# A Virtual Class Calculus

Erik Ernst
University of Aarhus, Denmark
eernst@daimi.au.dk

Klaus Ostermann
Darmstadt Univ. of Technology, Germany
ostermann@informatik.tu-darmstadt.de

William R. Cook
University of Texas at Austin, USA
cook@cs.utexas.edu

## Abstract

*Virtual classes* are class-valued attributes of objects. Like virtual methods, virtual classes are defined in an object's class and may be redefined within subclasses. They resemble inner classes, which are also defined within a class, but virtual classes are accessed through object *instances*, not as static components of a class. When used as types, virtual classes depend upon object identity – each object instance introduces a new family of virtual class types. Virtual classes support large-scale program composition techniques, including higher-order hierarchies and family polymorphism. The original definition of virtual classes in BETA left open the question of static type safety, since some type errors were not caught until runtime. Later the languages Caesar and gbeta have used a more strict static analysis in order to ensure static type safety. However, the existence of a sound, statically typed model for virtual classes has been a long-standing open question. This paper presents a virtual class calculus, *vc*, that captures the essence of virtual classes in these full-fledged programming languages. The key contributions of the paper are a formalization of the dynamic and static semantics of *vc* and a proof of the soundness of *vc*.

***Categories and Subject Descriptors*** D.3.3 [*Language Constructs and Features*]: Classes and objects, inheritance, polymorphism; F.3.3 [*Studies of Program Constructs*]: Object-oriented constructs, type structure; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

***General Terms*** Languages, theory

***Keywords*** Virtual classes, soundness

## 1. Introduction

*Virtual classes* are class-valued attributes of objects. They are analogous to virtual *methods* in traditional object-oriented languages: they follow similar rules of definition, overriding and reference. In particular, virtual classes are defined within an object's class. They can be overridden and extended in subclasses, and they are accessed relative to an object instance, using late binding. This last characteristic is the key to virtual classes: it introduces a dependence between static types and dynamic instances, because dynamic instances contain classes that act as types. As a result, the actual, dynamic value of a virtual class is not known at compile time, but it is known to be a particular class which is accessible as a specific attribute of a given object, and some of its features may be statically known, whereas others are not.

When an object is passed as an argument to a method, the virtual classes within this argument are also accessible to the method. Hence, the method can declare variables and create instances using the virtual classes of its arguments. This enables the definition and use of higher-order hierarchies [9, 28], or hierarchies of classes that can manipulated, extended and passed as a unit. The formal parameter used to access such a hierarchy must be immutable; in general a virtual class only specifies a well-defined type when accessed via an immutable expression, which rules out dynamic references and anonymous values.

Virtual classes from different instances are not compatible. This distinction enables family polymorphism [8], in which families of types are defined that interact together but are distinguished from the classes of other instances. Virtual classes support arbitrary nesting and a form of mixin-based inheritance [3]. The root of a (possibly deeply) nested hierarchy can be extended with a set of nested classes which automatically extend the corresponding classes in the original root at all levels.

Virtual classes were introduced in the late seventies in the programming language BETA, but documented only several years later [21]. Methods and classes are unified as *patterns* in BETA. Virtual patterns were introduced to allow redefinition of methods. Since patterns also represent classes, it was natural to allow redefinition of classes, i.e. virtual classes. Later languages, including Caesar [22, 23] and gbeta [7, 8, 9] have extended the concept of virtual classes while remaining essentially consistent with the informally specified model in BETA [20]. For example, they have lifted restrictions in BETA that prevented virtual patterns (classes) from inheriting other virtual patterns (classes). So in this sense the design of virtual classes has only recently been fully developed.

Unfortunately, the BETA language definition and implementation allows some unsafe programs and inserts runtime checks to ensure type safety. Caesar and gbeta have stronger type systems and more well-defined semantics. However, their type systems have never been proven sound. This raises the important question of whether there exists a sound, type-safe model of virtual classes.

This paper provides an answer to this question by presenting a formal semantics and type system for virtual classes and demonstrating the soundness of the system. This calculus is at the core of the semantics of Caesar and gbeta and would presumably be at the core of every language supporting family polymorphism [8] and incremental specification of class hierarchies [9].

The calculus does not allow inheritance from classes located in other objects than **this**, and we use some global conditions to prevent name clashes. The significance of these restrictions and the techniques used to overcome them in the full-fledged languages are described in Section 5 and 8. The approach to static analysis taken in this paper was pioneered in BETA, made strict and complete in gbeta, and adapted and clarified as an extension to Java in Caesar.

The claim that virtual classes are inherently not type-safe should now be laid to rest.

The primary contributions of this paper are:

- Development of *vc*—a statically typed virtual class calculus, specified by a big-step semantics with assignment. The formal semantics supports the addition of virtual classes to mainstream object-oriented languages.
- Proof of the soundness of the type system. This paper includes the theorems, and the proofs are available in an accompanying technical report [10]. We use a proof technique that was developed for big-step semantics of object-oriented languages [6]. The preservation theorem ensures that an expression reduces to a value of the correct type, or a null pointer error, but never a dynamic type error. No results are proven about computations that do not terminate.
- We strengthen the traditional approach to soundness in big-step semantics by proving a *coverage* lemma, which ensures that the rules cover all cases, including error situations. This lemma plays a role analogous to the progress lemma for a small-step semantics [29]: it ensures that evaluation does not get stuck as a result of a missing case in the dynamic semantics.

## 2. Overview of Virtual Classes

Virtual classes are illustrated by a set of examples using an informal syntax in the style of Featherweight Java [17] or ClassicJava [12]. The distinguishing characteristics of *vc* include the following:

- Class definitions can be nested to define virtual classes.
- An instance of a nested class can refer to its *enclosing object* by the keyword **out**.
- Objects contain mutable *variables* and immutable *fields*. Fields are distinguished from variables by the keyword **field**. Fields must all be initialized by constructor arguments.
- A type is described by a *path* to an object and the name of a class in that object.
- The types of arguments and the return type of a method can use virtual classes from other arguments.

These concepts are illustrated in the examples given below. A formal syntax for *vc* is defined in Section 3. The main difference between the informal and formal syntax is that the formal syntax unifies classes and methods into a single construct, thus highlighting the syntactic and semantic unification of these concepts.

```
class Base { // contains two virtual classes
  class Exp {}
  class Lit extends Exp {
    int value; // a mutable variable
  }
  Lit zero;     // a mutable variable
  out.Exp TestLit() {
        out.Lit l;
        l = new out.Lit();
        l.value = 3;
        l;
  }
}
```

**Figure 1.** Defining virtual classes for expressions.

```
class WithNeg extends Base {
  class Neg extends Exp {
    Neg(out.Exp e) { this.e = e; }
    field out.Exp e;
  }
  out.Exp TestNeg() {
        new out.Neg(TestLit());
  }
}
```

**Figure 2.** Adding a class for negation expressions.

```
class WithEval extends Base {
  class Exp {
    int eval() { 0; }
  }
  class Lit {
    int eval() { value; }
  }
  int TestEval() {
        out.TestLit().eval();
  }
}
```

**Figure 3.** Adding an evaluation method on expressions.

```
class NegAndEval extends WithNeg, WithEval {
  class Neg {
    Neg(out.Exp e) { this.e = e; }
    int eval() { −e.eval(); }
  }
  int TestNegAndEval() {
        out.TestNeg().eval();
  }
}
```

**Figure 4.** Combining the negation class and evaluation method.

### 2.1 Higher-Order Hierarchies

Virtual classes provide an elegant solution to the *extensibility problem* [5, 19]: how to easily extend a data abstraction with both new representations and new operations. This problem is also known as the *expression problem* because a canonical example is the representation of the abstract syntax of expressions [36, 34, 38]. We present a solution to a simplified version of a standardized problem definition [15].

In Figure 1, the class Base contains two virtual classes: a general class Exp representing numeric expressions and subclass Lit representing numeric literals. All classes in *vc* are virtual classes and can be arbitrarily nested. Top-level classes are virtual by means of an implicit root class containing all top-level declarations. The method TestLit is explained below.

A *family* is a collection of virtual classes that depend upon each other. For example, the classes Exp and Lit are a family that exists within class Base. A family can be extended by subclassing the class in which it is defined. For example, Figure 2 extends the family to include a class Neg representing negation expressions.

Every virtual class has an *enclosing object*, to which the class can refer explicitly via the keyword **out**. In Figure 2, class Neg contains a *field* of type **out**.Exp. The type **out**.Exp is a reference to the class Exp in the enclosing instance of Neg. In general the type

**out**.A in class B denotes the sibling A of B. Because of subclassing and late binding, the dynamic value of **out** in Neg may be an instance of WithNeg or a subclass thereof. The **out** keyword can be repeated to access enclosing objects of the enclosing object.

The test functions in Figures 1 and 2 create a test instance of each class. The objects are created by accessing a virtual class (Lit or Neg) in the enclosing object. The return type of the methods is out.Exp rather than Exp because activation records are treated as separate objects whose enclosing object is the object containing the method, hence a property of the object containing the method must be accessed via **out**, whereas method parameters are accessed via **this**. A test can be run by invoking new WithNeg().TestNeg().

Redefinition of a virtual class occurs when it is declared and it is already defined in a superclass. In Figure 3, Exp and Lit are redefined to include an eval method; it is a redefinition because the family WithEval extends Base and they both define Exp and Lit. All superclasses in *vc* are *virtual superclasses* because redefinition of a class that is used as superclass affects its subclasses as well, so that the entire family is redefined.

The *static path* of a class definition is the lexical address of a class definition defined by the list of names of lexically enclosing class definitions. The static paths of the class definitions in Figure 3 are WithEval, WithEval.Exp and WithEval.Lit. Static paths never appear in programs, because virtual classes are always accessed through an object instance, not a class. However, they are useful for referring to specific class definitions.

Note that references to classes are "late bound" just like methods: when Base.TestLit is called from WithEval.TestEval the references to Lit are interpreted as WithEval.Lit, not Base.Lit.

A virtual class can have multiple superclasses, as in the definition of NegAndEval in Figure 4, which composes WithNeg and WithEval and adds the missing implementation of evaluation for negation expressions.

Hierarchies are not only first-class values, they can also be composed as a consequence of composing the enclosing class. The semantics of this composition is that nested virtual classes are composed, continuing recursively into nested classes. This phenomenon was introduced as *propagating combination* in [7] and later referred to as *deep mixin composition* [38]. This is achieved by combining the superclasses of the virtual class using *linearization*. For example, the class NegAndEval.Neg implicitly extends class WithNeg.Neg. Its also extends both Base.Exp and WithEval.Exp.

This behavior is a form of mixin-based inheritance [3] in that new class bodies are inserted into an existing inheritance hierarchy. For example, although WithNeg.Neg in Figure 2 has Exp as a declared superclass, after linearization it has WithEval.Exp as its immediate superclass.

## 2.2 Path-based Types

The example in Figure 5 illustrates path-based types and family polymorphism. The argument types in the previous examples have had the form C or **out**.C, where **out** can be repeated multiple times. Types can also be named via fields, which are immutable object instances that may contain virtual classes. The variable n defined at the bottom of Figure 5 has type f1.Exp, meaning that only instances of Exp whose enclosing object is identical to the value of f1 may be assigned to n. In general, a type consists of a path that specifies how to access an object, together with a class name. To ensure that this is well-defined, the path must only contain **out** and/or immutable fields, but not mutable variables. Hence, type compatibility depends on object identity, but types do not depend on values in any other way. More specifically, the type system makes sure that two types are only compatible if they are known to have identical enclosing objects.

```
class Test {
  int Test(out.WithNeg f1, out.NegAndEval f2) {
    this.f1 = f1; this.f2 = f2;
    n = buildNeg(f1, n); // OK
    // n.eval(); -- Static error
    f2.zero = new f2.Lit(); // OK
    // n2 = buildNeg(f2, f1.zero) -- Static error
    n2 = buildNeg(f2, f2.zero); // OK
    n2.eval(); // OK
  }
  ne.Neg buildNeg(out.out.WithNeg ne, ne.Exp ex) {
    new ne.Neg(ex);
  }
  field out.WithNeg f1
  field out.NegAndEval f2
  f1.Exp n
  f2.Exp n2
}
new Test(new NegAndEval(), new NegAndEval())
```

**Figure 5.** Example of family polymorphism

Although the resulting types may resemble Java package/class names, they are very different because objects play the role of packages, and the class that creates a package can be subclassed.

## 2.3 Family Polymorphism

A *family object* is an object that provides access to a class family. A family object may be the enclosing object for an expression, but it may also be a method argument or the value of a field. As a provider of classes, and hence types, it enables type parameterization of classes and methods. But virtual classes are different from parameterized types: while type parameters are bound statically at compile-time, virtual classes are bound dynamically at runtime. Thus virtual classes enable a new kind of subtype polymorphism known as family polymorphism [8].

Family objects can also be used to create new objects, even though the classes in the family object are not known at compile time. To achieve the same effect in a main-stream language like Java, a factory method [13] must be used. However, the typing relation between related classes is then lost, whereas a family object testifies to the interrelatedness of its nested family classes.

In Figure 5, f1 and f2 inside Test are used as family objects. The constructor call in the last line of the example shows how f1 is polymorphically initialized with a subtype of its static types. The field f1 of class Test is declared to be an out.WithNeg, but the constructor is called with an argument of type NegAndEval, which illustrates that entire class hierarchies are first class values, subject to subtype polymorphism via their family objects, and the nested family classes are usable for both typing and object creation.

The assignments and calls in the body of the Test constructor illustrate the expressiveness of the type system. For example, although the buildNeg method is not aware of the eval method introduced by WithEval, it is possible to assign the result to n2 and call eval on the returned value. This is an important special case of family polymorphism where the types of arguments or the return type of a method depend on other arguments. The example also shows a few cases that are rejected by the type checker because they would potentially lead to a type error at runtime.

## 3. Syntax

The formal syntax of *vc* has been designed to make the presentation of the semantics as simple as possible, hence the formal syntax

**Grammar of *vc***

$$
\begin{array}{lcl}
\mathsf{CL} & ::= & \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\ \mathsf{K}\ \overline{\mathsf{CL}};\ \overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \overline{\mathsf{T}}\ \overline{\mathsf{v}}\ \} \\
\mathsf{K} & ::= & \mathsf{T}\ \mathsf{C}(\overline{\mathsf{T}}\ \overline{\mathsf{f}})\ \{\ \mathsf{e};\ \} \\
\mathsf{T} & ::= & \mathsf{path.C} \\
\mathsf{path} & ::= & \mathsf{spine}.\overline{\mathsf{f}} \\
\mathsf{spine} & ::= & \textbf{this}.\overline{\textbf{out}} \\
\mathsf{e} & ::= & \textbf{null} \mid \mathsf{e}\,;\mathsf{e} \mid \mathsf{path} \mid \mathsf{path.v} \mid \\
 & & \mathsf{path.v} = \mathsf{e} \mid \textbf{new}\ \mathsf{path.C}(\overline{\mathsf{e}})
\end{array}
$$

**Identifiers**

$$
\begin{array}{lccc}
\text{class names} & \mathsf{C} & & \\
\text{field names} & \mathsf{f} & & \\
\text{variable names} & \mathsf{v} & & \\
\text{members} & \mathsf{m} & = & \mathsf{f} \cup \mathsf{v}
\end{array}
$$

($\mathsf{C}$, $\mathsf{f}$, and $\mathsf{v}$ are pairwise disjoint)

**Figure 6.** Syntax of virtual class calculus *vc*

deviates from the informal syntax used in the examples in a few points that will be described in this section.

### 3.1 Notational Conventions

Our formal definitions use a number of syntactic conventions. A bar above a metavariable denotes a list: $\overline{\mathsf{p}}$ stands for $\mathsf{p}_1, ..., \mathsf{p}_k$ for some natural number $k \geq 0$. If $k = 0$ then the list is empty. The length of $\overline{\mathsf{p}}$ is $|\overline{\mathsf{p}}|$. The same notation is used for lists whose elements are separated by dots or commas, e.g., $\mathsf{f}_1.\mathsf{f}_2.\cdots.\mathsf{f}_k = \overline{\mathsf{f}}$. A list may also be represented by a combination of barred and unbarred variables: $\overline{\mathsf{f}}.\mathsf{f}$ stands for $\mathsf{f}_1.\cdots.\mathsf{f}_k.\mathsf{f}$, where $\mathsf{f}$ denotes the last item of the list. Following common convention, $\overline{\mathsf{T}}\ \overline{\mathsf{f}}$ represents a list of pairs $\mathsf{T}_1\ \mathsf{f}_1 \cdots \mathsf{T}_k\ \mathsf{f}_k$ rather than a pair of lists. An empty list is written $\mathsf{nil}_\mathsf{x}$, where $\mathsf{x}$ identifies the kind of items that the list should contain. The subscript $\mathsf{x}$ may be omitted if it is clear from context. The notation $[\mathsf{f}]$ represents a list with a single element $\mathsf{f}$. Finally, in function definitions with overlapping branches the first matching case is used.

### 3.2 Formal Syntax of *vc*

The formal syntax of *vc* is defined in Figure 6. A class definition $\mathsf{CL}$ consist of a name, the superclass names $\overline{\mathsf{C}}$, a constructor $\mathsf{K}$, a list of nested class definitions $\overline{\mathsf{CL}}$, declarations $\overline{\mathsf{T}}\ \overline{\mathsf{f}}$ of immutable fields, and declarations $\overline{\mathsf{T}}\ \overline{\mathsf{v}}$ of mutable variables. A constructor $\mathsf{K}$ consists of a return type $\mathsf{T}$, the class name, the formal parameters $\overline{\mathsf{T}}\ \overline{\mathsf{f}}$, and an expression $\mathsf{e}$. The constructor has a return type because it can return other things than the new object, which enables the encoding of methods as classes.

The keyword **field** from the informal syntax is not needed, because field and variable names are separate in the formal syntax and use different metavariables—$\mathsf{f}$ for fields and $\mathsf{v}$ for variables. Field and variable names must be unique within the program in order to simplify the handling of name clashes in connection with class composition. Class names are unique in that two definitions of the same class name must have a common superclass. We will later discuss the implications and possible relaxations of these restrictions. Note, however, that any program in which the names are reused can always be rewritten to a program with unique names.

Expressions include standard forms for the current object or any of the enclosing objects via $\mathsf{spine}$, access to fields of the current or an enclosing object via $\mathsf{path}$, access and assignment of variables, $\mathsf{path.v}$, and $\mathsf{path}\ .\mathsf{v} = \mathsf{e}$, and the null value, **null**. Method calls and object construction are unified in the expression **new** $\mathsf{path.C}(\overline{\mathsf{e}})$.

Types in the syntax of *vc* have the form $\mathsf{path.C}$. A $\mathsf{path}$ has the form $\textbf{this}.\overline{\textbf{out}}.\overline{\mathsf{f}}$. Thus a type allows a class $\mathsf{C}$ to be identified by navigating to any enclosing object and then traversing fields to find the object which contains $\mathsf{C}$.

**Metavariable**

$$
\text{static paths} \quad \mathsf{p} \quad ::= \quad \overline{\mathsf{C}}
$$

**Class table**

$$
\mathit{CT}(\mathsf{p}) = \mathit{CT2}(\mathsf{p}, \overline{\mathsf{CL}}_{root})
$$

$$
\frac{\mathsf{CL}_i = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\ ...\ \}}{\mathit{CT2}(\mathsf{C}, \overline{\mathsf{CL}}) = \mathsf{CL}_i}
$$

$$
\frac{\mathsf{CL}_i = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\ \mathsf{K}\ \overline{\mathsf{CL}}';\ ...\ \}}{\mathit{CT2}(\mathsf{C.p}, \overline{\mathsf{CL}}) = \mathit{CT}(\mathsf{p}, \overline{\mathsf{CL}}')}
$$

**All members**

$$
\mathcal{M}\!\mathit{embers}(\mathsf{nil}_\mathsf{p}) = \mathsf{nil}_{\mathsf{T}\,\mathsf{f}}, \mathsf{nil}_{\mathsf{T}\,\mathsf{v}}
$$

$$
\frac{\begin{array}{c}\mathcal{M}\!\mathit{embers}(\overline{\mathsf{p}}) = \overline{\mathsf{T}}\ \overline{\mathsf{f}}, \overline{\mathsf{T}}'\ \overline{\mathsf{v}} \\ \mathit{CT}(\mathsf{p}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\ \mathsf{K}\ \overline{\mathsf{CL}};\ \overline{\mathsf{T}}''\ \overline{\mathsf{f}}';\ \overline{\mathsf{T}}'''\ \overline{\mathsf{v}}'\}\end{array}}{\mathcal{M}\!\mathit{embers}(\mathsf{p}\ \overline{\mathsf{p}}) = \overline{\mathsf{T}}''\ \overline{\mathsf{f}}'\ \overline{\mathsf{T}}\ \overline{\mathsf{f}}, \overline{\mathsf{T}}'''\ \overline{\mathsf{v}}'\ \overline{\mathsf{T}}'\ \overline{\mathsf{v}}}
$$

**Constructor**

$$
\frac{\mathit{CT}(\mathsf{p}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\ \mathsf{K}\ \overline{\mathsf{CL}};\ \overline{\mathsf{T}}''\ \overline{\mathsf{f}}';\ \overline{\mathsf{T}}'''\ \overline{\mathsf{v}}'\}}{\mathcal{C}\!\mathit{onstr}(\mathsf{p}) = \mathsf{K}}
$$

**Figure 7.** Auxiliary definitions

Primitive types like $\mathsf{bool}$ and $\mathsf{int}$ are omitted; they just add complexity to the formalism without adding value. A member $\mathsf{m}$ is either a field or a variable.

### 3.3 Translating Informal Notation to *vc*

The translation of the informal language to the formal syntax of *vc* is straightforward. The most significant difference is that *vc* unifies methods and classes into a single definition construct. This technique originated in Simula, where classes were simply functions that returned the current activation record. In *vc* activation records are first-class values that are accessed by **this**. Thus a class is simply a definition that returns **this**, while a method is a definition that returns any other value.

Hence, method definitions in the informal language correspond to class declarations in *vc*, where the constructor represents the method body. More formally, the translation is as follows:
$\mathsf{T}\ \mathsf{C}(\overline{\mathsf{T}}\ \overline{\mathsf{f}})\ \{\ \overline{\mathsf{T}}\ \overline{\mathsf{v}};\ \underline{\mathsf{e}};\ \} \ \Rightarrow\ \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \{\ \mathsf{K}\ \mathsf{nil}_{\mathsf{CL}};\ \overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \overline{\mathsf{T}}\ \overline{\mathsf{v}}\ \}$
where $\mathsf{K} = \mathsf{T}\ \mathsf{C}(\overline{\mathsf{T}}\ \overline{\mathsf{f}})\ \{\ \mathsf{e};\ \}$. Method calls are translated by prefixing them with the keyword **new**.

As in Java, constructors in the informal syntax do not specify a return type or return value, but these must be specified in *vc*. For a class definition $\mathsf{C}$ in the informal syntax, the constructor return type is always $\textbf{out}.\mathsf{C}$ and the returned value is always **this**.

In the informal syntax a class definition with no superclasses may omit the **extends** clause. In the formal syntax it must be present, but the list of superclasses can be empty. The assignments of the constructor arguments is omitted in the formal syntax; instead, the name of the constructor arguments are matched against the field names. Constructors are required in *vc*, while the informal syntax assumes a default constructor if none is given.

The informal notation omits **this** when followed by **out** or a field. *vc* has no implicit scoping rules, and all access to fields, variables, and classes must be disambiguated by a $\mathsf{spine}$.

$$\begin{array}{l}
\iota_{\text{root}} \mapsto [\![ \ \bot \ \| \ \mathsf{C}_{\text{root}} \ \| \quad ]\!] \\
\iota_1 \mapsto [\![ \ \iota_{\text{root}} \ \| \ \mathsf{NegAndEval} \ \| \ \mathsf{zero} : \textbf{null} \ ]\!] \\
\iota_2 \mapsto [\![ \ \iota_{\text{root}} \ \| \ \mathsf{NegAndEval} \ \| \ \mathsf{zero} : \iota_5 \ ]\!] \\
\iota_3 \mapsto [\![ \ \iota_{\text{root}} \ \| \ \mathsf{Test} \ \| \ \mathsf{f1} : \iota_1 \ \mathsf{f2} : \iota_2 \ \mathsf{n} : \iota_4 \ \mathsf{n2} : \iota_6 \ ]\!] \\
\iota_4 \mapsto [\![ \ \iota_1 \ \| \ \mathsf{Neg} \ \| \ \mathsf{e} : \textbf{null} \ ]\!] \\
\iota_5 \mapsto [\![ \ \iota_2 \ \| \ \mathsf{Lit} \ \| \ \mathsf{value} : 0 \ ]\!] \\
\iota_6 \mapsto [\![ \ \iota_2 \ \| \ \mathsf{Neg} \ \| \ \mathsf{e} : \iota_5 \ ]\!]
\end{array}$$

**Figure 9.** Dynamic Heap after executing the example in Figure 5

The informal language allows more general expressions where the calculus only allows paths: $\mathsf{e.m}$, **new** $\mathsf{e.C}(\bar{\mathsf{e}})$, and $\mathsf{e.v} = \mathsf{e}'$. The general forms are translated into the calculus by rewriting $\mathsf{e.m}$ as **new this**.$\mathsf{C}'(\mathsf{e})$ where $\mathsf{C}'$ is a new local class with a field $\mathsf{T}$ $\mathsf{f}$ where $\mathsf{T}$ is the type of $\mathsf{e}$, and whose constructor returns **this**.$\mathsf{f.m}$. The translation is legal because the member is accessed through the new field. The other two constructs (**new** $\mathsf{e.C}(\bar{\mathsf{e}})$, and $\mathsf{e.v} = \mathsf{e}'$) are handled similarly. The consequence of this is that the formal treatment need not take types inside temporary objects into account. This is a significant simplification, and handling types in temporaries does not produce useful extra insight.

### 3.4 Auxiliary Definitions

Figure 7 gives some auxiliary definitions. A *static path* $\mathsf{p}$ is a list of class names $\overline{\mathsf{C}}$. The function *CT* looks up a class definition. We assume the existence of a globally available program in the form of a list of top-level class declarations $\overline{\mathsf{CL}}_{root}$, which would otherwise embellish many relations and functions. *CT* is a partial function from static paths to class definitions. It uses the helper function *CT2*, which recursively enters each class definition named in the path starting from root. For example, the static path $\mathsf{Base.Lit}$ denotes the definition of $\mathsf{Lit}$ inside $\mathsf{Base}$ in Figure 1.

A static path that identifies a valid class is called a *mixin*. The set of mixins in a program is equivalent to the static paths $\mathsf{p}$ for which $CT(\mathsf{p}) \neq \bot$. Since there is a one-to-one correspondence between a mixin (a static path) and its class definition, we also use the term mixin to refer to the body of the corresponding class, i.e., the part of a class declaration between the curly brackets `{ ... }`.

The function $\mathcal{M}embers$ collects all field and variable declarations found in a list of mixins $\bar{\mathsf{p}}$. The function $\mathcal{C}onstr(\mathsf{p})$ returns the constructor of $CT(\mathsf{p})$ given a static path $\mathsf{p}$.

## 4. Operational Semantics

The operational semantics is defined in big-step style. The semantic domains, evaluation relation, and helper functions are given in Figure 8. Both the operational semantics and the type system have also been implemented in Haskell.

### 4.1 Objects and the Heap

As in most object-oriented languages, an object in *vc* combines state and behavior. An **Object** is a tuple containing a pointer to its enclosing object $\iota$, a class name $\mathsf{C}$, and a list of fields and variables with their values.

The fields and variables are the state of the object; fields are immutable while variables can be updated. The heap is standard: a map $\mathsf{H}$ from addresses $\iota$ to objects. The top-level root object has the special address $\iota_{\text{root}}$. An example heap is given in Figure 9.

The features of the object are determined by the enclosing object $\iota$ and the class $\mathsf{C}$. The enclosing object specifies the environment containing the class from which the object $\iota'$ was created: an object $\iota'$ with enclosing object $\iota$ and class $\mathsf{C}$ must have been created by evaluating an expression equivalent to **new** $\iota.\mathsf{C}(...)$.

An object's features are defined by a list of mixins, or class bodies; these class bodies contain the declarations of members and nested classes. In *vc* there are no methods, but classes may be used as methods. The list of mixins of an object is computed from the class name and the mixins of the enclosing object.

Note that the definition of **Object** is optimized for a situation where all path expressions associated with an object should be understood relative to the same environment—the same enclosing object. It would be a relevant extension of *vc* to allow inheritance from classes inside other objects than **this** (i.e., to allow superclasses on the form $\mathsf{path.C}$), but it would then be necessary to maintain an environment for each mixin or for each feature. It is possible to do this, and for instance the static analysis and run-time support for gbeta maintains a separate enclosing object for each mixin. This causes a non-trivial amount of extra complexity, even though the basic ideas are unchanged. It is part of future work to extend *vc* correspondingly.

### 4.2 Mixin Computation

The $\mathcal{M}ix$ function computes the behavior, or mixin list, of an object $\iota$ in the heap $\mathsf{H}$. It does so by first computing the mixins of the enclosing object. All definitions of $\mathsf{C}$ and its superclasses are assembled into this mixin list. The mixin list of the root object has only a single element, namely the empty static path.

The $\mathcal{A}ssemble$ function[1] computes the mixin list for a class $\mathsf{C}$ relative to an enclosing mixin list $\bar{\mathsf{p}}$. It calls $\mathcal{D}efs$ to collect all the definitions of $\mathsf{C}$ located in any of the class bodies specified by $\bar{\mathsf{p}}$. If the resulting list of mixins is empty then the class is not defined and $\mathcal{A}ssemble$ returns $\bot$. Otherwise, the result is a list of static paths that identifies all definitions of $\mathsf{C}$ contained in the list of enclosing mixins.

As an example, let us consider the computation of $\mathcal{M}ix(\mathsf{H}, \iota_4)$ in the program in Figure 1-4 and the sample heap in Figure 9. Assume that the mixin list $\bar{\mathsf{p}}$ of the enclosing object $\iota_1$ has been computed to yield $[\mathsf{Base}, \mathsf{WithNeg}, \mathsf{WithEval}, \mathsf{NegAndEval}]$. Then $\mathcal{D}efs(\bar{\mathsf{p}}, \mathsf{Neg}) = [\mathsf{WithNeg.Neg}, \mathsf{NegAndEval.Neg}]$.

The complete mixin list must also include the mixins of all the superclasses. To do so, $\mathcal{A}ssemble$ maps $\mathcal{E}xpand$ over the list of static paths that was computed with $\mathcal{D}efs$, and linearizes the result. $\mathcal{E}xpand$ assembles each of the superclasses of $\mathsf{C}$, linearizes the result, and appends the class itself to the resulting list. In our example $[\,\mathcal{E}xpand(\bar{\mathsf{p}}, \mathsf{p}) \mid \mathsf{p} \leftarrow \mathsf{WithNeg.Neg}\ \mathsf{NegAndEval.Neg}\,] = \bar{\mathsf{p}}'\bar{\mathsf{p}}''$, where $\bar{\mathsf{p}}' = [\mathsf{Base.Exp}, \mathsf{WithEval.Exp}, \mathsf{WithNeg.Neg}]$ and $\bar{\mathsf{p}}'' = [\mathsf{NegAndEval.Neg}]$.

Linearization sorts an inheritance graph topologically, such that method calls are dispatched along the sort order. The function $\mathcal{L}inearize$ linearizes a list of mixin lists, i.e., it produces a single mixin list which contains the same mixins as those in the operands; the order of items in each of the input lists is preserved in the final result, to the degree possible. $\mathcal{L}inearize$ is defined in terms of a binary linearization function, $\mathcal{L}in2$. This function is an extension of the C3 linearization algorithm [1, 7] which has been used in gbeta and Caesar for several years. The linearization algorithm allows a programmer of a subclass to control the ordering of the class's mixins by choosing the order in which the superclasses appear in the **extends** clause.

$\mathcal{L}in2$ produces the same results as C3 linearization in every case where C3 linearization succeeds—this result follows trivially from the fact that the definition of C3 is just the four topmost cases in the definition of $\mathcal{L}in2$. The cases where C3 linearization fails are

[1] The $[\dots \mid \dots]$ notation used in the definition of $\mathcal{D}efs$, $\mathcal{A}ssemble$, and $\mathcal{E}xpand$ means list comprehension as for example in Haskell. Note that we append an element to a list by just writing the element to append after the list. For example, $[\,2n \mid n \leftarrow 1...5, n > 3\,]42$ is the list $[8, 10, 42]$.

**Objects and the Heap**:

$$\begin{array}{ll}
\textbf{Address} = \text{natural numbers} & \iota \\
\textbf{Object} = \{\llbracket\ \iota \parallel \mathsf{C} \parallel \overline{\mathsf{f}} : \overline{\mathsf{val}}\quad \overline{\mathsf{v}} : \overline{\mathsf{val}}'\ \rrbracket\} & \llbracket\ \dots\ \rrbracket \\
\textbf{Heap} = \textbf{Address} \xrightarrow[\text{fin}]{} \textbf{Object} & \mathsf{H} \\
\textbf{Value} = \textbf{Address} \cup \{\textbf{null}\} & \mathsf{val}
\end{array}$$

**Evaluation rules**:

$$\leadsto :\ \mathsf{e} \times \textbf{Heap} \times \textbf{Address} \rightarrow \textbf{Value} \cup \{\mathsf{TypeErr}, \mathsf{NullErr}\} \times \textbf{Heap}$$

$$\textbf{null}, \mathsf{H}, \iota \leadsto \textbf{null}, \mathsf{H} \quad \text{(R1)}$$

$$\frac{\mathit{Walk}(\mathsf{H}, \iota, \mathsf{path}) = \mathsf{val}}{\mathsf{path}, \mathsf{H}, \iota \leadsto \mathsf{val}, \mathsf{H}} \quad \text{(R3)}$$

$$\frac{\mathsf{e}, \mathsf{H}, \iota \leadsto \mathsf{val}, \mathsf{H}' \qquad \mathsf{e}', \mathsf{H}', \iota \leadsto \mathsf{val}', \mathsf{H}''}{\mathsf{e}\,\mathbf{;}\,\mathsf{e}', \mathsf{H}, \iota \leadsto \mathsf{val}', \mathsf{H}''} \quad \text{(R2)}$$

$$\frac{\mathsf{path}, \mathsf{H}, \iota \leadsto \iota', \mathsf{H} \qquad \mathsf{H}(\iota')(\mathsf{v}) = \mathsf{val}}{\mathsf{path.v}, \mathsf{H}, \iota \leadsto \mathsf{val}, \mathsf{H}} \quad \text{(R4)}$$

$$\frac{\begin{array}{c}\mathsf{path}, \mathsf{H}, \iota \leadsto \iota', \mathsf{H} \qquad \mathsf{e}, \mathsf{H}, \iota \leadsto \mathsf{val}, \mathsf{H}' \\ \mathsf{H}'(\iota')(\mathsf{v}) \neq \bot \qquad \mathsf{H}'' = \mathsf{H}'[\iota' \mapsto \mathsf{H}'(\iota')[\mathsf{v} \mapsto \mathsf{val}]]\end{array}}{\mathsf{path.v} = \mathsf{e}, \mathsf{H}, \iota \leadsto \mathsf{val}, \mathsf{H}''} \quad \text{(R5)}$$

$$\frac{\begin{array}{c}\mathsf{path}, \mathsf{H}, \iota \leadsto \iota', \mathsf{H} \qquad \mathsf{H} = \mathsf{H}_1 \\ \mathsf{e}_i, \mathsf{H}_i, \iota \leadsto \mathsf{val}_i, \mathsf{H}_{i+1}\ \text{ for } i \in \{1 ... |\overline{\mathsf{e}}|\} \\ \mathsf{H}' = \mathsf{H}_{|\overline{\mathsf{e}}|+1} \qquad \overline{\mathsf{p}} = \mathit{Assemble}(\mathit{Mix}(\mathsf{H}', \iota'), \mathsf{C}) \\ \mathit{Members}(\overline{\mathsf{p}}) = \overline{\mathsf{T}}\,\overline{\mathsf{f}}, \overline{\mathsf{T}}'\,\overline{\mathsf{v}} \qquad |\overline{\mathsf{f}}| = |\overline{\mathsf{val}}| \\ \iota'' \text{ is new in } \mathsf{H}' \qquad \mathit{Constr}(\mathsf{p}_{|\overline{\mathsf{p}}|}) = \mathsf{T}\ \mathsf{C}(\_)\{\mathsf{e}'; \} \\ \mathsf{H}'' = \mathsf{H}'[\ \iota'' \mapsto \llbracket\ \iota' \parallel \mathsf{C} \parallel \overline{\mathsf{f}} : \overline{\mathsf{val}}\quad \overline{\mathsf{v}} : \overline{\textbf{null}}\ \rrbracket\ ] \\ \mathsf{e}', \mathsf{H}'', \iota'' \leadsto \mathsf{val}, \mathsf{H}'''\end{array}}{\textbf{new}\ \mathsf{path.C}(\overline{\mathsf{e}}), \mathsf{H}, \iota \leadsto \mathsf{val}, \mathsf{H}'''} \quad \text{(R6)}$$

**Enclosing object**:

$$\mathit{Encl}(\llbracket\ \iota \parallel \_ \parallel \dots\ \rrbracket) = \iota$$

**Evaluation functions**:

$$\begin{array}{l}
\mathit{Walk}(\mathsf{H}, \iota, \textbf{this}) = \iota \\
\mathit{Walk}(\mathsf{H}, \iota, \mathsf{spine}.\textbf{out}) = \mathit{Encl}(\mathsf{H}(\iota')) \quad \text{if } \mathit{Walk}(\mathsf{H}, \iota, \mathsf{spine}) = \iota' \neq \iota_{\text{root}} \\
\mathit{Walk}(\mathsf{H}, \iota, \mathsf{path.f}) = \mathsf{val} \quad \text{if } \mathsf{H}(\mathit{Walk}(\mathsf{H}, \iota, \mathsf{path}))(\mathsf{f}) = \mathsf{val} \\
\\
\mathit{Walk}(\mathsf{H}, \iota, \mathsf{path.f}) = \mathsf{NullErr} \quad \text{if } \mathit{Walk}(\mathsf{H}, \iota, \mathsf{path}) = \textbf{null} \\
\mathit{Walk}(\mathsf{H}, \iota, \mathsf{path.f}) = \mathsf{TypeErr} \quad \text{if } \mathsf{H}(\mathit{Walk}(\mathsf{H}, \iota, \mathsf{path}))(\mathsf{f}) = \bot \\
\mathit{Walk}(\mathsf{H}, \iota, \mathsf{spine}.\textbf{out}) = \mathsf{TypeErr} \quad \text{if } \mathit{Walk}(\mathsf{H}, \iota, \mathsf{spine}) = \iota_{\text{root}}
\end{array}$$

**Error handling**:

$$\frac{\mathsf{path}, \mathsf{H}, \iota \leadsto \textbf{null}, \mathsf{H}}{\begin{array}{c}\mathsf{path.v}, \mathsf{H}, \iota \leadsto \mathsf{NullErr}, \mathsf{H} \\ \mathsf{path.v} = \mathsf{e}, \mathsf{H}, \iota \leadsto \mathsf{NullErr}, \mathsf{H} \\ \textbf{new}\ \mathsf{path.C}(\overline{\mathsf{e}}), \mathsf{H}, \iota \leadsto \mathsf{NullErr}, \mathsf{H}\end{array}} \quad \text{(ER1)}$$

$$\frac{\mathsf{path}, \mathsf{H}, \iota \leadsto \iota', \mathsf{H} \qquad \mathsf{H}(\iota')(\mathsf{v}) = \bot}{\begin{array}{c}\mathsf{path.v}, \mathsf{H}, \iota \leadsto \mathsf{TypeErr}, \mathsf{H} \\ \mathsf{path.v} = \mathsf{e}, \mathsf{H}, \iota \leadsto \mathsf{TypeErr}, \mathsf{H}\end{array}} \quad \text{(ER2)}$$

$$\frac{\begin{array}{c}\mathsf{path}, \mathsf{H}, \iota \leadsto \iota', \mathsf{H} \\ \mathit{Assemble}(\mathit{Mix}(\mathsf{H}, \iota'), \mathsf{C}) = \bot\end{array}}{\textbf{new}\ \mathsf{path.C}(\overline{\mathsf{e}}), \mathsf{H}, \iota \leadsto \mathsf{TypeErr}, \mathsf{H}} \quad \text{(ER3)}$$

$$\frac{\begin{array}{c}\mathsf{path}, \mathsf{H}, \iota \leadsto \iota', \mathsf{H} \\ \mathit{Assemble}(\mathit{Mix}(\mathsf{H}, \iota'), \mathsf{C}) = \overline{\mathsf{p}} \\ \mathit{Members}(\overline{\mathsf{p}}) = \overline{\mathsf{T}}\,\overline{\mathsf{f}}, \_ \qquad |\overline{\mathsf{e}}| \neq |\overline{\mathsf{f}}|\end{array}}{\textbf{new}\ \mathsf{path.C}(\overline{\mathsf{e}}), \mathsf{H}, \iota \leadsto \mathsf{TypeErr}, \mathsf{H}} \quad \text{(ER4)}$$

**Mixin Computation**:

$$\begin{array}{l}
\mathit{Mix}(\mathsf{H}, \iota_{\text{root}}) = [\mathsf{nil}_{\mathsf{C}}] \\
\mathit{Mix}(\mathsf{H}, \iota) = \mathit{Assemble}(\mathit{Mix}(\mathsf{H}, \iota'), \mathsf{C}) \\
\qquad \text{where } \mathsf{H}(\iota) = \llbracket\ \iota' \parallel \mathsf{C} \parallel \dots\ \rrbracket
\end{array}$$

$$\mathit{Assemble}(\overline{\mathsf{p}}, \mathsf{C}) = \mathit{Linearize}[\,\mathit{Expand}(\overline{\mathsf{p}}, \mathsf{p}) \mid \mathsf{p} \leftarrow \mathit{Defs}(\overline{\mathsf{p}}, \mathsf{C})\,]$$

$$\begin{array}{l}
\mathit{Defs}(\overline{\mathsf{p}}, \mathsf{C}) = \mathit{check}[\,\mathsf{p.C} \mid \mathsf{p} \leftarrow \overline{\mathsf{p}}, \mathit{CT}(\mathsf{p.C}) \neq \bot\,] \\
\quad \text{where } \mathit{check}(\overline{\mathsf{p}}) = \begin{cases} \bot & |\overline{\mathsf{p}}| = 0 \\ \overline{\mathsf{p}} & \mathit{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathit{Expand}(\overline{\mathsf{p}}, \mathsf{p}) = \mathit{Linearize}([\,\mathit{Assemble}(\overline{\mathsf{p}}, \mathsf{C}) \mid \mathsf{C} \leftarrow \overline{\mathsf{C}}\,]\,\mathsf{p}) \\
\quad \text{where } \mathit{CT}(\mathsf{p}) = \textbf{class}\ \mathsf{C}'\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\ \dots\ \}
\end{array}$$

$$\begin{array}{lcl}
\mathit{Linearize}(\mathsf{nil}_{\overline{\mathsf{p}}}) = \mathsf{nil}_{\mathsf{p}} & & \\
\mathit{Linearize}(\overline{\overline{\mathsf{p}}}\ \overline{\mathsf{p}}) = \mathit{Lin2}(\mathit{Linearize}(\overline{\overline{\mathsf{p}}}), \overline{\mathsf{p}}) & & \\
\mathit{Lin2}(\mathsf{nil}_{\mathsf{p}}, \mathsf{nil}_{\mathsf{p}}) & = & \mathsf{nil}_{\mathsf{p}} \\
\mathit{Lin2}(\overline{\mathsf{p}}\ \mathsf{p}, \overline{\mathsf{p}}'\ \mathsf{p}) & = & \mathit{Lin2}(\overline{\mathsf{p}}, \overline{\mathsf{p}}')\ \mathsf{p} \\
\mathit{Lin2}(\overline{\mathsf{p}}, \overline{\mathsf{p}}'\ \mathsf{p}') & = & \mathit{Lin2}(\overline{\mathsf{p}}, \overline{\mathsf{p}}')\ \mathsf{p}',\ \text{ if } \mathsf{p}' \notin \overline{\mathsf{p}} \\
\mathit{Lin2}(\overline{\mathsf{p}}\ \mathsf{p}, \overline{\mathsf{p}}') & = & \mathit{Lin2}(\overline{\mathsf{p}}, \overline{\mathsf{p}}')\ \mathsf{p},\ \text{ if } \mathsf{p} \notin \overline{\mathsf{p}}' \\
\mathit{Lin2}(\overline{\mathsf{p}}\,\mathsf{p}'\overline{\mathsf{p}}''\mathsf{p}, \overline{\mathsf{p}}'\mathsf{p}') & = & \mathit{Lin2}(\overline{\mathsf{p}}\,\overline{\mathsf{p}}''\mathsf{p}, \overline{\mathsf{p}}')\ \mathsf{p}'
\end{array}$$

**Figure 8.** Operational semantics of *vc*

exactly the cases covered by the bottommost clause in the definition of *Lin2*, i.e., the cases where the two operands contradict each other with respect to the ordering of shared mixins (intuitively this means that they disagree about which mixin should be the more specific one); in these cases, *Lin2* resolves the conflict by letting the rightmost operand decide the outcome.

The final result of computing $\mathit{Mix}(\mathsf{H}, \iota_4)$ is the mixin list $\mathsf{Base.Exp\ WithEval.Exp\ WithNeg.Neg\ NegAndEval.Neg}$ .

*Lin2* is a total function on lists of mixins, and the set of mixins in the result is equal to the union of the sets of mixins in the operands. For soundness the set of mixins is relevant but the ordering makes no difference, so this generalization of C3 enhances the expressive power without affecting type safety.

## 4.3 Evaluation Rules and Error Handling

The evaluation relation $\mathsf{e}, \mathsf{H}, \iota \leadsto \mathsf{r}, \mathsf{H}'$ reduces an expression, a heap, and a current object to a value or an error and a new heap. The current object plays the role of the environment.

The expression **null** evaluates to the null value (R1). An expression sequence $\mathsf{e}\,\mathbf{;}\,\mathsf{e}'$ evaluates to the result of evaluating $\mathsf{e}'$ in the heap that results from evaluating $\mathsf{e}$ (R2).

Evaluation of a path $\mathsf{path}$ does not affect the heap (R3). The value of the path is computed by the function $\Downarrow$, which "walks" a $\mathsf{path}$ from an address $\iota$ in the heap $\mathsf{H}$ to return the value specified by the path. As a base case, $\Downarrow$ returns $\iota$ when applied to the trivial path, **this**; $\mathsf{spine}.\textbf{out}^n$ locates the nth enclosing object of $\iota$; finally a path $\mathsf{path.f}$ finds the object $\iota'$ for $\mathsf{path}$ and then returns the value of the field $\mathsf{f}$ in the object $\iota'$.

Variable lookup $\mathsf{path.v}$ evaluates $\mathsf{path}$ to get $\iota'$, which is then looked up in the heap to get the variable's value (R4). An assignment $\mathsf{path.v} = \mathsf{e}$ evaluates $\mathsf{path}$ and $\mathsf{e}$ to $\iota'$ and $\mathsf{val}$ (R5). It then checks that the variable is defined on the object and updates the heap to set variable $\mathsf{v}$ of $\iota'$ to $\mathsf{val}$. The notation $\mathsf{H}(\iota)(\mathsf{m})$ means lookup of the value of a field or variable $\mathsf{m}$ in the object $\iota$. The notation $[\mathsf{v} \mapsto \mathsf{val}]$ appended to an object denotes (functional) update of the variable $\mathsf{v}$ of that object, and $\mathsf{H}[\iota \mapsto ...]$ denotes heap update.

In (R6) a new object **new** $\mathsf{path.C}(\overline{\mathsf{e}})$ is constructed by instantiating the virtual class C defined in the enclosing object $\iota'$ identified by path. The behavior $\overline{\mathsf{p}}$ of the new object is assembled from the mixins of the enclosing object as described in Section 4.2. If the enclosing object does not contain a definition of C, then $\mathcal{A}ssemble$ returns $\perp$ and rule (R6) does not apply. The mixin list $\overline{\mathsf{p}}$ also specifies the members and the most specific constructor of the new object. To construct the object, the heap is extended to define a new address $\iota''$ bound to a new object with enclosing object $\iota'$, class C, fields initialized to the evaluated constructor arguments, and variables initialized to null. The constructor body is then evaluated in the context of this new object. The result of the constructor is the result of the entire expression. If the constructor body is **this** (i.e., the class is used as a class in the conventional sense), then the result of the constructor call is $\iota''$.

Two different kinds of error can occur during evaluation: Type errors (TypeErr) and null pointer errors (NullErr). The rule (ER1) handles access to a property of an object, where the object is **null**. (ER2) to (ER4) define the situations in which a type error occurs, namely if a member to be read or written is not available (ER2), or when creating an instance of a class C, but the enclosing object has no definition of C, i.e., its mixin list is empty (ER3), or the number of parameters does not match (ER4).

The rules for propagating errors are standard and straightforward, so they are omitted; the sequel assumes that NullErr or TypeErr errors are propagated. The complete list of error rules is given in the technical report accompanying this paper [10].

## 5. Type System

The *vc* type system uses nominal typing based on *paths* to objects containing virtual classes. Typing domains, type checking rules, and functions for abstract interpretation are given in Figure 10.

### 5.1 Types

The type of an expression describes an object $\iota$ obtained by evaluation of it in one of two ways. In the first case a path which leads to the object $\iota$ itself is computed statically, and in the second case a path to the *enclosing* object of $\iota$ is computed, as well as a class name characterizing the class of $\iota$ itself. The former is an *object type*, u, and the latter is a *class type*, s. An object type contains more information than a class type, because every object type can be converted into a class type, but not vice versa. Since a path only makes sense as seen from a lexical point $\mathsf{p}'$ in the program, typing judgements have the form $\mathsf{p}' \vdash \mathsf{e} : \mathsf{t}$, where t is a type and $\mathsf{p}'$ represents the current **this** object.

An object type u has the form $\langle \mathsf{p} \rangle.\overline{\mathsf{f}}$. If an expression has the object type $\langle \mathsf{p} \rangle.\overline{\mathsf{f}}$ as seen from $\mathsf{p}'$, then p is a prefix of $\mathsf{p}'$, and the object denoted by the expression can be reached by going **out** $(|\mathsf{p}'| - |\mathsf{p}|)$ steps and then following $\overline{\mathsf{f}}$ in the heap. More formally, if the program and heap H are well-formed, the expression e is typable by $\mathsf{p}' \vdash \mathsf{e} : \langle \mathsf{p} \rangle.\overline{\mathsf{f}}$ in this program, the object $\iota_0$ is appropriate as **this** for $\mathsf{p}'$, and e evaluates by $\mathsf{e}, \mathsf{H}, \iota_0 \leadsto \iota, \mathsf{H}'$, then $\mathcal{W}alk(\mathsf{H}', \iota_0, \mathbf{this}.\mathbf{out}^j.\overline{\mathsf{f}}) = \iota$, where $j = \mathcal{D}epth(\mathsf{H}', \iota_0) - |\mathsf{p}|$.

A class type s is on the form $\langle \mathsf{p} \rangle.\overline{\mathsf{f}}.\mathsf{C}$. If an expression e has type $\langle \mathsf{p} \rangle.\overline{\mathsf{f}}.\mathsf{C}$ and $\mathsf{e}, \mathsf{H}, \iota_0 \leadsto \iota, \mathsf{H}'$ as above then $\langle \mathsf{p} \rangle.\overline{\mathsf{f}}$ is an object type describing the enclosing object $\mathcal{E}ncl(\mathsf{H}'(\iota))$, and $\iota$ is an instance of the class C which is nested in $\mathcal{E}ncl(\mathsf{H}'(\iota))$, or a subclass thereof.

The type checker computes object types for paths or path-like expressions (like a sequence containing a path as last element). For an expression like path.v or **new** path.C, an object type cannot be computed because, in general, there is no path to that object. However, there is always a path to its enclosing object in these cases, hence such expressions can be assigned a class type.

### 5.2 Abstract interpretation of the heap

The operational semantics defines functions to navigate a heap and compute mixin lists of objects. In particular, $\mathcal{E}ncl$ navigates to an enclosing object, $\mathcal{W}alk$H follows a path starting from some object, and $\mathcal{M}ix$ computes the mixin list of an object. An abstract interpretation of these functions is at the core of the type system: $\mathcal{E}$, $\mathcal{W}$, and $\mathcal{M}$ are the static versions of $\mathcal{E}ncl$, $\mathcal{W}alk$, and $\mathcal{M}ix$, respectively. They serve the same purpose as their dynamic counterparts, but they receive and produce types instead of objects. Before going into the details of their definition, we will at first state some properties of $\mathcal{E}$, $\mathcal{W}$, and $\mathcal{M}$ and discuss the connection with $\mathcal{E}ncl$, $\mathcal{W}alk$H, and $\mathcal{M}ix$ (the formal statements and proofs of these properties are in [10]).

The most important connections between the static and dynamic semantics are (a) if a navigation along a path is ok in the abstract interpretation of the heap then the corresponding navigation is also ok in the dynamic heap, and (b) navigation preserves agreement. Agreement, which is formally defined later in this section, states that an object $\iota$ has type t as seen from an object $\iota_0$ in a heap H, written $\mathsf{H}, \iota_0 \vdash \iota \rhd \mathsf{t}$. Given a well-formed program and a well-formed heap and $\mathsf{H}, \iota_0 \vdash \iota \rhd \mathsf{t}$, then the following holds:

1. Enclosing types agree with enclosing objects: if t is not the type of the root object, then $\mathcal{E}ncl(\mathsf{H}(\iota))$ exists and $\mathsf{H}, \iota_0 \vdash \mathcal{E}ncl(\mathsf{H}(\iota)) \rhd \mathcal{E}(\mathsf{t})$.
2. The statically known set of mixins is a subset of the dynamic set of mixins, $\mathcal{M}ix(\mathsf{H}, \iota) \supseteq \mathcal{M}(\mathsf{t})$.
3. If a field or variable exists according to the abstract interpretation then it exists in the heap: $\mathcal{E}xists(\mathsf{t}, \mathsf{m}) \Rightarrow \mathsf{H}(\iota)(\mathsf{m}) \neq \perp$.
4. If t is an object type u and a path is valid in both the heap and its abstract interpretation, then the results will agree: given $\mathcal{W}alk(\mathsf{H}, \iota, \mathsf{path}) = \mathsf{val}$ and $\mathcal{W}(\mathsf{u}, \mathsf{path}) = \mathsf{t}'$ then $\mathsf{H}, \iota_0 \vdash \mathsf{val} \rhd \mathsf{t}'$.

Both the heap and its abstract interpretation are also *enclosing-correct*, which informally means that for any declared field path.C f, the enclosing object of the value of the field must be equal to the object specified by the path, relative to the object containing the field. More formally, a well-formed dynamic heap ensures $\mathcal{W}alk(\mathsf{H}, \iota, \mathsf{path}) = \mathcal{E}ncl(\mathsf{H}(\mathcal{W}alk(\mathsf{H}, \iota, \mathsf{f})))$, where $\mathsf{path.C\ f} \in \mathcal{M}embers(\mathcal{M}ix(\mathsf{H}, \iota))$ and $\mathsf{H}(\iota)(\mathsf{f}) \neq \mathbf{null}$. Similarly, the static semantics ensures $\mathcal{W}(\mathsf{u}, \mathsf{path}) = \mathcal{E}(\mathcal{W}(\mathsf{u}, \mathsf{f}))$, where $\mathcal{D}clType(\mathsf{u}, \mathsf{f}) = \mathsf{path.C}$.

Let us now consider the definition of these functions in detail. The $\mathcal{W}$ function takes an object type u and a path path or a syntactic type T and produces an object type or a class type, if it succeeds. If the second argument is a path path, the intuition is that $\mathcal{W}$ computes a type for the object that is reached from the object described by u by traversing path in the heap. A naive approach would be to concatenate path to the path in u, but it would be hard to tell whether such a concatenated path leads to the same object as another concatenated path. The ability to decide whether two paths lead to the same object, however, is crucial for determining the subtyping relation, since only objects with identical enclosing object are compatible. For this reason, $\mathcal{W}$ returns a *canonical* representation of the combined path, namely a type. It is canonical in that the path inside the type has the form $\mathsf{spine}.\overline{\mathsf{f}}$. Object types can hence be compared by simple equality tests in order to determine whether they refer to the same object.

For the empty path **this**, $\mathcal{W}$ simply returns u (first case). For paths ending in **out**, the function $\mathcal{E}$ is used to find the enclosing type (second case). Paths ending in a field or a class are checked for validity: an appropriate field or class must exist. The last case in $\mathcal{W}$ extends the domain of the second argument to T; this is the only case where $\mathcal{W}$ returns a class type. As an exam-

**Typing domains**:

$$\begin{array}{lll@{\qquad}lll} \mathsf{u} & ::= & \langle \mathsf{p}\rangle.\overline{\mathsf{f}} & \mathsf{q} & ::= & \textbf{this} \mid \textbf{out} \mid \mathsf{f} \\ \mathsf{s} & ::= & \langle \mathsf{p}\rangle.\overline{\mathsf{f}}.\mathsf{C} & \mathsf{Q} & ::= & \overline{\mathsf{q}} \mid \overline{\mathsf{q}}.\mathsf{C} \\ \mathsf{t} & ::= & \mathsf{u} \mid \mathsf{s} & & & \end{array}$$

**Expression Typing**:

$$\frac{\mathcal{M}(\mathsf{t}) \neq \bot}{\mathsf{p} \vdash \textbf{null} : \mathsf{t}} \ \text{(T1)} \qquad \frac{\mathcal{W}(\langle \mathsf{p}\rangle, \mathsf{path}) = \mathsf{u}}{\mathsf{p} \vdash \mathsf{path} : \mathsf{u}} \ \text{(T3)}$$

$$\frac{\mathsf{p} \vdash \mathsf{e} : \mathsf{t} \quad \mathsf{p} \vdash \mathsf{e}' : \mathsf{t}'}{\mathsf{p} \vdash \mathsf{e}\,\mathbf{;}\,\mathsf{e}' : \mathsf{t}'} \ \text{(T2)} \qquad \frac{\mathsf{p} \vdash \mathsf{path} : \mathsf{u} \quad \mathcal{W}(\mathsf{u}, \mathit{DclType}(\mathsf{u},\mathsf{v})) = \mathsf{s}}{\mathsf{p} \vdash \mathsf{path.v} : \mathsf{s}} \ \text{(T4)}$$

$$\frac{\mathsf{p} \vdash \mathsf{path.v} : \mathsf{s} \qquad \mathsf{p} \vdash \mathsf{e} : \mathsf{t} \qquad \mathcal{C}(\mathsf{t}) <: \mathsf{s}}{\mathsf{p} \vdash \mathsf{path.v} = \mathsf{e} : \mathsf{t}} \ \text{(T5)}$$

$$\frac{\begin{array}{c} \mathsf{p} \vdash \mathsf{path} : \mathsf{u} \quad \mathsf{p}' \in \mathcal{M}(\mathsf{u.C}) \qquad \mathsf{p} \vdash \overline{\mathsf{e}} : \overline{\mathsf{t}} \\ \mathit{Constr}(\mathsf{p}') = \mathsf{T}_0\ \mathsf{C}(\overline{\mathsf{T}}\ \overline{\mathsf{f}})\ ... \qquad |\overline{\mathsf{T}}| = |\overline{\mathsf{t}}| \\ \mathsf{s}_i = \begin{cases} \mathcal{W}(\mathsf{u}, \textbf{this}.\mathsf{Q}) & \text{if } \mathsf{T}_i = \textbf{this}.\textbf{out}.\mathsf{Q} \\ \mathcal{W}(\mathsf{u}_j, \textbf{this}.\mathsf{Q}) & \text{if } \mathsf{T}_i = \textbf{this}.\mathsf{f}_j.\mathsf{Q} \wedge \mathsf{t}_j = \mathsf{u}_j \end{cases} \\ \text{for } i = 0...|\overline{\mathsf{t}}| \\ \mathcal{C}(\mathsf{t}_i) <: \mathsf{s}_i \text{ for } i = 1...|\overline{\mathsf{t}}| \end{array}}{\mathsf{p} \vdash \textbf{new}\ \mathsf{path.C}(\overline{\mathsf{e}}) : \mathsf{s}_0} \ \text{(T6)}$$

**Conversion to class types**:

$$\begin{array}{lcl} \mathcal{C} & :: & \mathsf{t} \rightarrow \mathsf{s} \\ \mathcal{C}(\langle \mathsf{p.C}\rangle) & = & \langle \mathsf{p}\rangle.\mathsf{C} \\ \mathcal{C}(\mathsf{u.f}) & = & \mathcal{W}(\mathsf{u}, \mathit{DclType}(\mathsf{u},\mathsf{f})) \\ \mathcal{C}(\mathsf{s}) & = & \mathsf{s} \end{array}$$

**Mixins**:

$$\begin{array}{lcl} \mathcal{M} & :: & \mathsf{t} \rightarrow \overline{\mathsf{p}} \\ \mathcal{M}(\langle\rangle) & = & [\mathsf{nil_c}] \\ \mathcal{M}(\mathsf{u.C}) & = & \mathit{Assemble}(\mathcal{M}(\mathsf{u}), \mathsf{C}) \\ \mathcal{M}(\mathsf{u}) & = & \mathcal{M}(\mathcal{C}(\mathsf{u})) \end{array}$$

**Enclosing object type**:

$$\begin{array}{ll} \mathcal{E} & :: \mathsf{t} \rightarrow \mathsf{u} \\ \mathcal{E}(\mathsf{u.C}) = \mathsf{u} & \\ \mathcal{E}(\mathsf{u}) & = \mathcal{E}(\mathcal{C}(\mathsf{u})) \end{array}$$

**Static lookup**:

$$\begin{array}{ll} \mathcal{W} & :: \mathsf{u} \times (\mathsf{path} \cup \mathsf{T}) \rightarrow \mathsf{t} \\ \mathcal{W}(\mathsf{u}, \textbf{this}) & = \mathsf{u} \\ \mathcal{W}(\mathsf{u}, \mathsf{spine}.\textbf{out}) & = \mathcal{E}(\mathcal{W}(\mathsf{u}, \mathsf{spine})) \\ \mathcal{W}(\mathsf{u}, \mathsf{path.f}) & = \mathcal{W}(\mathsf{u}, \mathsf{path}).\mathsf{f} \ \text{ if } \mathit{Exists}(\mathcal{W}(\mathsf{u}, \mathsf{path}), \mathsf{f}) \\ \mathcal{W}(\mathsf{u}, \mathsf{path.C}) & = \mathcal{W}(\mathsf{u}, \mathsf{path}).\mathsf{C} \ \text{ if } \mathit{Exists}(\mathcal{W}(\mathsf{u}, \mathsf{path}), \mathsf{C}) \end{array}$$

**Program Typing**:

$$\frac{\mathcal{M}(\langle \mathsf{p}\rangle.\mathsf{C}) \neq \bot}{\mathsf{p} \vdash \mathsf{C}\ \mathsf{OK}} \ \text{(WF1)} \qquad \frac{\mathcal{W}(\langle \mathsf{p}\rangle, \mathsf{T}) \neq \bot}{\mathsf{p} \vdash \mathsf{T}\ \mathsf{OK}} \ \text{(WF2)}$$

$$\frac{\mathsf{C} = \mathsf{C}' \Rightarrow \mathsf{T} = \mathsf{T}', \overline{\mathsf{T}}\ \overline{\mathsf{f}} = \overline{\mathsf{T}}'\ \overline{\mathsf{f}}'}{\mathsf{T}\ \mathsf{C}(\overline{\mathsf{T}}\ \overline{\mathsf{f}})\ \{\mathsf{e};\}\ \text{overrides}\ \mathsf{T}'\ \mathsf{C}'(\overline{\mathsf{T}}'\ \overline{\mathsf{f}}')\ \{\mathsf{e}';\}\ \mathsf{OK}} \ \text{(WF3)}$$

$$\frac{\begin{array}{c} \mathsf{K} = \mathsf{T}\ \mathsf{C}(\overline{\mathsf{T}}''\ \overline{\mathsf{f}}')\ \{\,\mathsf{e};\} \qquad \mathcal{M}(\langle \mathsf{p}\rangle.\mathsf{C}) = \overline{\mathsf{p}} \\ \mathit{Members}(\overline{\mathsf{p}}) = \overline{\mathsf{T}}''\ \overline{\mathsf{f}}',\ \_ \\ \mathsf{p} \vdash \overline{\mathsf{C}}\ \mathsf{OK} \quad \mathsf{p.C} \vdash \overline{\mathsf{T}}\ \mathsf{OK} \quad \mathsf{p.C} \vdash \overline{\mathsf{T}}'\ \mathsf{OK} \quad \mathsf{p.C} \vdash \mathsf{T}\ \mathsf{OK} \\ \mathsf{p.C} \vdash \mathsf{e} : \mathsf{t} \qquad \mathcal{C}(\mathsf{t}) <: \mathcal{W}(\langle \mathsf{p.C}\rangle, \mathsf{T}) \\ \mathsf{K'} = \mathit{Constr}(\mathsf{p}_j) \Rightarrow \mathsf{K}\ \text{overrides}\ \mathsf{K'}\ \mathsf{OK} \end{array}}{\mathsf{p} \vdash\ \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{\,\mathsf{K}\ \overline{\mathsf{CL}};\ \overline{\mathsf{T}}\,\overline{\mathsf{f}};\ \overline{\mathsf{T}}'\ \overline{\mathsf{v}}\}\ \mathsf{OK}} \ \text{(WF4)}$$

$$\frac{\begin{array}{c} \text{There is a strict partial order } \sqsubset_f \text{ on field names such that} \\ \forall \mathsf{p}, \mathsf{f}. \quad \mathsf{spine}.\overline{\mathsf{f}}.\mathsf{C}\ \mathsf{f} \in \mathit{Members}(\mathsf{p}) \Rightarrow \forall i.\ \mathsf{f}_i \sqsubset_f \mathsf{f} \\ \text{There is a strict partial order } \sqsubset_c \text{ on class names such that} \\ \forall \mathsf{p}.\ \mathit{CT}(\mathsf{p}) =\ \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \overline{\mathsf{C}}\ \{...\} \Rightarrow \forall i.\ \mathsf{C}_i \sqsubset_c \mathsf{C} \end{array}}{\mathit{CT}\ \text{is acyclic}} \ \text{(WF5)}$$

$$\frac{\begin{array}{c} \mathit{CT}\ \text{is acyclic} \\ \forall \mathsf{p}, \mathsf{p}', \mathsf{C} : \mathit{CT}(\mathsf{p.C}) \neq \bot, \mathit{CT}(\mathsf{p}'.\mathsf{C}) \neq \bot \Rightarrow \\ \mathsf{p}''.\mathsf{C} \in \mathcal{M}(\langle \mathsf{p}\rangle.\mathsf{C}) \cap \mathcal{M}(\langle \mathsf{p}'\rangle.\mathsf{C}) \\ \forall \mathsf{p} \neq \mathsf{p}' : \mathit{CT}(\mathsf{p}) =\ \textbf{class}\ \mathsf{C}\ ...\ \{\,\mathsf{K}\ \overline{\mathsf{CL}};\ \overline{\mathsf{T}}\,\overline{\mathsf{f}};\ \overline{\mathsf{T}}'\ \overline{\mathsf{v}}\} \\ \mathit{CT}(\mathsf{p}') =\ \textbf{class}\ \mathsf{C}'\ ...\ \{\,\mathsf{K}'\ \overline{\mathsf{CL}}';\ \overline{\mathsf{T}}''\ \overline{\mathsf{f}}';\ \overline{\mathsf{T}}'''\ \overline{\mathsf{v}}'\} \\ \Rightarrow \overline{\mathsf{f}} \cap \overline{\mathsf{f}}' = \emptyset, \overline{\mathsf{v}} \cap \overline{\mathsf{v}}' = \emptyset \\ \forall \mathsf{p}, \mathsf{C} : \mathit{CT}(\mathsf{p.C}) \neq \bot \Rightarrow \mathsf{p} \vdash \mathit{CT}(\mathsf{p.C})\ \mathsf{OK} \end{array}}{\mathit{CT}\ \mathsf{OK}} \ \text{(WF6)}$$

**Subtyping**:

$$\mathsf{s} <: \mathsf{s} \quad \text{(S-REFL)} \qquad \frac{\mathsf{s} <: \mathsf{s}' \quad \mathsf{s}' <: \mathsf{s}''}{\mathsf{s} <: \mathsf{s}''} \ \text{(S-TRANS)}$$

$$\frac{\mathcal{M}(\mathsf{u}) = \overline{\mathsf{p}} \qquad \mathit{CT}(\mathsf{p}_j.\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ ...\mathsf{C}'...}{\mathsf{u.C} <: \mathsf{u.C}'} \ \text{(S-DECL)}$$

**Declared type of member:**

$$\begin{array}{lcl} \mathit{DclType}(\mathsf{t},\mathsf{m}) & = & \mathsf{T} \quad \text{where } \mathsf{T}\ \mathsf{m} \in \mathit{Members}(\mathcal{M}(\mathsf{t})) \\ \mathit{Exists}(\mathsf{t},\mathsf{m}) & = & (\mathit{DclType}(\mathsf{t},\mathsf{m}) \neq \bot) \\ \mathit{Exists}(\mathsf{u},\mathsf{C}) & = & (\mathcal{M}(\mathsf{u.C}) \neq \bot) \end{array}$$

**Figure 10.** Typing rules

ple based on the definitions in Figures 1 and 2, we would have $\mathcal{W}(\langle \mathsf{WithNeg.Neg}\rangle.\mathsf{e}, \textbf{this}.\textbf{out}.\mathsf{Lit}) = \langle \mathsf{WithNeg}\rangle.\mathsf{Lit}$.

Object types can be converted into class types by means of the $\mathcal{C}$ function as follows: If the object type is just a static path and no field accesses, then the enclosing object is described by the same static path with the last element removed, and the class is that last element (first case). If the object type ends with a field, the field is replaced by its declared type (*DclType* is explained below) and the $\mathcal{W}$ is called to normalize the resulting path (second case). If the type is already a class type, there is nothing to do (third case).

The $\mathcal{M}$ function computes the statically known mixin structure of an object described by a type. The type $\langle\rangle$ describes the root object which has only one mixin, namely the empty class path: $[\mathsf{nil_c}]$ (first case). For an object type $\mathsf{u.C}$, $\mathsf{u}$ is a type that describes the enclosing object, hence its mixin list can be recursively computed from the enclosing object. This mixin list and the class name $\mathsf{C}$ are sufficient to compute the mixin list for this type by calling the *Assemble* function (second case). Finally, to compute the mixin list of an object type it is first converted to a class type (third case). For example, with the code in Figure 1-5 the mixin lists are $\mathcal{M}(\langle \mathsf{Test}\rangle.\mathsf{f1.Neg}) = [\mathsf{Base.Exp}, \mathsf{WithNeg.Neg}]$ and $\mathcal{M}(\langle \mathsf{Test}\rangle.\mathsf{f2.Exp}) = [\mathsf{Base.Exp}, \mathsf{WithEval.Exp}]$.

The *DclType* function uses $\mathcal{M}$ to look up a field or variable declaration in the mixin list of a given type.

$\mathcal{C}$, $\mathcal{E}$, $\mathcal{W}$, $\mathcal{M}$ and *DclType* depend on each other in non-trivial ways, so it is not obvious that evaluation of these functions will terminate. A proof is given in [10]. Informally, the functions terminate because the arguments to recursive calls of $\mathcal{W}$ inside $\mathcal{W}$ and *DclType* are smaller, and the recursive call inside $\mathcal{C}$ replaces a field by its declared type. The latter case is also guaranteed to terminate because programs are well-formed only if there are no cyclic dependencies on field types, as explained later in this section.

### 5.3 Subtyping

Subtyping determines the compatibility of values for assignment or parameter binding. It is defined only on class types but object types can always be converted to class types via $\mathcal{C}$. The main rule for the subtyping relation, (S-DECL), defines type compatibility through a combination of path equality and examination of declared subclass relationships. The latter is standard in object-oriented type systems: a class $B$ is a subtype of $A$ if $B$ is derived by subclassing from $A$. This traditional definition is modified in *vc* to take into account virtual classes: two classes can only be in a subtype relation *if they are contained in the same object*; this is a concrete manifestation of the fact that types depend on the enclosing object. Rule (S-DECL) ensures that subtypes are always based on the same object type $\mathsf{u}$. Since an object type describes a path to an object, the enclosing objects must be identical. This comparison for identical enclosing object types works because object types are paths in a normalized form.

### 5.4 Expression Typing

Expressions are given a type in the context of a static path $\mathsf{p}$ which describes the current object **this**. As in the operational semantics, an environment is not needed because method parameters are encoded as fields.

The **null** value (T1) has any meaningful type, whereby "meaningful" is checked by ensuring that the type has mixins. The type of a sequence is the type of the last expression in the sequence (T2). Paths (T3) are given a type using the static lookup function $\mathcal{W}$ explained in Section 5.2. As is obvious from the definition, paths have an object type. Variable lookup (T4) also uses $\mathcal{W}$, but in this case the type of the variable is passed instead of the variable name. This is a manifestation of the fact that variables cannot be used in types. This also means, however, that the type of a variable access is always a class type, not an object type.

An assignment (T5) is checked by computing a type for the left hand side, which is known to be a class type by (T4), computing a type for the right hand side and then checking whether the left side is a subtype of the right side. If the left hand type is an object type, it is converted to a class type first.

The rule for object creation (T6) is the most complex, which is not surprising given that it also handles method calls. First, the type of the enclosing object $\mathsf{u}$ is computed. The statically known mixin structure of the new object, $\mathcal{M}(\mathsf{u.C})$, is computed, and a mixin is selected via the choice of $\mathsf{p}'$, which is then used to find the constructor signature. Note that all mixins will provide the same signature due to program well-formedness. The types of the arguments are computed; their number must be equal to the number of constructor arguments. The actual set of mixins at runtime may be larger than the statically known set, but program well-formedness ensures that the signature of the most specific constructor at runtime is identical to the one in the statically selected constructor.

To compare the syntactic types specified in the constructor with the types of the actual arguments, class types $\mathsf{s}_i$ are computed for every syntactic type in the constructor, including the return type. Intuitively, the syntactic types $\mathsf{T}_i$ must be adapted to the *viewpoint* $\mathsf{p}$. To do that, the static lookup function $\mathcal{W}$ is used again. The types $\mathsf{T}_i$ are either of the form **this**.**out**.... or **this**.$\mathsf{f}_j$...., depending on whether the argument type comes from the environment or another argument. (Syntactically, $\mathsf{T}_i$ could also have the form **this**.$\mathsf{C}'$ for some class name $\mathsf{C}'$, but this type would not be useful because it would refer to a virtual class of an object that does not yet exist.)

The first case applies to the traditional situation where the type of the argument is taken from the environment; TestLit in Figure 1

$$\frac{\begin{array}{c}\mathsf{Test} \vdash \mathbf{this} : \langle \mathsf{Test}\rangle. \quad \mathcal{M}(\langle \mathsf{Test}\rangle.\mathsf{buildNeg}) = \mathsf{Test.buildNeg} \\ \mathsf{Test} \vdash \mathsf{f2} : \langle \mathsf{Test}\rangle.\mathsf{f2} \qquad \mathsf{Test} \vdash \mathsf{f2.zero} : \langle \mathsf{Test}\rangle.\mathsf{f2.Lit} \\ \mathit{Constr}(\mathsf{Test.buildNeg}) = \\ \mathsf{ne.Neg\ buildNeg(out.out.WithNeg\ ne,\ ne.Exp\ ex)} \\ \mathsf{s}_0 = \mathcal{W}(\langle \mathsf{Test}\rangle.\mathsf{f2}, \mathbf{this}.\mathsf{Neg}) = \langle \mathsf{Test}\rangle.\mathsf{f2.Neg} \\ \mathsf{s}_1 = \mathcal{W}(\langle \mathsf{Test}\rangle., \mathbf{this}.\mathbf{out}.\mathsf{WithNeg}) = \langle \mathsf{WithNeg}\rangle. \\ \mathsf{s}_2 = \mathcal{W}(\langle \mathsf{Test}\rangle.\mathsf{f2}, \mathbf{this}.\mathsf{Exp}) = \langle \mathsf{Test}\rangle.\mathsf{f2.Exp} \\ \mathcal{C}(\langle \mathsf{Test}\rangle.\mathsf{f2}) = \langle \mathsf{NegAndEval}\rangle. <: \mathsf{s}_1 \\ \mathcal{C}(\langle \mathsf{Test}\rangle.\mathsf{f2.Lit}) = \langle \mathsf{Test}\rangle.\mathsf{f2.Lit} <: \mathsf{s}_2 \end{array}}{\mathsf{Test} \vdash \mathbf{new}\ \mathbf{this}.\mathsf{buildNeg(f2,f2.zero)} : \langle \mathsf{Test}\rangle.\mathsf{f2.Neg}}$$

**Figure 11.** Type derivation for buildNeg(f2,f2.zero) in Figure 5

is an example. In this case, **this**.**out** refers to the enclosing object of the class. The type of this enclosing object is the type of path, or the object type $\mathsf{u}$. The actual argument type $\mathsf{s}_i$ is then found by navigating from $\mathsf{u}$ into the tail of $\mathsf{T}_i$ using $\mathcal{W}$.

The latter case applies if an argument type depends on the virtual class of another argument, as for example buildNeg in Figure 5. In this case, $\mathsf{f}_j$ is initialized with the value of $\mathsf{e}_j$ at runtime. The actual argument type $\mathsf{s}_i$ is then found by navigating from $\mathsf{t}_j$ into the tail of $\mathsf{T}_i$ using $\mathcal{W}$. If an argument is used as type provider for another argument, then the expression for the argument needs to have an object type. This restriction is enforced by the condition $\mathsf{u}_j = \mathsf{t}_j$ in (T6).

The complete list of argument types $\mathsf{s}_i$ is then checked to be subtypes of the formal argument types. Finally, the viewpoint-adapted constructor return type $\mathsf{s}_0$ is returned.

Figure 11 shows an example of a non-trivial usage of (T6) in the example from Figure 5. It has been slightly adjusted to fit to the formal syntax, see Section 3.3. The example illustrates only the last step in the typing derivation, the result of sub-derivations has been inlined. Notice in particular that the type of the expression contains the information that the result has the family f2.

### 5.5 Program Typing

In order to separate out the problem of cyclic inheritance relations and cyclic field type dependencies (the type of a field may depend on the value of other fields), declared names are partially ordered such that each of the two kinds of dependencies are known to be acyclic (WF5). Consequently, cyclic inheritance relations and cyclic relations via dependent types (which are expressed using fields) cannot occur. We could relax this restriction without affecting soundness, but with the current strict ruleset it is easy to see that the type analysis always terminates, without adding special checks for infinite loops in type computations.

The overall program well-formedness rule, (WF6), requires that the program is acyclic, that two class declarations of the same class name have a shared mixin, that field and variable declarations are unique, and that each class declaration is well-formed.

A class is OK (WF4) if the list of constructor arguments matches the list of fields in the statically known mixin structure of the class, if all superclasses are valid, if the type of the constructor expression is compatible to the declared return type, and if all other mixins that have the same class name have the same constructor signature, see also (WF3). The validity of superclass and type declarations ((WF1) and (WF2)) is checked using the $\mathcal{M}$ and $\mathcal{W}$ functions, which return $\perp$ if the class or type, respectively, is not known to exist in the context $\mathsf{p}$.

Note that (WF4) implies that fields can only be declared in new class declarations (i.e., if there is no inherited class declaration with the same name); this restriction is not essential and we could easily add initialized fields (declared as $\mathsf{T\,f} = \mathsf{e}$) to the calculus which could be declared in all classes. (In fact, we developed the whole calculus with initialized and redefinable fields before we decided to add constructors and let fields be initialized via constructor arguments.) We have chosen to leave out initialized fields because they do add a number of details to rules, but do not provide much additional insight. We could also have allowed field declarations everywhere and accepted the possibility for additional run-time $\mathsf{NullErr}$ errors due to uninitialized fields, but we felt that the current strict approach is useful because it illustrates how to statically ensure that all fields are initialized. Also note that the restriction on fields does not affect the ability to declare variables and classes (possibly used as methods) in all class declarations, so there are no restrictions on ordinary width subtyping in the calculus.

As mentioned, (WF6) requires globally unique member names; that is, field and variable names must be unique throughout the program. This may seem like a serious restriction that could interfere with separate compilation, but it is in fact just a simple way to emulate an approach which is usable in a full-fledged language and which does not interfere with separate compilation. In particular, the gbeta compilation process extends all declared names with a unique identification of the enclosing class body (i.e., something that corresponds to the static path to the scope of the declaration). It is then resolved statically which name declaration each name usage refers to, and the name usage is then extended correspondingly. As a result, if a given object contains multiple members named $\mathsf{m}$, they will at run-time be distinct members with extended names $\mathsf{p_1\_m}$, $\mathsf{p_2\_m}$, etc., and name usages will use these extended names for lookups. Hence, field and variable lookup uses early binding, which is also the desired semantics. In Caesar, such name clashes are detected and rejected at compile time, so the programmer has to rename one of the features in case of a clash.

For class or method lookup the desired semantics is late binding, so in this case the technique is slightly different. (WF6) requires any two declarations of a class with the same name to have a shared declaration of that class in their statically known sets of mixins. This global restriction may seem to interfere with separate compilation. However, it can be removed in a way which is similar to the one used for members. First, note that in *vc* it is easy to show that for a given class name $\mathsf{C}$ there must be a unique declaration of $\mathsf{C}$ which is in this sense shared among all declarations of $\mathsf{C}$. In gbeta it is required that an "introductory" class declaration—i.e., one where no other declarations of the same class are known statically—is marked syntactically, not unlike the distinction between `virtual` and `override` methods in *C#*. Each introductory declaration for a class is renamed with an identification of its enclosing class body, just like a member declaration. Each non-introductory class declaration is renamed like a member name *usage* to have the same extended name as its introduction. This implies that every class declaration has one particular introduction, which is resolved statically. Finally, class name usages are renamed to be like their extended statically known declarations. As a result, there is no need for global restrictions, and it is possible for multiple classes with the same name to coexist in the same object. With respect to binding time, there is early binding of the choice of class introduction (class *identity*), but late binding of the actual value (the dynamic set of mixins). Our formalization is thus much simpler, but it models the approach taken in full-fledged languages in a faithful albeit not always direct manner.

## 6. Wellformed Heaps and Agreement

The soundness of the operational semantics with respect to the type system depends upon having a well-formed heap, and agreement between a value and a type relative to a heap. The rules for heap well-formedness and agreement are given in Figure 12. Since the details of these definitions are not required to understand the *vc* calculus as such, the remainder of this section can be skipped by readers who are less interested in how the soundness result is reached.

**Well-formedness**:

$$\frac{\mathsf{H}(\iota)(\mathsf{m}) = \mathbf{null}}{\iota.\mathsf{m} : \mathsf{T}\ \ \mathsf{OK\ in\ H}} \qquad (\text{WF-NULL})$$

$$\frac{\begin{array}{c}\mathsf{H}(\iota)(\mathsf{m}) = \iota' \\ \mathcal{Walk}(\mathsf{H}, \iota', \mathbf{out}) = \mathcal{Walk}(\mathsf{H}, \iota, \mathsf{path}) \quad \mathsf{p.C} \in \mathcal{Mix}(\mathsf{H}, \iota')\end{array}}{\iota.\mathsf{m} : \mathsf{path.C}\ \ \mathsf{OK\ in\ H}} \qquad (\text{WF-MEM})$$

$$\frac{\mathsf{T\ m} \in \mathcal{Members}(\mathcal{Mix}(\mathsf{H}, \iota)) \Rightarrow \iota.\mathsf{m} : \mathsf{T}\ \ \mathsf{OK\ in\ H}}{\iota\ \ \mathsf{OK\ in\ H}} \qquad (\text{WF-OBJ})$$

$$\frac{\mathsf{H}(\iota_{\text{root}}) = [\![\ \bot \parallel \mathsf{C}_{\text{root}} \parallel\ \ ]\!]}{\iota_{\text{root}}\ \ \mathsf{OK\ in\ H}} \qquad (\text{WF-ROOT})$$

$$\frac{\forall \iota.\ \iota\ \ \mathsf{OK\ in\ H}}{\mathsf{H\ OK}} \qquad (\text{WF-HEAP})$$

**Agreement**:

$$\mathsf{H}, \iota_0 \vdash \mathbf{null} \triangleright \mathsf{val}_s \qquad (\text{A-NULL})$$

$$\mathsf{H}, \iota_0 \vdash \iota_{\text{root}} \triangleright \langle\rangle \qquad (\text{A-ROOT})$$

$$\frac{\begin{array}{c} j = \mathcal{Depth}(\mathsf{H}, \iota_0) - |\mathsf{p}| \\ \mathcal{Walk}(\mathsf{H}, \iota_0, \mathbf{this}.\mathbf{out}^j.\bar{\mathsf{f}}) = \iota \quad \mathsf{H}, \iota_0 \vdash \iota \triangleright \mathcal{C}(\langle \mathsf{p} \rangle.\bar{\mathsf{f}})\end{array}}{\mathsf{H}, \iota_0 \vdash \iota \triangleright \langle \mathsf{p} \rangle.\bar{\mathsf{f}}} \qquad (\text{A-OTYPE})$$

$$\frac{\mathsf{p'.C} \in \mathcal{Mix}(\mathsf{H}, \iota) \quad \mathsf{H}, \iota_0 \vdash \mathcal{Encl}(\mathsf{H}(\iota)) \triangleright \mathcal{E}(\mathsf{u.C})}{\mathsf{H}, \iota_0 \vdash \iota \triangleright \mathsf{u.C}} \qquad (\text{A-CTYPE})$$

**Auxiliary definitions**:

$$\mathcal{Depth}(\mathsf{H}, \iota) = \begin{cases} 0, & \text{if } \iota = \iota_{\text{root}} \\ 1 + \mathcal{Depth}(\mathsf{H}, \mathcal{Encl}(\mathsf{H}(\iota))) \end{cases}$$

**Figure 12.** Dynamic well-formedness and agreement

A heap is well-formed if all its objects are well-formed (WF-HEAP). An object is well-formed if all its members are well-formed (WF-OBJ). An object member is well-formed if its value in the heap is null (WF-NULL). Otherwise a member $\mathsf{m}$ of object $\iota$ is well-formed if the member value $\iota' = \mathsf{H}(\iota)(\mathsf{m})$ satisfies two conditions: (1) the enclosing object of the value, $\mathcal{Walk}(\mathsf{H}, \iota', \mathbf{out})$, is equal to the object $\mathcal{Walk}(\mathsf{H}, \iota, \mathsf{path})$ specified by the path in the declared type $\mathsf{path.C}$; and (2) the mixins of the value, $\mathcal{Mix}(\mathsf{H}, \iota')$, include a path ending with the class $\mathsf{C}$. There is a special rule for well-formedness of the root object because it does not have an enclosing object.

Type agreement is specified as the agreement of an object at $\iota$ with a type $\mathsf{T}$, relative to a dynamic heap $\mathsf{H}$ and a starting point $\iota_0$. The starting point specifies an address in the dynamic heap that is related to the base of the type. **null** agrees with all types (A-NULL), and the root object agrees with the empty object type (A-ROOT).

Rule (A-OTYPE) handles object types, $\langle \mathsf{p} \rangle.\bar{\mathsf{f}}$. The rules ensure that the class path $\mathsf{p}$ is a prefix of the spine of $\iota_0$, so the value $j$ represents the number of enclosing objects that must be traversed from $\iota_0$ to read an object with the same depth as $\mathsf{p}$. The path $\mathbf{this}.\mathbf{out}^j.\bar{\mathsf{f}}$ traverses to this object, and then traverses the field list

$\bar{f}$. The object $\iota$ must be located at the end of this path. In addition, $\iota$ must agree with the corresponding class type.

Rule (A-CTYPE) handles class types, $\langle p\rangle.\bar{f}.C$. It requires that the mixins of the value, $\mathcal{M}ix(H,\iota)$, include a path ending with the type's class C. It also requires that the actual enclosing object agrees with the enclosing type.

## 7. Soundness

The type system of *vc* is sound in the sense that a well-typed expression either returns a value that agrees with its type, terminates with a NullErr, or diverges, but never terminates with a TypeErr. The soundness result is composed of two formal results: *preservation* and *coverage*. Preservation is the standard theorem which characterizes the result of expressions that are well-typed and evaluate to a result. Coverage is a new technique for ensuring that errors do not prevent expressions from evaluating to a result.

Preservation assumes a valid program and heap. Given the static path p of a class in which an expression e has type t, and the address $\iota$ of an object that agrees with p; if the expression evaluates to a result r then either the result is NullErr or it is a value that agrees with t. Preservation also guarantees that the heap is still well-formed after the execution, and that the current object still agrees with its type.

THEOREM 1 (Preservation).

$$\begin{bmatrix} \mathit{CT}\ \mathsf{OK} \\ \mathsf{H}\ \mathsf{OK} \\ \mathsf{p} \vdash \mathsf{e} : \mathsf{t} \\ \mathsf{H}, \iota \vdash \iota \rhd \langle \mathsf{p} \rangle \\ \mathsf{e}, \mathsf{H}, \iota \rightsquigarrow \mathsf{r}, \mathsf{H}' \end{bmatrix} \Rightarrow \begin{bmatrix} \mathsf{H}'\ \mathsf{OK} \\ \mathsf{H}', \iota \vdash \iota \rhd \langle \mathsf{p} \rangle \\ \quad \mathsf{r} = \mathsf{val}\ \wedge\ \mathsf{H}', \iota \vdash \mathsf{val} \rhd \mathsf{t} \\ \vee \\ \quad \mathsf{r} = \mathsf{NullErr} \end{bmatrix}$$

This theorem only characterizes evaluations that terminate, which is a natural consequence of using a big-step semantics. Hence it is slightly weaker than the usual "progress and preservation" theorems in a small-step semantics, where it can be expressed that execution of a type correct program will never get stuck even if the execution continues forever.

Preservation alone does not ensure soundness however, because an expression may fail to evaluate due to a missing case in the evaluation rules. We have followed standard practice by including rules (ER1-4) to cover a variety of error cases in evaluation [14]. The complete list of error rules is given along with the soundness proof. The second half of our soundness proof ensures that *all* error cases have been handled. As a result, the only way an evaluation can fail to produce a value is if the computation diverges. This Lemma plays a role similar to the 'progress' theorem when using a small-step semantics.

The purpose of the coverage lemma is to show that the evaluation rules always produce a value unless the computation diverges. First we define a notion of finite evaluation. If the evaluation exceeds the bound for finite evaluation, it produces a special termination value. The evaluation rules for error propagation propagate this special value.

DEFINITION 1 (Finite Evaluation). *Define an evaluation relation $\rightsquigarrow_k$ as a copy of the rules for $\rightsquigarrow$. Replace each occurrence of $\rightsquigarrow$ in a premise by $\rightsquigarrow_{k-1}$. Replace $\rightsquigarrow$ in the conclusion of each rule and axiom with $\rightsquigarrow_k$. Note that the copied axioms are defined for all k. Add the following axiom:*

$$\mathsf{e}, \mathsf{H}, \iota \rightsquigarrow_0 \mathsf{KillErr}, \mathsf{H} \qquad (\text{KILL})$$

The finite evaluation relation $\rightsquigarrow_n$ returns KillErr if the derivation is more than $n$ derivations deep. It is thus a finite approximation of the normal evaluation of an expression. The coverage lemma states that finite evaluation always produces a value.

LEMMA 1 (Coverage). *For all natural numbers $n$ and* e*,* H*,* $\iota$*, there exists* r*,* H$'$ *such that*

$$\mathsf{e}, \mathsf{H}, \iota \rightsquigarrow_n \mathsf{r}, \mathsf{H}'$$

The coverage lemma ensures that the operational semantics produces a value even in the face of runtime errors, such as access to non-existing members, see (ER2) and (ER3) in Figure 8.

A terminating expression is one for which there is an $n$ such that finite evaluation $\rightsquigarrow_n$ does not return KillErr. If the expression does not return KillErr, then it cannot use the KILL axiom. As a result, the derivation in $\rightsquigarrow_n$ can be translated to a derivation in $\rightsquigarrow$. Thus every terminating expression has a corresponding derivation in $\rightsquigarrow$.

Theorem 1 and Lemma 1 ensure the soundness of *vc*: execution of well-typed expressions will either produce a value of the correct type, return NullErr, or else diverge. But evaluation will never access non-existing fields, variables, or classes, and is never stuck.

Note that all proofs are provided in [10].

## 8. Related and Future Work

The idea of virtual classes and their different kinds of bindings stems from BETA [21]. The concept of virtual superclasses was explored but never fully realized in BETA and has not been supported in the BETA compiler since the early eighties. Virtual classes in their general form as defined in this paper have been presented informally in the works on family polymorphism and higher-order hierarchies in gbeta [8, 9], delegation layers [28], and Caesar [23]. *vc* represents the core of these languages.

In gbeta, classes can have superclasses of the form path.C, which enables a new kind of dynamic composition that is not expressible in *vc*. However, we have analyzed the required extensions to *vc* in order to support this kind of inheritance, and based on the experience from gbeta it does not seem very hard, although it does introduce many new details in the rules and proofs (essentially, mixins must be on the form $\langle \mathsf{u} \rangle.\mathsf{p}$ rather than simply p, and each member in an object must have its own enclosing object). We expect to explore this extension in some future work. Delegation layers are more dynamic than *vc* in that they use object-based delegation instead of class-based inheritance, which enables polymorphic composition of types at runtime. It is also a natural part of our future work to create a version of *vc* building on delegation, but in this case it is not obvious how hard it is. In Caesar, virtual classes are combined with some aspect-oriented mechanisms which make the language very suitable for integrating independently-developed software components. As in *vc*, both Caesar and gbeta distinguish mutable variables from immutable fields and use this distinction during type checking.

Odersky et al have presented a calculus with path-dependent types called $\nu Obj$ [26]. The most important difference to $\nu Obj$ is that *vc* allows virtual classes whereas $\nu Obj$ focuses on virtual types only. This means that no objects can be created as an instance of a virtual type (abstract type member) and no implementation can be specified before the virtual type is final-bound to a concrete type. Although it is possible to create a class that has a virtual super-class in $\nu Obj$, this mechanism cannot express hierarchy specialization because the virtual superclass can only be replaced by a class that has *exactly* the same signature (e.g., does not add methods) [37]. Another difference is that *vc* has assignments, whereas $\nu Obj$ is purely functional. On the other hand, $\nu Obj$ is more powerful than *vc* w.r.t. the encoding of parametric polymorphism, which is not in the focus of this work. Finally, since our type-checker is completely syntax-directed (in particular, we have no subsumption rule), type-checking in *vc* is decidable, which is not the case for $\nu Obj$.

In [25], a language with nested inheritance is described, which has a number of similarities with virtual classes. An important difference to their approach is that they use classes in classes rather than classes in objects. The classes-in-classes model can trivially be simulated in a classes-in-objects model by using only one instance of each class containing virtual classes, but the converse does not hold—e.g., nested classes in [25] cannot access shared state of instances of enclosing classes. For example, in *vc* every nested class in Base and its subclasses can access the zero field declared in Figure 1. The expressive power of having access to the enclosing object is also illustrated by our straightforward encoding of methods by means of classes – accessing an instance variable foo of an object in a method bar is encoded as an access to the enclosing object **out**.foo in the corresponding class bar. Using nested inheritance, it would be possible to manually declare an instance variable, say 'enclosing', in each nested class and thus emulate the enclosing object, but it would require significantly more work to create and administrate such simulated enclosing objects, and it is not obvious that they could be given all the desired typing properties.

Another consequence is that a given program using nested inheritance has a fixed number of class families, whereas a given program in *vc* can have an unlimited number of distinct class families because every new family object contains a new class family. This enables a more fine-grained typing discipline in *vc*, because the type system will ensure that all these families are not mixed up. For example, this could be used to ensure that instances of Student nested in a given University are used only with the university from which they were obtained. With nested inheritance a simple instanceof test could reveal that all students were in fact members of the same class family, and hence the connection between a specific university and the associated students could not be expressed or enforced.

Family polymorphism by means of passing an instance of the enclosing class cannot be done directly in a classes-in-classes model. Instead, the authors of [25] propose the notion of *prefix types* to achieve a similar kind of polymorphism. Prefix types are a mechanism to refer to the (statically unknown) enclosing class of the class of an object. For example, A[b.class] denotes the enclosing class of the class of the object b.

The nested inheritance language itself is much bigger (and hence more complex) than our language. For example, there are seven different syntactic forms of type declarations and type schemas in [25], whereas the only form of type declaration is path.C in *vc*. Yet another difference is that different extensions to a class hierarchy cannot be combined in the nested inheritance language, as illustrated by our example in Figure 4. This is a consequence of the requirement in nested inheritance that the declared superclass of a class C must be a subtype of the inherited version of C, i.e., declared superclasses in redefinitions cannot be used to mix in additional features.

One feature which is well-known from related languages and calculi (including BETA and *$\nu$Obj*) is that of final-bindings. A virtual class/type may be final-bound, which means that the current value must remain unchanged (e.g., no additional mixins can be included). This feature is useful because it provides a lower bound on the value of a class, which opens more opportunities for assignments to variables having a given virtual class/type as their declared type. It would hence make sense to add final bindings to *vc* as well, but this extension is orthogonal to our work because our focus is on extensibility and not on genericity. Moreover, many years of experience with BETA seems to indicate that final bounds are not that important when initialized immutable fields are available, because such fields can be used to obtain a lower bound on all virtual classes in a given object. It is likely that the trade-off is different in languages like *$\nu$Obj* and Scala [27], because many details in the language design are different and closer to the functional paradigm.

There are a couple of other approaches that widen the expressibility of the static type system with respect to collaborating classes and parametric polymorphism but do not support incremental hierarchy specification [4, 33, 16].

Thorup proposes a virtual type system for Java [32]. It supports instantiation of a virtual class and hence late bound classes, but it does not support virtual superclasses. Furthermore, the type system relies on dynamic type checks.

There have been a couple of approaches for hierarchy refinement in the context of product lines (e.g., [2, 30]) but polymorphic usage of a hierarchy variant is not in the focus of these works. It will be interesting to explore how virtual classes improve the expressibility of languages with respect to product lines. Virtual classes are interesting from a software architecture point of view because they enable both incremental specification of class hierarchies and composition of different extensions to a class hierarchy, a problem that is hard to solve in conventional object-oriented languages. Hence, the language constructs in *vc* are well-suited to implement layered software architectures like mixin layers [30] or GenVoca [2].

Family classes used as argument types give rise to covariant typing, which is known to be non-trivial to handle in a type-safe manner. Other examples of a strict and safe treatment of covariance are the formalization of variant parametric types in [18], and the inclusion of wildcards into the J2SE 5 version of the Java platform [35]. Note, however, that virtual classes are different from variant parametric types or parametric types with wildcards, because those mechanisms do not support family polymorphism, but they provide a different kind of flexibility through structural equivalence among type applications.

The notion of having a first-class representation of a hierarchy is also highly relevant to the domain of aspect-oriented programming, which can be seen as an approach to have multiple cross-cutting decompositions (that is, hierarchies) of a system [31, 24].

The only prior work related to our coverage lemma that we know of is a paper by Fisher and Reppy [11]. They also improve on the traditional approach to proving type soundness for big-step semantics by differentiating diverging expressions from errors. They use an 'evaluation height function', whose definition is similar in structure to a small-step operational semantics, to count the number of steps during evaluation. Their soundness proof involves showing that a well-typed term with an evaluation height of $n$ will always evaluate to a value of the correct type. They *define* diverging programs as those for which the evaluation height function is undefined, but there is no proof that the evaluation height function correctly characterizes divergence of the operational semantics. In our technique, the correspondence between $\rightsquigarrow$ and $\rightsquigarrow_k$ is obvious by construction, all non-diverging programs have an evaluation tree because of the error rules, and missing rules are prevented due to the coverage lemma. Since Fisher and Reppy do not give full proofs, it is difficult to compare our techniques in detail.

## 9. Conclusions

We have presented the calculus *vc* of virtual classes with path-dependent types, described its dynamic and static semantics, and proved soundness. The approach to static analysis which was pioneered in BETA, made strict and complete in gbeta, and adapted for Java-like languages in Caesar has thereby been documented, clarified, and characterized as fundamentally sound. Our calculus has certain uniqueness requirements on declared names, but we have explained how these restrictions have been lifted in a full-fledged language at the cost of some extra complexity. All in all, we have hereby provided a foundation which shows that the widespread image of virtual classes as being inherently unsafe is too pessimistic.

**Acknowledgments**

We are very grateful to Sophia Drossopoulou and Christopher Anderson who have been involved in the process at an earlier stage,

and to Gary T. Leavens who helped us explaining several issues more clearly.

## References

[1] K. Barrett, B. Cassels, P. Haahr, D. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for Dylan. In *Proceedings OOPSLA '96*, pages 69–82. ACM Press, 1996.

[2] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5), 1994.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90. ACM SIGPLAN Notices 25(10)*, pages 303–311. ACM, 1990.

[4] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98. LNCS 1445*, pages 523–549. Springer, 1998.

[5] W. Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages*, LNCS 173. Springer-Verlag, 1990.

[6] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: FickleII. *ACM Transactions On Programming Languages and Systems*, 24(2):153–191, 2002.

[7] E. Ernst. Propagating class and method combination. In *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.

[8] E. Ernst. Family polymorphism. In *Proceedings ECOOP '01*, LNCS 2072, pages 303–326. Springer, 2001.

[9] E. Ernst. Higher-order hierarchies. In *Proceedings ECOOP '03*, LNCS. Springer, 2003.

[10] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. Technical report, University of Aarhus, Aarhus, Denmark, 2005.

[11] K. Fisher and J. Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Chicago, USA, 2003.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, 1999. Springer-Verlag.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[14] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[15] R. Harrejon, D. Batory, and W. R. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings ECOOP '05*. Springer, 2005.

[16] A. Igarashi and B. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2002.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[18] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of ECOOP '02*. Springer LNCS 2374, 2002.

[19] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of ECOOP '98*, LNCS 1445, 1998.

[20] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[21] O. L. Madsen and B. MÃ¸ller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89. ACM SIGPLAN Notices 24(10)*, pages 397–406, 1989.

[22] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02, ACM SIGPLAN Notices 37(11)*, pages 52–67, 2002.

[23] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings AOSD '03*, pages 90–99. ACM, 2003.

[24] M. Mezini and K. Ostermann. Modules for crosscutting models. In *International Conference on Reliable Software Technologies*. Springer LNCS 2655, 2003.

[25] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *Proceedings OOPSLA '04*, pages 99–115. ACM Press, 2004.

[26] M. Odersky, V. Cremet, C. RÃ¶ckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings ECOOP '03*. Springer LNCS, 2003.

[27] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings ACM SIGPLAN Conference on Object oriented programming systems languages and applications*, pages 41–57. ACM Press, 2005.

[28] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP '02. LNCS 2374*, pages 89–110. Springer, 2002.

[29] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[30] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98, LNCS 1445*, pages 550–570, 1998.

[31] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings International Conference on Software Engineering (ICSE) '99*, pages 107–119. ACM Press, 1999.

[32] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97. LNCS 1241*, pages 444–471, 1997.

[33] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *Proceedings ECOOP '99*, 1999.

[34] M. Torgersen. The expression problem revisited. In *European Conference on Object-Oriented Programming*, 2004.

[35] M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, Dec. 2004. `http://www.jot.fm/issues/issue_2004_12/article5`.

[36] P. Wadler. The expression problem. Message to java-genericity electronic mailing list, November 1998.

[37] M. Zenger. Personal communication, 2003.

[38] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, 2004.