The Green Language Type System

José de Oliveira Guimarães

Campus da UFSCar, Rodovia João Leme dos Santos, Km 110 - SP-264, Sorocaba - São Paulo, 18.052-780, Brazil

Abstract

A programming language that considers basic values and classes as objects brings more opportunities of code reuse and it is easier to use than a language that does not support this feature. However, popular statically-typed object-oriented languages do not consider classes as first-class objects because this concept is difficult to integrate with static type checking. They also do not consider basic values as objects for sake of efficiency. This article presents the Green language type system which supports classes as classless objects and offers a mechanism to treat basic values as objects. The result is a reasonably simple type system which is statically typed and easy to implement. It simplifies several other language mechanisms and prevents any infinite regression of metaclasses.

Key words: object-oriented languages, Green, type system, polymorphism

1 Introduction

This paper introduces the type system of the Green language [1] [2], an object-oriented and statically-typed language which supports garbage collection, classes as first-class objects, introspective reflection, and run-time metaobjects called shells [3]. In Green, one class may be subtype of another without being subclass of it.

Popular *statically-typed* object-oriented languages such as Eiffel [4], Java [5], C# [6], and C++ [7] have type systems with two limitations. First, classes are not objects. Second, polymorphism does not apply to basic values such as `31`,

Email address: `josedeoliveiraguimaraes@gmail.com`, *Phone*: `55 15 3229-5973` (José de Oliveira Guimarães).
URL: `http://www.dc.ufscar.br/~jose` (José de Oliveira Guimarães).

Preprint submitted to Computer Languages, Systems and Structures30 September 2008

`'A'`, and `3.14159`. That is, among classes `integer`, `char`, and `real`[1] there is no common superclass that obeys reference semantics. There cannot be a variable `x` such that "`x = 31`", "`x = 'A'`", and "`x = 3.14159`" are all legal at the same time. As will be seen, Java, C#, and Green support automatic conversions that simulates polymorphism with basic values.

Before going on, it is necessary to define some terms. An *object* is an aggregate of functions (methods) and date (instance variables) which exists only at runtime. A *class* describes the shape of its objects, their methods and instance variables. A *metaclass* is the class of a class. It is a regular class which describes the methods and instance variables of another class. A *first-class object* is an object that can be stored in variables, passed as parameter to a method, or receive a message. For short, objects which are not first-class should not be called objects, although sometimes they are. A *field* of a class is an instance method or variable or a static method or variable. A field is called "member" in C++.

1.1 Classes as Objects

We will now explain why it is important to have classes as objects. Statically typed languages usually support static variables and methods or similar mechanism, such as *once* methods of Eiffel.[2] A static variable `MaxX` in a class `Point` is a variable of the object representing class `Point`. Variable `MaxX` corresponds to an instance variable of the metaclass of `Point`. Java, C#, and C++ support static fields, but they do not treat classes as *first-class objects* — a class cannot be assigned to a variable, for example. This is a limitation as explained by the following items.

- There are two kinds of methods and variables – static and non-static. This is confusing to learn — static and non-static *fields* are declared in the same place (the class) but they are used through different syntaxes and have different meanings.

  Class `Point` of Fig. 1 is an example in Java of a class with static and non-static fields. It is immediately clear that two different concepts were put in the same place, the class. Static method `setMaxX`, for example, cannot be changed to

[1] Each of these languages has its own classes for integers, characters, and real numbers. Here we used the Green names for basic types.

[2] Every call to a *once* method after the first one will return the same value as the first call. Assume the method returns an object or value. Then, a once method plays the rôle of a constant — it does not depend on the object that received the message.

```
public class Point {
    public Point(int x, int y) {
       if ( x <= MaxX && y <= MaxY ) {
          this.x = x; this.y = y;
       }
    }
    public int  getX() { return this.x; }
    public int  getY() { return this.y; }
    public void setX(int x) { if ( x <= MaxX ) this.x = x; }
    public void setY(int y) { if ( y <= MaxY ) this.y = y; }
    private int x, y;

    static public int  getMaxX() { return MaxX; }
    static public int  getMaxY() { return MaxY; }
    static public void setMaxX(int new_MaxX) {
        MaxX = new_MaxX;
    }
    static public void setMaxY(int new_MaxY) {
        MaxY = new_MaxY;
    }
    static public Point getCenter() {
        return new Point(0, 0);
    }
    static private int MaxX = 1024, MaxY = 768;
}
```

Fig. 1. An example of a Java class mixing static and non-static fields

```
static public void setMaxX(int MaxX) {
  this.MaxX = MaxX;
}
```

There is no `this` pseudo-variable in static methods because no object receives message `setMaxX`. The syntax

```
Point.setMaxX(320);
```

just means that method `setMaxX` of `Point` should be called. It does not mean that object `Point` receives message `setMaxX(320)`. If this would be the case, `this` inside `setMaxX` would refer to the object `Point`.

Instance variables `x` and `y` cannot be accessed in static methods because there is no `this`. However, static variables `MaxX` and `MaxY` can be used inside instance methods — an asymmetry beginners to the language find difficulty to understand.

There is only one variable `MaxX` or `MaxY` at runtime. These variables are shared by all instances of `Point` — they are much like global variables that can be used only inside `Point`. However, there can be any number of `Point` objects each one with its own `x` and `y` variables.

Another source of confusion is the combination of inheritance and static

methods and variables. Inheritance was designed to allow a class to inherit instance methods and variables from another class. But in Java a subclass inherits the static fields too. Static method `setMaxX` of class `Point` can be accessed through a subclass `PolarPoint` of `Point`:

```
PolarPoint.setMaxX(320);
```

This mixes two different concepts and gives the impression that there is another set of static fields to `PolarPoint` as happens with instance fields. However, there is only one set of static fields in the program. In particular, there is only one `MaxX` variable.

Yet another source of misunderstanding is the access of static fields using variables. In Java, the code

```
Point p;
p.setMaxX(640);
```

is legal although it does not make sense.

In C++/Java, there is no constructor to initialize the static variables because there is no object which corresponds to the static fields. Therefore, the syntax for mechanisms to initialize static variables do not resemble constructors — another syntax for doing the same thing, set variables. C# employs a better syntax: a constructor for initializing static variables can be declared by putting keyword `static` before a method that has the class name, much like a normal constructor.

If classes were objects in Java/C++/C#, `this` could be used inside static methods and would refer to the object representing the class. And classes could be passed as parameters — there would be polymorphism with classes too. A message send like "`aClass.m()`" would call a static method "`m`" determined only at runtime, assuming `aClass` refer to a class.

- How can a metaobject be attached to a class to intercept message sends to its static methods? A metaobject attached to an object intercepts all messages sent to it. But a metaobject cannot be attached to a class because it is not an object. Of course, one can design a special language mechanism that allows a metaobject to be attached to a class. But these "class" metaobjects would be different from regular metaobjects — static and non-static methods obey different semantics and have different implementations.

  A consequence of this is that a metaobject cannot be attached to a class to control object creation. To see this, let us first explain object creation in *languages that consider classes as objects*. A method called `new` may be defined in a metaclass to create and return an object of the class. Since `new` is defined in the metaclass, the class (considered as an object) has a `new` method. So a message "`new()`" can be sent to the class to create a new object of it. As an example, to create an object of class `A` we could write

  ```
  a = A.new();
  ```

  "`A`" used anywhere in a statement is a reference to the object that is the class `A`. So, "`A`" in "`A.new()`" works as a variable that refers to the object that is class `A`. Now we can attach a metaobject to `A` that intercepts `new` messages, controlling object creation. To control object creation is as easy

as to intercept any message. This is not so easy in languages that do not consider classes as objects.

- Unless classes are objects, no method can accept them as parameter. For example, a `deepClone` method cannot accept a class as parameter and return a copy of it. A method to serialize[3] an object cannot accept a class as parameter. The same applies to a method that makes its parameter persistent. Or to a method `showObject` that shows information on its parameter on the screen. This method assumes all classes define a `toString() : String` method that returns a string with the object data in a format adequate for viewing. Unless classes are objects, they could not be showed by `showObject`.
- An Introspective Reflection Library (IRL) has classes describing the program structure: its classes, methods, method parameters, variables, etc. At runtime, one can ask for the methods of an object or a class, for example. If classes are not objects, there should be different methods for retrieving information on regular objects and on classes. For example, information on the set of methods of the object referred to by variable `d` could be obtained using

  ```
  s = d.getMethods();
  ```

  Method `getMethod` would be inherited from the top-class hierarchy, usually called `Object` (as in Smalltalk, C#, and Java). A similar syntax could be used for classes:

  ```
  s = Point.getMethods();
  ```

  Here `getMethods` should be a static method. Since it is not inherited, it should be added to all classes by the compiler — Green does something like this. Unless the compiler adds these `getMethods` methods, there should be a different mechanism for accessing information on the methods of the class (when considered as an object). This is what most languages do.

  Therefore, if classes are not objects, then objects and classes need separate treatment by the IRL.
- Some operations belong to metaclasses or to classes considered as objects. They do not refer to a specific class instance. They refer to general class characteristics as "what is the maximum value of an integer?", " How many precision digits does a real number offer?", and "what is the date of today?". The existence of these operations prove that static methods/variables or metaclass methods/variables are necessary — they should be allowed in object-oriented languages.

Eiffel, Java, C#, and C++ do not consider classes as first-class objects. Beta [8] [9] [10] does, but class variables and methods cannot be declared. It is as if static class variables and methods could not be declared in Java, C#, or

[3] Transform an object into a sequence of bytes that can be used to recreate the object. Used to transfer objects through a network or store whole objects in the hard disk.

C++. This restriction limits the usefulness of considering classes as objects.

A statically-typed object-oriented language could be designed to support classes as first-class objects. Since classes are objects, they should have classes, which we will call metaclasses (the class of a class). There are three main design possibilities:

(1) all classes share a common metaclass, say `Metaclass`, which is its own class;
(2) each class has its own metaclass, which is also a class and therefore has a metaclass;
(3) some classes (but not all) share a metaclass.

In option (1), `Metaclass` is its only class, a little bit confusing mechanism which means that a class has information on itself. That does not cause any recursion problem, however.

Option (2) may lead to a problem called "infinite regression of metaclasses" which happens when the metaclass hierarchy never ends. That is, a class `A` has a metaclass usually nameless but which we will call `MA`. Since `MA` is a class, it has a metaclass `MMA` which has a metaclass `MMMA`, and so on. The hierarchy is *potentially* infinite. Possibly the recursion ends in a common metaclass like in option (1) and, therefore, there is not infinite regression. Option (3) is usually used in conjunction with options (1) or (2).

In option (2), if there is infinite regression, it is necessary to create metaclasses at runtime, on demand, a slow operation that is not reasonable in a statically-typed language. It *may* be necessary to call the compiler to compile each metaclass. The dynamic creation of classes is also harmful to some compiler optimizations. It becomes more difficult to the compiler

- to inline methods;
- to replace a message send by a test with branches for each possible class of the message receiver (thus eliminating a runtime search for a method);
- to optimize the use of memory by the method tables, if they are used. Some compilers generate code for message sends using method tables, which are arrays with pointers to methods. Each class has its table. Since many table positions are empty (they do not point to methods), the tables can be packed [11] at compile time. When a class is created at runtime, packing is more difficult, which may be a problem if many classes are dynamically created. These tables may be very large, as large as the number of different message selectors of all program classes.

Considering the possibilities (1), (2), and (3), there is either a potentially infinite number of metaclasses or the recursion ends in some place. For example, `Metaclass` could be an object of itself. To understand the complexity of

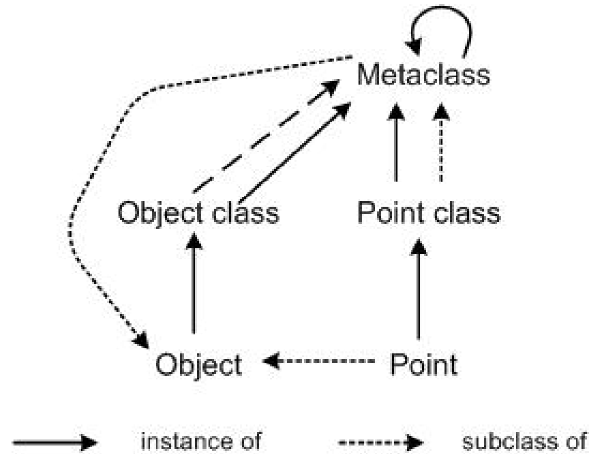

Fig. 2. Inheritance and “instance of” relationships in an imaginary language

a model employing metaclasses, let us study the imaginary language whose class hierarchy and “instance of” relationships are shown in Fig. 2.

In this language, it is difficult to measure the consequences of a modification in a class to other classes that inherit from it or are objects of it. For example, we can ask:

- what happens to `Object` if `Metaclass` is modified? The `Object` class is an object of `Metaclass`. If `Object` is modified, by changes in `Metaclass`, how these changes affect `Metaclass`, which is its subclass? If `Metaclass` is affected, is `Object` changed again?
- what happens to `Metaclass` if `Metaclass` is changed? `Metaclass` is the class of itself;
- what happens to `Point` class if `Metaclass` is modified? `Metaclass` is the class of `Point` class and its superclass. Changes in `Metaclass` may be propagated to `Point` class either by the relation “instance of” or “inheritance from”. Or it may not be propagated at all;
- what was created first: `Metaclass` or its superclass, `Object`? The `Object` class is `Object` of `Metaclass`!

A specialist in this language could answer rather easily these questions. However, regular programmers would get bogged in the intricacies of the arguments.

A real language may differ from the imaginary language used above. It could offer other inheritance/object relationships but all end up with uncommon, to say the less, links among classes and objects. That makes it difficult to understand the language.

One of the causes of confusion is the requirement “every class should inherit directly or indirectly from `Object`”. But why is this requirement desirable in a

statically-typed object-oriented language? To reuse inherited code, the methods of `Object`, and to increase polymorphism. One can declare

```
var anObject : Object;
```

and make `anObject` refer to any kind of object. Variable `anObject` may refer to objects of all subclasses of `Object`, its declared type, which includes everything: every regular object, every class, and every metaclass.

We consider that a subtype is always a subclass. We will see Green gets rid of inheritance in the subtype definition, thus allowing a variable of declared type `Any` refer to any object, including classes (which are objects).

The conclusion of the above discussion is that, to support classes as first-class objects that have classes, one should either create an infinite chain of metaclasses or design complex instance and inheritance relationships among the classes. This may be necessary in a dynamically-typed language but it would not be reasonable in a statically-typed language. We will soon see that the Green classes are objects, which themselves do not have classes, which avoids the problems discussed above.

This ends our discussion of the reasons for supporting classes as first-class objects and the related problems in statically-typed languages. Let us now return to the second limitation of some languages: the non-applicability of polymorphism to basic values.

1.2 Polymorphism and Basic Values

For the sake of efficiency, C++ and Eiffel use value semantics for basic values such as characters, integers, and reals. This means that values such as `31`, `'A'`, and `3.14` are not objects. They could not be passed as parameters to a method like

```
write( anObject : Object )
```

if we consider `Object` as the superclass of every other class. To pass a basic value to `write`, it is necessary to wrap it in an object and then pass this object to `write`. Or method `write` must be overloaded — one `write` method should be created for every basic type. Anyway, different treatments are necessary for objects and basic values. As will be seen, one can define a `write` method in Green that accepts *everything* as parameter: an object, a class object, or any basic value.

In the previous discussion, we were not concerned with known holes in the type systems of C++, Java, Eiffel, and Beta. The discussion focused on deeper characteristics of the type systems of these languages.

The remaining of this paper exposes the innovative features of the Green

```
class Point
      // the constructor appears before the public section
    proc init(x, y : integer)
      begin
      if x <= Point.MaxX   and   y <= Point.MaxY then
         begin
         self.x = x;
         self.y = y;
         end
      end

  public:

      /* this is a comment.
         the bodies of the following methods are not shown */
    proc getX() : integer
    proc setX( x : integer )
    proc getY() : integer
    proc setY( y : integer )
  private:
      /* instance variables */
    var x, y : integer;
end
```

Fig. 3. Syntax for class `Point`

type system and compares them to similar features of other statically-typed languages. The article is organized as follows. Section 2 exposes the Green type system: subtyping, classes as objects, the class hierarchy, and basic classes. Metaobjects of the language are described in Section 3. Section 4 concludes.

2 The Green Type System

Green is statically-typed and has a type system in which subtyping is different from subclassing. Every subclass is a subtype and there may be subtypes that are not subclasses. Before describing the details, let us show a bit of Green syntax. A class `Point` is declared as shown in Fig. 3. This class corresponds to the `Point` class in Java, without the static fields, given as example in Fig. 1. There are public and private sections like in other languages. The public section contains only methods, which are declared with the keyword `proc`. The class constructor is always a method called `init`. In general, Green follows a Pascal-like syntax with a bit of C.

The *signature* of a method is composed of its name, return value type (if it returns a value), and parameter types (in that order). The type of a class is the set of signatures of its public methods. Methods with name `init` are constructors and are not considered in the type (this will be explained later). As an example, the type of class `Point` of Fig. 3 is

```
{ getX() : integer, setX(integer),
  getY() : integer, setY(integer), ... }
```

in which `{` and `}` delimit the set as usual. The "..." represents the signatures of the inherited methods. Since `Point` does not explicitly inherit from any class, it is forced to inherit from class `AnyClass`.

A type S is subtype of a type T if S has at least all the signatures of T. That is, $T \subset S$. A method of a subtype should have the same argument types and return value type as the supertype method with the same name. The covariant and contravariant rules [12] do not apply. This definition of subtyping is used to check assignments, which includes parameter passing. An assignment of the kind

```
aa = bb;
```

is legal if the declared class of `bb` is *subtype* of the declared class of `aa`. For short, we say "class `B` is subtype of class `A`" instead of the longer "the type of class `B` is subtype of the type of class `A`".

Whenever a class `B` inherits from a class `A`, `B` will have at least all methods of `A`, implying the type of `B` is subtype of the type of `A`. Every subclass is a subtype and there may be subtypes that are not subclasses. The only requirement is that the subtype has at least all method signatures of the supertype.

Subtyping may substitute multiple inheritance in some situations. Green only supports single inheritance.

Constructors, the `init` methods, are not inherited and they are not part of the type of the class. Why? Because in general constructors are linked to the implementation of the particular class they are in. They usually accept parameters that are the initial values of some instance variables. Constructors parameters are in general not directly related to the abstraction the class represents. For example, a class `Queue` may have a constructor `init(max : integer)` in which `max` is the maximum number of `Queue` elements. Very probably class `Queue` uses an array to store its elements. A *subtype* `MyQueue` of `Queue` may use a linked list and its constructor may be `init()`. If constructors were part of the type, `MyQueue` would not be subtype of `Queue`. But it should.

The language supports abstract classes which may play the role of types (Java interfaces) of the language. An abstract class may have abstract methods,

which are bodyless. One cannot create objects of abstract classes.

The definitions of type and subtype have a very important property: they do not use the concept of inheritance. They only depend on the set of methods of the class or classes involved. As will be seen, an object in Green can have a type too, even though it exists only at runtime. The type of an object is the set of signatures of its public methods. Then the type of an object may be subtype of the type of a class. This feature allows a classless object be used where a regular object (that has a class) is expected. More about that will be presented soon.

The Green definition of subtyping is not new. It has been used in POOL-I [13], School [14], and Emerald [15]. Even though Emerald does not support inheritance. All of these languages, in slightly different forms, consider types as set of method signatures and consider class `B` subtype of class `A` whenever the type of `A` is subset of the type of `B`.

2.2 Class Objects

Fig. 3 shows a *class* `Point` which specifies the methods and instance variables objects of `Point` will have. Fig. 4 shows the declaration of an object that *is* the class `Point` — it has the methods and variables the class, as an object, has at runtime. In a program file, the code of Fig. 3 should follow that of Fig. 4. Both should be put in a file called "Point.g".

Objects like `Point` are called *class objects*; that is, *objects* that are *classes*. They play the role of metaclasses of other languages, although with much less responsibilities.

Class `Point` of Fig. 3 declares instance methods and variables which corresponds to the non-static fields of class `Point` in Java of Fig. 1.

Class object `Point` of Fig. 4 declares the class methods and variables which corresponds to the static fields of class `Point` in Java of Fig. 1.

The declaration of Fig. 4 is much like the declaration of an object in delegation or prototyped-based languages [16]. The word "`Point`" has two uses in a Green program:

- when used in a variable or parameter declaration as "`var p : Point`", it means "class `Point`", as defined in Fig. 3;
- when used in every other place, "`Point`" works as a constant variable that refers to the object described in Fig. 4. Therefore, `Point` can be stored in variables, passed as parameter, etc. That makes classes first-class objects

```
object Point

  public:
    proc getMaxX() : integer
      begin
      return MaxX;       // it could be self.MaxX
      end
    proc getMaxY() : integer
      begin
      return self.MaxY;  // it could be just MaxY
      end
    proc setMaxX( MaxX : integer )
      begin
      self.MaxX = MaxX;
      end
    proc setMaxY( MaxY : integer )
      begin
      self.MaxY = MaxY;
      end
    proc getCenter() : Point
      begin
      return Point.new(0, 0);
      end

  private:
    var MaxX : integer = 1024,
        MaxY : integer = 768;

end
```

Fig. 4. Object representing class `Point`

in Green. As an example, one can send a message to `Point`:

```
max = Point.getMaxX();
```

The syntax for using a class in a variable declaration or in a method call is the same as in C#/C++/Java. However, in Green, "`Point.getMaxX()`" is a real message send.

A message send "`p.getX()`" calls a method `getX` and inside it the special variable `self` refers to the object that received the message — the object referred to by "`p`". In "`Point.getMaxX()`", method `getMaxX` is called and again `self` may be used to refer to the object that received the message, `Point`. Green uses the same syntax for regular and class (static) methods.

Memory for a class object is created before the program starts running. Immediately after the creation, a parameterless method `init` of the class object is called, if one exists. This method works like the constructor of a class — it should be used to initialize the class-object variables.

Compare this with the static fields of a class, which correspond to class objects. Static variables are created when the class is loaded in Java or when the program starts running in C++. In these languages, there are more than one mechanism to initialize static variables and none of them resembles constructors, which are used to initialize instance variables. Green supports `init` constructors for both classes and class objects.

As in delegation-based languages, class objects may initialize variables in their declaration as is made with `MaxX` and `MaxY` in the example of Fig. 4. Initialization of *class* instance variables is illegal in Green because a class is just a type declaration — there is no memory associated to it.

Although class objects are similar to metaclasses, they do not have classes. Therefore, there is no class of a class, class of a class of a class, and so on. Then there is no infinite regression problem. However, a class object has a type, which is just the set of its public method signatures. Therefore class objects are type-checked as other objects. The type of *class object* `Point` is obtained using the compile-time function `type`:

```
var aClass : type(Point);
aClass = Point;
   /* print Point.getMaxX() */
Out.writeln( aClass.getMaxX() );
```

Variable `aClass` can refer to objects whose types are subtypes of class object `Point`, which includes class object `Point`. Then "`aClass = Point`" is correctly typed.

When someone wants a variable to refer to a subtype of class `Point`, she or he should declare it as

```
var p : Point
```

When the variable should refer to a subtype of *class object* `Point`, it should be declared as

```
var p : type(Point)
```

Therefore the type of a class and its class objects are available in Green. Function `type` gives access to a type of an object, in fact, the type of a class object. Without it, the type of a class object would be inaccessible to the programmer at compile-time. Section 3 shows an example in which function `type` is almost unavoidable.

There is only one class object `Point` in the whole program and it is accessed by a constant variable called `Point`. Methods of *class* `Point` (Fig. 3) can use variable `Point` to call methods of the class object `Point` and to access its private variables `MaxX` and `MaxY`. No other class can access the private part of class object `Point` and this class object cannot access the private part of *class*-`Point` objects.

2.2.1 Relationships between a Class and its Class Object

In C#/C++/Java, objects are created using operator `new`:

```
p = new Point(0, 0);
```

After keyword `new` there should appear the class name and real parameters. This use is legal only when the class declares a constructor which accepts the real parameters. Then the use of operator `new` depends on the class declaration, in particular, on the declarations of constructors.

Green employs something similar. Class constructors are always named `init`. For each method `init` in *class* `Point`,[4] the compiler adds a method `new` to *class object* `Point`. The `new` method has the same parameters as the `init` method of *class* `Point` and returns a class-`Point` object.

When the Green compiler reaches the `init` method of class `Point` of Fig. 3, it adds the following method to *class object* `Point`:

```
proc new() : Point
   begin
   var p : Point;
   p = newly allocated memory large enough for a Point object;
      // if new had parameters, they would be
      // passed to method init
   p.init();
   return p;
   end
```

Therefore Green uses a `new` method of the class object to create objects of the class. This means creation of objects is made through an object-oriented concept, message send. Contrast this with C++, C#, Java, and Eiffel that use operators for that.

A class object `PolarPoint` is unrelated to class object `Point` even if `PolarPoint` inherits from `Point`. That means methods of class object `Point`

[4] There could be more than one method `init` because Green supports method overloading — there may be more than one method with the same name provided the number and parameter types are different.

cannot be called through `PolarPoint` (as in `PolarPoint.getMaxX()`) and class `PolarPoint` cannot access the private methods and variables of class object `Point`. In particular, `new` methods of class object `Point` cannot be called using `PolarPoint`.

A class object provides variables shared by all class instances and supplies methods `new` for instance creation. None of these services should be "inherited" by a class object of a subclass as `PolarPoint`. As explained before, constructors should not be inherited. And to allow a class to access the variables of the class object of its superclass breaks encapsulation. One cannot change the class-object variables of a class because this may damage the code of an undetermined number of subclasses spread throughout the program.

2.2.2 On responsibilities

Class objects have two main responsibilities: to create class instances through `new` and to provide variables (like `MaxX` and `MaxY`) shared by all class instances (Fig. 4).

Class objects have some of the responsibilities of metaclasses [17] [18] [19] [20] of dynamically-typed languages. Metaclasses, in some languages, have much more duties. A metaclass may specify that its object, a class, is abstract or that it cannot be subclassed. A metaclass may automatically provide `get/set` methods for all instance variables declared in the class that is the metaclass instance. Green will allow the programmer to change some implementation characteristics of classes through a yet-to-be-described compile-time metaobject protocol (MOP). Note that a statically-typed language could hardly remain statically-typed and efficient if it provides all functionalities of metaclasses of dynamically-typed languages such as ObjVLisp [18] and ClassTalk [20]. So we chose to move some responsibilities of metaclasses to the compile-time metaobject protocol. We hope the MOP and class objects together will have most of the power of metaclasses of dynamic languages.

2.3 The Class Hierarchy

Class `AnyClass` is inherited by any class that does not explicitly inherit from another class. So, class `Point` of Fig. 3 inherits from `AnyClass`, which inherits from `Any`.

The Green class hierarchy is shown in Fig. 5 which also uses class `Point` and a `char` array to better present the inheritance relationship among the classes. In this figure, a class `B` below and to the right of a class `A` means `B` inherits

```
Any
   AnyClass
      Point
      AnyArray
         array(char)[]
         AnyClassArray
            array(Point)[]
   AnyClassObject
```

Fig. 5. The Green class hierarchy

from `A`. Then `Point` and `AnyArray` are subclasses of `AnyClass`. Class object `Point` is not shown in this Fig.. It is a *subtype* of `AnyClassObject`.

Class `Any` defines some generic methods such as

```
equals( other : Any ) : boolean
shallowClone() : Any
isObjectOf( aClass : AnyClassObject ) : boolean
```

among others. Method `isObjectOf` takes a class object as parameter and returns `true` if the class of the message receiver is class `aClass`. If the receiver is a class object, `false` is always returned. As an example, the expression

```
    (Point.new(0, 0)).isObjectOf(Point)
```

is `true`.

All class-`Any` methods are applicable to any object, even if it does not have a class (it is a class object). A message `isObjectOf` always returns `false` when sent to a class object.

Class `AnyClass`, inherited by all regular[5] classes, defines method

```
    getClassObject() : AnyClassObject
```

which returns the class object of the object. So, if `p` is declared as

```
    var p : Point = Point.new();
```

the expression

```
    p.getClassObject() == Point
```

will always be `true`.

The type of a class object is the set of its public method signatures. Since a class object is an object, that means objects in Green have types. Class `AnyClassObject` is *supertype* of the type of every *class object*. Since it is subclass of `Any`, the type of every class object is a *subtype* of `Any`. For short, we say "every class object is a subtype of `Any`". This means that every class object defines the `Any` methods `equals`, `shallowClone`, and so on. But who defines these methods in every class object? They are not inherited since class

[5] Every class but the basic classes `char`, `real`, `boolean`, etc.

objects are objects, not classes. The answer is: the compiler. It automatically adds all `Any` and `AnyClassObject` methods to every class object.

The scheme presented above consists of:

(a) a carefully designed class hierarchy. Methods available to every object, including class objects, were put in class `Any`. Methods available to every object of a class were put in class `AnyClass`. Methods that make sense only to class objects were put in class `AnyClassObject`;
(b) the addition of methods, by the compiler, to every class object. The methods added are just those of class `AnyClassObject` and its superclass `Any`. Therefore any class object has all the methods of `AnyClassObject` being a *subtype* of it;
(c) every programmer-defined or regular class inherits directly or indirectly from `AnyClass` and therefore is a *subtype* of it.

This scheme makes every object a *subtype* of `Any`. If the object is a class object, it is also a *subtype* of `AnyClassObject`. Therefore class objects, which represents the metaclasses in Green, are integrated in the type system. We are unaware of any other statically-type object-oriented language that achieves that. In general the metaclass responsibilities are left to static variables and methods which are not integrated into the type system. That is, the set of static variables and methods do not compose an object which has a type. The relationship between class objects and the type system is only possible because of two Green novelties:

(1) classes are classless objects which have types;
(2) methods are added to class objects to make them subtypes of `AnyClassObject` and `Any`. Then both class objects and normal objects have a common supertype, `Any`.

2.4 The Basic Classes

Green supports the basic classes `char`, `boolean`, `byte`, `integer`, `long`, `real`, and `double`. These classes inherit from class `AnyValue` which *does not* inherit from `Any`. It was necessary to create class `AnyValue` because basic classes (which inherit from `AnyValue`) and regular classes (which inherit from `Any`) differ in two important points:

- basic classes use value semantics and regular classes use reference semantics;
- regular classes can be subclassed but basic classes cannot, for efficiency reasons.

```
AnyValue
   integer
Any
   AnyClass
      Point
      Integer
   AnyClassObject
      class object Point (subtype)
```

Fig. 6. The Green class hierarchy with basic types

A variable whose type is a regular class is much like a pointer to a dynamically-allocated object. And a basic-class value is never dynamically allocated.

Class `AnyValue` has methods for getting information on the object like "`getClassInfo`" which returns an object with all the class information. So one could write

```
var ci : ClassInfo = 5.getClassInfo();
```

to get information on integers. Another method of `AnyValue` is `toString` for converting a basic value to a string.

The class objects of basic classes have `cast` methods for type conversions. For example,

```
var i : integer = char.cast('A');
```

assigns `65`, the ASCII of `'A'`, to `i`. These class objects have other methods such as for getting the size of basic values and for getting the minimum and maximum values of the class.

Following the conventions of showing the relationships class/subclass of Fig. 5, Fig. 6 shows more of the Green class hierarchy. Only the `integer` basic type is shown in the Fig..

There are wrapper classes, that obey reference semantics, for each of the basic classes. For example, there is a wrapper class `Integer` that inherits from `AnyClass` (not `AnyValue`). This class just holds an `integer` value and has a `get` method for getting it. The value is given at the creation of the `Integer` object and cannot be changed.

This can be implemented in any language. The new point in Green is that automatic conversion is provided between a wrapper-class object and its corresponding value. So, if `i` and `I` are declared as `integer` and `Integer`, respectively, the following code is legal.

```
I = 1;
i = 5*I + i*I;
++I;
```

A wrapper-class object can be used whenever the corresponding basic-class value is expected, and vice-versa. This mechanism allows one to program as if using a pure object-oriented language, in which everything is an object. It gives great freedom to the programmer. The flexibility of this mechanism is illustrated by a method

```
proc write( any : Any )
```

Everything can be passed as parameter: a basic-class value (`5`, `'A'`, `3.14`), a class object, or a regular object.

In language C# [6], a basic value is implicitly converted to the type `object` (the top-level class) whenever necessary:

```
object anObject = 1; // correct !
```

This mechanism is called boxing. Unboxing should be made by the casting operator `()`:

```
int anInt = (int ) anObject;
```

Green employs a more general and complex mechanism in which automatic conversions are made even within expressions and explicit casts are not demanded.

Note that the new version of Java [5] has a mechanism also called boxing that has exactly the same functionalities as the automatic conversions of Green.

3 Metaobjects

Green supports run-time metaobjects called *shells* [3] and has an Introspective Reflection Library (IRL). A metaobject attached to an object intercepts all messages sent to it. It can redirect the message to another object, check parameters, send the message to the original receiver, and so on. A shell class

```
shell class Control( type(Planet) )
  public:
    proc new() : Planet
      /* user-defined body is not described */
end
```

is a metaobject class in Green. A metaobject of `Control` may be attached to objects which are subtypes of `type(Planet)`, which obviously includes class object `Planet`. Remember `type(Planet)` is the type of class object `Planet`, it is the set of all public method signatures of class object `Planet`. Shell class `Control` can define only public methods of `type(Planet)`. A metaobject of shell class `Control` is attached to *class object* `Planet`, thus controlling its behavior, by the following statement:

```
Meta.attachShell( Planet, Control.new() )
```

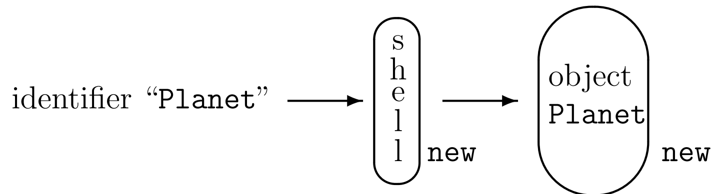

Fig. 7. Shell intercepting method `new` that creates an object. In the code, object `Planet` is referenced by identifier "`Planet`".

In the Green code, identifier "`Planet`" refers to the class object which we also call `Planet`. That is: in the code, "`Planet`" is a pseudo-variable that points to the class object `Planet`. After the shell attachment, identifier `Planet` refers to the shell object that refers to the class object `Planet` — see Fig. 7. In this figure, arrows mean references between objects.

After the attachment, any message `new` sent to class object `Planet` will be intercepted by the metaobject. Method `new` of the shell will be executed instead of method `new` of class object `Planet`. Therefore, in an object creation

```
p = Planet.new();
```

method `new` of the shell will be called — identifier `Planet` in the line above in fact refers now to the shell object. Note that this shell does not intercept *all* message sends to class object `Planet`. It only intercepts `new` messages.

This is one more advantage of defining classes as classless objects. No special metaobject protocol is necessary in order to intercept the very simple operation of object creation. The compile-time function `type` is fundamental in order to control object creation through metaobjects. Without it, the example above would be much more complex. We should have to create an abstract class, say `TypePlanet`, to replace `type(Planet)` in the declaration of shell class `Control`. This class should declare one abstract method for each public method of class object `Planet`. And all methods of `AnyClass` should be added to class object `Planet`. Only in this way `TypePlanet` would be a supertype of class object `Planet` and then a `Control` shell could be attached to it.

Open C++ 1.2 [21], Beta [8], and MetaXa[6] [22] [23] are statically-typed object-oriented languages that support metaobjects. In these languages, the creation of an object may be intercepted by using special metaobject protocols. Compare this to Green in which no protocol is necessary: one just has to attach a shell to the class object. The shell should define a `new` method. Object creation in Open C++ and MetaXa are more difficult to deal with than in Green because the former languages do not consider classes as first-class and *typed* objects. Beta does consider classes as first-class values but does not provide a `new` method to be intercepted.

[6] MetaXa is a Java extension that supports metaobjects.

4 Conclusion

Green does simulate the property "a basic value is an object" without giving up efficiency or making the language too complex. It just makes automatic casting between wrapper objects and basic values, a simple trick. There is loss of efficiency when, for example, an `integer` is converted to `Integer` or vice-versa. But this was an option of the programmer to make the code more readable or polymorphic.

Green considers classes as classless objects thus eliminating the need for metaclasses and complex hierarchies. Subtyping is not tied to subclassing. A subtype only needs to have all supertype method signatures. This makes it possible to type objects, since inheritance is not used in the subtype definition. Class objects have types and their use can be type-checked. The use of *subtyping* with classless class objects is the most important innovation of the Green type system. This mechanism made simpler or more generic most of the language features. It:

- eliminates the infinite regression problem since class objects are classless. There is no metaclass, no class `Object` which is superclass of its own metaclass, no strange "inheritance from" and "instance of" relationships;
- provides a simple type system since the definitions of type and subtype are also used with objects;
- makes the class hierarchy relatively simple. The top classes are four: `AnyValue`, `Any`, `AnyClass`, and `AnyClassObject`. Every basic class (`char`, `integer`, ...) is subclass of `AnyValue`. `AnyClass` is *superclass* of every user-defined class and `AnyClassObject` is *supertype* of every class object. Both classes are *subclasses* of `Any`. Although there are *four* top classes, the other classes relate to them by the normal subclass and subtype relationships, which are easy to understand;
- allows one to control object creation through metaobjects without any special language support;
- makes Green support some of the facilities of prototype-based languages such as one-of-a-kind object and less abstract programming. After all, class objects are concrete objects which are live during all runtime;
- can be easily implemented: a class object is much like the single object of a hidden class. In fact, in our compiler this hidden class inherits from `AnyClassObject`;
- makes uniform the declaration of instance and class (static) methods. In both, `self` refers to the object that received the message. The name of method "`init`" is used for both a class constructor and a class object constructor;
- allows the programmer to define methods or constants in a class object and use them without creating an object. This is important for methods like

writeln of class object Out for standard output and In for input. To output a variable is as easy as "Out.writeln(i)". This solution does not demand public global variables like cin and cout of C++ or public static variables like out of class System in Java. Green does not need to allow either global variables or public instance variables in order to supply globally accessible data like In and Out.

The concept of object is the easiest to learn in object-oriented programming. The class concept is a little more difficult since classes are not entities that will be alive at run time with which the user program can interact. Metaclasses are even more difficult to grasp because they are in an abstraction level above classes. And their existence usually requires relationships among classes, both subclass of and instance of, that are difficult to understand. The simplicity of the Green type system comes from putting the easiest to learn and more concrete concept, that of the object, in place of the more difficult concept, that of metaclass.

Even though class objects do not have all features of metaclasses of some dynamically-typed languages, they offer the services expected in a statically-typed language. The missing features, however, will hopefully be added to a compile-time metaobject protocol.

The Green object-oriented exception handling system [24] is a lengthy subject which could not be discussed in this paper. The Green exception system is an object-oriented version of the exception system of Java/C#/C++. Language Java has the most safe system of the three. Its system has specific rules for

- exception signalling with throw. Using the class of the thrown object, the compiler checks if the exception will be caught by some catch clause;
- exception declaration: in the method header one should declare the exceptions the method may throw.

These rules, in Java, prevent run-time errors. Green, instead of using catch clauses after a try block, groups the code of these clauses in an object that is attached to the try statement. Each catch clause corresponds to a method of the object (all of them have the name "throw"). This object is responsible for the error treatment of that try block. An exception is thrown by a statement "exception.throw(anObj)" in which exception is a special object. In this way we replace specific rules for the exception system by object-oriented rules since the exceptions are treated by an object and thrown by an object. In Green, we can build class hierarchies for error treatment, the Introspective Reflection Library can be used to discover information on the exception system (more than in other languages), metaobjects can modify the error treatment of a try block, and the object attached to a try block can be replaced dynamically. Then the exception system does not need specific rules — it obeys those of

the type system.

All the Green features presented in this paper, but metaobjects, are supported by the Green compiler [1] (metaobjects have been implemented by an old compiler). The compiler is freely available, including its source code (in Java) and the libraries. Guimarães [25] discusses some implementations details of this compiler, including code generation to Java. Green classes are translated to Java classes although this is apparently impossible since the subtype definition of Green is more general than that of java (a subtype in Java is a subtype in Green but the converse is not true). Papers [25] [26] explain how the Green type system can be simulated in Java.

Acknowledgments. This research was partially financed by FAPESP under process number 99/13006-8. I thank Ole Madsen for answering some questions on the BETA language and the referees for many important comments relating the clarity of the paper.

References

[1] J. Guimarães, The Green language, http://www.dc.ufscar.br/~jose/green/green.htm, 2004.

[2] J. Guimarães, The Green language, Computer Languages, Systems, & Structures, 32 (2006) 203-215.

[3] J. Guimarães, Reflection for statically-typed languages, in: European Conference on Object-Oriented Programming (ECOOP), 440-461, 1998.

[4] B. Meyer, Eiffel: The Language, Prentice Hall, New York, 1992.

[5] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Third edition, Prentice Hall PTR, 2005. Available at http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

[6] ECMA, C# Language Specification, 4 th edition, June 2006. Available at http://www.ecma-international.org/publications/standards/Ecma-334.htm, 2007.

[7] B. Stroustrup, The C++ programming language, Second edition, Addison Wesley, Reading, MA, 1991.

[8] S. Brandt, R. Schmidt, Reflection in a statically typed and object-oriented language — a meta-level interface for BETA, `http://www.daimi.au.dk/~beta`, 2003.

[9] S. Brandt, J.L. Knudsen, Generalising the BETA type system, in:European Conference on Object-Oriented Programming (ECOOP), 421-448, 1996.

[10] J.L. Knudsen, The Beta home page, `http://www.daimi.au.dk/~beta`, 2003.

[11] K. Driesen, U. Hölzle, Minimizing row displacement dispatch tables, in:Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, 141-155, 1995.

[12] W. Harris, Contravariance for the rest of us, Journal of Object-Oriented Programming 4 (1991) 10-18.

[13] P. America, F. V. D. Linden, A parallel object-oriented language with inheritance and subtyping, in: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, 161-168, 1991.

[14] N. Rodriguez, R. Ierusalimschy, J. Rangel, Types in School, SIGPLAN Notices 28 (1993) 81-89.

[15] R. Raj, E. Tempero, H. Levy, A. Black, N. Hutchinson, E. Jul, Emerald: a general-purpose programming language, Software: Practice and Experience 21 (1991) 91-118.

[16] R.B. Smith, Prototype-based languages (panel): object lessons from class-free programming, in:Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, 102-112, 1994.

[17] N. Bouraqadi-Saâdani, T. Ledoux, F. Rivard, Safe metaclass programming, in: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, 84-96, 1998.

[18] P. Cointe, Metaclasses are first class: the ObjVlisp model, in: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, New York, 156-162, 1997.

[19] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1983.

[20] T. Ledoux, P. Cointe, Explicit metaclasses as a tool for improving the design of class libraries, in: International Symposium on Object Technologies for Advanced Software (ISOTAS), Springer, Berlin, 38-55, 1996.

[21] S. Chiba, Open C++ programmer's guide, Technical Report 93-3, Department of Information Science, University of Tokyo, Tokyo, Japan, 1993.

[22] M. Golm, Design and implementation of a meta architecture for Java, Master's Thesis, University of Erlangen-Nurnberg, 1997.

[23] M. Golm, J. Kleinöder, MetaXa and the future of reflection, UTCCP Report, Center for Computational Physics, University of Tsukuba, Tsukuba, Japan, 1998.

[24] J. Guimarães, The Green language exception system, The Computer Journal 47 (2004) 651-661.

[25] J. Guimarães, Experiences in building a compiler for an object-oriented language, SIGPLAN Notices 38 (2003) 25-33.

[26] J. Guimarães, On Translation between Object-Oriented Languages. Available at
http://www.dc.ufscar.br/~jose/green/Articles.htm

Vitae

José de Oliveira Guimarães is a Professor of Computer Science at the Federal University of São Carlos (UFSCar), Campus Sorocaba-SP, Brazil. He received a BSc, a MSc, and a PhD in Computer Science in 1989, 1992, and 1996, respectively. His research interests include object-oriented programming, computational reflection, compiler optimizations, programming languages, complexity classes, quantum computing, and mathematical logic (mainly computability).