Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Grace's Inheritance

James Noble[a] Andrew P. Black[b] Kim B. Bruce[c]
Michael Homer[a] Timothy Jones[a]

a. Victoria University of Wellington, Wellington, New Zealand
b. Portland State University, Portland, Oregon, USA
c. Pomona College, Claremont, California, USA

Abstract This article is an apologia for the design of inheritance in the Grace educational programming language: it explains how the design of Grace's inheritance draws from inheritance mechanisms in predecessor languages, and defends that design as the best of the available alternatives. For simplicity, Grace objects are generated from object constructors, like those of Emerald, Lua, and Javascript; for familiarity, the language also provides classes and inheritance, like Simula, Smalltalk and Java. The design question we address is whether or not object constructors can provide an inheritance semantics similar to classes.

1 Introduction

Inheritance is one of the defining features of object-oriented programming — indeed for Wegner [Weg87], inheritance moves a language from being "object-based" to "object-oriented." In this apologia, we examine the design space for inheritance in object-oriented languages, particularly when generative object constructors are the major form of object creation, as they are in Emerald, JavaScript, Lua, and as they are in Grace.

Although our aim is to present a language designers' apologia for the inheritance mechanisms in Grace — which have proved to be the most interesting, frustrating, and controversial parts of its design — the concepts involved are not limited to any particular language. We hope that this apologia will be useful not only to programmers trying to understand why inheritance in Grace has turned out the way it has, but also to designers of future object-oriented languages who are driven to explore this design space. While our intention is to be historically objective, we are aware that history is written by the winners, in this case, the winners in the marketplace of ideas. Each of us recalls the history a little differently; motivations and the attribution of ideas to people are unreliable, even at this short remove. Perhaps we should have fictionalized this whole account, as do the Modula-3 authors in *How the language got its Spots* [Nel91, Ch. 8]. Lamentably, we lack their skill.

We start by introducing the Grace programming language, particularly object constructors and the way in which they relate to classes. Section 3 presents a series of

James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, Timothy Jones. *Grace's Inheritance*.

In Journal of Object Technology, vol. V, no. N, 2011, pages M:1–33. doi:10.5381/jot.201Y.VV.N.aN

designs for adding single inheritance to Grace. Section 4 describes how we generalised the designs to support trait-based reuse. Section 5 revisits related work; Section 6 concludes.

2 Objects and Classes in Grace

Grace is designed for education; we have sought to keep it as simple as possible, so that classroom time can be spent on the essential difficulties of programming, and not on the accidental difficulties of the language. Following this philosophy, Grace objects are self-contained, that is, each has its own fields and methods, along with a unique identity. Other objects can interact with an object only through *requests*, which can be used to examine and update public fields, and to execute methods. From outside an object, fields and methods are indistinguishable. Requesting a method is essentially equivalent to what Smalltalk and Ruby call "sending a message", a term that we avoid because, in the age of the Internet, we find it more confusing than helpful.

The names of Grace methods, like Smalltalk method selectors, take a variety of syntactic forms: unary-prefix and binary operators, and *sequences* of one or more name-parts, interspersed with argument lists. All method requests have the same dynamic binding semantics: they are resolved by the receiving object. Grace uses the reserved word **self** to refer to the current object; when **self** is the receiver of a named request, the word **self** (and the following dot) may be omitted.

2.1 Generative Object Constructors

Grace, while superficially similar to Java, C++ and Scala, is founded on a radically simpler object model [BBHN12, NHBB13, BBH+13]. Specifically, Grace is *object*-based, not *class*-based. Grace does have classes, but they can be fully explained in terms of methods and objects. There is no class–instance relation in Grace: objects own their own methods, rather than obtaining them from classes.

For both pedagogic and practical purposes, we wanted Grace to support immutable objects as a fundamental building block, and not just as a special case of a mutable object that happened not to contain any mutable state. The pedagogic motivation was that immutable objects are fundamentally simpler than mutable ones; for example, their semantics does not require updatable state. The practical motivation had to do with parallelism: without updatable state, the implementation is free to make copies, and need not be concerned with synchronization. Although parallelism was not amongst our early design goals, we sought to avoid decisions that would make it harder to add parallelism when the time came.

Other languages without classes, such as Self, have been based on prototypes: new objects are created by first cloning an existing object, and then modifying its attributes. In Self, a family of logically immutable objects with different field values (e.g., immutable points that differ in their coordinates, or immutable colours that differ in their RGB values) must be mutable at the language level, so that their attributes can be assigned their initial values. Relying on modification to create *all* objects is fundamentally at odds with our desire for Grace to support truly immutable objects.

For this reason, we envisioned from the beginning that Grace objects would be created by *object constructors* [BBHN12]. An object constructor is an expression that, *when executed*, constructs a new object that contains the methods and fields given therein. For example:

```
object {
    var size := 1
    def myThreshold = currentThreshold
    method grow(n) { size := size + n }
    print "I've made a new object!"
}
```

Each time we evaluate this constructor, we create a new object. All of these objects have unique identities, the same structure, and potentially different field values; the constant field myThreshold will take on the value of currentThreshold, which must be defined in an enclosing scope.

Emerald was the first language to be based on object constructors that generate new objects [BHJL07]; OCaml [LDF+12] and JavaScript [WB15] have similar constructs. Compare them to the static object literals of languages like Scala [Ode11] or Self [US91], which are evaluated only once, and so create just a single object.

Because Grace is an imperative language, the expression on the right hand side of a variable declaration var v := expression or definition def c = expression can have arbitrary effects. We even allow it to refer to self. This can be risky, because self may not be well-formed at the time this code is executed; for example, the object's invariants may not yet have been established. Given the existence of these expressions, allowing arbitrary executable code to appear at the top level of an object constructor adds convenience, but no additional danger, so we decided to allow this too.

```
object {
    def ... = codeWithEffectsInvolving(self)
    method ...
    moreCodeWithEffectsInvolving(self)
}
```

This code can be used for housekeeping tasks involving the object under construction.

We had previously decided that code at the top level of a file should be treated as if were enclosed by object { ... }. Together, these two decisions have the happy consequence of making

```
print "Hello, world"
```

a complete Grace program — one of our early language design goals.

This design pays dividends in simplicity: you have already seen most of the syntax of Grace. Within an object constructor, programmers can declare fields and methods, and write executable code; the same rules for declarations and code apply at the top level of a file. Within a method the rules are almost the same; the exception is that methods can not be declared directly inside other methods. The other feature of consequence is the *block*, a special syntax for representing λ-expressions. In addition to being concise, the block syntax allows return inside a block to return from the *enclosing method*; this allows us to follow Smalltalk in defining control structures such as conditionals and loops as requests of methods, rather than as new primitives.

2.2 Classes

Although object constructors are the *primordial* source of objects in Grace, it also seemed important to support classes. Our reasoning was that class-based languages predominate, and students will eventually need to transition from Grace to one of

them. Moreover, some instructors may prefer to start teaching with classes rather than objects, even though *we* prefer to start with objects and then move to classes.

Experience with initially-classless languages such as Self [US91], Lua [IdFC07], JavaScript, and Emerald has shown that, when languages do not provide classes, programmers tend to build their own class-like constructs. This can lead to multiple incompatible class libraries [Cro08, SRV+15]. In some cases, classes have been added to the language itself, or to its programming environment. The Self system was eventually extended to include a *"Subclass Me"* command [Ung02], and a class construct was officially added to JavaScript in the ECMAScript 6 standard [WB15]. Even Emerald acquired classes, first as an *emacs* macro, and then as a parser extension [HRB+87].

Thus, we determined to include classes in Grace as a shorthand that can be wholly explained in terms of object constructors [BBHN12, HN12]. The class

```
class point(xCoord, yCoord) {
    def x = xCoord
    def y = yCoord
    method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
}
```

is equivalent to a method that returns a new object by invoking an object constructor:

```
method point(xCoord, yCoord) {
    object {
        def x = xCoord
        def y = yCoord
        method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
    }
}
```

OCaml also takes this approach: an OCaml class declaration is (almost) syntactic sugar for a function that tail-returns an object constructor [LDF+12].

So far, so good: we have both classes and objects, and ontologically objects precede classes. By this we mean that classes are defined in terms of objects, not the other way around.

2.3 An alternative: Classes before Objects

It would be quite possible to retain exactly the current syntax for objects and classes, but with the opposite ontological precedence: the traditional approach in which objects are defined in terms of classes. Java, for example, does not have generative object constructors, but its anonymous inner classes can have a closely analogous effect. The Java fragment

```
new Object() {
    final int x = xCoord;
    final int y = yCoord;
    float distanceFromOrigin { return sqrt((x^2) + (y^2)) };
}
```

looks pretty much like an object constructor, but is actually the definition, followed by the immediate instantiation, of an anonymous subclass of `Object`. As with the Grace object constructor, the variables `xCoord` and `yCoord` are assumed to be defined in an enclosing scope.

This alternative ontology would make every object the (possibly sole) instance of a (possibly anonymous) class. OCaml object constructors, for example, are defined

in this way [LDF+12]. An OCaml class is like a function that returns the result of an object constructor, but this is only an analogy, not a equivalence. In particular, OCaml classes and objects can *inherit* only from classes.

Ruby takes this design one step further: every object is the sole instance of an anonymous class called its *eigenclass* [Sha13]). In Ruby, creating an instance of a class conceptually creates a unique eigenclass inheriting from the class, and then instantiates the eigenclass. In practice, eigenclasses are created lazily, only when required.

We resisted defining Grace objects in terms of classes for both conceptual and practical reasons. Conceptually, that definition seemed backwards: in an object-oriented language, objects should be the primary entity, and classes should be "the boxes in which the objects are packed"[1]. Practically, making objects self-contained meant that Grace had no class-metaclass relationship. This meant that we did not have to decide on what the class of a class would be, nor on the class of *that* class .... Although Smalltalk's solution to terminating this infinite metaclass regress is elegant [BDN+09, Ch. 13], we felt that with beginning student programmers, it is better to avoid the whole topic.

As we will see in the remainder of this apologia, defining *inheritance* in terms of object means that we must imbue objects with much of the complexity that would otherwise lie in classes. In particular, we were forced towards a "dualist" notion of object, in which every object carries within it the seeds of its own creation, like a cell that contains two copies of its own DNA. This closely parallels the idea that a class must be treated both as a *function* (when it plays the role of a superclass), and as the *fixpoint* of that function (when it plays the role of a generator of objects) [Coo89].

3 Designing Inheritance

For Grace, the design problem we faced was how to add inheritance to the conceptual model we had selected: objects preceding classes, objects created by object constructors, and classes defined in terms of objects. For ease of transition to other languages, we wanted to be able to say that one class inherited from another in a more or less conventional way, but to be able to explain what that meant purely in terms of objects.

3.1 First Steps toward Inheritance

In early versions of the Grace specification [BBN11], we were somewhat vague about the semantics of inheritance:

> Grace class declarations supports inheritance with "single subclassing, multiple subtyping" (like Java), by way of an **inherit** C clause in a class declaration or object literal. ... The right hand side of an **inherit** clause is restricted to be a class name ...

We were clearer about classes being equivalent to object constructors:

> Grace's class declarations can be understood in terms of a flattening translation to object constructor expressions that build the factory object. Understanding this translation lets expert programmers build more flexible factories.

[1] Sir Thomas Beecham, conductor and impresario, once addressed an orchestra: "Forget about bars. Look at the phrases, please. Remember that bars are only the boxes in which the music is packed." [AN79, p.18]

```
class superclass {
    print "start super init"
    method m { ... }
    method n { ... }
    var v := ...
    print "done super init"
}
```

```
class subclass {
    inherit superclass
    print "start sub init"
    method n { ... super.n ... }
    def c = ...
    print "done sub init"
}
```

```
class subclass {
    print "start super init"
    method m { ... }
    method n { ... }
    var v := ...
    print "done super init"
    print "start sub init"
    method n { ... super.n ... }
    def c = ...
    print "done sub init"
}
```

Figure 1 – at the top, subclass inherits from superclass. Below, the "flattened" equivalent of the subclass.

The idea was that a subclass inheriting from a superclass could be thought of as equivalent to a single, flattened class, as shown in Figure 1. This is very similar to the way inheritance (originally called "prefixing") was defined in Simula [DMN70]. We used a Smalltalk-style **super** to invoke methods defined in superclasses. Notice that the flattened version has multiple declarations of the method n, a feature that was necessary to give meaning to **super** and the flattening translation, but not really something that we thought desirable in a source language designed for novices.

To support static type-checking, and the checking of the override annotation, the language specification restricted a superclass to being "definitively static". This meant that the compiler had to be able to figure out which methods were inherited. As a consequence, the expression after **inherit** could not be a variable, or a parameter of an enclosing method.

This definition of inheritance is ambiguous because it does not make clear the state of the object under construction at the moment when the superclass is initialised. Which method n is installed? Does the field c exist? If it does exist, is it initialized? We were also troubled that the meaning of superclass in the **inherit** statement had to be different from the meaning of the same expression elsewhere in the program: instead of constructing an object, it somehow retrieved the template that would have been used to construct that object. It became clear that Grace needed a complete and coherent design for classes and class inheritance in terms of object constructors. Such a design needed to be complete enough for programs to be written without class declarations (because classes were merely a shorthand for methods returning objects), and coherent enough that we could explain it to students without blanching. The rest of this apologia describes our attempts to create this design, and defends the resulting language.

```
method graphic(canvas) {
    object {
        method image { required }
        method draw { canvas.render(image) }
        var name
        canvas.register(self)
        draw  // local method request
    }
}
```

```
def amelia = object {
    inherit graphic(canvas)
    def image is public = images.amelia
        // override image method
        // with a field accessor method
    name := "Amelia"
        // assign to inherited field
}
```

Figure 2 – An example object constructor method and an inheriting object

3.2 Design Concerns

Over time, we collected a list of concerns that influence the design of any inheritance mechanism. As implementations of Grace became available, and we were able to start writing larger examples, such as a graphics library designed for a first programming course [BDM16], and a Smalltalk-inspired collection library [GR83], and to explore how the concerns influenced the examples. This process helped convince us that there was no "obviously correct" resolution for many of the concerns. Figure 2 is an artificial example that illustrates some of the concerns, which we discuss below. The keyword required in the image method of graphic means that the method is abstract, in other words, that concrete subobjects are *required* to provide an overriding implementation.

Initialisation. One of the motivations for adopting object constructors was that they gave us a simple way of creating *already-initialized* objects, including objects with immutable fields. In contrast, in Smalltalk (and in Java), objects are created with all fields null, and an initialization method (confusingly called a "constructor" in Java) later gives them values. Not only does this require writing more code, it also means that all objects must be mutable, at least when they are created.

Registration. Is the identity of a superobject during initialisation the same as that of the final object? This is clearly the intention behind the request of canvas.register in graphic's initialisation in Figure 2.

Down-calls. A method in a superobject can make a self-request of a method that is defined in a subobject — sometimes called a "down-call". Can it do so during initialisation? The implementation of the draw method relies on such a down-call to the image method.

Stability. Is the set of methods available in an object the same throughout the object's lifetime? Can the implementations of these methods change? For example, which image method will be invoked by the request of draw at the end of graphic?

Preëxistence. Can an object inherit from any object to which it can refer? Does amelia have to inherit from a *method* that generates a new object, or will a preëxisting object suffice?

Simplicity. Recall that the goal of Grace is to reduce the "accidental" complexity of the programming language and allow students to focus on the "essential" difficulties of programming [BBHN12]. To meet this goal, we wanted an inheritance mechanism that was as simple as possible. The countervailing force towards complexity comes from the need to introduce students (but gently!) to the inheritance mechanisms they will eventually encounter in "industrial strength" languages.

Discussion

Of these design concerns, initialisation is one of the thorniest. This is because objects often have internal invariants that must be maintained if their methods are to behave correctly. When a method is inherited, its correct operation will generally require the invariants to hold in the "new" surroundings; ensuring this is surprisingly tricky.

Recall that in Grace, an object constructor (and thus a class) can contain initialisation code. This code can access **self**. As Gil and Shragai [GS09] point out, this is potentially dangerous; a method requested on **self** during initialization may make assumptions about the object that do not (yet) hold, because the object's construction is incomplete. In Figure 2 we can see that **var** name is declared uninitialised, and that **def** image will be initialised only after the amelia object constructor has run.

Nevertheless, for the purposes of Grace — teaching programming, including the pitfalls that it sometimes holds for the unwary — introducing a separate category of methods that are usable only during object creation seems like a high price to pay to avoid this danger. Moreover, the most likely pitfall — accessing an uninitialised variable — is already an error in Grace. Implementations are required to check for this error; we can extend this check to uninitialised *definitions* (**def**s) as well.

In this remainder of this section we discuss four designs for the semantics of single inheritance: delegation, concatenation, merged identity, and uniform identity. Formal models of these designs can be found in a companion paper [JHNB16].

It is tempting, in hindsight, to rephrase these design concerns as "criteria" that an inheritance mechanism must satisfy. This would not be historically accurate, because it was only *after* exploring various design options and their consequences that we realized, sometimes with great reluctance, that a particular concern would have to be resolved in a certain way. Indeed, if we had known the right set of criteria when we started, the design process would have been shorter by several years! Part of the motivation for this apologia is to describe the *process* through which these design concerns evolved into criteria.

3.3 Delegation

Our first design attempted to model inheritance between classes using delegation between objects. Since Lieberman and Stein's work of the mid-1980s [Lie86, Ste87], delegation has been seen as, more-or-less, an object-based version of inheritance [LSU87]. In delegation, one or more of the fields of an object r can refer to a delegate object; these fields are often called "parent" fields, following Self [CUwC91]. If r receives a request for which it has no method, the request is automatically handled by (one of) r's delegate object(s). Crucially, when the delegated method is invoked, **self** is bound to r, the object that received the original request, and *not* to the delegate that actually implements the method. This means that any **self** requests in the method will be received by r, not by the delegate. Thus, delegation provides late-binding semantics similar to Smalltalk's class inheritance, which allows self-sends to be overridden in

subclasses. It is this binding of self that distinguished delegation from forwarding.

Let us consider the example from Figure 2 as if it used delegation. Let us imagine that the inherit clause stores a reference to a delegate object, created by executing graphic(canvas). The result is two separate objects joined by a delegation reference. The treatment of requests is indeed what one would expect: a draw request received by amelia finds no appropriate method there, so it is delegated to the "parent", an instance of the graphic class. There, the draw method makes a self-request for image; because self is still bound to amelia, the response to this request will come from the definition of image in amelia. Unfortunately, delegation doesn't deal quite so well with all of the concerns of section 3.2.

Initialisation. Because all objects, including amelia, are generated by executing object constructors, their fields can be initialised when they are created; this includes the immutable image field of amelia. Note that this is a result of using a referentially-transparent semantics for the inherit statement, rather than delegation *per se*. The delegate graphic is created in the same way as any other object: by executing an object constructor. A delegation-based language that uses cloning, rather than object constructors, would still have to use variables for fields that are conceptually immutable.

Registration. Because the delegate and the delegator are separate objects, delegation doesn't support registration as in the way that Figure 2 seems to require. When the delegate, a graphic(canvas) instance, makes the canvas.register(self) request, self refers to the graphic object, and that is what will be registered with the canvas: at this point, amelia does not yet exist. Subsequent requests, say, of draw, from the canvas to the registered object, will be directed to the graphic object, not amelia. These requests of draw will fail, because image will be bound to the required method.

This behaviour is a symptom of the *split object problem* [BD96], also known as *object schizophrenia* [Her10]. Because two objects together implement the behaviour of amelia, it is possible for the wrong object [PB12] to be sent a request. Solving this problem requires changing our example program; for example, amelia might pass self, along with canvas, as an argument to graphic.

Down-calls. As we have seen, once both objects are constructed, self-requests in the delegate can be handled by methods in the delegating object. This is not the case during intialisation: at that time, self is bound to the graphic object itself, and so self-requests are directed to the graphic object.

Why do we choose these semantics? The creation of amelia has only just commenced when the request to graphic(canvas) that creates the graphic object is made. We reason that the inherit clause must first create and initialise the graphic; only after that can it bind the graphic as amelia's delegate. So, at the time of its creation, the graphic is nobody's delegate, and hence it is its own self.

Stability Delegation is stable in the sense that method resolution never changes over time, assuming that both the structure of the individual objects *and* the delegation relationships between them are fixed once they are created. Because two distinct objects collaborate to implement amelia, however, the method that is executed in response to a given request *will* depend on which of them first receives that request. To prevent confusion, amelia should guard references to the delegate closely.

Preëxistence Many delegation-based systems allow "inheritance" from existing objects. Taivalsaari claims that this is a defining characteristic of prototype-based systems [Tai99], but since Grace does not aspire to prototypes, we saw no compelling reason to allow it.

Discussion

We *did* see a compelling to reason *disallow* delegation to a preëxisting object. If this is allowed, delegation can exhibit "action at a distance", in which an operation on an object x can implicitly change another object y. For example, assigning to the name field of the graphic object, as amelia does, will change the name of every object that delegates to that graphic object. Other designers have seen action at a distance as an advantage, and have allowed the same delegate to be used by many other objects. In such systems, as Self's designers put it, "parents are shared parts of objects" [CUwC91].

In our example, because amelia creates a fresh delegate, this is not a problem. If one wishes to avoid action at a distance, a design based on delegation should require that the delegate be either fresh, or immutable.

Delegation to preëxisting objects can also breach instance-based protection. In Grace, as in Smalltalk and Self, objects are *autognostic*: an object can access other objects only through their public interfaces [Coo09]. Notice that this differs from languages (such as C++ and Java) where one instance of a class has access to the private components of other instances of that class. Consequently, in Grace, non-public — we call them confidential — methods cannot be requested by other objects, while objects in an inheritance relationship *can* request each other's confidential attributes. Delegation to a preëxisting object can effectively breach the protection provided by confidential: all an object need do is create a new object spy that delegates to the object holding the secret, and give spy a public method that makes a self-request for the secret. Note that this is a problem *only* when delegation (or inheritance) is permitted from a preëxisting object: if the subobject is responsible for *creating* the parent, then we can assume that it is already privy to the information that it contains.

3.4 Concatenation

The second design option we explored was concatenation, which Taivalsaari has proposed as an alternative to inheritance [Tai93, Tai95, Tai09]. If you think of delegation as working by *reference* to a superobject, you can think of concatenation as working by *value*. Inheritance from a superobject begins by making a copy of the superobject; the new definitions provided by the subobject are then "concatenated" onto the end of the copy. Concatenation results in a single object with a single identity.

A consequence of concatenation is that it is possible for an object to have two methods with the name m, one inherited (copied) from the parent object, and one provided by the subobject's constructor. This accommodates the conventional semantics of a **super**-request. An external request for m executes the "lowest" method, but a **super**-request can be used to invoke the next "higher" method, the one inherited from the superobject. Let's see how concatenation treats the concerns of section 3.2.

Initialisation. Because superobjects are created by executing object constructors, immutable fields can be initialised by that constructor. Concatenation requires the superobject to be *copied* during object creation, so the copy mechanism must correctly copy the values of initialised fields.

Registration. Once creation is complete, concatenation produces a single object, but *during* creation, a series of objects are created, copied, and extended. Each of these objects will have its own identity — even if it will be discarded once creation is complete. If self is captured during the initialisation of one of the superobjects, that object will be preserved rather than discarded. Recall our example from Figure 2, in which the graphic class registers each newly-created graphic with the canvas. Because concatenation makes a copy of the superobject, the original graphic object, which should have been inaccessible (and thus garbage collected), is the one that will be registered with the canvas. The *copy* will be extended into the final amelia object — and so amelia will *not* be registered with the canvas. As with delegation, the superobject's initialization code is executed before the subobject is created, and thus cannot know about the subobject.

Down-calls. As with delegation, once an object has been constructed, down-calls will work as expected. However, during the creation and initialisation of a superobject, the subobject will not yet have been created, so down-calls are impossible. In Figure 2, when the draw method is requested during graphic's initialisation, image will bind to the required method image, because the concatenation of amelia's methods has not yet occurred.

Stability. The stability of concatenation is very similar to the stability of delegation. The subobject is built up over time, with a different identity at each stage; method resolution for each of these identities is stable. Methods requested during initialization on a superobject will be resolved in the context of that superobject, and not in the context of the final, not-yet-created subobject.

Preëxistence. Concatenation relies on making copies of the superobjects that will to be inherited. This implies either that copy is a meta-operation that can be applied to all objects, or that an object is inheritable only if it defines a copy method.

Discussion

As we have seen, the concatenation design relies crucially on a mechanism for copying the superobject. Where does this copy mechanism originate? We might assume that it is "built-in" as part of the inheritance mechanism: that inheritance implicitly makes a (say, shallow) copy of the superobject. This assumption exposes two problems. First, it is not necessarily appropriate to allow copying every object — something that concatenation makes trivial by inheriting from the object and adding no attributes. Imagine an object representing some external resource, such as a canvas in a window system. This object might be designed under the assumption that it has exclusive access to the external canvas: making a copy violates that assumption. Second, even if copying is unobjectionable, a built-in copy primitive may do the wrong thing. Consider a list object that uses a vector object to store its elements. Making a shallow copy of the list object will result in the copy and the original *sharing* the vector as well as their contents. Making the copy deeper doesn't help; whatever the built-in copy primitive does, we can find an example where it does the wrong thing.

The obvious "fix" is to allow programmers to customize the behaviour of copy, as does Smalltalk with its postCopy hook method. Unfortunately, this will not work for immutable fields: the whole point of an immutable field is that it cannot be changed (except, perhaps, by reflective code [CM13]). Hence, a postCopy method cannot assign to immutable fields to repair the object's invariants. Recall that immutability after

creation was one of the reasons we based Grace on object constructors. Solving this problem would require building into Grace a declarative mechanism for specifying the semantics of copy [Li15].

The alternative assumption is that the programmer must supply a copy method for every object. Failure to provide such a method would mean that the object could not be inherited. This would impose a severe burden on the programmer — one not present in mainstream languages.

3.5 Merged Identity

In response to these problems — particularly the difficulty in defining copy, and the implications of copying for registration — we devised an alternative semantics that eliminated the implicit copy. Rather than copying the superobject, *merged identity* inheritance starts with the *actual* superobject and *mutates* it by adding in the new declarations from the body of the object constructor.

Left unchecked, merged identity would allow programmers to change the shape and value of any preëxisting object. To avoid this, we imposed the constraint that the superobject must be *fresh*, that is, it must be an object that did not exist before the object constructor was invoked. By this *freshness constraint* we hoped to hide the mutation performed by the object constructor.

The effect of the freshness constraint is that the expression in the **inherit** clause must generate a new object. For example, it can be a request on a class, or a request on a method that directly returns the result of an object constructor. A request of an existing object's copy method should also suffice; although copying is no longer part of the semantics of inheritance, copying will still be essential to many practical *uses* of inheritance. Once again, we consider the concerns from section 3.2 in the context of the example in Figure 2.

Initialisation. Primitive object creation is still handled by object constructors, so fields — including immutable fields — are initialised by those constructors. If merged identity is used to inherit from a copy of an existing object, that copy will be made by an explicit method request (e.g., **inherit** graphicProto.copy), and so that copy method can be written to maintain the invariants of both the original object and the copy — provided that the result of the copy is fresh.

Registration. Merged identity was designed to solve the object registration problem. Because a single identity is preserved throughout the construction process as the superobject is mutated into the new object, any registration performed on the superobject will apply to the final object.

Down-calls. Although an object's *identity* does not change during construction, the structure of the object *does* change as definitions from subobjects are merged into the object under construction. This means that code in Figure 2 will again fail, because amelia's overriding version of the image method will not yet be present when draw is requested during the construction of the superobject.

Stability. Merged identity does *not* provide stability during initialisation, because each subobject's definitions are added in turn to the object being constructed. Once objects are complete, the freshness constraint ensures that they cannot change again.

Preëxistence. The freshness constraint prevents inheritance from preëeisting objects.

Discussion

Merged identity allows registration in the super-object to work, but simultaneously exposes object mutation, which we think undesirable. Indeed, hiding mutation was the motivation for the freshness constraint. In our example, the new object is registered with canvas *before* the overriding definition of image is installed. If the canvas tries to draw the object immediately, draw will fail by executing the required image method; if it waits awhile, it will succeed. This is hardly satisfactory. In general, by allowing a reference to the object under construction to escape, registration exposes clients to the fact that objects are unstable.

There is a strong argument that "down-calls" during construction *should* have the behaviour described above, and should *not* bind to overriding methods. This argument motivated the design of C++, which adopts essentially this semantics. In general, a method body may depend on fields defined in the object in which it is written, and therefore these fields must be initialised before the method executes. During object construction, field initialization is inevitably intertwined with the execution of initializing expressions. If we assume that superobjects are initialized before subobjects, binding a downcall during initialisation to the method in a subobject puts that method at risk of accessing an uninitialised field. The reverse ordering, in which subobjects are initialized before superobjects, makes even less sense: this would mean that initialization code in the subobject could not use features of the superobject, nor override the default values of fields set by the superobject.

3.6 Uniform Identity

The merged identity design still did not seem satisfactory; the instability of objects during initialisation and our failure to hide it were particularly troubling. As well as making method resolution hard to explain — ease of explanation is important in a teaching language — the visibility of object mutation also caused practical problems in Grace's pedagogical graphics library (*objectdraw*, converted from Java [BDM06]). This suffered from the failure described in the previous paragraph, in which the required draw method of the superobject is requested before it has been overridden by the subobject.

These considerations led us to a "two-phase" semantics for object construction and inheritance that we call *uniform identity*. The first phase begins by creating a new object identity. Then, the skeleton structure of the object is built with that identity, by collecting all the declarations from the superobject constructors on the inheritance chain, and the declarations from the subobject's constructor itself. As with merged identity, uniform identity requires all superobjects to be fresh. At this point, the skeleton structure contains all the methods of the object under construction, and uninitialised slots for the immutable and mutable fields. So **self** now exists, in its final shape, although its fields have not yet been initialised.

In the second phase, the code inside the object constructors (but outside of the method bodies) is executed: this includes the expressions that initialise fields, and code at the top-level of the object constructors. This initialisation code is executed "top-down", beginning with the top-most superobject and finishing with object constructor (or class) for the subobject under construction.

If you prefer, you can think of construction as beginning at the bottom-most object constructor, the first statement of which is the inherit clause. This begins executing the inherited object constructor, starting with *its* inherit clause, and so on up the inheritance chain, until we reach a constructor that does not inherit. Once each object constructor's inherit statement has finished, the rest of that constructor's body is executed, initialising fields and running any in-line code. Whichever way you prefer to think about it, the result is the same.

Once again we consider the concerns from section 3.2.

Initialisation. As with all our other designs, mutable and immutable fields can be initialised, and uninitialised variables can be observed by the program, e.g., by requests in partially initialised objects.

Registration. With only one (uniform) object identity, registration works at any time during initialistion, but with the same caveat as with merged identity: the registrar can observe unitialised "immutable" fields that later become initialised.

Down-calls. Because the object under construction's methods are installed before any methods can be requested, down-calls (and up-calls) to methods work in the same way both during initialisation and afterwards; however, down-calls to fields will find them uninitialized.

Stability. Because all the methods and fields are installed before there is an opportunity for the object to be examined, the object's structure appears to be stable.

Preëxistence. Like merged identity, inheritance must be from a fresh object: a preëxisting object must be copied.

Discussion.

We would like to say that both phases of object creation happen *before* the object comes into existence. Indeed, from the point of view of a client, the fully-initialized object does spring into existence atomically at the end of the second phase. If cross-examined in a court of law, however, we would have to be very careful. The problem is that self can be accessed during initialization. As a consequence, uninitialised variables can be still be accessed, both within a single object constructor and between constructors that inherit from each other. In the example in Figure 2, although graphic's required image method will be successfully overridden during creation, the code will still fail because the image variable declared in amelia will be undefined at the time it is requested from draw.

Uniform identity closely mimics the inheritance semantics of Java. Note, though, that Grace's generative object constructors are quite different from what Java calls a "constructor", which is really an initialization method requested *after* the object has been constructed.

It is important to note that uniform identity breaks referential transparency. In every other inheritance design we've considered so far, the argument to the inherit clause is executed in exactly the same way as any other Grace expression: it evaluates to an object (which we will call the *parent*) that is completely initialised and fully-independent. Then, the object under construction does something to the parent: it delegates to it, or copies it and appends to it, or mutates its structure. With uniform identity this is no longer so: two-phase execution means that the construction of the

parent must be split into two parts, the first of which adds definitions to an existing object, while the second performs initialisation. The expression in the **inherit** clause *cannot* be evaluated to the object that would result in any other context; it must be treated as some kind of object generator that can be dissected into these two parts.

4 Multiple Reuse

After we had gained some experience with the uniform identity design for single inheritance, we began to consider how we might generalise it to allow reuse of multiple existing components, that is, something like multiple inheritance [Mey89]. Early in Grace's design we had guessed that we would need something like this, but had decided to postpone developing these features until we a found strong need for them.

We wanted multiple reuse for two reasons. First, we had designed several versions of a collections library for Grace: as others had found before us, such a library can benefit from multiple reuse because there are multiple independent axes of specialization. For example, collections may be mutable or immutable; they may be fixed size or variable size; they may provide iteration based on insertion order, or on an ordering relation, or offer no fixed iteration order [BSD03, Moo96]. Second, we had begun to experiment with Grace dialects, which are specialised libraries that can extend or restrict the core language [HJN+14]. Here too we found that dialects provided multiple independent features. We had a dialect for testing, and a dialect for drawing, and a dialect that required type declarations; we needed a simple way to construct dialects that offered combinations of these features.

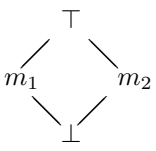
In this section, we present a number of designs we considered for multiple reuse in Grace: traits as objects, generalising uniform identity inheritance to multiple inheritance, a more flexible variant to better-support initialisation, and replacing Smalltalk-style super-calls with trait-style aliasing.

4.1 Traits as Objects

Our first design for multiple reuse was based on Smalltalk-style traits. The name "trait" has been applied to a variety of modularity constructs for objects, starting with the Xerox Star workstation [CBLL82, CA84] and continuing with trait objects in Self [US91]. More recently, it was applied by Ducasse *et al.* [DNS+06] to a mechanism for reusing groups of methods in Smalltalk.

Three features distinguished the Smalltalk trait, as originally envisaged, from contemporary multiple inheritance mechanisms. The first was the absence of any implicit priority between the methods obtained from a set of traits. Instead, conflicts had to be resolved *explicitly*. The second was a rich algebra of trait combination operations, which enable the programmer to combine simple traits into more complex traits, and to resolve conflicts between multiple traits. The third was the absence of instance variables from traits; this meant that traits were stateless, and thus the "diamond problem" of multiple inheritance did not occur.

The operations on traits, although derived independently, turned out to be similar to operations proposed by Bracha in his thesis [Bra92]. The trait operations supplemented the asymmetric *inheritance* operation common to Smalltalk and most of its successors by several additional operations: an asymmetric *use* operation, a symmetric *sum* operation, a *method exclusion* operation, and a *method alias* operation.



⊔	⊥	m_1	m_2	⊤
⊥	⊥	m_1	m_2	⊤
m_1	m_1	m_1	⊤	⊤
m_2	m_2	⊤	m_2	⊤
⊤	⊤	⊤	⊤	⊤

where $m_1 \neq m_2$

Figure 3 – the trait join operator. Notice that while $m \sqcup m = m$, when $m_1 \neq m_2$, $m_1 \sqcup m_2 = \top$. This ensures that conflicts are "sticky", and that sum is associative. (Figure reproduced from Schärli's Thesis [Sch05, p.17], with permission.)

Trait sum is similar to Jigsaw's merge, in that both operations are symmetric and commutative, although the mechanisms are different. Where Jigsaw imposes side-conditions to ensure that merged modules are disjoint, trait sum has no such restrictions. Instead, it represents omitted and conflicting methods by extending the set of methods $\mathcal{M}$ to a flat lattice $\mathcal{M}^*$, with new elements $\bot$ representing the absence of a method, and $\top$ representing a conflict. All other elements of the lattice are incomparable. Thus, the *join* operator for $\mathcal{M}^*$ is as shown in Figure 3.

These operations made it possible to compose traits into other traits and classes much more flexibly than was possible using inheritance alone. This enabled finer-grained code sharing and a reduction in both code duplication and in the need to cancel inappropriately inherited methods [BSD03].

Another important feature of Smalltalk traits is the *flattening property*, which means that a class (or trait) composed from a complex graph of traits is precisely equivalent to another class (or trait) constructed by copying trait methods into the using class, following a rule for each of the trait composition operations. Flattening made it possible build a tool, the traits browser, that provided the programmer with multiple views on the same program [BS04]. Viewed through the traits browser, the trait structure could be flattened away entirely, viewed in fine detail, or partially flattened to any degree required. The flattening property also means that while traits are valuable tools for program construction, modification, and understanding, they have no semantic significance, and need not even exist at runtime. This is in contrast to inheritance, which, because of **super** messages, cannot be flattened away.

In spite of all of these advantages, adding traits to Smalltalk increases the language's complexity. Traits were also a relatively new feature; we wondered if they would stand the test of time. Traits have been added to several other object-oriented languages [Ode11, Gro14, The15, PHP16] but had not been tried in a language designed for teaching novices, where simplicity and similarity with the mainstream are important. For these reasons, the Grace design team initially decided not to include traits in Grace, and instead to rely on conventional inheritance.

Once we had decided that some support for multiple reuse was necessary, traits seemed like the simplest solution, and we looked for ways to incorporate traits into Grace with minimal disturbance to the rest of the language. Our first design proposed modelling traits as Grace *objects*. This contrasts with the inheritance semantics described in the previous sections, where inheritance must be from a *fresh* object.

We proposed introducing **trait** as a new reserved word. If traits are just objects, this isn't strictly necessary, but the new syntax would allow us to check that trait objects contain no fields (neither **def**s not **var**s), and no inline initialisation code. A

trait keyword would also help programmers to be explicit about their intention when creating a trait object, and familiarize students with the terminology. These traits are real Grace objects, but objects that meet certain restrictions.

The second proposed change was to supplement the reserved word **inherit** by the new reserved word **use**, valid in the same place, but restricted to traits, so an object constructor might **use** enumerableTrait. Semantically, an object b **use**ing a trait t is equivalent to b *delegating* all requests of methods defined in the trait to the object t, unless b provides an overriding local definition. *Delegation* means that all self-requests made in the trait method are requests to b, not to t, as we discussed in subsection 3.3.

The various trait operations can also be defined in terms of delegation. The trait *sum* operation is defined so that $t + u$ is a new trait with all of the methods of trait t and all of the methods of trait u, where t's methods are delegated to t, u's methods are delegated to u, and methods common to both t and u are error methods (representing $\top$ in the method lattice). Similarly, the meaning of the *difference* operation is that $t - m$ is a new trait that delegates all of t's methods, other than those in the set of method names m, to t.

The **use** mechanism works fine for reusing behaviour from stateless immutable objects, such as a trait that provides unit tests with a family of assert methods, or provides collections with a family of internal iteration methods based on an external iterator. In Grace, the singleton object true has no state, just methods like

```
method or(another:Block) { self }
method and(another:Block) { another.apply }
method ifTrue(trueBlock:Block) ifFalse(falseBlock:Block) { trueBlock.apply }
```

This means we can reuse true, as shown in this example motivated by Homer *et al.*'s design for object-oriented pattern-matching [HNB+12].

```
class successfulMatch(result′, bindings′) {
    use true
    def result is public = result′
    def bindings is public = bindings′
    method asString {
        "SuccessfulMatch(result = {result}, bindings = {bindings})"
    }
}
```

Although each object can **use** just one trait, trait usage gives us the effect of multiple inheritance because the used trait can be the sum of several other traits, which can in turn be composed from still smaller traits.

Disallowing state in traits had several advantages, as Schärli and colleagues observed. In particular, although we had chosen to define the semantics using delegation, in the absence of state the trait methods could simply be copied into the delegating object. Because there were no fields, and no straightline code, no initialization was required, and all of the associated problems were avoided.

Importantly, **use** did not require the extra level of object wrapper implied by **inherit**, unless that level actually served some purpose. In contrast, both merged and uniform identity restrict the programmer to inheriting from a fresh object, forcing the programmer to make everything instantiable (typically via a class) or copyable, just in case someone might later wish to inherit from it.

Unfortunately, this design had two problems. First, the restriction that traits may not contain fields does not mean that they are stateless. Unlike Smalltalk, Grace has lexical scope, so a method in a trait can capture a variable in a surrounding scope.

```
class top(x′) {
    print "top {x′}"
    var x := x′
}

class mid {
    inherit top 13
    print "mid"
}
```

```
class side(n) {
    inherit top(n)
    print "side"
}

class bot {
    inherit mid as m
    inherit side 42 as s
    print "bot"
    method ambiguous {print(x)}
    method resolved {print "{s.x} {m.x}"}
}
```

Figure 4 – Classes with multiple inheritance

This means that the "diamond problem" reappears: if the "same" trait is reused twice through two different paths, the trait combination mechanism would need to check if the two trait objects are actually the same object, or are two distinct objects that have the same shape. In the first case, because $m \sqcup m = m$, the composition is sound, and results in the composed object acquiring all of the attributes of the trait object just once. In the second case, the composition results in a conflict for each method. This is because methods with the same name are contributed by two distinct trait objects; when $m_1 \neq m_2$, $m_1 \sqcup m_2 = \top$. Unfortunately, distinguishing these two cases requires reasoning about object identity, which is ultimately impossible to do statically. Because we wanted to know the shape of a composed object statically, for example, to perform type and overriding checks, we regarded this as a serious problem.

Second, because of the restriction that excludes fields from traits, traits would supplement inheritance rather than completely replace it. This meant that Grace would have two *different* reuse mechanisms: **inherit** clauses using uniform identity from classes, and **use** clauses offering delegation to already-constructed objects. We asked ourselves if it might not be possible to find a generalisation of single inheritance that gave us trait-like properties, but within the framework of a single mechanism.

4.2 Generalised Uniform Identity

In an attempt to answer this question, our next design for multiple reuse was to extend the uniform identity design from subsection 3.6 to something more like classical multiple inheritance. Unlike the traits proposal in subsection 4.1, where any stateless object could be reused, we return to uniform identity's constraint that only fresh objects are reusable.

A class or object constructor would be able to **inherit** from more than one superclass, with the semantics being a generalisation of the two-phase uniform identity semantics. In the first phase, a new object identity would be constructed, and then the attributes from every superclass would be incorporated into the new object. In the second phase, the initialisation code would be run. Because there are multiple superclasses, the order of running the initialisation code would be more complicated: the superclasses would be initialised in the order of appearance of the **inherit** clauses that name them.

Consider the example in Figure 4, when we instantiate the bot class. This will

print "top 13, mid, top 42, side, bot": initialisation would proceed as if the inherit clause invoked the superclass directly. The fact that "top" is printed twice shows that the top class is inherited twice.

This example illustrates why the *diamond problem* [Mal08, Mey97] does not occur with generalised uniform identity. According to Malayeri and Aldrich, "the diamond problem arises when a class C inherits an ancestor A through more than one path. This is particularly problematic when A has fields — should C inherit multiple copies of the fields or just one?" Because only *fresh* objects can be reused, it is impossible for two inherit clauses to refer to the same object. Specifically, inherit top 13 in mid and inherit top(n) in side(n) inherit from two separate objects, and thus the attributes of these two separate top objects would be added into the bot object. In particular, there would be two copies of the variable x, with different values. A self-request for x in class bot (as in the method ambiguous) could refer to either definition. Rather than resolving this ambiguity with an arbitrary rule, this proposal extended the inherit clause to include a local *nickname* that could be used to direct a request to a particular definition. In this example, the nicknames are m and s, and the resolved method will print "42 13" by assessing these two variable.

This proposal has more complex initialisation semantics than uniform identity for single inheritance, but retains its properties. With a consistent identity, objects can be registered; downcalls can be made during initialisation to methods but not to fields; method resolution follows the same rules both during and after construction; and mutation of objects during construction is visible.

4.3 Positional Inheritance

To avoid the initialisation problems described in the previous section, the Grace team explored another design, *positional inheritance*, in which inherit clauses can appear anywhere within a class or object constructor, rather that just at the top. This design simplifies initialisation by avoiding the two-phase protocol. Instead, the fields and method of the superobject are inserted into the subobject, *and* initialised, when the inherit clause is executed.

The advantage of this design is more flexible initialisation. Consider the example on the left of Figure 5, in which upcaller1 inherits from downcaller. During its initialization, downcaller makes a request on local while evaluating the argument to print; this will fail, because local has not yet been installed in the subobject being constructed by upcaller1.

The class upcaller2 on the right-hand side of Figure 5 shows how positional inheritance can resolve this problem. By moving the inherit clause after the declaration and initialisation of local, downcaller will be instantiated after local is initialised. The same class upcaller2 also illustrates another property of this design: methods are added to the structure of the new object only when the inherit clause is executed. This means that shape of an object changes during construction. In particular, the request of ping in upcaller2 must fail, because there is no method ping in the object at that time. The methods of downcaller, including ping, are added into the object being constructed by upcaller2 only when the inherit clause is executed.

It is possible to use the flexibility of positional inheritance to resolve this problem. We could we reorder the code in the subclass to look like upcaller3, which declares the field local before inheriting, and requests ping afterwards. Now everything would work, but whether the result will be what the programmer intended is another question.

```
class downcaller {
    print "local = {local}"
    method ping {print "ping"}
}

class upcaller1 {
    inherit downcaller as super
    ping
    def local = 7
}

class upcaller2 {
    ping
    def local = 7
    inherit downcaller as super
}

class upcaller3 {
    def local = 7
    inherit downcaller as super
    ping
}
```

Figure 5 – Classes with multiple inheritance

The positional inheritance design maintains most of the properties of generalised uniform identity (subsection 4.2), with the obvious exception of the stability of objects during construction and initialisation. Using nicknames for superclasses to request conflicting methods is also more complex than the use of **super** in single inheritance, particularly as a nickname does not refer to an individual object — as does every similar name in Grace — but is instead some new kind of "name-qualifier".

The Grace design team considered that in practice the flexibility of this design would make it too complex for a programming language for novices. We realized that "industrial strength" languages might make a different choice. C++'s inheritance model also has objects apparently changing class during initialisation, and Ruby's mixins imperatively modify object structures; these languages have been quite successful.

4.4 Method Aliasing

The next design we present is an attempt to avoid some of the complexity of generalised uniform identity. Rather than using local names to invoke superclass methods, we use *method aliasing* (as in Smalltalk traits) to access overridden or conflicting methods. In this design, an **inherit** clause may have a subsidiary **alias** clause that provides an additional name for a method, and an **exclude** clause that excludes a method in the trait from the object under construction.

```
class cat {
    method speak {print "meow"}
    def legs = 4
}

class fish {
    method speak {print "bubble"}
    def legs = 0
}
```

```
class catfish {
    inherit cat
      alias catSpeak = speak
      exclude legs
    inherit fish
      alias fishSpeak = speak

    method speak { catSpeak; fishSpeak }
}
```

Figure 6 – Catfish inherits from a cat and a fish

Consider the example in Figure 6, where a catfish class inherits from a cat class and a fish class. The semantics of inheritance is similar to that of generalised uniform identity: in a first phase, the object identity is allocated and its structure created, and in a second phase, the objects are initialised in the order of the **inherit** clauses.

```
trait cat {
    method speak {print "meow"}
    method legs {4}
}

trait fish {
    method speak {print "bubble"}
    method legs {0}
}
```

```
class catfish {
    use cat
      alias catSpeak = speak
      exclude legs
    use fish
      alias fishSpeak = speak

    method speak { catSpeak; fishSpeak }
}
```

Figure 7 – Catfish uses cat and fish traits.

The difference from generalised uniform identity is that **inherit** statements do not define nicknames. Instead, the alias clause establishes an additional name for one of the superobject's methods, here the speak method. The aliases catSpeak and fishSpeak can be used to request those methods, even though the local method speak has overridden them.

The **exclude** clause can be used to avoid method conflicts. By excluding a method that would otherwise cause a conflict, **exclude** can ensure that a method from another superobject is inherited. In Figure 6, excluding the method legs from the cat superobject means that catfish will inherit legs from fish.

This design has several advantages. We do not need nicknames, or a special kind of method request to invoke inherited methods. Moreover, every method in an object has a unique name; this stands in contrast to other formulations of multiple inheritance, in which there can be multiple methods with the same name, inherited from different places. Of course, this design also has some disadvantages, chiefly that it retains almost all the complexity of generalised uniform identity. The **exclude** and **alias** clauses do *add* some complexity, but bring with them the benefit of eliminating **super**, which is rarely understood by novices.

4.5 Instantiable Traits

The design for multiple reuse incorporated into the current version of Grace is based on traits, but restricts trait reuse to fresh objects (as in uniform identity), rather then any stateless object (as in our first trait design, discussed in subsection 4.1). A **trait** declaration must now be instantiable, returning a fresh object — like a class declaration, but excluding both fields and inline initialisation. Figure 7 shows how the catfish example from Figure 6 can be written with these traits. We convert legs from a **def** to a **method**, change the **class** keyword to **trait**, and replace the **inherit** keyword by **use**. In this design, as in Smalltalk traits, a subclass may **inherit** from just a single superclass, but can then **use** multiple traits. Methods declared in **use**d traits override method **inherit**ed from superobjects, and methods declared directly in the body of a class or object constructor override both those from superobjects and those from traits.

There are several reasons we selected this design for Grace. First, by eliminating explicit state from traits, we eliminate the code that would otherwise be necessary to initialise that state; with no initialisation code, trait composition is free of effects. Although *inheritance* still needs the two-phase creation and initialization semantics, use of a trait has a much simpler, single phase semantics. Second, as a consequence,

when multiple traits are combined in an object, there are no "order of initialisation" issues between the traits. Third, because of this simplicity, a moderately efficient implementation can be easier and simpler to write. More importantly, the semantics are easier to explain to students.

We emphasize that the trait syntax, like the class syntax, is just an abbreviation for a method that tail-returns an object that follows certain restrictions. Thus, **trait** cat in Figure 7 is *precisely* equivalent to

```
class cat {
    method speak {print "meow"}
    method legs {4}
}
```

or

```
method cat {
    object {
        method speak {print "meow"}
        method legs {4}
    }
}
```

In all three cases, cat is usable as a trait. We expect that programmers will choose to use the trait syntax to make clear their intentions, and to allow errors, such as accidentally including a **var** in a trait, to be detected earlier.

Other aspects of Grace's design mean that the apparently-draconian restrictions on traits are not as severe as they may appear. For example, although traits cannot contain fields, a trait definition can be lexically nested inside a method that declares and initializes mutable state, and methods in such a trait can close over that state. In the example below, the object returned by method counterTrait contains neither fields nor initialization code, and therefore qualifies as a trait. (Contrast this with the class graphic in Figure 2.) This is true even though the **trait** keyword does not appear in the example.

```
method counterTrait {
    var counter := 0
    print "side effect"
    object {
        method value { counter }
        method increment { counter := counter + 1}
    }
}
```

It is true that the method counterTrait itself has effects. These effects, though, are not part of the creation of the trait object. They occur whenever the method counterTrait is executed, and *before* it creates and returns the trait. As a consequence, each object that uses counterTrait gets its own (hidden) counter, and methods value and increment to access it.

This example also makes clear the benefit of representing traits as classes rather than as objects: each use of a trait class generates a *fresh* trait object. This eliminates any confusion about whether a trait used in two different objects, or used multiple times within a single object, creates shared state or multiple copies of the state.

If we wanted a single counter shared by all of the objects that use a trait, we might instead write:

```
var counter := 0
```

```
method sharedCounterTrait {
    object {
        method value { counter }
        method increment { counter := counter + 1}
    }
}
```

or, equivalently (using the **trait** syntax):

```
var counter := 0
trait sharedCounterTrait {
    method value { counter }
    method increment { counter := counter + 1}
}
```

Here, **var** counter has been moved out of the method and into the enclosing module. Now every sharedCounterTrait captures a reference to the same, module-level, counter variable. The semantics of sharing for traits are exactly the semantics of sharing in the rest of the language. It is quite reasonable to incorporate multiple instances of the counterTrait into a single object, provided that aliases for the two value and increment methods are declared to access the two counters.

It is interesting to note that some of the original trait designers have also concluded that stateful traits can be useful [BDNW07] — with the vital caveat that the state should be *hidden*, so as to avoid naming conflicts. This is exactly what the counterTrait achieves.

This, then, is where Grace now rests. It has "inherited" uniform identity **inherit**ance from Java, and the **use** of traits (including those with hidden state) from Smalltalk. Two mechanisms are undeniably more complex than one, but the two mechanisms share several common features: both are restricted to ensure that the object being reused is *fresh*, both themselves generate new fresh objects, and the alias and exclude clauses apply to both in exactly the same way. As this apologia has shown, we fought bravely against great odds to design a single mechanism that would support both conventional single inheritance and a satisfactory form of multiple inheritance, and failed. Somewhat to our surprise, the desire to support immutable objects made the problem significantly harder than it would have been without them.

An instructor who wishes to use Grace to teach Java-style inheritance can ignore traits, needing only to explain **alias** rather than **super** — which we believe to be an easier task. An instructor who is willing to make a fresh start with a somewhat simpler reuse construct can ignore inheritance and tell the students just about traits — and thus circumvent all of the initialisation issues that plague inheritance, while offering the additional ability to reuse multiple components. The uniformity between the **inherit** and **use** mechanisms will, we hope, make it easy to learn the second after the first has been mastered.

5 Related Work

Class-based languages began with Simula [BDMN79], the origin of much of the conceptual framework of object-orientation. Taivalsaari argues that the class-based understanding of programming is also "classical" in the sense of descending from the classical philosophy of Plato and Aristotle [Tai96, Tai99].

Smalltalk greatly expanded the role of classes beyond Simula [Bor86]. Unlike Simula, Smalltalk classes are themselves objects, and therefore instances of other (meta-)classes,

importing the power (and also the complexity) of Lisp-style computational reflection into object-oriented languages [Smi84]. Lisp returned the favour with a series of object-oriented extensions, culminating in the CLOS Meta-Object Protocol [KdRB91]. We also owe to Smalltalk the notion of 'static' declarations, in the shape of global variables with class visibility, as distinct from per-instance (and hence also per-class) variables.

The complexity of Smalltalk's meta-model inspired Lieberman to propose languages based purely on objects, with delegation as the sharing mechanism [Lie86]. This led to a general interest in "prototype-based" programming languages. Such languages, exemplified by Self [US91], create new objects by cloning existing objects, rather than by appealing to a class. Emerald [BHJL07] marked the start of a different object-based tradition; it also eschewed classes as a fundamental concept, but created new objects by executing object expressions, not by cloning existing objects.

The fates of Emerald and Self are instructive: both ended-up supporting a form of class. By 1991 Emerald had a "syntactic construct called a class that provides the functionality normally expected of classes" [HRB+91]. Self's programming style was based on composing its objects from two parts: an instance prototype that defined all the instance variables, and a "trait object" that provided the methods. What would in Smalltalk have been a class was represented by a trait object and a prototype object delegating to that trait. Traits were also linked to each other by a delegation relationship that corresponded to class inheritance [UCCH91]. By 1995, Self objects had been effectively given a "copy down parent" attribute, which caused slots from that parent to be copied into the object whenever it was edited. This explains how, for example, if an extra slot were to be added to a superobject's prototype, the same slot would also be added to every subobjects' prototype. Eventually, a "Subclass me" button was added to the Self IDE, which copied both instance object and trait objects, recreating the class-like structure with delegation and copy-down parent links configured correctly.

Dony et al. modelled a range of different designs for each of class-, object-, and prototype-based languages, by writing a product-line of definitional interpreters in Smalltalk [DMC99, BD96]. Grace is approximately language L17 in their taxonomy. Grace makes no distinction between accessing variables and requesting methods; it avoids dynamic modification of object structure; it uses creation *ex nihilo* to define an object's structure; it creates already initialised objects; and it provides implicit "delegation", whereby method requests can be dispatched to definitions in superclasses without any explicit syntax. We say "approximately" because our design does not quite fit their taxonomy: we restrict inheritance (or "delegation") to fresh objects, and the entire "split" object is treated as a first-class entity. Of course, delegation to fresh objects can also be regarded as inheritance, especially as the independent parts of the objects have no separate identity. Interestingly, L17 is one of the two categories Dony et al. recommend for future language designs. Taivalsaari et al. [NTM97] surveyed further contemporary research along these lines.

Since the late 1980s, Eiffel has incorporated class-based multiple inheritance with deep renaming (rather than Grace's shallow aliasing), exclusion, and repeated inheritance [Mey92]. Eiffel also intertwines type and class, so that an object implementing several different types can have multiple implementations of methods with the same name, disambiguated by the type in which those definitions originate. Eiffel's development environment can generate the "flat form" of a class: unfortunately because of Eiffel's type-aware semantics, the flattening cannot be used to represent all the

possible behaviours of the class. Subsequently, other languages, notably C++, have incorporated features from Eiffel.

Another stream of work is based on mixins, rather than classes [BC90]. Unlike CLOS, or Eiffel multiple superclasses, mixins are applied one at a time, in a linear order specified by the programmer. Bracha's Jigsaw formalised mixin-composition in a class-based style, along with a rich algebra of composition operations, including merge, restrict, select, project, override, and rename [BL92, Bra92]. Jigsaw's *rename* is deep, like Eiffel's, and thus different from trait *alias*, which is shallow. Jigsaw's modules also contain type *declarations* as well as component definitions, and its composition rules are designed to preserve type soundness. Flatt et al. [FKF98, FKF99] develop a semantics for classes and class-like mixins (without composition operators) in a core language, and have incorporated mixins and traits into Scheme (now Racket), based on classes and macros. Lagorio et al. [LSZ12] modelled Jigsaw in a class-based formalism based on Featherweight Java [IPW01]; Corradi et al. [CSZ11] extended this formalism to handle family polymorphism [Ern01]. More recently, class mixins have been incorporated into Newspeak (alongside family polymorphism) [BvdAB+10] and Dart [Bra16].

Ducasse et al. revived traits to provide multiple reuse for Smalltalk [DNS+06]: in their design, a class inherits from a single superclass, and then incorporates a trait, which may be composed from other traits using sum, alias, and exclude operations. All conflicts between traits in this composition must be resolved explicitly; there are no default rules. A key property of Smalltalk traits is that composition problems can always be fixed by the *user* of a trait; the composition operations are powerful enough that it is never necessary to ask the trait *provider* to package things differently. Smalltalk traits can be understood using a model based on flattening, rather than dynamic dispatch to the trait, although these have been shown to be equivalent [NDS06]. Scala [OSV11] and Java 8 [GJS+15] incorporate variants of traits, though they rely on super-sends rather than aliasing or renaming. Scala traits are much richer (and more complex) than Grace traits: they differ from classes mainly in not allowing parameters in their constructors. The order in which traits are added is important, both because of initialization issues, and for resolving uses of **super**. Grace avoids these issues by restricting traits from having visible state, and by making trait composition symmetric.

A discussion of object-based inheritance systems would be incomplete without referring to OCaml [LDF+12], which is not dissimilar to the language of our base model. Both languages have object constructors, classes as a shorthand for methods that return fresh objects, a form of symmetric multiple inheritance, and structural typing. Grace draws on traditional object-based polymorphism rather than OCaml's row polymorphism [RV97].

There are some significant differences between objects in Grace and OCaml. Even though OCaml classes are described as syntactic sugar, objects (or other classes) can inherit *only from classes*. In contrast, in all of our models, objects can inherit from any fresh object, whether or not it is defined by a class. OCaml also has a more complex, but in some ways less powerful, initialisation model than Grace. In OCaml, field initialisers are evaluated in the enclosing lexical scope, and initialisation code must be sequestered into initialisation blocks, which are run late. Neither of these semantic twists reflects a straightforward reading of program's source code. In contrast, in our models initialisation code is always executed in the order written, and in the context of the object under construction.

Given our focus on initialisation as a key reason for preferring object constructors, we must also compare Grace's approach to initialisation to others', particularly those that avoid the sort of problem we have described. The "Hardhat" approach [GS09, ZCPS12] restricts constructors to avoid exposing partially initialised objects. Delayed Types [FX07] deploys ownership types to similar effect, with more annotation overhead. Masked Types [QM09] takes an alternative approach, tracking ("masking") uninitialised features of objects and preventing their use. Freedom Before Commitment [SM11] proposes another approach, where constructors trigger the construction process of their sub-components, with restrictions on parameters to prevent uninitialised field accesses. Finally, placeholders (or futures) can be used to initialise circular structures safely, either by delaying initialisation until the whole structure is complete (placeholders), or by building the parts of the structure lazily when they are first used (futures) [SMPN13, Bra15]. Given the intended audience for Grace, we felt that any of these features would add too much complexity. In teaching, it is well to let students first appreciate a problem, and then use that problem to motivate a solution.

6 Conclusion

> **inheritance**, noun: a thing that is inherited. *he came into a comfortable inheritance. I don't want a penny of your inheritance.*
> figurative : *the European cultural inheritance.*
> mass noun. the action of inheriting: the inheritance of traits.
>
> Mac OS X dictionary, Version 2.2.1 (178).

As the above definition shows, "inheritance" has two meanings. In this apologia we have attempted to both outline the legacy that we "inherited" from other language designs, and to explain the design process that underlies Grace's inheritance mechanism. We hope this presentation helps the reader to make sense of Grace's inheritance mechanism, and to appreciate the reasons for the remaining complexity. In particular, we believe that understanding "the road not taken" can help explain Grace's approach to inheritance. Perhaps our experiences can guide other language designers to their destination by a more direct route.

A question that we cannot answer is this: had we seen our destination from the start, would we have chosen to go there? Is the current design the result of our being seduced into continually adding features to match expectations derived from languages like Java? Would designers with more foresight have taught themselves simpler approaches to programming that worked better with a more limited inheritance facility?

At the start, we hoped for a simple, complete and coherent description of classes and inheritance in terms of object constructors. Our resulting design is some sense complete and coherent: we have implemented this inheritance model, and we have programmed in it without surprises. A paper at ECOOP'16 presents the semantics of the key designs alternatives in a formal model called *Graceless* [JHNB16]. What we hope to have shown in this apologia is that inheritance foiled our efforts to create an even simpler programming language. Inheritance, especially in combination with immutable objects, is neither simple nor obvious: programming language design necessarily involves tradeoffs between a number of different design goals, of which simplicity can neither be the first nor the most important. Whether the resulting

design is actually graceful, graceless, or something in between we must leave to the judgement of those that inherit it.

References

[AN79] Harold Atkins and Archie Newman. *Beecham Stories: Anecdotes, Sayings and Impressions of Sir Thomas Beecham.* Furtura Publications, 1979.

[BBH+13] Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*, pages 129–134, 2013.

[BBHN12] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, pages 85–98, 2012.

[BBN11] Andrew P. Black, Kim B. Bruce, and James Noble. The Grace programming language draft specification version 0.132. Technical report, gracelang.org, 2011.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. Joint European Conf. on Object-Oriented Programming and ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311. ACM Press, 1990. URL: `http://dx.doi.org/10.1145/97946.97982`.

[BD96] Daniel Bardou and Christophe Dony. Split objects: A disciplined use of delegation within objects. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 122–137, New York, NY, USA, 1996. ACM. URL: `http://doi.acm.org/10.1145/236337.236347`, `doi:10.1145/236337.236347`.

[BDM06] Kim B. Bruce, Andrea Pohoreckyj Danyluk, and Thomas P. Murtagh. *Java: An Eventful Approach.* Pearson Education, 2006.

[BDM16] Kim B. Bruce, Andrea Pohoreckyj Danyluk, and Thomas P. Murtagh. *Programming with Grace.* Pomona College, 2016. Draft of 12 May 2016.

[BDMN79] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studentlitteratur, 1979.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, 2009. URL: `http://pharobyexample.org`.

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. *Stateful Traits*, pages 66–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. URL: `http://dx.doi.org/10.1007/978-3-540-71836-9_4`, `doi:10.1007/978-3-540-71836-9_4`.

[BHJL07] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the Emerald programming language. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 11–1–11–51, New York,

NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1238844.1238855, doi:10.1145/1238844.1238855.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *ICCL'92, Proceedings of the 1992 International Conference on Computer Languages*, pages 282–290, 1992.

[Bor86] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 36–40, 1986.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modules, and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[Bra15] Gilad Bracha. Newspeak programming language draft specification version 0.095. Technical report, Ministry of Truth, 2015.

[Bra16] Gilad Bracha. Mixins in Dart, August 2016. https://www.dartlang.org/articles/mixins/, Retrieved January 2017.

[BS04] Andrew P. Black and Nathanael Schärli. Traits: Tools and methodology. In *ICSE*, pages 676–686, Edinburgh, Scotland, May 2004.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection classes. In *OOPSLA*, pages 47–64, 2003.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *ECOOP*, ECOOP'10, pages 405–428. Springer-Verlag, 2010.

[CA84] Gael A. Curry and Robert M. Ayers. Experience with traits in the Xerox Star workstation. *IEEE Transactions on Software Engineering*, 10(5):519–527, 1984.

[CBLL82] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An approach to multiple-inheritance subclassing. In *SIGOA conference on Office Information Systems*, pages 1–9, 1982.

[CM13] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP*, pages 154–178, 2013.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, Department of Computer Science, May 1989.

[Coo09] William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, pages 557–572, 2009.

[Cro08] Douglas Crockford. *JavaScript: the Good Parts*. O'Reilly, 2008.

[CSZ11] Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig: modular composition of nested classes. In *PPPJ*, pages 101–110, 2011.

[CUwC91] Craig Chambers, David Ungar, and Bay wei Chang. Parents are shared parts of objects: Inheritance and encapsulation. In *Lisp and Symbolic Computation*, pages 207–222, 1991.

[DMC99] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Classifying prototype-based programming languages. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 2. Springer-Verlag, 1999.

[DMN70] Ole-Johan Dahl, Björn Myhrhaug, and Kristen Nygaard. *SIMULA: Common Base Language*. Norwegian Computing Center, October 1970.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, March 2006. URL: http://doi.acm.org/10.1145/1119479.1119483, doi:10.1145/1119479.1119483.

[Ern01] Erik Ernst. Family polymorphism. In *ECOOP '2001 — Object-Oriented Programming*, ECOOP '01, pages 303–326, London, UK, UK, 2001. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646158.680013.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 171–183, New York, NY, USA, 1998. ACM. URL: http://doi.acm.org/10.1145/268946.268961, doi:10.1145/268946.268961.

[FKF99] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, UK, 1999. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645580.658808.

[FX07] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, 2007.

[GJS+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle, 2015.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gro14] Groovy Team. The Groovy programming language. Technical report, Apache Inc, 2014.

[GS09] Joseph(Yossi) Gil and Tali Shragai. Are we ready for a safer construction environment? In *ECOOP*, pages 495–519, 2009.

[Her10] Stephan Herrmann. Demystifying object schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '10, pages 2:1–2:5, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1929999.1930001, doi:10.1145/1929999.1930001.

[HJN+14] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. Graceful dialects. In *ECOOP*, pages 131–156, 2014.

[HN12] Michael Homer and James Noble. Graceful patterns for patterns in Grace. In *Proceedings of the 19th Conference on Pattern Languages of Programs*, PLoP '12, pages 11:1–11:15, USA, 2012. The Hillside Group. URL: http://dl.acm.org/citation.cfm?id=2821679.2831281.

[HNB+12] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. Patterns as objects in Grace. In *Dynamic Language Symposium*, pages 17–28, New York, NY, USA, 2012. ACM.

[HRB+87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language report. Technical Report 87-10-07, University of Washington, Department of Computer Science, October 1987.

[HRB+91] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language report. Computer Science, UBC, October 1991.

[IdFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM. URL: `http://doi.acm.org/10.1145/1238844.1238846`, `doi:10.1145/1238844.1238846`.

[IPW01] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLaS*, 23(3):396–450, 2001.

[JHNB16] Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object inheritance without classes. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6107`, `doi:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.13`.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[LDF+12] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.00 documentation and user's manual, 2012.

[Li15] Paley Guangping Li. *Object Cloning for Ownership Systems*. PhD thesis, Victoria University of Wellington, 2015.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 214–223, New York, NY, USA, 1986. ACM. URL: `http://doi.acm.org/10.1145/28697.28718`, `doi:10.1145/28697.28718`.

[LSU87] Henry Lieberman, Lynn Stein, and David Ungar. Treaty of Orlando. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 43–44, New York, NY, USA, 1987. ACM. URL: `http://doi.acm.org/10.1145/62138.62144`, `doi:10.1145/62138.62144`.

[LSZ12] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw: Replacing inheritance by composition in Java-like languages. *Inf. Comput.*, 214:86–111, 2012.

[Mal08] Donna Malayeri. Cz: Multiple inheritance without diamonds. In *Companion to the 23rd ACM SIGPLAN Conference on Object-*

oriented Programming Systems Languages and Applications, OOPSLA Companion '08, pages 923–924, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1449814.1449910`, `doi:10.1145/1449814.1449910`.

[Mey89] Bertrand Meyer. Re: Eiffel vs. C++. email to `comp.lang.eiffel`, June 1989.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, second edition, 1997.

[Moo96] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA*, pages 235–250, 1996.

[NDS06] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 5:66–90, 2006.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[NHBB13] James Noble, Michael Homer, Kim B. Bruce, and Andrew P. Black. Designing Grace: Can an introductory programming language support the teaching of software engineering? In *26th Conference on Software Engineering Education and Training (CSEET)*, 2013.

[NTM97] James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Conecepts, Languages, Applications*. Springer-Verlag, 1997.

[Ode11] Martin Odersky. *The Scala Language Specification: Version 2.9*. Programming Methods Laboratory, EPFL, Switzerland, 2011.

[OSV11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming In Scala*. artima, 2011.

[PB12] Nick Park and Bob Baker. The wrong trousers. 2entertain DVD, February 2012.

[PHP16] PHP Team. PHP programmer's manual. Technical report, PHP, 2016.

[QM09] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65, 2009.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *POPL*, pages 40–53, 1997.

[Sch05] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, February 2005.

[Sha13] Pat Shaughnessy. *Ruby Under A Microscope*. No Starch Press, 2013.

[SM11] A. J. Summers and P. Müller. Freedom before commitment — a lightweight type system for object initialisation. In *OOPSLA*, pages 1013–1032, 2011.

[Smi84] Brian Cantwell Smith. Reflection and semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM. URL: `http://doi.acm.org/10.1145/800017.800513`, `doi:10.1145/800017.800513`.

[SMPN13] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix—safe modular circular initialisation with placeholders and placeholder types. In *ECOOP*, pages 205–229, 2013.

[SRV+15] Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. Does JavaScript software embrace classes? In *SANER*, pages 73–82, 2015.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 138–146, New York, NY, USA, 1987. ACM. URL: `http://doi.acm.org/10.1145/38765.38820`, `doi:10.1145/38765.38820`.

[Tai93] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming*. PhD thesis, University of Jyväskylä, 1993.

[Tai95] Antero Taivalsaari. Delegation versus concatenation or cloning is inheritance too. *SIGPLAN OOPS Mess.*, 6(3):20–49, July 1995. URL: `http://doi.acm.org/10.1145/219260.219264`, `doi:10.1145/219260.219264`.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

[Tai99] Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 1. Springer-Verlag, 1999.

[Tai09] Antero Taivalsaari. Simplifying JavaScript with concatenation-based prototype inheritance. Technical Report Raportti 6, Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos, 2009.

[The15] The Rust Team. The Rust programming language. Technical report, Mozilla Inc, 2015.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, July 1991. URL: `http://dx.doi.org/10.1007/BF01806107`, `doi:10.1007/BF01806107`.

[Ung02] David Ungar. How to program in Self 4.1. Technical report, Sun Microsystems, Inc, 2002.

[US91] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp Symb. Comput.*, 4(3):187–205, July 1991. URL: `http://dx.doi.org/10.1007/BF01806105`, `doi:10.1007/BF01806105`.

[WB15] Allen Wirfs-Brock, editor. *ECMAScript 2015 Language Specification*. Ecma International, 6th edition, 2015.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *OOPSLA*, pages 168–182, 1987.

[ZCPS12] Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. Object initialization in X10. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages

207–231, Berlin, Heidelberg, 2012. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_10, doi:10.1007/978-3-642-31057-7_10.

Acknowledgments This work was supported in part by the Royal Society of New Zealand Marsden Fund, and a James Cook Fellowship.