

Object-Oriented Genetic Programming

An Initial Implementation

Russ Abbott (abbott@aero.org)

The Aerospace Corp, P.O. Box 92957, Los Angeles, Ca. 90009

Abstract. This paper describes **oogp**, an **object-oriented genetic programming** system. Oogp provides traditional genetic programming capabilities in an object-oriented framework. Among the advantages of object-oriented genetic programming are: (a) strong typing, (b) availability of existing class libraries for inclusion in generated programs, and (c) straightforward extensibility to include features such as iteration as object-oriented methods. **Oogp** is written in Java and makes extensive use of Java's reflection capabilities. **Oogp** includes a relatively straightforward but apparently innovative simplification capability. **Keywords:** object-oriented, genetic programming

1. Background

Genetic programming (GP) (Koza 1990) is the use of a genetic algorithm (Holland 1975) to generate computer programs.

Koza's original genetic programming system generated Lisp-like program-expressions. A typical GP program is a deeply nested expression. To avoid type incompatibilities, functions were required to operate on data of any type that might appear in that run.

Montana (Montana 1995) extended Koza's formulation to include strong typing. In a strongly typed genetic programming system, program generation and evolution are constrained so that functions are nested in such a way that only results of appropriate types are passed as arguments.

Here we describe the initial version of an object-oriented GP system.

2. Oogp language basics

Oogp generates Java-like programs in a Java environment (Sun reference 1). The fundamental units of execution are method calls to Java objects. Oogp uses Java's `java.lang.reflect` (Sun reference 2) capability to generate, manipulate, and execute the programs that it generates.

Wrapper classes

Java allows two kinds of data: primitives (such as `int` and `boolean`) and objects. The `java.lang.reflect.Method.invoke()` method accepts arguments as an array of objects and returns an object result. To accommodate primitives, which cannot be passed as objects, `invoke()` accepts

wrapped primitives (e.g., `int's` as `Integer's` and `boolean's` as `Boolean's`) and returns wrapped results. Although this greatly simplifies the use of `invoke()`, it creates another problem. Java wrapper classes define only immutable objects.

Oogp gets around the problem of immutable vs. primitive data by defining mutable wrapper classes. The oogp `Int` class is similar to the Java `Integer`, and the oogp `Bool` class is similar to the Java `Boolean` except that they define mutable objects. When an oogp-generated program is executed, a double wrapping and unwrapping occurs. `Int's` and `Bool's` are unwrapped to `Integer's` and `Boolean's` before `invoke()` is called. The `invoke()` method then does a second unwrapping to pass primitives to the underlying method. Returned values are doubly wrapped on the way back. Within oogp, all computation on integers and booleans is done on `Int's` and `Bool's`.

A simple illustrative example

Much of the initial development work on oogp was done with the even-parity problem in mind. Prior to attacking that problem, we generated a solution to the even simpler even-odd problem: given an integer, return `true` or `false` depending on whether the integer is even or odd.

Since `Int` has both a `mod()` method and an `isZero()` method, this is a fairly easy problem. The following is the simplest solution.

```

Bool fun_0(Int int_0) {
    int_0.mod(2);
    int_0.isZero();
}

```

In most runs, the preceding was generally the first solution generated. (Oogp can be asked to generate multiple solutions.)

The above illustrates the sort of programs generated by oogp. In particular:

- Although expressed as functions oogp-generated programs are methods to be applied to their first arguments.
- Oogp programs consist of a return type, arguments, and a body, which is a program statement. The program returns as its value the value returned by its body.
- Data are actual Java objects. Variables are references to objects.
- Method calls within oogp programs are Java methods executed by `java.lang.reflect.Method.invoke()`. They operate on Java objects, possibly changing them. In the above example, `mod()` changes the `int_0` object from its original value to either 0 or 1.
- All methods return values. The `mod()` method returns the object on which it operates, which has been changed to store the mod value. (In this case, the returned value is ignored in the program.) The `isZero()` method returns a (newly created) `Bool` object. Unlike `mod()`, it does not modify the object upon which it is called.
- All program statements return values. The Block statement returns the value of its last element, in this case, `int_0.isZero()`.
- Oogp includes the standard statements and control structures: Assignment-Stmt, Block-Stmt, If-Stmt, and While-Stmt.
- Block statements may include local variables. The following oogp-generated solution to the even-odd problem illustrates some of these statement types.

```

Bool fun_1(Int int_0) {
    Int int_1;
    int_1 = int_0.mod(2);
    if (int_1.isNotZero())
        int_1.isNotEqualTo(int_0);
    else true;
}

```

In this example, after the first statement is executed, `int_0` and `int_1` refer to the same object, which has been changed to hold the value 0 or 1 depending upon whether it was originally even or odd. Then the `if` statement returns `true` if `int_1` is zero, i.e., if `int_1.isNotZero()` is false, and `false` otherwise: `int_1` is the same as (hence equal to) `int_0`.

- Because every statement returns a value, If-Statements always have two branches. As in the example above, one of the branches may be a value rather than a method call.
- Method calls do not cascade. (This should change in future versions.)
- There are no arithmetic or boolean operators. All such operations are performed with method calls. `Int`, for example, has `plus()`, `minus()`, `divideBy()`, and `times()` methods. `Bool` has `not()`, `and()`, and `or()` methods. In all these cases, the object upon which the method is called is changed. These methods also return that object as their value.
- Arguments to method calls are variables or constants; method calls may not be nested. (This should change in future versions.)

3. Evolutionary mechanisms

Oogp uses an evolutionary mechanism similar to traditional GP. Programs are represented as parse trees along with associated symbol tables, which keep track of variables.

As in standard GP, crossover occurs through the substitution of one subtree for another. As in strongly typed GP, the subtree to be implanted is required to return an object of a type compatible with the context in which it will be placed. The crossover operation produces one offspring.

Mutation is similar to crossover except that a random subtree is generated instead of being extracted from an element in the population.

Oogp uses a steady-state (rather than generational) population discipline. Whenever a new child is generated, it replaces an existing population element.

Tournament selection, with a tournament size of 2, is used to select both elements to reproduce (the element with the higher fitness value is selected) and elements to be replaced (the element with the lower fitness value is selected).

When generating a random program statement, the statement is constrained to return an object that satisfies two `java.Class.isAssignableFrom()` constraints.

```
maxType.isAssignableFrom(stmtType)
    &&
stmtType.isAssignableFrom(minType)
```

The most general constraint would have `maxType` be `java.lang.Object` and `minType` be `void`, the class `reflect` uses as the `returnType()` of `void` methods. (This is treated as a special case since no class is considered assignable from `void`.)

Local variables

Whenever a statement list is generated, a local symbol table is created along with it. The system then allows itself to generate new local variables for use on the left-hand-side of assignment statements and as arguments to method calls.

Exploring the Class library

Initially, `oogp` generates objects from the classes given to it as either the types of the input arguments or the return type of the function it is generating.

However, methods in these classes may require arguments or produce values from classes `oogp` has not yet seen. These newly seen classes are added to the list of classes that used in generating programs. Through this mechanism, `oogp` is capable of exploring the reachable class library.

The ability to explore the class library has costs and benefits. An obvious benefit is that the system can include in its solutions methods from classes it finds. The cost is that exploring the class library may distract it from its other evolutionary tasks. To prevent the latter, `oogp` may be told to limit its use of classes to a specific list.

Automatic simplification

`Oogp` includes an automatic simplification capability. The simplification algorithm

deletes random elements from a generated program. Only subtrees whose removal would leave the reduced function both syntactically correct and type compatible with the original are selected for deletion. Whenever a new-best element is found, it is simplified until any further deletion would reduce the fitness value. All elements in the generated simplification chain are included in the population. Although no attempt is made to reason about programs, and some simplifications may be missed, this approach has been remarkably successful.

4. Function execution

Once generated, programs are executed in the usual way. A limit may be placed on the number of steps allowed. When exceeded, an `ExcessiveStepsException` is thrown. If *any* exception occurs during execution, execution terminates and the program being executed is given a negative fitness value.

5. Iteration

To investigate higher-order problems, we created an `abstract List Class`, similar to `ArrayList` but having elements of uniform type. Adapting Kirshenbaum (Kirshenbaum 2001) to an object-oriented framework, we built an `iterate()` method of the following form into that class.

```
Object iterate(<constant>, <fun>)
```

The first argument is a simple value. The second is a function of the form:

```
<type> fun(Object currentElement,
    Object previousResult) {
    <Compute and return a new
    result based on the
    previousResult and the
    currentElement.>
}
```

The first argument is used as the initial result when applying the function argument to the first element of the list.

Since `List` is abstract, subclasses are defined for specific element types. We used both `IntList` and `BoolList` subclasses. Here is the (obvious) solution `oogp` generated to the problem of finding the sum of a list. The input is an `IntList` object, a list whose elements are all `Int`'s.

```

Int fun_0(IntList intlist_0) {
    intlist_0
    .iterate(0,
        fun_0(Int element;
            Int result;
            result
            .plus(element)););
}

```

This function starts with 0 and adds the elements of the list to it.

Implementing `iterate()`

Since `iterate()` is just another method in a class, one would like not to have to build anything special into oogp to use it. This goal has not been achieved.

The `iterate()` method is defined to return an `Object` result. Yet in problems such as sum-a-list the desired return type is much more restricted. In sum-a-list, one wants `iterate()` to return an `Int`.

To get around this problem, a special case is built into oogp to allow `iterate()` to be embedded in any context but to require that the function passed as its second argument to return the type required by that context.

Even parity

A standard GP problem is the even parity problem: given a set of Boolean values, return `true` or `false` depending on whether the set has an even number of `true` values. Even-parity is relatively easy for a gp system (Kirshenbaum 2001) when an `iterate()` method is available. Oogp was able to find the following solution fairly quickly.

```

fun_0(BoolList boollist_0) {
    boollist_0
    .iterate(true,
        fun_0(Bool element;
            Bool result;
            if (element)
                result.not();
            else result;));
}

```

6. Conclusions and comments

Oogp is an object-oriented approach to genetic programming that makes existing program libraries available for use to a genetic programming mechanism. Additional work is needed on the allowed

syntax of the language (cascading method calls and method calls as arguments), and more work is needed on integrating meta-function methods like `iterate()`.

On a more far-reaching scale, much work needs to be done to allow oogp to define new classes. It seems unlikely that a general Class definition capability will be helpful. Just as a general loop generation capability seems too general to use, a general Class definition capability is unlikely to facilitate the evolution of useful classes. However, just as the inclusion of a more targeted iteration capability allows a GP system to generate useful programs more easily, it would be worth exploring the encoding of design patterns (Gamma 1994) into an object-oriented GP system. If a GP system were constrained to create classes within the framework of well-known design patterns, it is possible that new classes could be generated in a useful manner.

References

- Gamma E., et al, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- Holland J.H., *Adaptation in natural and artificial system*, Ann Arbor, The University of Michigan Press, 1975
- Kirshenbaum, Evan, *Iteration Over Vectors in Genetic Programming*. HP Laboratories Technical Report HPL-2001-327, December 17, 2001.
(<http://www.kirshenbaum.net/evan/publications/GECCO-2001-tr.pdf>)
- Koza, John R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems* Stanford University Computer Science Department Technical Report STAN-CS-90-1314, June 1990.
- Koza, John R.; Bennett, Forrest H, III; Andre, David; and Keane, Martin A. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann, 1999.
- Montana, David J. "Strongly typed genetic programming." *Evolutionary Computation* 3(2):199–230, 1995.
- Sun Microsystems. <http://java.sun.com/>.
- Sun Microsystems. <http://java.sun.com/j2se/1.4/docs/api/java/lang/reflect/package-frame.html>.