Grammatical Evolution : Evolving Programs for an Arbitrary Language

Conor Ryan, JJ Collins & Michael O Neill

Dept. Of Computer Science And Information Systems
University of Limerick
Ireland
{Conor.Ryan|J.J.Collins|Michael.ONeill}@ul.ie

Abstract. We describe a Genetic Algorithm that can evolve complete programs. Using a variable length linear genome to govern how a Backus Naur Form grammar definition is mapped to a program, expressions and programs of arbitrary complexity may be evolved. Other automatic programming methods are described, before our system, *Grammatical Evolution*, is applied to a symbolic regression problem.

1 Introduction

Evolutionary Algorithms have been used with much success for the automatic generation of programs. In particular, Koza's [Koza 92] Genetic Programming has enjoyed considerable popularity and widespread use. Koza's method originally employed Lisp as its target language, and others still generate Lisp code. However, most experimenters generate a homegrown language, peculiar to their particular problem.

Many other approaches to automatic program generation using Evolutionary Algorithms have also used Lisp as their target language. Lisp enjoys much popularity for a number of reasons, not least of which is the property of Lisp of not having a distinction between programs and data. Hence, the structures being evolved can directly be evaluated. Furthermore, with reasonable care, it is possible to design a system such that Lisp programs may be safely crossed over with each other and still remain syntactically correct.

Evolutionary Algorithms have been used to generate other languages, by using a grammar to describe the target language. Researchers such as Whigham [Whigham 95] and Wong and Leung's LOGENPRO system [Wong 95] used context free languages in conjuntion with GP to evolve code. Both systems exploited GP's use of trees to manipulate parse trees, but LOGENPRO did not explicitly maintain parse trees in the population, and so suffered from some ambiguity when trying to generate a tree from a program. Whigham's work did not suffer from this, and had the added advantage of allowing an implementor to bias [Whigham 96] the search towards parts of the grammar.

Another attempt was that of Horner [Horner 96] who introduced a system called Genetic Programming Kernel (GPK), which, similar to standard GP, employs trees to code genes. Each tree is a derivation tree made up from the BNF definition. However, GPK has been criticised [Paterson 97] for the difficulty associated with generating the first generation - considerable effort must be put into ensuring that all the trees represent valid sequences, and that none grow without bounds. GPK has not received widespread usage.

An approach which generates C programs directly, was described by Paterson [Paterson 97]. This method was applied to the area of evolving caching algorithms in C with some success, but with a number of problems, most notably the problem of the chromosomes containing vast amounts of introns. This method will be more fully discussed in section 3.

We describe a different approach to using BNF definitions, and develop a system that evolves individuals containing no introns. This system can be used to evolve programs in any language. We adopt the approach that the genotype must be mapped to the phenotype [Keller 96] [Ryan 97a] [Ryan 97b] [Gruau 94] rather than treating the actual executable code as data. In this respect we diverge from Koza's approach, but with the result that the individuals tend to be much smaller.

2 Backus Naur Form

Backus Naur Form (BNF) is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of `terminals`, which are items that can appear in the language, i.e $+, -$ etc. and `non-terminals`, which can be expanded into one or more terminals and non-terminals. A grammar can be represented by the tuple, $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. For example, below is a possible BNF for a simple expression, where

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, Cos, Tan, Log, +, -, /, *, X, ()\}$$

$$S =< expr >$$

And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>      (A)
             | ( <expr> <op> <expr> ) (B)
             | <pre-op> ( <expr> )    (C)
             | <var>                  (D)

(2) <op> ::= + (A)
           | - (B)
           | / (C)
           | * (D)
```

```
(3) <pre-op> ::= Sin (A)
               | Cos (B)
               | Tan (C)
               | Log (D)

(4) <var> ::= X
```

Unlike a Koza-style approach, there is no distinction made at this stage between what he describes as functions (operators in this sense) and terminals (variables in this example), however, this distinction is more of an implementation detail than a design issue.

Whigham [Whigham 96] also noted the possible confusion with this terminology and used the terms **GPFunctions** and **GPTerminals** for clarity. We will also adopt this approach, and use the term **terminals** with its usual meaning in grammars.

Table 1 summarizes the production rules and the number of choices associated with each. When generating a sentance for a particular language, one must choose carefully which productions are to used, as, depending on the choices made, a sentance may be quite different from the desired one, possibly even of a different length.

Rule no.	Choices
1	4
2	4
3	4
4	1

Table 1. The number of choices available from each production rule.

We propose to use a genetic algorithm to control what choices are made at each juncture, thus allowing the GA to control what production rules are used. In this manner, a GA can be used to generate any manner of code in any language.

3 Genetic Algorithm for Developing Software

Genetic Algorithm for Developing Software (GADS) as described by [Paterson 97] uses fixed length chromosomes which encode which production rules are to be applied. If, when interpreting a gene, it doesn't make syntactic sense, e.g. trying to apply rule 2.A as the first production, or when no operator is available, it is ignored.

If, at the end of the chromosome, there are gaps in the expression, i.e. non-terminals which did not have any terminal chosen for them, a default value is inserted. The default value must be tailored for each production rule. In [Paterson 97] it was suggested that an empty production is the suitable approach, however, this is not always possible. Consider the BNF above, rules 2 and 3 must produce an operator of some description, otherwise the entire expression would be compromised. Thus, an arbitrary decision must be taken about which rule should be used as the default. If, for example, an individual was generated that had the non terminal $< op >$ in it, with all of the genes exhausted, one must decide which of the rules $2A..2D$ should be used as a default rule. Clearly, an unfortunate or misguided choice could harm the evolution of a population.

4 Grammatical Evolution

The GADs approach suffers from a number of drawbacks. In particular, as the number of productions grows, the chance of any particular production being chosen by a gene reduces. Paterson's suggestion to combat this was to simply increase the population size or the genome size. Another suggestion was to duplicate certain production rules in the BNF.
This results in a serious proliferation of introns, consider (using our own notation, not Patersons) the following chromosome using the productions from above.

1D	1A	2B	3B	3A	....	4A

Because of the initial rule that was used, there is only one possible choice rule to apply, and any other rule that occurs in the genome must be ignored. This can lead to a situation where the majority of genome consists of introns.
Instead of coding the transitions, our system codes a set of pseudo random numbers, which are used to decide which choice to take when a non terminal has one or more outcomes. A chromosome consists of a variable number of binary genes, each of which encodes an 8 bit number.

220	203	17	3	109	215	104	30

Table 2. The chromosome of an individual. Each gene represents a random number which can be used in the translation from genotype to phenotype.

Consider rule #1 from the previous example:

```
(1) <expr> ::= <expr> <op> <expr> | ( <expr> <op> <expr> ) |
          <pre-op> ( <expr> ) | <var>
```

In this case, the non-terminal can produce one of four different results, our system takes the next available random number from the chromosome to decide which production to take. Each time a decision has to be made, another pseudo random number is read from the chromosome, and in this way, the system traverses the chromosome.

In natural biology, there is no direct mapping from gene to physical expression [Elseth 95]. When genes are expressed they generate proteins, which, either independantly or, more commonly in conjunction with other proteins, created by other genes, affect physical traits. We treat each transition as a protein, on their own, each transition cannot generate a physical trait. However, when other proteins are present, physical traits can be generated. Moreover, while a particular gene always generates the same protein, the physical results depend on the other proteins that are present immediately before and after.

Consider the individual in Table 2. The fourth gene generates the number 3, which, in our system, is analgous to the *protein* 3. Notice that this will be generated regardless of where the gene appears on the chromosome. However, it may have slightly different effects depending on what other proteins have been generated previously. The following section describes the mapping from genotype to phenotype in detail.

It is possible for individuals to run out of genes, and in this case there are two alternatives. The first is to declare the individual invalid and punish them with a suitably harsh fitness value; the alternative is to wrap the individual, and reuse the genes. This is quite an unusual approach in EAs, as it is entirely possible for certain genes to be used two or more times. Each time the gene is expressed it will always generate the same protein, but depending on the other proteins present, may have a different effect. The latter is the more biologically plausible approach, and often occurs in nature. 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.

To complete the BNF definition for a C function, we need to include the following rules with the earlier definition:

```
<func> ::= <header>

<header> ::= float symb(float X) { <body> }

<body> ::= <declarations><code><return>

<declarations ::= float a;

<code> ::= a = <expr>;
```

```
<return> ::= return (a);
```

Notice that this function is limited to a single line of code. However, this is because of the nature of the problem, the system can easily generate functions which use several lines of code. Specifically, if the rule for `code` was modified to read:

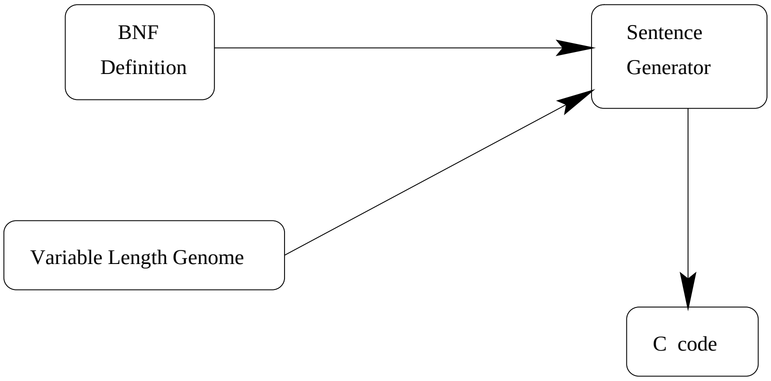

Fig. 1. The Grammatical Evolution System

```
<code> ::= <line>;  | <line>; <code>

<line> ::= <var> = <expr>
```

then the system could generate functions of arbitrary length.

4.1 Example Individual

Consider an individual made up of the following genes (expressed in decimal for clarity) :

220	203	17	3	109	215	104	30

These numbers will be used to look up the table in Section 2 which describes the BNF grammar for this particular problem. The first few rules don't involve any choice, so all individuals are of the form:

```
float symb(float x)
```

```
{
a = <expr>;
return(a);
}
```

Concentrating on the `<expr>` part, we can see that there are four productions to choose from. To make this choice, we read the first gene from the chromosome, and use it to generate a protein in the form of a number. This number will then be used to decide which production rule to use, thus s we have 220 MOD $4 = 0$ which means we must take the first production, namely, 1A. We now have the following

```
<expr> <op> <expr>
```

Notice that if this individual is subsequently wrapped, the first gene will still produce the protein 220. However, depending on previous proteins, we may well be examining the choice of another rule, possibly with a different amount of choices. In this way, although we have the same protein it results in a different physical trait.
Continuing with the first expression, a similar choice must be made, this time using 203 $MOD\ 4 = 3$, so the third choice is used, that is 1C.

```
<var> <op> <expr>
```

There is no choice to be made for `var`, as there is only one possible outcome, so X, is inserted. Notice that no number is read from the genome this time.

```
X <op> <expr>
```

The mapping continues, as summarized in table 3, until eventually, we are left with the following expression:

```
X + Sin ( X )
```

Notice how not all of the genes were required, in this case the extra genes are simply ignored.

5 Operators

Due to the fact that genes are mapped from genotype to phenotype there is no need for problem specific genetic operators, and GE employs all the standard operators of Genetic Algorithms. There are however, two new operators **Prune** and **Duplicate** , which are peculiar to GE.

Expression	Rule	Gene	Number Generated
<expr>	n/a		
<expr><op><expr>	1A	220	0
<var><op><expr>	1D	203	3
X<op><expr>	4	n/a	
X+<expr>	2A	17	1
X+pre_op(<expr>)	2C	19	3
X+Sin(<expr>)	3A	109	1
X+Sin(<var>)	1D	215	3
X+Sin(X)	4	n/a	

Table 3. Mapping a genome onto an expression

5.1 Duplicate

Gene duplication involves making a copy of a gene or genes [Elseth 95], and has been used many times in Evolutionary computation [Schutz 97]. Duplicated genes can lie adjacent to the original gene, or can lie at a completely different region. Repeated gene duplications are presumed to have given rise to the human hemoglobin gene family. The hemoglobin protein is involved with transporting oxygen around the body through the bloodstream. The original hemoglobin gene duplicated to form an α - globin, and β - globin genes. Each of these underwent further duplications resulting at present in four variations on the α - globin gene, and five of the β - globin gene. Different combinations of these genes are expressed during development of the human, resulting in the hemoglobin having different binding properties for oxygen. The presence of a copy of a gene can therefore have benefits. Biologically the results of gene duplications can be to supply genetic material capable of:

(i) Acquiring lethal mutation(s). If two copies of a gene are present in a genome, and one happens to be mutated such that it's protein product is no longer functional having the second copy means that this probability of harmful mutation is reduced.

(ii) Evolving new functions. If a mutation is favourable it may result in an altered protein product, perhaps with new functionality, without changing the function of the original gene's protein product.

(iii) Duplicating a protein in the cell, thus producing more of it. The increased presence of a protein in a cell may have some biological effects.

Goldberg [Goldberg 89] stated that using operators such as duplication in a variable-length genotype enables them to solve problems by combining relatively short, well-tested building blocks to form longer, more complex strings that increasingly cover all features of a problem.

The duplication operator used in GE is a multiple gene duplication operator. The number of genes to be duplicated is picked randomly. The duplicated genes are placed into the position of the last gene on the chromosome, if the (X * X) part of the expression were duplicated, this could result in a useful extension to the genome, especially if a multiplication operator happend to be between the original expressions genes and the duplicated one. Gene duplication in GE is essentially analogous to producing more copies of a gene(s) to increase the presence of a protein(s) in the cell.

5.2 Pruning

Individuals in GE don't necessarily use all their genes. For example, to generate the expression $X * X + X$ only five genes are required, but an individual that generates this expression could well be made up of many more. The remaining genes are introns and serve to protect the crucial genes from being broken up in crossover. While this certainly serves the selfish nature of genes in that it increases the longevity of a particular combination, it is of no advantage to us. In fact, it would make sense for an individual to be as long as possible, to prevent its combination from being disrupted. The fact that such disruption could be crucial to the evolution of the population as a whole is of consequence to the individual. Figure 2 illustrates the effect of too many introns.

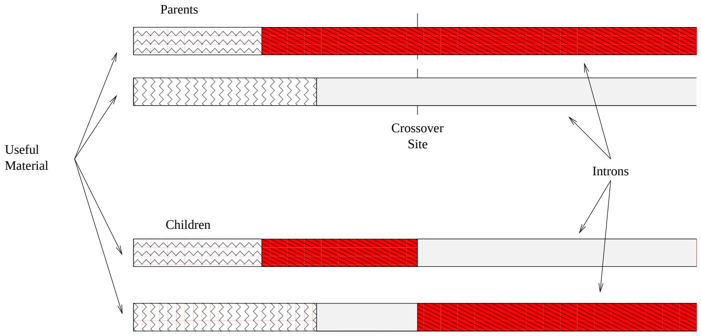

Fig. 2. Crossover being hampered by introns

To reduce the number of introns, and thus increase the likelihood of beneficial crossovers, we introduce the **prune** operator. **Prune** is applied with a probability to any individuals that don't express all of their genes.. Any genes not used in the genotype to phenotype mapping process are discarded.

The effects of pruning are dramatic; FASTER; BETTER CROSSOVER.

6 The Problem Space

As proof of concept, we have applied our system to a symbolic regression problem used by Koza [Koza 92]. The system is given a set of input and output pairs, and must determine the function that maps one onto the other. The particular function Koza examined is $X^4+X^3+X^2+X$ with the input values in the range $[-1..+1]$. Table 4 contains a tableau which summarizes his experiments.

Objective :	Find a function of one independant variable and one dependant variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target functions is the quartic polynomial $X^4 + X^3 + X^2 + X$
GPTerminal Set:	X (the independant variable)
GPFunction Set	$+, -, *, \%, \sin, \cos, \exp, \log$
Fitness cases	The given sample of 20 data points in the interval $[-1, +1]$
Raw Fitness	The sum, taken over the 20 fitness cases, of the error
Standardised Fitness	Same as raw fitness
Hits	The number of fitness cases for which the error is less than 0.01
Wrapper	None
Parameters	$M = 500$, $G = 51$

Table 4. A Koza-style tableau

The production rules for $< expr >$ are given above. As this and subsequent rules are the only ones that require a choice, they are the ones that will be evolved.

We adopt a similar style to Koza of summarizing information, using a modified version of his *tableau.* Notice how our *terminal operands* and *terminal operators* are analgous to *GPTerminals* and *GPfunctions* respectively.

6.1 Results

GE consistently discovered the target function when both the duplication and pruning operators were employed. Examination of potential solutions from the other runs showed they tended to fixate on operators such as `Sin` or `Exp` as these gene rated a curve similar to that of the target function.

Space constraints prevent us from describing the nature of individuals generated by GE as it was constructing the correct solution. However, the system exploits the variable length genome by concentrating on the start of the string initially. Although the target function can be represented by eighteen distinct genes, there are several other possibilities which, while not perfectly fit, are considerably shorter. Individuals such as $X^2 + X$ generate curves quite comparable to the target, using a mere five genes.

Objective :	Find a function of one independant variable and one dependant variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target functions is the quartic polynomial $X^4 + X^3 + X^2 + X$
Terminal Operands:	X (the independant variable)
Terminal Operators	The binary operators $+, *, /,$ and $-$ The unary operators Sin, Cos, Exp and Log
Fitness cases	The given sample of 20 data points in the interval $[-1, +1]$
Raw Fitness	The sum, taken over the 20 fitness cases, of the error
Standardised Fitness	Same as raw fitness
Hits	The number of fitness cases for which the error is less than 0.01
Wrapper	Standard productions to generate C functions
Parameters	$M = 500$, $G = 51$

Table 5. Grammatical Evolution Tableau

Typically, individuals of this form were discovered early in a run, and contained valuable gene sequences, particularly of the form $X * X$ which, if replicated could subsequently be used to generate individuals with higher powers of X. GE is subject to problems of dependencies similar to GP [O'Reilly 97], i.e. the further from the root of a genome a gene is, the more likely its function is to be affected by other genes.

By biasing individuals to a shorter length, they were encouraged to discover shorter, albeit less fit expressions early in a run, and then generate progressively longer genomes and hence increasingly complex functions in subsequent generations.

7 Conclusions

We have described a system, Grammatical Evolution (GE) that can map a binary genotype onto a phenotype which is a high level program. Because our mapping technique employs a BNF definition, the system is language independant, and, theoretically can generate arbitrarily complex functions.

We believe GE to have closer biological analogies to nature than GP, particularly with its use of linear genomes and the manner in which it uses proteins to affect traits. Other schemes tend to use either the direct evolution of physical traits or the use of simple one to one mappings from genotype to phenotypic traits.

This paper serves an introduction to GE. We believe there is great promise to the system, and that the ability to evolve any high level language would surely be a huge boost for Evolutionary Algorithms, both in terms of increased usage and acceptance, and in terms of the complexity of the problems it can tackle.

Using BNF definitions, it is possible to evolve multi line functions; these functions can even declare local variables and use other high level constructs such as loops. The next step is to apply GE to a problem that requires such constructs.

Although GE is quite different from the functional nature GP, in both its structure and its aims, there is still quite a lot of similarity. We hope that GE will be able to benefit from the huge amount of work already carried out on GP, particularly in the fundamentals, i.e. ADFS, indexed memory, etc. many of which can be coded in BNF with relative ease.

References

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

[Goldberg 89] Goldberg D E, Korb B, Deb K. Messy genetic algorithms: motivation, analysis, and first results.
Complex Syst. 3

[Gruau 94] Gruau, F. 1994. *Neural Network synthesis using cellular encoding and the genetic algorithm.* PhD Thesis from Centre d'etude nucleaire de Grenoble, France.

[Horner 96] Horner, H *A C++ class library for GP*. Vienna University of Economics.

[Keller 96] Keller, R. & Banzhaf, W. 1996. GP using mutation, reproduction and genotype-phenotype mapping from linear binary genomes into linear LALR phenotypes. In *Genetic Programming 1996*, pages 116-122. MIT Press.

[Koza 92] Koza, J. 1992. *Genetic Programming.* MIT Press.

[O'Reilly 97] O'Reilly, U. 1997. The impact of external dependency in Genetic Programming Primitives. In *Emerging Technologies 1997*, pages 45-58. University College London. *To Appear.*

[Paterson 97] Paterson, N & Livesey, M. 1997. Evolving caching algorithms in C by GP. In *Genetic Programming 1997*, pages 262-267. MIT Press.

[Ryan 97a] Ryan, C. & Walsh P. 1997. The Evolution of Provable Parallel Programs. In *Genetic Programming 1996*, pages 406-409. MIT Press.

[Ryan 97b] Ryan, C. 1997. Shades - A Polygenic Inheritance scheme. In *Proceedings of Mendel '97*, pages 140-147. PC-DIR, Brno, Czech Republic.

[Schutz 97] Schutz, M. 1997. Gene Duplication and Deletion, in the Handbook of Evolutionary Computation. (1997) Section C3.4.3

[Whigham 95] Whigham, P. 1995. Inductive bias and genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 461-466. UK:IEE.

[Whigham 96] Whigham, P. 1996. Search Bias, Language Bias and Genetic Programming. In *Genetic Programming 1996*, pages 230-237. MIT Press.

[Wong 95] Wong, M. and Leung, K. 1995. Applying logic grammars to induce subfunctions in genetic prorgramming. In *Proceedings of the 1995 IEEE conference on Evolutionary Computation*, pages 737-740. USA:IEEE Press.

This article was processed using the LaTeX macro package with LLNCS style