Evolving Turing Complete Representations

John R. Woodward
School of Computer Science
The University of Birmingham
B15 2TT UK
J.R.Woodward@cs.bham.ac.uk

Abstract-

Standard GP, chiefly concerned with evolving functions which are mappings from inputs to output, is not Turing Complete. This paper raises issues resulting from attempts at extending standard GP to Turing Complete representations. Firstly, there is a problem when a contiguous peice of code is moved to a new location (in a different program) by crossover. In general its functionality will be altered if global memory is used, as other parts of the program may access the same peice of memory. Secondly, traditional crossover does not respect modules. Crossover can disrupt a group of instructions that were working together (e.g. in the body of a loop) in one parent, but end up separated in two different offspring after reproduction.

A crossover operator is proposed that only operates at the boundaries of modules. The identification of module boundaries is made easy by using a representation in which explicit modules are defined, in contrast with other representations where the module boundaries would have to be identified by some other means.

The halting problem is a central issue, however as a consequence of this crossover operator we are more likely to produce self terminating programs, thus saving time when testing.

1 Introduction

Genetic Programming (GP) is the biologically inspired search of the space of programs using crossover (XO) and mutation operators [BNKF98]. Typically, the program space does not include Turing Complete (TC) programs but a subset (e.g. logically expressions or mathematical expressions). Recently, however, researchers have begun to extend standard GP to include memory and iteration making it TC [Tel94c] (see figure 1). For the purpose of this paper, a TC representation will mean any representation capable of expressing a model of computation proved to be equivalent to a Turing Machine, for example The C Programming Language. Other examples are given in section 2. Given that the representation GP is evolving is TC we are now searching the space of algorithms or effective procedures (Church's Thesis). Turing Complete Genetic Programming (TCGP) will mean the search of the space of TC representations, as opposed to standard GP.

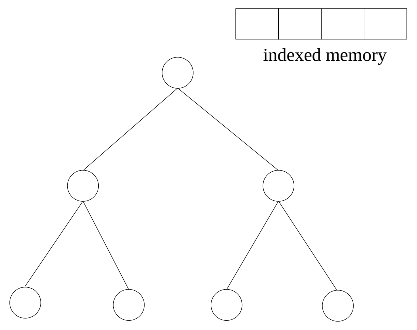

Figure 1: Standard GP is not Turing Complete but can be supplemented by adding a `repeat until` function and adding read and write functions to the function set. The `repeat until` loop allows iteration. The read and write functions allow the system to store data in a global array (illustrated as set of boxes).

As GP techniques become more sophisticated, the problems attempted will become more demanding often requiring a more expressive representation than that used with standard GP. Standard GP has concentrated on functional/reactive programs which are a direct mapping from inputs to output. The same input produces the same output and does not depend on the history of the inputs. For some problems, however, the history is important and to tackle this memory is required in the system (see [Tel94a] for a discussion).

Generalization is the goal of learning, and the use of iteration and memory may help in expressing regularities in data, and therefore should be expressible in the representation. In addition, for a given problem it maybe difficult to decide beforehand what is the most appropriate representation, which has an inherent bias in it, the classic example being perceptrons inability to express non linearly separable problems. If a TC representation is used, the system can in principle evolve a suitable representation. This paper addresses some of the issues of exploring the space of TC programs by looking at a suitable XO operator and representation.

The outline of the paper is as follows. In section 2 representations and operators used by other researchers are described and problems with these approaches are examined in section 3. In section 4, TC landscapes are considered and composition is defined. In section 5, a representation and XO operator are proposed, and how it avoids the issues is discussed in section 6. In section 7 the paper is concluded.

2 Related Work

This section reviews a number of representations and operators used in TCGP. Unfortunately, these systems have been tried on different problems so it is not possible to compare the results directly. Often the solutions evolved are general (i.e. they work on unseen test cases), which is the central goal of learning. Often noiseless training data was used so one cannot comment on issues of over fitting. In all cases, upper bounds were placed on the execution time of programs being evolved to avoid the problem of evaluating non terminating programs.

Tanomaru[Tan93] directly evolves the state transition table of Turing Machines. XO exchanges contiguous rows of the transition table, but this can result is inconsistent machines referring to non-existent states. Mutation randomly changes a single symbol in transition table. In further experiments, XO is dropped and mutation is enhanced by allowing states to be added or deleted, or the replacement of a whole automaton with an entirely new one. Better results were obtained than with the previous operators. For other examples see [TA96, VR01].

In a seminal paper, Cramer [Cra85] uses a programming language JB (list structure) which is a subset of a TC language (which corresponds to the primary recursive functions), the advantage being that programs are guaranteed to halt. The data is stored as global variables. He comments on the inappropriateness of XO given the epistatic nature of JB, and introduces a modified language TB which has a tree-like structure. In TB, XO is the exchange of subtrees between parents and embodies the intuitive notion of XO as the exchange of useful substructures (sub solutions), however it still uses global variables.

Huelsbergen[Hue98] evolves an assembly like language which operates on a global register. His instruction set includes mathematical and logical functions along with conditional jumps to control program flow. XO exchanges contiguous groups of instructions. Mutation operates on single instructions. A second XO operator is also used for comparison purposes; headless chicken XO exchanges contiguous groups of instructions with a randomly created set of instructions, and thus operates as a macro mutation, which interestingly, performs better than standard XO.

Nordin et al. extend the Compiling GP System (CGPS)[NB95] , which uses instructions of fixed length, to Automatic Induction of Machine Code GP (AIM-GP)[NBF99], which uses instructions of variable length. The original motivation was to directly evolve machine code avoiding the interpretation process high level languages have to go through, achieving an impressive improvement in speed of around 1000 times compared to LISP implementations. In CGPS, XO points are readily identifiable as instructions are of fixed length. To avoid this issue in AIM-GP, instructions are grouped together in fixed length instruction blocks (the size of the blocks being a globally defined parameter). XO exchanges these instruction blocks. As more than one instruction may appear in an instruction block, XO cannot separate them. Mutation can either affect a whole instruction block, an instruction or the operand of an instruction.

Push is a strongly typed, stack based programing language designed for evolution [SR02]. While modularity has been introduced into GP in the form of automatically defined functions, in Push modularity (along with recursion and control structures) 'comes for free'. It should be noted, however, that modularity (recursion and control structures) comes for free in any TC representation! GP may produce potentially syntactically invalid programs during evolution; there can be either too few or too many arguments on a stack. To avoid this problem, if extra arguments are present they are ignored, and if too few arguments are present the instruction is ignored, thus all programs evolved are syntactically valid. Traditional genetic operators are used which exchange randomly chosen sub expressions with either a new random sub expression or one from another individual.

3 Issues

We now consider the following issues; global memory interference, the disruption of modules by XO, the halting problem, the building block hypothesis, and the genotype phenotype mapping by considering the representations in section 2 and how they are manipulated by their respective genetic operators.

All the models in section 2 have global memory. Let us consider this in the context of the representation used by Huelsbergen [Hue98]. If a section of code is performing some function, it will in general access the registers (the global memory). However, if this chuck of code is selected for XO and is inserted into another program it will perform a different function as there is no guarantee that the rest of the program has left the specific registers in an appropriate state. Register machines suffer from the problem of what we will call *global memory interference*, as what one chuck of code would do in one program, when it is crossed over will do something else when inserted in a different program. Similarly, the crossed over code may overwrite data in registers the rest of the program was using. It is perhaps not surprising that headless chicken XO substantially outperforms standard XO in [Hue98]. GP with indexed memory also suffers from this problem as the memory is a global array accessed by read and write instructions. Nordin et al.[NB95] make the note of the problem of register interference and overcome it by storing a backup of the registers and restoring them after a function call is completed.

Program flow is controlled in some models by either absolute or relative jumps. For example, Turing Machines typically use absolute jumps to states, whereas register machines typically use relative jumps. Let us first consider absolute jumps. If a contiguous piece of code is selected for XO, and transferred to a new location, whatever function it was performing before, it will almost certainly be performing a different function in its new context as the absolute position of the code may have been moved in the XO process. Relative jumps do solve this problem, as instructions in a contiguous chunk of code will still refer to each other, independent of their absolute position in the overall program. However, XO may separate instructions that were being jumped from and to (i.e. acting together), so in the context of a new program entirely different instructions may be jumped to. Thus XO can completely destroy the functionality of a contiguous section of code by separating it. For example consider a number of instructions which are acting as the body of a loop (i.e. they are the content of the loop), XO can choose a point in the body of the loop and separate instructions which were working together. In general, if a chunk of code is selected for XO it will perform a different function after XO. Let us refer to this as *disruption of modules by XO*.

We cannot know beforehand if a program will halt or not. The methods discussed in section 2 all adopt the simplest solution by putting an upper limit on the computing time of each program in the population. This limit is removed after evolution, allowing the program to compute on input which may require longer to process than examples included in the training set. (Some researchers also impose a memory limit, however this is implied by the time limit as reading and writing to memory require time). The main objection to this method is that if the limit is set too low it will prohibit solutions evolving, and if it is set too high time is wasted waiting for non terminating programs. Unfortunately, none of the researcher in section 2 report the proportion of programs which had to be terminated because they reached the upper limit, or how this proportion varied during the evolution. Teller [Tel94b] describes an alternative method, based on an analogy with cooking popcorn, where the population is evaluated after a certain proportion of the programs have halted. Possible other solutions include gradually increasing the time limit as evolution progresses, or self adapting the limit. As a consequence of the XO operator proposed, the chance of evolving a terminating program in the next generation is increased (see section 6). This is a by product of the original motivation of this work in proposing an operator which does not suffer from the problems of global memory interference and disruption of modules by XO raised above.

There has been discussion about the building block hypothesis, mainly in the genetic algorithm community but also in the GP community. The motivation of XO is that it will combine good parts of one individual with good parts of another. However, due to the potentially destructive nature of XO in TC representations, this would appear not to be the case. Tanomaru [Tan93] drops XO in his second set of experiments as *"the idea of XO is based on the building blocks hypothesis, which is unlikely to hold in the case of automata generation"*. Huelsbergen [Hue98] raises the question of how beneficial building blocks are given that headless chicken macro mutation substantially outperforms standard XO. Nordin et al. [NBF99] claim that as compound instructions may appear in one instruction block, XO cannot separate these so it is easier to protect building blocks against XO. However, the size of instruction blocks is controlled by the user, and therefore does not evolve, limiting the potential building blocks that may be produced.

Syntactically similar programs do not produce semantically similar programs; changing one instruction in a program can have catastrophic effect on its behaviour. Teller[Tel94b] states *"because the space of algorithms is so discontinuous, the mutation operator might almost as well erase the old individual and make a new one from scratch"*, which indeed this was one of the operators Tanomaru[Tan93] used. In evolution it is desirable to maintain a behavioural link between a program and its offspring. Teller[Tel94b] makes the point that the success of standard GP is due to this implicit assumption when evolving functions (by functions he means representations without iteration and memory), however when the GP system is extended to be able to evolve algorithms, the landscape of the space is highly discontinuous. A small change in a program can produce huge changes in its behaviour. In terms of GP terminology, the genotype is the program and the phenotype is the functionality. Ideally genetic operators will generate programs with a similar phenotype to the parent program.

4 Turing Compete Landscapes

GP moves around the search space using XO and mutation operators. If applied without considering the representation they can produce syntactically invalid programs. However, most practitioners design operators, or interpretations of the representation, to prevent this occurring. Perhaps more importantly, is that the genetic operators may not produce the required effect. The essence of XO is to take a 'contiguous chunk of material' from one individual and exchange it with a contiguous chunk of material from another individual. The motivation for mutation is simpler than XO, but again simply applying it without considering the representation can produce undesirable mutations.

The landscape metaphor, often used in Evolutionary Computation, is realized in GP using the notion of graphs (chapter 2 of [LP02]). Nodes in the graph represent programs. Links between nodes indicate that one program can be reached from the other by one application of a genetic operator. If probabilistic operators are used, the links can be labeled with the probability that the transition occurs. If XO operators are used, the probabilities are also time varying as the probability of the transition will depend on the current population. The connectivity of the landscape is defined by the representation and the genetic operators. If a different representation or a different operator is used, this will change the connectivity of the landscape. We will call a landscape associated with a TC representation (and specified operator) a TC landscape. There are many equivalent TC representations, and therefore, if suitable operators are chosen for each representation, the landscapes defined will be identical. Hence a landscape has a universal nature, independent of the representation and operators which define it (i.e. a TC landscape constructed using one TC representation can be constructed using any TC representation). Let us consider a landscape defined by a representation and a mutation operator.

THEOREM: Given two TC representations, R and S, and a mutation operator, $mutR$ that acts on programs represented in R, there exists an identical mutation operator $mutS$ that can be applied to programs in S producing an identical landscape.

PROOF: If R and S are TC representations, there exists emulators, $emRS$ and $emSR$ that allow programs expressed in one representation to be expressed in the other (this is a direct consequence of them being Turing Complete). Suppose we have a program $R^{'}$ expressed in R and it is mutated giving $R^{''} = mutR(R^{'})$. This operation can be emulated in S by $mutS(S^{'}) = emSR(mutR(emRS(S^{'})))$.

The landscape defined by representation R and operator $mutR$ can be emulated in representation S (see Figure 2). This result simply says that it is not worth considering the suitability of one representation compared to another, but rather the representation along with operators i.e. the landscape. For example, one may consider a tree based representation to be more suitable than a linear representation, however this theorem shows that whatever tree based mutation operator is proposed, there exists an operator that can be applied to the linear representation defining an identical landscape. For example, we could evolve a Turing Machine representation or a standard GP with indexed memory, but ultimately whatever landscape is defined by the Turing Machine representation and the operator we use, there will exist an operator which acts on the standard GP with indexed memory representation, which produces and identical landscape. While this theorem says that equivalent landscapes can be defined with different representations, it may be easier to express the landscape in one representation compared to another. The theorem applies to deterministic mutation, but can simply be extended to include stochastic mutation and XO.

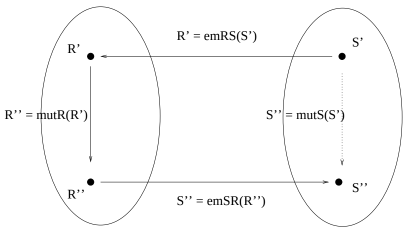

Figure 2: The ellipse on the left (right) is the space of all programs represented in R (S). The operator $mutS$ can be emulated by first emulating program S' in representation R, second apply $mutR$ and third emulating back to the original representation. Thus identical landscapes can be defined in both representations

Next we briefly consider composition as this forms the basis for our proposed XO operator. Programs describe the partially recursive functions, obtained by the base functions, zero, successor and projection, and three processes of building functions from these, namely composition, recursion and unbounded minimization. Here we will concentrate on composition. If $f(x)$ and $g_1(x), ..., g_n(x)$ are computable then $h(x) = f(g_1(x), ..., g_n(x))$ is also computable. $h(x)$ is said to be constructed by composition. The outputs of each g function are the inputs to f (see figure 3). Moreover, if f and $g_1, g_2, ..., g_n$ are total, h is total. The converse however is not true; if h is total we cannot deduce that f and $g_1, g_2, ..., g_n$ are total. In terms of programs, if a program halts on all inputs it corresponds to a total function (i.e. defined on all inputs). If a program does not halt on all inputs it corresponds to a partial function (i.e. undefined on some inputs). Composition corresponds directly with constructing programs from smaller programs, which is closely related to the building block hypothesis.

5 Proposed Representation and Crossover Operator

In this section we define a module and outline some of its features. The functional requirements of XO are described. Finally, it is stated how these functional requirements avoid the issues raised in section 3.

A module can be defined as a function defined in terms

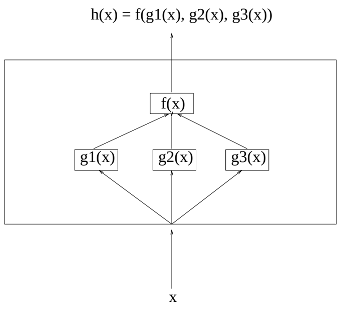

Figure 3: Function $h(x) = f(g_1(x), g(x)_2, g_3(x))$ is constructed by feeding in the outputs of $g_1(x)$, $g_2(x)$ and $g_3(x)$ into $f(x)$. The large rectangle represents $h(x)$ and the smaller rectangles represent $f(x)$,$g_1(x)$, $g_2(x)$ and $g_3(x)$.

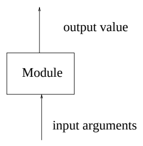

Figure 4: A module and can be thought of as a black box. It takes either none, one or many arguments (shown by an arrow entering the box) and returns a value (shown by an arrow leaving the box). Internally the module may consist entirely of primitives or other modules (e.g. see figure 3)

of primitives (i.e. the function and terminal set) or previously defined modules. A module takes a number of arguments (maybe none, one or many) and returns a value (see figure 4), it can be thought of as a mapping from inputs to output. One module may call another module, passing arguments and returning a result. In this way more complex modules can be constructed by calling simpler modules. This is the same idea that is used with automatically defined functions (ADFs) [BNKF98]; initially ADFs are constructed only from primitives but later new ADFs can be constructed from previously defined ADFs. (i.e. previously defined ADFs can be called just as if they were primitives) (see figure 6). For a discussion of modularity see [Woo03],

Both representation and genetic operators need to be considered together as these define the landscape of the search space. Consideration of the building block hypothesis, the intuitive idea of XO, composition of functions, and the halting problem leads to the representation and XO operator described in this section. The operations we want to apply can be expressed either in terms of a representation or in terms of the functions they compute. Here, the genetic operators are illustrated in terms of functions as this is the main idea we want to convey. Consider a function f, which takes three arguments and g_1, g_2, g_3 each which take one argument. Given two functions,

$$f(g_1(x), g_2(x), g_3(x))$$

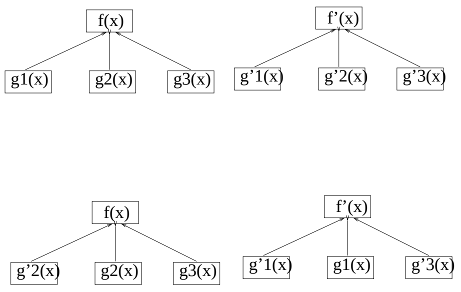

Figure 5: The functions before (above) and after (below) crossover. The functions $g_1(x)$ and $g_2^{'}(x)$ have been exchanged.

and

$$f^{'}(g_1^{'}(x), g_2^{'}(x), g_3^{'}(x))$$

are operated on by a XO operator to produce

$$f(g_2^{'}(x), g_2(x), g_3(x)),$$

and

$$f^{'}(g_1^{'}(x), g_1(x), g_3^{'}(x))$$

Here $g_1(x)$ and $g_2^{'}(x)$ have been exchanged.

XO exchanges a randomly selected subfunction from one function for a a randomly selected subfunction from another function. Note that in the case above f and g_1, g_2, g_3 maybe either functions from the function set or modules (i.e. functions constructed by composition from the function set). This describes our functional requirements, and one implementation of this is shown in figure 7.

Mutation is not considered in this paper. It could operate either at the level of modules, replacing a module with a randomly generated module, or could operate at the 'submodule' level, mutating either single instructions or groups of instructions within a module.

The issues raised in section 3 and how this model overcomes them are now stated. Global memory interference is avoided as each module does it processing locally, there is no global memory in the model. XO does not disrupt modules as it operates at the level of modules, it cannot separate instructions and moves modules about with no regard for their origin. The probability of producing a non halting program is reduced as the modules chosen for XO are more likely to represent total functions (more is said about this in section 6). As the XO operator respects modules, it is more likely that useful building blocks may propagate over the generations, rather than having their functionality destroyed. Finally, semantic links are maintained as subfunctions are swapped, hence, XO and mutation can only produce syntactically valid programs. No obscure methods of interpretation need to be invented (as with the Push language[SR02]) to avoid the problems of stack underflows or overflows. This model of local processing is illustrated in figure 6 where each module only communicates with other

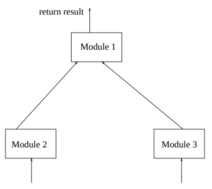

Figure 6: A modular architecture. A program is composed of modules, each does it processing locally (i.e. it cannot write to global memory, only local memory), communicating with the rest of the program by being passed arguments and returning a value. Here modules 2 and 3 receive input, process it and pass their output to module 1. One way to implement this is shown in figure 7

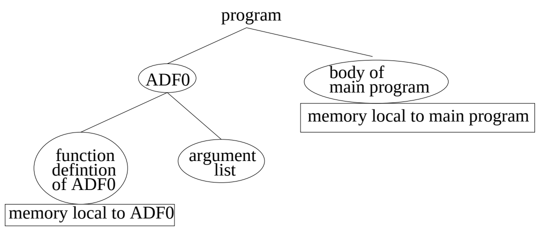

Figure 7: This figure shows how modules could be implemented, here GP with *local* indexed memory is used. If a module is moved to a different location it will perform the same function.

modules by receiving arguments and passing values.

It should be emphasized that what is described above is precisely what standard GP does, subtrees in an individual are replaced with other subtrees, which corresponds exactly to the idea of composition. When a subtree is moved by XO to a new location it performs the same function it was before XO. However, this is not what happens with GP with indexed memory, as the memory is global and so suffers from global memory interference (i.e. when a subtree is moved by XO to a new location it will not perform the same function it was before XO). A model which would realize the above operators is GP with *local* indexed memory with ADFs (see figure 7).

6 Discussion

Some GP practitioners have started to include modularity, memory and iteration into standard GP. Often this includes extending the representation by adding automatically defined functions, external indexed memory or an iterative construct. These features are implicitly present in any TC representation. The interesting question is how to explore the space of TC programs. It makes no sense to talk about a good TC representation for TCGP, as what can be done in one representation can be achieved in any other TC representation. However, it may be easier to express a genetic operator for one representation and difficult to express it for another (e.g. expressing XO at modular boundaries for a Turing Machine representation is possible in principle but more involved in practice). Here we propose a representation where module boundaries are explicit.

It is appealing to equate building blocks with modules, however there are a number of problem with this. Firstly it is very difficult to assign a fitness value to a module based on its performance when it is part of a larger program (this problem is not addressed in this paper). Secondly, the larger a module is the more likely it is to be disrupted by XO. In the examples in section 2 the XO points are chosen with uniform probability across the representation, and no attempts are made to identify structure within the program, thus the larger a module, the more likely it is to be disrupted. This is avoided by using a representation where the modules are explicit and XO operators *only* at the level of modules.

One problem with TCGP is that many genetic operators destroy the semantic links between parents and offspring. In this paper, careful consideration has been given to the reasons why. The problem of global memory interference was introduced and this led to the proposal that modules must do their memory processing locally, having arguments passed to them and returning a value, rather than referring to global memory. The problem of program flow was also discussed, and the disruption XO can cause if instructions in one potential module are separated by a XO operation. This suggested, along with consideration of construction of computable functions by composition and the building block hypothesis, that modules should be moved around as complete independent units.

The halting problem is a central issue for TCGP. If an upper limit is not imposed on the execution time of a program in the population, GP would end up waiting indefinitely for non-terminating programs. Any reasonable fitness function should favour termination over non termination (within the limit imposed), typically assigning the lowest fitness value to non-terminating programs. This provides an evolutionary pressure towards halting programs. As stated earlier, the composition of terminating programs will produce a terminating program. The converse is not true; observing that a program halts (on the test cases) does not imply that all the modules in that program halt. However, with a XO operator that respects the functionality of the modules, along with a fitness function that favours halting programs, halting modules are more likely to survive and propagate through the population over a number of generations. Hence we are more likely to produce halting programs. The upper limit cannot be removed as there will always be a chance that non-terminating programs will be produced, however this XO operator will reduce the chances of producing a non-terminating program in the next generation, and therefore reduce the amount of time waiting for non-terminating programs. There is no guarantee that mutation of a halting program will produce a halting program, but mutation should still be used as it is an important method of introducing new genetic material into the population, preventing premature convergence. Hence as evolution progresses the mutation rate should be lowered in comparison with the XO rate, which is what most practitioners do anyway.

It is productive to consider three different but very similar TC representations, namely GP with indexed memory. GP with indexed memory and ADFs, and GP with *local* indexed memory and ADFs. These illustrate the differences between TC representations, namely a TC representation, a TC representation with explicitly defined modules, and a TC representation with *local* memory and explicitly defined modules. Firstly, GP with indexed memory is Turing Complete, however it is argued that this is not a suitable representation to evolve because of the problem of global memory interference. Secondly, GP with indexed memory and ADFs is also Turing Complete. ADFs do not add any expressiveness to the representation. What ADFs do achieve is that they make modules explicit and XO needs to identify modules, as it operates at this level. Finally consider GP with *local* indexed memory and ADFs. When XO exchanges subtrees, it is unlikely that they will be performing the same function they were performing before XO, due to global memory interference. Hence only memory local to each ADF is allowed. Instructions in a given ADF can only read and write to local memory associated with that ADF.

All TC representations are implicitly capable of expressing modularity. If the representation is TC, the functions it can represent correspond to primary recursive functions and this implies they are capable of modularity. However it may be difficult to identify modules in any TC representation. If we provide an explicit method to express modules, XO can readily identify what portions of code it can exchange. It should be stated that while there is a way to explicitly express modules, there will also be implicit modules. For example, within a single explicitly defined module, implicit modules will also exist.

The output of GP is often novel and messy making its interpretation problematic. As GP starts to scale to larger and larger problems, the job of interpretation can only get more difficult. The XO operator presented provide a more structured approach resulting in clearly defined modules. As programs are evolved in a modular fashion this makes the analysis potentially easier as each module can be interpreted in isolation.

7 Summary

Standard GP is often not concerned with the evolution of TC representations, however recently researchers begun to extend standard GP in an attempted to evolve TC representations. This paper reviews a number of TC representations and genetic operators used to manipulate them. We point out a number of potentially undesirable effects result from these representations and their genetic operators. Firstly, there is the problem of global memory interference. If a piece of code is moved to another location in a different program by XO, it may perform a different function depending on the state of the global memory. Secondly, standard XO can disrupt modules by separating instructions which were acting together.

A novel XO operator is introduced which operates at the level of (explicitly defined) modules avoiding these problems. The first problem is avoided by only having memory, which is local to an explicit module. The second problem is avoided by only allowing XO to operate at the level of modules which are explicit in this representation, as opposed to implicit in some models (e.g. Turing Machines and assembly language type representations where modules exist but are not immediately obvious). Hence modules are moved around by the XO operator as a functional units, performing the same function they were performing before being moved by the XO operator. Each module is passed a number of arguments, which are processed locally (i.e. a module only has access to local memory), and returns a value.

This XO operator corresponds to the construction of functions by composition, one of the operations used in the theory of recursive functions. It is stated that by using a XO, which is essentially a composition operator, programs that are evolved are more likely to be self-terminating (self-terminating meaning that the program terminates within the upper bound set by the user). If a module appears in a number of self-terminating programs, then it is likely that the module is also self-terminating. Provided the fitness function assigns a high fitness value to halting programs, and a low value to non-terminating programs (i.e. which have to be halted externally as they have reached the upper bound on execution time), there will be an evolutionary pressure towards self-terminating programs. As programs are observed to halt, the modules within them are more likely to propagate through the population with the use of this XO operator. The same cannot be said about mutation operators (i.e. we cannot predict if a self-terminating program will still self-terminate after mutation) and therefore it is suggested that the mutation rate is reduced during the evolution to increase the chances of evolving a self-terminating program using this XO operator.

8 Acknowledgments

Xin Yao, Jon Rowe, Stefano Cattani,

Bibliography

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998.

[Cra85] N. L. Cramer. A representation for the adaptive generation of simple programs. In *International Conference on Genetic Algorithms and Their Applications.*, pages 183–187, July 1985.

[Hue98] Lorenz Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 158–166, San Francisco, CA, USA, 1998. Morgan Kaufmann.

[LP02] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[NB95] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[NBF99] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[SR02] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[TA96] Julio Tanomaru and Akio Azuma. Automatic generation of turing machines by a genetic approach. In Daniel Borrajo and Pedro Isasi, editors, *The First International Workshop on Machine Learning, Forecasting, and Optimization (MALFO96)*, pages 173–184, Gatafe, Spain, 10–12 1996.

[Tan93] Julio Tanomaru. Evolving turing machines from examples. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, volume 1363 of *LNCS*, Nimes, France, October 1993. Springer-Verlag.

[Tel94a] Astro Teller. The evolution of mental models. In *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[Tel94b] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press.

[Tel94c] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA, 27-29 1994. IEEE Press.

[VR01] Edgar E. Vallejo and Fernando Ramos. Evolving turing machines for biosequence recognition and analysis. In Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming: 4th European conference*, volume 2038 of *LNCS*, pages 192–203, Berlin, 18-20 April 2001. Springer.

[Woo03] J. R. Woodward. Modularity in genetic programming. In *Genetic Programming, Proceedings of EuroGP 2003*, Essex, UK, 14-16 2003. Springer-Verlag.