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,
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. Its typing issues has usually 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 solution, which can be more easily applied to Java-like languages, in which nested types are considered properties of classes.

We introduce the mechanism of *variant path types*, which provide a flexible means to express intra-group relationship (among classes) that has to be preserved through extension. In particular, improving and extending existing works on groups and exact types, we feature the new notions of *exact* and *inexact qualifications*, providing rich abstractions to express various kinds of set of objects, with a flexible subtyping scheme. We formalize a safe type system for variant path types on top of Featherweight Java. Our development results in a complete solution for scalable extensibility, similarly to previous attempts based on dependent type systems.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; Polymorphism; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Object-oriented constructs; Type structure

General Terms Design, Languages, Theory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010…$5.00

Keywords Scalable extensibility, subtyping, variance, variant path types

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 [12], higher-order structures [13], nested inheritance [25], and grouping mechanisms [3, 20, 30], all share this common goal, which we shall refer to as *scalable extensibility*, the term coined in the work by Nystrom et al. [25]. 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 seamlessly working for both single classes and groups of classes, *(iii)* a flexible subtyping relation for nested class-types, and *(iv)* a group-polymorphism mechanism.

It is very well known that scalable extensibility suffers from the covariance problem: in the standard framework of "inheritance is subtyping" of object-oriented languages, the mutual inter-relationships of classes in the same group cannot be safely preserved by group extension. Languages supporting scalable extensibility usually solve this problem by a rather expressive dependent type (or class) system, as in JX [25], Scala [28], or `gbeta` [13] (notable exceptions include Bruce's work [3] and Concord [20]). Although there are several studies on simple core calculi for languages with dependent types—such as for Scala and `gbeta` [27, 10, 14]—such languages are typically more complex than the standard Java setting and more difficult to manage. In particular, the fact that nested types are accessed through a restricted set of expressions raises subtle interactions with somewhat orthogonal aspects such as immutability of fields and variables—see Section 5 for a more detailed discussion. It is therefore interesting to study whether scalable extensibility can be achieved in a language without dependent types, for possibly easier application to mainstream Java-like languages, by starting from existing work on self-references (*MyType*), grouping mechanisms, and exact types [5, 3, 20, 29].

Our Contributions In [29], we started approaching this issue by seeking a minimal set of features for supporting features related to family polymorphism [12] in the context of scalable extensibility at only one level of nesting. This set includes an inheritance mechanism for group of classes decoupled from subtyping, a notion of relative path types to express mutual dependency preserved by inheritance, and family polymorphic methods through type variables over groups—thereby, dependent types are avoided.

In this paper, we advance this approach by supporting arbitrary levels of group hierarchies and by a new typing construct which we name *variant path types*[1], which achieve flexibility and expressiveness in both subtyping and accessibility of the hierarchy. Based on our previous work [29], this construct first extends the concept of *relative path types* to work in a deeply nested structure. Then, 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, such types feature 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 polymorphism at the group level (or binary methods in a broad sense) by restricting subtyping, inexact qualification recovers subtyping by preventing unsafe invocations of (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 (nested) 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 [32] in Java 5.0 [15]). 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:

- the introduction of the notion of variant path types for safe scalable extensibility; and
- formalization of a core language ˆFJ extending Featherweight Java [17] (FJ) with a sound type system of variant path types.

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

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

This paper is an extended version of [19].

Rest of This Paper Section 2 describes the basic framework of classes with arbitrary level of nesting and extension; 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 ˆFJ. 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, by considering a simplified setting without static types.

2.1 Grouping Classes by Nesting

As in previous approaches [12, 13, 28, 25], we see a class as both a mechanism to generate objects and one to group classes. Considering the “graph” example [12], which is described by a class definition like:

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

Here, 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`, instead of types, 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`, resp.) inside class `Graph` is assigned to variable `e` (`n`, resp.).

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

```
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 an overriding method 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 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 [11]. 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.

2.2 Extension inside Group

As discussed elsewhere [13, 25], 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 [25, 31], as in Figure 1.

The group class AST (which stands for abstract syntax trees) has classes Literal and Plus for abstract syntax tree nodes that extend another member Expr of the same class. 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

```
class AST{
  field root;
  class Expr extends Object{
    method toString(){ return ""; }
  }
  class Literal extends Expr {
    field val;
    method toString(){ return val; }
  }
  class Plus extends Expr {
    field op1, op2;
    method toString(){
      return this.op1.toString()+
               "+"+this.op2.toString();
    }
    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

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. [25], 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. In this way, bodies of superclasses can be given a linear order.

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 a number of interesting relationships among classes in a group to be expressed.

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 among 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 [12]. In other words, cross-group references 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* [29], 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 [25, 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 another member of its group. A general form $\hat{}\cdots\hat{}\mathtt{This}.\mathtt{C}_1.\mathtt{C}_2.\cdots.\mathtt{C}_n$ 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 $\mathtt{C}_1.\mathtt{C}_2.\ldots.\mathtt{C}_n$.[2]

In the previous graph example, the intra-group relationship between Edge and Node is expressed by using type ^This.Node, which means Graph.Node in the class of Graph.Edge and CWGraph.Node in the class of CWGraph.Edge, and ^This.Edge. Figure 2 shows a complete graph example written in our language. Here, nodes hold a reference to an 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 also clarifies the need to disallow cross-group

[2] An operator similar to ^ is often introduced as a special form of qualification .out [14, 9] and, in Tribe [9], .out can appear anywhere in a path type. We allow ^ to apply only to This or type variables (introduced later) since in our setting—where nested classes are properties of classes, rather than objects as in Tribe—symbol ^ in the middle of a type expression will simply cancel a preceding qualification.

```
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 = colorToWeight(s.color, d.color);
      super.connect(s, d);
  } }
}
```

Figure 2. Graph and CWGraph Classes

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 $\mathtt{C}_1.\cdots.\mathtt{C}_n$—which we call *absolute path types*, since they denote a certain class independently of the location 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 types) 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 type safety 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 two Graph.Nodes with a CWGraph.Edge, the call to connect() causes the attempt to access field color on 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 exact types 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.[3] 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(..);
```

[3] 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*.

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 for 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 finally .AST.Expr (which was written AST.Expr) with the usual meaning. We call "." *inexact qualification* and "@" *exact qualification*. Here, @ at the head can be considered an exact qualification over the top level, or a 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, on the other hand, 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 "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 some member of 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();
    }
```

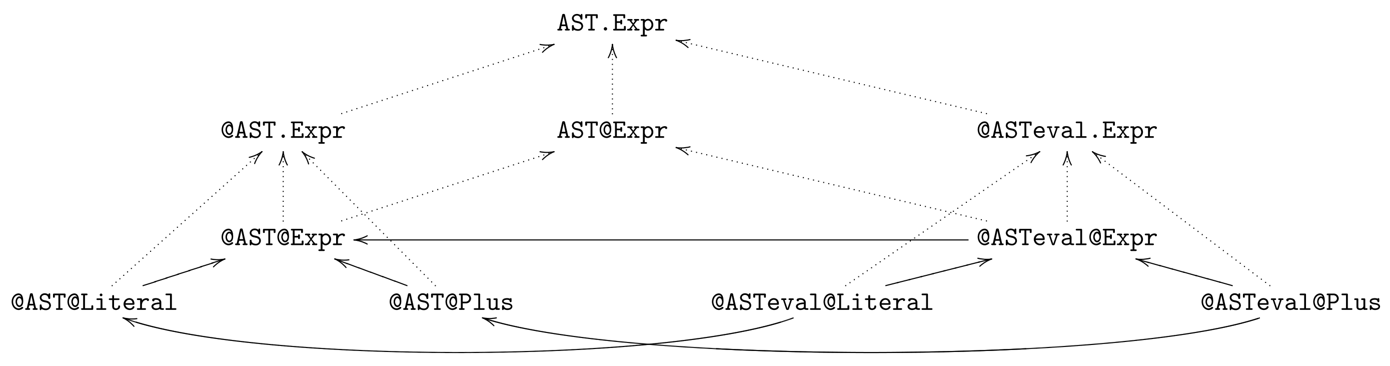

Figure 3. The rich subtyping hierarchy for the expression example. Dotted arrows represent subtyping while solid arrows represent inheritance, which is *not* subtyping.

```
      void replaceOp1(^This.Expr e) {
         this.op1 = e; return;
      }
    }
  }
```

Type `^This.Expr` is used to denote the two operands of a `Plus` expression. An instance of `@AST@Plus` can contain any `@AST.Expr`, that is, instances of type `@AST@Plus`, or `@AST@Literal`, as operands, whereas an instance of `@ASTeval@Plus` can contain any `@ASTeval.Expr`. Therefore, types `This`, `^This`, `^^This`, and so on, are exact types.

3.4 Inexact Qualification and Access Restriction

Since an inexact type is a common supertype of many exact types, it is clear from the substitutability principle that it should provide a more restricted access to its methods and fields than any of those exact types, and in particular, its exact version. For example, it is easy to see that `@Graph@Edge` provides `connect()`, which takes only `@Graph@Node`s (not `@CWGraph@Node`), whereas `Graph.Edge` does not.

Actually, even a partially inexact type can allow access to methods taking relative path types. For example, `connect()` can be safely invoked on `@Graph.Edge` since the argument type `^This.Node` of `connect()` only requires arguments to belong to the same graph—it does not require `Node`s to be exact (due to inexact qualification). 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.Node` is replaced with exact `@Graph`, a prefix of `@Graph.Edge`.[4]

[4] One might want to use `^This@Edge` and `^This@Node` rather than `^This.Edge` and `^This.Node` in the graph example in Figure 2. 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()`, then invoking it on `s` or `d` inside `connect()` have to be prohibited because the type of `s` or `d` is (partially) inexact. To invoke `equal()`, their types would have to be `^This@Node`, which is fully exact. See the discussion below, too.

To clarify the differences between `^This.C` and `This`, we describe an example of `equal()` for checking (syntactic) equality of abstract syntax trees. It also demonstrates usefullness of partially inexact types.

A natural design choice would be to add this method to class `AST.Expr` with signature `boolean equal(This e)`, and implement it, for example, in class `Plus` as follows:

```
class AST {
  ...
  class Plus extends Expr{
    ^This.Expr op1, op2;
    ...
    boolean equal(This e){
      return this.op1.equal(e.op1) &&
             this.op2.equal(e.op2);
    }
  }
}
```

Unfortunately, this implementation does not typecheck: since `equal()` takes a relative path type `This`, which requires the receiver type to be fully exact, it cannot be invoked on `this.op1` of type `^This.Expr`. In fact, it *should not* typecheck—it can happen that the (run-time) type of receiver `this.op1` and argument `e.op1` are not the same. It is also weird that `equal()` takes `This`, since it will only amount to allowing comparison of two ASTs whose root nodes are of the same kind. Thus, a correct version would take `^This.Expr` instead of `This` to enable comparison between trees with arbitrary kinds of roots. However, simply changing `This` to `^This.Expr` would not work—`e.op1` is disallowed this time.

To make it work, we have to simulate multi-dispatching based on both receiver and argument types (a similar style of programming for Scala is shown by Zenger and Odersky [34]). A solution to this problem with variant path types is shown in Figure 4. Each class is equipped with auxiliary methods `eqLit()` and `eqPlus()`, specialized to different kinds of AST nodes. In class `Expr` these methods provide a default behavior to return `false`. Actual code of com-

```
class AST {
  class Expr{
    ...
    // default implementations
    boolean equal(^This.Expr e){ return false;}
    boolean eqLit(^This.Lit e){ return false; }
    boolean eqPlus(^This.Plus e){ return false; }
  }
  class Lit extends Expr {
    int i;
    ...
    boolean equal(^This.Expr b) {
      return b.eqLit(this);
    };
    boolean eqLit(^This.Lit e) {
      return e.i == this.i;
    }
  }
  class Plus extends Expr {
    ^This.Expr op1, op2;
    ...
    boolean equal(^This.Expr b) {
      return b.eqPlus(this);
    };
    boolean eqPlus(^This.Plus e) {
      return this.op1.equal(e.op1) &&
             this.op2.equal(e.op2);
    }
  }
}
```

Figure 4. Implementation of the method `equal()`.

parison is coded in specialized versions, `Lit.eqLit()` and `Plus.eqPlus()`, where the argument has the same type as the class in which they are defined. If the argument and the class are different kinds, the answer should be `false`, inherited from `Expr`. Finally, and more interestingly, in `equal()` in `Lit` and `Plus`, the exact kind of the root of one of the nodes (namely `this`) is revealed, so comparison is delegated to specialized versions by swapping the argument and the receiver. Note that `this` of type `This` is passed to a method taking `^This.Lit` or `^This.Plus`; in general, it is safe to allow subtyping between `This` and `^This.C` in nested class `D` that extends `C`. This implementation correctly typechecks and works as expected. Moreover, it is easy to add another kind of expressions (other than `Lit` and `Plus`) by extending `AST`.

3.5 Parametric Methods for Group-Polymorphic Methods

One of the central ideas in family polymorphism [12] 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 [29], 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 ˆFJ based on Featherweight Java [17]. 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 ˆFJ, a nested class can extend either `Object`, which is an empty class, or another class in the same group, though some other languages [20, 25] 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.

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

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 "$\mathtt{\overline{T}\ \overline{f};}$" for "$\mathtt{T_1\ f_1;\ldots;\ T_n\ f_n;}$", where n is the length of $\overline{\mathtt{T}}$ and $\overline{\mathtt{f}}$, and "$\mathtt{this.\overline{f}=\overline{f};}$" as shorthand for "$\mathtt{this.f_1=f_1;\ldots;this.f_n=f_n;}$" and so on. Sequences of field declarations, parameter names, method definitions, nested class definitions are assumed to contain no duplicate names. So, we sometimes view a sequence as a mapping: for example, $\overline{\mathtt{L}}(\mathtt{C})$ denotes a class $\mathtt{L}_i$ of name `C` and similarly for $\overline{\mathtt{M}}$. 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, whereas types starting with $\mathtt{X}^n$, which corresponds to `^ ··· ^X` (with `^` n times) in the previous section, are called relative path types. $\mathtt{X}^0$ is abbreviated to `X` by omitting the superscript. Here, we extend the prefixing operation from `This` to all type variables. Also note that for notational convenience we use absolute path types such as `@C@D`, instead of the common notation `C.D` used in the last section, for `new` expressions and names of classes. A qualification of the form `@C` is called exact whereas `.C` is called inexact. In particular, a type without any inexact qualification is called an exact type, ranged over by `E` as shown above.

As in FJ, `Object` is a special class name, whose definition does not appear in the class table. Moreover, in ^FJ, qualifications `@Object` and `.Object` are allowed everywhere even though `@A@Object` is not defined in the class table. Allowing such qualifications makes the definitions of lookup functions simple: `@A@Object` is simply assumed to have no members. We include `/` (read "top-level") without any qualification also 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 $\lhd$ is read "`extends`." A method declaration can be parameterized by exact type variables $\overline{\mathtt{X}}$. 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 of 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) $\mathit{CT}(\mathtt{A@C}) = \mathtt{class\ C\ \cdots}$ for every $\mathtt{A@C} \in \mathit{dom}(\mathit{CT})$; (2) if $\mathit{CT}(\mathtt{A@C})$ has a nested class declaration `L` of name `D`, then $\mathit{CT}(\mathtt{A@C@D}) = \mathtt{L}$; and (3) $\mathtt{A@Object} \notin \mathit{dom}(\mathit{CT})$ for any $\mathtt{A} \in \mathit{dom}(\mathit{CT})$. A program is a pair $(\mathit{CT}, \mathtt{e})$ 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 and Lookup Functions

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 $\mathit{classes}(\mathtt{A})$ to list up nested classes inside `A` after hierarchical composition of `A`. It requires the following auxiliary operator $\mathtt{L_1}\Leftarrow\mathtt{L_2}$ to compose a superclass $\mathtt{L_1}$ with a subclass $\mathtt{L_2}$:

$$\begin{array}{c}
\mathtt{class\ C} \lhd \mathtt{D\{\overline{T}\ \overline{f};\ \overline{L}_1\ \overline{M}_1\}}\\
\Leftarrow\mathtt{class\ C} \lhd \mathtt{D\{\overline{U}\ \overline{g};\ \overline{L}_2\ \overline{M}_2\}}\\
= \mathtt{class\ C} \lhd \mathtt{D\ \{\overline{T}\ \overline{f};\ \overline{U}\ \overline{g};\ (\overline{L}_1\Leftarrow\overline{L}_2)\ \ (\overline{M}_1\Leftarrow\overline{M}_2)\}}
\end{array}$$

Here, $\overline{\mathtt{L}}_1\Leftarrow\overline{\mathtt{L}}_2$ 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 $\Leftarrow$. Similarly, $\overline{\mathtt{M}}_1\Leftarrow\overline{\mathtt{M}}_2$ 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 $\overline{\mathtt{M}}_1$, since $\overline{\mathtt{M}}_2$ are overriding definitions. Their straightforward definitions are omitted here for brevity. In the definition of $\mathtt{L_1}\Leftarrow\mathtt{L_2}$, the `extends` clauses have to match in order to preserve inheritance structure of nested classes.

Actually, we define a more general function $\mathit{classes}(\mathtt{A}, i)$ instead of $\mathit{classes}(\mathtt{A})$, which is considered $\mathit{classes}(\mathtt{A}, n)$ where n is the length of `A` (that is, the number of simple class names in `A`). The auxiliary argument i, which is a natural number, controls how deep hierarchical composition is performed to list up nested classes: for example, $\mathit{classes}(\mathtt{@C@D}, 0)$ lists up only the nested classes that appears in $\mathit{CT}(\mathtt{@C@D})$ without taking inheritance into account at all; $\mathit{classes}(\mathtt{@C@D}, 1)$ will compose the top-level class `C` with its superclasses and returns nested classes appearing exactly in D. In most cases, we use the specialized version *classes*(A); the significance of the auxiliary argument will be clarified in typing of classes.

The definition of *classes*$(\mathtt{A}, i)$ appears at the top of Figure 5. The first rule says that A@Object has no nested classes and the second that / is the top-level. The third rule, which deals with full composition, means that nested classes in A@C are obtained by composing nested classes in C in *classes*$(\mathtt{A}, i)$ 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. The fourth rule, on the other hand, means that, when i is less than the length of A@C, nested classes in A@D are ignored.

For example, consider the following ˆFJ 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, 1) 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.

At this point, we can check that there are no cycles in the inheritance relation at all levels. First, cycles at the top level can be easily detected; if there is no cycle, then *classes*(@C) is defined for any $\mathtt{@C} \in \mathit{dom}(\mathit{CT})$. Second, the absence of cycles in *classes*(@C) can be checked for each C, ensuring well-definedness of *classes*(@C@D) for any $\mathtt{D} \in \mathit{dom}(\mathit{classes}(\mathtt{@C}))$. We can repeat this procedure until the maximum level of nesting is reached.

One may wonder if cycles can be detected earlier when a class table is given or later as part of typechecking. First of all, cycles can be detected only when hierarchical composition is taken into account: for example, class C2 in the classes below

```
class C1 {
  class D1 extends D2 {}
}
class C2 extends C1 {
  class D2 extends D1 {}
}
```

contains a cycle of nested classes D1 and D2 but it cannot be detected unless C1 and C2 are composed. Of course, C1 is already an ill-formed class since D2, specified as the superclass of D1, is missing. However, cycles should be detected before typechecking—the process in which we find C1 is ill formed—because typechecking uses lookup functions, which works only when there are no cycles. Thus, cycles should be detected now. In what follows, we assume there are no cycles in the given class table.

Thanks to *classes*$(\mathtt{A}, i)$, it is now easy to define functions to look up fields and methods from a given class name. The definitions of field/method lookup functions are also in Figure 5. Function *fields*$(\mathtt{A}, i)$, which is similar to *classes*$(\mathtt{A}, i)$, enumerates all field names of A (and its superclasses) with their types. Similarly, *mtype*$(\mathtt{m}, \mathtt{A}, i)$ 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, and @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 $\Delta \vdash \mathtt{E}_1$ <# $\mathtt{E}_2$, one for subtyping $\Delta \vdash \mathtt{S}$ <: $\mathtt{T}$, one for type well-formedness $\Delta \vdash \mathtt{T}$ ok, and one for typing $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$. Here, Δ, called *bound environment*, is a finite mapping written $\overline{\mathtt{X}}$<:$\overline{\mathtt{T}}$ 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 $\overline{\mathtt{x}}$:$\overline{\mathtt{T}}$ 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 [17], 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 $\Delta \vdash \overline{\mathtt{S}}$ <: $\overline{\mathtt{T}}$ (similarly for type equivalence and matching); $\Delta \vdash \mathtt{T}_1$ ok, $\ldots$, $\Delta \vdash \mathtt{T}_n$ ok to $\Delta \vdash \overline{\mathtt{T}}$ ok; and $\Delta; \Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Delta; \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

4.3.1 Auxiliary Definitions

We first define a few auxiliary operations used in typing rules. $\mathtt{T}^n$ denotes a type obtained by dropping the last n

$\boxed{\mathit{classes}(\mathtt{A}, i)}$

$$\mathit{classes}(\mathtt{A@Object}, i) = \bullet \qquad \frac{(\overline{\mathtt{L}} \text{ are all top-level classes})}{\mathit{classes}(\mathtt{/}, i) = \overline{\mathtt{L}}}$$

$$\frac{i \geq |\mathtt{A@C}| \qquad \mathtt{class\ C \lhd D\ \{ \cdots \overline{L} \cdots \}} \in \mathit{classes}(\mathtt{A}, i) \qquad \mathit{classes}(\mathtt{A@D}, i) = \overline{\mathtt{L}}'}{\mathit{classes}(\mathtt{A@C}, i) = \overline{\mathtt{L}}' \Leftarrow \overline{\mathtt{L}}}$$

$$\frac{i < |\mathtt{A@C}| \qquad \mathtt{class\ C \lhd D\ \{ \cdots \overline{L} \cdots \}} \in \mathit{classes}(\mathtt{A}, i)}{\mathit{classes}(\mathtt{A@C}, i) = \overline{\mathtt{L}}}$$

$\boxed{\mathit{fields}(\mathtt{A}, i)}$

$$\mathit{fields}(\mathtt{A@Object}, i) = \bullet$$

$$\frac{\mathtt{class\ C \lhd D\ \{\overline{T}\ \overline{f};\ \cdots \}} \in \mathit{classes}(\mathtt{A}, i) \qquad \mathit{fields}(\mathtt{A@D}, i) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{A@C}, i) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

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

$$\frac{i < |\mathtt{A@C}| \qquad \mathtt{class\ C \lhd D\ \{ \cdots \overline{M}\ \}} \in \mathit{classes}(\mathtt{A}, i) \qquad \mathtt{<\overline{X} \lhd \overline{U}>S_0\ m(\overline{S}\ \overline{x})\{\ \cdots\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{A@C}, i) = \mathtt{<\overline{X} \lhd \overline{U}>}\overline{\mathtt{S}} \rightarrow \mathtt{S_0}}$$

$$\frac{n \geq |\mathtt{A@C}| \qquad \mathtt{class\ C \lhd D\ \{ \cdots \overline{M}\ \}} \in \mathit{classes}(\mathtt{A}, i) \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{A@D}, i) = \mathtt{<\overline{X} \lhd \overline{U}>}\overline{\mathtt{S}} \rightarrow \mathtt{S_0}}{\mathit{mtype}(\mathtt{m}, \mathtt{A@C}, i) = \mathtt{<\overline{X} \lhd \overline{U}>}\overline{\mathtt{S}} \rightarrow \mathtt{S_0}}$$

Figure 5. ˆFJ: Lookup Functions

qualifications from T; it is defined by:

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

Note that $(\cdot)^n$ is an operation on types whereas $\mathtt{X}^n$ is just a syntactic entity.

By using the prefixing operation, (simultaneous) type substitution $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$ of types for type variables is defined as follows:

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

Note that $\mathtt{X}^n$ is replaced with the corresponding prefix of T. $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}$ is defined straightforwardly.

An *exact type substitution* $[\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}]$, which requires an exact type when $\mathtt{X}^n$ is replaced, is similarly defined below:

$$\begin{array}{lcll}
[\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}]\mathtt{/} & = & \mathtt{/} & \\
[\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}]\mathtt{X}_i{}^n & = & \mathtt{T}_i{}^n & (\text{if } \mathtt{T}^n \text{ is exact}) \\
[\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}]\mathtt{Y}^n & = & \mathtt{Y}^n & (\text{if } \mathtt{Y} \notin \overline{\mathtt{X}}) \\
[\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}](\mathtt{S@C}) & = & ([\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}]\mathtt{S})\mathtt{@C} & \\
[\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}](\mathtt{S.C}) & = & ([\overline{\mathtt{T}}/\mathtt{@}\overline{\mathtt{X}}]\mathtt{S})\mathtt{.C} &
\end{array}$$

Notice that the argument T may contain inexact qualifications: for example, $[\mathtt{@C.D}_1/\mathtt{@X}]\mathtt{X\hat{}1@D}_2 = \mathtt{@C@D}_2$ (whereas $[\mathtt{C.D}_1/\mathtt{@X}]\mathtt{X\hat{}1@D}_2$ is undefined).

exact(T) (*inexact*(T), resp.) denotes a type in which all inexact (exact, resp.) qualifications in T are replaced by exact (inexact, resp.) ones. They are defined by:

$$\begin{array}{lclcl}
\mathit{exact}(\mathtt{/}) & = & \mathit{inexact}(\mathtt{/}) & = & \mathtt{/} \\
\mathit{exact}(\mathtt{X}^n) & = & \mathit{inexact}(\mathtt{X}^n) & = & \mathtt{X}^n \\
\mathit{exact}(\mathtt{T@C}) & = & \mathit{exact}(\mathtt{T.C}) & = & \mathit{exact}(\mathtt{T})\mathtt{@C} \\
\mathit{inexact}(\mathtt{T@C}) & = & \mathit{inexact}(\mathtt{T.C}) & = & \mathit{inexact}(\mathtt{T})\mathtt{.C}
\end{array}$$

4.3.2 Type Equivalence

The judgment $\Delta \vdash \mathtt{S} \equiv \mathtt{T}$ can be read "type S is equivalent to T under Δ." The rules are shown in Figure 6. 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 in fact equivalent. The fifth rule means that if $\mathtt{X}^n$ is equivalent to an exact type `E@C`, then its enclosing class $\mathtt{X}^{n+1}$ is to `E` and hence $\mathtt{X}^n$ and $\mathtt{X}^{n+1}\mathtt{@C}$ are equivalent: for example,

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

can be derived.

4.3.3 Matching

The subtyping relation will be defined by using the inheritance relation, which is formalized as matching here. The judgment $\Delta \vdash \mathtt{E}_1 \ \mathtt{<\#} \ \mathtt{E}_2$ can be read "exact type $\mathtt{E}_1$ matches $\mathtt{E}_2$" or simply "$\mathtt{E}_1$ extends $\mathtt{E}_2$." The rules are shown in Figure 6. The matching relation is a partial order including type equivalence and `@Object` as the top element, as seen in the first three rules. The fourth 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 fifth 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 last rule deals with `extends` clauses.

4.3.4 Subtyping

The judgment form for subtyping $\Delta \vdash \mathtt{S} \ \mathtt{<:} \ \mathtt{T}$ can be read "`S` is subtype of `T` under Δ." Subtyping rules are shown in Figure 6. As usual, subtyping is reflexive and transitive with `Object` as the top type and a type variable (with some prefixing) is a subtype of (the corresponding prefix of) its declared upper bound. The sixth rule intuitively means that exactness can be forgotten. The third 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@Plus` is a subtype of `AST@Plus`, which includes `Plus` from both `AST` and `ASTeval`. The last rule roughly means that inexact types are related if one inherits the other—it is parallel to the last rule of matching.

4.3.5 Type Well-formedness

The judgment form for well formed types is $\Delta \vdash \mathtt{T}$ ok, read as "`T` is well formed under Δ." The type well-formedness rules are also in Figure 6. 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 implicitly exist, due to nested inheritance, hence the function *classes* is used in the last two rules. Note that `A@Object` or `A.Object` is always well formed if `A` is well formed.

4.3.6 Typing

Typing for expressions. 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 7; readers who are familiar with languages with matching [2], in particular LOOJ [4], will notice some similarities.

The key rules are T-FIELD for field access and T-INVK for method invocation. The rule T-FIELD means that the type of field access $\mathtt{e}_0\mathtt{.f}_i$ 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 *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

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

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.

In T-INVK, 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 $\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{S}_0$. 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[5]. For example, assume

$$\begin{aligned}&\mathit{mtype}(\mathtt{@Graph@Edge}, \mathtt{connect})\\ &= (\mathtt{This}^1\mathtt{@Node}, \mathtt{This}^1\mathtt{@Node}) \rightarrow \ \mathtt{void}\ .\end{aligned}$$

Then

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

should be derived but not

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

In order to express this condition, we use exact type substitution $[\mathtt{T}/\mathtt{@X}]$ defined before. In this example,

$$[\mathtt{Graph.Edge}/\mathtt{@This}]\mathtt{This}^1\mathtt{@Node}$$

is not well defined, making the second judgment above not 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,

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

is derivable.

[5] This requirement is essentially the same as *exactness preservation* [26].

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

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

$$\frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Delta \qquad \mathtt{T}^n \text{ is exact}}{\Delta \vdash \mathtt{X}^n \equiv \mathtt{T}^n} \qquad \frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Delta \qquad \mathtt{T}^n = \mathtt{S@C}}{\Delta \vdash \mathtt{X}^n \equiv \mathtt{X}^{n+1}\mathtt{@C}}$$

$$\frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}{\Delta \vdash \mathtt{S@C} \equiv \mathtt{T@C}} \qquad \frac{\Delta \vdash \mathtt{S} \equiv \mathtt{T}}{\Delta \vdash \mathtt{S.C} \equiv \mathtt{T.C}}$$

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

$$\frac{\Delta \vdash \mathtt{E}_1 \equiv \mathtt{E}_2}{\Delta \vdash \mathtt{E}_1 \mathtt{ <\# } \mathtt{E}_2} \qquad \frac{\Delta \vdash \mathtt{E}_1 \mathtt{ <\# } \mathtt{E}_2 \qquad \Delta \vdash \mathtt{E}_2 \mathtt{ <\# } \mathtt{E}_3}{\Delta \vdash \mathtt{E}_1 \mathtt{ <\# } \mathtt{E}_3} \qquad \Delta \vdash \mathtt{E <\# @Object}$$

$$\frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X}^n \mathtt{ <\# } \mathit{exact}(\mathtt{T}^n)} \qquad \frac{\mathtt{X}\mathtt{<:}\mathtt{U} \in \Delta \qquad \mathit{exact}(\mathtt{U}^n) = \mathtt{E@C}}{\Delta \vdash \mathtt{X}^n \mathtt{ <\# } \mathtt{X}^{n+1}\mathtt{@C}}$$

$$\frac{\Delta \vdash \mathtt{E}_1 \mathtt{ <\# } \mathtt{E}_2}{\Delta \vdash \mathtt{E}_1\mathtt{@C <\# } \mathtt{E}_2\mathtt{@C}} \qquad \frac{\Delta \vdash \mathtt{E <\# A} \qquad \mathtt{class\ C} \lhd \mathtt{D}\ \{\cdots\} \in \mathit{classes}(\mathtt{A})}{\Delta \vdash \mathtt{E@C <\# E@D}}$$

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

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

$$\frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X}^n \mathtt{ <: } \mathtt{T}^n} \qquad \frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Delta \qquad \mathtt{T}^n = \mathtt{S.C}}{\Delta \vdash \mathtt{X}^n \mathtt{ <: } \mathtt{X}^{n+1}\mathtt{.C}}$$

$$\Delta \vdash \mathtt{T@C <: T.C} \qquad \frac{\Delta \vdash \mathtt{S <: T}}{\Delta \vdash \mathtt{S@C <: T@C}} \qquad \frac{\Delta \vdash \mathtt{S <: T}}{\Delta \vdash \mathtt{S.C <: T.C}}$$

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

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

$$\Delta \vdash \mathtt{@Object} \text{ ok} \qquad \Delta \vdash \mathtt{Object} \text{ ok} \qquad \frac{\mathtt{X}\mathtt{<:}\mathtt{T} \in \Delta \qquad \Delta \vdash \mathtt{T}^n \text{ ok}}{\Delta \vdash \mathtt{X}^n \text{ ok}}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\cdots\} \in \mathit{classes}(\mathtt{/})}{\Delta \vdash \mathtt{@C} \text{ ok}} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\cdots\} \in \mathit{classes}(\mathtt{/})}{\Delta \vdash \mathtt{C} \text{ ok}}$$

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

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

Figure 6. ˆFJ: Rules for type equivalence, matching, subtyping, and type well-formedness

Typing for methods. The judgment for well-formed 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 *inexact*(`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 (modulo renaming of type parameters) with the overridden, but, unlike FJ, this condition will be checked by T-CLASS.

Typing for classes. 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; and (2) methods are correctly overriding. The second line means

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l}\mathtt{This}{<:}\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_{n-1} \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \vdash \overline{\mathtt{L}}\ \text{ok in}\ \mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_n \qquad \vdash \overline{\mathtt{M}}\ \text{ok in}\ \mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_n \\ \quad \vdash \mathtt{class\ D}\lhd\mathtt{D}'\mathtt{\{\}}\ \text{ok in}\ \mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_n \\ \quad \text{for any}\ \mathtt{D}\ \text{such that}\ \mathtt{class\ D}\lhd\mathtt{D}'\mathtt{\{}\cdots\mathtt{\}} \in \mathit{classes}(\mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_n)\ \text{and}\ \mathtt{D} \notin \mathit{dom}(\overline{\mathtt{L}}), \\ \qquad \left(\begin{array}{l}\text{for any}\ \mathtt{m},\ i \in \{1,\ldots,|\mathtt{A@C}|\}, \\ \text{if}\ \mathit{mtype}(\mathtt{m},\mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_n,i-1) = \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}{\rightarrow}\mathtt{S}_0\ \text{and} \\ \quad \mathtt{class\ C}_i\lhd\mathtt{C}_i{}'\mathtt{\{..\}} \in \mathit{classes}(\mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_{i-1},i-1)\ \text{and} \\ \quad \mathit{mtype}(\mathtt{m},\mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}'_i\mathtt{@}\cdots\mathtt{@C}_n,i) = \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{U}}'\mathtt{>}\overline{\mathtt{S}}'{\rightarrow}\mathtt{S}_0{}', \\ \text{then}\ \overline{\mathtt{U}}',\mathtt{S}_0{}',\overline{\mathtt{S}}' = \overline{\mathtt{U}},\mathtt{S}_0,\overline{\mathtt{S}}\end{array}\right)\end{array}}{\vdash \mathtt{class\ C}_n \lhd \mathtt{C}_n{}'\ \mathtt{\{}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{L}}\ \overline{\mathtt{M}}\ \mathtt{\}}\ \text{ok in}\ \mathtt{@C}_1\mathtt{@}\cdots\mathtt{@C}_{n-1}} \qquad \text{(T-CLASS)}$$

Figure 7. ˆFJ: Typing Rules

that a nested class D implicitly inherited inside $\mathtt{C}_n$ is equivalent to an explicit class with the empty body and it must be well-formed, too. This condition ensures the signatures of methods inherited from all superclasses of $\mathtt{@C}_1\cdots\mathtt{@C}_n\mathtt{@D}$ are identical.

The last big condition ensures correct method overriding, which is more involved to check than it may first appear, because one class may inherit definitions from multiple superclasses. For concreteness, consider a class $\mathtt{@C}_1\mathtt{@C}_2\mathtt{@C}_3$ to see what this condition means. When $i = 1$, it says

if $\mathit{mtype}(\mathtt{m},\mathtt{@C}_1\mathtt{@C}_2\mathtt{@C}_3,0) = \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}{\rightarrow}\mathtt{S}_0$ and $\mathtt{class\ C}_1\lhd\mathtt{C}_1{}'\mathtt{\{..\}} \in \mathit{classes}(\mathtt{/},0)$ and $\mathit{mtype}(\mathtt{m},\mathtt{@C}_1{}'\mathtt{@C}_2\mathtt{@C}_3,1) = \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{U}}'\mathtt{>}\overline{\mathtt{S}}'{\rightarrow}\mathtt{S}_0{}'$, then $\overline{\mathtt{U}}',\mathtt{S}_0{}',\overline{\mathtt{S}}' = \overline{\mathtt{U}},\mathtt{S}_0,\overline{\mathtt{S}}$.

It means that the signature of a method defined exactly in $\mathtt{@C}_1\mathtt{@C}_2\mathtt{@C}_3$ is the same as the one inherited from $\mathtt{@C}_1{}'\mathtt{@C}_2\mathtt{@C}_3$ (or $\mathtt{@C}_1{}''\mathtt{@C}_2\mathtt{@C}_3$, and so on). So, it amounts to checking consistency of the signatures of the methods in the class

$$\mathtt{@C}_1\mathtt{@C}_2\mathtt{@C}_3$$

against those defined (if any) in the lowest class in the chain of classes

$$\mathtt{@C}'_1\mathtt{@C}_2\mathtt{@C}_3 \mathrel{\mathtt{<\#}} \mathtt{@C}''_1\mathtt{@C}_2\mathtt{@C}_3 \mathrel{\mathtt{<\#}} \cdots$$

(where $\mathtt{@C}_1 \mathrel{\mathtt{<\#}} \mathtt{@C}_1{}' \mathrel{\mathtt{<\#}} \mathtt{@C}_1{}''$). When $i = 2$, the condition implies the consistency of the signatures of the methods in the classes that appear in the previous step

$$\mathtt{@C}_1\mathtt{@C}_2\mathtt{@C}_3 \mathrel{\mathtt{<\#}} \mathtt{@C}'_1\mathtt{@C}_2\mathtt{@C}_3 \mathrel{\mathtt{<\#}} \mathtt{@C}''_1\mathtt{@C}_2\mathtt{@C}_3 \mathrel{\mathtt{<\#}} \cdots$$

against those in the plane of classes

$$\begin{array}{ccccc}
\vdots & & \vdots & & \\
\overset{\#}{\vee} & & \overset{\#}{\vee} & & \\
\texttt{@C}_1\texttt{@C}''_2\texttt{@C}_3 & \texttt{<\#} & \texttt{@C}'_1\texttt{@C}''_2\texttt{@C}_3 & \texttt{<\#} & \cdots \\
\overset{\#}{\vee} & & \overset{\#}{\vee} & & \\
\texttt{@C}_1\texttt{@C}'_2\texttt{@C}_3 & \texttt{<\#} & \texttt{@C}'_1\texttt{@C}'_2\texttt{@C}_3 & \texttt{<\#} & \cdots
\end{array}$$

(where $\texttt{@C}_2$ <# $\texttt{@C}_2{}'$ <# $\texttt{@C}_2{}''$). Note that, in this step, methods inherited from different directions, that is, $\texttt{@C}'_1\texttt{@C}_2\texttt{@C}_3$ and $\texttt{@C}_1\texttt{@C}'_2\texttt{@C}_3$ are checked against each other, even if $\texttt{@C}_1\texttt{@C}_2\texttt{@C}_3$ does not have a method m. Finally, when $i = 3$, the merged plane, obtained by combining the chain and the plane above, is checked against the three dimension space, which covers $\texttt{@C}_1\texttt{@C}_2\texttt{@C}_3$'s all superclasses, which have not been covered in the previous steps.

Finally, a program (CT, e) is well formed if all (top-level) classes in *CT* are well formed and $\emptyset; \emptyset \vdash \texttt{e} : \texttt{T}$ for some T.

4.4 Operational Semantics

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

The reduction rules are given below. We write $[\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{e}/\texttt{y}]\texttt{e}_0$ for the expression obtained from $\texttt{e}_0$ by replacing $\texttt{x}_1$ with $\texttt{d}_1$, $\ldots$, $\texttt{x}_n$ with $\texttt{d}_n$, and y with e. There are two reduction rules, one for field access and one for method invocation, which are straightforward, thanks to lookup functions. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules(if $\texttt{e} \longrightarrow \texttt{e}'$ then $\texttt{e.f} \longrightarrow \texttt{e}'\texttt{.f}$, and the like), omitted here.

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

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

We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

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 [33, 17]. For brevity, we only sketch the proofs in Appendix; full proofs appear in an extended version of the paper, available at http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/.

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

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

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

THEOREM 3 (Type Soundness). *If* $\emptyset; \emptyset \vdash \texttt{e} : \texttt{T}$ *and* $\texttt{e} \longrightarrow \texttt{e}'$ *with* $\texttt{e}'$ *being a normal form, then* $\texttt{e}'$ *is a value* v *such that* $\emptyset; \emptyset \vdash \texttt{v} : \texttt{A}$, *for some* A *such that* $\emptyset \vdash \texttt{A} \texttt{<:} \texttt{T}$.

5. Related Work and Discussion

Nested Inheritance. The present work has emerged as an enhancement of language constructs for lightweight family polymorphism [29], 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 [25], 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 p.class, which means p's run-time class (here, p is a sequence of final field accesses on a final variable). 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. The notion called prefix types is 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 argument'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 must be taken into account; 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 to change at different program points. That's why `n` is (and must be) qualified with `final`. On the other hand, our language design is completely orthogonal to assignments, which are therefore not considered in ˆFJ calculus—we expect they 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 implies 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 [26] have extended JX to support the mechanism called nested intersection, which is similar to symmetric mixin composition in Scala [28, 27]. It would be interesting future work to add nested intersection to ˆFJ.

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[6] 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.

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 $\texttt{This}^n$.

Concord [20] 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 [21] (more precisely, virtual patterns) in Beta [22] 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

[6] 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.

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 [12] and also higher-order hierarchies [13], 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 [16]. 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. For example, one can implement hash tables by a class that has a virtual class implementing elements; then, elements from different instances of hash tables will not be mixed. Scala [27, 28] and CaesarJ [23] 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 [14], which models `gbeta`-style virtual classes described above.

On the one hand, these languages are more powerful than ours in the following points. First, as mentioned above, groups are finer grained and their number is unbounded since they are expressed by objects. Second, they can better deal with the situation where the identity of a group is abstracted out. For example, consider hash tables that are put into a data structure such as a list. Then, information on which hash tables are held by the list is lost in general. Nevertheless, it is still possible to extract an element from a hash table and put it back to the same hash table without exactly knowing a type of an element. In some sense, the type systems of these languages are equipped with some kind of existential types. On the other hand, in our language, once exact type information is lost, there is no way of recovering it. For example, it is not possible to invoke `replaceOp1()` on inexact `AST.Plus`. We expect this limitation can be lifted by introducing a mechanism similar to the unpacking operation in the context of existential types [24] or by a mechanism similar to wildcard capture [32].

On the other hand, our typing mechanism seems to have the advantage that it is easy to express, say, all sorts of expressions by inexact qualifications. Since there is only a single kind of qualification for those path dependent types, it does not seem very easy to express such a type.

More recently, Clarke et al. have proposed another virtual class calculus called Tribe [9], in which nested classes are members of an object of their enclosing class, too. Tribe generalizes types for existing languages of virtual classes by allowing both final field access `.f` and class access `.C`, which can appear in any order. For example, an expression `this.f` (where `f` is a final field) is also a singleton type, which denotes the value of `this.f`; `this.C` means *some* object of the class `C` nested in `this`; `C.f` refers to the object in field `f` of some object of `C` (or one of its subclasses); and `C.D` refers to *some* object of the class `D` nested in some object of `C` (or one of its subclasses).

Tribe types provide fine control over subtyping in a way similar to, but different from ours. While their qualification `.C` roughly corresponds to our inexact qualification, the qualification `.f` can be considered "very exact" qualification in the sense that it always denotes a single object, rather than objects of a single class as our exact qualification denotes.

We believe that these languages, in which nested classes are treated as members of objects, should benefit from our exact qualification, which provides an intermediate degree of exactness between very exact qualification by final field accesses and ordinary inexact qualifications. For example, exact types are useful to express standard binary methods such as equality, which—as far as we understand—does not seem very straightforward to express with Tribe types only.

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, and proved to be sound.

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 [29]. Moreover, it would be useful to study alternative syntactic sugar for variant path types, to support common programming patterns as in [25]. Implementation issues are also left for future work but we believe that the techniques described in Nystrom et al. [25] 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 No. 18200001 and Graint-in-Aid for Young Scientists (B) No. 18700026 from MEXT of Japan (Igarashi), and from the Italian PRIN 2006 Project "EOS DUE—Extensible Object Systems: Dynamic and Unpredictable Environments" (Viroli).

A. Proof Sketches

We sketch the proofs of Theorems 1 and 2. (Theorem 3 is their easy consequence.) The structure of the proof of subject reduction is similar to those for Featherweight Java and Featherweight GJ [17]. So, we first prove various substitution lemmas, which are all proved by induction on the derivations, including the following four:

LEMMA 1 (Type Substitution Preserves Subtyping). *If* $\mathtt{X}\mathrel{<:}\mathtt{U},\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ *and* $\emptyset \vdash \mathtt{U}$ *ok and* $\emptyset \vdash \mathtt{A} \mathrel{<:} \mathtt{U}$*, then* $[\mathtt{A}/\mathtt{X}]\Delta \vdash [\mathtt{A}/\mathtt{X}]\mathtt{S} \mathrel{<:} [\mathtt{A}/\mathtt{X}]\mathtt{T}$.

LEMMA 2 (Type Substitution Preserves Type Well-formedness). *If* $\mathtt{X}\mathrel{<:}\mathtt{U},\Delta \vdash \mathtt{T}$ *ok and* $\emptyset \vdash \mathtt{A},\mathtt{U}$ *ok and* $\emptyset \vdash \mathtt{A} \mathrel{<:} \mathtt{U}$*, then* $[\mathtt{A}/\mathtt{X}]\Delta \vdash [\mathtt{A}/\mathtt{X}]\mathtt{T}$ *ok.*

LEMMA 3 (Type Substitution Preserves Typing). *If* $\mathtt{X}\mathrel{<:}\mathtt{U},\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\emptyset \vdash \mathtt{U}$ *ok and* $\emptyset \vdash \mathtt{A} \mathrel{<:} \mathtt{U}$*, then there exists* $\mathtt{T}'$ *such that* $[\mathtt{A}/\mathtt{X}]\Delta;[\mathtt{A}/\mathtt{X}]\Gamma \vdash [\mathtt{A}/\mathtt{X}]\mathtt{e} : \mathtt{T}'$ *and* $[\mathtt{A}/\mathtt{X}]\Delta \vdash \mathtt{T}' \mathrel{<:} [\mathtt{A}/\mathtt{X}]\mathtt{T}$.

LEMMA 4 (Substitution Preserves Typing). *If* $\Delta;\Gamma,\overline{\mathtt{x}}{:}\overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}_0$ *and* $\Delta;\Gamma \vdash \overline{\mathtt{d}} : \overline{\mathtt{S}}$ *and* $\Delta \vdash \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}$*, then there exists* $\mathtt{T}_0{}'$ *such that* $\Delta;\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{T}_0{}'$ *and* $\Delta;\Gamma \vdash \mathtt{T}_0{}' \mathrel{<:} \mathtt{T}_0$.

In the proofs of the last two lemmas about typing, we also use lemmas stating that type substitution is covariant, i.e., if $\Delta \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{S}_2$, then $\Delta \vdash [\mathtt{S}_1/\mathtt{X}]\mathtt{T} \mathrel{<:} [\mathtt{S}_2/\mathtt{X}]\mathtt{T}$, and that exact type substitution is contravariant, i.e., If $\Delta \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{S}_2$ and $[\mathtt{S}_2/\mathtt{@X}]\mathtt{T}$ is well defined, then $\Delta \vdash [\mathtt{S}_2/\mathtt{@X}]\mathtt{T} \mathrel{<:} [\mathtt{S}_1/\mathtt{@X}]\mathtt{T}$.

Then, we prove properties of lookup functions: a field or method of some class is also present in any of its subclasses and, if a method type lookup succeeds, then a method body lookup also succeeds and the body is well typed.

LEMMA 5. *If* $\emptyset \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{A}'$ *and* $\mathtt{T}\ \mathtt{f} \in \mathit{fields}(\mathtt{A}')$*, then* $\mathtt{T}\ \mathtt{f} \in \mathit{fields}(\mathtt{A})$.

LEMMA 6. *If* $\emptyset \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{A}'$ *and* $\mathit{mtype}(\mathtt{A}') = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S}_0$*, then* $\mathit{mtype}(\mathtt{A}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S}_0$.

LEMMA 7. *If* $\mathit{mtype}(\mathtt{m},\mathtt{A}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T}_0$*, then there exist* $\overline{\mathtt{x}}$, $\mathtt{e}_0$, $\mathtt{B}$ *and* $\mathtt{T}_0$ *such that* $\mathit{mbody}(\mathtt{m},\mathtt{A}) = \mathtt{<}\overline{\mathtt{X}}\mathtt{>(}\overline{\mathtt{x}}\mathtt{)}\,\mathtt{e}_0$ *and* $\Delta \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{B}$ *and* $\mathtt{This}\mathrel{<:}\mathit{inexact}(\mathtt{B}),\overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{U}};\mathtt{this}:\mathtt{This},\overline{\mathtt{x}}{:}\overline{\mathtt{T}} \vdash \mathtt{e}_0 : \mathtt{S}_0$ *and* $\mathtt{This}\mathrel{<:}\mathit{inexact}(\mathtt{B}),\overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{U}} \vdash \mathtt{S}_0 \mathrel{<:} \mathtt{T}_0$.

A.1 Proof of Theorem 1

By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$ with case analysis on the last rule used. We show only main cases.

Case R-FIELD: $\quad \mathtt{e} = \mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{).f}_i \qquad \mathit{fields}(\mathtt{A}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\mathtt{e}' = \mathtt{e}_i$

By T-FIELD, and T-NEW, we have

$$\begin{array}{lll} \emptyset \vdash \mathtt{A}\ \mathrm{ok} & \emptyset;\emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} & \emptyset \vdash \overline{\mathtt{S}} \mathrel{<:} [\mathtt{A}/\mathtt{This}]\overline{\mathtt{T}} \\ \emptyset \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{A}' & \mathtt{U}\ \mathtt{f}_i \in \mathit{fields}(\mathtt{A}') & \mathtt{T} = [\mathtt{A}/\mathtt{This}]\mathtt{U} \end{array}$$

By Lemma 5, $\mathtt{U}\ \mathtt{f}_i \in \mathit{fields}(\mathtt{A})$ and $\mathtt{U} = \mathtt{T}_i$. Thus, $\emptyset;\emptyset \vdash \mathtt{e}_i : \mathtt{S}_i$ and $\emptyset \vdash \mathtt{S}_i \mathrel{<:} \mathtt{T}$, finishing the case.

Case R-INVK: $\quad \mathtt{e} = \mathtt{e}_r\mathtt{.<}\overline{\mathtt{E}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)}$
$\mathtt{e}_r = \mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{)}$
$\mathit{mbody}(\mathtt{m},\mathtt{A}) = \mathtt{<}\overline{\mathtt{X}}\mathtt{>(}\overline{\mathtt{x}}\mathtt{)}\,\mathtt{e}_0$
$\mathtt{e}' = [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{e}_r/\mathtt{this}][\overline{\mathtt{E}}/\overline{\mathtt{X}},\ \mathtt{A}/\mathtt{This}]\mathtt{e}_0$

By T-INVK and T-NEW, we have

$$\begin{array}{ll} \emptyset \vdash \mathtt{A}\ \mathrm{ok} & \emptyset;\emptyset \vdash \mathtt{new\ A(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{A} \\ \emptyset \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{A}' & \mathit{mtype}(\mathtt{m},\mathtt{A}') = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{S}_0 \\ \emptyset \vdash \overline{\mathtt{E}}\ \mathrm{ok} & \emptyset \vdash \overline{\mathtt{E}} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{@This}]\overline{\mathtt{U}} \\ \emptyset;\emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} & \emptyset \vdash \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{@This}]\overline{\mathtt{T}} \\ \mathtt{T} = [\overline{\mathtt{E}}/\overline{\mathtt{X}},\mathtt{A}/\mathtt{This}]\mathtt{S}_0\ . & \end{array}$$

By Lemma 6, $\mathit{mtype}(\mathtt{A}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{S}_0$. Then, by Lemma 7, there exist $\mathtt{B}$ and $\mathtt{U}_0$ such that

$$\begin{array}{l} \emptyset \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{B} \\ \mathtt{This}\mathrel{<:}\mathit{inexact}(\mathtt{B}),\overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{U}};\overline{\mathtt{x}}{:}\overline{\mathtt{S}},\mathtt{this}{:}\mathtt{This} \vdash \mathtt{e}_0 : \mathtt{U}_0 \\ \mathtt{This}\mathrel{<:}\mathit{inexact}(\mathtt{B}),\overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{U}} \vdash \mathtt{U}_0 \mathrel{<:} \mathtt{S}_0\ . \end{array}$$

We can prove that $\emptyset \vdash \mathtt{A} \mathrel{\texttt{<\#}} \mathtt{B}$ implies $\emptyset \vdash \mathtt{A} \mathrel{<:} \mathit{inexact}(\mathtt{B})$. Then, by Lemma 3, there exists $\mathtt{U}_0{}'$ such that

$$\begin{array}{l} \emptyset;\overline{\mathtt{x}} : [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{This}]\overline{\mathtt{S}},\mathtt{this} : \mathtt{A} \vdash [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{This}]\mathtt{e}_0 : \mathtt{U}_0{}' \\ \emptyset \vdash \mathtt{U}_0{}' \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{This}]\mathtt{U}_0\ . \end{array}$$

We also have

$$\emptyset \vdash [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{This}]\mathtt{U}_0 \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{A}/\mathtt{This}]\mathtt{S}_0$$

by Lemma 1. Finally, by Lemma 4, there exists $\mathtt{U}_0{}''$ such that

$$\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{U}_0{}'' \qquad \emptyset \vdash \mathtt{U}_0{}'' \mathrel{<:} \mathtt{U}_0{}'\ .$$

Finally, by S-TRANS, $\emptyset \vdash \mathtt{U}_0{}'' \mathrel{<:} \mathtt{T}$, finishing the case.

Case RC-INVK-RECV: $\quad \mathtt{e} = \mathtt{e}_0\mathtt{.<}\overline{\mathtt{E}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{>}$
$\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$
$\mathtt{e}' = \mathtt{e}_0{}'\mathtt{.<}\overline{\mathtt{E}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}$

By T-INVK, we have

$$\begin{array}{ll} \emptyset;\emptyset \vdash \mathtt{e}_0 : \mathtt{T}_0 & \\ \emptyset \vdash \mathit{exact}(\mathtt{T}_0) \mathrel{\texttt{<\#}} \mathtt{A}_0 & \mathit{mtype}(\mathtt{m},\mathtt{A}_0) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{S}_0 \\ \emptyset \vdash \overline{\mathtt{E}}\ \mathrm{ok} & \emptyset \vdash \overline{\mathtt{E}} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T}_0/\mathtt{@This}]\overline{\mathtt{U}} \\ \emptyset;\emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} & \emptyset \vdash \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T}_0/\mathtt{@This}]\overline{\mathtt{T}} \\ \mathtt{T} = [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T}_0/\mathtt{This}]\mathtt{S}_0 & \end{array}$$

By the induction hypothesis, there exists $\mathtt{T_0}'$ such that

$$\emptyset;\emptyset \vdash \mathtt{e_0}' : \mathtt{T_0}' \qquad \emptyset \vdash \mathtt{T_0}' \mathrel{<:} \mathtt{T_0}\ .$$

We have $\emptyset \vdash \mathit{exact}(\mathtt{T_0}') \mathrel{\texttt{<\#}} \mathtt{A_0}$. Since $[\mathtt{T_0}/\mathtt{@This}]\overline{\mathtt{U}}$ and $[\mathtt{T_0}/\mathtt{@This}]\overline{\mathtt{T}}$ are well defined, by contravariance of exact type substitution and Lemma 1,

$$\begin{aligned}&\emptyset \vdash [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}/\mathtt{@This}]\overline{\mathtt{U}} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}'/\mathtt{@This}]\overline{\mathtt{U}}\\ &\emptyset \vdash [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}/\mathtt{@This}]\overline{\mathtt{T}} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}'/\mathtt{@This}]\overline{\mathtt{T}}\ .\end{aligned}$$

By T-INVK,

$$\emptyset;\emptyset \vdash \mathtt{e_0}'.\mathtt{<}\overline{\mathtt{E}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}'/\mathtt{This}]\mathtt{S_0}\ .$$

Finally, by covariance of type substitution, we have

$$\emptyset \vdash [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}'/\mathtt{This}]\mathtt{S_0} \mathrel{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}][\mathtt{T_0}/\mathtt{This}]\mathtt{S_0}$$

finishing the case. □

A.2 Proof of Theorem 2

By induction on $\mathtt{e}$. We show only main cases.

Case: $\mathtt{e} = \mathtt{e_0.f}_i$

If $\mathtt{e_0}$ is not a value, by the induction hypothesis, $\mathtt{e_0} \longrightarrow \mathtt{e}'_0$ for some $\mathtt{e}'_0$; then, use RC-FIELD.

On the other hand, if $\mathtt{e_0}$ is a value, then, by T-FIELD, it must be of the form $\mathtt{new\ A_0(}\overline{\mathtt{v}}\mathtt{)}$ and $\mathtt{T\ f}_i \in \mathit{fields}(\mathtt{A_0}')$ for some $\mathtt{A_0}'$ such that $\emptyset \vdash \mathtt{A_0} \mathrel{\texttt{<\#}} \mathtt{A_0}'$. By Lemma 5 and T-NEW, $\mathit{fields}(\mathtt{A_0}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \in \mathtt{T\ f}_i$. Then, $\mathtt{e_0.f}_i \longrightarrow \mathtt{v}_i$.

Case: $\mathtt{e} = \mathtt{e_0.<}\overline{\mathtt{E}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}$

If $\mathtt{e}_i$ is not a value, by the induction hypothesis, $\mathtt{e}_i \longrightarrow \mathtt{e}'_i$ for some $\mathtt{e}'_i$; then, use RC-INVK-RECV (if $i = 0$) or RC-INVK-ARG (otherwise).

On the other hand, if $\mathtt{e_0}$ is a value, then by T-INVK, it must be of the form $\mathtt{new\ A_0(}\overline{\mathtt{v}}\mathtt{)}$ and $\mathit{mtype}(\mathtt{m},\mathtt{A_0}') = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S_0}$ for some $\mathtt{A_0}'$ such that $\emptyset;\emptyset \vdash \mathtt{A_0} \mathrel{\texttt{<\#}} \mathtt{A_0}'$. By Lemma 6, $\mathit{mtype}(\mathtt{m},\mathtt{A_0}) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{S}}\rightarrow\mathtt{S_0}$ and, by Lemma 7, $\mathit{mbody}(\mathtt{m},\mathtt{A_0}) = \mathtt{<}\overline{\mathtt{X}}\mathtt{>(}\overline{\mathtt{x}}\mathtt{)e}'$ where $|\overline{\mathtt{x}}| = |\overline{\mathtt{e}}|$. Thus, we have

$$\mathtt{e} \longrightarrow [\overline{\mathtt{e}}/\overline{\mathtt{x}}, \mathtt{new\ A_0(}\overline{\mathtt{v}}\mathtt{)/this}][\overline{\mathtt{E}}/\overline{\mathtt{X}}, \mathtt{A_0}/\mathtt{This}]\mathtt{e}'$$

finishing the case. □

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] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *Proceedings of International Conference on Aspect-Oriented Software Design (AOSD'07)*, pages 121–134, Vancouver, BC, March 2007.

[10] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Proceedings of International Symposium on Mathematical Foundations of Computer Science*, Springer LNCS, pages 1–23, September 2006.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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, October 1999.

[18] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, 2006.

[19] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Informal Proceedings of the International Workshop on Foundations and Development of Object-Oriented Languages (FOOL/WOOD 2007), Nice, France*, January 2007. Available at `http://foolwood07.cs.uchicago.edu/accepted.html`.

[20] 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.

[21] 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.

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

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

[24] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. of the 12th ACM POPL*, 1985.

[25] 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)*, pages 99–115, Vancouver, BC, October 2004.

[26] 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.

[27] 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.

[28] Martin Odersky and Matthias Zenger. Scalable component abstraction. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57, San Diego, CA, October 2005.

[29] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 2007. To appear. A preliminary summary appeared in *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS2005)*, Springer LNCS vol. 3780, pages 161–177, November, 2005.

[30] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *Proceedings of 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes on Computer Science*, pages 550–570, Brussels, Belgium, July 1998. Springer Verlag.

[31] 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.

[32] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11), December 2004. Special issue: OOPS track at SAC 2004, pp. 97–116.

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

[34] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, March 2004.