Grammar based function definition in Grammatical Evolution.

Michael O' Neill & Conor Ryan
Dept. of Computer Science And Information Systems
University of Limerick
Ireland
Michael.ONeill@ul.ie & Conor.Ryan@ul.ie
Phone: +353-61-202730

Abstract

We describe the use of grammars as an approach to automatic function definition in Grammatical Evolution. The automatic generation of functions allows the evolution of both the function and the code belonging to the main body of the program which has the ability to call upon the evolving function. As proof of concept we apply a grammar using automatic function definition to the Santa Fe ant trail. Results show that the evolutionary search successfully evolves code for the automatically defined function and calls upon this function from the main code to generate solutions to the Santa Fe trail. An analysis of the number of successful runs shows a speed-up in terms of the number of generations required to find a solution when compared to standard Grammatical Evolution.

1 Introduction

Grammatical Evolution (GE) is an evolutionary algorithm approach to automatic program generation in which a population is comprised of individuals composed of variable length binary strings. The system employs a genotype-to-phenotype mapping process in order to generate the output programs. The mapping process employs a Backus Naur Form (BNF) grammar definition that permits the system to evolve code in any language, and ensures its syntactic correctness.

GE has been applied successfully to various problem domains including, symbolic regression, finding trigonometric identities, robot control, and evolving caching algorithms[Ryan et al. 98a] [Ryan et al. 98b] [Ryan, O'Neill 98] [O'Neill, Ryan 99a] [O'Neill, Ryan 99b] [O'Neill, Ryan 99e] [O'Neill et.al. 2000]. Some analysis of the systems features has been conducted in [O'Neill, Ryan 99c] [O'Neill, Ryan 99d] [O'Neill, Ryan 2000].

We now demonstrate the use of the BNF grammar as an approach to automatic function definition. The evolution of functions has been examined by various researchers in the Genetic Programming community including [Angeline, Pollack 92] [Kinnear 94] [Rosca 95] [Koza 92] [Koza 94], and have grown in sophistication since the original ADF format adopted in [Koza 92].

Before describing our grammar based function definition approach we firstly give a brief overview of GE, and the problem approached.

2 Grammatical Evolution

When tackling any problem with GE a suitable BNF definition must first be decided upon. The BNF can be either the specification of an entire language, or perhaps more usefully a subset of a language geared towards the problem at hand. For example, a BNF for the Santa Fe Ant Trail could be;

```
N = {<code>,<line>,<condition>,<op>}

T = {left(),right(),move(),food_ahead(),
      else,if,{,},(,)}

S = <code>
```

And P can be represented as:

```
(1) <code> :: = <line>                (A)
              | <code><line>          (B)

(2) <line> :: = <condition>           (A)
              | <op>                  (B)

(3) <condition> :: = if(food_ahead()){
                        <line>
                     }
                     else{
                        <line>
                     }
```

```
(4) <op> :: =    left();              (A)
               | right();             (B)
               | move();              (C)
```

where the operations `left()`, `right()`, `move()`, and `food_ahead()`, are all functions written in the C programming language, and `N` is the set of non-terminals, `T`, the set of terminals, `P`, a set of production rules that map the elements of `N` to `T`, and `S` is a start symbol which is a member of `N`.

The genotype is then used to map the start symbol onto terminals by reading codons of 8 bits to generate a corresponding integer value, from which an appropriate production rule is selected. A rule is selected by using the following, (Integer Gene Value) MOD (Number of Production Rules for the current non-terminal). Considering rule #4 from above, i.e. given the non-terminal `<op>` there are three production rules to select from. If we assume the codon being read produces the integer 6, then $6\ MOD\ 3\ =\ 0$ would select rule `(A) left()`. Each time a production rule has to be selected to map from a non-terminal, another codon is read, and in this way, the system traverses the genome.

During the genotype to phenotype mapping process it is possible for individuals to run out of codons, and in this case we wrap the individual, and reuse the codons. This is quite an unusual approach in EA's, as it is entirely possible for certain codons to be used two or more times. The technique of wrapping the individual draws inspiration from the gene overlapping phenomenon which has been observed in many organisms in nature [Elseth 95]. In GE, each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal, may have a different effect, that is, it may select a different production rule. What is crucial, however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is because the same choices are made each time. It is possible that an incomplete mapping could occur, even after wrapping, and in this event the individual in question is given the lowest fitness value possible, then the selection and replacement mechanisms should operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules over and over. For example, given an individual with three codons, all of which specify rule 0 from below,

```
(1) <expr> :: = <expr><op><expr>      (0)
              | (<expr><op><expr>)    (1)
              | <preop>(<expr>)       (2)
              | <var>                 (3)
```

even after wrapping the mapping process would be incomplete and would carry on indefinitely unless stopped. This occurs because the nonterminal `<expr>` is being mapped indefinitely by production rule 0, i.e., it becomes `<expr><op><expr>`. Therefore, the leftmost `<expr>` after each application of a production would itself be mapped to a `<expr><op><expr>`, resulting in an expression continually growing as follows:
`<expr><op><expr><op><expr><op><expr>` etc. Such an individual is dubbed "invalid" as it will never undergo a complete mapping to a set of terminals.

To reduce the number of invalid individuals being passed from generation to generation a steady state replacement mechanism is employed. A consequence of the steady state method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals.

Results presented in [O'Neill, Ryan 99d] show that the degenerate genetic code employed in GE also plays a role in preserving individual validity. With such a representation scheme, different codes can represent the same symbol, or in GE, the same production rule. Modifications are thus allowed to occur to the genotype without affecting the phenotype, or functionality. Mutation events which do not alter the phenotypic fitness are termed "neutral mutations" [Kimura 83] [Banzhaf 94]. A degenerate coding mechanism is also employed in nature and a comparison between GE's genetic code and that of molecular biology can be seen in Figure 1.

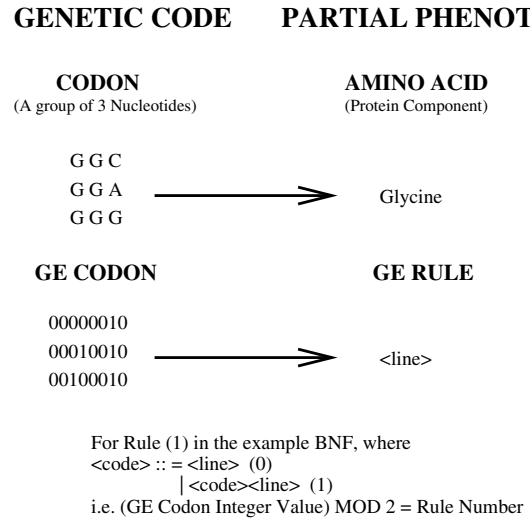

Figure 1: A comparison between genetic code degeneracy in molecular biology and Grammatical Evolution, note how a single point mutation at the third base in the biological genetic code will always result in the selection of the amino acid Glycine. Amino acids are the components of proteins which are responsible for phenotypic traits. In Grammatical Evolution we give an example of how a single bit mutation in a codon can result in the application of the same production rule $< line >$. The production rules in Grammatical Evolution are combined in such a way as to produce a syntactically correct program in the language specified by the grammar.

An overview of how GE operates in comparison with a biological genetic system can be seen in Figure 2. It can be seen

that genes are expressed as proteins which produce a phenotype. More than one protein may be responsible for producing any one phenotype, and a similar situation occurs in GE. Here a codon results in the selection of rules which map non-terminals onto either more non-terminals, or terminals, it is a combination of these terminals and non-terminals that produce the phenotype, or program.

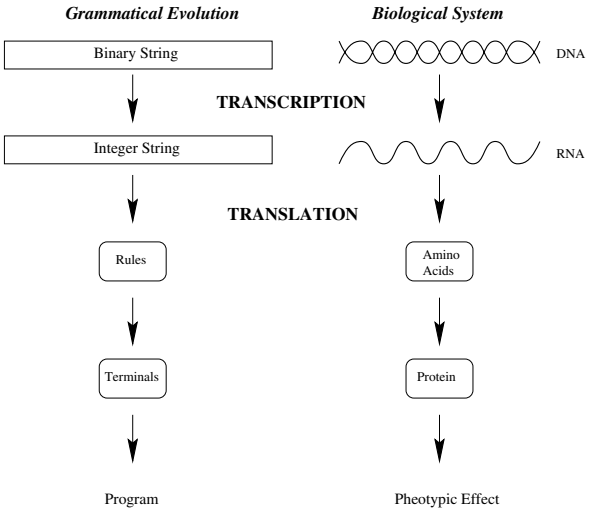

Figure 2: A comparison between Grammatical Evolution and molecular biology which shows the mapping process from genotype to phenotype.

3 The Problem Space

Experiments were conducted using the Santa Fe ant trail as described in [Koza 92]. A brief description of this problem domain follows.

The Santa Fe ant Trail is a common problem tackled in the area of Genetic Programming, and can be considered a deceptive planning problem with many local and global optima [Langdon, Poli 98]. GE has been previously shown to outperform GP on this problem [O'Neill, Ryan 99a]. The objective is to find a computer program to control an artificial ant so that it can find all 89 pieces of food located on a non-continuous trail within a specified number of time steps. The trail is located on a 32 by 32 square grid. The code evolved is then executed in a loop until the number of time steps allowed has elapsed. The ant can only turn left, right, move forward one square, and may also look ahead one square in the direction it's facing to determine if that square contains a piece of food. The actions left, right, and move each take one time step to execute.

While there are many possible fitness cases to the Santa Fe trail only one case was taken for the purposes of this experiment. The ant started in the top left hand corner of the grid facing the first piece of food on the trail. A summary of the problem can be seen in Table 1.

Objective :	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe trail.
Terminal Operands:	left(), right(), move()
Terminal Operators	food_ahead()
Fitness cases	One fitness case
Raw Fitness	Number of pieces of food before the ant times out with 615 operations.
Standardised Fitness	Total number of pieces of food less the raw fitness.
Hits	Same as raw fitness.
Wrapper	Standard productions to generate C functions
Parameters	Population = 500 Generations = 20

Table 1: Grammatical Evolution Tableau for the Santa Fe Trail

4 Grammar Based Function Definition

The grammar that has been used previously on this problem is presented in Section 2. Below is the grammar we used for automatic function definition.

```
(1) <prog> ::= void evolved(){
                    <code>
               }
               void adf0(){
                    <adfcode>
               }

(2) <code> ::= <line>                (A)
             | <code><line>          (B)

(3) <line> ::= <condition>           (A)
             | <op>                  (B)

(4) <condition> ::= if(food_ahead()){
                        <line>
                    }
                    else{
                        <line>
                    }

(5) <op> ::= left();                 (A)
           | right();                (B)
           | move();                 (C)
           | adf0();                 (D)

(6) <adfcode> ::= <adfline>                (A)
                | <adfcode> <adfline>      (B)
```

```
(7) <adfline> :: = <adfcondition>    (A)
                 | <adfop>           (B)

(8) <adfcondition> :: = if(food_ahead()){
                           <adfline>
                        }
                        else {
                            <adfline>
                        }

(9) <adfop> :: = left();     (A)
               | right();    (B)
               | move();     (C)
```

As can be seen from above, the grammar for automatic function definition would appear to be larger and more complex than the standard grammar used on this problem. However, appearances are deceptive in this case, as in fact the only difference is that the new grammar allows the main function to call upon the automatically defined function, whereas the automatically defined function cannot call upon itself.

The first rule `prog` determines that the evolved code will be comprised of a main function called *evolved()* which is called in a loop until the predetermined number of time steps has elapsed. This is the function into which the code evolved by the grammar in Section 2 is placed. The second function in the rule `prog` is the automatically defined function *adf0()*. The body for the main function and that of *adf0()* is described by two separate non-terminals. The differentiation is made to avoid recursive function calls by *adf0()*. Whereas the main function can call upon the terminals operations `left()`, `right()`, `move()`, and `adf0()`, the automatically defined function can only call `left()`, `right()`, and `move()`.

The approach used here is similar to the one adopted in [Koza 92] by predetermining the number of automatically defined functions and their parameters. We also take this approach as we wish to demonstrate that GE can benefit from grammar based function definition. A brief discussion on extensions to grammar based function definition can be found in section 6.

5 Results

The grammar described in Section 3 was used to successfully evolve solutions to the Santa Fe trail. An example solution produced is given below.

```
void  evolved(){
 if(food_ahead()){
   move();
 }
 else{
   adf0();
 }
 if(food_ahead()){
    move();
 }
 else{
    adf0();
 }
 move();
 right();
 if(food_ahead()){
    adf0();
 }
 else{
    right();
 }
}

void  adf0(){
 if(food_ahead()){
    left();
 }
 else{
    right();
 }
}
```

Note how the main function efficiently utilises the automatically defined function to call upon its generalised behaviour of turning either left or right depending on the presence or absence of food in the neighbouring grid.

A preliminary examination has revealed that in the case of the Santa Fe trail there is an improvement in performance when using the grammar based function definition. Figure 3 shows the number of successful runs out of 20 over 20 generations in the case of both grammars presented in this paper. Here we can see that in the presence of the grammar defined function definition, runs are successful faster in terms of the number of generations elapsed and frequency of successful runs is increased.

6 Discussion

The results presented show that in the presence of grammar based function definition we get better solutions faster, than in their absence, and the output code is in a more readable format. The use of automatically defined functions has the potential to yield generalisation properties as has been demonstrated with Genetic Programming, by their incorporation into automatically defined functions. For further discussion on the role of automatically defined functions in genetic programming please refer to [Koza 94] as a starting point.

In order to prevent the grammar defined function (*adf0*) from calling itself we specified a new non-terminal `<adfcode>` that differentiates the content of the function from the main

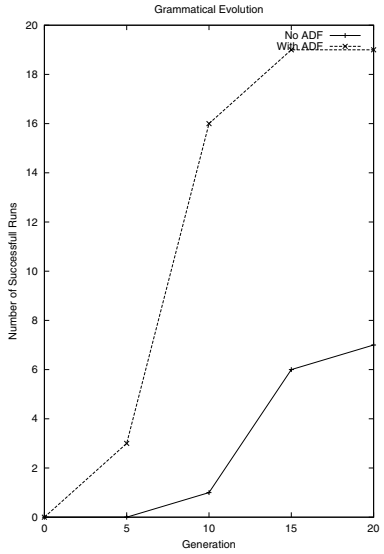

Figure 3: A comparison between Grammatical Evolutions performance in the case when grammar based function definition is employed and when it is absent. In the presence of grammar based function definition there are a greater number of successful runs in the intitial generations, and more successful runs in total than in the case when they are absent.

code. The distinction meant that there was a repetition of similar rules from the main function in the rules for <adfcode>. Repetition of these rules could have been avoided had an attribute grammar being employed. Attribute grammars are context sensitive and would allow us to specify if the current non-terminal was part of the grammar defined function or belonging to the main calling code. Future work will involve adapting the mapping process of GE to allow the use of attribute grammars.

In Section 3 we described the grammar for automatic function definition. Notice that there are no parameters or return statements, however, with a simple modification to the grammar these can be incorporated with ease. For example, in order to pass a float to the automatically defined function the rules for <prog> and <op> would become:

```
(1) <prog> :: =

        void evolved(){
             <code>
        }
        void adf0(float floatvar){
             <adfcode>
        }

(5) <op> :: = left();              (A)
            | right();             (B)
            | move();              (C)
            | adf0(floatvar);      (D)
```

Similarly we could incorporate additional automatically defined functions by simple modifications to the grammar. In the case that more than one automatically defined function existed in the grammar we could impose a hierarchical approach to their invocation similar to that employed in [Koza 94]. In order to allow the dynamic definition of functions with GE, one approach would be to modify the grammar on the fly to incorporate any new function definitions that are generated during the mapping process, the initial grammar having been given this ability. The above approach will, however, require further investigation.

7 Conclusions

We have successfully demonstrated the use of grammar based function definition in Grammatical Evolution by evolving solutions to the Santa Fe ant trail. Results also indicate an improvement in performance for Grammatical Evolution when grammar based function definition is employed on the problem domain examined. Future work will involve an investigation into dynamic grammar based function definition and the use of attribute grammars in GE.

References

[Angeline, Pollack 92] Angeline P.J., Pollack J.B. The Evolutionary Induction of Subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc, pp 236-241.

[Banzhaf 94] Banzhaf, W. 1994. Genotype-Phenotype Mapping and Neutral Variation - A case study in Genetic Programming. In *Parallel Problem Solving from Nature III*. Springer.

[Elseth 95] Elseth Gerald D., Baumgardner Kandy D. 1995. Principles of Modern Genetics. *West Publishing Company*

[Kimura 83] Kimura, M. 1983. The Neutral Theory of Molecular Evolution. Cambridge University Press.

[Kinnear 94] Kinnear K. Alternatives in Automatic Function Definition: A Comparison of Performance. In K. Kinnear, ed., *Advances in Genetic Programming*, MIT Press, 1994, pp 119-141.

[Koza 92] Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

[Koza 94] Koza, J. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

[Langdon 98] Langdon, W. 1998. Genetic Programming and Data Structures. Kluwer Academic Publishers.

[Langdon, Poli 98] Langdon, W. & Poli, R. Why Ant's Are Hard. In *Proceedings of Genetic Programming 1998*, pages 193-201 .

[O'Neill et.al. 2000] O'Neill M., Collins J.J., and Ryan C. Automatic Generation of Robot Behaviours using Grammatical Evolution. In *Proceedings of AROB 2000: The Fifth International Symposium on Artificial Life and Robotics*, Oita, Japan.

[O'Neill, Ryan 2000] O'Neill M., Ryan C. Crossover in Grammatical Evolution: A Smooth Operator? In *Proceedings of the Third European Workshop on Genetic Programming 2000*.

[O'Neill, Ryan 99a] O'Neill M., Ryan C. Evolving Multi-line Compilable C Programs. In *Proceedings of the Second European Workshop on Genetic Programming 1999*.

[O'Neill, Ryan 99b] O'Neill M., Ryan C. Automatic Generation of Caching Algorithms. In *Proceedings of EUROGEN'99: A short course on Evolutionary Computation*, Jyvaskyla, Finland, pages 127-134.

[O'Neill, Ryan 99c] O'Neill M., Ryan C. Under the Hood of Grammatical Evolution. In *Proceedings of GECCO'99: The Genetic and Evolutionary Computation Conference 1999*, Vol. 2, pp. 1143-1148.

[O'Neill, Ryan 99d] O'Neill M., Ryan C. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond, in *ECAL'99: Proceedings of the Fifth European Conference on Artificial Life*, Lausanne, Switzerland.

[O'Neill, Ryan 99e] O'Neill M., Ryan C. Steps Towards Khepera Dance Improvisation. In *Proceedings of the First International Khepera Workshop 1999*.

[Rosca 95] Rosca J.P. An Analysis of Hierarchical Genetic Programming. Technical Report 566, University of Rochester, Computer Science Department, New York.

[Ryan et al. 98a] Ryan C., Collins J.J., O'Neill M. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming*, pages 83-95 . Springer-Verlag.

[Ryan et al. 98b] Ryan C., O'Neill M., Collins J.J. 1998. Grammatical Evolution: Solving Trigonometric Identities. In *Proceedings of Mendel '98: 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks and Rough Sets*, pages 111-119.

[Ryan, O'Neill 98] Ryan C., O'Neill M. Grammatical Evolution: A Steady State Approach. In *Late Breaking Papers, Genetic Programming 1998*, pages 180-185.