Genetic Programming with Primitive Recursion

Stefan Kahrs
Department of Computer Science
University of Kent at Canterbury
Canterbury CT2 7NF
United Kingdom
smk@kent.ac.uk

ABSTRACT

When Genetic Programming is used to evolve arithmetic functions it often operates by composing them from a fixed collection of elementary operators and applying them to parameters or certain primitive constants. This limits the expressiveness of the programs that can be evolved. It is possible to extend the expressiveness of such an approach significantly without leaving the comfort of terminating programs by including primitive recursion as a control operation.

The technique used here was *gene expression programming* [2], a variation of grammatical evolution [8]. Grammatical evolution avoids the problem of program bloat; its separation of genotype (string of symbols) and phenotype (expression tree) permits to optimise the generated programs without interfering with the evolutionary process.

Track

Genetic Programming

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Program Complexity**]: General; D.3.3 [**Language Constructs and Features**]: Conbtrol Structures; I.2.2 [**Automatic Programming**]: Program synthesis, Program Transformation

General Terms

Algorithms, Theory, Experimentation

Keywords

Grammatical evolution, primitive recursion, program transformation

1. INTRODUCTION

Primitive Recursion is a scheme to define functions over the natural numbers. Its expressiveness was explored in detail in the 1920s and 1930s [7]; Primitive recursion is a scheme that permits to define a function by descending recursion in one variable, or to phrase it in the imperative paradigm: it has for-loops as the only iterative control structure. Therefore primitive recursive programs always terminate.

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
ACM 1-59593-186-4/06/0007.

Moreover, terminating programs whose behaviour is *not* expressible through primitive recursion have necessarily very poor (worst case) termination behaviour, because as a consequence of Kleene's normalisation theorem [3] the run-time of any such program could not be bounded by a primitive recursive function, pushing them way beyond what complexity theorists regard as feasible.

Of course, genetic programming with loops is not new (see e.g. [4, 5]), but in the unconstrained case non-terminating loops are common and require special monitoring. Bounded loops have been looked at amongst the programming constructs for genetic programming before as well (also in [4]), but the fact that they induce a hugely important subclass of programs has not attracted much attention in this context.

2. REPRESENTING PRIMITIVE RECURSIVE FUNCTIONS

A fairly natural and very compact representation of primitive recursion is given in [1]. This uses two connectives, (generalised) composition `Cn` and the primitive recursion operator `Pr`, and as primitives the constant-0 function `Z`, the successor function `S`, and the argument selectors $\mathtt{id}_n^m$ which select the n-th argument out of m. In the syntax tree of a primitive recursive function `Cn` and `Pr` would be the only internal nodes, while the primitive functions would only occur as leaves.

The meaning of these operators is given as follows: the expression $\mathtt{Cn}[f, g_1, \ldots, g_n]$ defines a new function h through (generalised) function composition as follows:

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

The operator `Pr` defines a new function through the scheme of primitive recursion; if $h = \mathtt{Pr}[f, g]$ then h defines the following function:

$$\begin{aligned} h(0, \vec{x}) &= f(\vec{x}) \\ h(n+1, \vec{x}) &= g(h(n, \vec{x}), n, \vec{x}) \end{aligned}$$

Here, $\vec{x}$ stands for an argument vector, and the functions f, g as well as the g_i are (previously defined) primitive recursive functions. These definitional schemes have some implicit restrictions regarding the arities of the functions involved, e.g. all the functions g_i in the generalised composition need to have the same arity. To support an evolutionary approach it is useful to relax these condtions by applying conventions for missing or redundant arguments — redundant arguments can be ignored, missing ones can be set to 0.

3. TREES AS LISTS

Grammatical evolution and gene expression programming represent trees over a first-order signature as sequences of symbols of that signature, viewing the list as a tree-traversal — level-order traversal [9] in case of gene expression programming, pre-order traversal in case of grammatical evolution. We chose the former as it has advantages for creating an initial population, although it does not behave quite as well under cross-over breeding as grammatical evolution [6].

Not all randomly produced lists of such symbols correspond to proper trees, there is a possibility of both overflow and underflow. Underflow gaps can be filled either using default constants or [2] by selecting them from a second list (the "tail") of only nullary symbols; with the latter approach, underflow can simply be avoided as well by not allowing nullary symbols in the first list, a measure which ensures that such trees are balanced.

Trees can be recovered from these lists. Breeding between individuals represent in this way can cheaply be realised by a list cross over.

4. DISCOVERING PATTERNS

Grammatical evolution distinguishes between the list of function symbols, and its associated syntax. The separation between these two concepts opens up a new opportunity: the trees can be manipulated, to optimise the code, without interfering with the genetic make-up of an individual.

Optimisations for this approach are indeed vital: the combinations of for-loops create complex computational patterns with mostly trivial outcomes. It is useful to discover patterns which *reduce the complexity class* of the computation, i.e. when a loop can be eliminated. While $\mathtt{Pr}[f, g]$ generally corresponds to a for-loop, there is no point to run that loop if the function g is not strict in its first argument, the loop state; in that case the expression can be expressed by an if-then-else. The implementation looks out for a number of computational patterns of this kind that improve run-time and tree-size.

5. EXPERIMENTS

Experiments with the approach used cross-over breeding and selection as genetic operators. That means: a fixed proportion of the best performers were made to survive to the next generation, and the remainder of the population was restocked through cross-breeding the best two out of four individuals. The lack of a mutation operator in the implementation was responsible for a certain amount of genetic drift. To some extent its absence was compensated by interpreting the symbol lists as level-order traversals, because this creates in itself a degree of randomness whenever parents are crossed over at incompatible points. Keeping the best performers allowed to direct genetic drift towards optimal solutions.

Relatively simple arithmetic functions (partity-check, polynomials) would typically be found within 40 generations, for population sizes of a few thousand, and tree sizes of about 20 internal nodes. For more complex, or random functions only approximations were found. Singling out certain primitive recursive functions (addition, multiplication) as additional primitives transformed the search space, but these transformations turned out to be counter-productive.

6. PROBLEMS WITH THE APPROACH

As mentioned, *all* programs not expressible through primitive recursion have necessarily a very poor worst-case runtime. The flip-side of this argument is that *some* primitive recursive programs show poor run-time behaviour as well, e.g. the experiments came across functions whose runtimes were doubly-exponential, or worse.

Although these were relatively rare (less than one in 1000 individuals), some measure was still needed to stop such programs from bringing an entire simulation to a standstill. The simple solution was to stop "large numbers" being used for loop iteration; individuals attempting to do that were regarded as unfit.

7. CONCLUSIONS

Primitive recursive functions are a viable tool for genetic programming, for several reasons: (i) they are sufficiently expressive; (ii) the small number of primitives keeps program redundancies at bay; (iii) the search space is not polluted with non-terminating programs.

The specific approach employed was grammar evolution [8], which has two main advantages for this application. One is the general advantage of avoiding program bloat, the other is the separation between genotype and phenotype. The latter is of particular importance, as it allows to perform a considerable number of simplifications on syntax trees without interfering with the genetic information.

8. REFERENCES

[1] George Boolos and Richard Jeffrey. *Computability and Logic*. Cambridge University Press, 1974.

[2] Candida Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[3] Stephen Cole Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53:41–73, 1943.

[4] John R. Koza, Forrest H. Bennett, David Andre, and Martin A. Keane. *Genetic Programming III*. Morgan Kaufmann Publishers, 1999.

[5] William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, 1998.

[6] M. O'Neill and C. Ryan. Crossover in grammatical evolution: A smooth operator. In *Proceedings of the European Conference on Genetic Programming*, number 1802 in Lecture Notes in Computer Science, pages 149–162. Springer-Verlag, 2000.

[7] Rózsa Péter. Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion. *Mathematische Annalen*, 110:612–632, 1934.

[8] Conor Ryan, J. J. Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 14-15 1998. Springer-Verlag.

[9] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988.