# Exploiting Reflection in Object Oriented Genetic Programming

Simon M. Lucas

Dept. of Computer Science
University of Essex, Colchester CO4 3SQ, UK
sml@essex.ac.uk

**Abstract.** Most programs currently written by humans are object-oriented ones. Two of the greatest benefits of object oriented programming are the separation of interface from implementation, and the notion that an object may have state. This paper describes a simple system that enables object-oriented programs to be evolved. The system exploits reflection to automatically discover features about the environment (the existing classes and objects) in which it is to operate. This enables us to evolve object-oriented programs for the given problem domain with the minimum of effort. Currently, we are only evolving method implementations. Future work will explore how we can also evolve interfaces and classes, which should be beneficial to the automatic generation of structured solutions to complex problems. We demonstrate the system with the aid of an evolutionary art example.

## 1 Introduction

The majority of programs currently being developed by programmers are written in object-oriented languages such as Java, C++, C# and Smalltalk. In contrast, the vast majority of evolved programs use an expression tree representation. Evolving program trees was first done by Cramer [1], and later re-discovered and popularised by Koza [2]. Koza went on to introduce automatically defined functions (ADFs) [3], which provide a degree of program modularity, and can significantly improve the performance of the evolutionary algorithm. Teller [4] introduced indexed memory to allow functions to have state (making them, mathematically speaking, not functions any more, unless we consider the memory as an additional argument to the function).

Some applications can get by with functions that operate only on primitive data types, such as general classes of number. Many types of program, however, can be much more elegantly and compactly expressed by using additional data structures, such as arrays, vectors, matrices and various collections (e.g. lists, sets and maps for example). Strongly Typed GP (Montana, [5]) was developed to support this, and restricts the evolved program tree to call functions with arguments of the correct type. This can drastically reduce the size of the search space, while still allowing all sensible solutions to be reachable. Other alternatives to standard tree-based GP include Linear GP [1, 6, 7], where a program

is specified as a sequence of instructions, and Cartesian GP [8], where nodes are positioned and connected with some spatial constraints. Also of interest are GraphGP [9], Linear Graph GP [10], and Grammatical Evolution [11], in which trees are generated by a context-free grammar.

Of more direct relevance this paper is the work of Bruce[12, 13] on object oriented genetic programming (OOGP), and of Langdon[14] on evolving abstract data types. Langdon and Bruce independently evolved data types such as stacks and queues. Bruce investigated the difficulty of evolving methods that needed to cooperate such that some shared memory would be used in a compatible way. For example, when implementing a stack it is vital that the `push` and `pop` methods cooperate properly. Both Bruce and Langdon, however, focused mostly on evolving tree-type expressions to implement particular methods.

The most similar prior work to this paper is Abbott's initial exploration of OOGP [15]. Abbott also uses reflection to make method invocations, and mentions the ability to exploit existing class libraries as an advantage of the approach, though the the parity problem he uses as an example does not demonstrate this, and instead uses specially defined classes and methods to help solve the problem.

Abbott raises doubts regarding whether the evolution of new classes is really feasible. This is clearly a fundamental point, since without the definition of new classes and interfaces, we are only doing a very limited form of object oriented programming. Indeed, much of the difficulty of software design lies in the identification of useful abstractions. The high level design involves identifying appropriate classes and interfaces. When we make a good job of this, implementing the methods is usually relatively straightforward. On the other hand, if we make no attempt at the higher level design, then the method implementations tend to be complex, poorly structured, hard to maintain and offer very little re-use.

While there are many impressive achievements reported in the recent book of Koza *et al* [16], most of the evolved programs and circuits are relatively small, typically having tens of nodes rather than hundreds or thousands. Making GP scale up to larger systems is still a challenge, and the space in which we search is of fundamental importance.

To quote Koza [2]:

> "if we are interested in getting computers to solve problems without being explicitly programmed, the structures that we really need are COMPUTER PROGRAMS"

Whether we use standard GP or OOGP, we can still theoretically search some space containing all possible computer programs. However, different representations (and their associated operators) structure the space in different ways, and this can have important implications for how evolvable the structures are in that space.

There is also an important practical point. When we evolve a program in standard GP we begin by specifying the set of functions and terminals. How we choose these can be critical to the success of the evolutionary process. It is also a criticism levelled at GP as a general machine learning method: that the choice

of function set strongly biases the search. Hence, the user in making this choice has already partially solving the problem, or in the worst case, prevented a good solution from being found.

In practice, when a programmer writes OO software, they typically make extensive use of existing class libraries. This paper investigates a simple but powerful idea: the use of reflection to enable evolutionary algorithms to directly exploit such class libraries.

## 2 Evolving Object-Oriented Programs

Currently, there seems to be very little work being done on OOGP. At the time of writing, a *Google* search for the phrase "Object Oriented Genetic Programming" returned only 32 pages, compared with 75,900 pages for "Genetic Programming". Of these 32 hits, the majority referred to object oriented implementations of standard tree-based GP.

There are many reasons why we should explore the idea of evolving object-oriented programs:

- To directly exploit the power of existing OO class libraries.
- To better model real-world problems where objects have state.
- To better integrate evolved object classes with an existing project.
- For a certain class of problem, OO programs might be more *evolvable* than expression trees. Object classes (with fields and methods) may provide a more appropriate unit of evolution than function-trees.
- The OO paradigm seems to promote better software re-use than the functional paradigm that standard GP is based on. Evolved OO programs might include classes that can be re-used in other applications.
- The OO paradigm suggests additional genetic operators based on sub-classing existing classes, defining new interfaces.
- Evolution is also possible at the object level, where the state of the objects evolve rather than the program code.

These points surely warrant further study. Well written OO programs can be a joy to read, understand and modify. One interesting aspect of OOGP is whether we can evolve OO software that is similarly well designed. Also of interest here are Object Oriented Software Metrics [17], both in aiding the objective analysis of evolved programs, and perhaps building these metrics into the fitness function used during evolution.

We wish to stimulate interest in this under-explored area. We will show how easy it is to evolve object oriented programs with the aid of the Java Reflection API. Reflection allows a program to discover at run time the class (type) of an object, the methods that can be invoked on objects of a class, whether the methods are class or instance methods, and the fields of an object. For each method we can look up its return type and the types of its parameters.

Using this information it is straightforward to randomly generate a program as a sequence of method invocations on a set of objects. This means that we can

write an entirely generic OOGP system that will evolve method implementations to match a specified interface. If this approach fails to converge in practice, owing to the size of the search space, we are still free to then restrict the space by applying prior knowledge of the problem domain by restricting the set of existing methods to be used.

## 3 Our Reflection-based OOGP Implementation

Suppose we have written an object-oriented program, but wish to evolve some parts of it. The standard way to approach this using toolkits such as ECJ [18] is to define a set of functions that are appropriate to the problem at hand, and then evolve an expression-tree that uses those functions. For each function it is necessary to write a new Java Class definition that sub-classes `GPNode`. This involves some straightforward but slightly tedious manual effort for each new class of problem to be addressed.

### 3.1 The Java Reflection API

An interesting alternative is to use reflection to determine directly the set of possible methods and variables to be used. Manual intervention is still possible, in that we can specify the set of object classes and/or methods to be considered, but it is no longer necessary to write wrapper classes or methods to invoke these.

Reflection in an OO language is where a program is able to discover things about itself at run time, such as the class of an object, the fields associated with it, its superclass, the set of interfaces it implements, its set of constructors, and the set of methods that can be invoked on it. Java has classes called Method for each method defined for a class. By getting a reference to the Method object, we can then invoke the corresponding method. Using reflection we can generate and evolve the objects constructed by a program, and the methods invoked on those objects. In languages such as Smalltalk or certain OO variants of LISP, it is even possible to generate new classes at run-time. In Java, this is not easy to do directly - it involves auto-generating Java source files, compiling them, and then loading the compiled classes. As a minimum, all the user of the system need do is specify which method implementations should be evolved, the set of objects to invoke methods on, and the set of objects to use as arguments.

### 3.2 Evolving Method Implementations

Given a method signature (i.e. its return type and list of parameter types), we wish to evolve the code that implements the method.

We consider a method implementation to be a sequence of *Instruction*s. Each Instruction consists of a Method reference (the Method to be invoked), an Object reference (the Object to invoke it on), and an array of Objects - which are the arguments to the method. The Object reference is ignored if the method is static

(i.e. a class method and not an instance method). The array of arguments may be either null or zero length if the method to be invoked takes no parameters.

To avoid confusion with existing Java Reflection classes, such as `Method` we call our implementation of a method a `Program`. Our implementation of `Program` allows the random generation of programs if a specified size $n$, simply by making $n$ calls top the Instruction Generator. The use of a sequence is reminiscent of Linear GP. We allow programs to be mutated by adding a new randomly generated instruction or deleting an existing one - in the prototype, these events are equally likely.

### 3.3 Random Instruction Generation

To assist in generating programs we implemented a class called `Instruction Generator`. This takes two arguments: a set of active objects that we wish to allow methods to be invoked on, and a set of argument objects: objects that we wish to select the set of arguments from.

We use reflection to get the set of methods callable on each object. In Java there are two alternative ways of doing this, with the reflection methods called `getDeclaredMethods()` and `getMethods()`. The former gets all the methods declared in this class i.e. public, protected and private ones, but not any inherited methods. The latter gets all the inherited public methods. Actually, in Java Reflection, the private and protected modifiers only express preferences, and can be overridden. Hence, it is possible to discover the set of all methods that can be called on an object by calling `getDeclaredMethods()` first on the class of that object, then the superclass of the object and so on until we reach Object, the root of Java's Object hierarchy.

Using this we build up two look-up tables called methodLut and typedArgs, both of type HashMap from Java's Collections API. The purpose of methodLut is to provide direct access to the set of callable methods for each active Object - hence methodLut is keyed on the active object, and each entry in methodLut is a collection of methods. To be considered callable, each of the method's parameters must have an object of an appropriate type in the argument set. In order to aid this check, we organise the set of argument objects into the typedArgs HashMap. The key for typedArgs is the type of parameter (i.e. an instance of Class), while each entry in the map is a set of objects of that type. Each object may have many entries in the type map, one for each parameter type that it can be passed as. In other words, an Object is entered in the map for its own Class and for all its super-classes, and for all the interfaces implemented by all those classes.

There is a slight problem with the above description: we use the set of active objects and the set of argument objects to construct the set of instructions, where each instruction is a method to be invoked on a particular object using a particular set of objects as arguments. What we also need is to be able to run the same program (sequence of instructions), but using different objects as arguments, and different objects to invoke them on. For example, if we are evolving a program to draw a picture, then each new version of the program may need to draw the picture on a different Graphics Object. There are several

possible solutions that allow this, such as storing the actors and arguments sets in an array, and then having the instruction storing the index of the actor or argument. Then, by changing the object stored at that index, we can achieve the desired effect.

The solution we preferred, however, is to define an `ObjectHolder` class, which has two fields: `Class objectClass`, and `Object objectValue`. This structure provides local fine-grain control of the object classes and values.

### 3.4 Primitive Arguments

Methods that take primitive arguments (such as `int` or `double` have to be handled specially, since Java draws a distinction between Objects and Primitives. This distinction is made for the sake of efficiency, but means that we have to write extra code to deal with this. When invoking a Method using reflection, the set of arguments to the method is passed to an array of Object (i.e. of type `Object[]`). However, it is not possible to place primitives in such an array - instead we must fill the array with Object-based wrappers (such as Integer and Double). The reflection API will then automatically extract the underlying primitive values from these Objects, which in Java are immutable. Worse still, for our purposes here, the Integer class is declared final from our point of view, which precludes the possibility of defining our own sub-class of the wrapper class i.e. the Java language does not permit a class definition of the form `MyInteger extends Integer`.

### 3.5 Timing Results

In order to judge the overheads involved in running this form of GP, we studied the cost of computing the sum of squares of the set of integers from zero to $n-1$. Each integer in the range is first cast to a double before being being multiplied by itself. This showed that using reflection for a method invocation can be over 100 times slower than making a direct method call. Whether this is an issue depends on the complexity of the method being called, but it is a potential problem with our approach.

## 4 Example Application: Evolutionary Art

Here we choose an application where we already have a number of useful classes defined for the problem domain. The objective is to provide an implementation of an evolvable picture, for use in an evolutionary art application. We've decoupled the picture evolution and drawing from the rest of the program (that deals with the GUI, for example) by defining an interface for an Evolvable Picture. This is shown in Figure 1. This takes just two method - one for producing a mutated child, and the other for drawing itself on a Java Graphics Object.

Figure 2 shows most of our implementation of the Evolvable Picture interface that exploits our OOGP system. There are a few points to note. We use an

```
public interface EvolvablePicture {
    // gets a randomly varied child
    public EvolvablePicture getChild();

    // draw on the graphics object
    public void draw(Graphics g, Dimension d);
}
```

**Fig. 1.** *The interface for an Evolvable Picture.*

`ObjectHolder` to hold a reference to the current graphics object, and also to specify the type (class) of object that it will hold. Then, each call to `draw` updates the object to be the current Graphics object.

The implementation of the `getChild` method is very simple - and mainly involves calling the two-argument constructor with a mutated version of the program. The other main part of our `EvolvableObjectPainter` is the default constructor. Here we simply set up the `ArrayList`s of active objects and of argument objects. For the argument objects, we put a handful of Double and Integer numbers in there - we've omitted these for space reasons.

Our evolvable art GUI allows picture evolution by having the user click his preferred image to breed from by mutation at each generation. Using this with the `EvolvableObjectPainter` implementation described above successfully exploited the Java Graphics API, and Figure 4 shows the twelve pictures displayed on generation 30 of one of the runs.

Note that the type-safe Instruction Generation described above is very important in cutting the size of the search space. For example, given a Graphics2D object, there are 92 public methods that can be called, many of which take several arguments. Given our set of constants (about 10) that we set up in the constructor, this would lead to a huge set of possible `Instructions`, many of which would lead to invocation exceptions without proper type checking in advance. A fragment of an evolved program is shown in Figure 3. Note that all the method calls shown were generated automatically using reflection.

## 5 Conclusions

The set of class libraries or APIs that are associated with a language such as Java, both those supplied as part of the language platform, and additions provided by third parties, are the end result of a long software design process. Together, they are based on hundreds of expert man years of effort, and provide a vast source of knowledge useful in the construction of real-world problems. They distill much of what the computer science community knows about writing software for practical applications.

The main contribution of this paper has been to demonstrate how it is possible to tap into this knowledge source, and evolve interesting programs that are

```
public class EvolvableObjectPainter implements EvolvablePicture {
    Program p;
    ObjectHolder graphics;
    static int len = 20; // the starting length for progs

    public EvolvableObjectPainter() {
        ArrayList active = new ArrayList();
        graphics = new ObjectHolder( Graphics2D.class );
        active.add( graphics );
        ArrayList args = new ArrayList();
        // ... add some numbers to args...
        args.add( new Integer(1) );
        // .. lines omitted

        InstructionGenerator gen =
           new InstructionGenerator(active, args);
        p = new Program( gen, len, null );
    }
        // ... two-arg constructor used below omitted
    public EvolvablePicture getChild() {
        System.out.println("Chose: " + p);
        return new EvolvableObjectPainter(p.mutatedCopy(), graphics);
    }

    public void draw(Graphics g, Dimension d) {
        graphics.setObject( g );
        p.execute();
    }
}
```

**Fig. 2.** *OOGP Implementation of the Evolvable Picture interface.*

```
java.awt.Graphics.fillArc(int,int,int,int,int,int) :
  : 100 : 10 : 100 : 1 : 100 : 100
java.awt.Graphics.fillArc(int,int,int,int,int,int) :
  : 100 : 100 : 100 : 10 : 10 : 100
java.awt.Graphics.drawRect(int,int,int,int) :
  : 100 : 100 : 100 : 25
java.awt.Graphics.setColor(java.awt.Color) :
  : java.awt.Color[r=64,g=93,b=251]
java.awt.Graphics.drawOval(int,int,int,int) :
  : 100 : 25 : 1 : 25
java.awt.Graphics2D.translate(double,double) :
  : -0.2617993877991494 : 0.5235987755982988
 ...
```

**Fig. 3.** *Sample evolved picture program.*

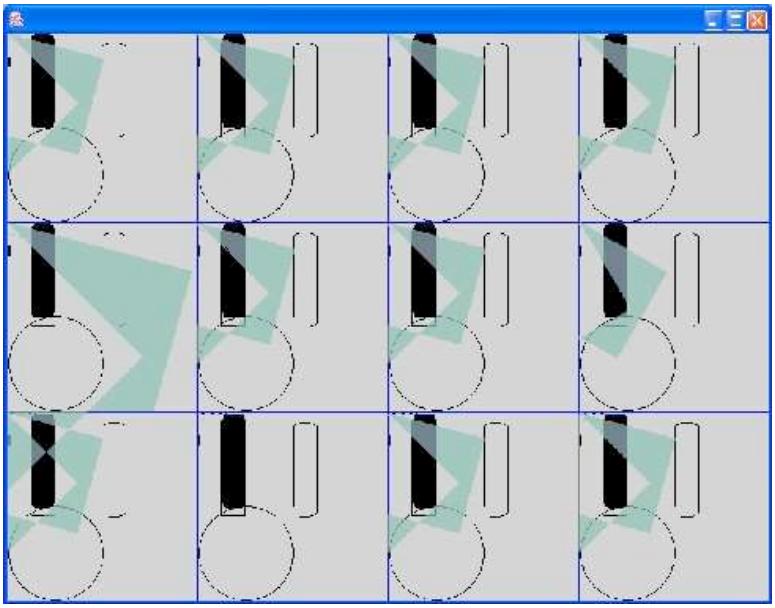

**Fig. 4.** Running the Evolvable Picture application. All the drawing methods used were picked out by the evolutionary algorithm with the aid of reflection on a Java Graphics2D object.

able to exploit these class libraries given only the smallest and most easily provided of clues, such as a method signature and a set of objects to use as actors and parameters. There is a huge potential for future exploitation.

An important challenge is to evolve programs that are not only simple and efficient, but perhaps well designed in terms of developing and deploying re-usable class libraries. What sort of environment do we need in order for something like the Java *Collections* API to evolve?


### Acknowledgements

The author(s) would like to thank the members of the X group at the University of Y for useful discussions of this work.


## References


1. Nichael Lynn Cramer, "A representation for the adaptive generation of simple sequential programs", in *Proceedings of an International Conference on Genetic Algorithms and the Applications*, John J. Grefenstette, Ed., Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985, pp. 183–187.
2. J.R. Koza, *Genetic Programming: on the programming of computers by means of natural selection*, MIT Press, Cambridge, MA, (1992).

3. J.R. Koza, *Genetic ProgrammingII: automatic discovery of reusable programs*, MIT Press, Cambridge, MA, (1994).
4. Astro Teller, "Turing completeness in the language of genetic programming with indexed memory", in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Orlando, Florida, USA, 27-29 June 1994, vol. 1, pp. 136–141, IEEE Press.
5. D.J. Montana, "Strongly typed genetic programming", *Evolutionary Computation*, vol. 3, pp. 199 – 230, (1995).
6. Markus Brameier and Wolfgang Banzhaf, "Effective linear genetic programming", Interner Bericht des Sonderforschungsbereichs 531 *Computational Intelligence* CI–108/01, Universität Dortmund, April 2001.
7. Markus Brameier and Wolfgang Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining", *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 1, pp. 17–26, 2001.
8. Julian F. Miller and Peter Thomson, "Cartesian genetic programming", in *Genetic Programming, Proceedings of EuroGP'2000*, Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, Eds., Edinburgh, 15-16 April 2000, vol. 1802 of *LNCS*, pp. 121–132, Springer-Verlag.
9. Riccardo Poli, "Evolution of graph-like programs with parallel distributed genetic programming", in *Genetic Algorithms: Proceedings of the Seventh International Conference*, Thomas Back, Ed., Michigan State University, East Lansing, MI, USA, 19-23 1997, pp. 346–353, Morgan Kaufmann.
10. Wolfgang Kantschik and Wolfgang Banzhaf, "Linear-graph GP—A new GP structure", in *Proceedings of the 4th European Conference on Genetic Programming, EuroGP 2002*, Evelyne Lutton, James A. Foster, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, Eds., Kinsale, Ireland, 3-5 2002, vol. 2278, pp. 83–92, Springer-Verlag.
11. M O'Neill and C Ryan, "Grammatical evolution", *IEEE Transactions on Evolutionary Computation*, vol. 5, pp. 349 – 358, (2001).
12. Wilker Shane Bruce, *The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs*, PhD thesis, School of Computer and Information Sciences, Nova Southeastern University, 3100 SW 9th Avenue, Fort Lauderdale, Florida 33315, USA, December 1995.
13. Wilker Shane Bruce, "Automatic generation of object-oriented programs using genetic programming", in *Genetic Programming 1996: Proceedings of the First Annual Conference*, John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, Eds., Stanford University, CA, USA, 1996, pp. 267–272, MIT Press.
14. William B. Langdon, "Evolving data structures with genetic programming", in *Proc. of the Sixth Int. Conf. on Genetic Algorithms*, Larry J. Eshelman, Ed., San Francisco, CA, 1995, pp. 295–302, Morgan Kaufmann.
15. Russ Abbott, "Object-oriented genetic programming: An initial implementation", in *International Conference on Machine Learning: Models, Technologies and Applications*, 2003, pp. 24 – 27.
16. John R. Koza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, 2003.
17. M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.
18. Sean Luke, "A Java-based Evolutionary Computation and Genetic Programming Research System", `http://www.cs.umd.edu/projects/plus/ec/ecj/`.