An investigation into automatically defined function representations in Grammatical Evolution

Erik Hemberg, Michael O'Neill, Anthony Brabazon
Natural Computing Research & Applications Group, University College Dublin, Ireland
erik.hemberg@ucd.ie,m.oneill@ucd.ie, anthony.brabazon@ucd.ie

Abstract

Automatically defined functions are a fundamental tool adopted in Genetic Programming to allow problem decomposition and leverage modules in order to improve scalability to larger problems. We examine a number of function representations using a grammar-based form of Genetic Programming, Grammatical Evolution. The problem instances include variants of the ant trail, static and dynamic Symbolic Regression instances. On the problems examined we find that irrespective of the function representation, the presence of automatically defined functions alone is sufficient to significantly improve performance on problems that are complex enough to justify their use.

Grammatical evolution, meta grammars, modularity

1 Introduction

In many examples of problem solving humans use a divide-and-conquer approach through the construction of sub-solutions, which may be reused and combined in a hierarchical fashion to solve the problem as a whole. Similarly Genetic Programming (GP) [5] provides the ability to automatically create, modify and delete modules, which can be used in a hierarchical fashion.

This study investigates the adoption of the principles of automatically capturing modularity from GP, and to couple these to an adaptive representation. The contribution is the extension of this approach to grammatical GP systems by using dynamic definition of modules with fixed module signatures. Further, this paper also introduces a novel meta grammar approach to modularity and compares this approach to other grammar based approaches.

The paper begins with a brief introduction to Grammatical Evolution (GE) in Sec. 2, followed by a discussion on automatically defined function representations in Sec. 2.2. Experiments, results and discussion are presented in Sec. 3, and in Sec. 4 conclusions and future work can be found.

2 Grammatical Evolution

Grammar formalism in Evolutionary computation was introduced by Hicklin [4], Grammatical Evolution [9] is a grammar-based form of GP which marries principles from molecular biology to an underlying grammatical representation. Rather than exclusively manipulating tree structures as in standard GP, an abstract genotype that contains the instructions on how to build a sentence or structure in a target language as specified in the grammar. In GE the genetic search operators are traditionally applied to the genotypic binary or integer strings. A codon in GE is the value used to select the production from the current rule. in order to get a valid choice the codon value is divided by the number of productions in the rule and the remainder is used to select which production is chosen.

In this paper we turn our attention towards the structures responsible for modularity in GP, namely, automatically defined functions (ADFs). We introduce a variant of GE which evolves the input grammar itself as this will be one of the ADF representations examined later in this study. This approach is referred to as $(GE)^2$, Grammatical Evolution by Grammatical Evolution [10], or meta-Grammar GE.

2.1 Meta Grammars in Grammatical Evolution

$(GE)^2$, is based on the GE algorithm. This is a meta grammar Evolutionary Algorithm in which the input grammar is used to specify the construction of another syntactically correct grammar. The generated grammar is then used in a mapping process to construct a solution. In Fig. 1 a meta-grammar GE is displayed.

In order to allow evolution of a grammar $(GE)^2$, we must provide a grammar to specify the form a grammar can take. See [9] for further examples of what can be represented with grammars and [12, 13, 1] for an alternative approach to grammar evolution. By allowing an Evolutionary Algorithm to adapt its representation (here through the evolution of

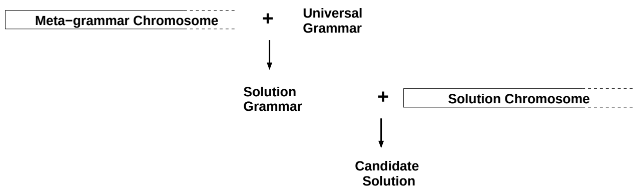


Figure 1: An overview of the meta-grammar approach to GE.

the grammar) it is possible to automatically incorporate biases into the search process. In this case we can allow $(GE)^2$ evolve and automatically define a number of functions.

In $(GE)^2$ the meta grammar dictates the construction of the solution grammar. In this study two separate, variable-length, genotypic binary chromosomes were used, the first chromosome to generate the solution grammar from the meta grammar and the second chromosome generates the solution itself. Crossover operates between homologous chromosomes, that is, the solution grammar chromosome from the first parent recombines with the solution grammar chromosome from the second parent, with the same occurring for the solution chromosomes. For evolution to be successful it must simultaneously evolve the meta grammar and the structure of solutions based on the evolved meta grammar, and as such the search space is larger than in standard GE.

There have been a number of studies of a meta-grammar approach to GE [10, 7, 2]. The original study [10] investigated the feasibility of this approach and demonstrated its effectiveness in dynamic environments. In the mGGA [7] the meta-grammar approach was shown as an effective method as an alternative binary string Genetic Algorithm through the provision of a mechanism to achieve modularity. An observation of some of the solutions and solution grammars evolved by meta-grammar GE has exposed a tendency of generating grammars that did not have the possibility to produce many different strings [7, 2].

2.2 Automatically Defined Functions in GE

There has been a large body of research on modularity in GP and effects on its scalability (e.g., [6, 11]). Some previous work with GE [8, 3] has also been undertaken, but none used the meta-grammar approach. In [3] functions were dynamically created using a dynamic grammar approach that allowed specification of multiple functions and a variable number of arguments for each function. The newly created ADFs were dynamically appended onto the core grammar in such a manner that it was possible to invoke them from the main function. The meta-grammar approach also allows this but separates the input genomes that create and use functions to two separate chromosomes.

The meta-grammar generates the content of the ADFs the number of ADFs that the solution grammar can use. Wrapping is used on both chromosomes, if the mapping is still incomplete the individual is invalid and is assigned the worst possible fitness. Below is an example meta-grammar used for the Ant trails.

```
adf_mg - A meta-grammar that can evolve grammars.
<g> ::=
       <def_fun_u>
       "<prog>      ::= public Test() { while(get_Energy_Left()) { <code>} } "
       "<code>      ::= <line> | <code> <line>"
       "<line>      ::= <condition> | <op>"
       "<condition> ::= if (food_ahead()==1) { <line> } else { <line>}"
       "<op>        ::=  left(); | right(); | move(); | adf*();"
<def_fun_u>    ::= <def_fun_s> | <def_fun_u> <def_fun_s>
<def_fun_s>    ::= "public void adf*() {" <adfcode> "}"
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition> | <adfop>
<adfcondition> ::= if (food_ahead()==1) { <adfline> } else { <adfline> }
<adfop>        ::= left(); | right(); | move();
```

`adf*()` is a function call to a defined function, where a codon is used to select which function is called. In the above example quotes are used to escape symbols, e.g. not expand non-terminals in the meta-grammar, instead expand them in the solution grammar. In a solution grammar with multiple defined ADFs are post-proceed to make each function signature unique.

3 Experiments & Results

In this study we wish to determine if one of the three ADF representations for GE have a performance advantage across a range of benchmark problems.

The meta grammar (adf_mg) approach is compared to a standard GE grammar (std), a GE grammar with the ability to define one method (adf) and a GE grammar that can define any number of methods (adf_dyn). None of the grammars

Fixed chromosome size	100, (200 for normal GE)
Initialisation	Random
Selection operation	Tournament
Tournament size	3
Replacement	Generational
Max wraps	1
Generations	50
Population Size	500
Elite Size	2
Crossover probability meta	0.9
Crossover probability solution	0.9
Mutation probability meta	0.05
Mutation probability solution	0.05, (0.05 for normal GE)

Table 1: Parameters for the GE algorithm

allow ADFs to call ADFs. Unless noted 30 runs were made and the significance of the results is tested by a t-test with p-value=0.05.

The settings in the experiments were used to investigate differences between the approaches of using automatically defined functions. The experimental settings in Table 1 were adopted. The chromosomes were variable-length vectors of integers (4 byte integers) and had the same initial length. One-point crossover where the same crossover point is used for both parents and integer mutation where a new value was randomly chosen are used.

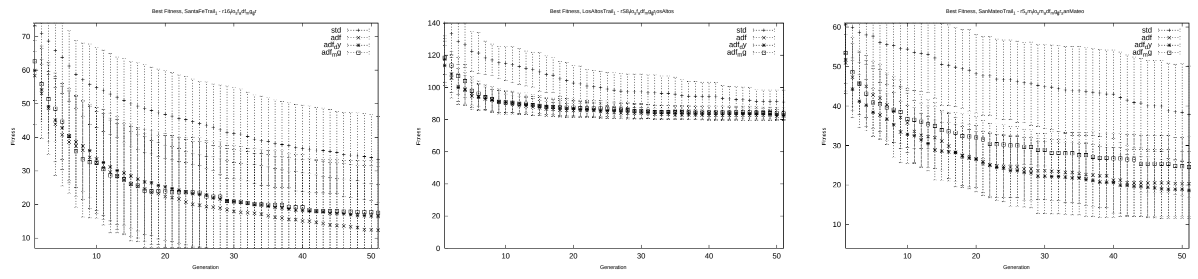
Ant trails

Three Ant trails, the Santa Fe Ant trail, Los Altos Trail from [5] and San Mateo Trail [6], are tested. None of the ADF functions for the ant trails take any arguments. See 2.2 for the meta grammar used.

```
std - The standard GE grammar.
<prog>      ::= <code>
<code>      ::= <line> | <code> <line>
<line>      ::= <condition>  | <op>
<condition> ::= if(food_ahead()==1) {<code>} else {<code>}
<op>        ::=  left(); | right(); | move();

adf - GE grammar with only one ADF.
<prog>         ::= "public Ant() { while(get_Energy() > 0) {"<code>"}} "
                   "public void adf0() {"<adfcode>"}"
<code>         ::= <line> | <code> <line>
<line>         ::= <condition>  | <op>
<condition>    ::= if (food_ahead()==1) {<line>} else {<line>}
<op>           ::=  left(); | right(); | move(); | adf0();
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition>  | <adfop>
<adfcondition> ::= if (food_ahead()==1) {<adfline>} else {<adfline>}
<adfop>        ::= left(); | right(); | move();

adf_dyn - The grammar which allows multiple function definition is shown
below. adf*() is expanded to enumerate all the allowed functions.
<prog>         ::= "public Ant() { while(get_Energy() > 0) {"<code>"} }"<adfs>
<adfs>         ::= <adf_def>  | <adf_def> <adfs>
<adf_def>      ::= " public void adf*() {"<adfcode>"}"
<code>         ::= <line> | <code> <line>
<line>         ::=  <condition>  | <op>
<condition>    ::= "if(food_ahead()==1) {"<line>"} else {"<line>"}"
<op>           ::=  adf*();
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition>  | <adfop>
<adfcondition> ::= "if (food_ahead()==1) {"<adfline>"} else {"<adfline>"}"
<adfop>        ::= left(); | right(); | move();
```

Results - Ant trails

A plot of the Santa Fe Ant trail is shown in Figure2(a) The average best fitness of the last generation is $\overline{std} = 37.9$, $\overline{adf} = 20.33$, $\overline{adf\_dyn} = 18.63$ and $\overline{adf\_mg} = 24.57$ A t-test on the last generation confirms that ADFs are significantly better. For the Los Altos Ant trail a plot is shown in Figure 2(b) The average best fitness of the last generation for the Los Altos trail is $\overline{std} = 33.46$, $\overline{adf} = 12.38$, $\overline{adf\_dyn} = 16.47$ and $\overline{adf\_mg} = 17.57$ When performing a t-test it shows that runs with ADFs are performing significantly better by the last generation compared to the ones with no ADFs. For the San Mateo trail 2(c) the average best fitness of the last generation is $\overline{std} = 90.9$, $\overline{adf} = 82.23$, $\overline{adf\_dyn} = 82.67$ and



(a) Santa Fe Ant Trail (b) Los Altos Ant trail (c) San Mateo Ant trail

Figure 2: Plot with error bars over the generations for the Los Altos and San Mateo ant trail. A t-test confirms, that the fitness differs significantly between standard GE and the other grammars on all problems at the final generation.

$\overline{adf\_mg} = 83.67$ Also for this trail with slightly different behaviour it is significantly better by the last generation to use ADFs. For all the Ant trails it seems like it is beneficial to use ADFs.

Symbolic Regression

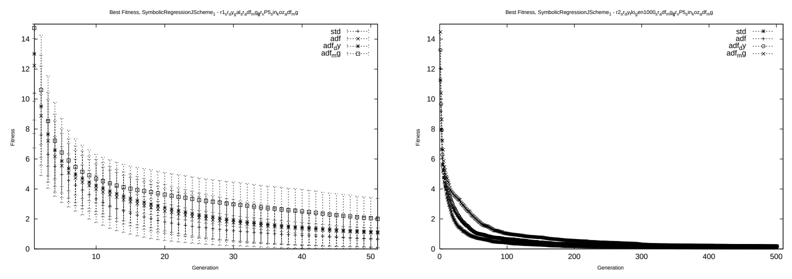
A number of fitness functions for symbolic regression were examined, they were inspired by [6]. The statically defined grammars takes one argument, while the meta grammar approach allows the defined methods to take a variable number of arguments. $x+x^2+x^3+x^4+x^5$(0), $x+x^2+x^3+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}$(1) and $x+x^2+x^3+x^4+x^5+x^6+x^7$(2). To create dynamic problems two Symbolic Regression problems that change periodically are created. In the first the period is every 10 periods (0) switches between (1). In the second problem every 10 generations a polynomial on degree higher then the currently highest is added in (2) $f_0(x) = x, f_t(x) = f_{t-1}(x) + x^{degree(f_{t-1}(x))+1}$, $0 \leq t \leq$ generation/period. The general random constant generates 1000 samples in the range -1.000 to 1.000. All symbolic regression grammars use a naive protected division operator (d), 0.0 was returned if the divisor equalled 0.

The meta-grammar approach for the symbolic regression problems allows creation of any number of functions with variable number of function arguments. The `ADFPARAM` and `ADFUSE` in the grammar indicates where the grammar inserts and uses function arguments. `adf*(ADFARG)` is expanded when the meta grammar is processed to incorporate the defined number of functions and their arguments.

```
std - For the standard GE grammar
<expr> ::= ( <op> <expr> <expr> ) | <var>
<op>   ::= +|-|*|d
<var>  ::= x|(GeneralRandomConstant)

adf - The GE grammar can define one function with one argument.
<prog>     ::= <expr> ") ) (define adf0 (lambda (x) ("<adfexpr1>") ) )"
<expr>     ::= ( <op> <expr> <expr> ) | <var> | (adf0 <expr> )
<op>       ::= +|-|*|d
<var>      ::= x|(GeneralRandomConstant)
<adfexpr1> ::= <op> <adfexpr> <adfexpr>
<adfexpr>  ::= ( <op> <adfexpr> <adfexpr> ) | <adfvar>
<adfvar>   ::= x|(GeneralRandomConstant)

adf_dyn -  The GE grammar for creating any number of functions expands adf* to
denote each generated function. Each function takes one argument.
<prog>     ::= <expr> ") ) "<adfs>
<expr>     ::= ( <op> <expr> <expr> ) | <var> | (adf* <expr> )
<op>       ::= +|-|*|d
<var>      ::= x|(GeneralRandomConstant)
<adfs>     ::= <adf_def> | <adf_def> <adfs>
<adf_def>  ::= "(define adf*(lambda (x) ("<adfexpr1>") ) )"
<adfexpr1> ::= <op> <adfexpr> <adfexpr>
<adfexpr>  ::= ( <op> <adfexpr> <adfexpr> ) | <adfvar>
<adfvar>   ::= x|(GeneralRandomConstant)

adf_mg - meta-GE grammar multiple parameters
<g> ::=
       "<prog>   ::= <expr> ) )"
       <adfs>
       "<expr>   ::= ( <op> <expr> <expr> ) | <var> | adf* (ADFARG)"
       "<adfarg> ::= ( <op> <expr> <expr> ) | <var>"
       "<op>     ::= +|-|*|d"
       "<var>    ::= x|(GeneralRandomConstant)"
<adfs>     ::= <adf_def>split<adfs> | <adf_def>
<adf_def>  ::= "(define adf* (lambda ("<adfparam>") ("<adfexpr1>") ) )"
<adfparam> ::= ADFPARAM <adfparam> | ADFPARAM
<adfexpr1> ::= <adfop> <adfexpr> <adfexpr>
<adfexpr>  ::= ( <adfop> <adfexpr> <adfexpr> ) | <adfvar>
<adfvar>   ::= ADFUSE|(GeneralRandomConstant)
<adfop>    ::= +|-|*|d
```

Results - Symbolic Regression

For the symbolic regression problems none seemed to benefit from ADFs. First a plot of symbolic regression problem (0) is shown in Figure3. For some problems 1000 runs were made instead of 30. Also the number of generations where increased to 500 3(b) to watch how the problem behaved after a longer run. The average best fitness of the last generation

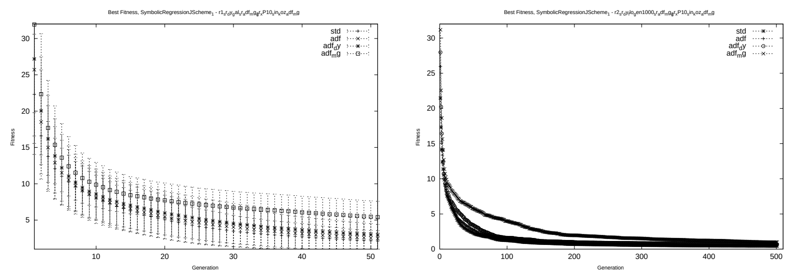


(a) 50 Generations, 1000 runs (b) 500 Generations

Figure 3: Plot with error bars over generations for (0). A t-test confirms that the standard GE is significantly better over a 1000 runs and 50 generations and for 30 runs and 500 generations

is for 50 generations $\overline{std} = 0.67$, $\overline{adf} = 1.06$, $\overline{adf\_dyn} = 1.13$ and $\overline{adf\_mg} = 2.00$ and for 500 generations $\overline{std} = 0.008$, $\overline{adf} = 0.015$, $\overline{adf\_dyn} = 0.018$ and $\overline{adf\_mg} = 0.019$ The standard GE performs significantly better for the longer runs 3(b) and nothing else is discovered with a larger sample.

A plot of (1) is shown in Figure4 The average best fitness of the last generation is for 50 generations $\overline{std} = 2.11$,

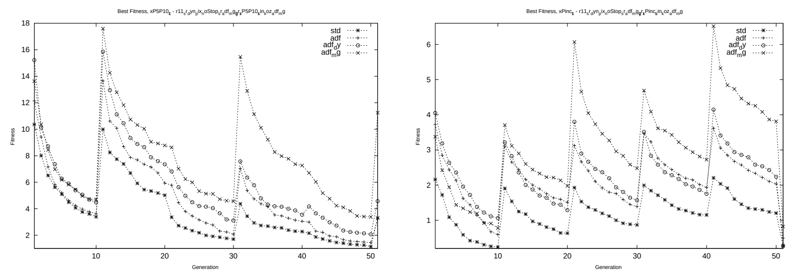


(a) 50 Generations, 1000 runs (b) 500 Generations

Figure 4: Plot with error bars over generations for (1). A t-test confirms, p-value=0.05 that the standard GE is significantly better over a 1000 runs and 50 generations and for 30 runs and 500 generations, except for one adf for 500 generations.

$\overline{adf} = 2.47$, $\overline{adf\_dyn} = 2.94$ and $\overline{adf\_mg} = 5.4$ and for 500 generations $\overline{std} = 0.046$, $\overline{adf} = 0.053$, $\overline{adf\_dyn} = 0.07$ and $\overline{adf\_mg} = 0.098$ Standard GE is significantly better over a 1000 runs and 50 generations and for 30 runs and 500 generations, except for one adf for 500 generations.

Plots of dynamic problems, switching between (0) and (1) and increasing the polynomial by one term every 10 generations (2) is shown in Figure5. The average best fitness of the last generation is when switching between (0) and



(a) (0) and (1) switching (b) (2)

Figure 5: Plot with error bars over generations for dynamic functions with a period of 10. Left periodic switching between (0) and (1) and right (2). A t-test confirms, that the standard GE is significantly better then Meta Grammar

(1) $\overline{std} = 3.31$, $\overline{adf} = 3.23$, $\overline{adf\_dyn} = 4.57$ and $\overline{adf\_mg} = 11.25$. Only adf_mg differs significantly from std. The average best fitness of the last generation for (2) $\overline{std} = 0.273$, $\overline{adf} = 0.49$, $\overline{adf\_dyn} = 0.28$ and $\overline{adf\_mg} = 0.82$, here

there are no significant differences. Symbolic Regression does not show up any clear benefits from the incorporation of ADFs.

The performance for the grammars differs on all the problems, it seems like both the type as well as the size of the problem is very influential. In [3] it was argued that the meta grammar approach would benefit from structure preserving operators.

4 Conclusion & Future work

On the problems examined we find that irrespective of the representation, the presence of automatically defined functions alone is sufficient to significantly improve performance on some problems. In some instances we observe an additional overhead with the adoption of a meta-grammar form of function representation.

After noting the quite different behaviours it would be interesting to investigate more problems. Especially tests with problems that use a variable number of arguments to functions. A more thorough study on when the problem warrants use of ADF could be very helpful. Further the problems where ADFs are used with more advantage the quantity and structure of the ADFs could help guide further improvements. The grammars can be investigated further by allowing the ADFs to recursively call itself or other ADFs to certain depth might help the expressiveness of the ADF.

5 Acknowledgement

This research is based upon works supported by the Science Foundation Ireland under Grant No. 06/RFP/CMS042.

References

[1] R.M.A. Azad, C. Ryan, T. Yu, R.L. Riolo, and B. Worzel. An Examination of Simultaneous Evolution of Grammars. *Genetic Programming Theory and Practice* {*III*}, 9:141–158.

[2] I Dempsey. *Grammatical Evolution in Dynamic Environments*. Phd thesis, University College Dublin, 2007.

[3] Robin Harper and Alan Blair. Dynamically defined functions in grammatical evolution. In *CEC 2006*, pp 9188–9188, 2006.

[4] J.F. Hicklin. *Application of the genetic algorithm to automatic program generation*. University of Idaho, 1986.

[5] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, 1992.

[6] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[7] Michael O'Neill and Anthony Brabazon. mGGA: The meta-grammar genetic algorithm. In Maarten Keijzer, et al, eds, *EuroGP 2005*, volume 3447 of *LNCS*, pp 311–320, 2005.

[8] Michael O'Neill and Conor Ryan. Grammar based function definition in grammatical evolution. In Darrell Whitley, et al, eds, *GECCO-2000*, pp 485–490, 2000

[9] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer, USA, 2003.

[10] Michael O'Neill and Conor Ryan. Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code. In Maarten Keijzer, et al, eds, *Eurogp2004*, volume 3003 of *LNCS*, pp 138–149, 2004.

[11] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. `http://www.gp-field-guide.org.uk`, 2008.

[12] Yin Shan, Robert I. McKay, Rohan Baxter, Hussein Abbass, Daryl Essam, and Nguyen Xuan Hoai. Grammar model-based program evolution. In *CEC2004*, pp 478–485, 2004.

[13] Peter Alexander Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Canberra, Australia, 14 October 1996.