Dependent Classes

Vaidas Gasiunas Mira Mezini Klaus Ostermann
Technische Universität Darmstadt, Germany
{gasiunas,mezini,ostermann}@informatik.tu-darmstadt.de

Abstract

Virtual classes allow nested classes to be refined in subclasses. In this way nested classes can be seen as dependent abstractions of the objects of the enclosing classes. Expressing dependency via nesting, however, has two limitations: Abstractions that depend on more than one object cannot be modeled and a class must know all classes that depend on its objects. This paper presents *dependent classes*, a generalization of virtual classes that expresses similar semantics by parameterization rather than by nesting. This increases expressivity of class variations as well as the flexibility of their modularization. Besides, dependent classes complement multimethods in scenarios where multi-dispatched abstractions rather than multi-dispatched methods are needed. They can also be used to express more precise signatures of multimethods and even extend their dispatch semantics. We present a formal semantics of dependent classes and a machine-checked type soundness proof in Isabelle/HOL [29], the first of this kind for a language with virtual classes and path-dependent types.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects, polymorphism, inheritance; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

General Terms Languages, Theory

Keywords dependent classes, virtual classes, dynamic dispatch, multiple dispatch, multimethods, variability

1. Introduction

A *virtual class* is an attribute of an object. Analogous to virtual *methods* in traditional object-oriented languages, virtual classes are defined within their enclosing object's class, can be overridden and extended in subclasses, and are accessed relative to the enclosing object, using late binding. As a result, the actual definition of a virtual class referred to by a term obj.VC is not fully known at compile time. VC is some class accessible as the attribute with same name of the object obj; some of VC's features may be statically known through the static type of obj, others may vary dynamically depending on the dynamic type of obj.

Virtual classes and related mechanisms have proved useful in various situations: To define families of collaborating objects [10, 24], to develop large-scale extensible components [30, 1, 33, 31], to address the "expression problem" - the possibility to extend both the set of data structures and the set of operations [12, 11], and to modularize features that involve multiple classes [26, 1].

The downside is, however, that virtual classes must be nested within other classes. Nesting requires to cluster together all classes that depend on instances of a particular class, which may unnecessarily introduce coupling between them. It also limits extensibility: When new classes need to be modeled as depending on the instances of an existing class, one must modify that class and its subclasses. Further, nesting limits expression of variability since the interface and the implementation of a virtual class can only depend on its *single* enclosing object.

There are, however, various application scenarios, where classes that depend on arbitrarily many objects are needed; in this paper, we consider one such scenario in more detail: Modeling aggregate objects, the type and implementation of which depend on the types of their constituent parts. Our running example is that of shapes (e.g., lines, boxes, circles, etc.), as well as various aggregations thereof (e.g., intersection or union) that are shapes again. The shapes live in different kinds of spaces (e.g., 2D, 3D, etc.) and have different properties (e.g., solid, bounded, convex, etc.).

The operations available on a shape and their implementations depend on its properties and the space it lives in. The interface and implementation of aggregated shapes may also depend on their constituent parts, e.g., the union of a solid shape with any shape is a solid shape, while the union of two bounded shapes is a bounded shape. Hence, aggregated shapes must be modeled as objects that depend on more than one object, a feature that virtual classes do not support. Furthermore, nesting all kinds of shapes within space classes would introduce unnecessary dependencies and restrict the extensibility of the system.

To address these problems, we propose a generalization of virtual classes, which we call *dependent classes*. A dependent class is a class whose structure depends on *arbitrarily many* objects; this dependency is expressed explicitly over class parameters, rather than by nesting. In a sense, dependent classes can be seen as an combination of virtual classes with multi-dispatch [9, 3, 34, 6]. The notion of path-dependent types, as employed by most proposals for static type systems for virtual classes to keep track of the dependencies between a type and the "family" or "enclosing" object, is generalized as well, such that types can depend on an arbitrary number of objects described by paths.

The contributions of this paper are:

- Dependent classes are motivated and the design of a language with dependent classes is discussed.
- The vc^n calculus – a formal definition of the static and dynamic semantics of such a language is defined together with a machine-checked soundness proof in Isabelle/HOL [29]. Since soundness proofs for languages with path-dependent types are usually either quite sketchy or quite complex[1], it is hard to make sure that the proof is free of bugs. Indeed, we have discovered various bugs and unnecessary well-formedness conditions that we probably would not have discovered with a hand-written proof. We hope that our Isabelle formalization and proof can be reused and adapted by others working on variants of virtual classes. As further validation, we have also implemented an interpreter for vc^n. The formalization, the proof, and the interpreter can be downloaded at [15].
- New results in the meta-theory of path-dependent types are presented. A completeness theorem is formulated and proved which shows that the type system is optimal in a certain sense. Also, surprisingly weak conditions under which the type system is decidable are identified.
- Apart from supporting dependency on multiple objects, the advantage of the proposed calculus over previous formalizations of virtual classes is a better balance between simplicity and decidability on one side and expressive power on the other side.
- Dependent classes and the type system for them also contribute to languages with multimethods. First, due to their enhanced subtype relation, they provide a more expressive dispatch semantics. Second, they can be used to describe more precise signatures of multimethods.

The remainder of this paper is organized as follows. Sec. 2 presents and motivates dependent classes and explains their relation to virtual classes and to multimethods. A calculus that formalizes the static and dynamic semantics of dependent classes is presented in Sec. 3. Properties of the calculus - soundness, decidability and completeness - are discussed in Sec. 4. Sec. 5 discusses possible variations on the dispatch semantics. Related work is discussed in Sec. 6 and Sec. 7 concludes the paper.

[1] For example, the soundness proof of vc [12] is 12 double-column pages long; the print-out of the vc^n proof is about 70 pages.

2. Dependent Classes in a Nutshell

This section presents a Java-like language with dependent classes by the example of modeling families of shapes. In Sec. 2.1, basic constructs for defining and refining dependent classes as well as type declarations using dependent classes are featured by considering only simple geometrical objects, e.g., points and vectors, that only need dependent classes with a single parameter. In Sec. 2.2, aggregated shapes, e.g., unions and intersections of shapes, are considered, featuring the need for depending on multiple parameters and on the corresponding semantics. Sec. 2.3 and 2.4 discuss the relation of dependent classes to virtual classes and multimethods, respectively.

2.1 Defining and Refining Dependent Classes

When we talk about points and lines, we usually have some specific space in mind — a plane or a three-dimensional space, Euclidean or non-Euclidean. The properties and the implementation of points and vectors depend on the kind of space to which they belong. For instance, points in a two-dimensional space have only two coordinates while points in a 3D space also have a third coordinate.

Further, when working with points and lines, we would like to make sure that some restrictions are obeyed with respect to the enclosing space. For example, one can add a vector to a point and the result of this operation is again a point. For such an operation we would like to ensure that (a) the vector we add is from the same space as the point being added to, and (b) the result is also a point in the same space.

To see how the requirements just outlined are modeled with dependent classes, consider the implementation of the example in a Java-like language with dependent classes in Fig. 1. The variation of spaces is modeled by a simple inheritance hierarchy (lines 1 - 7). Dependent abstractions such as points and vectors are modeled by dependent classes `Point` (lines 9, 15, 19) and `Vector` (line 12) that take an instance of `Space` as a parameter.

This parameterization has two implications. First, `s` is a *field* of `Point`, which can be used to retrieve the space the point belongs to; this field is immutable and its value is passed as a constructor parameter. A more interesting implication is that `Point` is declared as a "relative" class, the definition of which depends on its parameter `s`. There are several declarations of `Point` (lines 9, 15, and 19) - one for each kind of space.

```
1  abstract class Space { ...
2    abstract Point(s: this) getOrigin();
3  }
4  class 2DSpace extends Space { ...
5    Point(s: this) getOrigin() { return new 2DPoint(this, 0, 0); }
6  }
7  class 3DSpace extends Space { ... }
8
9  abstract class Point(Space s) { ...
10   abstract Point(s:s) add(Vector(s:s) v);
11 }
12 abstract class Vector(Space s) { ...
13   abstract Vector(s:s) scale(double d);
14 }
15 abstract class Point(2DSpace s) {
16   abstract double getX();
17   abstract double getY();
18 }
19 abstract class Point(3DSpace s) {
20   abstract double getX();
21   abstract double getY();
22   abstract double getZ();
23 }
24
25 class 2DPoint(2DSpace s, double x, double y) extends Point {
26   ...
27   double getX() { return x; }
28   double getY() { return y; }
29   Point(s:s) add(Vector(s:s) v) {
30           return new 2DPoint(s, x + v.getX(), y + v.getY());
31   }
32 }
```

Figure 1. Points and vectors as dependent classes of a space

More specific declarations implicitly inherit from the more general declarations, which means that the declarations of Point for 2DSpace (line 15) and 3DSpace (line 19) implicitly inherit from the declaration of Point for Space (line 9). Some operations on points and vectors are available for any kind of space. For example, lines 10 and 13 declare generic methods to add a vector to a point and to multiply a vector by a scalar value. In addition, points in a plane or in a 3D space also have coordinates and respective operations for accessing them.

Besides implicit inheritance relations, explicit subclasses of dependent classes can also be declared. For example, Point is an abstract class which can be implemented in different ways. 2DPoint is a sample implementation of a point in a plane using Cartesian coordinates. Thus, we declare 2DPoint as a subclass of Point, and constrain its s parameter to 2DSpace, which means that 2DPoint is available only in planes, but not in other spaces. 2DPoint has additional parameters, x and y, which are fields that are initialized via constructor parameters. It is required that a subclass has all the constructor parameters of its superclasses, whereby the parameters are matched by name. Fields x and y are not used for dispatch in our example, we use them just as normal fields for storing state, but in principle they could also be used for dispatch.

```
Vector(s: v1.s) normal(Vector(s: 3DSpace) v1,
                       Vector(s: v1.s) v2) { ... }
```

Figure 2. Signature of normal method

Similar to virtual classes, dependent classes can be used to describe types that depend on objects. Dependent types are never compatible if they depend on different objects [12]. That is, every instance of a Space class has its own universe of types and objects, and the type system can make sure that different universes will never be mixed, even if both Space instances are instances of the same variant of Space.

To type generic methods such as add and scale correctly, we should be able to express that they operate on objects from the same space – in terms of family polymorphism [10], we would like to express that these operations operate on objects of the same family. Similar to languages with virtual classes, such typing is expressed in our approach by means of path-dependent types. The type Vector(s: s) in line 10, which is a short form for Vector(s: this.s), is an example of a path-dependent type. It expresses that the object is a Vector whose field s is of type this.s.

The type this.s is an example of a *path type*; these are types that have a single instance, namely the object pointed to by the path expression. Hence, the declaration that some object o is of type this.s means that o is equal to the object referred to by the expression this.s. Since this refers to the receiver point object in the context of a call to the method Point.add(...), the type declaration Vector(s: this.s) expresses that the parameter must be a vector that belongs to the same space as the receiver point.

For further examples of types in the language consider the method normal in Fig. 2; it takes two vectors in a 3D space as parameters and returns a vector that is perpendicular to both of them. The type declaration for the first parameter, Vector(s: 3DSpace), states that the first parameter can be a vector of any 3DSpace. The type of the second parameter and the return type is Vector(s:v1.s); it is more precise and states that these vectors must be from the same space as the first vector.

2.2 Dependency on Multiple Parameters

In general, dependent classes can be defined with an arbitrary number of parameters, thereby enabling to model abstractions that depend on several other abstractions. To illustrate dependent classes with multiple parameters and their benefits, consider an extension of our example. Imagine that we develop a library of shapes, a fragment of which is shown in Fig. 3.

There is a general Shape abstraction and a set of concrete shapes, some of them (e.g., Box) available for any space, others (e.g., Circle) specific to some concrete space. The set of operations available on shapes and their implementations may depend on the type of the space to which shapes belong;

```
abstract class Shape(Space s) {
  abstract bool pointInside (Point(s: s) pt);
}

abstract class Bounded(Space s) extends Shape {
  abstract Box(s: s) getBoundBox();
}

abstract class Solid (Space s) extends Shape {
  abstract bool pointOnEdge(Point(s: s) pt);
  abstract double getContent ();
}

class Box(Space s, Point(s:s) pt1, Point(s:s) pt2)
extends Bounded, Solid { ...
  bool pointInside (Point(s: s) pt) {
    return pt1.lessEqual(pt) && pt.lessEqual(pt2);
  }
  Box(s:s) getBoundBox() { return this; }
}

class Box(2DSpace s, Point(s:s) pt1, Point(s:s) pt2) { ...
  double getContent() {
    return (pt2.getX() − pt1.getX()) ∗
           (pt2.getY() − pt1.getY());
  }
}

class Box(3DSpace s, Point(s:s) pt1, Point(s:s) pt2) { ...
  double getContent() {
    return ... ∗ (pt2.getZ() − pt1.getZ());
  }
}

class Circle(2DSpace s, Point(s:s) c, double r)
extends Bounded { ... }
```

Figure 3. Shapes and their properties

their signatures and implementations will involve points and vectors of the space, so it is beneficial to define the shapes as dependent classes of Space and its subclasses.

Shapes can have different properties: One can differentiate between bounded and infinite shapes, solid and manifold shapes, closed and open shapes, etc. The properties of a shape may determine the operations available on it: for example, we can compute the bounding box of a bounded shape, we can compute the content[2] of a solid shape, or we can test whether a point is on its boundary. We can model these variations by abstract subclasses of Shape, such as Bounded and Solid in Fig. 3. Concrete shapes then inherit from a subset of such abstract classes, depending on the properties that they have, and implement their operations. For example, a Box is both Solid and Bounded.

More sophisticated shapes can be built from primitive shapes using various composition operations, such as union, intersection, difference and inversion. Such composite shapes can be described as objects that aggregate other shapes. For an example, consider the class Union at line 1 of Fig. 4, which describes the union of two shapes.[3] A union of two shapes is again a shape, so we declare Union as a subclass of Shape. Union is a dependent type of Space, and using path-dependent types we declare that only the union of shapes from the same space is possible.

[2] Content is a generalized concept for volume or area depending on space dimensions.

```
1  class Union(Space s, Shape(s:s) s1, Shape(s:s) s2)
2  extends Shape { ...
3    bool pointInside (Point(s:s) pt) {
4      return s1.pointInside(pt) || s2.pointInside(pt);
5    }
6  }
7
8  class Union(Space s, Bounded(s:s) s1, Bounded(s:s) s2)
9  extends Bounded { ...
10   Box(s:s) boundBox() {
11     return joinBounds(s1.getBoundBox(), s2.getBoundBox());
12   }
13 }
14
15 class Union(Space s, Solid(s:s) s1, Shape(s:s) s2)
16 extends Solid { ...
17   double getContent() { return s1.getContent(); }
18 }
19
20 class Union(Space s, Shape(s:s) s1, Solid(s:s) s2)
21 extends Solid { ...
22   double getContent() { return s2.getContent(); }
23 }
24
25 class Union(Space s, Solid(s:s) s1, Solid(s:s) s2) { ...
26   double getContent() { /* some approximation */ }
27 }
28
29 class Union(2DSpace s, Box(s:s) s1, Box(s:s) s2) {
30   double getContent() { ... }
31 }
32
33 class Union(Space s, Solid(s:s) s1, s1 s2) {
34   double getContent() { return s1.getContent(); }
35 }
36
37 class Intersection (Space s, Shape(s:s) s1, Shape(s:s) s2)
38 extends Shape { ... }
39
40 class Difference (Space s, Shape(s:s) s1, Shape(s:s) s2)
41 extends Shape { ... }
```

Figure 4. Composite shapes

The interface and subtype relations of a composite shape may vary depending on the type of the shapes it composes. For example, the union of two bounded shapes is again a bounded shape, while the union of a solid shape with any shape is again a solid shape. In order to model such subtype relations, the definition of Union has to vary with respect to two parameters. The declaration at line 8 of Fig. 4 refines Union for the case when its parameters are both Bounded and states that this refinement of Union has a more specific superclass than the general definition, namely Bounded. To specify that the union of two shapes, of which at least one is

[3] A union of two shapes is a shape consisting of the points of the both shapes.

```
1 void test (Box(s: 2DSpace) b, Circle (s: b.s) c) {
2     Union(s: b.s, s1: b, s2: c) u = new Union(b.s, b, c);
3     Box(s:b.s) b2 = u.boundBox();
4     bool inside = u. pointInside (new 2DPoint(b.s ));
5     double cont = u. getContent ();
6 }
```

Figure 5. Test case for the union shape

solid, is again solid, two declarations are given: One, where s1 is Solid and s2 is Shape (line 15), and another, where s1 is Shape and s2 is Solid (line 20). In this way, the inheritance relations of Union depend on the types of the two parameters.

Not only the interface, but also the implementation of a class may vary depending on multiple parameters of it. Different operations of Union can be implemented at different levels of abstraction. For the pointInside method, an operation inherited from Shape, a generic implementation independent of the parameter types is given in Line 3: Testing whether a point is inside a union is reduced to testing whether the point is inside one of its components (line 3). Other operations can be implemented differently for different parameter types of the class. For example, the content of a union of two shapes, only one of which is solid, is equal to the content of the solid component (lines 17 and 22). If both shapes are solid, we can only provide an inefficient approximate algorithm (lines 26), while a relatively efficient algorithm can be given to compute the content of the union of two boxes (line 30) in a 2DSpace.

Path types can also be used for refining dependent classes. For example, line 33 gives a refinement of Union for the case when a solid shape is combined with itself. This is expressed by specifying s1 as the type of s2, which means that s2 must point to the same object as s1. For this special case, we can provide very a efficient implementation of getContent method: It simply forwards the method call to s1.

A concrete instance of a dependent class inherits all declarations that match the given parameter types. A class declaration matches when the dynamic types of the given parameters are subtypes of the types expected by the declaration. For example, in Fig. 5, line 2, we construct a union of a two-dimensional box (solid rectangle) b and a circle c. The path-dependent type declarations make sure that the box and the circle share the same space. A box is solid and bounded, while a circle is bounded, but not solid. For this particular combination of the dynamic types of the parameters to the constructor of Union only the declarations of Union at lines 1, 8, and 15 in Fig. 4 match. The types of fields pt1 and pt2 in the Box types are not specified, which means that the most general types of these fields from the declarations of Box is assumed.

In the example, the static type of u is Union(s: b.s, s1: b, s2: c), which is in fact the most precise type available for the object that we assign to u. This type says that the object is a Union of shapes b and c, and its space is same as the space of b. Given this type declaration, the type checker can infer that u is an instance of all matching declarations of Union as well as their direct and indirect superclasses: Bounded, Solid and Shape. Any operation declared within any matching class declaration is available for u. Hence, the type checker will consider all calls on u in the method test as safe.

If less information is available statically, the operational behavior will still remain the same - we support true subtype polymorphism with late binding. For example, if the static type of the parameters b and c above would be Shape(s: Space) and Shape(s: b.s), respectively, the operational behavior of the methods would remain the same as the one in Fig. 5 (e.g., the same implementation of pointInside is executed), except that the calls to boundBox and getContent would be rejected by the type checker.

2.3 Dependent Classes and Virtual Classes

Virtual classes can be encoded with dependend classes, just as single-dispatched methods can be encoded in a multi-dispatch language: Dependent classes with exactly one parameter correspond to traditional virtual classes. A dependent class with only one parameter can hence be encoded as virtual class by nesting it inside the declaration of the class it depends on.[4] For our example, this means that definitions of Point, Vector, Shape, and so on, must be nested within definitions of Space, 2DSpace and 3DSpace.

However, such an encoding has a severe drawback, because nesting has a negative impact on software properties such as coupling and extensibility. Nesting requires to cluster all classes that depend on a particular class, thus introducing redundant dependencies. For example, Point and Vector must be implemented within the same class together with much more specific classes, e.g., Shape and its concrete subclasses. Furthermore, nesting limits extensibility: With each new type of shape or some other class that depends on Space, the latter and its subclasses must be modified. Dependent classes do not have these problems, because they are defined outside the declarations of classes they depend on.

Because of the nested structure, virtual classes are conceptually seen as an inherent part of the enclosing class. Such a view limits the applicability of virtual classes, because there are situations, where a class definition may depend on another class without being an integral part of it. For example, if we consider the Adapter design pattern [14], the implementation of an adapter may depend on the type of its adaptee object. To express this dependency with virtual classes, we would have to define adapters as nested virtual classes of adaptees. This obviously does not make sense: Adaptees would know about their adapters, which contradicts the main design goal of the Adapter pattern. Hence, virtual classes cannot be used for expressing dependencies of adapters on adaptees, if necessary.

[4] There are variations in the semantics of different formalisms and languages supporting virtual classes, thus the encoding possibilities are also different. More on this in Sec. 6.

Since dependent classes can be defined outside the classes they depend on, they do not need to be considered as their logical part. This shift of view opens up new application areas for dependent classes in addition to situations where virtual classes are useful [10, 24, 30, 31, 1, 33, 11, 26]. An adapter e.g., can be easily defined as a dependent class of its adaptee, enabling polymorphic selection and dependent typing of the adapter.

The adapter-adaptee relation is also addressed by expanders [35], where an adapter is defined as an expander of its adaptee. In some sense, expanders could be seen as dependent classes of the objects that they expand. However, expanders share the identity of the objects they expand, while dependent classes construct new objects, which can have a many-to-one relationship with their parameter objects. That is, by using dependent classes one can create multiple adapter instances of the same type for the same adaptee. For illustration, consider the situation where there are multiple views on some data model and adapters are used to adapt model classes to the abstractions of the view. If such adapters are stateful, an adapter object is needed for each pair of a view and a data model object.

Nesting also limits the possibilities to express variation. Every class can only be nested inside one class; hence, variations can be expressed along one dimension only. Dependent classes can vary along several dimensions simultaneously. For instance, we have demonstrated that the interface and the implementation of Union depends on the dynamic type of two shapes.[5] This cannot be easily encoded in a model of virtual classes with strict hierarchical nesting.

For an example of an application, where dependency from types in different inheritance hierarchies is needed, consider again the example of using adapters to adapt model classes to view abstractions. As long as there is a single view definition, single dependency of an adapter on its adaptee is sufficient. However, one can envisage an inheritance hierarchy of views, expressing variations on how model elements are displayed. In this case, the functionality of the adapters may also depend on the type of the view in addition to the type of the model element being adapted.

Our experience with using CaesarJ to model such scenarios [1] shows that it is indeed cumbersome to express dependencies along two variation axes with virtual classes – one has to resort to some sort of conditional logic for compensating limited dispatch power.[6] In contrast, adapters that depend both on target and adaptee are naturally expressed by dependent classes with two parameters: one for the view and another for the adaptee (the data model object, in our case).

[5] Union could also be refined for different kinds of space.

[6] In [1], we used a mechanism called *dynamic wrapper selection* to address this problem.

2.4 Dependent Classes and Multimethods

Since dependent classes and multimethods share similar semantics for dispatching functionality, the question arises as how they interact in terms of language design. We believe that dependent classes generalize multimethods in two ways.

Application scenarios that typically use multimethods can benefit from dependent classes in situations when operations dispatched by multiple parameters need to be reified. Our running example could be seen as an application of dependent classes to reify union and intersection of Shapes, which would typically be implemented as multi-methods.

There can be different reasons for reifying operations: To postpone computation (Command design pattern[14]), to cache computation results, or to provide mutable attributes that influence the computation. Our reification of union was mostly driven by a design decision: In our scenario, a union of two shapes is not so much a computation, but rather an abstraction with its own interface (e.g., we can compute its content, bounding box, etc.).

One could envisage reifications of other operations. For example, consider a draw operation defined on Shape, which takes the output medium as a parameter (e.g. a screen, a PDF or some drawing format); we could reify it by constructing a ShapeDrawer class that depends on two parameters – the shape and the output medium. Such a class could have additional display attributes as well, e.g., the color or line style that depend on some external input. The reification of the draw operation would allow to enrich the drawing functionality with additional behavior, e.g., the capability to cache intermediate computation results that are necessary for shape display (e.g., a triangulation of a 3D object), and/or to observe changes in its shape to update the display.

There is a second way in which dependent classes generalize multi-methods. Standard multi-methods [6] use "plain" types, as illustrated in Fig. 6. Multi-methods using dependent rather than plain classes for type declarations are more powerful with respect to both static and dynamic semantics.

```
abstract Shape union(Shape s1, Shape s2);
Shape union(Box s1, Box s2) { ... }
Shape union(Box s1, Circle s2) { ... }
```

Figure 6. Shape union as a "standard" multi-method

First, dependent classes extend the type system and would allow for more precise signatures of multi-methods. For example, in the signature of the union multi-method, they could be used to require that only a union of shapes from the same space can be computed. The same constraint cannot be statically stated and enforced in a language with multi-methods and plain types. Furthermore, in such a language, one cannot express dependencies of the result type of a multi-method on the types of its parameters. Such an explicit dependency declaration would allow the type checker to know e.g., that a union of a Box and a Circle is a solid and

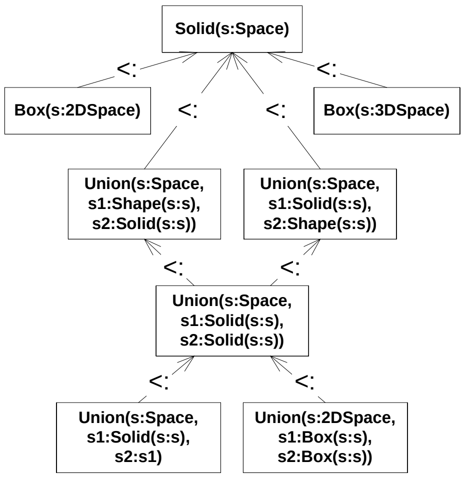
Solid(s:Space)
Box(s:2DSpace)
Box(s:3DSpace)
<:
Union(s:Space, s1:Shape(s:s), s2:Solid(s:s))
Union(s:Space, s1:Solid(s:s), s2:Shape(s:s))
Union(s:Space, s1:Solid(s:s), s2:Solid(s:s))
Union(s:Space, s1:Solid(s:s), s2:s1)
Union(s:2DSpace, s1:Box(s:s), s2:Box(s:s))

Figure 7. Types by which getContent is dispatched

bounded shape, and thus it is safe to call on it methods that are available for both Solid and Bounded types (cf. Fig. 5).

Second, dependent classes also increase the expressive power of dispatch. The dispatch of a multi-method selects the implementation of the method that is the most-specific among those that match. Since selection and matching is done by comparing parameter types by the subtype relation, an extension of the subtype relation also extends the expressive power of dispatch.

For illustration, consider the definitions of getContent in Fig. 3 and Fig. 4. They are available for a variety of types that are depicted in Fig. 7 together with their subtype relations. The subtype relation of dependent classes takes into account not only explicit inheritance relations (e.g., Box extends Solid); it also recursively compares the types of the fields and takes into account path types. In this respect, dispatch based on dependent classes can be seen as a variant of predicate dispatch [13] with predicates that express field types and identities between objects denoted by paths.

3. Semantics

In this section, we present a formal calculus that precisely describes the dynamic and static semantics of dependent classes. This calculus is called vc^n to indicate that it generalizes previous formalizations of virtual classes [12]. The style and notation is similar to the one of Featherweight Java [17]. A bar above a metavariable denotes a list: $\overline{f}$ stands for $f_1, \ldots, f_k$ for some natural number $k \geq 0$. If $k = 0$ then the list is empty, denoted by ϵ. Following common convention, $\overline{t}\ \overline{f}$ represents a list of pairs $t_1\ f_1 \cdots t_k\ f_k$. List notation is also used to denote repeated application to all members of a list; for example, $\Gamma \vdash \overline{e} : \overline{t}$ denotes the conjunction of all $\Gamma \vdash e_i : t_i$ for each list index i. To keep the notation lightweight we assume a globally available program P.

3.1 Syntax

The syntax of vc^n is defined on the left-hand side of Fig. 8. We have made a few design decisions to keep the calculus simple in order to focus just on the core semantics of dependent classes, and to ease the soundness proofs. For the informal explanation of the concepts and the examples in Sec. 2 we used an informal language that is close to the syntax of Java to make the examples more accessible to a broader public. Besides we used various language features that are not interesting from a semantics perspective, but are useful for practical programming. For example, various predefined types, such as int, double, String, and operations on them are not available in the formal calculus. In the following, we explain the formal syntax and its differences from the informal language.

A program P in vc^n consists of multiple class declarations. A class declaration, D, starts with a class name (note that the class keyword is skipped), followed by a list of field declarations and the return type of the class constructor. The list of declared fields also specifies the list of constructor parameters. A class can have an arbitrary number of super-classes specified in its **extends** clause. The body of a class declaration contains its constructor expression, which is called when the class is instantiated.

There is no special syntactical category to encode methods. As usual in formal accounts of virtual classes [32, 12], we use the syntax of class declarations to encode both classes and methods. A method declaration is encoded in vc^n by a class declaration: Method parameters are encoded as constructor parameters, the return type as the constructor return type and the implementation as the constructor expression. For "normal" classes, i.e., those that do not encode methods, we assume the expression this to be the default constructor body, i.e., the constructor simply returns the constructed object. The default return type is the empty path ϵ – the path pointing to this. Method calls are encoded as constructor calls. Multimethods can hence be encoded by using class declarations as methods.

In the calculus all declarations are at the top level, which means that it is not possible to nest methods within class declarations. However, the nested style can be easily translated to the parametric style. The implicit this parameter of nested methods is replaced by an additional explicitly declared parameter. An example of such translation will be given at the end of the subsection.

A *path*, referred to by p or q, is a sequence of fields; it refers to the object that is reached by navigating over the fields in the sequence starting from **this**. As a special case, the empty path ϵ refers to **this**.

A type, referred to by t or u, can be a class type $C(\overline{f} : \overline{t})$, a path p, or a value v. The *class type* $C(\overline{f} : \overline{t})$ represents all objects of C and its subclasses, whose fields f_i have values compatible with the respective types t_i. The only instance of

Syntax:

$$
\begin{array}{lll}
P & ::= & \overline{D} \\
D & ::= & C(\overline{f} : \overline{t}) : t\ \textbf{extends}\ \overline{C}\ \{e\} \\
p, q & ::= & \overline{f} \\
t, u & ::= & p \mid C(\overline{f} : \overline{t}) \mid v \\
e & ::= & \textbf{this} \mid e.f \mid \textbf{new}\ C(\overline{f} = \overline{e}) \mid v \\
v & ::= & C(\overline{f} = \overline{v})
\end{array}
$$

C – class names
f – field names

Context:

$$\Gamma ::= C(\overline{f} : \overline{t}) \mid \varnothing$$

Computation:

$$C(\ldots f_i = v_i \ldots).f_i \hookrightarrow v_i \quad \text{(RED-FIELD)}$$

$$\frac{\ldots \{e\} \in \text{Select}(C(\overline{f} = \overline{v}))}{\textbf{new}\ C(\overline{f} = \overline{v}) \hookrightarrow [C(\overline{f} = \overline{v})/\textbf{this}]\ e} \quad \text{(RED-NEW)}$$

Congruence:

$$\frac{e \hookrightarrow e'}{e.f \hookrightarrow e'.f} \quad \text{(REDC-FIELD)}$$

$$\frac{e \hookrightarrow e'}{\textbf{new}\ C(...f = e...) \hookrightarrow \textbf{new}\ C(...f = e'...)} \quad \text{(REDC-NEW)}$$

Select Declaration:

$$\frac{\varnothing \vdash D \in \text{Match}(C(\overline{f} : \overline{v}))}{D \in \text{Select}(C(\overline{f} = \overline{v}))} \quad \text{(SELECT)}$$

Figure 8. Syntax and operational semantics

a path type is the object referenced by the path. A *value type*, v, has the value v as its only member.

Types that contain paths are called *relative types*, as they are defined only relative to some object (referred to by **this**). On the contrary, *absolute types* are combinations of class and value types. For instance, Vector(s: v1.s) is a relative type, whereas Vector(s: 3DSpace) is an absolute type.

Most type relations of the calculus are defined relative to a typing context Γ, which is either empty ($\varnothing$) or defines the type of **this**. Relative types make sense only in a non-empty context, while an empty context can be used with absolute types. During static type checking (program well-formedness), the context will always be non-empty, whereas during runtime checking (typing intermediate expressions during evaluation) the expression **this** does not occur (it is replaced by a value) and the context will always be empty.

The calculus requires that in a class type the types of all fields that are available in this class are specified (this requirement is enforced in the well-formedness rules). In the informal language we allow to omit some of the field type annotations. In this case, the field types from the declaration of the class are assumed. In case of multiple class declarations, we assume the types of the most general declaration of the class as the default type of that field. An alternative solution would be to mark one of the class declarations which contains the default field types explicitly with some default keyword.

Like Featherweight Java [17], vc^n supports only functional style object-oriented programming. Classes have only immutable fields that are at the same time their constructor parameters. The body of a constructor is hence an expression, rather than a list of statements.

An expression e can be **this**, a field access, a class constructor call, or a value v. We use the new keyword to mark constructor calls and distinguish them from values. In the formal syntax, constructor calls take parameters by name,

```
Shape(s: Space) extends ε : ε { this }

pointInside (sh: Shape(s:Space), pt: Point(s:sh.s))
     : Bool { False }

Union(s: Space, s1: Shape(s:s), s2: Shape(s:s)) extends Shape
     : ε { this }

pointInside (
     sh: Union(s:Space,s1:Shape(s:sh.s),s2:Shape(s:sh.s)),
     pt: Point(s:sh.s)) : Bool {
  new or(a = new pointInside(sh=sh.s1, pt=pt),
         b = new pointInside(sh=sh.s2, pt=pt))
}
```

Figure 9. Example in the formal syntax

rather than by position. This makes it easier (in fact: trivial) to define the mapping from constructor parameters to field names, which would otherwise be cumbersome in the presence of multiple inheritance.

A value v is a class name together with values for its fields. Values can be used both as expressions and as types, but they are not part of the written syntax: They occur as expressions only in intermediate programs during rewriting and as types of intermediate programs containing values (we use a small-step operational semantics).

In the informal language we specified certain classes and methods as abstract. The informal meaning for a class being abstract is that it cannot be instantiated, while an abstract method has no implementation, but it can be called. However, formalization of abstract dependent classes is postponed for future work. Nevertheless, we decided to use abstract annotations in the informal language, because they significantly increase the understandability of the examples. When converting the examples to the calculus, abstract methods must be encoded as concrete methods with a default implementation.

For illustrating the encoding of the informal language in the calculus, consider the example in Fig. 9. It shows how the Shape declaration from Fig. 3 and the first declaration of Union from Fig. 4 could be encoded in the formal syntax. The implementations of the pointInside method are taken out from the class declarations and declared on the top level. Their implicit this parameter is encoded by an explicit sh parameter with an appropriate type. The declarations of Union and Shape are extended with the default constructor and the default return type. The declaration of Shape also receives an extends clause with an empty list of parents. The abstract pointInside method of Shape is replaced by a method with a default implementation. The method calls in the implementation of pointInside for Union are replaced by corresponding constructor calls.

3.2 Operational Semantics

The operational semantics in small-step style is given on the right-hand side of Fig. 8. There are only two computation rules: field access (RED-FIELD) and constructor call (RED-NEW). The other two reduction rules are just congruence rules.

Field access applied to a value, $C(\ldots f_i = v_i \ldots)$, is resolved by looking up the value of the field. The reduction of a constructor call uses the Select relation to select a declaration of class C for the given parameter values $\overline{v}$.

The Select relation is responsible for selecting one of the declarations that match the given parameter values. The set of matching declarations is defined by the Match relation, which will be discussed in Sec. 3.5. Intuitively, a declaration matches a given set of parameter values to a constructor call, if the types of these parameter values are more specific than the corresponding field types of the declaration.

Once a matching declaration is selected, the evaluation proceeds with the expression of the selected class declaration e, whereby **this** in e is replaced by the value of the constructed object.

The definition of Select determines the dispatch strategy. The non-deterministic strategy [5] in Fig. 8 (any matching declaration can be selected) is the most general definition that is sufficient to prove soundness of the calculus. We have proved that any definition for Select, that fulfills the following condition is sound: Whenever a well-formed type has any matching declarations, then the selection for this type succeeds and selects one of the matching declarations. Different choices in the design space of the dispatch mechanism will be discussed in Sec. 5.

3.3 Path Normalization and Type Equivalence

To determine whether two dependent types are equivalent, it is necessary to define an equivalence relation on those kinds of expressions that types may depend on. Type systems that allow types to depend on arbitrary, possibly non-terminating, expressions are often undecidable. Types in vc^n may only depend on path expressions, and for the latter a decidable equivalence relation can be defined, as shown in Fig. 10.

Two paths are equivalent (rule $\simeq$-PATH) if they have the same *normal form*; the definitions for type equivalence (rules $\simeq$-VALUE, $\simeq$-CLASS, and $\simeq$-PATH) just propagate path equivalence to the type level.

Intuitively, a path is in a normal form, if neither the path itself, nor any part of it, are declared as aliases of other paths. Accordingly, path normalization (rules $\leadsto$-FIELD1, $\leadsto$-FIELD2) can be seen as the process of eliminating alias paths. To normalize a path, declarations of field types from the context are used: For each path that is valid in the context, we can determine a type which is declared as its bound in the context (see path bound rules in Fig. 10). The bound of a normalized path is always a class type, because only class types are allowed in typing contexts.

For illustration, consider the context Union(s: Space, s1: Shape(s:s), s2: Shape(s:s)). The class type Shape(s:s) is declared in it as the bound of the path s1; by navigating further in the context, we determine that the bound of s1.s is the path s (see rule $\prec$-FIELD), which means that s1.s is declared as an alias of s and can hence be normalized to s. An excerpt of the derivation of this path normalization is shown below:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma \vdash \mathsf{s1} \leadsto \mathsf{s1}} \quad \dfrac{\cdots}{\Gamma \vdash \mathsf{s1} \prec \mathsf{Shape(s{:}s)}}}{\Gamma \vdash \mathsf{s1.s} \prec \mathsf{s}}\,(\prec\text{-F}) \qquad \dfrac{\dfrac{\cdots}{\Gamma \vdash \mathsf{s} \prec \mathsf{Space}} \quad \Gamma \vdash \epsilon \leadsto \epsilon}{\Gamma \vdash \mathsf{s} \leadsto \mathsf{s}}\,(\leadsto\text{-F1})}{\Gamma = \mathsf{Union(s{:}\ Space,\ s1{:}\ Shape(s{:}s),\ s2{:}\ Shape(s{:}s))} \vdash \mathsf{s1.s} \leadsto \mathsf{s}}\,(\leadsto\text{-F2})$$

We have proved the following theorem which characterizes the meaning of path equivalence: *Two paths are equivalent if and only if they are indistinguishable by the operational semantics in all extensions of the program.* The "only if" direction is required for type soundness; the "if" direction is a completeness property which states that the path normalization is optimal, i.e., any bigger path equivalence relation would be unsound. We give a formal statement of the completeness property in Sec. 4.2.

3.4 Type Translation

Fig. 11 defines the translation of a type t relative to another type u. Intuitively, the *translation* of a type t relative to u can be thought of as reinterpreting t by assuming u as the type of **this**; technically, it is achieved by replacing the occurrences of ϵ (the path pointing to **this**) by u. Type translation is needed to adapt types from a non-local context to the local context. For instance, the declared types of constructor parameters need to be translated to the context of a constructor call.

Two kinds of translations are defined in Fig. 11: strong translation denoted by $[t]_u$, and weak translation denoted by $\lceil t \rceil_u$. Whereas strong translation only allows to replace ϵ with a path ($[\cdot]$-PATH) or a value ($[\cdot]$-VALUE), weak translation additionally allows to replace ϵ with a class type ($\lceil \cdot \rceil$-

Path Bound:

$$C(\overline{f}:\overline{t}) \vdash \epsilon \prec C(\overline{f}:\overline{t}) \quad (\prec\text{-THIS})$$

$$\frac{\Gamma \vdash p \rightsquigarrow p' \qquad \Gamma \vdash p' \prec C(\overline{f}:\overline{t})}{\Gamma \vdash p.f_i \prec t_i} \quad (\prec\text{-FIELD})$$

Path Normalization:

$$\Gamma \vdash \epsilon \rightsquigarrow \epsilon \quad (\rightsquigarrow\text{-THIS})$$

$$\frac{\Gamma \vdash p.f \prec C(\overline{f}:\overline{t}) \qquad \Gamma \vdash p \rightsquigarrow p'}{\Gamma \vdash p.f \rightsquigarrow p'.f} \quad (\rightsquigarrow\text{-FIELD1})$$

$$\frac{\Gamma \vdash p.f \prec p' \qquad \Gamma \vdash p' \rightsquigarrow p''}{\Gamma \vdash p.f \rightsquigarrow p''} \quad (\rightsquigarrow\text{-FIELD2})$$

Type Equivalence:

$$\frac{\Gamma \vdash p \rightsquigarrow p'' \qquad \Gamma \vdash p' \rightsquigarrow p''}{\Gamma \vdash p \simeq p'} \quad (\simeq\text{-PATH})$$

$$\frac{\forall i.\ t_i \simeq t'_i}{\Gamma \vdash C(\overline{f}:\overline{t}) \simeq C(\overline{f}:\overline{t'})} \quad (\simeq\text{-CLASS})$$

$$\Gamma \vdash v \simeq v \quad (\simeq\text{-VALUE})$$

Figure 10. Path- and Type Equivalence

Strong Type Translation:

$$[p']_p = p.p' \quad ([\cdot]\text{-PATH})$$

$$\frac{\forall i.\ [t_i]_t = t'_i}{[C(\overline{f}:\overline{t})]_t = C(\overline{f}:\overline{t'})} \quad ([\cdot]\text{-CLASS})$$

$$[v]_t = v \quad ([\cdot]\text{-VALUE})$$

$$[\epsilon]_v = v \quad ([\cdot]\text{-VALUETHIS})$$

$$\frac{[p]_{t_i} = t'}{[f_i.p]_{C(\overline{f}:\overline{t})} = t'} \quad ([\cdot]\text{-CLASSFIELD})$$

$$\frac{[p]_{v_i} = t'}{[f_i.p]_{C(\overline{f}=\overline{v})} = t'} \quad ([\cdot]\text{-VALUEFIELD})$$

Weak Type Translation:

$$\frac{[t']_t = t''}{\lceil t' \rceil_t = t''} \quad (\lceil\cdot\rceil\text{-WEAKEN})$$

$$\frac{\forall i.\ \lceil t_i \rceil_t = t'_i}{\lceil C(\overline{f}:\overline{t}) \rceil_t = C(\overline{f}:\overline{t'})} \quad (\lceil\cdot\rceil\text{-CLASS})$$

$$\frac{\lceil p \rceil_{t_i} = t'}{\lceil f_i.p \rceil_{C(\overline{f}:\overline{t})} = t'} \quad (\lceil\cdot\rceil\text{-CLASSFIELD})$$

$$\lceil \epsilon \rceil_{C(\overline{f}:\overline{t})} = C(\overline{f}:\overline{t}) \quad (\lceil\cdot\rceil\text{-CLASSTHIS})$$

Figure 11. Type translation

CLASSTHIS). If both $[t]_u$ and $\lceil t \rceil_u$ are defined for some types u and t then $[t]_u = \lceil t \rceil_u$ holds.

Strong translation is used in situations where replacing the assumed type of **this** by a subtype must not change the result, such as the expected type of a constructor parameter – the constructor call must still be valid when the type to be instantiated is a subtype of what is statically known. Weak translation is used where this strong guarantee is not needed, such as for a field access.

3.5 Subtyping

Subtyping rules are shown in Fig. 12. Subtyping has deliberately been defined in an algorithmic style in order to demonstrate decidability. In particular, there is no subsumption or transitivity rule; rather, transitivity follows as a lemma.

Two equivalent types are subtypes of each other (<:-EQUIV). This is the only rule that accepts a path or a value as a supertype: The value itself is the only subtype of a value type and a subtype of a path must be an equivalent path.

The comparison of a value type with a class type (<:-VALUECLASS) is defined in terms of comparing two class types by replacing the value with the most specific class type that is compatible with it.

The comparison of a path p with a class type (<:-PATHCLASS) is the most sophisticated subtyping definition. The comparison is reduced to a class type comparison. For this purpose, the most specific class type that is a supertype of p must be constructed. A class type for p could be computed as the bound of the normalized p. However, this type is too weak to type-check many interesting programs. For illustrating the issue, consider the following example:

```
parallel (v1: Vector(s: Space), v2: Vector(s: v1.s)) : Bool {...}

test (v: Vector(s: Space)) : Bool { parallel (v, v); }
```

The function parallel tests if two vectors are parallel. It expects two vectors from the same space as parameters. The function test calls parallel to check if a vector is parallel to itself. If we use the path bound as discussed above, this

Subtyping:

$$\frac{\Gamma \vdash t \simeq t'}{\Gamma \vdash t <: t'} \qquad (\text{<:-E}\textsc{quiv})$$

$$\frac{\begin{array}{c}\forall i.\ \exists j.\ f'_j = f_i \ \wedge\ \Gamma \vdash t'_j <: t_i \\ C \in \mathrm{Parents}(\Gamma, C', \overline{f'} : \overline{t'}, \varnothing) \\ \Gamma \vdash C'(\overline{f'} : \overline{t'})\ \mathrm{OK}\end{array}}{\Gamma \vdash C'(\overline{f'} : \overline{t'}) <: C(\overline{f} : \overline{t})} \qquad (\text{<:-C}\textsc{lass})$$

$$\frac{\begin{array}{c}\Gamma \vdash p \rightsquigarrow p' \qquad \Gamma \vdash p' \prec C'(\overline{f'} : \overline{t'}) \\ \Gamma \vdash C'(\overline{f'} : \overline{t'})\ \mathrm{OK} \\ \Gamma \vdash C'(\overline{f'} : p'.\overline{f'}) <: C(\overline{f} : \overline{t})\end{array}}{\Gamma \vdash p <: C(\overline{f} : \overline{t})} \qquad (\text{<:-P}\textsc{ath}\text{C}\textsc{lass})$$

$$\frac{\Gamma \vdash C'(\overline{f'} : \overline{v'}) <: C(\overline{f} : \overline{t})}{\Gamma \vdash C'(\overline{f'} = \overline{v'}) <: C(\overline{f} : \overline{t})} \qquad (\text{<:-V}\textsc{alue}\text{C}\textsc{lass})$$

Match Declarations:

$$\frac{\begin{array}{c}D \in P \qquad C(\overline{f} : \overline{t'}) = \mathrm{Sig}(D) \\ \forall i.\ \exists t''.\ [t'_i]_{C(\overline{f}:\overline{t})} = t'' \ \wedge\ \Gamma \vdash t''\ \mathrm{OK} \ \wedge\ \Gamma \vdash t_i <: t''\end{array}}{\Gamma \vdash D \in \mathrm{Match}(C(\overline{f} : \overline{t}))} \qquad (\text{M}\textsc{atch})$$

Auxiliary Definitions:

$$\mathrm{Sig}(C(\overline{f} : \overline{t}) : u\ \textbf{extends}\ \overline{C}\{e\}) = C(\overline{f} : \overline{t}) \qquad (\text{S}\textsc{ig})$$

$$\frac{\begin{array}{c}C(\overline{f} : \overline{t}) \ldots \in P \\ \forall i, j.\ i \neq j \Rightarrow f_i \neq f_j\end{array}}{\mathrm{Fields}(C) = \overline{f}} \qquad (\text{F}\textsc{ields})$$

$$\frac{\begin{array}{c}\overline{f'} = \mathrm{Fields}(C) \qquad \overline{f'} \subseteq \overline{f} \\ \forall i, j.\ (f_i = f'_j) \Rightarrow (t_i = t'_j)\end{array}}{\mathrm{MakeType}(C, \overline{f} : \overline{t}) = C(\overline{f'} : \overline{t'})} \qquad (\text{M}\textsc{ake}\text{T}\textsc{ype})$$

Parents:

$$\begin{array}{c}\mathrm{Parents}(\Gamma, C, \overline{f} : \overline{t}, S) = \{C\} \cup (\bigcup_{C' \in S' \setminus S} \mathrm{Parents}(\Gamma, C', \overline{f} : \overline{t}, S \cup \{C\})) \\ \text{where } S' = \{C''_i \mid \Gamma \vdash \ldots \textbf{extends}\ \overline{C''} \ldots \in \mathrm{Match}(\mathrm{MakeType}(C, \overline{f} : \overline{t}))\}\end{array} \qquad (\text{P}\textsc{arents}\text{-D}\textsc{ef})$$

Figure 12. Subtyping

call would not pass the type checker: The actual type of the second argument passed to parallel is v, while its formal type, translated to the current context, is Vector(s: v.s). The bound of v in the context of the call is Vector(s: Space) (as specified in the declaration of test), which is not a subtype of the formal type Vector(s: v.s).

For this reason, the type of fields of $C'(\overline{f'} : \overline{t'})$ are further specialized. We know that each field f' actually has the type $p'.f'$. Hence, by substituting each $\overline{t'_i}$ for $p'.\overline{f'_i}$, a more specific class type is constructed, which is still a supertype of p. The resulting type is finally compared with $C(\overline{f} : \overline{t})$. In the example, Vector(s:Space) (the bound of v) is specialized to Vector(s: v.s).

Let us now focus on the rule for comparing class types (<:-CLASS) in Fig. 12. For a class type $C'(\overline{f'} : \overline{t'})$ to be a subtype of class type $C(\overline{f} : \overline{t})$, it must be well-formed in Γ (to be defined later), the types of the corresponding fields must be more specific, and C must be a *parent* of $C'(\overline{f'} : \overline{t'})$.

The function $\mathrm{Parents}(\Gamma, C, \overline{f} : \overline{t}, S)$ determines all parents of class C relative to field types $\overline{f} : \overline{t}$ in the context Γ. The function MakeType constructs a valid type from a class C and field types $\overline{f} : \overline{t}$ by selecting only those fields that are declared for class C. The idea of the algorithm implemented by the function $\mathrm{Parents}(\Gamma, C, \overline{f} : \overline{t}, S)$ is to collect superclasses from all the declarations of C that match the given field types, and then recursively collect the parents of these superclasses. Further, a class C is also considered a parent of itself. For example, the parents of Union for field types [s: 2DSpace, s1: Shape(s:s), s2: Solid(s:s)] are Solid, Shape, and Union.

The last parameter of the Parents function is an accumulator that remembers the classes already visited by the algorithm. By checking that no class is visited twice, termination is ensured even in the presence of cyclic inheritance relations. Since parents of a class are relative to the types of its fields, it can happen that for some type t holds $\Gamma \vdash C(f : t) <: C'(f : t)$, for another type t' we have $\Gamma \vdash C'(f : t') <: C(f : t')$ and for yet another type t'' both relations may hold simultaneously.

The relation Match defines the set of class declarations that match a class type $C(\overline{f} : \overline{t})$. It is used to determine which declarations contribute to a given class type. Intuitively, a declaration D matches a class type, if the type is of the same class as the declaration and the type is compatible with the type of **this** assumed by the declaration. A declaration assumes that the type of **this** is at least as specific as the signature of the declaration (see Fig. 12 for the definition of Sig). For a type $C(\overline{f} : \overline{t})$ to be compatible with the signature $C(\overline{f} : \overline{t'})$, the types $\overline{t}$ must be more specific than the types $\overline{t'}$.

However, the types $\overline{t}$ are defined relative to Γ, while the declared types $\overline{t'}$ are valid only in the context of the declaration. Thus, to compare these types, the declared types need to be translated to types relative to Γ. For a declaration D to match $C(\overline{f} : \overline{t})$, which is relative to Γ, the latter should be suitable as the type of **this** in the context of the declaration. Hence, we can use $C(\overline{f} : \overline{t})$ to translate $\overline{t'}$ to types relative to Γ; the translated types, t'', should be supertypes of the corresponding types from $\bar{t}$.

Strong translation guarantees that the translated types t'' are equivalent to $\overline{t'}$. To illustrate why strong translation is needed for matching, consider matching type u, which describes a Union of two arbitrary shapes, against D - the most general declaration of Union, which expects two shapes from the same space:

u =Union(s:Space, s1: Shape(s:Space), s2: Shape(s:Space))
D=Union(s:Space, s1: Shape(s:s), s2: Shape(s:s))

The declaration D should not match u, because its signature is more specific than u; it requires that this.s1.s = this.s2.s = this.s, which cannot be ensured by assuming **this** to be u. D is not included into the set of matching declarations of u, because strong translation $\lfloor \text{Shape(s:s)} \rfloor_u$ fails. If weak translation were used instead, D would incorrectly match, because $\lceil \text{Shape(s:s)} \rceil_u$ = Shape(s:Space).

We conclude this sub-section by a short consideration of the dependency of the operational semantics on the type system. Since Match is also used in the operational semantics (Select rule in Fig. 8), the question raises how much the operational semantics depends on the type system. The answer is that only a small subset of the typing rules is actually needed in the operational semantics. This is because the type to match in Fig. 8 is always a value type (a value used as type). Translation of any type relative to a value always produces an absolute type that does not contain paths. This means that for the operational semantics we just need to compare values with absolute types and never have to deal with paths.

3.6 Expression Typing and Well-Formedness

Fig. 13 specifies type assignment for expressions. The type of **this** is the empty path ϵ (TYPE-THIS). As usual for a small-step semantics, we also need a typing rule for values. The type of a value is the corresponding value type (TYPE-VALUE). Since values occur only during execution, this rule is only used for runtime type checking, i.e., for the preservation theorem (Sec. 4.1).

Typing of a field access expression (TYPE-FIELD) is performed by first computing the type t of the prefix e and then translating the type f relative to t. Weak translation produces the most specific type that captures all possible objects when navigating from any object of type t over field f.

For a constructor call expression (TYPE-NEW), the types of the actual parameters are computed. If there is any class declaration that matches these types, the return type t'', which is identical for all declarations of a class (as stated by the WF-PROG rule), is translated to the appropriate context (assuming the type of the constructed object as the type of **this**) and returned as the type of the expression. Again, weak translation is used, because it is safe to assign a more general type to an expression, if the precise type cannot be described.

The rules for well-formed types check (a) whether all paths exist in the given context (WF-PATH) by using path normalization and bounding, and (b) whether all classes exist (WF-CLASS) with matching field names.[7]

A class declaration is well-formed (WF-DECL), if all type declarations are well-formed in the context of the declaration. The constructor expression must be well-typed and its type must be a subtype of the declared return type. The set of fields in the class declarations must include all fields of direct superclasses. Further, it is required that values are not used in the types of fields. This ensures the property that the bound of a normalized path is always a class type.

Finally, two conditions are imposed on a program P in order for it to be well-formed (rule WF-PROG): (a) all declarations must be well-formed, and (b) all declarations of the same class must have the same sets of fields and identical return types.

4. Properties of *vc*^{n}

In this section, meta-theoretical properties of *vc*^{n} will be discussed: The soundness, the decidability, and expressiveness of the type system.

4.1 Soundness

We have used the standard method to prove the soundness of the calculus by a progress and a preservation theorem [36]. The progress theorem states that every well-typed expression in a well-typed program is either a value or can be further reduced. The preservation theorem ensures that if well-typed expression e is reduced to e', then the type of e' is a subtype of the type of e.

THEOREM 1 (Progress). *If* P OK *and* $\varnothing \vdash e : t$ *then* $\exists v.\ e = v$ *or* $\exists e'.\ e \hookrightarrow e'$

THEOREM 2 (Preservation). *If* P OK *and* $\varnothing \vdash e : t$ *and* $e \hookrightarrow e'$
then $\exists t'.\ \varnothing \vdash e' : t' \ \wedge\ \varnothing \vdash t' <: t$

The proofs of both theorems have been verified by the Isabelle/HOL proof assistant and are available for download at [15].

To understand why these theorems hold, we present a few key lemmas from the proof. Lemma 1 justifies the soundness of substituting **this** by the newly constructed object in (RED-NEW) (Fig. 8). It states that the type t' of the expression after the substitution is a subtype of the type of the expression before the substitution. The substitution of **this** in e changes the assumed type of **this** in e, thus the old type of e has to be translated relative to the v - the new type of **this**. The assumption $\varnothing \vdash v <: \lfloor u \rfloor_v$ should be read as: v is appropriate as value of **this** in the context where the type of **this** is u.

[7] The auxiliary function Fields is defined in Fig. 12.

Expression Typing:

$$\frac{\Gamma \vdash \epsilon \text{ OK}}{\Gamma \vdash \textbf{this} : \epsilon} \quad (\textsc{Type-This})$$

$$\frac{\Gamma \vdash e : t \qquad \lceil f \rceil_t = t' \qquad \Gamma \vdash t' \text{ OK}}{\Gamma \vdash e.f : t'} \quad (\textsc{Type-Field})$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{e} : \overline{t} \qquad \Gamma \vdash C(\overline{f} : \overline{t}) \text{ OK} \\ \Gamma \vdash C(\overline{f} : \overline{t'}) : t'' \ldots \in \text{Match}(C(\overline{f} : \overline{t})) \\ \lceil t'' \rceil_{C(\overline{f}:\overline{t})} = t''' \qquad \Gamma \vdash t''' \text{ OK}\end{array}}{\Gamma \vdash \textbf{new}\, C(\overline{f} = \overline{e}) : t'''} \quad (\textsc{Type-New})$$

$$\frac{\Gamma \vdash v \text{ OK}}{\Gamma \vdash v : v} \quad (\textsc{Type-Value})$$

Well-Formed Types:

$$\frac{\Gamma \vdash p \leadsto p' \qquad \Gamma \vdash p' \prec t \qquad \Gamma \vdash t \text{ OK}}{\Gamma \vdash p \text{ OK}} \quad (\textsc{WF-Path})$$

$$\frac{\text{Fields}(C) = \overline{f} \qquad \forall i.\ \Gamma \vdash t_i \text{ OK}}{\Gamma \vdash C(\overline{f} : \overline{t}) \text{ OK}} \quad (\textsc{WF-Class})$$

$$\frac{\Gamma \vdash C(\overline{f} : \overline{v}) \text{ OK}}{\Gamma \vdash C(\overline{f} = \overline{v}) \text{ OK}} \quad (\textsc{WF-Value})$$

Well-Formed Declaration:

$$\frac{\begin{array}{c}\Gamma = C(\overline{f} : \overline{t}) \\ \Gamma \vdash C(\overline{f} : \overline{t}) \text{ OK} \qquad \Gamma \vdash t \text{ OK} \\ \overline{t} \text{ does not contain values} \\ \Gamma \vdash e : t' \qquad \Gamma \vdash t' <: t \\ \forall i, \overline{f'}.\ \text{Fields}(C_i) = \overline{f'} \ \Rightarrow\ \overline{f'} \subseteq \overline{f}\end{array}}{C(\overline{f}, \overline{t}) : t \textbf{ extends } \overline{C}\ \{e\} \text{ OK}} \quad (\textsc{WF-Decl})$$

Well-Formed Program:

$$\frac{\begin{array}{c}\forall D \in P.\ D \text{ OK} \\ \left[\begin{array}{l}\forall D, D' \in P : \\ \quad D = C(\overline{f} : \overline{t}) : t \ldots \ \wedge\ D' = C(\overline{f'} : \overline{t'}) : t' \ldots \\ \quad \Rightarrow \overline{f} = \overline{f'} \ \wedge\ t = t'\end{array}\right]\end{array}}{P \text{ OK}} \quad (\textsc{WF-Prog})$$

Figure 13. Typing

LEMMA 1 (Substitution). *If* $u \vdash e : t$ *and* $\varnothing \vdash v <: [u]_v$ *and* $\varnothing \vdash v \text{ OK}$ *then* $\exists t'.\ \varnothing \vdash e[v/\textbf{this}] : t' \ \wedge\ \varnothing \vdash t' <: [t]_v$

Lemmas 2 and 3 state that matching declarations and subtyping relations are preserved at runtime. Lemma 2 states that if a declaration D matches a type t, then the match will be preserved at runtime for any possible value v of **this** (t is translated to a corresponding runtime type by translating it relative to the value of **this**). Analogously, lemma 3 states that subtype relations are preserved at runtime. Further, lemma 4 states that subtypes produce more matching declarations.

LEMMA 2 (Preservation of Matching). *If* $u \vdash D \in \text{Match}(t)$ *and* $\varnothing \vdash v <: [u]_v$ *and* $\varnothing \vdash v \text{ OK}$ *, then* $\varnothing \vdash D \in \text{Match}([t]_v)$

LEMMA 3 (Preservation of Subtyping). *If* $\varnothing \vdash v <: [u]_v$ *and* $\varnothing \vdash v \text{ OK}$ *and* $u \vdash t' <: t$*, then* $\varnothing \vdash [t']_v <: [t]_v$

LEMMA 4 (Monotonicity of Matching). *If* $\Gamma \vdash D \in \text{Match}(t)$ *and* $\Gamma \vdash t' <: t$*, then* $\Gamma \vdash D \in \text{Match}(t')$

Preservation of matching and preservation of subtyping hold due to the properties of strong translation. The definition of matching (rule MATCH in Fig. 12) computes the strong translation $[t_i']_{C(\overline{f}:\overline{t})}$ for each field type with respect to the assumed type of **this**, $C(\overline{f} : \overline{t})$. Replacing the assumed type of **this** by a subtype must not invalidate matching relations. This is the case due to an invariance property of strong translation: Only equivalent types will be produced when the context type is strengthened; hence, the subtype check in (MATCH) cannot fail.

LEMMA 5 (Invariance of Strong Translation). *If* $[t]_u = t'$ *and* $\Gamma \vdash t' \text{ OK}$ *and* $\Gamma \vdash u' <: u$*, then* $\exists t''.\ [t]_{u'} = t'' \ \wedge\ \Gamma \vdash t'' \simeq t'$

Weak translation has only a weaker property as stated in lemma 6. This property is, however, sufficient for soundness because weak translation is only used for field access and return types of constructors, where a loss in precision does not influence soundness.

LEMMA 6 (Covariance of Weak Translation). *If* $\lceil t \rceil_u = t'$ *and* $\Gamma \vdash t' \text{ OK}$ *and* $\Gamma \vdash u' <: u$*, then* $\exists t''.\ \lceil t \rceil_{u'} = t'' \ \wedge\ \Gamma \vdash t'' <: t'$

4.2 Completeness

In order for the type system to be sound, path normalization must have the property that equivalent paths are indistinguishable in the operational semantics. This can be expressed formally as follows:

LEMMA 7 (Soundness of Path Normalization). *If* P OK *and* $t \vdash t$ OK *and* $t \vdash p \simeq p'$*, then for all* v *with* $\varnothing \vdash v$ OK *and* $\varnothing \vdash v <: [t]_v$ *(think:* v *is a value of* **this** *that is allowed by* t*) and* $v.p \hookrightarrow_* v_1$ *and* $v.p' \hookrightarrow_* v_2$*, we have* $v_1 = v_2$.

However, even a very trivial path equivalence relation such as $\Gamma \vdash p \simeq p' :\Leftrightarrow p = p'$ would have this property. In order to demonstrate the expressiveness of path normalization, we have proven a completeness property,[8] which says that the implication also holds in the reverse direction if we also consider possible extensions of the program - a fixed program may be too limited to distinguish two paths. For this reason, we added the program that we are talking about in the formulas in the following lemma:

LEMMA 8 (Completeness of Path Normalization). *If* P OK *and* $P, t \vdash t$ OK *, then* $P, t \vdash p \simeq p'$ ***if and only if*** *for all extensions* $P' = P, P''$ *of* P *such that* P' OK *and all* v *with* $P', \varnothing \vdash v$ OK *and* $P', \varnothing \vdash v <: [t]_v$ *and* $P' \vdash v.p \hookrightarrow_* v_1$ *and* $P' \vdash v.p' \hookrightarrow_* v_2$*, we have* $v_1 = v_2$.

In other words, if two paths are operationally indistinguishable, then they will be equivalent in the type system. Since type equivalence is just path equivalence propagated to the type level (see $\simeq$-CLASS), the result applies to types as well.

4.3 Decidability

The definitions of most relations are syntax directed, i.e., at least one of the relation arguments in premises is a structural part of the relation arguments in the conclusion, while the other arguments remain unchanged. This applies to type equivalence, to strong and weak translation, and to expression typing. Decidability is less obvious for path normalization, for path bounding, and for subtyping. Hence, we will illustrate how these relations can be turned into terminating algorithms.

Figure 14 describes an algorithm for path normalization and bounding, which is equivalent to the rules in Fig. 10, i.e., if $\Gamma \vdash p \rightsquigarrow p'$, then $normalize(\Gamma, \varnothing, p) = p'$; if normalization is not possible, the algorithm generates an error (either raised explicitly or due to pattern matching failure).

It is easy to see that derivations of path normalization are unique. Therefore, if during normalization of p we encounter p again, then normalization of p is not possible and an error is raised. We use the second parameter Δ to keep track of the paths, the normalization of which can cause such cycles, and throw an error if we are about to normalize a path, which is already in Δ. The only place that can cause a cycle is the path normalization in the premises of rule $\rightsquigarrow$-FIELD2; all other premises normalize structurally smaller paths.

The algorithm always terminates, because Δ must grow with each recursive call on a path that is not structurally smaller. On the other hand, Δ cannot grow indefinitely, because it includes only paths computed by path bounding. It is easy to see that every possible path bound is a declared type in the context. Hence, the set of all path bounds in a fixed context is finite and the algorithm is guaranteed to terminate.

The subtyping definitions can directly be implemented by a recursive algorithm. Its termination can be proved by a measurement function that assigns a natural number to each pair of types, such that the measure of types, compared by a subtype relation in premises, is always smaller than the measure of the types, compared in the conclusion. Such a measure function is described in Fig. 15. It basically states that the depth of the type on the right-hand side of the subtype relation must decrease in at most two inference steps, if we do not count the intermediate inference rules for Match.

5. Dispatch

In this section, possible dispatch strategies are discussed, each answering the question as which of the class declarations matching a constructor should be selected for execution in a different way. All discussed strategies are specializations of the most general strategy defined by the function Select in Fig. 8. This means that our selection of a specific dispatch strategy does not compromise the soundness of the calculus, as long as it can guarantee that something will be selected from every valid matching set of declarations:

PROPERTY 1. *If* P OK *and* $\varnothing \vdash t$ OK *and* $\varnothing \vdash D \in Match(t)$ *for some declaration* D*, then there exists a declaration D' such that* $D' \in Select(t)$ *and* $\varnothing \vdash D' \in Match(t)$.

A reasonable assumption to expect from any dispatch strategy is that it implements overriding: More specific declarations should hide more general ones. Declaration D' is more specific than declaration D, if any value that matches D' also matches D:

DEFINITION 1. *Declaration* D' *is more specific than* D*, if for all* $v = C(\overline{f} = \overline{v})$ *with* $\varnothing \vdash v$ OK *and* $\varnothing \vdash D' \in match(C(\overline{f} : \overline{v}))$ *it follows that* $\varnothing \vdash D \in match(C(\overline{f} : \overline{v}))$.

Definition 1 describes the desired property of overriding, but it is not constructive, because it quantifies over all possible values. A constructive way to compare two declarations could be achieved by comparing their signatures by the

[8] The proof is available at [15].

$$\text{normalize}(\Gamma, \Delta, \epsilon) = \epsilon$$

$$\text{normalize}(\Gamma, \Delta, p.f) = \begin{cases} \mathit{error} & \text{if } p.f \in \Delta, \\ \text{normalize}(\Gamma, \Delta \cup \{p'\}, p') & \text{if } t = p', \\ \text{normalize}(\Gamma, \Delta, p).f & \text{if } t = C(\overline{f} : \overline{t}), \end{cases}$$
$$\text{where } \; t = \text{bound}(\Gamma, \Delta, p.f)$$

$$\text{bound}(C(\overline{f} : \overline{t}), \Delta, \epsilon) = C(\overline{f} : \overline{t})$$
$$\text{bound}(\Gamma, \Delta, p.f) = t, \text{ where}$$
$$C(\ldots f : t \ldots) = \text{bound}(\Gamma, \Delta, \text{normalize}(\Gamma, p))$$

Figure 14. Path Normalization Algorithm

$$\text{depth}(p) = 0 \qquad \text{depth}(v) = 0$$
$$\text{depth}(C(\overline{f}, \overline{t})) = \max(\text{depth}(\overline{t})) + 1$$

$$\text{measure}(t', t) = \begin{cases} 2 \times \text{depth}(t) + 1 & \text{if } t' \text{ is not a class type} \\ 2 \times \text{depth}(t) & \text{if } t' \text{ is a class type} \end{cases}$$

Figure 15. Measure function showing decidability of subtyping

subtype relation. The problem, however, is that the signatures may involve paths and, thus, cannot be compared in the global (empty) context. The solution is to use the signature of the declarations to compare as contexts, as in the constructive definition of the overrides relation below:

DEFINITION 2. *Declaration D' overrides D ($D' \ll D$), if* $\text{Sig}(D') \vdash \text{Sig}(D') <: \text{Sig}(D)$ *and not* $\text{Sig}(D) \vdash \text{Sig}(D) <: \text{Sig}(D')$.

The overrides relation has the property that D' overrides D implies D' is more specific than D. By using it, the definition of Select can be refined so that it guarantees that the overridden declarations are hidden (SELECT-OVER in Fig. 16). Given a non-empty set of declarations, a declaration can be found that is not overridden by any other declaration from the set. This is because the overriding relation is transitive and asymmetric. Thus, for such a definition of Select, constructor calls in a well-formed program will always succeed.

The rule SELECT-OVER is, however, not deterministic because there can be several declarations that do not override each other. There are different methods for eliminating this non-determinism. Following the tradition of multi-dispatch, these methods can be classified into symmetric and asymmetric ones.

Symmetric dispatch requires that only the most specific declaration can be selected. A possible definition of symmetric dispatch is given by rule (SELECT-SYMM) in Fig. 16. It requires that from the set of the matching declarations we can select one declaration that overrides all the others.

The type checking rules of Fig. 13 are not sufficient to guarantee that symmetric dispatch will always succeed. For example, the declarations Union(s:Space, s1:Solid(s:s), s2:s1) and Union(s:2DSpace, s1:Box(s:s), s2:Box(s:s)) in Fig. 4 do not override each other and both could match the type Union(s:2DSpace, s1:Box(s:s), s2:s1).

The simplest constructive way to guarantee that symmetric dispatch always succeeds, is to require that the overriding relation defines a total order on all declarations of the same class. The problem is that such a requirement is very strict, i.e., it rejects programs that fulfill property 1. We could try to borrow less restrictive solutions that are available for symmetric dispatch for methods [2, 4, 28]. It is, however, not straightforward, because the relations that are easy to compute in a simple type system, where types correspond to plain classes, may be difficult to compute or even undecidable in a type system with dependent types. For example, it is difficult to determine if two types are overlapping, i.e. if they have a common (valid) value in the program. It is also difficult to check if a type is an upper bound of an intersection of two other types (i.e. it is a supertype of all their common subtypes). Definition of tolerant constructive well-formedness rules that guarantee the property 1 for symmetric dispatch is a topic for future research.

The general principle of asymmetric dispatch is to define an additional ordering relation that supplements the order of the overriding relation. The ordering relation $D <_t D'$ says that D precedes D' when they are incomparable by overriding relation. In the general case, the ordering relation may be not absolute, but relative to t - the type being matched. Rule SELECT-ASSYM in Fig. 16 defines the general principle of asymmetric dispatch: the matching declarations are first filtered by the overriding relation, then from the declarations that are not overridden, we select the one that is the smallest by the additional ordering relation.

The supplementary ordering relation can be defined in different ways. The simplest way is to define it explicitly by assigning order numbers to declarations in the program or by having precedence declarations similar to the precedence declarations of aspects in AspectJ [21]. The order can also be determined implicitly, by considering the order of class declarations, the order of field declarations and the order of parent classes in the **extends** clause. For example, one could consider extending the mixin linearization algorithm of the *vc* [12] calculus for multiple fields.

In vc^n, one could also consider new variations on dispatch that would not make sense for multimethods. For example, classes that only contribute structure (such as new supertypes or fields) but not behavior (only default constructor) could be excluded from the dispatch algorithm. In vc^n, all declarations of a class must have the same set of fields, but if we would add self-initializing fields or mutable fields that do not need to be initialized via constructor parameters,

Select for overriding:

$$\frac{\varnothing \vdash D \in \mathrm{Match}(C(\overline{f} : \overline{v})) \qquad \forall D' \in \mathrm{Match}(C(\overline{f} : \overline{v})).\ \neg D' \ll D}{D \in \mathrm{Select}(C(\overline{f} = \overline{v}))} \ (\textsc{Select-Over})$$

Select for symmetric dispatch:

$$\frac{\varnothing \vdash D \in \mathrm{Match}(C(\overline{f} : \overline{v})) \qquad \forall D' \in \mathrm{Match}(C(\overline{f} : \overline{v})).\ D' \neq D \ \Rightarrow\ D \ll D'}{D \in \mathrm{Select}(C(\overline{f} = \overline{v}))} \ (\textsc{Select-Symm})$$

Select for asymmetric dispatch:

$$\frac{\begin{array}{c} X = \{D.\ \varnothing \vdash D \in \mathrm{Match}(C(\overline{f} : \overline{v})) \ \wedge\ (\forall D' \in \mathrm{Match}(C(\overline{f} : \overline{v})).\ \neg D' \ll D)\} \\ D \in X \qquad\qquad \forall D' \in X.\ D' \neq D \ \Rightarrow\ D <_{C(\overline{f}:\overline{v})} D' \end{array}}{D \in \mathrm{Select}(C(\overline{f} = \overline{v}))} \ (\textsc{Select-Asymm})$$

Figure 16. Variations of Dispatch

then a class declaration could extend the object layout without interfering with the dispatch mechanism.

6. Related Work

The idea of virtual classes stems from BETA [23] and became more popular since Ernst's paper on family polymorphism [10]. Since then, much related work on virtual classes has been published [19, 25, 1, 18, 30, 31, 12, 32, 8, 5]. The key difference between dependent classes and vc^n on the one side and related work on virtual classes on the other side [19, 25, 1, 18, 30, 31, 12, 32, 8, 5] consists in the generalization of the dependency on multiple parameters and the resulting implications for expressiveness and the type system (such as types that depend on multiple objects rather than a single object). We are not aware of any other previous work in this direction. Hence, in the following, we concentrate on those properties of vc^n that can be directly compared to other approaches via encoding of virtual classes by dependent classes. In other words, for this comparison we consider only dependent classes with a single parameter. Figure 17 gives a brief summary of the comparison.

Virtual classes as envisioned in Beta are properties of enclosing objects (*object families in Fig. 17*). Some approaches, Concord, .FJ and J&, propose a static version of virtual classes where nested classes are properties of enclosing classes. The implications of this difference are discussed in detail in [12].

The classes-in-objects approaches, CaesarJ, vc, νObj, FS_{alg}, Tribe and vc^n, differ from each other in their support for what we call *path wildcards*, referring to the ability to replace a path or part of a path in a type by a class name. For example, by using the type Space.Point instead of s.Point we refer to points whose class belongs to *any* instance of Space. Both Tribe and vc^n allow free mixing of paths and class names. These approaches are also the only ones supporting *free access to enclosing objects* in types, such as a type p1.s.Vector that refers to the enclosing object of a Point object p1. A detailed discussion why this is useful can be found in [5].

Cross-family inheritance (Fig. 17) refers to the ability to inherit from a class that is not part of the same family. This is relatively easy in static approaches, but becomes more complicated with true virtual classes because a virtual class can then have multiple enclosing objects. Tribe has special constructs (adoption and over-the-top types) to allow inheritance from top-level classes - this avoids the aforementioned complications because top-level classes do not have an enclosing object. In vc^n, a class can inherit from any other class, including classes that belong to arbitrary foreign families. It is even possible to encode *inheritance from a dependent type* by a superclass clause of the form extends path.C. This is illustrated in the following example, which encodes a hypothetical version of traditional virtual classes with inheritance from dependent types (top) in vc^n (bottom). In the encoding, E has an additional constructor parameter, but since it has a singleton type (a path), there is only one possible value which can be passed. In fact, it would be easy to devise an extension of vc^n, where constructor arguments with singleton type can be initialized automatically.

```
class C {
   class D { }
}
class E extends path.D { ... }
```

```
C
D(out: C)
E (...,  out: path) extends D
```

Decidability is not obvious when dealing with dependent type systems, and in fact some dependent type systems are undecidable. It is not trivial to deal with heaps and *mutable state* in path-dependent type systems, hence many formal languages have been formulated as pure functional systems. *Final bindings* increase the expressiveness of the type system with respect to encoding generics, but it is not easy to reconcile final bindings with polymorphic constructors and parents (superclasses), hence none of the languages supports both. Finally, the languages differ in whether arbitrary expressions can be used as receiver or argument of a constructor call (e.g., for *nesting constructor calls*). Some languages allow only paths in these positions and rely on encodings of arbitrary expressions through local variables. It is not always obvious, however, whether these encodings have the desired

	Concord [19]	Caesar/J [25, 1]^5	FJ [18]	J& [30, 31]	vc [12]	νObj [32]	FS_{alg} [8]	Tribe [5]	vc^n
Object Families	no	yes	no	no	yes	yes	yes	yes	yes
Path wildcards	n/a	yes	n/a	n/a	no	no	no	yes	yes
Free access to enclosing object	n/a	no	n/a	n/a	no	no	no	yes	yes^4
Cross-Family Inheritance	no	no	no	yes	no	no	no	yes^1	yes
Inheritance from dependent type	no	no	no	no	no	no	no	no	yes
Decidability	yes	yes	yes	yes	yes	no	yes	yes^3	yes
Mutable State	no	yes	no	yes^2	yes	no	no	no	no
Final Bindings	no	no	no	no	no	yes	yes	no	no
Polymorphic Parents and Constructors	yes	yes	yes	yes	yes	no	no	yes	yes
Free Constructor Nesting	yes	yes	yes	no	no	no	no	no	yes

Figure 17. Comparison with the single-dispatch fragment of vc^n

Parts of this table have been copied from [5]. ^{1}) but only from top-level classes; ^{2}) the formal language in [30] uses a heap but has no mutation; ^{3}) decidability is not discussed in [5], but the authors have subsequently developed a variant of Tribe with a decidability proof (personal communication); ^{4}) since enclosing objects (which are not lexically enclosing in vc^n) are stored in ordinary fields; ^{5}) there is no formal definition of Caesar/J, though

typing properties, hence we feel it is desirable to support free expression composition directly.

DEEP [16] is another language which can encode a variant of virtual classes, but it is very different from the other languages discussed above and hard to compare, since it allows general dependent types (not just dependencies on paths), uses a prototype-based approach rather than classes, supports virtual classes only through relatively elaborate encodings, and uses many unconventional techniques such as unifying terms and types.

As already discussed in Sec. 2, dependent classes are complementary to multimethods [9, 3, 34, 6] and cannot be replaced by them. Even encoding of the operational semantics of dependent classes by means of multi-dispatched factory methods is hard, because a factory method would need to be defined for each possible combination of the declarations of a dependent class. Even more problematic is the modeling of the subclasses of dependent classes, because a subclass implicitly inherits all variations of its superclass. Also, we believe that it is not possible to model the type system of vc^n using any of the type systems that have been proposed for multi-dispatch.

Predicate dispatch [13, 27] is a generalization of multi-dispatch. With predicate dispatch, a method can have multiple declarations with different predicate expressions. A method declaration is applicable to the argument values of a method call when the predicate expression evaluates to true. The dispatch selects the most specific declaration of the method that is applicable for the given arguments. The predicate expression of the most specific declaration must logically imply the predicate expressions of other declarations. Predicate dispatch is more powerful than the dispatch that we presented for dependent classes. It would be rather straightforward to generalize the operational semantics of vc^n to use predicate dispatch but it is not obvious what the implications for the type system would be.

7. Conclusions and Future Work

In the paper, we presented the concept of dependent classes, a generalization of virtual classes. The definition of a dependent class is parameterized by and dispatched over multiple parameters.

In addition to supporting multi-dispatch, the parametric style also breaks the strict nesting structure of virtual class declarations. This implies a new view, in which virtual classes are not seen as an inherent part of the enclosing object, but rather as classes that simply depend on the types of other objects. As a result, unnecessary coupling between classes and extensibility problems caused by nesting are avoided.

By their very idea dependent classes are related to multimethods in the sense that class definitions can depend on multiple parameters in a similar way as method definitions do. The vc^n calculus for dependent classes presented in this paper not only encodes the multi-methods, but also extends their semantics by supporting dispatch over a more powerful subtype relation.

This paper is the first one to present a mechanically verified soundness proof of a language with path-dependent types and virtual classes. We hope that the existence of this proof increases the trust and interest in virtual classes and makes it easy for other researchers to investigate new variants of virtual classes by adapting our proof.

There are several areas of future work. We will formalize abstract dependent classes and will investigate the issue of modular checking of dispatch completeness and uniqueness. In addition, we plan to investigate the relation to dependently-typed lambda calculi, such as the calculus of constructions [7]. First experiments suggest that it might indeed be possible to encode some typical theorem proving examples in vc^n. Also, we plan to investigate the relation between dependent classes and Haskell's multi-parameter type classes [20], in particular those variants with support for so-called *overlapping instances*, building upon the results presented in [22]. Another interesting issue is interaction between dependent classes and generics.

Another area will be to further study the usefulness of dependent classes in various application scenarios. For example, dependent classes are interesting in the context of supporting variability in software product lines, because they can be used to model classes that depend on multiple variation points.

8. Acknowledgment

This work is supported by European Commission grant IST-33710: Aspect-Oriented, and Model-Driven Product Line Engineering (AMPLE), 2006 - 2009, and by the German Ministry of Education and Research grant 01ISF20A (feasiPLe project).

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[2] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 182–192, New York, NY, USA, 1992. ACM Press.

[3] C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings ECOOP '92*, LNCS 615, pages 33–56. Springer, 1992.

[4] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. In *Proceedings OOPSLA '94*, pages 1–15, New York, NY, USA, 1994. ACM Press.

[5] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A Simple Virtual Class Calculus. In Proceedings of AOSD'07, 2007.

[6] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. *SIGPLAN Not.*, 35(10):130–145, 2000.

[7] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[8] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In *Proceedings MFCS*, Springer LNCS, Sept. 2006.

[9] L. DeMichiel and R. Gabriel. The Common Lisp Object System: An overview. In *Proceedings ECOOP '87*, pages 243–252, 1987.

[10] E. Ernst. Family polymorphism. In *Proceedings ECOOP '01*, pages 303–326, London, UK, 2001. Springer-Verlag.

[11] E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings ECOOP '03*, LNCS 2743, pages 303–329. Springer-Verlag, 2003.

[12] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *Proceedings POPL '06*, pages 270–282. ACM Press, 2006.

[13] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 186–211. Springer, 1998.

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

[15] V. Gasiunas, M. Mezini, and K. Ostermann. Formal soundness proof of the vc^n calculus, 2006. http://www.st.informatik.tu-darmstadt.de/static/pages/projects/mvc/index.html.

[16] D. Hutchins. Eliminating distinctions of class: using prototypes to model virtual classes. In *Proceedings OOPSLA '06*, pages 1–20. ACM Press, 2006.

[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 1999.

[18] A. Igarashi, C. Saito, and M. Viroli. Lightweight family polymorphism. In *Programming Languages and Systems, Third Asian Symposium (APLAS'05)*, pages 161–177. Springer LNCS 3780, 2005.

[19] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple dependent types: Concord. In *Workshop on Formal Techniques for Java-like Programs at ECOOP 2004*, 2004.

[20] S. P. Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In *Procedings of the Haskell Workshop 1997*, June 1997.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings ECOOP '01*, pages 327–353, London, UK, 2001. Springer-Verlag.

[22] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE'06*. ACM Press, Oct. 2006.

[23] O. L. Madsen and B. Möller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89*, pages 397–406. ACM Press, 1989.

[24] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02*, pages 52–67. ACM Press, 2002.

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

[26] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings SIGSOFT '04/FSE-12*, pages 127–136. ACM Press, 2004.

[27] T. Millstein. Practical predicate dispatch. In *Proceedings OOPSLA '04*, pages 345–364. ACM Press, 2004.

[28] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings ECOOP '99*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer Verlag, 1999.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, 2004.

[31] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *Proceedings OOPSLA '06*, pages 21–36. ACM Press, 2006.

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

[33] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings OOPSLA '05*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[34] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley, Redwood, CA, USA, 1996.

[35] A. Warth, M. Stanojevic, and T. Millstein. Statically scoped object adaptation with expanders. In *Proceedings OOPSLA '06*, pages 37–56. ACM Press, 2006.

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