

A Virtual Class Calculus

Erik Ernst

University of Aarhus, Denmark
eerst@daimi.au.dk

Klaus Ostermann

Darmstadt University 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*.

1. Introduction

Virtual classes are class-valued attributes of objects. They are analogous to virtual *methods* in traditional object-oriented languages: they follow the same 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 the traditional model of 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, 25], or hierarchies of classes that can be 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 redefine 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 [18]. Methods and classes are unified as *patterns* in BETA. Patterns can be virtual such that redefinition of methods can be expressed, but the unification of methods and classes gave rise to the notion of redefinition of classes, i.e., to virtual classes. Later languages, including Caesar [20, 21] and gbeta [7, 8, 9] have extended the concept of virtual classes while remaining essentially consistent with the informally specified model in BETA [19]. 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 allowed some potentially unsafe programs, while inserting 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 situation 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 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. We present a natural (big-step) semantics with support for assignment. The formal semantics can support introduction of virtual classes into mainstream object-oriented languages.
- Proof of the soundness of the type system.¹ We use a proof technique that was developed for natural semantics of object-oriented languages [6]. The subject reduction theorem ensures preservation of types under reduction: 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.

¹The proofs are given in the appendix as a convenience for reviewers who want to check details; the proofs are not part of the submission itself and will be provided in a separate technical report.

- We strengthen the traditional approach to soundness in natural 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 [26]: 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 [15] or ClassicJava [10]. 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 special 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.
- Types are described by paths to an enclosing object that can refer to outer objects and fields, and a class name.
- The types of arguments and 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 syntactic and semantic construct. This unification highlights the uniform treatment of virtual classes and (virtual) methods.

2.1 Higher-Order Hierarchies

Virtual classes provide an elegant solution to the *extensibility problem* [5, 17]: 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 [33, 31, 35]. We present a solution to a simplified version of a standardized problem definition [13].

In Figure 1, the class *Base* contains two virtual classes: a general class *Exp* representing numeric expressions and subclass *Lit* representing numeric literals. Virtual classes can be arbitrarily nested. All classes in *vc* are virtual classes. 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 Base { // contains two virtual classes
  class Exp {}
  class Lit extends Exp {
    int value;
  }
  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.

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* which is defined in the enclosing object of *Neg*. In general the type **out.A** in class *B* is a reference to the sibling *A* of *B*. Because of subclassing and late binding, the dynamic value of **out** may not be an instance of *WithNeg*; it may be an instance of a class that extends *WithNeg*. 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**. We will describe the encoding of methods in more detail in the next section. A test can be run by invoking `new WithNeg().TestNeg()`.

Redefinition of a virtual class occurs when it is declared and it is also defined in the outer class's superclass. In Figure 3, *Exp*

```

class Test {
  int Test(out.WithNeg e2, out.NegAndEval e4) {
    this.e2 = e2; this.e4 = e4;
    n = buildNeg(e2, n); // OK
    // n.eval(); -- Static error
    e4.zero = new e4.Lit(); // OK
    // n2 = buildNeg(e4, e2.zero) -- Static error
    n2 = buildNeg(e4, e4.zero); // OK
    n2.eval(); // OK
  }
  ne.Neg buildNeg(out.out.WithNeg ne, ne.Exp ex) {
    new ne.Neg(ex);
  }
  field out.WithNeg e2
  field out.NegAndEval e4
  e2.Exp n
  e4.Exp n2
}
new Test(new NegAndEval(), new NegAndEval())

```

Figure 5. Example of family polymorphism

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 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* [35].

The superclasses of virtual classes are combined and *linearized* to merge the multiple definitions that specify the superclasses of a virtual class. 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 has Exp as a declared superclass, after linearization it has WithEval.Exp as its immediate superclass.

2.2 Path-based Types

The example in Fig. 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 in Fig. 5

has type e2.Exp, meaning that only instances of Exp whose enclosing object is identical to the value of e2 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 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.

Although the resulting types may resemble Java package/class names, they are very different because classes play the role of packages. And as we have seen above, the “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. In this sense virtual classes are related to parameterized types, but due to the dynamic choice of family object it enables 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 [11] 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 Fig. 5, e2 and e4 inside Test are used as family objects. The constructor call in the last line of the example shows how e2 is polymorphically initialized with a subtype of its static types. The fact that the field e2 of class Test is declared to be an out.WithNeg, but the constructor is actually called with an argument of type NegAndEval, illustrates that an entire class hierarchy can be a first class value subject to subtype polymorphism via its family object, and the classes in this hierarchy 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 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: \bar{p} stands for p_1, \dots, p_k for some natural number $k \geq 0$. If $k = 0$ then the list is empty. The length of \bar{p} is $|\bar{p}|$. The same notation is used for lists whose elements are separated by dots or commas, e.g., $f_1.f_2.\dots.f_k = \bar{f}$. A list may also be represented by a combination of list barred and unbarred variables: $\bar{f}.f$ stands for $f_1.\dots.f_k.f$, where f denotes the last item of the list. Finally, we adopt the common use of $\overline{T \bar{f}}$ to represent a list of pairs $T_1 f_1 \dots T_k f_k$ rather than a pair of lists. An empty list

Grammar of vc

$$\begin{aligned}
 CL & ::= \mathbf{class} \ C \ \mathbf{extends} \ \overline{C} \ \{ \ K \ \overline{CL}; \ \overline{T} \ \overline{f}; \ \overline{T} \ \overline{v} \} \\
 K & ::= T \ C(\overline{T} \ \overline{f}) \ \{ e; \} \\
 T & ::= \mathit{path}.C \\
 \mathit{path} & ::= \mathit{spine}.\overline{f} \\
 \mathit{spine} & ::= \mathbf{this.out} \\
 e & ::= \mathbf{null} \mid e; e \mid \mathit{path} \mid \mathit{path}.v \mid \\
 & \quad \mathit{path}.v = e \mid \mathbf{new} \ \mathit{path}.C(\overline{e})
 \end{aligned}$$

Identifiers

class names	C
field names	f
variable names	v
members	$m = f \cup v$

($C, f,$ and v are pairwise disjoint)

Figure 6. Syntax of virtual class calculus vc

Metavariable

static paths $p ::= \overline{C}$

Class table

$$CT(p) = CT(p, CL_{root})$$

$$\frac{CL_i = \mathbf{class} \ C \ \mathbf{extends} \ \overline{C} \ \{ \dots \}}{CT(C, \overline{CL}) = CL_i}$$

$$\frac{CL_i = \mathbf{class} \ C \ \mathbf{extends} \ \overline{C} \ \{ \ K \ \overline{CL}' ; \dots \}}{CT(C.p, \overline{CL}) = CT(p, \overline{CL}')}$$

All members

$$Members(\mathit{nil}_p) = \mathit{nil}, \mathit{nil}$$

$$\frac{
 \begin{aligned}
 &Members(\overline{p}) = \overline{T} \ \overline{f}, \ \overline{T}' \ \overline{v} \\
 &CT(p) = \mathbf{class} \ C \ \mathbf{extends} \ \overline{C} \ \{ \ K \ \overline{CL}; \ \overline{T}'' \ \overline{f}'; \ \overline{T}''' \ \overline{v}' \} \\
 &Members(p \ \overline{p}) = \overline{T}'' \ \overline{f}' \ \overline{T} \ \overline{f}, \ \overline{T}''' \ \overline{v}' \ \overline{T}' \ \overline{v}
 \end{aligned}
 }{
 }$$

Constructor

$$\frac{CT(p) = \mathbf{class} \ C \ \mathbf{extends} \ \overline{C} \ \{ \ K \ \overline{CL}; \ \overline{T}'' \ \overline{f}'; \ \overline{T}''' \ \overline{v}' \}}{Constr(p) = K}$$

Figure 7. Auxiliary definitions

is written nil_x , where the metavariable x identifies the kind of items that the list should contain. The subscript x may be omitted if it is clear from context. Sometimes we use the notation $[f]$ to create a list with a single element f . Meta-variables corresponding to syntactic categories are written with a different font, e.g., the meta-variable path corresponds to the syntactic category path . 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 CL consist of a name, the names \overline{C} of class definitions being extended, a constructor K , a list of nested class definitions \overline{CL} , declarations $\overline{T} \ \overline{f}$ of immutable fields, and declarations $\overline{T} \ \overline{v}$ of mutable variables. A constructor K consists of a return type T , the definition name followed by formal parameters $\overline{T} \ \overline{f}$, and an expression

e . The constructor parameters must be identical to the fields of the class definition, including all the fields of all class definitions being extended. The constructor has a return type because it can return something different from 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— f for fields and v for variables. Field and variable names must be unique within the program in order to simplify the semantics of 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 this restriction. 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 spine , access to fields of the current or an enclosing object via path , access and assignment of variables, $\mathit{path}.v$, and $\mathit{path}.v = e$, and the null value, **null**. Method calls and object construction are unified in the expression $\mathbf{new} \ \mathit{path}.C(\overline{e})$.

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

Primitive types like **bool** and **int** are omitted; they just add complexity to the formalism without adding value.

3.3 Translating Informal Notation to vc

The translation of the informal language to the formal syntax of vc is straightforward. As in Java, constructors in the informal language do not specify a return type or return value, but these must be specified in vc . Similarly, 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 names of the fields. For a class definition C in the informal syntax, the constructor return type is always **out.C** and the return expression of a constructor is always **this**. Constructors are required in vc , while the informal syntax assumes a default constructor if no constructor is given.

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.

In the informal notation we omit **this** unless the resulting expression is empty. vc has no implicit scoping rules, hence all access to fields, variables, and classes must be disambiguated by an appropriate spine .

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:

$$T \ C(\overline{T} \ \overline{f}) \ \{ \ \overline{T} \ \overline{v}; \ e; \} \Rightarrow \mathbf{class} \ C \ \mathbf{extends} \ \{ \ K \ \mathit{nil}_{CL}; \ \overline{T} \ \overline{f}; \ \overline{T} \ \overline{v} \}$$

where $K = T \ C(\overline{T} \ \overline{f}) \ \{ e; \}$. Method calls are translated by prefixing them with the keyword **new**.

In our informal language we used more general expressions where the calculus only allows paths: $e.m$, $\mathbf{new} \ e.C(\overline{e})$, and $e.v = e'$. The general forms are translated into the calculus by rewriting $e.m$ as $\mathbf{new} \ \mathbf{this}.C'(e)$ where C' is a new local class with a field $T \ f$ where T is the type of e , and whose constructor returns $\mathbf{this}.f.m$. The translation is legal because the member is accessed through the new field. The other two constructs ($\mathbf{new} \ e.C(\overline{e})$, and $e.v = e'$) are handled similarly. The consequence of this is that the

$$\begin{aligned}
\iota_{\text{root}} &\mapsto \llbracket \perp \parallel C_{\text{root}} \parallel \rrbracket \\
\iota_1 &\mapsto \llbracket \iota_{\text{root}} \parallel \text{NegAndEval} \parallel \text{zero} : \text{null} \parallel \rrbracket \\
\iota_2 &\mapsto \llbracket \iota_{\text{root}} \parallel \text{NegAndEval} \parallel \text{zero} : \iota_5 \parallel \rrbracket \\
\iota_3 &\mapsto \llbracket \iota_{\text{root}} \parallel \text{Test} \parallel e2 : \iota_1 \ e4 : \iota_2 \ n : \iota_4 \ n2 : \iota_6 \parallel \rrbracket \\
\iota_4 &\mapsto \llbracket \iota_1 \parallel \text{Neg} \parallel e : \text{null} \parallel \rrbracket \\
\iota_5 &\mapsto \llbracket \iota_2 \parallel \text{Lit} \parallel \text{value} : 0 \parallel \rrbracket \\
\iota_6 &\mapsto \llbracket \iota_2 \parallel \text{Neg} \parallel e : \iota_5 \parallel \rrbracket
\end{aligned}$$

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

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* p is a list of class names \bar{C} .

The function CT looks up a class definition in the program. We assume the existence of a globally available program CL_{root} , which would otherwise embellish many relations and functions. In order to deal with the nesting hierarchy of classes, CT is a partial function from static paths to class definitions. CT finds the corresponding class definition for each class name in the static path while following the lexical nesting structure of the program. When applied to one argument, CT looks up static paths in the main program. For example, the class definition `Lit` inside `Base` in Figure 1 is referred to by the static path `Base.Lit`.

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

The function $Members$ collects all field and variable declarations found in a list of mixins \bar{p} . The function $Constr(p)$ returns the constructor of $CT(p)$ given a static path p .

4. Operational Semantics

The operational semantics is defined in big-step style. The formal definition is shown 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 ι , a class name 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 H from addresses ι to objects. The top-level root object has the special address ι_{root} . An example heap can be found in Fig. 9.

The behavior of the object is determined by the enclosing object ι and the class C . The enclosing object specifies the environment containing the class from which ι was created: an object ι with enclosing object ι' and class C must have been created by evaluating an expression equivalent to `new $\iota'.C(\dots)$` .

An object's behavior is defined by a list of mixins, or class bodies; these class bodies contain the declarations that are visible as members of the object. In vc these declarations all define virtual classes, but any number of them may act as methods of the object. The list of mixins of an object can be computed from the class name and the enclosing object.

4.2 Mixin Computation

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

The $Assemble$ function computes the mixin list for a class C relative to an enclosing mixin list \bar{p} . It calls $Defs$ to collect all the definitions of C located in any of the class bodies specified by \bar{p} . The notation used in the definition of $Defs$ means list comprehension as for example in Haskell. If the resulting list of mixins is empty then the class is not defined and $Assemble$ returns \perp . The result is a list of static paths that identifies all definitions of C contained in the list of enclosing mixins.

As an example, let us consider the computation of $Mix(H, \iota_4)$ in the program in Fig. 1-4 and the sample heap in Fig. 9. We assume we have already computed the mixin list \bar{p} of the enclosing object ι_1 , which is `Base WithEval WithNeg NegAndEval`. Then $Defs(\bar{p}, \text{Neg}) = \text{WithNeg.Neg NegAndEval.Neg}$.

The complete mixin list must also include the mixins of all the ancestors of these classes. To do so, $Assemble$ maps $Expand$ over this list of static paths, and linearizes the result. In applications of `map` we implicitly assume that the function that is mapped over the list is curried. $Expand$ assembles each of the superclasses of C , linearizes the result, and appends the class itself to the resulting list. In our example $\mathbf{map} \ Expand(\bar{p}) (\text{WithNeg.Neg NegAndEval.Neg}) = \bar{p}'\bar{p}''$, where $\bar{p}' = \text{Base.Exp WithEval.Exp WithNeg.Neg}$ and $\bar{p}'' = \text{NegAndEval.Neg}$.

Linearization is a technique with which an inheritance graph is sorted topologically, such that method calls can be dispatched along the sort order. The function $Linearize$ linearizes a list of mixin lists, i.e., it produces a single mixin list which contains the same mixins as those in the operands, in an order which is controlled by the operands. $Linearize$ is defined in terms of a binary linearization function, $Lin2$. 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 has been designed such that the ordering of mixins in a virtual class can be controlled by the programmer of a subclass, in a similar spirit as when the programmer of a subclass can decide to override a method in any mainstream object-oriented programming language, see [1, 7].

$Lin2$ 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 $Lin2$. The cases where C3 linearization fails are 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 $Mix(H, \iota_4)$ would be the mixin list `Base.Exp WithEval.Exp WithNeg.Neg NegAndEval.Neg`

It is easy to check that $Lin2$ is a total function on lists of mixins, and that the set of mixins in the result is equal to the union of the sets of mixins in the two operands. For soundness, only the set of mixins is relevant whereas the ordering makes no difference, and hence this generalization of C3 enhances the flexibility and expressive power of the language without affecting type safety.

4.3 Evaluation Rules and Error Handling

The evaluation relation $e, H, \iota \rightsquigarrow r, H'$ 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.

Objects and the Heap:

$$\begin{aligned} \text{Address} &= \text{natural numbers} & \iota \\ \text{Object} &= \{ \llbracket \iota \parallel C \parallel \bar{f} : \overline{\text{val}} \quad \nabla : \overline{\text{val}} \rrbracket \} & \llbracket \dots \rrbracket \\ \text{Heap} &= \text{Address} \xrightarrow{\text{fin}} \text{Object} & H \\ \text{Value} &= \text{Address} \cup \{ \text{null} \} & \text{val} \end{aligned}$$

Evaluation rules:

$\rightsquigarrow: e \times \text{Heap} \times \text{Address} \rightarrow \text{Value} \cup \{ \text{TypeErr}, \text{NullErr} \} \times \text{Heap}$

$$\text{null}, H, \iota \rightsquigarrow \text{null}, H \quad (R1) \quad \frac{\Downarrow H(\iota, \text{path}) = \text{val}}{\text{path}, H, \iota \rightsquigarrow \text{val}, H} \quad (R3)$$

$$\frac{e, H, \iota \rightsquigarrow \text{val}, H' \quad e', H', \iota \rightsquigarrow \text{val}', H''}{e; e', H, \iota \rightsquigarrow \text{val}', H''} \quad (R2) \quad \frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad H(\iota')(v) = \text{val}}{\text{path}.v, H, \iota \rightsquigarrow \text{val}, H} \quad (R4)$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad e, H, \iota \rightsquigarrow \text{val}, H' \quad H'(\iota')(v) \neq \perp \quad H'' = H'[\iota' \mapsto H'(\iota')[v \mapsto \text{val}]]}{\text{path}.v = e, H, \iota \rightsquigarrow \text{val}, H''} \quad (R5)$$

$$\frac{\begin{array}{l} \text{path}, H, \iota \rightsquigarrow \iota', H \quad H = H_1 \\ e_i, H_i, \iota \rightsquigarrow \text{val}_i, H_{i+1} \text{ for } i \in \{1 \dots |\bar{e}|\} \\ H' = H_{|\bar{e}|+1} \quad \bar{p} = \text{Assemble}(\text{Mix}(H', \iota'), C) \\ \text{Members}(\bar{p}) = \bar{T} \bar{f}, \bar{T}' \nabla \quad |\bar{f}| = |\overline{\text{val}}| \\ \iota'' \text{ is new in } H' \quad \text{Constr}(p|\bar{p}|) = T C(-)\{e';\} \\ H'' = H'[\iota'' \mapsto \llbracket \iota' \parallel C \parallel \bar{f} : \overline{\text{val}} \quad \nabla : \text{null} \rrbracket] \\ e', H'', \iota'' \rightsquigarrow \text{val}, H''' \end{array}}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{val}, H'''} \quad (R6)$$

Enclosing object:

$$\text{Encl}(\llbracket \iota \parallel - \parallel \dots \rrbracket) = \iota$$

Evaluation functions:

$$\begin{aligned} \Downarrow H(\iota, \text{this}) &= \iota \\ \Downarrow H(\iota, \text{spine.out}) &= \text{Encl}(H(\Downarrow H(\iota, \text{spine}))) \\ \Downarrow H(\iota, \text{path}.f) &= \text{val} \quad \text{where } H(\Downarrow H(\iota, \text{path}))(f) = \text{val}' \\ \Downarrow H(\iota, \text{path}.f) &= \text{NullErr} \quad \text{where } \Downarrow H(\iota, \text{path}) = \text{null} \\ \Downarrow H(\iota, \text{path}.f) &= \text{TypeErr} \quad \text{where } H(\Downarrow H(\iota, \text{path}))(f) = \perp \\ \Downarrow H(\iota, \text{spine.out}) &= \text{TypeErr} \quad \text{where } \Downarrow H(\iota, \text{spine}) = \iota_{\text{root}} \end{aligned}$$

Error handling:

$$\frac{\text{path}, H, \iota \rightsquigarrow \text{null}, H}{\text{path}.v, H, \iota \rightsquigarrow \text{NullErr}, H} \quad (ER1) \quad \frac{\text{path}.v = e, H, \iota \rightsquigarrow \text{NullErr}, H}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{NullErr}, H}$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad H(\iota')(v) = \perp}{\text{path}.v, H, \iota \rightsquigarrow \text{TypeErr}, H} \quad (ER2) \quad \frac{\text{path}.v = e, H, \iota \rightsquigarrow \text{TypeErr}, H}$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad \text{Assemble}(\text{Mix}(H, \iota'), C) = \perp}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{TypeErr}, H} \quad (ER3)$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad \text{Assemble}(\text{Mix}(H, \iota'), C) = \bar{p} \quad \text{Members}(\bar{p}) = \bar{T} \bar{f}, - \quad |\bar{e}| \neq |\bar{f}|}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{TypeErr}, H} \quad (ER4)$$

Mixin Computation:

$$\begin{aligned} \text{Mix}(H, \iota_{\text{root}}) &= [\text{nil}_C] \\ \text{Mix}(H, \iota) &= \text{Assemble}(\text{Mix}(H, \iota'), C) \\ &\text{where } H(\iota) = \llbracket \iota' \parallel C \parallel \dots \rrbracket \end{aligned}$$

$$\text{Assemble}(\bar{p}, C) = \text{Linearize}(\text{map Expand}(\bar{p}) \text{Defs}(\bar{p}, C))$$

$$\text{Defs}(\bar{p}, C) = \text{check}[p.C \mid p \leftarrow \bar{p}, CT(p.C) \neq \perp] \quad \text{where } \text{check}(\bar{p}) = \begin{cases} \perp & |\bar{p}| = 0 \\ \bar{p} & \text{otherwise} \end{cases}$$

$$\text{Expand}(\bar{p}, p) = \text{Lin2}(\text{Linearize}(\text{map Assemble}(\bar{p}) \bar{C}), p) \quad \text{where } CT(p) = \text{class } C \text{ extends } \bar{C} \{ \dots \}$$

$$\begin{aligned} \text{Linearize}(\text{nil}_{\bar{p}}) &= \text{nil}_p \\ \text{Linearize}(\bar{p} \bar{p}) &= \text{Lin2}(\text{Linearize}(\bar{p}), \bar{p}) \\ \text{Lin2}(\text{nil}_p, \text{nil}_p) &= \text{nil}_p \\ \text{Lin2}(\bar{p} p, \bar{p}' p) &= \text{Lin2}(\bar{p}, \bar{p}') p \\ \text{Lin2}(\bar{p}, \bar{p}' p') &= \text{Lin2}(\bar{p}, \bar{p}') p', \text{ if } p' \notin \bar{p} \\ \text{Lin2}(\bar{p} p, \bar{p}') &= \text{Lin2}(\bar{p}, \bar{p}') p, \text{ if } p \notin \bar{p}' \\ \text{Lin2}(\bar{p} p' \bar{p}' p, \bar{p}' p') &= \text{Lin2}(\bar{p} \bar{p}' p, \bar{p}') p' \end{aligned}$$

Figure 8. Operational semantics of *vc*

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

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

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

update of the variable v of that object, and $H[\iota \mapsto \dots]$ denotes heap update.

A new object **new path.C**(\bar{e}) is constructed (R6) by instantiating the virtual class C defined in the enclosing object ι' identified by path . The behavior \bar{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 Assemble returns \perp and rule (R6) does not apply. The mixin list \bar{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 ι'' bound to a new object with enclosing object ι' , 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 ι'' .

Two different kinds of error can occur during evaluation: Type errors and null pointer errors, denoted by `TypeErr` and `NullErr`, respectively. The rule (ER1) applies in the case we want to evaluate a property of an object, but the object is **null**. (ER2) and (ER3) define the situations in which a type error occurs, namely if either a member that we want to read or write is not available (ER2), or if we want to create an instance of a class `C`, but the enclosing object has no definition of `C` (that is, its mixin list is empty) or the number of parameters does not match (ER3).

The rules for propagating errors are standard and straightforward, so we omitted them from Figure 8 and just assume in the following that `NullErr` or `TypeErr` errors are propagated. The complete list of error rules are provided together with the proof of soundness.

5. Type System

The type system of `vc` uses nominal typing based on *paths* to objects containing virtual classes. A path identifies an object that has a specific virtual class. It is identified statically by traversing immutable fields from the current object **this** or one of its enclosing objects. The type system is defined in Figure 10. Types in the type system can have two different shapes: *object types* `u` of the form $\langle p \rangle.\bar{f}$, and *class types* `s` of the form $\langle p \rangle.\bar{f}.C$. An object type describes a type by a path to the object: If an expression has type $\langle p \rangle.\bar{f}$ in a typing context p' , then p is a prefix of p' , and the object denoted by the expression can be reached by going **out** ($|\bar{p}'| - |\bar{p}|$) steps and then following \bar{f} in the heap. A class type describes an object by a path to the *enclosing* object, and a class name. A class type is weaker than an object type in that every object type can be converted into a class type by means of the \mathcal{C} function but not vice versa. Every class type $\langle p \rangle.\bar{f}.C$ has the form `u.C` with $u = \langle p \rangle.\bar{f}$. This means that every class type `u.C` contains an object path `u` describing the enclosing object.

Typically, the type checker can compute 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 the object itself in the heap. However, there is always at least a path to the enclosing object in these cases, hence such expressions are typically assigned a class type.

5.1 Static Lookup and Conversion to Class Types

Static lookup, defined in the \mathcal{W} function, is at the core of the type system. It takes an object type `u` and a path `p` and produces an object type or a class type, if it succeeds. The intuition is that `u` is a description of a particular object and we want a description of the object that is reached from the object described by `u` following path in the heap. A naive approach would be to concatenate path to the path in `u`, but it would be hard to say whether such a concatenated path leads to the same object as another concatenated path. Instead, \mathcal{W} returns a *normalized* version of the combined path which has again the form of a type. These normalized paths (i.e., types) can be compared by simple equality tests.

For the empty path **this**, \mathcal{W} simply returns `u` (first case). For paths ending in **out**, the path to the enclosing object is returned: first the rest of the path is looked up, then it is converted to a class type by means of the \mathcal{C} function (explained below) and the last class name is removed (second case). Paths ending in a field or a class are checked for validity: an appropriate field or class must exist. The last branch in \mathcal{W} extends the domain of the second argument to T ; it is the only case where \mathcal{W} returns a class type and not an object type. For example, in Figure 2, $\mathcal{W}(\langle \text{WithNeg.Neg} \rangle.e, \text{this.out.Lit}) = \langle \text{WithNeg} \rangle.\text{Lit}$

Object types can be converted into a class type by means of the \mathcal{C} function as follows: If the object type is just a static path as starting point and no field accesses, then the enclosing object is described by the same static path with the last element removed (first case). If the object type ends with a field access, we replace the field access with the declared type of the field ($\mathcal{DclType}$ is explained below) and call \mathcal{W} to normalize again and get a type for it (second case). If the type is already a class type, we are done (third case).

5.2 Mixin Computation

The \mathcal{M} function is the static counterpart of the \mathcal{Mix} function. It 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: $[\text{nil}_c]$ (first case). If we have an object type `u.C`, then `u` is a type that describes the enclosing object, hence we can recursively compute the mixin list of the enclosing object. This mixin list and the class name `C` are sufficient to compute the mixin list for this type by calling the *Assemble* function (second case). Finally, if we need to compute the mixin list for an object type we convert it into a class type first (third case). For example, with the code in Fig. 1-5 we have $\mathcal{M}(\langle \text{Test} \rangle.e2.\text{Neg}) = \text{Base.Exp WithNeg.Neg}$ and $\mathcal{M}(\langle \text{Test} \rangle.e4.\text{Exp}) = \text{Base.Exp WithEval.Exp}$.

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

\mathcal{C} , \mathcal{W} , \mathcal{M} and $\mathcal{DclType}$ depend on each other in non-trivial ways, so it is not obvious that evaluation of these functions will terminate. A formal proof is in the appendix. Informally, the functions terminate because the arguments to recursive calls of \mathcal{W} inside \mathcal{W} and $\mathcal{DclType}$ 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. Since we compare class types, we know that we have object types for their respective enclosing objects. In (S-DECL) we insist on having exactly the same object type for the enclosing object for both parameters. Since an object type describes a path to an object we can be sure that the enclosing objects are indeed identical. This comparison for identical enclosing object types is useful only 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 `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 Sec. 5.1. As is obvious from the definition, paths have an object type. For variable lookup (T4) we use \mathcal{W} again, but this

Typing domains:

$$\begin{aligned} u &::= \langle p \rangle . \bar{f} & q &::= \mathbf{this} \mid \mathbf{out} \mid f \\ s &::= \langle p \rangle . \bar{f} . C & Q &::= \bar{q} \mid \bar{q} . C \\ t &::= u \mid s \end{aligned}$$

Expression Typing:

$$\frac{\mathcal{M}(t) \neq \perp}{p \vdash \mathbf{null} : t} \quad (T1) \quad \frac{\mathcal{W}(\langle p \rangle, \text{path}) = u}{p \vdash \text{path} : u} \quad (T3)$$

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

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

$$\frac{\begin{array}{l} p \vdash \text{path} : u \quad p' \in \mathcal{M}(u.C) \quad p \vdash \bar{e} : \bar{f} \\ \text{Constr}(p') = T_0 \mathcal{C}(\bar{T} \bar{f}) \dots \quad |\bar{T}| = |\bar{f}| \\ s_i = \begin{cases} \mathcal{W}(u_j, \mathbf{this}.Q) & \text{if } T_i = \mathbf{this}.f_j.Q \\ & \text{and } t_j = u_j \\ \mathcal{W}(u, \mathbf{this}.Q) & \text{if } T_i = \mathbf{this}.out.Q \end{cases} \\ \text{for } i = 0 \dots |\bar{f}| \\ \mathcal{C}(t_i) <: s_i \text{ for } i = 1 \dots |\bar{f}| \end{array}}{p \vdash \mathbf{new path}.C(\bar{e}) : s_0} \quad (T6)$$

Class types:

$$\begin{aligned} \mathcal{C} &:: t \rightarrow s \\ \mathcal{C}(\langle p \rangle . C) &= \langle p \rangle . C \\ \mathcal{C}(u.f) &= \mathcal{W}(u, \text{DeclType}(u, f)) \\ \mathcal{C}(s) &= s \end{aligned}$$

Mixins:

$$\begin{aligned} \mathcal{M} &:: t \rightarrow \bar{p} \\ \mathcal{M}(\langle \rangle) &= [\text{nil}_c] \\ \mathcal{M}(u.C) &= \text{Assemble}(\mathcal{M}(u), C) \\ \mathcal{M}(u) &= \mathcal{M}(C(u)) \end{aligned}$$

Enclosing object type:

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

Static lookup:

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

Program Typing:

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

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

$$\frac{\begin{array}{l} K = T \mathcal{C}(\bar{T}'' \bar{f}') \{e; \} \quad \mathcal{M}(\langle p \rangle . C) = \bar{p} \\ \text{Members}(\bar{p}) = \bar{T}'' \bar{f}', _ \\ p \vdash \bar{C} \text{ OK} \quad p.C \vdash \bar{T} \text{ OK} \quad p.C \vdash \bar{T}' \text{ OK} \quad p.C \vdash T \text{ OK} \\ p.C \vdash e : t \quad \mathcal{C}(t) <: \mathcal{W}(\langle p.C \rangle, T) \\ K' = \text{Constr}(p_j) \Rightarrow K \text{ overrides } K' \text{ OK} \end{array}}{p \vdash \mathbf{class } C \text{ extends } \bar{C} \{ K \bar{C}L; \bar{T} \bar{f}; \bar{T}' \bar{v}' \} \text{ OK}} \quad (\text{WF4})$$

There is a strict partial order \sqsubset_f on field names such that $\forall p, f. \text{spine}.\bar{f}.C \in \text{Members}(p) \Rightarrow \forall i. f_i \sqsubset_f f$

There is a strict partial order \sqsubset_c on class names such that $\forall p. CT(p) = \mathbf{class } C \text{ extends } \bar{C} \{ \dots \} \Rightarrow \forall i. C_i \sqsubset_c C$

CT is acyclic

$$\frac{\begin{array}{l} CT \text{ is acyclic} \\ \forall p, p', C : CT(p.C) \neq \perp, CT(p'.C) \neq \perp \Rightarrow \\ p''.C \in \mathcal{M}(\langle p \rangle . C) \cap \mathcal{M}(\langle p' \rangle . C) \\ \forall p \neq p' : CT(p) = \mathbf{class } C \dots \{ K \bar{C}L; \bar{T} \bar{f}; \bar{T}' \bar{v}' \} \\ CT(p') = \mathbf{class } C' \dots \{ K' \bar{C}'L; \bar{T}' \bar{f}'; \bar{T}'' \bar{v}'' \} \\ \Rightarrow \bar{f} \cap \bar{f}' = \emptyset, \bar{v} \cap \bar{v}' = \emptyset \\ \forall p, C : CT(p.C) \neq \perp \Rightarrow p \vdash CT(p.C) \text{ OK} \end{array}}{CT \text{ OK}} \quad (\text{WF6})$$

Subtyping:

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

$$\frac{\mathcal{M}(u) = \bar{p} \quad CT(p_j.C) = \mathbf{class } C \text{ extends } \dots C' \dots}{u.C <: u.C'} \quad (\text{S-DECL})$$

Declared type of member:

$$\begin{aligned} \text{DeclType}(t, m) &= T \text{ where } T m \in \text{Members}(\mathcal{M}(t)) \\ \text{Exists}(t, m) &= (\text{DeclType}(t, m) \neq \perp) \\ \text{Exists}(u, C) &= (\mathcal{M}(u.C) \neq \perp) \end{aligned}$$

Figure 10. Typing rules

time we pass the type of the variable instead of the variable name. This is a manifestation of the fact that variables cannot be used in types. This also means, however, that we do not get an object type but rather a class type for a variable access.

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 rhs is a subtype of the lhs. Since the type of the lhs is possibly an object type it needs to be converted to a class type first.

The rule for object creation (T6) is the most complex one, which is not surprising because it also has the role of method calls.

First, the type of the enclosing object u is computed. The statically known mixin structure of the new object, $\mathcal{M}(u.C)$, is computed, and a mixin is selected via the choice of p' , which is then used to find the signature of a constructor. 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.

In order to compare the syntactic types specified in the constructor with the types of the actual arguments, we compute class types

$$\begin{array}{l}
p \vdash \mathbf{this} : \langle \text{Test} \rangle. \quad \mathcal{M}(\langle \text{Test} \rangle.\text{buildNeg}) = \text{Test}.\text{buildNeg} \\
\text{Test} \vdash e4 : \langle \text{Test} \rangle.e4 \quad \text{Test} \vdash e4.\text{zero} : \langle \text{Test} \rangle.e4.\text{Lit} \\
\text{Constr}(\text{Test}.\text{buildNeg}) = \\
\text{ne.Neg buildNeg}(\text{out}.\text{out}.\text{WithNeg } \text{ne}, \text{ne}.\text{Exp } \text{ex}) \\
s_0 = \mathcal{W}(\langle \text{Test} \rangle.e4, \mathbf{this}.\text{Neg}) = \langle \text{Test} \rangle.e4.\text{Neg} \\
s_1 = \mathcal{W}(\langle \text{Test} \rangle., \mathbf{this}.\text{out}.\text{WithNeg}) = \langle \text{WithNeg} \rangle. \\
s_2 = \mathcal{W}(\langle \text{Test} \rangle.e4, \mathbf{this}.\text{Exp}) = \langle \text{Test} \rangle.e4.\text{Exp} \\
\mathcal{C}(\langle \text{Test} \rangle.e4) = \langle \text{NegAndEval} \rangle. <: s_1 \\
\mathcal{C}(\langle \text{Test} \rangle.e4.\text{Lit}) = \langle \text{Test} \rangle.e4.\text{Lit} <: s_2 \\
\hline
\text{Test} \vdash \mathbf{new } \mathbf{this}.\text{buildNeg}(e4, e4.\text{zero}) : \langle \text{Test} \rangle.e4.\text{Neg}
\end{array}$$

Figure 11. Illustration of the typing derivation for the buildNeg(e4,e4.zero) expression in Fig. 5

s_i for every syntactic type in the constructor, including the return type. Intuitively, the syntactic types T_i must be adapted to the *viewpoint* p . To do that, the static lookup function \mathcal{W} is used again. The types T_i are either of the form **this.out**.... or **this.f_j**.... The latter case applies if an argument type depends on the virtual class of another argument, as for example buildNeg in Fig. 5. TestLit in Fig. 1 is an example for the former case. Syntactically, T_i could also have the form **this.C'** for some class name C' , but this type would not be useful because it would refer to a virtual class of an object that does not yet exist. In the case $T_i = \mathbf{this.out}....$ we know that **this.out** is a reference to the enclosing object, for which we already have an object type u , hence we can start navigating into the tail of T_i starting at u using \mathcal{W} . In the case $T_i = \mathbf{this.f}_j....$ we can make use of the fact that we know that f_j will be initialized with the value of e_i at runtime, hence we can start the navigation at t_j . If an argument is used as type provider for a different argument, then the expression for the argument needs to have an object type, see restriction $t_j = u_j$ in (T6). Finally, it is checked that the argument types are subtypes of the formal argument types and the “viewpoint-adapted” constructor return type s_0 is returned.

Fig. 11 shows an example of a non-trivial usage of (T6) in the example from Fig. 5. It has been slightly adjusted to fit to the formal syntax, see Sec. 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 $e4$.

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), we require that declared names can be 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}

Well-formedness:

$$\begin{array}{l}
\frac{H(\iota)(m) = \mathbf{null}}{\iota.m : T \text{ OK in } H} \quad (\text{WF-NULL}) \\
\frac{H(\iota)(m) = \iota' \quad \Downarrow H(\iota', \mathbf{out}) = \Downarrow H(\iota, \text{path}) \quad p.C \in \text{Mix}(H, \iota')}{\iota.m : \text{path}.C \text{ OK in } H} \quad (\text{WF-MEM}) \\
\frac{T m \in \text{Members}(\text{Mix}(H, \iota)) \Rightarrow \iota.m : T \text{ OK in } H}{\iota \text{ OK in } H} \quad (\text{WF-OBJ})
\end{array}$$

$$\frac{H(\iota_{\text{root}}) = \llbracket \perp \parallel C_{\text{root}} \parallel \rrbracket}{\iota_{\text{root}} \text{ OK in } H} \quad (\text{WF-ROOT})$$

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

Agreement:

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

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

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

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

Auxiliary definitions:

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

Figure 12. Dynamic well-formedness and agreement

functions, which return \perp if the class or type, respectively, is not known to exist in the context p .

Note that (WF4) implies that fields can only be declared in new class declarations (i.e., there is no inherited class declaration with the same name); this restriction is not essential and we could easily add initialized fields (declared as $T f = e$) to the calculus which could be declared in all classes. (In fact, we developed the whole calculus with initialized fields before deciding to add constructors and letting fields be initialized via constructor arguments.) We chose 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 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.

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

wellformedness and agreement are given in Figure 12. Since the exact details of these definitions are not required in order 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.

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 m of object ι is well-formed if the member value $\iota' = H(\iota)(m)$ satisfies two conditions: (1) the enclosing object of the value, $\Downarrow H(\iota', \mathbf{out})$, is equal to the object $\Downarrow H(\iota, \text{path})$ specified by the path in the declared type path.C ; and (2) the mixins of the value, $\mathcal{M}ix(H, \iota')$, include a path ending with the class C . There is a special rule for wellformedness of the root object because it does not have an enclosing object.

Type agreement is specified as the agreement of an object at ι with a type T , relative to a dynamic heap H and a starting point ι_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, $\langle p \rangle.\bar{f}$. The rules ensure that the class path p is a prefix of the spine of ι_0 , so the value j represents the number of enclosing objects that must be traversed from ι_0 to read an object with the same depth as p . The path **this.out** $'.\bar{f}$ traverses to this object, and then traverses the field list \bar{f} . The object ι must be located at the end of this path. In addition, ι 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 path \bar{f} specified in the 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: *subject reduction* and *coverage*. Subject reduction 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.

Subject reduction 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 ι 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 . Subject Reduction also guarantees that the heap is still well-formed after the execution, and that the current object still agrees with its type.

THEOREM 1 (Subject Reduction).

$$\left[\begin{array}{l} CT \text{ OK} \\ H \text{ OK} \\ p \vdash e : t \\ H, \iota \vdash \iota \triangleright \langle p \rangle \\ e, H, \iota \rightsquigarrow r, H' \end{array} \right] \Rightarrow \left[\begin{array}{l} H' \text{ OK} \\ H', \iota \vdash \iota \triangleright \langle p \rangle \\ r = \text{val} \wedge H', \iota \vdash \text{val} \triangleright t \\ \vee \\ r = \text{NullErr} \end{array} \right]$$

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.

Subject reduction 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 [12]. 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 a evaluation can fail to produce a value is if the computation diverges. This Lemma plays a role similar to the ‘progress’ theorem when using 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:*

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

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, ι , there exists r, H' such that*

$$e, H, \iota \rightsquigarrow_n r, H'$$

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 exists 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 the appendix .

8. Related and Future Work

The idea of virtual classes and their different kinds of bindings stems from BETA [18]. 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 [25], and Caesar [21]. 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 it appears to be relatively easy to make several kinds of extensions of this nature (essentially, the mixins of an object must be a list of $\langle u \rangle.p$ rather than a list of p). 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.

Odersky et al have presented a calculus with path-dependent types called *νObj* [24]. The most important difference to *νObj* is that *vc* allows virtual classes whereas *νObj* focuses on virtual types only. This means that no objects can be created as an instance of a virtual type (an abstract type member) and no implementation can be specified before the virtual type is final-bound to a concrete type. Although it is technically possible to create a class that has a virtual super-class in *νObj*, 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) [34]. Another difference is that *vc* has assignments, whereas *νObj* is purely functional. On the other hand, *νObj* 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 *νObj*.

In [23], 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 [23] 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*. In general, not having the possibility to access shared state in the enclosing object is a serious restriction w.r.t. the expressiveness of the languages. It is not obvious whether and how this could be simulated in the language in [23].

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 instance of *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.

Moreover, family polymorphism by means of passing an instance of the enclosing class is of course not possible in a classes-in-classes model. Instead, the authors of [23] 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 [23], 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 *νObj*) 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.

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, 30, 14].

Thorup proposes a virtual type system for Java [29]. It supports the instantiation of a virtual class and hence late bound instantiation 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, 27]) 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 an interesting feature 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 particularly well-suited to implement layered software architectures like mixin layers [27] or *GenVoca* [2].

When family classes are used as argument types or return types, they give rise to covariant typing. Other examples of a strict and safe treatment of covariance are the formalization of variant parametric types in [16], and the inclusion of wildcards into the J2SE 5 version of the Java platform [32]. 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 [28, 22].

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, and hence the often heard claims that virtual classes are inherently unsafe should now dwindle.

Acknowledgments

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

References

- [1] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 69–82. ACM Press, 1996.
- [2] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genovoca 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*, volume 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] 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.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
- [12] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
- [13] R. Harrejon, D. Batory, and W. R. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings ECOOP '05*. Springer, 2005.
- [14] A. Igarashi and B. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2002.
- [15] 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.
- [16] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of ECOOP '02*. Springer LNCS 2374, 2002.
- [17] 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.
- [18] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89. ACM SIGPLAN Notices 24(10)*, pages 397–406, 1989.
- [19] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.
- [20] M. Mezini and K. Ostermann. Integrating independent components with on-demand modularization. In *Proceedings OOPSLA '02, ACM SIGPLAN Notices 37(11)*, pages 52–67, 2002.
- [21] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.
- [22] M. Mezini and K. Ostermann. Modules for crosscutting models. In *International Conference on Reliable Software Technologies*. Springer LNCS 2655, 2003.
- [23] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 99–115. ACM Press, 2004.
- [24] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings ECOOP '03*. Springer LNCS, 2003.
- [25] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP '02. LNCS 2374*, pages 89–110. Springer, 2002.
- [26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [27] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98, LNCS 1445*, pages 550–570, 1998.
- [28] 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.
- [29] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97. LNCS 1241*, pages 444–471, 1997.
- [30] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *Proceedings ECOOP '99*, 1999.
- [31] M. Torgersen. The expression problem revisited. In *European Conference on Object-Oriented Programming*, 2004.
- [32] M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. Gafer. 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.
- [33] P. Wadler. The expression problem. Message to java-genericity electronic mailing list, November 1998.
- [34] M. Zenger. Personal communication, 2003.
- [35] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, 2004.

A. Lemmas and proofs

In this section we present some formal results which characterize vc , along with proofs. First we show that the type analysis of vc is decidable. The remaining results are concerned with the soundness proof. These results are divided into three groups—results concerned with syntactic entities; results concerned with the static information about the heap; and results concerned with both the static and the dynamic heap, including subject reduction. Notationally, we use some small marks to make internal references in certain proofs more convenient and precise. In particular, result number 1 would be marked like ¹:this, and references to it are shown as (1). Similarly, statements which must be proved can be marked like [?]:this, with references shown as (?2).

A.1 About the Decidability of Typing

The static analysis of vc is decidable, because the type rules are syntax directed and because the auxiliary functions are computed by directly specified, terminating algorithms. The only non-trivial point is that the partial ordering \sqsubseteq_f of field names required in program well-formedness is needed in order to show that the computation of \mathcal{C} , \mathcal{M} , and \mathcal{W} always terminates for all well-formed programs.

To show this we can assume without loss of generality that all field names in use are on the form f_i where the index i respects the partial ordering, in the sense that $f_i \sqsubseteq_f f_{i+1}$ for all i . Then define:

DEFINITION 2 (*Weight*). The weight of a concatenated path Q , $Weight(Q)$, is a function from natural numbers to natural numbers

that counts the number of fields in \mathcal{Q} with each index,

$$\begin{aligned} \text{Weight}(\text{nil})(-) &= 0 \\ \text{Weight}(\bar{q}.q) &= \text{Weight}(\bar{q}) + \text{Weight}(q) \\ \text{Weight}(\bar{q}.C) &= \text{Weight}(\bar{q}) \\ \text{Weight}(f_i)(k) &= \begin{cases} 1, & \text{if } k = i \\ 0, & \text{otherwise} \end{cases} \\ \text{Weight}(q)(-) &= 0, \text{ if } q \in \{\text{this}, \text{out}\} \end{aligned}$$

Concatenated path weights are totally ordered as follows: If w_1 and w_2 are concatenated path weights then $w_1 < w_2$ iff there is an n_0 such that

$$(\forall n > n_0. w_1(n) = w_2(n)) \quad \wedge \quad (w_1(n_0) < w_2(n_0))$$

So the weight of a concatenated path is a histogram of its fields which maps all other numbers than the ones which are field indices to zero; elements other than fields are ignored. It is easy to see that the defined ordering of weights is indeed a total order and that the all-zero weight is minimal.

A.1.1 Termination of \mathcal{C} , \mathcal{M} , and \mathcal{W}

To see termination of these functions, use induction based on a pair which is the weight of the arguments and the length of the argument, so for the argument list $(\langle p \rangle.\bar{f}.Q)$ or $(\langle p \rangle.\bar{f}.\bar{q}.C')$, the weight will be $(\text{Weight}(\bar{f}.\bar{q}), |p.\bar{f}.\bar{q}|)$. Let these values be ordered as follows: if $w' < w$ then $(w', s') < (w, s)$, and if $s' < s$ then $(w, s') < (w, s)$. Using this measure it is easy to see that all (mutually recursive) invocations of \mathcal{C} , \mathcal{M} , and \mathcal{W} by the same functions are made using strictly smaller arguments. For the innermost invocations this fact can be established directly by checking the form of the arguments; for the invocations where a returned result is given as an argument, as in $\mathcal{C}(\mathcal{W}(u, \text{path}))$, we need to use the fact that \mathcal{W} and \mathcal{C} will return a result which is at most as large as the given arguments, and \mathcal{C} returns a strictly smaller result when given an object type as argument.

A.2 Syntax Related Results

First we need to establish some simple properties involving only syntactic entities.

DEFINITION 3 (Well-defined static path). A static path p is well-defined iff the indicated classes are present in the given program, i.e., if $CT(p)$ is defined.

A static path unambiguously identifies a syntactic class body if and only if it is well-defined, and all static paths in use must be well-defined.

DEFINITION 4 (Homogeneous \bar{p}). A list of static paths, \bar{p} , is homogeneous iff all its elements have the same length, i.e., $\forall i, j \in \{1 \dots |\bar{p}|\}. |p_i| = |p_j|$.

We later show that all mixin lists provided by the static semantics are homogeneous, which is a natural consequence of using a model where each object has only one enclosing object. We also need an auxiliary concept of being a syntactic subclass:

DEFINITION 5 (Syntactic subclass). C is a direct syntactic subclass of C' in a list of mixins \bar{p} , written $\bar{p} \vdash C :< C'$, iff for some j , $CT(p_j.C) = \text{class } C \text{ extends } \dots C' \dots \{ \dots \}$. The reflexive and transitive closure is denoted syntactic subclass and written with a star as in $\bar{p} \vdash C :<^* C'$.

LEMMA 2 (Basic properties of *Assemble*). If *Assemble*(\bar{p}, C) is defined then the result is a non-empty list. If *Assemble*(\bar{p}, C) = \bar{p}' then \bar{p} are prefixes of \bar{p}' , i.e., for $p' \in \bar{p}'$ there is a $p \in \bar{p}$ and a C' such that $p' = p.C'$. Moreover, $\bar{p} \vdash C :<^* C'$. If all static paths in \bar{p} are well-defined then all static paths in \bar{p}' are well-defined, too. Finally, if \bar{p} is homogeneous then \bar{p}' is homogeneous, too.

Proof: Easy induction in the definitions of *Assemble*, *Expand*, *Defs*, *Linearize*, and *Lin2*. \square

We sometimes need to consider a list of mixins as a set of mixins, which just implies that we ignore the ordering and possible duplicates in the list. For conciseness we do not show this conversion explicitly, but it is applied whenever a list of mixins is used in a context that requires a set, e.g., in expressions like $\bar{p} \cup \bar{p}'$.

LEMMA 3 (Set properties of *Assemble* aux. functions). With implicit conversion of each list into the set of elements in the list whenever a set is required, the following relations hold:

1. $\text{Lin2}(\bar{p}, \bar{p}') = \bar{p} \cup \bar{p}'$
2. $\text{Linearize}(\bar{p}) = \bigcup \bar{p}_i$
3. $\bar{p} \subseteq \bar{p}' \wedge \text{Defs}(\bar{p}, C) \neq \perp \Rightarrow \text{Defs}(\bar{p}, C) \subseteq \text{Defs}(\bar{p}', C)$
4. $\bar{p} \subseteq \bar{p}' \wedge \text{Expand}(\bar{p}, p) \neq \perp \Rightarrow \text{Expand}(\bar{p}, p) \subseteq \text{Expand}(\bar{p}', p)$

Proof: Easy inductions and usage of the definitions of *Defs*, *Expand*, *Linearize*, and *Lin2*. \square

LEMMA 4 (Monotonicity of *Assemble*). If *Assemble*(\bar{p}, C) $\neq \perp$ and $\bar{p} \subseteq \bar{p}'$ then *Assemble*(\bar{p}, C) \subseteq *Assemble*(\bar{p}', C). If *Assemble*(\bar{p}, C') $\neq \perp$, *Assemble*(\bar{p}, C) $\neq \perp$, and $\bar{p} \vdash C' :<^* C$ then *Assemble*(\bar{p}, C') \supseteq *Assemble*(\bar{p}, C).

Proof: For the first part of the lemma, assume that *Assemble*(\bar{p}, C) $\neq \perp$. The definition of *Assemble* then shows that *Defs*(\bar{p}, C) is defined and that *Expand*(\bar{p}, p') is defined for each $p' \in \text{Defs}(\bar{p}, C)$. Lemma 3 then yields *Assemble*(\bar{p}, C) \subseteq *Assemble*(\bar{p}', C) by monotonicity of all functions involved. The second part is shown by induction in the number of direct syntactic subclass steps involved in $\bar{p} \vdash C' :<^* C$. In the base case there are zero steps and $C = C'$ which makes the result immediate. For the induction step, assume that *Assemble*(\bar{p}, C) $\neq \perp$ and $\bar{p} \vdash C' :<^* C$ because $\bar{p} \vdash C' :< C''$ and $\bar{p} \vdash C'' :<^* C$, then there is a $p' \in \bar{p}$ such that $CT(p'.C') = \text{class } C' \text{ extends } \bar{C} \{ \dots \}$, and $C'' = C_j$ for some j . This means that $p'.C' \in \text{Defs}(\bar{p}, C')$, and then, implicitly converting lists to sets and using Lemma 3 as well as various function definitions,

$$\begin{aligned} \text{Assemble}(\bar{p}, C') &= \\ \text{Linearize}(\mathbf{map} \text{Expand}(\bar{p}) \text{Defs}(\bar{p}, C')) &\supseteq \\ \text{Expand}(\bar{p}, p'.C') &= \\ \text{Lin2}(\text{Linearize}(\mathbf{map} \text{Assemble}(\bar{p}) \bar{C}), p'.C') &\supseteq \\ \text{Linearize}(\mathbf{map} \text{Assemble}(\bar{p}) \bar{C}) &\supseteq \text{Assemble}(\bar{p}, C'') \end{aligned}$$

By the induction hypothesis, *Assemble*(\bar{p}, C'') \supseteq *Assemble*(\bar{p}, C), hence *Assemble*(\bar{p}, C') \supseteq *Assemble*(\bar{p}, C) as required. \square

In short, *Assemble*(\bar{p}, C) appends C or a syntactic superclass to some of \bar{p} , preserves well-definedness and homogeneity in \bar{p} , co-varies with \bar{p} , and contra-varies with C .

A.3 Results Involving the Static Heap

The static semantic entities mimic the dynamic entities to a large extent. We make this connection more explicit here, by defining some auxiliary functions that produce and investigate static objects, i.e., values on the form $\llbracket u \parallel C \parallel \bar{p} \rrbracket$, denoted by the symbol so . Here, u is a type that describes the enclosing object, C is the statically known class of the object, and \bar{p} is the statically known list of mixins of the object. The type of the enclosing object corresponds directly to the enclosing object which is the first component of an object in the dynamic heap, and C corresponds to (but need not be the same as) the second component of the object; the list of mixins defines the features of the object by its members, so the last component of the static object also corresponds to the last component of the dynamic object, although this connection is less direct.

DEFINITION 6 (Static heap). We define the static heap, \mathcal{H} , functions to extract the enclosing object, \mathcal{E} , and the class, Cls_s , of a static object, and a function that computes the depth of a static object.

- $\mathcal{H}(t) = \begin{cases} \llbracket \perp \parallel C_{root} \parallel \rrbracket & \text{if } t = \langle \rangle \\ \llbracket u \parallel C \parallel \bar{p} \rrbracket & \text{if } \mathcal{C}(t) = u.C \text{ and } \mathcal{M}(t) = \bar{p} \end{cases}$
- $Depth_s(t) = \begin{cases} 0, & \text{if } t = \langle \rangle. \\ 1 + Depth_s(u) & \text{if } \mathcal{C}(t) = u.C \end{cases}$
- $\mathcal{E}(t) = u$, if $\mathcal{C}(t) = u.C$
- $Cls_s(t) = C$, if $\mathcal{C}(t) = u.C$

LEMMA 5 (The static heap). The static heap \mathcal{H} is a well-defined, partial function. If \bar{p} is well-defined, $t = \langle p \rangle.Q$, and $\mathcal{H}(t)$ is defined then computing $\mathcal{H}(t)$ will only involve well-defined static paths. Finally, if $\mathcal{H}(t) = \llbracket t' \parallel C \parallel \bar{p} \rrbracket$ then t' is an object type, $\exists p' : p'.C \in \bar{p}$, and \bar{p} is homogeneous.

A well-defined partial function is a relation that relates at most one value in the range to each value in the domain.

Proof: An easy induction in the definitions of the relevant functions, using that *Assemble* by Lemma 2 preserves well-definedness, and that prefixes of well-defined paths are themselves well-defined. \square

Next, we establish that the syntactic subclass relation contra-varies with the corresponding mixin sets, that syntactic subclass is implied by subtype, and hence that the subtype relation contra-varies with the corresponding mixin sets. This connection is the motivation for having the notion of syntactic subclass.

LEMMA 6. If $\mathcal{M}(u) = \bar{p}$, $\mathcal{M}(u.C') = \bar{p}'$, $\mathcal{M}(u.C'') = \bar{p}''$, and $\bar{p} \vdash C' :<^* C''$, then $\bar{p}' \supseteq \bar{p}''$.

Proof: Follows directly from the definition of \mathcal{H} , \mathcal{M} , and Lemma 4. \square

LEMMA 7. If $s' <: s$ then there is an object type u and classes C and C' such that $s' = u.C'$, $s = u.C$, and $\mathcal{M}(u) \vdash C' :<^* C$.

Proof: Easy induction in the proof of $s' <: s$. \square

LEMMA 8 (Subtype implies more static mixins). If $s' <: s$ then $\mathcal{E}(s') = \mathcal{E}(s)$ and $\mathcal{M}(s) \neq \perp \Rightarrow \mathcal{M}(s') \supseteq \mathcal{M}(s)$.

Proof: By Lemma 7 there exist u , C , and C' such that $s' = u.C'$ and $s = u.C$, so $\mathcal{E}(s') = u = \mathcal{E}(s)$. Lemma 7 also yields $\mathcal{M}(u) \vdash C' :<^* C$. By (S-DECL) $\mathcal{M}(u)$ is defined and there is a $\bar{p} \in \mathcal{M}(u)$ such that $CT(\bar{p}.C')$ is defined, so $\mathcal{M}(u.C')$ is also defined. Assuming that $\mathcal{M}(s)$ is defined we can use Lemma 6 which yields $\mathcal{M}(u.C') \supseteq \mathcal{M}(u.C)$, as required. \square

The following lemmas show that the relation in the static setting between the mixin sets of an object and that of its enclosing object is the same as in the dynamic heap, and a similar correspondence exists for the depth of an object and for the enclosing object of a field or variable.

LEMMA 9 (Static heap respects mixins). If $\mathcal{H}(t) = \llbracket u \parallel C \parallel \bar{p} \rrbracket$ then $\bar{p} = Assemble(\mathcal{M}(u), C)$.

Proof: By the definition of \mathcal{H} , $\bar{p} = \mathcal{M}(t)$ and $\mathcal{C}(t) = u.C$. The definition of \mathcal{C} and \mathcal{M} shows that $\forall t : \mathcal{M}(t) = \mathcal{M}(\mathcal{C}(t))$, so $\bar{p} = \mathcal{M}(t) = \mathcal{M}(\mathcal{C}(t)) = \mathcal{M}(u.C) = Assemble(\mathcal{M}(u), C)$. \square

LEMMA 10 (Static heap respects depth). If $\bar{p} \in \mathcal{M}(t)$ then $|\bar{p}| = Depth_s(t)$.

Proof: Induction in the computation of $\mathcal{M}(t)$.

Case ($\mathcal{M}(\langle \rangle) = [nil_c]$): Trivial.

Case ($\mathcal{M}(u.C) = Assemble(\mathcal{M}(u), C)$): By the induction hypothesis, for any $\bar{p}' \in \mathcal{M}(u)$ we have $|\bar{p}'| = Depth_s(u)$. By Lemma 2, for any $\bar{p} \in Assemble(\mathcal{M}(u), C)$, $|\bar{p}| = |\bar{p}'| + 1 = Depth_s(u.C)$.

Case ($\mathcal{M}(u) = \mathcal{M}(\mathcal{C}(u))$): Assume $\bar{p} \in \mathcal{M}(u)$, then also $\bar{p} \in \mathcal{M}(\mathcal{C}(u))$. By the induction hypothesis, $|\bar{p}| = Depth_s(\mathcal{C}(u))$. But $Depth_s(u) = Depth_s(\mathcal{C}(u))$ because $\mathcal{C}(\mathcal{C}(u)) = \mathcal{C}(u)$, so $|\bar{p}| = Depth_s(u)$. \square

We now introduce a low-level version of \mathcal{W} which is a device to prove some properties of \mathcal{W} .

DEFINITION 7. The definition of \mathcal{N} is as follows:

$$\begin{aligned} \mathcal{N}(u.\mathbf{this}.Q) &= \mathcal{N}(u.Q) \\ \mathcal{N}(u.f.\mathbf{out}.Q) &= \mathcal{N}(u.DeclPath(u, f).Q) \\ \mathcal{N}(\langle p.C \rangle.\mathbf{out}.Q) &= \mathcal{N}(\langle p \rangle.Q) \\ \mathcal{N}(t) &= t \end{aligned}$$

We now show that \mathcal{N} terminates. Every step in the computation of \mathcal{N} preserves that the second argument is a concatenated path (i.e., on the form Q), and that it decreases the weight of the concatenated path, or leaves the weight unchanged but shortens the path. Because the length of the concatenated path is finite the latter kind of steps cannot occur an infinite number of times in sequence, so the process will always terminate.

Note that \mathcal{W} is just a ‘safe version’ of \mathcal{N} —it performs checks to ensure that each transformation is well-defined, but otherwise it delegates the real work to \mathcal{N} .

LEMMA 11 (\mathcal{W} and \mathcal{N}). If $\mathcal{W}(u, \text{path}.C^?) = t$ then $\mathcal{N}(u, \text{path}.C^?) = t$. Moreover, $\mathcal{N}(u, \bar{q}.\bar{q}') = \mathcal{N}(\mathcal{N}(u, \bar{q}).\bar{q}')$.

Proof: Easy inductions in the shape of the function arguments.

LEMMA 12 (Static heap is enclosing-correct). If $\mathcal{W}(u, \text{path}) = u'$ and $DeclPath(u', f) = \text{path}'$ then $\mathcal{W}(u', \text{path}') = \mathcal{E}(\mathcal{W}(u, \text{path}.f))$.

Proof: Assume that $\mathcal{W}(u, \text{path}) = u'$ and $DeclPath(u', f) = \text{path}'$. This implies that there is a C such that $DeclType(u', f) = \text{path}'.C$, hence $Exists(u', f)$, so $\mathcal{W}(u, \text{path}.f)$ is defined, and it is easy to see that $\mathcal{W}(u, \text{path}.f) = u'.f$. The definition of \mathcal{C} now yields $\mathcal{C}(u'.f) = \mathcal{N}(u'.\text{path}'.C) = \mathcal{N}(u'.\text{path}')$. Finally we use Lemma 11 again: $\mathcal{E}(\mathcal{W}(u, \text{path}.f)) = \mathcal{E}(\mathcal{W}(u, \text{path}).f) = \mathcal{E}(u'.f) = \mathcal{N}(u'.\text{path}') = \mathcal{W}(u', \text{path}')$. \square

We shall need one more result which shows that we can “shift” a step from one path to another.

LEMMA 13 (Shifting a static step). Assume $\mathcal{E}(u) = u'$ and $\mathcal{W}(u, \text{spine.out}.\bar{f}) = u''$, then $\mathcal{W}(u', \text{spine}.\bar{f}) = u''$.

Proof: By induction in the shape of the path $\text{spine.out}.\bar{f}$.

Case (**this.out**): Assume that $\mathcal{E}(u) = u'$ and $\mathcal{W}(u, \mathbf{this.out}) = u''$, then $\mathcal{C}(\mathcal{W}(u, \mathbf{this})) = u''.C$ for some C . Since $\mathcal{W}(u, \mathbf{this}) = u$ we get $u'' = \mathcal{E}(u) = u'$, hence $\mathcal{W}(u', \mathbf{this}) = u''$.

Case (**spine.out.out**): Assume that $\mathcal{E}(u) = u'$ and $\mathcal{W}(u, \text{spine.out.out}) = u''$, then $\mathcal{C}(\mathcal{W}(u, \text{spine.out})) = u''.C$ for some C . Let $u''' = \mathcal{W}(u, \text{spine.out})$. Note that $u''' = \mathcal{E}(u''')$. By the induction hypothesis, $\mathcal{W}(u', \text{spine}) = u'''$, so $\mathcal{C}(\mathcal{W}(u', \text{spine})) = u''.C$, from which we conclude that $\mathcal{W}(u', \text{spine.out}) = u''$.

Case (**spine.out.f**): Assume that $\mathcal{E}(u) = u'$ and $\mathcal{W}(u, \text{spine.out.f}) = u''$, then $u'' = u'''.f$ where

$u''' = \mathcal{W}(u, \text{spine.out}.\bar{f})$ and $\text{DeclType}(u''', f) \neq \perp$.
By the induction hypothesis, $\mathcal{W}(u', \text{spine}.\bar{f}) = u'''$, so
 $\mathcal{W}(u', \text{spine}.\bar{f}.f) = u'''.f = u''$, as required. \square

A.4 Results Involving Both Heaps

First we need to introduce a couple of auxiliary functions and a new concept of heap compatibility.

DEFINITION 8 (Dynamic inspection functions). *If $H(\iota) = \llbracket \iota' \parallel C \parallel _ \rrbracket$ then $\text{Encl}(H(\iota)) = \iota'$ and $\text{Cls}(H(\iota)) = C$.*

DEFINITION 9 (Heap compatibility). *H' is compatible with H iff H' is defined in at least all those ι where H is defined, and for each ι where both H and H' are defined, $H'(\iota)$ differs from $H(\iota)$ at most in the values of variables.*

To support the intuition behind this concept, note that the class of an object, the enclosing object, and the objects accessible through its fields are significant for the static analysis, whereas the values of variables may change freely as long as the declared types are respected. This reflects the fact that types may depend on the values of fields and the enclosing object, but not on the values of variables. The next two results show that evaluation preserves compatibility.

LEMMA 14 (Immutability of objects). *If $H(\iota) = \llbracket \iota' \parallel C \parallel \bar{f} : \text{val} \dots \rrbracket$ and $e, H, _ \rightsquigarrow _$, H' , then ι is defined in H' , and $H'(\iota) = \llbracket \iota' \parallel C \parallel \bar{f} : \text{val} \dots \rrbracket$.*

Proof: Easy induction in the proof tree for the evaluation. \square

COROLLARY 15 (Evaluation yields compat. heap). *If $e, H, _ \rightsquigarrow _$, H' then H' is compatible with H .*

The next lemma is also easy, but it is important for the static analysis that paths never change, because they are used in types.

LEMMA 16 (Values of paths are immutable). *If $\Downarrow H(\iota, \text{path}) = \iota'$ and H' compatible with H then $\Downarrow H'(\iota, \text{path}) = \iota'$.*

Proof: Since path has the form **this.out**. \bar{f} , which means that only enclosing objects and fields are evaluated, the Lemma follows directly from Lemma 14. \square

LEMMA 17 (Evaluation from enclosing). *If $\Downarrow H(\iota, \text{spine.out}.\bar{f}) = \iota'$ then $\text{Encl}(H(\iota)) \neq \perp$ and $\Downarrow H(\text{Encl}(H(\iota)), \text{spine}.\bar{f}) = \iota'$. If $\Downarrow H(\iota, \text{spine}.\bar{f}) = \iota'$ and $\text{Encl}(H(\iota')) = \iota$ then $\Downarrow H(\iota', \text{spine.out}.\bar{f}) = \iota'$.*

Proof: The first part is an easy induction in the shape of path, using the cases **this.out**, **spine.out.out**, and **spine.out**. $\bar{f}.f$ because these cases inductively describe all the possible paths on the form **spine.out**. \bar{f} . The second part is an easy induction in the shape of path based on the cases **this**, **spine.out**, and **spine**. $\bar{f}.f$, which inductively describes all shapes of path, but allows for insertion of **outin** in the desired position. \square

The next lemma shows various properties about agreement, including that it is sufficient to ensure the syntactic subclass relation in the agreement rules because the desired mixin relation follows, and that nesting preserves agreement:

LEMMA 18 (Agreement). *Assume that CT OK, H OK, and $H, \iota_0 \vdash \iota \triangleright t$. Then*

1. $\text{Depth}(H, \iota) = \text{Depth}_s(t)$
2. If $\text{Mix}(H, \text{Encl}(H(\iota))) = \bar{p}$ or $\iota = \iota_{\text{root}}$ then $\bar{p} \vdash \text{Cls}(H(\iota)) \text{ :<}^* \text{Cls}_s(t)$
3. If $\text{Depth}_s(t) > 0$ then $H, \iota_0 \vdash \text{Encl}(H(\iota)) \triangleright \mathcal{E}(t)$

4. $\text{Mix}(H, \iota) \supseteq \mathcal{M}(t)$
5. If $\iota_0 = \text{Encl}(H(\iota_1))$ then $H, \iota_1 \vdash \iota \triangleright t$
6. If $\mathcal{C}(t) <: s$ and $\mathcal{M}(s) \neq \perp$ then $H, \iota_0 \vdash \iota \triangleright s$

Proof: To enable concise references to assumptions we number them as follows: ¹:CT OK, ²:H OK, and ³: $H, \iota_0 \vdash \iota \triangleright t$.

1. Induction in the proof of (3).

Case (A-NULL): Not applicable (ι cannot be **null**).

Case (A-ROOT): Trivial.

Case (A-OTYPE): In this case $H, \iota_0 \vdash \iota \triangleright \mathcal{C}(\langle p \rangle.\bar{f})$ where $\langle p \rangle.\bar{f} = t$. Note that $\langle p \rangle.\bar{f} \neq \langle \rangle$ because $\mathcal{C}(\langle p \rangle.\bar{f})$ is defined, and $\mathcal{C}(\langle p \rangle.\bar{f}) \neq \langle \rangle$ because $\langle \rangle$ is not a class type. Then

$$\begin{aligned} \text{Depth}(H, \iota) &= \\ &\quad // \text{by the induction hypothesis} \\ \text{Depth}_s(\mathcal{C}(\langle p \rangle.\bar{f})) &= \\ &\quad // \text{by def. of Depth}_s \\ 1 + \text{Depth}_s(\mathcal{E}(\mathcal{C}(\langle p \rangle.\bar{f}))) &= \\ &\quad // \mathcal{E}(\mathcal{C}(\langle p \rangle.\bar{f})) = \mathcal{E}(\langle p \rangle.\bar{f}) \\ 1 + \text{Depth}_s(\mathcal{E}(\langle p \rangle.\bar{f})) &= \\ &\quad // \text{by def. of Depth}_s \\ \text{Depth}_s(\langle p \rangle.\bar{f}). & \end{aligned}$$

Case (A-CATYPE): Here, $H, \iota_0 \vdash \text{Encl}(H(\iota)) \triangleright u$, where $t = u.C$. Note that $\iota \neq \iota_{\text{root}}$ because $\text{Encl}(H(\iota))$ is defined. Then

$$\begin{aligned} \text{Depth}(H, \iota) &= \\ &\quad // \text{by def. of Depth} \\ 1 + \text{Depth}(H, \text{Encl}(H(\iota))) &= \\ &\quad // \text{by the induction hypothesis} \\ 1 + \text{Depth}_s(u) &= \\ &\quad // u = \mathcal{E}(u.C) \\ 1 + \text{Depth}_s(\mathcal{E}(u.C)) &= \\ &\quad // \text{by def. of Depth}_s \\ \text{Depth}_s(u.C). & \end{aligned}$$

2. Induction in the proof of (3).

Case (A-NULL): Not applicable ($\iota \neq \text{null}$).

Case (A-ROOT): In this case $\iota = \iota_{\text{root}}$ and $t = \langle \rangle$. Hence $\text{Cls}_s(t) = C_{\text{root}}$, and by H OK, $\text{Cls}(H(\iota)) = C_{\text{root}}$, which shows that $\bar{p} \vdash \text{Cls}(H(\iota)) \text{ :<}^* \text{Cls}_s(t)$ for arbitrary \bar{p} .

Case (A-OTYPE): We have $H, \iota_0 \vdash \iota \triangleright \mathcal{C}(\langle p \rangle.\bar{f})$ where $\langle p \rangle.\bar{f} = t$. By the induction hypothesis $\bar{p} \vdash \text{Cls}(H(\iota)) \text{ :<}^* \text{Cls}_s(\mathcal{C}(\langle p \rangle.\bar{f}))$, and the desired result then follows from $\text{Cls}_s(\mathcal{C}(\langle p \rangle.\bar{f})) = \text{Cls}_s(\langle p \rangle.\bar{f})$.

Case (A-CATYPE): In this case $t = u.C$ such that $C = \text{Cls}_s(t)$, and there exists a p' such that $p'.C \in \text{Mix}(H, \iota)$, and $H, \iota_0 \vdash \text{Encl}(H(\iota)) \triangleright u$. From H OK we get ι OK in H, and this must have been shown using (WF-OBJ) because $\iota \neq \iota_{\text{root}}$, because $\text{Encl}(H(\iota))$ is defined. Moreover, since $\iota \neq \iota_{\text{root}}$ we must also have $\bar{p} = \text{Mix}(H, \text{Encl}(H(\iota)))$. From the definition of Mix we get $\text{Mix}(H, \iota) = \text{Assemble}(\bar{p}, \text{Cls}(H(\iota)))$, and $\bar{p} \vdash \text{Cls}(H(\iota)) \text{ :<}^* C$ then follows from Lemma 2, which concludes the case.

3. If t is a class type, $t = u.C$, then (3) by (A-CATYPE) yields $H, \iota_0 \vdash \text{Encl}(H(\iota)) \triangleright u$ and since $u = \mathcal{E}(t)$ we are done. Otherwise t is an object type, so from (A-OTYPE) we get $H, \iota_0 \vdash \iota \triangleright \mathcal{C}(t)$, hence by the class type case $H, \iota_0 \vdash \text{Encl}(H(\iota)) \triangleright \mathcal{E}(\mathcal{C}(t))$, and the result then follows from $\mathcal{E}(\mathcal{C}(t)) = \mathcal{E}(t)$.
4. Induction in $\text{Depth}_s(t)$.

Case ($Depth_s(t) = 0$): The definition of $Depth_s$ shows by an easy induction that $t = \langle \rangle$ when $Depth_s(t) = 0$. Moreover by 1., $Depth(H, \iota) = 0$, so $\iota = \iota_{root}$ by a similar induction. The result then follows immediately from the definitions of $Mix(H, \iota_{root})$ and $\mathcal{M}(\langle \rangle)$.

Case ($Depth_s(t) = k + 1$):

$$\begin{aligned}
\mathcal{M}(t) &= \\
&\text{// by Lemma 9} \\
Assemble(\mathcal{M}(\mathcal{E}(t)), Cls_s(t)) &\subseteq \\
&\text{// by 3., the ind.hyp., and Lemma 4} \\
Assemble(Mix(H, Encl(H(\iota))), Cls_s(t)) &\subseteq \\
&\text{// by 2. and Lemma 4} \\
Assemble(Mix(H, Encl(H(\iota))), Cls(H(\iota))) &= \\
&\text{// definition of Mix} \\
Mix(H, \iota) &
\end{aligned}$$

5. Induction in $Depth_s(t)$.

Case ($Depth_s(t) = 0$): As in the proof of 4., $t = \langle \rangle$ and $\iota = \iota_{root}$, so the result follows immediately from (A-ROOT).

Case ($Depth_s(t) = k + 1$): We must consider class types and object types separately.

- $t = u.C$: By (3) and (A-CSTYPE) there is a p' such that $p'.C \in Mix(H, \iota)$, and $H, \iota_0 \vdash Encl(H(\iota)) \triangleright u$. The induction hypothesis yields $H, \iota_1 \vdash Encl(H(\iota)) \triangleright u$, hence by (A-CSTYPE) $H, \iota_1 \vdash \iota \triangleright t$ as required.
- $t = \langle p \rangle.\bar{f}$: By (3) and (A-OTYPE), $j = Depth(H, \iota_0) - |p|$, $\Downarrow H(\iota_0, \mathbf{this.out}^j.\bar{f}) = \iota$, and $H, \iota_0 \vdash \iota \triangleright C(\langle p \rangle.\bar{f})$. But then with $j' = j + 1$, $j' = Depth(H, \iota_1) - |p|$, by Lemma 17 $\Downarrow H(\iota_1, \mathbf{this.out}^{j'}.\bar{f}) = \iota$, and by the previous case $H, \iota_1 \vdash \iota \triangleright C(\langle p \rangle.\bar{f})$, which by (A-OTYPE) yields $H, \iota_1 \vdash \iota \triangleright t$.

6. We need to consider class types and object types separately.

Case ($t = s'$): Assume $\mathcal{C}(s') <: s$, i.e., ${}^4:s' <: s$, and ${}^5:\mathcal{M}(s) \neq \perp$. Let ${}^6:C = Cls_s(s)$. Using (1) (2) (3) with part 4. of this lemma yields ${}^7:Mix(H, \iota) \supseteq \mathcal{M}(s')$. From (4) (5) with Lemma 8 we get ${}^8:\mathcal{M}(s') \supseteq \mathcal{M}(s)$ and ${}^9:\mathcal{E}(s') = \mathcal{E}(s)$. Using (6) (7) and the definition of \mathcal{M} then shows that ${}^{10}:\mathcal{M}(s) = Assemble(\mathcal{M}(\mathcal{E}(s)), C)$. An inspection of the definition of $Assemble$ shows that for all \bar{p}'' and C' where $Assemble(\bar{p}'', C') = \bar{p}'''$, $\exists p'. p'.C' \in \bar{p}'''$. Noting that $\mathcal{M}(\iota_{root})$ is the list of lenght one containing the element nil_c (rather than the empty list), it is easy to see that mixin lists from the static heap are never empty, so in this case we get $\exists p'. p'.C \in \mathcal{M}(s)$, and hence from (7) (8) that ${}^{11}:\exists p'. p'.C \in Mix(H, \iota)$. Moreover, (3) via (A-CSTYPE) yields ${}^{12}:H, \iota_0 \vdash Encl(H(\iota)) \triangleright \mathcal{E}(s')$. Finally, using (9) (12) we get ${}^{13}:H, \iota_0 \vdash Encl(H(\iota)) \triangleright \mathcal{E}(s)$, and then from (11) (13) via (A-CSTYPE) that $H, \iota_0 \vdash \iota \triangleright s$, as required.

Case ($t=u$): Assume $\mathcal{C}(u) <: s$, then from $\mathcal{C}(\mathcal{C}(u)) = \mathcal{C}(u)$ we also have ${}^4:\mathcal{C}(\mathcal{C}(u)) <: s$. From (3) by (A-OTYPE) we get ${}^5:H, \iota_0 \vdash \iota \triangleright \mathcal{C}(u)$. Now we can use (1) (2) (5) (4) with this lemma again because it matches the class type case for which the proof is given above, yielding $H, \iota_0 \vdash \iota \triangleright s$ as required. \square

Agreement can sometimes be established from heap soundness alone, namely with respect to the point of view of a class body which corresponds to one of the mixins of the given object. We use the phrase *local view type* to denote such a type because it is a view upon the object as seen from itself.

LEMMA 19 (Agreement with local view types). *Assume CT OK, H OK, and $Mix(H, \iota) = \bar{p}$, then for any i : $H, \iota \vdash \iota \triangleright \langle p_i \rangle$.*

Proof: By induction in $Depth(H, \iota)$.

Case ($Depth(H, \iota)=0$): In this case $\iota = \iota_{root}$, so $\bar{p} = p_i = nil_c$, so we just need to show that $H, \iota_{root} \vdash \iota_{root} \triangleright \langle \rangle$, which follows directly from (A-ROOT).

Case ($Depth(H, \iota)=k+1$): We must prove that $H, \iota \vdash \iota \triangleright \langle p_i \rangle$ using (A-OTYPE), because $\langle p_i \rangle$ is an object type and (A-ROOT) does not apply. By part 1 of this lemma, $Depth_s(\langle p_i \rangle) = k + 1$, so there exists C_i such that $p_i = C_1 \dots C_{k+1}$. Let $j = Depth(H, \iota) - |C_1 \dots C_{k+1}| = 0$, and note that $\Downarrow H(\iota, \mathbf{this}) = \iota$, which establishes the two first premises for (A-OTYPE).

For the last premise of (A-OTYPE) note that $\mathcal{C}(\langle C_1 \dots C_{k+1} \rangle) = \langle C_1 \dots C_k \rangle.C_{k+1}$, so we need to show $H, \iota \vdash \iota \triangleright \langle C_1 \dots C_k \rangle.C_{k+1}$. In this case we must use (A-CSTYPE) because $\langle C_1 \dots C_k \rangle.C_{k+1}$ is a class type. For the first premise of (A-CSTYPE) we let $p' = C_1 \dots C_k$, such that $p'.C_{k+1} = p_i \in \bar{p} = Mix(H, \iota)$. Finally we need to show that $H, \iota \vdash Encl(H(\iota)) \triangleright \langle C_1 \dots C_k \rangle$. By the definition of Mix , $Mix(H, \iota) = Assemble(Mix(H, Encl(H(\iota))), Cls(H(\iota)))$. From $C_1 \dots C_{k+1} \in Mix(H, \iota)$ and Lemma 2 we conclude $C_1 \dots C_k \in Mix(H, Encl(H(\iota)))$. The induction hypothesis then provides $H, Encl(H(\iota)) \vdash Encl(H(\iota)) \triangleright \langle C_1 \dots C_k \rangle$ and then part 5 of this lemma finishes the case. \square

Agreement does not depend on the values of object members, which makes it a very persistent property.

LEMMA 20 (Agreement is persistent). *Assume CT OK, H OK, H' OK, H' compatible with H, and $H, \iota_0 \vdash \iota \triangleright t$. Then $H', \iota_0 \vdash \iota \triangleright t$.*

Proof: By induction in $Depth(H, \iota)$.

Case ($Depth(H, \iota)=0$): In this case $\iota = \iota_{root}$ and $t = \langle \rangle$, so we just need to show that $H', \iota_0 \vdash \iota_{root} \triangleright \langle \rangle$, which follows directly from (A-ROOT).

Case ($Depth(H, \iota) = k + 1, t = u.C$): From $H, \iota_0 \vdash \iota \triangleright u.C$ by (A-CSTYPE), there is a p' such that $p'.C \in Mix(H, \iota)$, and $H, \iota_0 \vdash Encl(H(\iota)) \triangleright u$. Since H' is compatible with H , $Mix(H', \iota) = Mix(H, \iota)$, so $p'.C \in Mix(H', \iota)$, too. By the induction hypothesis we get $H', \iota_0 \vdash Encl(H(\iota)) \triangleright u$, so $Encl(H'(\iota)) = Encl(H(\iota))$ finishes the case.

Case ($Depth(H, \iota) = k + 1, t = \langle p \rangle.\bar{f}$): From $H, \iota_0 \vdash \iota \triangleright \langle p \rangle.\bar{f}$ by (A-OTYPE), $j = Depth(H, \iota_0) - |p|$, $\Downarrow H(\iota_0, \mathbf{this.out}^j.\bar{f}) = \iota$, and $H, \iota_0 \vdash \iota \triangleright C(\langle p \rangle.\bar{f})$. But then also $j = Depth(H', \iota_0) - |p|$ because heap compatibility ensures unchanged enclosing objects, and by Lemma 16, $\Downarrow H'(\iota_0, \mathbf{this.out}^j.\bar{f}) = \iota$. Finally, the previous case shows $H', \iota_0 \vdash \iota \triangleright C(\langle p \rangle.\bar{f})$, which yields $H', \iota_0 \vdash \iota \triangleright \langle p \rangle.\bar{f}$, as required. \square

Finally, we shall need the following result which shows that agreement can imply the existence of a path between objects.

LEMMA 21 (Agreement implies path). *If CT OK, H OK, $H, \iota_0 \vdash \iota \triangleright u$, and $H, \iota_0 \vdash \iota' \triangleright \mathcal{W}(u, \text{path})$, then $\Downarrow H(\iota, \text{path}) = \iota'$.*

Proof: Assume ${}^1:CT$ OK, ${}^2:H$ OK, ${}^3:H, \iota_0 \vdash \iota \triangleright u$, and ${}^4:H, \iota_0 \vdash \iota' \triangleright \mathcal{W}(u, \text{path})$. Let $u = \langle p \rangle.\bar{f}$ and $\text{path} = \mathbf{this.out}^k.\bar{f}'$. The proof then proceeds by induction in k .

Case (0): An easy induction in $|\bar{f}|$ shows that $\mathcal{W}(u, \text{path}) = \langle p \rangle.\bar{f}.\bar{f}'$. From (3) (4) with (A-OTYPE) we get ${}^5:\Downarrow H(\iota_0, \mathbf{this.out}^j.\bar{f}) = \iota$ where $j = Depth(H, \iota_0) - |p|$, and ${}^6:\Downarrow H(\iota_0, \mathbf{this.out}^j.\bar{f}.\bar{f}') = \iota'$. But then there exist ι_i such that $H(\iota_i)(\bar{f}'_i) = \iota_{i+1}$ for $i \in \{1..n\}$ where $\iota_n = \iota'$ and (by (5))

$\iota_1 = \iota$, which we can directly use to show $\Downarrow H(\iota_0, \mathbf{this}.\bar{f}') = \iota'$, as required.

Case (k+1): Since $\mathcal{W}(u, \mathbf{this.out}^{k+1}.\bar{f}')$ is defined, $\mathcal{W}(u, \mathbf{this.out}^{k+1})$ is also defined, so we must have $u \neq \langle \rangle$ and hence ${}^7: \text{Depth}_s(u) > 0$. Now from (1) (2) (3) (7) with Lemma 18.3, ${}^8: H, \iota_0 \vdash \text{Encl}(H(\iota)) \triangleright \mathcal{E}(u)$. By Lemma 13, ${}^9: \mathcal{W}(u, \mathbf{this.out}^{k+1}.\bar{f}') = \mathcal{W}(\mathcal{E}(u), \mathbf{this.out}^k.\bar{f}')$. Application of the induction hypothesis to (1) (2) (8) (4) (9) yields ${}^{10}: \Downarrow H(\text{Encl}(H(\iota)), \mathbf{this.out}^k.\bar{f}') = \iota'$. Finally from (10) with Lemma 17 we get $\Downarrow H(\iota, \mathbf{this.out}^{k+1}.\bar{f}') = \iota'$, as required. \square

The next few lemmas establish correspondences between the static and the dynamic world. First we show that a member predicted by static analysis will also exist at run-time, then we show that agreement is preserved in some important cases, and finally we show that object creation and assignment preserve heap well-formedness.

LEMMA 22 (Memory Lookup Succeeds). *Assume that CT OK, H OK, $H, \iota_0 \vdash \iota \triangleright u$, and $\text{Exists}(u, m)$. Then $H(\iota)(m) \neq \perp$.*

Proof: By the definition of Exists , $\text{DclType}(u, m) = t' \neq \perp$, so there is a T such that $T m \in \text{Members}(p)$ for some $p = \mathcal{M}(u)$. By Lemma 18.4, $\text{Mix}(H, \iota) \supseteq \mathcal{M}(u)$, so $p \in \text{Mix}(H, \iota)$, too. By H OK and ι OK in H which uses (WF-OBJ) because $\iota \neq \iota_{\text{root}}$ since ι_{root} has no members, $H(\iota)(m)$ is either **null** or ι' , so $H(\iota)(m) \neq \perp$. \square

LEMMA 23 (Path lookup preserves agreement). *If CT OK, H OK, $H, \iota_0 \vdash \iota \triangleright u$, $\Downarrow H(\iota, \text{path}) = \text{val}$, and $\mathcal{W}(u, \text{path}) = u'$, then $H, \iota_0 \vdash \text{val} \triangleright u'$.*

Proof: If $\text{val} = \mathbf{null}$ then the result is trivial. Otherwise $\text{val} = \iota'$. For easy reference to assumptions we number them as follows: ${}^1: \text{CT OK}$, ${}^2: \text{H OK}$, ${}^3: H, \iota_0 \vdash \iota \triangleright u$, ${}^4: \Downarrow H(\iota, \text{path}) = \iota'$, and ${}^5: \mathcal{W}(u, \text{path}) = u' = \langle p \rangle.\bar{f}$. We now prove by induction in $\text{Weight}(\bar{f}.\text{path})$ that $H, \iota_0 \vdash \iota' \triangleright u'$.

Case ($\bar{f} = \text{nil}_f$, $\text{path} = \mathbf{this.out}^k$): In this case $u = \langle C_1 \dots, C_n \rangle$ so by direct evaluation of \mathcal{W} we conclude that $u' = \langle C_1 \dots C_{n-k} \rangle$ where $n \geq k$, and then by (R4) that $\iota' = (\text{Encl} \circ H)^k(\iota)$, and by direct evaluation that $u' = (\mathcal{E} \circ H_s)^k(u)$. Using this and the fact that $\text{Depth}_s(u) = n$ with Lemma 18.3 k times we get $H, \iota_0 \vdash \iota' \triangleright u'$, as required.

Case ($\text{path} = \text{spine}.\bar{f}.f$): By the definition of $\Downarrow H$ there is a ι'' such that ${}^6: \Downarrow H(\iota, \text{spine}.\bar{f}) = \iota''$ and ${}^7: H(\iota'')(f) = \iota'$. By the definition of \mathcal{W} there is a $u'' = \langle p'' \rangle.\bar{f}'$ such that ${}^8: \mathcal{W}(u, \text{spine}.\bar{f}) = u''$, ${}^9: \text{Exists}(u'', f)$, and ${}^{10}: u' = u''.f = \langle p'' \rangle.\bar{f}' . f$. Applying the induction hypothesis to (1) (2) (3) (6) (8) yields ${}^{11}: H, \iota_0 \vdash \iota'' \triangleright u''$. From (11) we get ${}^{12}: j'' = \text{Depth}(H, \iota_0) - |p''|$, and from (9) we conclude that there exist path' and C such that ${}^{13}: \text{path}' . C f \in \text{Members}(\mathcal{M}(u''))$. Using (1) (2) (11) with Lemma 18.4 yields $\text{Mix}(H, \iota'') \supseteq \mathcal{M}(u'')$, which with (13) shows that $\text{path}' . C f \in \text{Members}(\text{Mix}(H, \iota''))$, and then (2) (7) shows that ${}^{14}: \Downarrow H(\iota'', \text{path}') = \text{Encl}(H(\iota''))$, and there is a p' such that ${}^{15}: p' . C \in \text{Mix}(H, \iota')$. Now use (8) (13) with Lemma 12 to get ${}^{16}: \mathcal{W}(u'', \text{path}') = \mathcal{E}(u')$. We apply the induction hypothesis to (1) (2) (11) (14) (16) which produces $H, \iota_0 \vdash \text{Encl}(H(\iota')) \triangleright \mathcal{E}(u')$. Noting that $\mathcal{E}(u') = \mathcal{E}(C(u'))$, this and (15) with (A-CTYPE) yields ${}^{17}: H, \iota_0 \vdash \iota' \triangleright C(u')$. From (11) we get $\Downarrow H(\iota_0, \mathbf{this.out}^{j''}.\bar{f}'') = \iota''$, which with (7) yields ${}^{18}: \Downarrow H(\iota_0, \mathbf{this.out}^{j''}.\bar{f}' . f) = \iota'$. Finally, using (12) (18) (17) with (A-OTYPE) we obtain $H, \iota_0 \vdash \iota' \triangleright u'$, as required. \square

LEMMA 24 (Variable lookup preserves agreement). *If CT OK, H OK, $H, \iota_0 \vdash \iota \triangleright u$, $\mathcal{W}(u, \text{DclType}(u, v)) = s$, and $H(\iota)(v) = \text{val}$, then $H, \iota_0 \vdash \text{val} \triangleright s$.*

Proof: If $\text{val} = \mathbf{null}$ then the result is trivial. Otherwise let $\text{val} = \iota'$ and assume the following: ${}^1: \text{CT OK}$, ${}^2: \text{H OK}$, ${}^3: H, \iota_0 \vdash \iota \triangleright u$, ${}^4: \mathcal{W}(u, \text{DclType}(u, v)) = s = u'.C$, and ${}^5: H(\iota)(v) = \iota'$. By the definition of \mathcal{W} and using (4) there exists a path such that ${}^6: \text{DclType}(u, v) = \text{path}.C$, hence ${}^7: \mathcal{W}(u, \text{path}) = u'$. By (1) (2) (3), ${}^8: \text{Mix}(H, \iota) \supseteq \mathcal{M}(u)$, which shows that $\text{path}.C v \in \text{Members}(\text{Mix}(H, \iota))$, so by (2) (5) we get ${}^9: \Downarrow H(\iota, \text{path}) = \text{Encl}(H(\iota'))$ and there exists p' such that ${}^{10}: p'.C \in \text{Mix}(H, \iota')$. Now we can use (1) (2) (3) (9) (7) with Lemma 23 to get ${}^{11}: H, \iota_0 \vdash \text{Encl}(H(\iota')) \triangleright u'$, and finally (10) (11) with (A-CTYPE) yields $H, \iota_0 \vdash \iota' \triangleright u'.C$, as required. \square

LEMMA 25 (Heap update preserves well-formedness). *Assume that CT OK, H OK, $H, \iota_0 \vdash \iota \triangleright u$, $H, \iota_0 \vdash \text{val} \triangleright t$, and $C(t) <: \mathcal{W}(u, \text{DclType}(u, v))$. Then $H[\iota \mapsto H(\iota)[v \mapsto \text{val}]] \text{OK}$.*

Proof: Let $H' = H[\iota \mapsto H(\iota)[v \mapsto \text{val}]]$. It is obvious that H' differs from H only in that $H'(\iota)$ maps v to val rather than to its previous value in H , so we only need to consider ι OK in H' , and only for the member v . If the new value val is **null** then the result is trivial; so assume this is not the case and let $\text{val} = \iota'$. Assume ${}^1: \text{CT OK}$, ${}^2: \text{H OK}$, ${}^3: H, \iota_0 \vdash \iota \triangleright u$, ${}^4: H, \iota_0 \vdash \iota' \triangleright t$, and ${}^5: C(t) <: \mathcal{W}(u, \text{DclType}(u, v))$. From (5) we conclude that $\text{DclType}(u, v)$ is defined, let $\text{DclType}(u, v) = \text{path}.C$. By the definition of \mathcal{W} , when $\mathcal{W}(u, \text{path}.C) = u'.C$ is defined also $u' = \mathcal{W}(u, \text{path})$ is defined, and so is $\mathcal{M}(u'.C)$. Hence, we can use (1) (2) (4) (5) with Lemma 18.6 to obtain ${}^6: H, \iota_0 \vdash \iota' \triangleright \mathcal{W}(u, \text{path}).C$. Note that $\text{Depth}_s(\mathcal{W}(u, \text{path}).C) > 0$ because it is a class type, and $\mathcal{E}(\mathcal{W}(u, \text{path}).C) = \mathcal{W}(u, \text{path})$. With (1) (2) (6) using Lemma 18.3, this yields ${}^7: H, \iota_0 \vdash \text{Encl}(H(\iota')) \triangleright \mathcal{W}(u, \text{path})$, and then (1) (2) (3) (7) with Lemma 21 yields ${}^8: \Downarrow H(\iota, \text{path}) = \text{Encl}(H(\iota'))$. From (6) via (A-CTYPE) we now obtain that there is a p' such that $p'.C \in \text{Mix}(H, \iota')$, and by Lemma 14 and Corollary 15 via the definition of Mix we conclude ${}^9: p'.C \in \text{Mix}(H', \iota')$. Finally, since $H'(\iota)(v) = \iota'$, (8) (9) with (WF-MEM) shows that $\iota.v: \text{path}.C \text{ OK}$ in H' , as required. \square

LEMMA 26 (Object creation preserves well-formedness).

Assume CT OK, H OK, $H, \iota_0 \vdash \iota \triangleright u$, $H, \iota_0 \vdash \bar{\text{val}} \triangleright \bar{t}$, $\bar{p} = \text{Assemble}(\text{Mix}(H, \iota), C)$, $\text{Members}(\bar{p}) = \bar{T} \bar{f}, \bar{T}' \bar{v}$, $|\bar{f}| = |\bar{\text{val}}|$, ι' new in H ,

$$s_i = \begin{cases} \mathcal{W}(u', \mathbf{this}.Q) & \text{if } T_i = \mathbf{this}.f_j.Q \text{ and } t_j = u' \\ \mathcal{W}(u, \mathbf{this}.Q) & \text{if } T_i = \mathbf{this.out}.Q \\ \text{for } i \in \{1 \dots |\bar{f}|\} \end{cases}$$

and $C(\bar{t}) <: \bar{s}$. Then $H[\iota' \mapsto \llbracket \iota \parallel C \parallel \bar{f} : \bar{\text{val}} \quad \bar{v} : \bar{\text{null}} \rrbracket] \text{OK}$.

Proof: Let $H' = H[\iota' \mapsto \llbracket \iota \parallel C \parallel \bar{f} : \bar{\text{val}} \quad \bar{v} : \bar{\text{null}} \rrbracket]$. Assume ${}^1: \text{CT OK}$, ${}^2: \text{H OK}$, and ${}^3: H, \iota_0 \vdash \iota \triangleright u$. ${}^4: H, \iota_0 \vdash \bar{\text{val}} \triangleright \bar{t}$, ${}^5: \bar{p} = \text{Assemble}(\text{Mix}(H, \iota), C)$, ${}^6: \text{Members}(\bar{p}) = \bar{T} \bar{f}, \bar{T}' \bar{v}$, ${}^7: |\bar{f}| = |\bar{\text{val}}| = n$, ${}^8: \iota'$ new in H ,

$${}^9: s_i = \begin{cases} \mathcal{W}(u', \mathbf{this}.Q) & \text{if } T_i = \mathbf{this}.f_j.Q \text{ and } t_j = u' \\ \mathcal{W}(u, \mathbf{this}.Q) & \text{if } T_i = \mathbf{this.out}.Q \\ \text{for } i \in \{1 \dots n\} \end{cases}$$

and ${}^{10}: C(t_i) <: s_i$, for $i \in \{1 \dots n\}$.

By (WF-HEAP) showing that $H' \text{ OK}$ means showing that every object in H' is wellformed. By (8) and the definition of H' , H' is compatible with H , and H is undefined at ι' . Inspection of the rules (WF-ROOT), (WF-OBJ), (WF-NUL), and (WF-MEM) shows the

value of each member declared in a mixin of the object is either \perp , **null**, or an address ι_1 at which H is defined. Hence, ι' is not an enclosing object or the value of any member of any object in H, so from H OK follows ι_2 OK in H for all ι_2 where H is defined, and hence also ι_3 OK in H' for all ι_3 where H is defined. Since H' is defined in $D \cup \{\iota'\}$ where D is the domain of H, we have now dealt all addresses where H' is defined except ι' , so we need only show ι' OK in H'. Since $Encl(H(\iota'))$ is defined we know that $\iota' \neq \iota_{root}$, so we must use (WF-OBJ) to show this.

Note that ¹¹: $\iota \neq \iota'$ because H is defined at ι . From (5) via Lemma 14 we get ¹²: $\bar{p} = Assemble(Mix(H', \iota), C) = Mix(H', \iota')$.

Now we need to show that the value of each member $m \in Members(\bar{p})$ satisfies the implication in the premise of (WF-OBJ). We have to consider variables and fields separately, and we have to use induction for the fields.

If m is a variable v then from the definition of H' we conclude $H'(\iota')(v) = \mathbf{null}$, and by (WF-NUL) the implication is trivially satisfied.

To deal with fields we need to consider their ordering and handle the “smallest” ones first—by (1) fields are ordered such that for each field f , $DeclType(\bar{p}, f) = \mathbf{this.out}^k.\bar{f}.C \Rightarrow \forall i. f_i \sqsubset_f f$, i.e., “a field only depends on smaller fields”.

- Consider the smallest field f_j among the fields of ι' . Since f_j cannot depend on other fields in ι' , its declared type must be $\mathbf{this.out}^{k+1}.\bar{f}'' . C''$ for some k , \bar{f}'' , and C'' . By (9j) this implies that ¹³: $s_j = \mathcal{W}(u, \mathbf{this.out}^k.\bar{f}'' . C'')$. Note that (13) implies that s_j is a class type with class C'' and ¹⁴: $\mathcal{E}(s_j) = \mathcal{W}(u, \mathbf{this.out}^k.\bar{f}'')$. By (4j) we get ¹⁵: $H, \iota_0 \vdash val_j \triangleright t_j$. If $val_j = \mathbf{null}$ then we finish by using (WF-NUL) because $H'(\iota')(f_j) = \mathbf{null}$.

Otherwise there exists a ι'' such that $val_j = \iota''$. Note that by (13) $\mathcal{M}(s_j)$ is defined. Using this and (1) (2) (15) (10j) with Lemma 18.6 we conclude ¹⁶: $H, \iota_0 \vdash \iota'' \triangleright s_j$. From (14) (16) via (A-CTYPE) we conclude that there is a p' such that ¹⁷: $p'.C'' \in Mix(H, \iota'') = Mix(H', \iota'')$ and ¹⁸: $H, \iota_0 \vdash Encl(H(\iota'')) \triangleright \mathcal{W}(u, \mathbf{this.out}^k.\bar{f}'')$. Using (1) (2) (3) (18) with Lemma 21 yields $\Downarrow H(\iota, \mathbf{this.out}^k.\bar{f}'') = Encl(H(\iota''))$. Since H' is compatible with H this immediately yields ¹⁹: $\Downarrow H'(\iota, \mathbf{this.out}^k.\bar{f}'') = Encl(H'(\iota''))$. and with Lemma 17 this yields ²⁰: $\Downarrow H'(\iota', \mathbf{this.out}^{k+1}.\bar{f}'') = Encl(H'(\iota''))$. Finally, (20) (17) shows that the member related implication from (WF-OBJ) holds for f_j , so we are done.

- For a field f_j other than the one with the lowest order we assume that the member related implication holds for all fields with lower order than f_j . If the declared type of f_j is on the form $\mathbf{this.out}.Q$ then the proof in the previous case can be reused. Otherwise by (9j) the declared type T_j is $\mathbf{this.f}_m.\bar{f}'' . C''$ for some m , \bar{f}'' , and C'' . Moreover, t_m is an object type u' , such that ²¹: $s_j = \mathcal{W}(u', \mathbf{this.f}_m.\bar{f}'' . C'')$ and by (9j), ²²: $\mathcal{C}(t_j) <: s_j$. By (4j) (4m) we get ²³: $H, \iota_0 \vdash val_j \triangleright t_j$ and ²⁴: $H, \iota_0 \vdash val_m \triangleright u'$.

If $val_j = \mathbf{null}$ then we are done. Otherwise there exists a ι'' such that $val_j = \iota''$. Using (21) and the definition of \mathcal{W} we get $\mathcal{M}(s_j) \neq \perp$, and (1) (2) (23) (22) (21) with Lemma 18.6 then yields ²⁵: $H, \iota_0 \vdash \iota'' \triangleright \mathcal{W}(u', \mathbf{this.f}_m.\bar{f}'')$. Note that $Depth_s(\mathcal{W}(u', \mathbf{this.f}_m.\bar{f}'')) > 0$ because it is a class type, so from (1) (2) (25) with Lemma 18.3 we get ²⁶: $H, \iota_0 \vdash Encl(H(\iota'')) \triangleright \mathcal{W}(u', \mathbf{this.f}_m.\bar{f}'')$. Let $u' = \langle p'' \rangle.\bar{f}''$, then $\mathcal{W}(u', \mathbf{this.f}_m.\bar{f}'') = \langle p'' \rangle.\bar{f}''.\bar{f}''$, and then (26) with (A-OTYPE) for some suitable l yields $\Downarrow H(\iota_0, \mathbf{this.out}^l.\bar{f}''.\bar{f}'') = Encl(H(\iota''))$, but then there is an ι'''

such that $\Downarrow H(\iota_0, \mathbf{this.out}^l.\bar{f}''') = Encl(H(\iota'''))$, which from (24) via (A-OTYPE) shows that $val_m = \iota'''$ and in particular $val_m \neq \mathbf{null}$. This and (1) (2) (24) (26) with Lemma 21 shows that $\Downarrow H(\iota', \mathbf{this.f}_m.\bar{f}'') = Encl(H(\iota''))$. Since H' is compatible with H this and Lemma 16 yields $\Downarrow H'(\iota', \mathbf{this.f}_m.\bar{f}'') = Encl(H'(\iota''))$ and using $H'(\iota')(f_m) = \iota'''$ we can construct ²⁷: $\Downarrow H'(\iota', \mathbf{this.f}_m.\bar{f}'') = Encl(H'(\iota''))$. To conclude, (27) is the first clause from the right hand side of the member related implication, and the other clause (about the existence of a mixin ending in C'') is an easy consequence of (25) as in the first case of this proof. \square

Finally we reach the subject reduction theorem, which essentially states that evaluation of an expression with a given type leads to a result which also has that type or it raises a `NullErr`, and the receiver will preserve its type, and heap will remain well-formed.

Proof of Theorem 1: By induction in the structure of the derivation of the evaluation judgment. We start by assuming the left hand side of the implication and then show that the right hand side holds. For easy reference we number the parts as follows: ¹: *CT* OK, ²: H OK, ³: $p \vdash e : t$, ⁴: $H, \iota \vdash \iota \triangleright \langle p \rangle$, and ⁵: $e, H, \iota \rightsquigarrow r, H'$. The task is then to show ¹: H' OK, and either ^{2a}: $H', \iota \vdash val \triangleright t$ where $r = val$, or ^{2b}: $r = \text{NullErr}$. This is sufficient because by Lemma 20 and Corollary 15, (2), (4), (5), and H' OK ensure $H', \iota \vdash \iota \triangleright \langle p \rangle$.

Case (R1,R2): Trivial.

Case (R3): The usage of (R3) implies that $e = \text{path}$, $r = val$, $H' = H$, $t = u$, and ⁶: $\Downarrow H(\iota, \text{path}) = val$. In proving (3), $p \vdash \text{path} : u$, we must have used (T3), which implies ⁷: $\mathcal{W}(\langle p \rangle, \text{path}) = u$. Using (1) (2) (4) (6) (7) with Lemma 23 yields $H, \iota \vdash val \triangleright u$, which is (?2a). Since $H = H'$, the remaining result (?1) is trivial, which finishes the case.

Case (R4): From the evaluation we get ⁶: $\text{path}, H, \iota \rightsquigarrow \iota', H$ and ⁷: $H(\iota')(v) = val$, and from the typing, ⁸: $p \vdash \text{path} : u$ and ⁹: $\mathcal{W}(u, DeclType(u, v)) = s$. The induction hypothesis applied to (1) (2) (8) (4) (6) yields ¹⁰: $H, \iota \vdash \iota' \triangleright s$. Using (1) (2) (10) (9) (7) with Lemma 24 yields ¹¹: $H, \iota \vdash val \triangleright s$. Since (?1) is just (2) and (11) is (?2a), this concludes the case.

Case (R5): The evaluation yields ⁶: $\text{path}, H, \iota \rightsquigarrow \iota', H$, ⁷: $e, H, \iota \rightsquigarrow val, H'$, ⁸: $H'(\iota')(v) \neq \perp$, and ⁹: $H'' = H'[\iota' \mapsto H'(\iota')[v \mapsto val]]$. From the typing judgment, (T5), we get ¹⁰: $p \vdash \text{path}.v : s$, ¹¹: $p \vdash e : t$, and ¹²: $\mathcal{C}(t) <: s$. Moreover, (10) via (T4) yields ¹³: $p \vdash \text{path} : u$ and ¹⁴: $\mathcal{W}(u, DeclType(u, v)) = s$. Applying the induction hypothesis on (1) (2) (13) (4) (6) yields ¹⁵: $H, \iota \vdash \iota' \triangleright u$. Applying the induction hypothesis on (1) (2) (11) (4) (7) yields ¹⁶: H' OK and ¹⁷: $H', \iota \vdash val \triangleright t$. Now use (7) with Corollary 15 to conclude that H' is compatible with H, and use (1) (2) (16) (15) with Lemma 20 to get ¹⁸: $H', \iota \vdash \iota' \triangleright u$. Next, use (1) (16) (18) (17) (12) (14) (9) with Lemma 25 to conclude ¹⁹: H'' OK, which shows (?1). Finally, note that H'' is compatible with H' because the only difference between them is the value of one variable, and then use (1) (16) (19) (17) with Lemma 20 to get $H'', \iota \vdash val \triangleright t$, which is (?2a).

Case (R6): The evaluation $\mathbf{new path.C}(\bar{e}), H, \iota \rightsquigarrow val, H'''$ and the typing $p \vdash \mathbf{new path.C}(\bar{e}) : s_0$ together yield the following:

- ⁶: $\text{path}, H, \iota \rightsquigarrow \iota', H$
- ⁷: $e_i, H_i, \iota \rightsquigarrow val_i, H_{i+1}$, for $i \in \{1..n\}$
- ⁸: $\bar{p} = Assemble(Mix(H', \iota'), C)$

- ⁹: $\text{Members}(\bar{p}) = \bar{T} \bar{f}, \bar{T} \bar{v}$
- ¹⁰: $|\bar{f}| = n$
- ¹¹: l'' new in H'
- ¹²: $\text{Constr}(p_c) = T_0 C(\bar{T} \bar{f}) \{e';\}$
- ¹³: $H'' = H'[l'' \mapsto \llbracket l' \parallel C \parallel \bar{f} : \overline{\text{val}} \bar{v} : \overline{\text{null}} \rrbracket]$
- ¹⁴: $e', H'', l'' \rightsquigarrow \text{val}, H'''$
- ¹⁵: $p \vdash \text{path} : u$
- ¹⁶: $p'_c \in \mathcal{M}(u.C)$
- ¹⁷: $p \vdash \bar{e} : \bar{t}$
- ¹⁹: $\text{Constr}(p'_c) = T_0 C(\bar{T} \bar{f}) \dots$
- ²⁰: $s_i = \begin{cases} \mathcal{W}(u', \text{this.Q}) & \text{if } T_i = \text{this.f}_j.Q \text{ and } t_j = u' \\ \mathcal{W}(u, \text{this.Q}) & \text{if } T_i = \text{this.out.Q} \end{cases}$
for $i \in \{0 \dots n\}$
- ²¹: $\mathcal{C}(t_i) <: s_i$, for $i \in \{1 \dots n\}$

where $n = |\bar{e}|$, $p_c = p|\bar{p}|$, $H_1 = H$, and $H' = H_{n+1}$.

The choice of symbols above implies that the constructors found at the end of \bar{p} in the dynamic case and at the end of \bar{p}' in the static case must have the same signature. It is easy to see that the last element of $\text{Assemble}(\bar{p}, C)$, if defined, will be on the form $p'.C$. Hence, by (1), $\mathcal{M}(p_c)$ and $\mathcal{M}(p'_c)$ must have a mixin $p''.C$ in common, and this together with (WF3) ensures that they will have the same constructor signature.

Applying the induction hypothesis to (1) (2) (15) (4) (6) we obtain ²²: $H, \iota \vdash l' \triangleright u$. For each $i \in \{1 \dots n\}$ we obtain the following implication by the induction hypothesis:

$$\left[\begin{array}{l} (1) \\ H_i \text{ OK} \\ (17i) \\ H_{i, \iota} \vdash l \triangleright \langle p \rangle \\ (7i) \end{array} \right] \Rightarrow \left[\begin{array}{l} \text{23i: } H_{i+1} \text{ OK} \\ \text{24i: } H_{i+1, \iota} \vdash l \triangleright \langle p \rangle \\ \text{25i: } H_{i+1, \iota} \vdash \text{val}_i \triangleright t_i \end{array} \right]$$

We can establish the left hand side of this implication for all $i \in \{1 \dots n\}$, which shows that the right hand side holds, i.e., that (23i)...(25i) hold. For $i = 1$ we use (2) (4). For $i > 1$ we use the result from the implication with $i - 1$. In particular, ²⁶: H' OK and ²⁷: $H', \iota \vdash l \triangleright \langle p \rangle$.

Using Corollary 15 and Lemma 20 as usual we get ²⁸: $H', \iota \vdash l' \triangleright u$ from (22) etc., and ²⁹: $H', \iota \vdash \overline{\text{val}} \triangleright \bar{t}$ from (25i) with $i \in \{1 \dots n\}$, etc. Now use (1) (26) (28) (29) (8) (9) (10) (11) (12) (20) (21) (13) with Lemma 26 to conclude that ³⁰: H'' OK. Moreover, (13) directly shows that ³¹: $\text{Encl}(H''(l'')) = l'$.

We need to obtain results associated with (14). At this point it is crucial that we use induction in the shape of the evaluation derivation and not the typing derivation, because there is no typing judgment corresponding to (14). However, we can rely on program well-formedness to obtain such a typing judgment, and then use the induction hypothesis on (14) together with that typing. As (8) (12) (13) shows, e' is the expression returned from the constructor in the last (most specific) mixin of l'' , namely $\bar{p}|\bar{p}|$; to avoid repeating this unwieldy expression many times we let $p'' = \bar{p}|\bar{p}|$. By (1) there exists a type t' such that ³²: $p'' \vdash e' : t'$ and ³³: $\mathcal{C}(t') <: \mathcal{W}(\langle p'' \rangle, T_0)$. Since p'' is one of the mixins in l'' we can use (1) (30) with Lemma 19 to get ³⁴: $H'', l'' \vdash l'' \triangleright \langle p'' \rangle$. Finally we use the induction hypothesis on (1) (30) (32) (34) (14) which yields ³⁵: H''' OK and ³⁶: $H''', l'' \vdash \text{val} \triangleright t'$. Result (35) is obviously useful because it is (?1). Result (36) is not so helpful because we need to prove that val has a certain type *as seen from* ι , but (36) is concerned with the type of val as seen from l'' . Nevertheless, it is used below in a more indirect manner.

The last task is to show that the final result, val , agrees with s_0 . If $\text{val} = \text{null}$ then agreement is trivial and we are done. Otherwise there is a ι_v such that $\text{val} = \iota_v$. From (1) (35) (36) (33) with Lemma 18.6 we get ³⁷: $H''', l'' \vdash \iota_v \triangleright \mathcal{W}(\langle p'' \rangle, T_0)$. By the grammar, $T_0 = \text{path}'.C'$ for some path' and C' , so $\mathcal{W}(\langle p'' \rangle, T_0)$ is a class type with class C' , i.e. ³⁸: $\mathcal{W}(\langle p'' \rangle, T_0) = u_0.C'$ for some object type u_0 , and since we must have used (A-CTYPE) in the proof of (37) we get ³⁹: $\exists p'''. p'''.C' \in \text{Mix}(H''', \iota_v)$.

To finish the proof we need to consider two cases for the shape of the path in the declared return type of the constructor, path' :

- $\text{path}' = \text{this.out}^{k+1}.\bar{f}''$: It is easy to see that all depths are non-negative and ⁴⁰: $\mathcal{E}(u_0.C') = u_0 = \mathcal{W}(\langle p'' \rangle, \text{path}')$, so ⁴¹: $\text{Depth}_s(u_0.C') = 1 + \text{Depth}_s(u_0) > 0$. Now use (1) (35) (37) (41) (40) with Lemma 18.3 to get ⁴²: $H''', l'' \vdash \text{Encl}(H''(l_v)) \triangleright \mathcal{W}(\langle p'' \rangle, \text{path}')$. Let ⁴³: $p''' = C_1 \dots C_m$. Using (1) (35) (34+Corollary 15 and Lemma 20) (43) with Lemma 18.1 we get ⁴⁴: $\text{Depth}(H''', l'') = \text{Depth}_s(\langle p'' \rangle) = m$. Since $\mathcal{W}(\langle p'' \rangle, \text{path}') = \langle C_1 \dots C_{m-k-1} \rangle.\bar{f}''$ we get from (42) (44) via (A-OTYPE) that ⁴⁵: $j = \text{Depth}(H''', l'') - (m-k-1) = k+1$ and ⁴⁶: $\Downarrow H''(l'', \text{this.out}^{k+1}.\bar{f}'') = \text{Encl}(H''(l_v))$. Using (31) and Lemma 14 we get ⁴⁷: $\text{Encl}(H''(l'')) = l'$, and then using (46) (47) with Lemma 17, ⁴⁸: $\Downarrow H''(l', \text{this.out}^k.\bar{f}'') = \text{Encl}(H''(l_v))$. Based on $T_0 = \text{this.out}^{k+1}.\bar{f}''$, (20) yields ⁴⁹: $s_0 = \mathcal{W}(u, \text{this.out}^k.\bar{f}''.C')$. From (28) etc. with Corollary 15 and Lemma 20 we get ⁵⁰: $H''', \iota \vdash l' \triangleright u$, and from (49) we get ⁵¹: $\mathcal{E}(s_0) = \mathcal{W}(u, \text{this.out}^k.\bar{f}'')$. Using (1) (36) (50) (48) (51) with Lemma 23 yields ⁵²: $H''', \iota \vdash \text{Encl}(H''(l_v)) \triangleright \mathcal{E}(s_0)$. From (49) we conclude that $\text{Cls}_s(s_0) = C'$. Finally, we use (39) (52) with (A-CTYPE) to conclude that ⁵³: $H''', \iota \vdash \iota_v \triangleright s_0$, which is (?2a).
- $\text{path}' = \text{this.f}_j.\bar{f}''$ and $t_j = u'$: Using $\mathcal{W}(\langle p'' \rangle, \text{path}'.C') = \langle p'' \rangle.f_j.\bar{f}''$. C' and (37) we get ⁴⁷: $H''', l'' \vdash \iota_v \triangleright \langle p'' \rangle.f_j.\bar{f}''$. C' , which by (A-CTYPE) yields ⁴⁸: $H''', l'' \vdash \text{Encl}(H''(l_v)) \triangleright \langle p'' \rangle.f_j.\bar{f}''$. From (1) (35) (34 + Corollary 15 and Lemma 20) via Lemma 18.1 we get ⁵⁰: $\text{Depth}(H''', l'') = \text{Depth}_s(\langle p'' \rangle) = |p''|$. Now, (48) (50) by (A-OTYPE) yields ⁵¹: $j' = \text{Depth}(H''', l'') - |p''| = 0$ and ⁵²: $\Downarrow H''(l'', \text{this.f}_j.\bar{f}'') = \text{Encl}(H''(l_v))$. By the definition of $\Downarrow H''$ this implies ⁵³: $\Downarrow H''(l'', \text{this.f}_j) = \iota_j$ where $H''(l'')(f_j) = \iota_j$. From (13) and using Lemma 14 we get $\iota_j = \text{val}_j$. But then ⁵⁴: $\Downarrow H''(l_j, \text{this.f}_j) = \text{Encl}(H''(l_v))$. Knowing more about t_j and val_j , and using Corollary 15 and Lemma 20 as usual we deduce from (25j) that ⁵⁵: $H''', \iota \vdash \iota_j \triangleright u'$. From (20), using our knowledge about path' and hence T_0 , we now get $s_0 = \mathcal{W}(u', \bar{f}''.C')$, hence ⁵⁶: $\mathcal{E}(s_0) = \mathcal{W}(u', \bar{f}'')$. Finally, (1) (35) (55) (54) (56) with Lemma 23 yields ⁵⁷: $H''', \iota \vdash \text{Encl}(H''(l_v)) \triangleright \mathcal{E}(H_s(t))$, and then (39) (57) with (A-CTYPE) yields ⁵⁸: $H''', \iota \vdash \iota_v \triangleright s_0$, which is (?2a).

By inspection of (20) we can see that no other cases than these two are possible for path' , which again shows that (?2a) holds in all cases, and thus the proof of this case is hereby complete.

Case (ER1): The error rules are easy to handle, so we do not cover them in full detail. This rule is of course a shorthand for three rules, one for each conclusion. However, they may be handled identically: Since the heap is unchanged the required H OK is immediate, and so is $H, \iota \vdash l \triangleright \langle p \rangle$. Finally, $r = \text{NullErr}$ satisfies the disjunction.

Case (ER2): Consider the first of the two rules. From $p \vdash \text{path}.v : t$ we conclude by (T4) that $p \vdash \text{path} : u$ and

$\mathcal{W}(u, \text{DclType}(u, v)) = t \neq \perp$. Applying the induction hypothesis on CT OK, H OK, $p \vdash \text{path} : u$, $H, \iota \vdash \iota \triangleright (p)$, and $\text{path}, H, \iota \rightsquigarrow \iota', H$ yields $H, \iota \vdash \iota' \triangleright u$. But then by Lemma 22 $H(\iota')(v) \neq \perp$. This is a contradiction, so it cannot be the case that the evaluation derivation is based on this rule, so we need not show that the right hand side of the soundness implication holds. The same proof works for the second rule.

Case (ER3): By (T6) we get ${}^6 p \vdash \text{path} : u$ and ${}^7 p \vdash \bar{e} : \bar{t}$. Apply the induction hypothesis to (1) (2) (6) (4) and $\text{path}, H, \iota \rightsquigarrow \iota', H$ to get ${}^8 H, \iota \vdash \iota' \triangleright u$. Next, (1) (2) (8) with Lemma 18.4 yields ${}^9 \text{Mix}(H, \iota') \supseteq \mathcal{M}(u)$. From (T6) we have $\text{Assemble}(u, C) = \mathcal{M}(u.C) \neq \perp$, so from (9) with Lemma 4 we get $\text{Assemble}(\text{Mix}(H, \iota'), C) \neq \perp$. This is a contradiction, so it cannot be the case that the evaluation derivation is based on this rule.

Case (ER4): This case is when the constructor is given an incorrect number of arguments. We already argued at the beginning of the case (R6) that this cannot occur.

Case (ERP1...ERP7): These error propagation cases just ensure that an error in a subtree of an evaluation is always propagated as the result of the entire evaluation, and the induction hypothesis is applied to conclude that only `NullErr` can be the result, never `TypeErr`.

Case (ERH1...ERH4): This rule will only be used in a context where $p \vdash \text{path} : u$, and by the induction hypothesis, $\mathcal{W}(u, \text{DclType}(u, v)) = t \neq \perp$. Now we can use Lemma 23 on each prefix of the path to ensure that agreement exists at each step, and Lemma 18.1 to conclude that the statically predicted number of enclosing objects exists, such that evaluation of each **out** step will succeed, and finally Lemma 22 to show that for each field lookup the field will be defined, although possibly **null**, and hence the returned result may be a value or `NullErr`, but never `TypeErr`. \square

A.5 Error handling

The rules dealing with error situations are shown in Fig. 13.

LEMMA 27 (Coverage of path evaluation). *For all H , path and ι , $\Downarrow H(\iota, \text{path}) \in \text{Value} \cup \{\text{TypeErr}, \text{NullErr}\}$.*

Proof: We prove the lemma by induction on structure of path .

Case (this): Definition of \Downarrow .

Case (spine.out): First, note that for spines, \Downarrow never returns **null** or `NullErr`:

$$\Downarrow H(\iota, \text{spine}) \in \text{Address} \cup \{\text{TypeErr}\}$$

By induction hypothesis, $\Downarrow H(\iota, \text{spine})$ may be `TypeErr` or ι' . (ErH3) handles `TypeErr`. If it is ι' , then $H(\iota')$ may be \perp or an object. Case 6 in definition of \Downarrow handles \perp . Case 2 in definition of \Downarrow handles objects.

Case (path.f): By induction hypothesis, $\Downarrow H(\iota, \text{path})$ may be **null**, `Err`, or ι' . Case 4 in definition of \Downarrow handles **null**. (ErH3) handles `Err`. If it is ι' , then $H(\iota')(f)$ may be \perp or `val`. Case 5 in definition of \Downarrow handles \perp . Case 3 in definition of \Downarrow handles `val`.

Proof of Lemma 1: By induction on n with cases on the structure of e . The base case for the induction, $n = 0$ is trivial by rule (Kill).

Case (null): Trivial by rule (T1).

Case ($e ; e$): Immediate from induction hypothesis, and rules (R2) and (ErP5).

$$\frac{\text{path}, H, \iota \rightsquigarrow \text{null}, H}{\text{path}.v, H, \iota \rightsquigarrow \text{NullErr}, H \quad \text{path}.v = e, H, \iota \rightsquigarrow \text{NullErr}, H \quad \text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{NullErr}, H} \quad (\text{ER1})$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad H(\iota')(v) = \perp}{\text{path}.v, H, \iota \rightsquigarrow \text{TypeErr}, H \quad \text{path}.v = e, H, \iota \rightsquigarrow \text{TypeErr}, H} \quad (\text{ER2})$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad \text{Assemble}(\text{Mix}(H, \iota'), C) = \perp}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{TypeErr}, H} \quad (\text{ER3})$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad \text{Assemble}(\text{Mix}(H, \iota'), C) = \bar{p} \quad \text{Members}(\bar{p}) = \bar{T}\bar{f}, \bar{v} \quad |\bar{e}| \neq |\bar{f}|}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{TypeErr}, H} \quad (\text{ER4})$$

$$\frac{\Downarrow H(\iota, \text{path}) = \text{Err}}{\text{path}, H, \iota \rightsquigarrow \text{Err}, H} \quad (\text{ERP1})$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \text{Err}, H}{\text{path}.v, H, \iota \rightsquigarrow \text{Err}, H \quad \text{path}.v = e, H, \iota \rightsquigarrow \text{Err}, H \quad \text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{Err}, H} \quad (\text{ERP2})$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad e, H, \iota \rightsquigarrow \text{Err}, H'}{\text{path}.v = e, H, \iota \rightsquigarrow \text{Err}, H'} \quad (\text{ERP3})$$

$$\frac{e, H, \iota \rightsquigarrow \text{Err}, H'}{e; e', H, \iota \rightsquigarrow \text{Err}, H'} \quad (\text{ERP4})$$

$$\frac{e, H, \iota \rightsquigarrow \text{val}, H' \quad e', H', \iota \rightsquigarrow \text{Err}, H''}{e; e', H, \iota \rightsquigarrow \text{Err}, H''} \quad (\text{ERP5})$$

$$\frac{1 \leq j \leq |\bar{e}| \quad \text{path}, H_1, \iota \rightsquigarrow \iota', H_1 \quad e_i, H_i, \iota \rightsquigarrow \text{val}_i, H_{i+1} \text{ for } i = 1 \dots j-1 \quad e_j, H_j, \iota \rightsquigarrow \text{Err}, H_j}{\text{new path}.C(\bar{e}), H_1, \iota \rightsquigarrow \text{Err}, H_j} \quad (\text{ERP6})$$

$$\frac{\text{path}, H, \iota \rightsquigarrow \iota', H \quad H = H_1 \quad e_i, H_i, \iota \rightsquigarrow \text{val}_i, H_{i+1} \text{ for } i \in \{1 \dots |\bar{e}|\} \quad H' = H_{|\bar{e}|+1} \quad \bar{p} = \text{Assemble}(\text{Mix}(H', \iota'), C) \quad \text{Members}(\bar{p}) = \bar{T}\bar{f}, \bar{v} \quad |\bar{f}| = |\text{val}| \quad \iota'' \text{ is new in } H' \quad \text{Constr}(p_{|\bar{p}|}) = \text{T}C(_)\{e'; \} \quad H'' = H'[\iota'' \mapsto \llbracket \iota' \rrbracket C \parallel \bar{f} : \text{val} \quad \bar{v} : \text{null} \rrbracket] \quad e', H', \iota'' \rightsquigarrow \text{Err}, H''}{\text{new path}.C(\bar{e}), H, \iota \rightsquigarrow \text{Err}, H''} \quad (\text{ERP7})$$

$$\Downarrow H(\iota, \text{spine.out}) = \text{TypeErr}, \quad \text{if } \Downarrow H(\iota, \text{spine}) = \iota_{\text{root}} \quad (\text{ERH1})$$

$$\Downarrow H(\iota, \text{path}.f) = \text{NullErr}, \quad \text{if } \Downarrow H(\iota, \text{path}) = \text{null} \quad (\text{ERH2})$$

$$\Downarrow H(\iota, \text{path}.f) = \text{TypeErr}, \quad \text{if } H(\Downarrow H(\iota, \text{path}))(f) = \perp \quad (\text{ERH3})$$

$$\Downarrow H(\iota, \bar{q}.q) = \text{Err}, \quad \text{if } \Downarrow H(\iota, \bar{q}) = \text{Err} \quad (\text{ERH4})$$

Figure 13. Error handling

Case (path): Follows from Lemma 27.

Case (path.v): By Lemma 27, evaluation of path can be **null**, `Err`, or ι' . (Er1) handles **null**. (ErP2) handles `Err`. If it is ι' then $H(\iota')(v)$ may be \perp or `val`. (Er2) handles \perp . (R4) handles `val`.

Case (path.v=e): Includes the steps from $\text{path}.v$, except the last step where path evaluates to `val`. Then e can evaluate to `Err` or `val`. (ErP3) handles `Err`. (R5) handles `val`.

Case (new path.C(\bar{e})): Includes the steps from $\text{path}.v$, except the last step where path evaluates to `val`. Then e_i may evaluate to `Err` or `val`. (ErP6) handles `Err`. $|\bar{e}| \neq |\bar{f}|$ is handled by (Er4). $\text{Assemble}(\text{Mix}(H, \iota'), C) = \perp$ is handled by (Er3). Finally, e' may evaluate to `Err` or `val`. (ErP7) handles `Err`. (R6) handles `val`.