What Shall We Tell the Children (About Inheritance)?

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract
Since the groundbreaking work of Kamin, Reddy, and particularly Cook in the late 1980s, there has been broad agreement that the meaning of inheritance in object-oriented programming languages can be best explained using generator functions and their fixpoints. Consequently, it is a little surprising to realize that no current mainstream programming language actually explains inheritance to its users in this way. Instead, most languages make up a “story” that purports to explain inheritance, but that on closer inspection contains serious flaws. It is as if, being asked to explain the facts of life to our children, we are so embarrassed by the truth that we make up a story about storks, knowing even as we do so that it defies the laws not only of biology but also of physics. This paper explores both the truth and the fictions about how objects are brought into the world. My hope is that future programming languages can tell the children, if not the whole truth, then at least a partial truth that is consistent with the laws of mathematics.

Categories and Subject Descriptors D.3.3 [Programming Languages]: classes and objects; inheritance

General Terms Languages, Design

Keywords inheritance semantics, object initialisation, immutable objects

1. Introduction
Inheritance is one of the central concepts in object-oriented programming. Despite its importance, there seems to be a lack of consensus on the proper way to describe inheritance.

Thus wrote Cook and Palsberg, in their 1989 OOPSLA paper “A Denotational Semantics of Inheritance and its Correctness” [7]. That paper goes on to tell a story about the meaning of inheritance that has become broadly accepted as the standard semantics. Working independently, Kamin [12] and Reddy [15] had previously published their own semantics, which are broadly similar. (Kamin’s is less compositional because it involves taking fix points over the whole program, rather than individual class generators.)

This paper does not contain any new results. It’s goal is to explain to you, the reader, as clearly as possible what you probably already know: how inheritance works. We look at operational and denotational definitions of inheritance, and notice where they differ. We consider the complexity that is unexpectedly introduced by an apparently simple concept: immutable objects. Finally, we consider the question in the title: what shall we tell the children about inheritance. Is the typical object-oriented programmer ready to hear the truth about the facts of life? Or should we continue to make up stories full of fictions and half-truths?

I am interested in these issues because I am involved in designing a new programming language called Grace [5], which aims to be an “as simple as possible” language for teaching object-oriented programming to computer scientists. Because the target audience comprises novice programmers, we want to introduce no more concepts than necessary, but because they will, we hope, become computer scientists, we want those concepts to conform to the laws of logic and mathematics, and not be things that will have to be “unlearned” later. My point of view is that it’s OK for novices computer scientists not to be told the whole truth; it’s not OK to tell them outright falsehoods. I agree with the late John Reynolds in believing that for this we should be convicted of moral turpitude and have our tenure revoked [16].

2. How Objects Work
Many languages don’t allow objects to exist unless they have been generated by a class. This “classes first” approach constrains the teaching sequence, and makes it harder to introduce objects to students.

Grace follows a few other languages, starting with Emerald [4], and including Reddy’s ObjectTalk calculus [15].
O’Caml [13] and Scala [19], in providing stand-alone objects that are independent of any class. A Grace object is very simple: it is a mapping from method names (semantically, elements of an arbitrary index set) to method bodies. A method body is a “parameterised behaviour”, that is, a parameterised expression whose evaluation can have effects. The act of “sending a message” to an object, which in Grace we call “requesting a method” of the object, involves parameterising the method using the arguments provided in the method request, and evaluating the method body.

Here is a point object in Grace syntax.

```python
def point34 = object {
    method x { 3 }
    method y { 4 }
    method magnitude {
        ( self.x^2 + self.y^2 ).sqrt
    }
    method isSmaller ( p ) {
        self.magnitude < p.magnitude
    }
}
```

It’s an immutable object, so it doesn’t need “instance variables”; we could give it “instance constants”, but we won’t bother with this detail here. Apart from the basic syntax — that def defines a name, that object constructs an object, that method introduces a method, and that . and the operator symbols request methods — there is one thing here that needs careful explanation: self.

We can explain self operationally or denotationally. The operational explanation is that self refers to “the message receiver itself” [10, p. 50]. Whenever a method is requested of an object, a reference to that object is pushed onto the execution stack; self means the object thus referenced [10, p. 603–4]. Consequently, when p.magnitude is executed, a reference to p is pushed, and any occurrence of self in p’s magnitude method thus refers to p.

The denotational explanation says that self is the object created by the object constructor in which it is statically nested. More formally, an object is a mapping from method names to method bodies; the environment in which the method bodies are interpreted is the syntactically-enclosing environment, augmented by bindings for the method’s parameters and for self. To actually build this mapping, the object constructor must build a generator function: a function from objects to objects. The object that we want is the fixpoint of this function. If we assume that an object declaration does not have a side-effect on the state, we can paraphrase and simplify Reddy’s formulation [15], to get:

\[
\eta = \text{fix}(\lambda s. \rho_0[m_i \mapsto (\lambda \vec{\alpha}. \{\vec{e}_i\} (\eta[\vec{\alpha} \mapsto \vec{a}, \text{self} \mapsto s]))])
\]  

(1)

Here, \(m_i\) is the name of the \(i\)th method, \(\vec{p}_i\) its parameter list, and \(\eta\) the environment surrounding the object constructor. Square brackets are used to build a new environment by adding a list of bindings to an existing environment; \(\rho_0\) is the empty environment. Methods are functions that take a list of arguments \(\vec{a}\), and return the meaning of the expression in the method body. (This will usually be a function from a store to a pair consisting of a store and a value, but for our purposes, it doesn’t really matter.) Here, the method body meaning is determined in an environment in which the method’s parameters are bound to its arguments, and self is bound to the parameter to the generator function.

It’s fine to have two explanations of something, so long as they correspond. As Cook and Palsberg [7] showed, the operational and denotational definitions do mostly correspond, but notice one place in which they differ. The operational definition binds self when a method is requested, whereas the denotational definition binds self when the object is created. Specifically, this means that there is no way that the denotational definition can allow the isSmaller method from point34 to be “reused” in some other object. Such reuse would require an “unbound” self that could be “rebound” after reuse, but the denotational semantics states that self is bound at the time that the method suite for point34 is created. In contrast, the operational definition — along with most implementations — binds self when a method is requested.

Reasonable people may disagree on whether the denotational view is actually a falsehood, or just incommensurable with the operational view. We will return to this point in Section 4.

### 3. Adding Classes

As soon as we consider adding classes, we must confront the question: for what purpose? In a classic paper, Borning [6] lists eight roles played by classes in Smalltalk. Primary amongst these roles is generating new objects, so we will look first at how to support that role. We have already seen how to generate a single object, using an object constructor. It is easy to create an object that plays the object generation role of a class; we need only wrap an object constructor in a method, like this.

```python
def pointFactory = object {
    method x(xcoord)(ycoord) {
        object {
        method x { xcoord }
        method y { ycoord }
    }
    method magnitude {
        ( self.x^2 + self.y^2 ).sqrt
    }
    method isSmaller ( p ) {
        self.magnitude < p.magnitude
    }
}
```
literals binds it earlier, at object construction time. We can be bound late, at method-request time, but the denotational semantics must be preserved. For example, suppose that we follow Reddy [15] and try to define a manhattanPoint method, which calculates its magnitude differently from the points made by pointFactory.

\[
\text{method magnitude} \{ \text{self}.x.abs + \text{self}.y.abs \} ,
\]

but which is otherwise similar.

We would like to inherit the remaining methods, and specifically the isSmaller method, from pointFactory, but we can’t. There are two reasons for this. The first is philosophical: pointFactory is an object, and the central tenet of object-oriented programming is that objects are encapsulated behaviour. In other words, the only thing that we can do to an object is request that it execute a method. In the case of pointFactory, there is just one method: \(x()y()\). Moreover, we can’t create a new object with

\[
\text{def point513 = pointFactory.x(5)y(13)} ,
\]

extract the isSmaller method from that object, and somehow “reuse” it in manhattanPoint. At least, we can’t admit to such an act without admitting to moral turpitude, for we would have to admit that we told a falsehood about encapsulation.

It’s possible that we can weasel our way out of “the encapsulation corner” by arguing that object encapsulation applies only to clients, and that as programmers we have special privileges. Even if that argument were to prevail, there is a second reason that we can’t “reuse” the isSmaller method from point513 in manhattanPoint. According to the denotational semantics, that method has its self pseudo-variable bound to point513, so the request self.magnitude must — according to the semantics that we have taught our students — be a request on the magnitude method of point513, which is the Euclidean magnitude method, and not what we wanted. To put it another way: we should like self to be bound late, at method-request time, but the denotational semantics binds it earlier, at object construction time.

It should now be clear that to serve as a basis for inheritance, a class needs to contain a version of the inheritable methods in which self remains unbound. In the world of semantics, these are sometimes called pre-methods [1]: Cook and Palsberg [7] call them generators. In contrast, an instance needs versions of its methods that are ready for execution, in these versions self is already bound. Given the standard fixpoint semantics, we need to keep two versions of the methods: one, which I’ll call the template, for use in inheritance, and the other, which I’ll call the behaviour, for execution when a method is requested on an object. Indeed, the standard semantics requires that, for a class with \(n\) instances, there are \(n + 1\) different versions of the method suite: \(n\) for the objects, all different in the binding of self, and 1 for the template, in which self is unbound. The template is used both for generating new instances and for generating new classes through inheritance.

We know that this doesn’t correspond to reality: it is pragmatically important that all the objects of a given class are able to share the same code, at least if they are in the same memory. Moreover, it does not provide a useful basis for reasoning about the cost of a new instance. We should be asking ourselves why what has been the standard semantics of inheritance for the last 25 years is so far removed from the standard implementation.

An alternative way of formulating the semantics would be to keep just the template, and to re-generate the behaviour for an instance on demand, whenever it is required, by taking the appropriate fixpoint. This pretty much what real implementations do: they parameterise methods by a self pointer at the instant they are executed. In that way, all the instances of a class, and the class itself, can share a single copy of the method suite.

If the language in question supports nesting and lexical binding, as does Grace, then each object may need its own bindings for the free variables of the methods, even though the code of the methods is shared. Closures, known at least since Steele’s Rabbit compiler [17], allow the code to be shared, but also allow each object to have its own set of free variable bindings. Thus, an alternative implementation would be to treat self as a free variable that is captured when the closures for an object’s methods are generated.

5. The Story So Far . . .

We have noted that the standard semantics for objects does not correspond to the standard implementation, and instead requires two “versions” of each method: a pre-method for the template and a fixed method for the behaviour. We have also noted that the fixed method can be generated on demand from the pre-method, but not the other way around. It seems reasonable to ask why we don’t just put the pre-methods in the objects, doing away with the need to keep both versions. If we make that change, then the need for classes as a device for supporting inheritance goes away.
I understand that there are conceptual reasons to distinguish phenomena from concepts, and that the “Scandinavian School” of object-oriented language design views classes as modelling concepts, and objects as modelling phenomena. The relationship between human thought processes and language design that lies behind this view is perhaps best articulated by Madsen et al. [12, Ch. 18], although I have also argued for the importance of inheritance as an aid to human understanding [3, §5]. Without either endorsing or invalidating this view, I would like to focus here on the technical goal of minimising the number of basic constructs in our language. It is, after all, quite possible to model both phenomena and concepts using the same language construct; for example, prototype objects can be used to represent concepts, while their clones are used to represent phenomena. It is also possible to model concepts using classifiers (such as types) rather than generators (such as classes).

If we do put pre-methods into objects, and bind self at method execution time, we still need some mechanism to play the role of object factories. However, the operation of inheritance could be supported by objects themselves. This is the key idea behind Taivalsaari’s observation that “cloning is inheritance too” [13]: we can use a clone of an arbitrary object as the source of supply for methods to be injected into an “inheriting” object. This model would seem to radically simplify the set of concepts needed to describe inheritance. Is there a catch?

I believe that there is, and it arises from an unexpected source: immutable objects. Let’s take a closer look at how Grace supports immutability, and how its objects differ from those in other languages.

6. Immutable Objects: Creation and Initialisation

Objects generally have internal invariants that should be maintained if their methods are to behave correctly. When object are mutable, the normal mechanism for establishing invariants is to first create the object, and then to initialise the fields to satisfy the invariants. Correct operation of inherited methods generally requires the same invariants to hold in the “new” surroundings; ensuring this is often surprisingly tricky. Let’s look at how a few languages create and initialise objects.

6.1 Smalltalk

Smalltalk doesn’t have any language support for immutable objects. Objects with methods but no instance variables aren’t very useful, unless they are singletons, because any such object must behave identically to every other object of its class. There is no language-support for instance-specific constants; such constant values must be stored in variables, and it’s up to the programmer to communicate their values from the factory method (usually in the class) to the instance.

The key observation that simplifies many of the issues around object initialisation is that object initialisation is not the same as object creation. This distinction is clear in Smalltalk, where object creation cannot be customised by the programmer; instead, all instance variables are automatically created with the value nil. Initialisation takes place after the object has been created, by executing a method. In Pharo Smalltalk, for example, the method new in class Behavior is defined as follows:

```
Behavior >> new
"Answer a new initialized instance of the receiver (which is a class) ..."
↑ self basicNew initialize
```

Here the method Behavior >> basicNew creates the new instance, with all instance variables bound to nil. After the instance has been created, the initialize method in that instance assigns to the variables of the instance, registers the instance, and does whatever else is required by the object’s designer. This makes inheriting initialisation easy: initialize is a real method, and can be inherited like any other. The standard pattern for initialisation is to first super initialize, and only then to initialise one’s own instance variables. If it seems likely that a subclass might need to change the initial value of an inherited variable, the superclass should initialise it using a default value method [2, p. 86], which a subclass can override. Using a method for initialisation also leaves open the option of lazy initialisation [2, p.85], a pattern that is also applicable to other languages.

6.2 Java

The situation in Java is similar to that in Smalltalk; this is clear at the virtual machine level, although somewhat less so at the level of the language syntax. Java object creation is accomplished by the new bytecode. The newly-created object is then passed to a statically bound “method” of the creating class called "<init>": this method is responsible for initialising the object. Java differs from Smalltalk in that the "<init>" method is not written by the programmer but composed by the Java compiler. It comprises the assignment of declared initial values to fields, a call to another "<init>" method (such as the "<init>" of the superclass), which might have been written by the programmer but is otherwise inserted by the compiler, and execution of the code from the so-called “constructor”, as well as any code sequences enclosed in braces in the class definition. Based on the volume of questions on Java discussion websites, it seems likely that few Java programmers understand how the initialisation process works. Even the invocation of "<init>" requires the special bytecode invokespecial: it’s not a normal method invocation, because "<init>" methods are statically bound and not inherited, but neither is it a static invocation, because this is bound to the object being initialised.

What about constants? Java doesn’t really have constants: instead it has “final variables”, which start off life, like all
variables, with a default value and are then assigned to exactly once, usually in the "<init>" method. When can one assign to a final variable? Only when it is definitely unassigned. When is that? The Java Language Specification devotes an entire chapter to explaining the concept of definite assignment [11, Ch. 16].

Why does Java have the complexity of final variables rather than simple constants? I suspect that at least part of the reason is because true constants in objects would need to get their values when the enclosing object is created, something that is not supported by Java’s object creation mechanism.

6.3 OCaml

The OCaml language is interesting because it was designed as an object-based extension of the mostly-functional language ML. OCaml allows mutable fields in objects, but immutability is the default: any field to which the programmer wishes to assign must be explicitly declared as mutable. The designers of OCaml therefore needed to find a story for object creation that did not rely on mutability.

OCaml objects can be created either using classes, which are like functions that create a new object each time they are called, or using immediate object expressions, which generate a single object, but which cannot be inherited [13, Ch. 3]. A let binding in the class (but outside the object) can be used to bind a name to an object; naturally, the binding may be different on each call of the class, so this can be used to get the effect of instance-specific constants. Since such a binding is determined before the new object is created, there can be no question of using the initialisation expression on the right-hand side of the let to manipulate the object: it does not yet exist.

A field inside an OCaml object is given its value before the new object is created, and the expression that computes its initial value cannot refer to the new object. Mutable fields can alternatively be initialised immediately after the object has been created; this is accomplished using an “initializer”, which is an anonymous hidden method. Because the initializer is a method, it can access both self and the instance variables. Unlike other methods, initialisers do not override inherited initialisers; instead, all of the initialisers are run in sequence.

6.4 Grace

Two of our goals for Grace are supporting immutable objects, and making object construction much simpler than in Smalltalk or Java. Immutable objects — not just as a usage pattern (variable objects that happen not to change) but as a concept recognised by the implementation — are important both to support a functional style of object-oriented programming, and to enable efficient implementation in distributed memory computers. Another compelling reason to support immutability is that, even in a parallel program, immutable objects can be accessed without synchronisation.

The full benefits of immutability require that objects are created as immutable. While it is possible to make all objects initially mutable and later to “freeze” them, this introduces its own set of complexities and inconsistencies, as users of JavaScript have found [3]. Moreover, even if a coherent story can be told at the language level, the memory coherency mechanisms of a distributed multiprocessor are unlikely to find it convincing. Thus, we sought a way to make object creation in Grace sufficiently powerful that separate and later initialisation would rarely be required.

Grace achieves this with a combination of two features. First, variables and constants defined inside an object constructor can be set — as part of the creation process — to the result of executing an arbitrary expression. These expressions can include self-requests on the object that is currently under construction. As Gil and Shragai [9] point out, this is potentially dangerous, because the requested method may make assumptions about the object that do not (yet) hold, since the object’s construction is incomplete. 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 one that we already trap.

The second feature of Grace that helps us to create fully-initialised objects is lexical nesting: an object constructor can be lexically enclosed in another structure, such as a method, and the parameters and local bindings of that enclosing structure can be used to calculate the values of the components of the object under construction. Indeed, this is exactly what is happening in the pointFactory example in Section 3: the body of the method x accesses xcoord, a parameter of the method that encloses the object constructor.

Taken together, these features let us build objects with useful constant fields; here is a (highly unrealistic) example.

```plaintext
1 method checker(number) {
2 object {
3   def oneTwoList = aList.generatedby { lst -> aList.with(1,2) ++ lst }
4   def evenCache = self.computesEven
5   method computesEven is confidential {
6     oneTwoList.at(number) == 2}
7   method isEven {evenCache}
8   print "This is a silly example"
9   manager.register(self)
10  }
11 }
12 }
```
In this example, the method checker() answers an object (defined on lines 2–11) whose only public method, isEven, allows clients to ask if number is even. The confidential method computesEven performs an expensive calculation: it determines if number is even by using it to index an infinite list 1, 2, 1, 2, ... and checking if the extracted value is 2.

This example shows that the value of a constant such as evenCache may be defined in terms of a method request, and that the requested method may refer to other constants. What is the operational semantics of such an object constructor? In particular, what object does self denote on line 5? It should be the object under construction, which therefore needs a computesEven method, which in turn needs the definition of oneTwoList. Note that Grace also allows statements at the top-level of the object, such as on lines 9 and 10; this code will be executed for its effect during object construction. Allowing such code is convenient, but doesn’t actually add any extra power: arbitrary effectful code can in any case be part of an expression used to initialise a field.

These considerations led us to a “two-phase” semantics for object construction. In the first phase, the skeleton of the object is built: space is allocated for constants, variables, and a dictionary of methods. The methods are closures, which may bind names defined in the object constructor (such as oneTwoList), or in the surrounding scope (such as number). It’s OK for the methods to reference as-yet-undefined constants: all will be well so long as the constants are defined before the method is requested. At the end of this phase, self has come into existence, but its fields have not yet been defined. In the second phase, the code inside the object constructor (but outside of the method bodies) is executed: in our example, this includes the expressions that give values to the fields oneTwoList and evenCache, and the print and register requests. We would like to say that both phases happen at object creation time, and indeed, from the point of view of a client, the object does spring into existence atomically. However, if cross-examined in a court of law, we would have to be very careful. If the definition of oneTwoList were moved after the definition of evenCache, then the request of computesEven would access an uninitialised constant, which could not happen if constant declaration were truly atomic.

Evaluating this design honestly, we are forced to admit that it is excessively complex. Perhaps more importantly, it does not meet our goal: under certain circumstances, objects creation can be visibly non-atomic, and the timing of external requests can determine whether or not they fail by accessing an uninitialised variable.

In Grace, methods are public by default. The method computesEven is explicitly annotated as confidential, which means that it can be requested only of self.

7. Inheriting Initialisation

Inheritance and initialisation interact in sometimes-surprising ways. As Gil and Shragai [9] discuss, the very feature that gives inheritance its power — being able to override inherited methods — means that inherited code can never be sure what a self-request will do. The initial design of Grace avoided these problems by inheriting from already initialised objects. Let’s look at the canonical example:

```plaintext
    def colourPointFactory = object {
        method x(xcoord)y(ycoord)colour(c) {
            object {
                inherits pointFactory.x(xcoord)y(ycoord)
                method colour { c }
            }
        }
    }
```

The original semantics was to copy the methods and fields of the object specified in the inherits clause (the super-object) into the object under construction (the sub-object). This semantics has the property that the super-object was fully created before the sub-object even began to exist, so there is no possibility of overriding methods that may be involved in its initialisation — essentially Gil and Shragai’s monomorphism restriction. This is both good and bad: it preserves safety, but makes it harder for the sub-object to change the behaviour of the super object in ways that may be both safe and useful.

The “object copy” semantics had two effects that surprised us. The first is that any object is now copyable, even if it has no copy method: object {inherits x} returns a copy of x. It also means that if the creation code of the super-object is modest [9], for example, by exposing a reference to self to some other object, then that reference will be to the super-object and not to the sub-object.

Because of these issues, we changed the semantics of inherits to eliminate the implicit copy and instead require the programmer to supply a fresh object in the inherits clause. This fresh object is then mutated into the sub-object. This changed the symptom of modesty: now, if the super-object is modest, other objects can observe the mutation.

To try to conceal the mutation, we delayed the execution of code that might expose self as far as possible. We did this by extending the “2-phase initialisation” rule, introduced in Section 6.4, to cover inheritance too. Informally, the idea is that, in the first phase, the “shell” of the sub-object, including its fields and methods, is constructed first, and is given a new object identifier id. Then the shell of each of the super objects up the inheritance chain is concatenated onto the first shell; the whole composite object keeps the identifier id. In the second phase, all of the code involved in object construction is executed, starting with the ultimate super-object and working down to the sub-object. During this process, self is always id, which means both that any references exposed
externally are to the composite object, and that code used to calculate initial values can be overridden by sub-objects.

The disadvantages of this proposal include all of the disadvantages of the original 2-phase construction rule. The rule is baroque; it’s hard to imagine explaining this to a classroom full of novice programmers without having to apologise. Mutation is still visible: although the shell of the object is complete before self is exposed, not all of the fields are initialised, and other objects can consequently see constants change from uninitialised to initialised. These disadvantages are compounded by the loss of referential transparency: an object expression in an inherits clause has a meaning different from the same expression in other contexts. Moreover, the “freshness” requirement means that in practice the superobject expression is almost certainly a request on a factory method, which effectively means that classes are required for inheritance.

8. What shall we tell the children?

Is there a simple and consistent story for inheritance and object initialisation? I believe that there is, in the presence of what may seem to be draconian language restrictions: eliminate from object constructors both initialised declarations and top-level statements! These restrictions mean that no user-written code is executed during object construction, self cannot escape, and creation code cannot be incorrectly overridden, thus observing the “hardhat” restrictions \[9\]. Its actually fine to allow variable declarations without initialisers inside an object constructor, but this adds no power, so for simplicity let’s imagine that we eliminate all declarations.

It may seem that in imposing these restrictions, we have severely impaired our programming language, but that is actually is not so. This is because Grace supports nesting and lexical scope. Indeed, definitions and variables declared inside an object constructor are not really instance constants and instance variables in the conventional sense. They are just local variables of the constructor body; they will continue to exist after the body has completed execution if and only if they are referenced by the returned object, that is, if they are captured by the body of one or more of the object’s methods.

To see this, lets suppose that two methods in an object representing a game wish to share a variable; for the sake of example, we will call them setBoardSize and boardSize. All that is required is for this variable to be declared in some scope surrounding the methods:

```grace
method newGame {
  var size
  object {
    method setBoardSize(n) { size := n }
    method boardSize { size }
  }
}
```

Here, the factory method newGame declares a local variable size, and then returns a new object, whose methods close over size. The effect is exactly the same as if size were declared inside the object.

If the effect is the same, how does moving the variables outside of the object help? Because the object’s initialisation expressions are also moved out of the object, so the object constructor is guaranteed to be free of side-effects. Let’s rewrite the example from Section \[6,4\] in this style:

```grace
method checker(number) {
  def result = object {
    method computesEven is confidential {
      oneTwoList.at(number) == 2
    }
    method isEven { evenCache }
  }
  def oneTwoList = aList.generatedby {
    lst -> aList.with(1,2) ++ lst }
  def evenCache = result.computesEven
  print "This is a silly example"
  manager.register(result)
  result
}
```

Notice that we have moved all of the defs, and all of the top-level statements, out of the object constructor and into the factory method checker. The methods in the object result still have access to evenCache and oneTwoList. This code has similar “2-phase” semantics to the original version — first build the object exporting the method isEven, and then initialise the constants oneTwoList and evenCache. However, this semantics is now explicit in the ordering of the executable code in the factory method.

In addition to forcing the evaluation of initialisation expressions to be completed before an object is constructed, these restrictions have at least three negative consequences. First, it seems likely that having declaration and use of variables close together aids in code comprehension, so a method body using temporaries and parameters is easier to understand than a method body using instance variables, and a method body using instance variables is easier to understand than a method body using variables from the surrounding scope. Second, methods that exist solely for the purpose of initialising constants must be moved outside of the object being constructed, and into the enclosing object, where they can be accessed from the initialisation expression. This may be considered good or bad, depending on your point of view. However, a method that is both used to initialise a constant and is exposed to clients poses a problem: it must either be duplicated (clearly bad), or the client version (inside the object) must be implemented by requesting the version in the enclosing object. This is not as bad, but certainly adds more complexity than the version with a single method.

The final negative consequence arises because, in Grace, everything is an object: there is an implicit object constructor surrounding all “top level” code. If we disallow executable code in object constructors, where do we put it? We
could postulate an implicit “main method” that surrounds the whole program, but this seems like a hack. Shouldn’t the principal unit of composition in an object-oriented language be the object?

It is worth nothing that OCaml effectively implements these same “draconian restrictions”, not by banning executable code from inside an object, but by effectively evaluating it (and thus determining the values of an object’s fields) before the object is constructed, as described in Section 6.3. Thus, OCaml does not allow initialisation expressions to reference self. This overcomes the first objection to the restriction (loss of locality), but not the others.

Notice that in this restricted version of Grace, it is still possible for methods, invoked after the object has been constructed, to execute arbitrary initialisation code, expose self, and perform side effects. This is just what happens with the initialize method in Smalltalk, OCaml “initializers”, and Java “constructors”. The innovation in Grace — as in OCaml — is that this is no longer the only way of giving values to instance variables: in the common case where a variable’s value can be determined before the object containing it is created, the variable can already have its value at object creation time. As one of the anonymous referees pointed out, while this facility is necessary for immutable objects, it is also beneficial for mutable objects.

9. Conclusion

Defining the exact semantics of inheritance in Grace has proved to be surprisingly difficult. We started out with a number of requirements, all of which seemed reasonable, given the goals of Grace, and which were not in obvious conflict with each other:

R1: it should be possible to create immutable objects (in addition to mutable objects);

R2: it should be possible to inherit from objects (in addition to classes); and

R3: the semantics of object inheritance should be very similar to those of class inheritance.

Our original “object copy” semantics achieved R1 and R2 but not R3. In eliminating the copy, and adopting instead the “fresh object” semantics, we achieved R3, but at the expense of R1 and R2. Eliminating executable code from object constructors enables us to achieve R1, R2, and R3. However, it too has some disadvantages. If we simply forbid code in the constructor to mention self or other local variables — essentially OCaml’s solution — then we are imposing a restriction that is absent from “mainstream” object-oriented languages — a possible negative for our intended audience. Forcing the state variables of the object out of the object constructor and into the enclosing scope avoids that restriction, but imposes what may be seen as an even more significant one; in any case, it is also a departure from the mainstream.

None of these solutions seems “obviously right” for a teaching language. Is it possible to satisfy all three requirements without stepping outside the mainstream? Can MASPECGHians help us to find a better solution, or to choose between the alternatives described in this paper?

Acknowledgments

The author thanks Kim Bruce, James Noble, and David Ungar for fruitful discussions that have helped to shape this article. The anonymous referees also provided valuable guidance.

References


