

A Representation for the Adaptive Generation of Simple Sequential Programs

Michael Lynn Cramer
Texas Instruments Inc.
PO Box 226015, MS 238
Dallas, TX 75266

ABSTRACT

An adaptive system for generating short sequential computer functions is described. The created functions are written in the simple "number-string" language **JB**, and in **TB**, a modified version of **JB** with a tree-like structure. These languages have the feature that they can be used to represent well-formed, useful computer programs while still being amenable to suitably defined genetic operators. The system is used to produce two-input, single-output multiplication functions that are concise and well-defined. Future work, dealing with extensions to more complicated functions and generalizations of the techniques, is also discussed.

INTRODUCTION

The techniques of adaptive Genetic Algorithms [**GAs**]¹ have been shown to be useful in many areas. Initially, these systems involved the adjusting of a fixed set of parameters in order to optimize the performance of a given algorithm². Much work has been done toward the goal of evolving the algorithms *themselves*, particularly in Production System-like domains^{1(chap8),3,4}. This paper discusses work towards developing a sequential programming language that is suitable for manipulation by **GAs** so as to permit the adaptive generation of simple computer functions from low-level computational primitives.

FUNCTIONAL REPRESENTATION

The scheme that we will follow is first to find a suitably powerful programming language, and then encode the programs in this language in such a way as to make them amenable to the standard Genetic Operators [**GOs**].

The basic language to be used is a variation of the algorithmic language **PL** having the following operators:

```
(:INC VAR) ;;add 1 to the variable VAR  
(:ZERO VAR) ;;set the variable VAR to 0  
(:LOOP VAR STAT) ;;perform the statement STAT VAR times  
(:GOTO LAB) ;;jump to the statement with label LAB
```

Programs in **PL** consist of an arbitrary number of globally-scoped (positive) integer variables and statements containing operators of the above forms. Two simple example **PL** Programs are:

```
;;Set variable V0 to have the value of V1  
(:ZERO V0)  
(:LOOP V1 (:INC V0))  
;;Multiply V3 to V4 and store the result in V5  
(:ZERO V5)
```

(:LOOP V3 (:LOOP V4 (:INC V5)))

While PL can be shown to be Turing Equivalent⁵, we will be interested in the language subset PL-{:GOTO}. This language subset has two useful properties: first, while it is not fully Turing Equivalent, it still comprises a powerful set of functions (specifically, the set of primitive recursive functions)⁵ and second, programs written in PL-{:GOTO} are guaranteed to halt. Finally, we make two small extensions to the language. First, a :SET operator, which accepts two variables and sets the value of the first variable equal to that of the second. (As can be seen in the examples above, this operation is trivially definable in PL-{:GOTO}; if so desired, it can be considered a macro or subroutine operator.) Secondly, we define a :BLOCK operator that accepts two statements as arguments and evaluates the two statements sequentially. (This is essentially just a grouping operation that has no effect on the overall structure of the language.)

Now, the encoded representation for our programs should have two characteristics:

(Goal 1) It should be amenable to the standard GOs.

(Goal 2) The representation should produce only well-formed programs, even when subjected to the GOs. While some representations, e.g. character-strings, might be well suited for the mechanisms of GOs, the random generation and/or altering of characters is not likely to produce, say, a useful FORTRAN program. Consequently, it is strongly desirable that the chosen representation be such that all such generated programs stay in the space of syntactically correct programs. Not all such generated programs would be useful (adapation would be expected to correct that); it is only important at this point that such programs be well formed.

This paper will consider lists of integers as a representation for these programs where the object the integer represents (variable, operator, etc.) is determined by the integer's position in the list. Clearly such a representation satisfies Goal 1 above, the standard GOs (Crossover, Mutation, Inversion) would be well defined on such a list. To satisfy Goal 2, we need to define a decoding of an arbitrary list into a well-formed program.

THE JB LANGUAGE

A first attempt at such a decoding is the language JB. The list of integers is first divided into statements of some length large enough for the longest statement size, (three in the present case). Any integers left over at the end of this list are ignored. The first of these statements is defined to be the Main Statement [MS] and the remaining N_{as} statements are the Auxiliary Statements [AS]. Syntactically, these statements are interpreted as follows:

(0 4 2) -> (:BLOCK AS₄ AS₂)

(1 6 0) -> (:LOOP V₆ AS₀)

(2 1 9) -> (:SET V₁ V₉)

(3 17 8) -> (:ZERO V₁₇) ;;the 8 is ignored

(4 0 5) -> (:INC V₀) ;;the 5 is ignored

Here the symbols of the forms V_n and AS_n represent, respectively, example Variables and Auxiliary Statements.

This body of statements is embedded in an environment containing N_{bv} body-variables (initialized to 0) and N_{iv} input-variables. At the end of the execution of the program, any of the $N_{votot} = (N_{iv} + N_{bv})$ available variables can be returned as output.

The function is entered by executing the MS, which, typically, will call on one or more of the AS's. An example **JB** program would be:

(0 0 1 3 5 8 1 3 2 1 4 3 4 5 9 9 2)

This would be grouped into the following Statements:

(0 0 1) ;;main statement -> (:BLOCK AS₀ AS₁)

(3 5 8) ;;auxiliary statement 0 -> (:ZERO V₅)

(1 3 2) ;;auxiliary statement 1 -> (:LOOP V₃ AS₂)

(1 4 3) ;;auxiliary statement 2 -> (:LOOP V₄ AS₃)

(4 5 9) ;;auxiliary statement 3 -> (:INC V₅)

This is the same as the **PL** multiplication program above.

As can be seen, virtually (see below) any list (of sufficient length) of integers chosen from the range $[0, N_{rand}-1]$ can be used to generate a well-formed **JB** program. Where $N_{rand} = N_{vtot} * N_{as} * N_{op}$ (N_{op} is the total number of operator types). A particular language object (variable, AS, operator-type) needed for the program can then be extracted from a given integer in the list by taking the modulus of that integer with respect to the respective number above. This ensures random selection over all syntactic types. Two problems arise from this straight forward use of the **JB** language. The first, a minor problem, is that a **JB** integer-list will not define a correct program when a loop is created among the Auxiliary Statements. In practice, with a moderate number of AS's this is a rare occurrence. Moreover, it is easy to remove such programs during the expansion of the body of the program. (In any case, this problem will be removed in the **TB** language below.)

A second, more serious problem is that while the mechanisms of the applications of the **GOs** are very simple in the **JB** language, the semantic implications of their use are quite complicated. Because of the structure of **JB**, semantic positioning of a integer-list element is extremely sensitive to change. As a specific example, consider a large complicated program beginning with a **:BLOCK** statement in the top-level Main Statement. A single, unfortunate, mutation converting this operator to a **:SET** would destroy any useful features of the program. Secondly, this strongly epistatic nature of **JB** seems incompatible with Crossover, given Crossover's useful-feature-passing nature. A useful **JB** substructure shifted one integer to the right will almost certainly contain none of its previously useful properties.

THE TB LANGUAGE

In an effort to alleviate these problems, we consider a modified version of **JB**. This language, called **TB**, takes advantage of the implicit tree-like nature of **JB** programs.

TB is fundamentally the same as **JB** except that the Auxiliary Statements are no longer used. Instead, when a **TB** statement is generated, either at its initial creation or as a result of the application of a **GO** (defined below), any subsidiary statements that the generated statement contains are recursively expanded at that time. The **TB** programs no longer have the simple list structure of **JB**, but instead are tree-like. Because we are simply recursively expanding the internal statements without altering the actual structure of the resulting program, the **TB** programs still satisfy Goal 2. Indeed, it can be seen that, because of its tree-like structure, **TB** does not suffer from the problem of internal loops described above. Thus, all possible program trees do indeed describe syntactically correct programs.

An example of a **TB** program is:

```
(0 (3 5) (1 3 (1 4 (4 5) ) ) )
```

This expands to the same **PL** and **JB** multiplication programs given above.

The standard **GOs** are defined in the following way:

Random Mutation could be defined to be the random altering of integers in the program tree. This would be valid but would encounter the same "catastrophic minor change" problems as did **JB**. Instead, Random Mutation is restricted to the statements near the fringe of the program tree. Specifically: 1) to leaf statements, i.e., those that contain operators that do not themselves require statements as arguments (:INC, :SET, :ZERO). And 2) to non-leaf statements (with operators :BLOCK, :LOOP) whose sub-statement arguments are themselves leaf operators. Inside a statement, mutation of a variable simply means randomly changing the integer representing that variable. Mutating an operator involves randomly changing the integer representing the operator and making any necessary changes to its arguments, keeping any of the integers as arguments that are still appropriate, and recursively expanding the subsidiary statements as necessary.

Similarly, following Smith⁶, we restrict the points at which Crossover can occur. Specifically, Crossover on **TB** is defined to be the exchange of subtrees between two parent programs; this is well-defined and clearly embodies the intuitive notion of Crossover as the exchange of (possibly useful) substructures. This method is also without the problems that Crossover entails in **JB**. In a similar manner, we could define Inversion to be the exchange of one or more subtrees within a given program.

EXAMPLE

As a concrete example, an attempt was made to "evolve" concise, two-input, one-output multiplication functions from a population of randomly generated functions. As discussed by Smith^{3(Chap5)} a major problem here is one of "hand-crafting" the evaluation function to give partial credit to functions that, in some sense, exhibit multiplication-like behavior, without actually *doing* multiplication.

After much experimentation, the following scheme for giving an evaluation score was used. For a given program body to be scored, several instantiations of the function were made, each having a different pair of input variables [IVs]. Each of these test functions was given a number of pairs of input values and the values of all of the function's variables were collected as output variables [OVs]. The resulting output values were examined and compared against the various combinations of input values and IVs. The following types of behavior were noted and each successive type given more credit: 1] OVs that had changed from their initial values. (Is there any activity in the function?) 2] Simple Functional dependence of an OV on an IV. (Is the function noticing the input?) 3] The value of an IV is a factor of the value of an OV. (Are useful loop-like structures developing?) 4] Multiplication. (Is an OV exactly the product of two IVs.)

Furthermore, rather than accept input and/or output in arbitrary variables, scores were given an extra weight if the input and/or output occurred in the specific target variables. To ensure that the functions remain reasonably short, functions beyond a certain length are penalized harshly. Finally, a limit is placed on the length of time a function is permitted to run; any function that has not halted within in this time is aborted.

A number of test runs were made for the system with a population size of fifty. These were compared against a set of control runs. The control runs were the same as the regular runs except that there was no partial credit given; all members of the population were given a low, nominal score until they actually started multiplying correctly. All runs were halted at the thirtieth generation. The system produced the desired multiplication functions 72% more often than the control sample.

FUTURE WORK

Finally, a number of questions remain concerning the present system and its various extensions:

Extensions of the Present System: Generation of other types of simple arithmetic operations seem to be the next step in this direction. Given the looping nature of the underlying **PL** language it seems obvious that the system should be well suited for also generating addition functions. However, it is less clear that it would do equally well attempting to generate, e.g., subtraction or division functions, to say nothing of more complicated mathematical functions. Indeed, the results of the control case above show that it is difficult not to produce multiplication in this language; generation of other types of functions would prove an interesting result. On the other hand, are there other, comparably simple, languages that are better suited to other types of functions?

Concerning Extensions of the Language: A useful feature of the original **JB** language is its suitability for the mechanisms of the **GOs**. Can some further modification be made to the current **TB** language to bring it back into line with a more traditional bit-string representation? Are these modifications, in fact, really desirable? Alternatively, would it be useful to modify the languages to make **GOs** less standard? For example, would it be productive to formalize the subroutine swapping nature of the present method of Crossover and define a program as a structure comprising a number of subroutines, where the application Crossover and Inversion was restricted to the swapping of entire subroutines, and Random Mutation restricted to occurring inside the body of a subroutine?

ACKNOWLEDGEMENTS

I would like to thank Dr. Dave Davis for innumerable valuable discussions and Dr. Bruce Anderson for preserving the environment that made this work possible.

REFERENCES

1. Holland, John H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
2. Bethke, A., *Genetic Algorithms as Function Optimizers*, Ph.D. Thesis, University of Michigan, 1980.
3. Smith, S.F., *A Learning System Based on Genetic Adaptive Algorithms*, Ph.D. Thesis, Univ. of Pittsburgh, December, 1980.
4. Holland, J.H. and J. Reitman, *Cognitive Systems Based on Adaptive Algorithms*, in *Pattern Directed Inference Systems*, Waterman and Hayes-Roth, Ed. Academic Press, 1978.
5. Brainerd, W.S. and Landweber L.H., *Theory of Computation*, Wiley-Interscience, 1974.
6. Smith, S.F., *Flexible Learning of Problem Solving Heuristics through Adaptive Search*, Proc. IJCAI-83, 1983.

Tom
Westerdale

**PROCEEDINGS OF
AN INTERNATIONAL CONFERENCE ON
GENETIC ALGORITHMS
AND THEIR APPLICATIONS**

**July 24-26, 1985
at
Carnegie-Mellon University
Pittsburgh, PA**

**Sponsored By
Texas Instruments, Inc.
U.S. Navy Center for Applied Research
in Artificial Intelligence
(NCARAI)**

**John J. Grefenstette
Editor**