inexact type `Graph.Edge` is prohibited. If the type of `e` were declared to be `@(Graph.Edge)` so that `connect()` can be invoked: the assignment

```
@(Graph.Edge) e = new CWGraph.Edge(..);
```

before the invocation would be prohibited because `@(Graph.Edge)` is *not* a supertype of `@(CWGraph.Edge)`. (Expressions `new` will be given exact types since the class is known.)

3.3 Exact and Inexact Qualifications and Subtyping

In the above section, `@` was treated as an operator to absolute path types. However, in our setting, we have found that it is more natural to consider that `@` is rather a new kind of qualification in addition to `.`, in order to control the degree of exactness in a more fine-grained manner! So, for class `AST.Expr`, say, variant path types now feature four kinds of types: a *fully exact* type `@AST@Expr` (which was written `@(AST.Expr)` above), *partially inexact* types `.AST@Expr` and `@AST.Expr`, and usual `.AST.Expr`. We call the "dot" *inexact qualification* and the "`@`" *exact qualification*. Here, `@` at the head can be considered an exact qualification over the top level, or package. An inexact qualification over the top level can be omitted for syntactic analogy with Java, writing e.g. `AST.Expr` instead of `.AST.Expr`. (In the formal calculus introduced in the next section, even "the top level" will be made explicit as the symbol `/` and, for example, `AST.Expr` will be formally written `/.AST.Expr`.)

The intuition behind a type like `@A.B` is as "the common supertype of all the members that extends `B` inside class `A`" (`@A@B` included). So, type `@AST.Expr` is a common supertype of `@AST@Expr`, `@AST@Literal`, and `@AST@Plus`. Similarly, `A@B` is read as "the common supertype of member `B` in the group `A` or its subclasses" (`@A@B` included). So, `AST@Expr` is a common supertype of `@AST@Expr` and `@ASTeval@Expr` but not `@AST@Literal`. Figure 3 shows the subtyping hierarchy for abstract syntax nodes. The name "variant path types" comes from the two kinds of qualifications, which introduce different variance with respect to the simple class name after qualification: symbol `@` acts as invariant—`T@D` is a subtype of `T@E` only when `D` = `E`—and `.` acts as covariant—`T.D` is a subtype of `T.E` when `D` extends `E` (inside the class of `T`).

Now, dots in relative path types are also considered inexact qualification: for instance `This.B` would be "the common supertype of all the members that extends `B` inside the current class", and `^This.B` "the common supertype of all the members that extends `B` inside the enclosing class". Thus, type `This.Expr` used inside the code of class `AST` would denote the set of all nodes of the current version of abstract syntax tree. Now, `AST` with type annotations can be written as follows:

```
class AST {
  class Expr {..}
  class Literal extends Expr{..}
  class Plus extends Expr{
    ^This.Expr op1, op2;
    String toString(){
      return this.op1.toString()
             + "+" + this.op2.toString();
    }
    boolean equal(This e){
       return this.op1.equal(e.op1)
              && this.op2.equal(e.op2);
    }
    void replaceOp1(^This.Expr e) {
       this.op1 = e; return;
    }
  }
}
```

Since type `@AST.Expr` is a common supertype to all kinds of expressions of `AST`, it is clear from the substitutability principle that it should provide a more restricted access to its methods and fields than `@AST@Expr`. In this case, since `equal()` takes a relative path type `This`, it cannot be invoked on `@AST.Expr` as the receiver type is not exact. However, a method taking a relative path type may be invoked on a (partially) inexact type. For example replaceOp1() requires the argument to be any expression (hence inexact `.Expr`) that belongs to the same group (hence `^This`). So, this method can be invoked on both `@AST@Plus` and `@AST.Plus`. The rule of thumb is that a method taking a relative path type can be invoked when the type replacing `^···^This` is exact: in this case, `^This` in `^This.Expr` is replaced with exact `@AST`, a prefix of `@AST.Plus`.

Remark: One might want to use `^This@Edge` and `^This@Node` rather than `^This.Edge` and `^This.Node` in the graph example above. The choice would not matter in this particular code because nested classes `Node` and `Edge` do not have a binary method (such as `equal()` taking an argument of type `This`). If `Node` had `equal()`, invoking it inside `connect()` on `s` or `d` would be prohibited because the type is (partially) inexact. In such a case, their type must be `^This@Node`, which is fully exact.

[2] Note that the same notation "`Graph.Node`" is used sometimes to denote a single *class* named `Node` nested in `Graph` and sometimes to denote an inexact *type*.

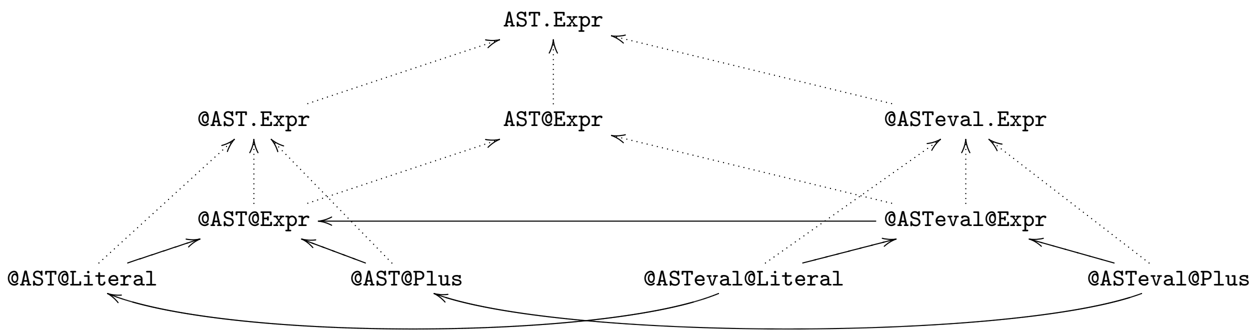

Figure 3. The rich subtyping hierarchy for the expression example. Dotted arrows represent subtyping while solid arrows represent inheritance, which is *not* subtyping.

3.4 Parametric Methods for Group-Polymorphic Methods

One of the central ideas in family polymorphism [11] is that it should be possible to develop functionalities that can work uniformly over different families. Recasting it to our framework, it means that we should be able to write methods accepting as formal arguments instances of members of the same group, where different invocations may be concerned about different groups.

As an example, we consider the method `connectAll()` that takes as input an array of edges and two nodes of *any* group (of graphs) and connects each edge to the two nodes. We achieve it by adding parametric methods in the style of Java 5.0 to our language, but with new features of *exact type variables* with qualification. More concretely, method `connectAll()` is written as follows:

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

Method `connectAll()` is defined as parametric in an exact type variable `G`—which represents the group used for each invocation—with upper-bound `Graph`; and the arguments are of type `G@Edge[]`, `G@Node` and `G@Node`, respectively. It can be invoked as follows:

```
@Graph@Edge[] ges = .. ;
@Graph@Node gn1 = .. , gn2 = .. ;
@CWGraph@Edge[] ces = .. ;
@CWGraph@Node cn1 = .. , cn2 = .. ;

<@Graph>connectAll(ges, gn1, gn2);   // OK
<@CWGraph>connectAll(ces, cn1, cn2); // OK
<@Graph>connectAll(ces, gn1, gn2);
     // compile-time error
<Graph>connectAll(ces, gn1, gn2);
     // compile-time error
```

In the first invocation of the example code, instantiation of `G` with `@Graph` is specified, hence edges and nodes of family `Graph` can be passed, and similarly in the second invocation for `CWGraph`. The third invocation is not well typed, as `ces` has type `@CWGraph@Edge`, which does not belong to the group `@Graph`. (In other words, it is not a subtype of `@Graph@Edge`.) Finally, the last one is not well typed, either, since an inexact type `Graph` is passed to an exact type variable. Notice that the introduction of exact type variables is crucial: `connect()` is allowed to be invoked in the method body exactly for the reason that `G` is an exact type and, if the fourth invocation were allowed, it would lead to unsoundness.

Finally, as developed in our previous work [17], a type inference mechanism can also be designed by extending that in Java 5.0, so that the instantiation of type variables can be automatically inferred—it is left for future work.

4. Formalizing Variant Path Types

In this section, we formalize the ideas described in the previous section as a small core calculus called $\mathrm{FJ}_{\mathrm{path}}$ based on Featherweight Java [16]. What we model here includes nested classes with hierarchical composition, variant path types, and parametric methods only with exact type variables, as well as the usual features of FJ, that is, fields, object instantiation, and recursion by `this`. In $\mathrm{FJ}_{\mathrm{path}}$, a nested class can extend either `Object`, which is an empty class, or another class in the same group, though some other languages [19, 23] allow a more liberal style of inheritance. We drop typecasts since one of our points is to show scalable extensibility is possible without resorting to typecasts, which are used to get around restrictions imposed by a naive type system. We assume every type variable to be exact for simplicity and hence drop the `exact` keyword; non-exact type variables would be easy to add.

4.1 Syntax

The abstract syntax of types, class declarations, method declarations, and expressions is given in below. Here, n is a natural number (0 or positive integers); the metavariables `C` and `D` range over (simple) class names; `X` and `Y` range over (exact) type variables; `S`, `T`, `U`, and `V` range over types; `f` and `g` range over field names; `m` ranges over method names; and `x` ranges over variables.

$\mathtt{A}$	$::=$	$\mathtt{/} \mid \mathtt{A@C}$	*run-time types*
$\mathtt{E}$	$::=$	$\mathtt{/} \mid \mathtt{X}^n \mid \mathtt{E@C}$	*exact types*
$\mathtt{T}$	$::=$	$\mathtt{/} \mid \mathtt{X}^n \mid \mathtt{T@C} \mid \mathtt{T.C}$	*types*
$\mathtt{L}$	$::=$	$\mathtt{class\ C} \triangleleft \mathtt{C}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\ \}$	*classes*
$\mathtt{M}$	$::=$	$\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T}}\mathtt{>T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{return\ e;\}}$	*methods*
$\mathtt{e}$	$::=$	$\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.<}\overline{\mathtt{E}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{)}$	*expressions*

Following the custom of FJ, we put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$;" for "$\mathtt{T}_1\ \mathtt{f}_1;\ldots;\ \mathtt{T}_n\ \mathtt{f}_n;$", where n is the length of $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$, and "$\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}};$" as shorthand for "$\mathtt{this.f}_1\mathtt{=f}_1;\ldots;\mathtt{this.f}_n\mathtt{=f}_n;$" and so on. Sequences of field declarations, parameter names, method definitions, nested class definitions are assumed to contain no duplicate names. We write the empty sequence as $\bullet$, denote the length of a sequence using $|\cdot|$ and concatenation of sequences using a comma. Unlike the previous section, we make the top level explicit as `/` in the formal syntax but we often abbreviate `/@C` to `@C` and `/.C` to `C`.

Run-time types, which represent classes from which objects are instantiated, are also called absolute path types, while types starting with $\mathtt{X}^n$, which corresponds to `^.. ^X` (with `^` n times) in the previous section, are called relative path types. Here, we extend the prefixing operation from `This` to all type variables. Also note that for notational convenience we use absolute path types for `new` expressions and names of classes. A qualification of the form `@C` is called exact while `.C` is called inexact. In particular, a type without any inexact qualification is called an exact type, ranged over by `E` as shown above. In what follows, we use the notation $\mathtt{T}^n$ to drop the last n qualifications from `T`; it is defined by:

$$\begin{array}{lcll} \mathtt{T}^0 & = & \mathtt{T} & \\ (\mathtt{X}^n)^m & = & \mathtt{X}^{n+m} & \\ (\mathtt{T@C})^n & = & \mathtt{T}^{n-1} & (n > 0) \\ (\mathtt{T.C})^n & = & \mathtt{T}^{n-1} & (n > 0) \end{array}$$

Note that $(\cdot)^n$ is an operation on types while $\mathtt{X}^n$ is just a syntactic entity. By using the prefixing operation, (simultaneous) substitution $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$ of types for type variables is defined as follows:

$$\begin{array}{lcl} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{/} & = & \mathtt{/} \\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{X}_i^n & = & \mathtt{T}_i^n \\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{S@C}) & = & ([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S})\mathtt{@C} \\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{S.C}) & = & ([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S})\mathtt{.C} \end{array}$$

Note that $\mathtt{X}^n$ is replaced with the corresponding prefix of `T`. We also use a notation *exact*(`T`) (*inexact*(`T`), resp.) to denote types in which all inexact (exact, resp.) qualifications in `T` are replaced by exact (inexact, resp.) ones. We include `/` (read “top-level”) without any qualification mostly for technical convenience and, as seen in rules for well-formed types and typing, `/` by itself cannot appear in any program texts.

A class declaration consists of its name, the simple name of its superclass, field declarations, methods, and nested classes. The symbol $\triangleleft$ is read “`extends`.” A method declaration can be parameterized by type variables $\overline{\mathtt{X}}$, which we assume to be exact for simplicity—it is easy to extend the language to inexact type variables. 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 (type) variables includes the special variable `this` (`This`, resp.), which cannot be used as the name of a (type, resp.) parameter to a method.

A class table *CT* is a finite mapping from run-time types `A` to (top-level or nested) class declarations and is assumed to satisfy the following sanity conditions to identify a class table with a set of top-level classes: (1) $CT(\mathtt{A@C}) = \mathtt{class\ C\ ..}$ for every $\mathtt{A@C} \in dom(CT)$; (2) if $CT(\mathtt{A@C})$ has an inner class declaration `L` of name `D`, then $CT(\mathtt{A@C@D}) = \mathtt{L}$; and (3) $\mathtt{Object} \notin dom(CT)$. A program is a pair $(CT, \mathtt{e})$ of a class table and an expression. To lighten the notation in what follows, we always assume a *fixed* class table *CT*.

4.2 Hierarchical Composition

As discussed in Section 2, a complete definition of a nested class is obtained by propagating composition of enclosing classes in a top-down manner. We define a function *classes*(`A`) to list up nested classes inside `A` after hierarchical composition of `A`. It requires the following auxiliary operator $\mathtt{L}_1\mathtt{<+}\mathtt{L}_2$ to compose a superclass $\mathtt{L}_1$ with a subclass $\mathtt{L}_2$:

$$\begin{array}{l} \mathtt{class\ C} \triangleleft \mathtt{E}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{L}}_1\ \overline{\mathtt{M}}_1\}\ \mathtt{<+}\ \mathtt{class\ C} \triangleleft \mathtt{E}'\{\overline{\mathtt{U}}\ \overline{\mathtt{g}};\ \overline{\mathtt{L}}_2\ \overline{\mathtt{M}}_2\} \\ = \mathtt{class\ C} \triangleleft \mathtt{E}'\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{U}}\ \overline{\mathtt{g}};\ (\overline{\mathtt{L}}_1\mathtt{<+}\overline{\mathtt{L}}_2)\ (\overline{\mathtt{M}}_1\mathtt{<+}\overline{\mathtt{M}}_2)\} \end{array}$$

Here, $\overline{\mathtt{L}}_1\mathtt{<+}\overline{\mathtt{L}}_2$ denotes the set union of classes from $\overline{\mathtt{L}}_1$ and $\overline{\mathtt{L}}_2$ where classes of the same name are recursively composed by `<+`. Similarly, $\overline{\mathtt{M}}_1\mathtt{<+}\overline{\mathtt{M}}_2$ denotes the set union of methods from $\overline{\mathtt{M}}_1$ and $\overline{\mathtt{M}}_2$ where methods in $\overline{\mathtt{M}}_2$ have priorities over the method of the same name in $\overline{\mathtt{M}}_1$. Their straightforward definitions are omitted here for brevity. Here, there is no condition on `E` and `E'` for the composition to be well defined but, in a well-typed program, `E` is expected to be a subclass of `E'`.

The definition of function *classes*(`A`) is in Figure 4: the first rule says that `Object` has no nested classes and the second that `/` is the top-level. The third rule means that nested classes in `A@C` are obtained by composing nested classes in `C` in *classes*(`A`) with those in its superclass `A@D`. Note that $\overline{\mathtt{L}}$ are also the result of composition till the depth of the enclosing class `A`.

For example, consider the following $\mathrm{FJ}_{\mathrm{path}}$ classes:

```
class AST extends Object {
  class Expr extends Object {
    T m() { return e_1; }
  }
  class Lit extends Expr {
  }
  class Plus extends Expr {
    T m() { return e_3; }
  }
}
class ASTE extends AST {
  class Expr extends Object {
  }
  class Lit extends Expr {
    T m() { return e_5; }
  }
  class Plus extends Expr {
    T m() { return e_6; }
  }
}
```

Then, *classes*(`/@ASTE`) returns nested classes `Expr`, `Lit`, `Plus` obtained by composing ones inside `ASTE` and its superclass `AST`, i.e.,

```
class Expr extends Object {
  T m() { return e_1; }
}
class Lit extends Expr {
  T m() { return e_5; }
}
class Plus extends Expr {
  T m() { return e_6; }
}
```

Here, method `m` in class `@AST@Plus` has disappeared as it is overridden by one in class `@ASTE@Plus`, which implicitly extends `@AST@Plus`.

Thanks to *classes*(`A`), it is now easy to define functions to look up fields and methods from a given class name. The definitions of lookup functions are also in Figure 4. Function *fields*(`T`, `A`), which is similar to *classes*(`T`, `A`), enumerates all field names of `A` (and its superclasses) with their types, which are resolved with the first argument, which is usually the type of the receiver. Similarly, *mtype*(`m`, `A`) returns the signature of method `m` in `A`.

Now, it is fairly easy to read off how class bodies are linearized, i.e., in what order members are looked up: for example, methods of an instance of `@ASTE@Lit` will be searched in `@ASTE@Lit`, `@AST@Lit`, `@ASTE@Expr`, `@AST@Expr` in this order.

4.3 Type System

The main judgments of the type system consist of one for type equivalence $\Delta \vdash \mathtt{S} \equiv \mathtt{T}$, one for matching $\Delta \vdash \mathtt{E}_1\ \mathtt{<\#}\ \mathtt{E}_2$, one for subtyping $\Delta \vdash \mathtt{S}\ \mathtt{<:}\ \mathtt{T}$, one for type well-formedness $\Delta \vdash \mathtt{T}\ \mathsf{ok}$, and one for typing $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$. The rules are given in Figures 5 and 6. Here, Δ, called *bound environment*, is a finite mapping written $\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{T}}$ from type variables $\overline{\mathtt{X}}$ to types $\overline{\mathtt{T}}$ and records declarations of type variables with their respective upper bounds. Similarly, Γ, called *type environment*, is a finite mapping written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$ from variables $\overline{\mathtt{x}}$ to $\overline{\mathtt{T}}$ and records declarations of method parameters with their respective types. As seen later, Δ usually contains `This<:T`, in which `T` represents the class where the judgment is made.

Following the custom of FJ [16], we abbreviate a sequence of judgments in the obvious way: $\Delta \vdash \mathtt{S}_1\ \mathtt{<:}\ \mathtt{T}_1, \ldots, \Delta \vdash \mathtt{S}_n\ \mathtt{<:}\ \mathtt{T}_n$

$\boxed{\textit{classes}(\mathtt{A})}$

$$\textit{classes}(\mathtt{Object}) = \bullet \qquad \frac{(\overline{\mathtt{L}} \text{ are all top-level classes})}{\textit{classes}(\mathtt{/}) = \overline{\mathtt{L}}}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D}\ \{..\ \overline{\mathtt{L}}..\ \} \in \textit{classes}(\mathtt{A}) \qquad \textit{classes}(\mathtt{A@D}) = \overline{\mathtt{L}}'}{\textit{classes}(\mathtt{A@C}) = \overline{\mathtt{L}}'\mathtt{<+}\overline{\mathtt{L}}}$$

$\boxed{\textit{fields}(\mathtt{A})}$

$$\textit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ ..\ \} \in \textit{classes}(\mathtt{A})}{\textit{fields}(\mathtt{A@C}) = \textit{fields}(\mathtt{A@D}), \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

$\boxed{\textit{mtype}(\mathtt{m}, \mathtt{A})}$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D}\ \{..\ \overline{\mathtt{M}}\ \} \in \textit{classes}(\mathtt{A}) \qquad \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>S_0\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{\ ..\ \}} \in \overline{\mathtt{M}}}{\textit{mtype}(\mathtt{m}, \mathtt{A@C}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S_0}}$$

$$\frac{\mathtt{class\ C} \triangleleft \mathtt{D}\ \{..\ \overline{\mathtt{M}}\ \} \in \textit{classes}(\mathtt{A}) \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \textit{mtype}(\mathtt{m}, \mathtt{A@D}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S_0}}{\textit{mtype}(\mathtt{m}, \mathtt{A@C}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S_0}}$$

Figure 4. $\mathrm{FJ_{path}}$: Lookup Functions

to $\Delta \vdash \overline{\mathtt{S}}$ <: $\overline{\mathtt{T}}$ (similarly for type equivalence and matching); $\Delta \vdash \mathtt{T_1}$ ok, ..., $\Delta \vdash \mathtt{T}_n$ ok to $\Delta \vdash \overline{\mathtt{T}}$ ok; and $\Delta; \Gamma \vdash \mathtt{e_1}:\mathtt{T_1}$, ..., $\Delta; \Gamma \vdash \mathtt{e}_n:\mathtt{T}_n$ to $\Delta; \Gamma \vdash \overline{\mathtt{e}}:\overline{\mathtt{T}}$.

4.3.1 Type Equivalence

The judgment $\Delta \vdash \mathtt{S} \equiv \mathtt{T}$ can be read "type S is equivalent to T under Δ." The first three rules say that it is indeed an equivalence relation, and the last two that it is a congruence. The key rule is the fourth rule, which says that if the upperbound of a type variable is exact, then the two types are indeed equivalent. The fifth rule means that if $\mathtt{X}^n$ is equivalent to an exact type E.C, then its enclosing class $\mathtt{X}^{n+1}$ also has C and they are equivalent: for example,

$$\mathtt{X}\ \mathtt{<:}\ \mathtt{@Graph@Node} \vdash \mathtt{X} \equiv \mathtt{X}^1\mathtt{@Node}$$

can be derived.

4.3.2 Matching

The subtyping relation will be defined by using the inheritance relation, which is formalized as matching here. The judgment $\Delta \vdash \mathtt{E_1}$ <# $\mathtt{E_2}$ can be read "exact type $\mathtt{E_1}$ matches $\mathtt{E_2}$" or simply "$\mathtt{E_1}$ extends $\mathtt{E_2}$." So, the matching relation is defined essentially as the transitive closure of type equivalence, as seen in the first two rules. The third rule means that, if X is assumed to be a subtype of T, then it must extend *exact*(T) whatever it is instantiated with. The fourth rule is similar to the fifth rule for type equivalence: for example, the matching judgment

$$\mathtt{This}\ \mathtt{<:}\ \mathtt{Graph.Node} \vdash \mathtt{This}\ \mathtt{<\#}\ \mathtt{This}^1\mathtt{@Node}$$

can be derived by this rule. The second last rule deals with class extension.

4.3.3 Subtyping

The judgment form for subtyping $\Delta \vdash \mathtt{S}$ <: T can be read "S is subtype of T under Δ." As usual, subtyping is reflexive and transitive and a type variable (with some prefixing) is a subtype of (the corresponding prefix of) its declared upper bound. The third last rule intuitively means that a type denoting a nested class C exactly is included in a type denoting a nested class C and its subclasses. The second last rule might look counterintuitive since exact qualification works covariantly. Note that, however, if T is not exact, the resulting type T@C is not exact, either. For example, @ASTeval.Expr, which includes all kinds of expressions in ASTeval.Expr, is a subtype of AST.Expr, which includes all kinds of expressions in AST and ASTeval. The last rule roughly means that inexact types are related if one inherits the other—matching is used in this rule.

4.3.4 Type-Wellformedness

The judgment form for well formed types is $\Delta \vdash \mathtt{T}$ ok, read as "T is well formed under Δ." A type is well formed when the class that the type points to in A exists. Even when the class of a given name is not in the domain of the class table, it may exist due to nested inheritance, hence the function *classes* is used in the last two rules.

4.3.5 Typing

The typing judgment form $\Gamma \vdash \mathtt{e} : \mathtt{T}$ is read "expression e is given type T under Γ." The typing rules are shown in Figure 6; readers who are familiar with languages with matching [2], in particular LOOJ [4], will notice some similarities. The key rules are T-FIELD and T-INVK. The rule T-FIELD means that the type of field access $\mathtt{e_0.f}_i$ is obtained by looking up field declarations from the class that matches the receiver type. Note that, if $\mathtt{f}_i$'s type is declared to be relative, then $\mathtt{This}^i$ will be replaced with the corresponding prefix of the receiver type: for example, if $\textit{fields}(\mathtt{@CWGraph@Node}) = \mathtt{This}^1\mathtt{@Edge\ edg}$ and $\Gamma = \mathtt{x} : \mathtt{@CWGraph@Node}, \mathtt{y} : \mathtt{This}^1\mathtt{@Node}$, then

$$\mathtt{This}\ \mathtt{<:}\ \mathtt{CWGraph.Node}, \ldots; \Gamma \vdash \mathtt{x.edg} : \mathtt{@CWGraph@Edge} \text{ and}$$
$$\mathtt{This}\ \mathtt{<:}\ \mathtt{CWGraph.Node}, \ldots; \Gamma \vdash \mathtt{y.edg} : \mathtt{This}^1\mathtt{@Edge}.$$

$$\boxed{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}$$

$$\Delta \vdash \mathtt{T} \equiv \mathtt{T} \qquad \frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}{\Delta \vdash \mathtt{T} \equiv \mathtt{S}} \qquad \frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T} \qquad \Delta \vdash \mathtt{T} \equiv \mathtt{U}}{\Delta \vdash \mathtt{S} \equiv \mathtt{U}} \qquad \frac{\mathtt{X} \mathrel{\mathtt{<:}} \mathtt{T} \in \Delta \qquad \mathtt{T}^n \text{ is exact}}{\Delta \vdash \mathtt{X}^n \equiv \mathtt{T}^n}$$

$$\frac{\Delta \vdash \mathtt{X}^n \equiv \mathtt{E@C}}{\Delta \vdash \mathtt{X}^n \equiv \mathtt{X}^{n+1}\mathtt{@C}} \qquad \frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}{\Delta \vdash \mathtt{S@C} \equiv \mathtt{T@C}} \qquad \frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}{\Delta \vdash \mathtt{S.C} \equiv \mathtt{T.C}}$$

$$\boxed{\Delta \vdash \mathtt{E}_1 \mathrel{\mathtt{<\#}} \mathtt{E}_2}$$

$$\frac{\Delta \vdash \mathtt{E}_1 \equiv \mathtt{E}_2}{\Delta \vdash \mathtt{E}_1 \mathrel{\mathtt{<\#}} \mathtt{E}_2} \qquad \frac{\Delta \vdash \mathtt{E}_1 \mathrel{\mathtt{<\#}} \mathtt{E}_2 \qquad \Delta \vdash \mathtt{E}_2 \mathrel{\mathtt{<\#}} \mathtt{E}_3}{\Delta \vdash \mathtt{E}_1 \mathrel{\mathtt{<\#}} \mathtt{E}_3} \qquad \frac{\mathtt{X} \mathrel{\mathtt{<:}} \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X}^n \mathrel{\mathtt{<\#}} \mathit{exact}(\mathtt{T}^n)}$$

$$\frac{\Delta \vdash \mathtt{X}^n \mathrel{\mathtt{<\#}} \mathtt{E@C}}{\Delta \vdash \mathtt{X}^n \mathrel{\mathtt{<\#}} \mathtt{X}^{n+1}\mathtt{@C}} \qquad \frac{\Delta \vdash \mathtt{E} \mathrel{\mathtt{<\#}} \mathtt{A} \qquad \mathtt{class\ C} \lhd \mathtt{D}\ \{..\} \in \mathit{classes}(\mathtt{A})}{\Delta \vdash \mathtt{E@C} \mathrel{\mathtt{<\#}} \mathtt{E@D}} \qquad \frac{\Delta \vdash \mathtt{E}_1 \mathrel{\mathtt{<\#}} \mathtt{E}_2}{\Delta \vdash \mathtt{E}_1\mathtt{@C} \mathrel{\mathtt{<\#}} \mathtt{E}_2\mathtt{@C}}$$

$$\boxed{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}}$$

$$\frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}} \qquad \frac{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T} \qquad \Delta \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{U}}{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{U}} \qquad \frac{\mathtt{X} \mathrel{\mathtt{<:}} \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X}^n \mathrel{\mathtt{<:}} \mathtt{T}^n}$$

$$\frac{\Delta \vdash \mathtt{X}^n \mathrel{\mathtt{<:}} \mathtt{T.C}}{\Delta \vdash \mathtt{X}^n \mathrel{\mathtt{<:}} \mathtt{X}^{n+1}\mathtt{.C}} \qquad \Delta \vdash \mathtt{T@C} \mathrel{\mathtt{<:}} \mathtt{T.C}$$

$$\frac{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}}{\Delta \vdash \mathtt{S@C} \mathrel{\mathtt{<:}} \mathtt{T@C}} \qquad \frac{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T} \qquad \Delta \vdash \mathit{exact}(\mathtt{S})\mathtt{@C} \mathrel{\mathtt{<\#}} \mathit{exact}(\mathtt{T})\mathtt{@D}}{\Delta \vdash \mathtt{S.C} \mathrel{\mathtt{<:}} \mathtt{T.D}}$$

$$\boxed{\Delta \vdash \mathtt{T}\ \text{ok}}$$

$$\Delta \vdash \mathtt{Object}\ \text{ok} \qquad \frac{\mathtt{X} \mathrel{\mathtt{<:}} \mathtt{T} \in \Delta \qquad \Delta \vdash \mathtt{T}^n\ \text{ok}}{\Delta \vdash \mathtt{X}^n\ \text{ok}} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{..\} \in \mathit{classes}(/)}{\Delta \vdash \mathtt{C}\ \text{ok}}$$

$$\frac{\Delta \vdash \mathtt{T}\ \text{ok} \qquad \Delta \vdash \mathit{exact}(\mathtt{T}) \mathrel{\mathtt{<\#}} \mathtt{A} \qquad \mathtt{class\ C} \lhd \mathtt{D}\ \{..\} \in \mathit{classes}(\mathtt{A})}{\Delta \vdash \mathtt{T@C}\ \text{ok}}$$

$$\frac{\Delta \vdash \mathtt{T}\ \text{ok} \qquad \Delta \vdash \mathit{exact}(\mathtt{T}) \mathrel{\mathtt{<\#}} \mathtt{A} \qquad \mathtt{class\ C} \lhd \mathtt{D}\ \{..\} \in \mathit{classes}(\mathtt{A})}{\Delta \vdash \mathtt{T.C}\ \text{ok}}$$

Figure 5. $\mathrm{FJ}_{\mathrm{path}}$: Matching, Subtyping, and Type Well-formedness Rules

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.

The first line means that the type of the receiver $\mathtt{T}_0$ matches (i.e., inherits) a class `A` that has method `m` with the signature $\mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{S}_0$. The second and third lines roughly mean that the actual type arguments must be subtypes of the corresponding upperbounds $\overline{\mathtt{U}}$ and the types of the actual value arguments must be subtypes of the corresponding formal; the substitution is applied since $\mathtt{U}_i$ may include $\mathtt{X}_i, \ldots, \mathtt{X}_{i-1}$ and $\overline{\mathtt{T}}$ may include $\overline{\mathtt{X}}$. As discussed in the last section, binary methods can be invoked only when the receiver type is exact and, in general, prefixed `This` must be exact. For example, assume $\mathit{mtype}(\mathtt{@Graph@Edge}, \mathtt{connect}) = (\mathtt{This}^1\mathtt{@Node}, \mathtt{This}^1\mathtt{@Node}) \rightarrow \mathtt{void}$. Then

$$\cdot; \mathtt{x} : \mathtt{@Graph@Edge}, \mathtt{y} : \mathtt{@Graph@Node} \vdash \mathtt{x.connect(y,y)} : \mathtt{void}$$

should be derived but not

$$\cdot; \mathtt{x} : \mathtt{Graph.Edge}, \mathtt{y} : \mathtt{Graph@Node} \vdash \mathtt{x.connect(y,y)} : \mathtt{void}$$

In order to express this condition, we use another substitution operator $[\mathtt{T}/\mathtt{@X}]$, which requires $\mathtt{X}^n$ in `T` be replaced with an exact type.[3] The definition, which is omitted here, is derived from that of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$ by imposing a side condition "$\mathtt{T}_i^n$ is exact" on the second clause. Thus, $[\mathtt{Graph.Edge}/\mathtt{@This}]\mathtt{This}^1\mathtt{@Node}$ is not well defined, making the second judgment above non-derivable. Note that, even if the receiver type $\mathtt{T}_0$ contains inexact qualification, $[\mathtt{T}_0/\mathtt{@This}]$ may succeed as in $[\mathtt{@Graph.Edge}/\mathtt{@This}^1\mathtt{@Node}]\mathtt{This}^1\mathtt{@Node} = \mathtt{@Graph@Node}$. So,

$$\cdot; \mathtt{x} : \mathtt{@Graph.Edge}, \mathtt{y} : \mathtt{Graph@Node} \vdash \mathtt{x.connect(y,y)} : \mathtt{void}$$

is derivable.

The judgment for methods is of the form $\vdash$ `M` ok in `A`, read "method `M` is ok in `A`." The rule T-METHOD checks whether the method body is well typed, provided that `this` is of type `This` and that formal type and value parameters are given declared upper bounds and declared types, respectively. `This` is bounded by $\mathit{inexact}(\mathtt{A})$, where `A` is the class name in which the method is declared, since the method, which may be inherited to subclasses of `A`, has to work for any subclass of `A`. Like FJ, the signatures of overriding methods must be identical with the overridden, but this condition will be checked by T-CLASS (unlike FJ).

The judgment for classes is of the form $\vdash$ `L` ok in `A`, read "class `L` is ok in `A`." The rule T-CLASS means that a class is well formed if (1) its superclass, field types, nested classes, and methods are all well formed; (2) it extends `Object`—in other words, there is no cycle in the inheritance relation; (3) defined methods $\overline{\mathtt{M}}$ correctly override ones inherited from the superclass; and (4) nested classes $\overline{\mathtt{L}}$ correctly override those inherited from the superclass. For the condition (4), another judgment $\vdash \mathtt{L}$ overrides $\mathtt{L}'$ in `A` is introduced and it is checked by the second last rule of Figure 6 that methods in `L` and classes further nested in $\overline{\mathtt{L}}$ correctly override those in $\overline{\mathtt{L}}'$. Note that $\overline{\mathtt{L}}_s$ in the rule T-CLASS are not necessarily class definitions in the class table; rather, they are obtained by combining nested

[3] This requirement is essentially the same as *exactness preservation* [24].

$\boxed{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}}$

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

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathit{exact}(\mathtt{T}_0) \mathrel{\texttt{<\#}} \mathtt{A} \qquad \mathit{fields}(\mathtt{A}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : [\mathtt{T}_0/\mathtt{This}]\mathtt{T}_i} \tag{T-FIELD}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathit{exact}(\mathtt{T}_0) \mathrel{\texttt{<\#}} \mathtt{A} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{A}) = \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{U}}\texttt{>}\overline{\mathtt{T}} \rightarrow \mathtt{S}_0 \\ \Delta \vdash \overline{\mathtt{E}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{E}} \mathrel{\texttt{<:}} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T}_0/\texttt{@This}]\overline{\mathtt{U}} \\ \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{\texttt{<:}} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T}_0/\texttt{@This}]\overline{\mathtt{T}}\end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0.\texttt{<}\overline{\mathtt{E}}\texttt{>}\mathtt{m}(\overline{\mathtt{e}}) : [\overline{\mathtt{E}}/\overline{\mathtt{X}}, \mathtt{T}_0/\mathtt{This}]\mathtt{S}_0} \tag{T-INVK}$$

$$\frac{\Delta \vdash \mathtt{A}_0\ \text{ok} \qquad \mathit{fields}(\mathtt{A}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{\texttt{<:}} ([\mathtt{A}_0/\mathtt{This}]\overline{\mathtt{T}})}{\Delta; \Gamma \vdash \mathtt{new}\ \mathtt{A}_0(\overline{\mathtt{e}}) : \mathtt{A}_0} \tag{T-NEW}$$

$\boxed{\vdash \mathtt{M}\ \text{ok in}\ \mathtt{A}}$

$$\frac{\begin{array}{c}\forall i \in 1..|\overline{\mathtt{U}}|.(\mathtt{This} \mathrel{\texttt{<:}} \mathit{inexact}(\mathtt{A}), \mathtt{X}_1 \mathrel{\texttt{<:}} \mathtt{U}_1, \ldots, \mathtt{X}_{i-1} \mathrel{\texttt{<:}} \mathtt{U}_{i-1} \vdash \mathtt{U}_i\ \text{ok}) \\ \Delta = \mathtt{This} \mathrel{\texttt{<:}} \mathit{inexact}(\mathtt{A}), \overline{\mathtt{X}} \mathrel{\texttt{<:}} \overline{\mathtt{U}} \\ \Delta \vdash \mathtt{T}_0, \overline{\mathtt{T}}\ \text{ok} \qquad \Delta; \mathtt{this} : \mathtt{This}, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{S}_0 \qquad \Delta \vdash \mathtt{S}_0 \mathrel{\texttt{<:}} \mathtt{T}_0\end{array}}{\vdash \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{U}}\texttt{>}\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\texttt{\{ return e; \}}\ \text{ok in}\ \mathtt{A}} \tag{T-METHOD}$$

$\boxed{\vdash \mathtt{L}\ \text{overrides}\ \mathtt{L}'\ \text{in}\ \mathtt{A}}$

$$\frac{\begin{array}{c}\vdash \mathtt{A@D}_1 \mathrel{\texttt{<\#}} \mathtt{A@D}_2 \\ \text{for any } \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{U}}\texttt{>}\mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\{..\} \in \overline{\mathtt{M}}, \\ \text{if } \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{U}}'\texttt{>}\mathtt{S}'_0\ \mathtt{m}(\overline{\mathtt{S}}'\ \overline{\mathtt{y}})\{..\} \in \overline{\mathtt{M}}', \text{then } [\overline{\mathtt{X}}/\overline{\mathtt{Y}}](\overline{\mathtt{U}}', \mathtt{S}'_0, \overline{\mathtt{S}}') = \overline{\mathtt{U}}, \mathtt{S}_0, \overline{\mathtt{S}} \\ \text{for any } \mathtt{D} \in (\mathit{dom}(\overline{\mathtt{L}}) \cap \mathit{dom}(\overline{\mathtt{L}}')), \vdash \overline{\mathtt{L}}(\mathtt{D})\ \text{overrides}\ \overline{\mathtt{L}}'(\mathtt{D})\ \text{in}\ \mathtt{A@C}\end{array}}{\vdash \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}_1\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\ \}\ \text{overrides}\ \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{D}_2\ \{\ \overline{\mathtt{U}}\ \overline{\mathtt{g}};\ \overline{\mathtt{L}}'\ \overline{\mathtt{M}}'\}\ \text{in}\ \mathtt{A}}$$

$\boxed{\vdash \mathtt{L}\ \text{ok in}\ \mathtt{A}}$

$$\frac{\begin{array}{c}\vdash \mathtt{A@C}'\ \text{ok} \qquad \vdash \mathtt{A@C}' \mathrel{\texttt{<\#}} \mathtt{Object} \\ \mathtt{This} \mathrel{\texttt{<:}} \mathit{inexact}(\mathtt{A@C}) \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \vdash \overline{\mathtt{L}}\ \text{ok in}\ \mathtt{A@C} \qquad \vdash \overline{\mathtt{M}}\ \text{ok in}\ \mathtt{A@C} \\ \text{for any } \texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{U}}\texttt{>}\mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \mathtt{x})\{..\} \in \overline{\mathtt{M}}, \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{A@C}') = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{U}}'\texttt{>}\overline{\mathtt{S}}' \rightarrow \mathtt{S}_0{}', \text{then } [\overline{\mathtt{X}}/\overline{\mathtt{Y}}](\overline{\mathtt{U}}', \mathtt{S}_0{}', \overline{\mathtt{S}}') = \overline{\mathtt{U}}, \mathtt{S}_0, \overline{\mathtt{S}} \\ \mathit{classes}(\mathtt{A@C}') = \overline{\mathtt{L}}_s \\ \text{for any } \mathtt{D} \in (\mathit{dom}(\overline{\mathtt{L}}) \cap \mathit{dom}(\overline{\mathtt{L}}_s)), \vdash \overline{\mathtt{L}}(\mathtt{D})\ \text{overrides}\ \overline{\mathtt{L}}_s(\mathtt{D})\ \text{in}\ \mathtt{A@C}\end{array}}{\vdash \mathtt{class}\ \mathtt{C} \triangleleft \mathtt{C}'\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\ \}\ \text{ok in}\ \mathtt{A}} \tag{T-CLASS}$$

Figure 6. $\mathrm{FJ}_{\mathrm{path}}$: Typing Rules

classes in all superclasses of $\mathtt{A@C}$. Also, $\overline{\mathtt{L}}_s$ are extended by $\overline{\mathtt{L}}$ *implicitly*—the `extends` clauses of $\overline{\mathtt{L}}$ do not refer to $\overline{\mathtt{L}}_s$; this is why valid method overriding is checked in $\vdash \mathtt{L}$ overrides $\mathtt{L}'$ in $\mathtt{A}$.

4.4 Operational Semantics

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

The reduction rules are given below. 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 $\mathtt{y}$ with $\mathtt{e}$. 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 $\longrightarrow$.

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

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{A}) = \texttt{<}\overline{\mathtt{X}}\texttt{>}(\overline{\mathtt{x}})\mathtt{e}_0}{\mathtt{new}\ \mathtt{A}(\overline{\mathtt{e}}).\texttt{<}\overline{\mathtt{E}}\texttt{>}\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{A}(\overline{\mathtt{e}})/\mathtt{this}][\overline{\mathtt{E}}/\overline{\mathtt{X}}, \mathtt{A}/\mathtt{This}]\mathtt{e}_0} \tag{R-INVK}$$

4.5 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 [29, 16]. For brevity, we omit the proofs, which will appear in a full version of the paper, which will be available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/`.

The set of values, mentioned in Theorem 2, are defined by: $\mathtt{v} ::= \mathtt{new\ A(\overline{v})}$, where $\overline{\mathtt{v}}$ can be empty.

THEOREM 1 (Subject Reduction). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$, *then* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{T}'$, *for some* $\mathtt{T}'$ *such that* $\emptyset \vdash \mathtt{T}' <: \mathtt{T}$.

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

5. Related Work

Nested Inheritance. The present work has emerged as an enhancement of language constructs for lightweight family polymorphism [17], with arbitrary levels of nesting, explicit inheritance between nested classes in the same group, and generalized relative path types with inexact qualification. The resulting language design is very close to Nystrom et al.'s JX language [23], though without exploiting dependent types(/classes).

JX supports an extension mechanism called nested inheritance that allows an inheritance hierarchy to be nested in another class and such a hierarchy to be inherited and extended by extending the enclosing class, just as our proposal. Indeed, it is very similar how class definitions are composed. Moreover, JX allows a class to extend another class outside the group.

Key ideas in their type system are dependent classes and prefix types. Dependent classes are type expressions of the form `x.class`, which means `x`'s run-time class. Using dependent classes, a method `equal` would take an argument of type `this.class`, which guarantees that the run-time classes of the receiver and the argument agree. Prefix types are usually used with dependent classes to express an enclosing class of a dependent class. For example, `Graph[n.class]` means `n.class`'s innermost enclosing class, which is a subclass of `Graph`. By combining the fact that inheritance *is* considered subtyping, they are useful when two arguments have to share the same enclosing class as in `connect_all()` as in Section 3. For example, here is its variant `make_loop()` written in JX

```
void make_loop(final Graph.Node n,
               Graph[n.class].Edge e) {
  n.src = n.dst = e;
}
```

JX's static type system guarantees that the actual arguments's run-time types share the same enclosing class, which must be a subclass of `Graph`. Since inheritance is subtyping, `CWGaph.Node` is a subtype of `Graph.Node` and so `make_loop()` can be invoked with `CWGraph.Node` and `CWGraph.Edge`. Since types now refer to expressions, the interaction with side-effects gets rather tricky; JX poses the restriction that `.class` can be preceded only by a sequence of zero or more accesses of final fields to final variables (including `this`) to avoid the meaning of the same dependent class expression changes at different program points. That's why `n` is (and must be) qualified with `final`. Although we do not formalize assignments in $\mathrm{FJ}_{\mathrm{path}}$, we expect thay can be easily and safely added with the usual typing rule.

Instead of dependent classes, we use type variables and `This` to achieve the separation of types and expressions for ease of typechecking. In particular, we observe that value arguments of JX also play the role of type arguments. It will be more apparent by comparing with the definition of `make_loop()` in our language:

```
<exact X extends Graph.Node>
  void make_loop(X n, ^X@Edge e) {.. }
```

Notice that `X` plays the role of `n.class` in the JX code. Following how `connect_all()` is written is Section 3, it can also be written

```
<exact X extends Graph>
  void make_loop(X@Node n, X@Edge e) {.. }
```

We believe that separating type variables gives more intuitive method signatures, especially when parametric types are involved; for example, if `connect_all()`, which takes arrays, is to be written in JX, the method definition seems to be something like:

```
void connect_all(final Graph g,
                 g.class.Edge[] es,
                 g.class.Node[] ns) {.. }
```

or

```
void connect_all(final Graph.Edge e,
                 e.class[] es,
                 Graph[e.class].Node[] ns) {.. }
```

which requires a *value parameter* `g` or `e`, which is *not* required by the method body.

One consequence of this design of JX seems that, as opposed to the common understanding, subtyping does *not* quite imply substitutability, which we think is not very intuitive: if an expression in a program is replaced with another, which is of a subtype of the original, the program can become ill-typed. For example, suppose class `C`, which has the subclass `D`, has method `equal()` that takes an argument of type `this.class`. Then, `c.equal(c)` would be well typed under the assumption that `c` has type `C`. Since `D` is a subtype of `C` in JX, one might expect that `d` of type `D` would be substitutable for `c` and so `d.equal(c)` would be also well typed but, in fact, it is not. In our type system, subtyping is substitutability thanks to the distinction between exact and inexact qualifications: `c.equal(c)` is allowed only when `c` is given an exact type `@C` and it can be replaced only by another expression of the same exact type.

More recently, Nystrom, Qi, and Myers [24] have extended JX to support the mechanism called nested intersection, which is similar to symmetric mixin composition in Scala [26, 25]. It would be interesting future work to add nested intersection to $\mathrm{FJ}_{\mathrm{path}}$.

Matching. A series of work [2, 7, 6, 4] by Bruce and his colleagues has been addressing statically safe type systems for languages with the notion of *MyType* (corresponding to `This` in this paper). As we have also discussed, even if one class extends another, the object type from the former is not always a subtype of that from the latter due to binary methods—methods whose argument types include *MyType*. Instead of subtyping, they introduce the matching relation on object types, which reflects the class hierarchy and plays an important role in typechecking binary methods. In the language called $\mathcal{LOOM}$ [6], the notion of *hash types* of the form `#T` is introduced; `#T` behaves as a common supertype[4] of all types that match `T` but binary methods cannot be invoked on it. Our inexact qualification can be considered a generalization of hash types in the context of nested classes. It may be worth noting that in some other languages of theirs [5, 3, 4], hash types are "default" (requiring no special symbols such as `#`) and objects types on which binary methods can be invoked are called exact types and written `@T`.

Also, they have introduced match-bounded polymorphic methods [7] to describe generic methods that work on different types that match the same interface. Polymorphic methods in this paper can be viewed as match-bounded polymorphic methods in disguise, since if an exact type `E` is a subtype of `T`, then `E` matches *exact*(`T`). Our choice is mainly for the sake of familiarity and uniformity with usual subtype-bounded polymorphic methods.

[4] Subtyping is not explicitly mentioned in their paper but there are typing rules to convert from one (exact) type to its hash version and from a hash type to another hash type which is matched by the former.

Later, the notion of *MyType* is extended from self-recursive object types to mutually recursive object types, resulting in the notion of *MyGroup* [5, 8, 3]. Here, mutually recursive classes are put in a group, which is extensible just as classes, and *MyGroup*, which changes its meaning along group extension, is used to express mutual references among classes. In this paper, groups and classes are unified into a single mechanism of classes, which can be arbitrarily nested. Accordingly, *MyType* and *MyGroup* are unified into a relative path type $\mathtt{This}^n$.

Concord [19] is another language that also has the notion of groups and *MyGroup*. A main difference from the present work is that Concord does not support nesting of groups but allows a class in a group to extend an absolute type, a class outside the enclosing group. It would be interesting future work to extend our language to allow a class to extend non-siblings.

Virtual Classes. Historically, virtual classes [20] (more precisely, virtual patterns) in Beta [21] have been very influential to much work on the design of languages that support scalable extensibility by using nesting structure of classes. The basic idea of virtual classes is to allow classes to be attributes of objects just as methods, by putting nested class definitions in another class and those nested classes to be inherited and further extended in a subclass. Although the original proposal was not statically type-safe, virtual classes are useful to describe not only generic data structures but also mutually recursive classes such as nodes and edges of graphs and their extensions.

Ernst, who coined the term "family polymorphism," improved Beta's static analysis in the development of the language `gbeta` to ensure the safety of the use of virtual classes as extensible mutually recursive classes [11] and also higher-order hierarchies [12], which refer to a mechanism that allows extensible class hierarchies just as in the example of `AST` in this paper.

Nested classes in `gbeta` are designed to be members (or attributes) of an object of their enclosing class as in Beta. So, in order to instantiate a nested class, an enclosing class has to be instantiated first and then a constructor of the nested class is invoked on the enclosing instance (that is, the instance of the enclosing class) as in inner classes of Java [15]. Unlike Java, however, objects from the same nested class with different enclosing instances are distinguished by the static analysis, making it possible to create many copies of the same group and prevent objects from different copies from being mixed. Scala [25, 26] and CaesarJ [22] adopt a similar mechanism of virtual classes. From the type system point of view, such a mechanism can be considered like dependent types [1]. In fact, a type is a path of (immutable) field accesses followed by a class name in the virtual class calculus [13], which models `gbeta`-style virtual classes described above. Since there is only a single kind of qualification for those path dependent types, it does not seem very easy to express types for, say, all sorts of expressions roughly corresponding to `AST.Expr`.

6. Concluding Remarks

We have proposed variant path types to support safe scalable extensibility. Relative path types, a natural extension of *MyType* by Bruce et al. in the context of nested classes, enable to describe inter-relationship among classes in the same group, preserved by extension of the enclosing class. Also, exact and inexact qualifications give flexible abstractions for various kinds of set of instances with a rich subtyping hierarchy. The type system has been formalized as an extension of Featherweight Java.

Other than proving subject reduction, main future work of this research concerns evaluating the applicability to a full-blown language such as Java. For example, it is interesting to investigate type inference for parametric methods, which we have already done to some degree in previous work [17]. Implementation issues are also left for future work but we believe that the techniques described in Nystrom et al. [23] can be applied to our proposal, as the semantics of inheritance of our language is similar (in fact, simpler).

Acknowledgments. We thank Vincent Siles for finding bugs in proofs and helping us prove a fixed lemma and anonymous reviewers for useful comments. 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 and Graint-in-Aid for Young Scientists (B) No. 18700026 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] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994. Preliminary version in POPL 1993, under the title "Safe type checking in a statically typed object-oriented programming language".

[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 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.

[5] 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.

[6] 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.

[7] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes on Computer Science*, pages 27–51, Aarhus, Denmark, August 1995. 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] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Proc. MFCS*, Springer LNCS, September 2006.

[10] Erik Ernst. Propagating class and method combination. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes on Computer Science*, pages 67–91, Lisboa, Portugal, June 1999. Springer Verlag.

[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, Budapest, Hungary, June 2001. Springer Verlag.

[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] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL2006)*, pages 270–282, Charleston, SC, January 2006.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.

[15] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, August 2002. A special issue with papers from the 7th International Workshop on Foundations of Object-Oriented Languages (FOOL7). An earlier version appeared in *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP2000)*, Springer LNCS 1850, pages 129–153, June, 2000.

[16] 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.

[17] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 161–177, Tsukuba, Japan, November 2005. Springer-Verlag.

[18] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In Boris Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP2002)*, volume 2374 of *Lecture Notes in Computer Science*, pages 441–469, Málaga, Spain, June 2002. Springer-Verlag.

[19] 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.

[20] 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.

[21] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, 1993.

[22] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings of International Conference on Aspect Oriented Software Development (AOSD'03)*, pages 90–99. ACM, 2003.

[23] 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.

[24] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 21–36, Portland, OR, October 2006.

[25] 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.

[26] Martin Odersky and Matthias Zenger. Scalable component abstraction. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, San Diego, CA, October 2005.

[27] 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.

[28] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proceedings of ACM Symposium on Applied Computing (SAC'04)*, pages 1289–1296, March 2004.

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