An informal inheritance of objects, classes, and inheritance in Grace

Kim B. Bruce

August 11, 2015

Abstract

We present a brief informal semantics of objects and classes (including inheritance) in Grace. An important goal is to present a clear description of how objects are initialized when created.

1 Introduction

The semantics of object-oriented languages can be complex, especially the semantics of object creation in the presence of inheritance. We refer the reader to [Bru02] for the formal semantics of an idealized object-oriented languages. However, that work simplifies aspects of object-oriented languages. In particular, it does not allow the code initializing instance variables to refer to **self** or **super**. As a result, the code initializing instance variables may not make method requests on methods defined in the class or its superclasses.

A variety of papers have been written that discuss the complexities of initialization (cite them), and make suggestions on language restrictions to control that complexity and/or avoid problems. It is also worth noting that there are often subtle differences in the meaning of initialization even in languages as similar as C++ and Java.

A simple restriction that makes initialization relatively straightforward is the one cited above that was made in [Bru02]. Do not allow the initialization code to refer to **self**. However, that is very restrictive and frequently results in requiring programmers to make explicit requests to initialization methods after objects have been constructed. In theory, this is not a problem, but, in practice, programmers, especially novices, often forget to invoke this code. As a result, we have not made this restriction in Grace. Programmers are allowed to make **self**-requests in initialization code.

The meaning of **self**-requests in initialization code is not always obvious in the presence of inheritance. We provide a simple example to expose the problem:

```
class b.new(z: Number) {
  def x: Number = self.m(z)
  method m(n: Number) -> Number{
    print "b's m with {n}"
    n
  }
  ...
}

class c.new(w: Number) {
  inherits b.new(w+1)
  def y: Number = self.m(w)
  method m(n: Number) -> Number {
    print "c's m with {n}
      2*n
   }
```

```
    ...
}
```

In the code above, b is a class that takes a single parameter z, and provides method m that prints a short message and then returns the value of its parameter. It also provides a **def** x that is initialized via a **self**-request of m with argument z.

Class c inherits b.new(w+1), adds a new **def** y that is initialized by a **self**-request of m, and overrides the definition of method m so it now prints a different message and returns twice its parameter.

The semantics of initialization in the presence of inheritance is generally that when an object is created, the initialization code for the superclass is run first, followed by the initialization code specified in the subclass.[1]

In the example, the initialization for x in the superclass is the result of a **self**-request of m. When we evaluate b.new(5), it is clear that initializing x will result in the execution of the body of method m in class b. In particular, it will print `b's m with 5` and return the value 5.

However, it is not obvious which version of m should be executed to initialize x when creating a new object from class c. In fact, C++ and Java differ in the choice of which code to execute in this case. In C++, when an object is created from the subclass (derived class), the initialization code for the superclass is executed in a context where the meaning of **self** is an object of the superclass.

If we evaluate c.new(5) using the initialization protocol of C++ then when x is initialized (as a result of the **inherits** b.new(6) clause), the **self**-request will result in executing the body of m in b. Thus it will print `b's m with 6` and x will be given the value 6. Now that the superclass initialization code has run, the initialization code in c will be run, resulting in a **self**-request of the overridden m specified in c. The execution of m will result in printing `c's m with 5` and y will be initialized to 10.

Java differs from C++ in executing the initialization code for the superclass in a context where the meaning of **self** is an object of the subclass. Thus the execution of the initializing code in the superclass to initialize x will result in the execution of the overridden m, printing out `c's m with 6` and initializing x with value 12. As before y will be initialized by executing the overridden m, printing `c's m with 5` and y will be initialized to 10. A complete specification of the creation of new class instances in Java can be found in section 12.5 of the Java Language Specification [GJS+14].

Now that we've seen that the semantics of initialization can involve choices, in the next section we provide a description of the semantics of object creation in Grace.

2 The semantics of object creation (without inheritance)

We begin with the semantics of evaluating an object expression without inheritance[2] for simplicity.

Suppose we have an object expression with the following form:

```
object {
    def di = ei
    var vi := fi
    method m(pi) {
        bi
    }
    ci
}
```

While we have listed these in the order **def**s, **var**s, **method**s, and then initialization code (ci), they may in fact occur in any order.

When this expression is evaluated the following takes place:

[1] When we specify running the initialization code for the superclass, we mean running not only the code specified in the superclass, but also all the initialization code from its superclasses as well – i.e., the recursive definition.

[2] In actuality all objects inherit from a top class that provides default specification for methods like ==, !=, and asString, for example, but we ignore that for now.

1. Space is allocated with room for all **def**s, **var**s, and **method**s declared in the expression. The meaning of **self** is associated with this (as-yet-uninitialized) object.

2. The meanings of the methods are associated with the method names in the object. Because they are closures, no code is actually executed.

3. The initialization code (including computing the (initial) values of **def**s and **var**s) is executed from top to bottom in the **object** expression.

Because space is allocated for each of the features of the object before the initialization code is executed, the initialization code for **def**s and **var**s as well as the bodies of methods may refer to features that are defined anywhere in the object expression. For examples, methods may be mutually recursive.

We note however that the last (initialization) stage of the evaluation of the object expression may result in accessing a **def** or **var** that has not yet been executed. It is the programmer's responsibility to ensure that does not happen. If it does happen it will raise an exception.

Evaluating a class construction expression is similar. The class construction is essentially a method that evaluates to an object expression. Creating an object from a class involves first evaluating the arguments and then going through the three steps above: (1) allocating space and setting **self**, (2) associating methods with their meanings, and (3) running the initialization code.

3 The semantics of object creation (with inheritance)

The semantics of objects defined using inheritance is only slightly more complicated than described in the previous section. For simplicity we assume that we have an object defined by inheritance from a class.

```
object {
  inherits c.new(a)
  def di = ei
  var vi := fi
  method m(pi) {
    bi
  }
  ci
}
```

When this expression is evaluated the following takes place:

1. Space is allocated with room for all **def**s, **var**s, and **method**s declared in the expression as well as those inherited from the superclass.[3] If a method in the new object overrides a method in the superclass then only the new method is allocated space. The meaning of **self** is associated with this (as-yet-uninitialized) object.

2. The meanings of the methods defined in the new object are associated with the method names in the object. Any method names not assigned a meaning are then associated with the corresponding methods from the superclass. References of the form **super**.m(...) in code in the new object definition are interpreted as requests of the corresponding methods from the superclass.

3. The initialization code (including code specifying the (initial) values of **def**s and **var**s) is executed for the superclass (with the parameters provided in the **inherits** clause). Finally the initialization code for the new object is executed.

[3]To be legal, names of new **def**s and **var**s listed in the new object must not overlap with any names that are associated with the superclass. Methods in the new object may overlap with those associated with the superclass if the signatures are specializations of those from the superclass. A method may also override a **def** or **var** from the superclass, but, for simplicity, we ignore that possibility here.

Notes:

1. During the execution of the initialization code for the superclass, the meaning of **self** is that specified in part 1, i.e., it represents the new object.

2. Because the overriding methods in the new object have replaced those from the superclass, all **self**-requests in the superclass initialization code of methods that are overridden in the new object execute using the method body defined in the new object.

Defining a subclass of a class is similar.

References

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics.* MIT Press, 2002.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.