Turing Completeness in the Language of Genetic Programming with Indexed Memory

Astro Teller
4019 Winterburn Ave
Pittsburgh, PA 15207

Carnegie Mellon University
Department of Computer Science
astro@cs.cmu.edu

Abstract: Genetic Programming is a method for evolving functions that find approximate or exact solutions to problems. There are many problems that traditional Genetic Programming (GP) cannot solve, due to the theoretical limitations of its paradigm. A Turing machine (TM) is a theoretical abstraction that express the extent of the computational power of algorithms. Any system that is Turing complete is sufficiently powerful to recognize all possible algorithms. GP is not Turing complete. This paper will prove that when GP is combined with the technique of indexed memory, the resulting system is Turing complete. This means that, in theory, GP with indexed memory can be used to evolve any algorithm.

I. Introduction

The nature of this paper is theoretical. There are a wide variety of issues concerning the evolution of algorithms (as opposed to the functions of GP). These issues will only be touched on briefly at the end of the paper. This paper makes no claims on the actual practicality of evolving an arbitrary algorithm, this paper is about what is theoretically possible. Also, it should be noted that this proof was chosen for its simplicity and does not represent how algorithms are likely to evolve.

This paper assumes very little familiarity with either GP or indexed memory. A complete introduction to the topic of traditional GP can be found in "Genetic Programming: On the Programming of computers by means of natural selection." [Koza 1991] For a discussion of indexed memory see "The Evolution of Mental Models." [Teller 1994] What is needed for a full understanding of the proof in this paper is some knowledge of Turing machines and Turing completeness. For a primer on this try "An Introduction to Automata and Complexity Theory." [Hopcroft 1979]

This paper will strive to leave the reader with a solid sense of the computational completeness of the described system and a glimmer of the unexplored complexities this completeness introduces into the field of evolutionary computation.

II. Genetic Programming

Genetic Programming is a strategy for evolving functions that perform well on assigned tasks. These evolved functions are represented in GP as Lisp-like expressions consisting of non-terminals (atomic functions) and terminals (e.g. variables and constants). Simple GP functions might look like:

- **(* 2 (IF (= 4 x) ELSE (x/(- 4 x))
  THEN (cos (exp (+ x (* (+ 5 6) x))))))**

- **(exp (/ 9 x) (- (+ (+ 9 8) x) 12))**

Given a sufficiently expressive set of mathematical functions, variables, and constants, this type of "language" can represent many desired functions, such as $\mathbf{x^3*cos(2*x)}$. This function might evolve in the following form:

- **(* (* x (* x x)) (cos (* x 2)))**

The process of finding a GP function that is a good or perfect approximation to the target function can be summarized as follows. The population is initialized with a set of randomly generated individuals. Each member from the current pool of functions is tested to determine its error on some task. A new pool is created in which the functions with lower error have higher representation. The new pool is then subjected to various kinds of recombination. Two popular varieties are mutation and crossover. Over time the most successful individuals in the population become increasingly accurate approximations of the correct solution to the task.

This language is not powerful enough to express many algorithms. For example, there is a simple procedure for checking whether a string is of the form $\mathbf{0^n1^n}$, but there is no way to make such a procedure in this kind of Lisp-like structure. There is no mechanism for variable length strings to be shown to the function and no way for the function to iterate an arbitrary number of times.

We could encode the $\mathbf{0^n1^n}$ problem by giving the GP function a series of tests. The variable **x** could be used to give the current character (**x** = {0,1,2}) where 2 means "end of string". The fitness (average success) of the function could be based on whether it returned a positive or negative number when it received the 2. Here the GP environment might interpret positive numbers as "in the form $\mathbf{0^n1^n}$" and negative numbers as "not in the form $\mathbf{0^n1^n}$". But since each function has no access to the previous inputs and the previous inputs (where x was equal to 0 or 1) determine the correct response when the end of string is reached, this problem is unsolvable for traditional GP. In general, solutions to these kind of problems requires something extra: **memory**.

III. Indexed Memory

Indexed memory is a simple addition to the GP paradigm. **Read** and **Write** are added as new non-terminals in the language. Each GP function is given access to its own array of integers, indexed over the integers. The expression (Read 5) returns an integer in just the same way that (* 5 17) does. In general, (Read X) returns the integer stored in memory position X, where X is an integer. And (Write -19 101) returns the old value of memory position 101 and has the side effect of changing the value of memory position 101 to be -19. A simple GP function that utilized indexed memory might be:

(IF (< (Read 4) (Read (Read (+ x 4))))
THEN (Sqrt (+ (* x 8) 1)) ELSE (* x x))

Using indexed memory, a GP function could save past inputs and use them in a process like the $\mathbf{0^n1^n}$ problem. For each 0 or 1 that the function received, it could store the value in a memory location. Then, when **x** = 2, it could decide whether to return a positive or negative number based on the values stored in its memory. However, since the series of 0's and 1's can be arbitrarily long and the GP function with indexed memory is of fixed size, there is a length of strings above which the function cannot always classify them correctly. So memory is necessary, but there is another element missing from traditional GP: **iteration**.

IV. Notation

GP plus indexed memory (GP+IM) has a key ingredient that was missing from GP. But, for the reason just given, the procedure for checking $\mathbf{0^n1^n}$ cannot be represented with any GP function, even if it contains Read's and Write's. However, writing such a $\mathbf{0^n1^n}$ checker using iteration is possible.

This paper will prove that any algorithm can be written in the following language:

Repeat
Evaluate < GP+IM function >
Until
< Some specific state of the memory >

For the rest of the paper the term "GP+IM machine" will be used as an abbreviation for this Repeat-Until loop that evaluates the combination of genetic programming and indexed memory. The term "GP+IM function" will refer to the function inside the Repeat loop that is a traditional GP function with the addition of indexed memory.

To keep the proof simple, I will use the set of non-terminals described below. **X**, **Y**, and **Z** are all either terminals (constants and variables) or further sub-functions made up of these same non-terminals.

(IF X THEN Y ELSE Z): this returns Y if X is non-zero otherwise it returns Z . Notice that the test is evaluated and then either Y or Z is evaluated, **but not both**. This is important because **Write** has a side effect.

(= X Y): this returns 1 if the two arguments evaluate to equal values and 0 otherwise

(AND X Y): this returns 0 if either argument is zero; otherwise it returns 1. Notice here that AND evaluates **both** arguments before returning a value.

(ADD X Y): This returns the addition of X and Y.

(SUB X Y): This returns X minus Y.

(Read X): This returns the value of the memory element specified by indexing the memory array by the value of the argument (i.e. (READ X) returns Memory[X]).

(Write Y X): This returns the value of the memory element specified by indexing the memory array by the value of the second argument (Memory[X]) and **then** puts the value of first argument into the memory position indexed by the second argument (Memory[X] <=== Y).

V. The Proof

This section will prove that the GP+IM machine is Turing complete. There are several ways this could be shown, but the approach taken here will be to assume that there is some arbitrary Turing machine(TM) that we want. This section will show how to construct a GP+IM machine that duplicates the functionality of this arbitrary TM. This having been done for an arbitrary TM, this satisfies the claim that the GP+IM machine is Turing complete.

Since any GP+IM function might be generated randomly in the first generation, it is sufficient to show that the desired GP+IM function exists. The following statement cannot be over-stressed. **This is not a proof about evolution. This is a proof about the language in which genetically evolved functions (and now algorithms) are usually written.** As mentioned before, the issue of how likely one is to generate such a function at random or through evolution is the topic of another paper. [Teller, to appear]

Suppose that there is some algorithm T. This algorithm can be expressed as a TM. Any TM, in turn, can be expressed as a set of seven items:

- Q a finite set of allowable states of the machine
- Γ a finite set of allowable tape symbols
- B a special (Blank) tape symbol (member of Γ)
- Σ a finite set of allowable input symbols (subset of Γ)
- q a start state (an element of Q)
- F a finite set of final states (subset of Q)
- Δ a transition function (Q x Γ ---> Q x Γ x {Left, Right})

So the target Turing machine T has a $\{Q_T, \Gamma_T, B_T, \Sigma_T, q_T, F_T, \Delta_T\}$. To prove that GP+IM is a Turing Complete system it is sufficient to construct $\{Q_{GP+IM}, \Gamma_{GP+IM}, B_{GP+IM}, \Sigma_{GP+IM}, q_{GP+IM}, F_{GP+IM}, \Delta_{GP+IM}\}$ and to prove the behavioral isomorphism between them. The subscript "GP+IM" will be shortened to "G" on the GP+IM machine under construction.

As the reader will soon see, the array of integers available to **Read** and **Write** is infinite in size. This is not possible in practice. However, because any proof of Turing completeness requires infinite memory, this is no limit on the theoretical statement of the following proof. In fact, this infinite array can be simulated by a linked list of elements that initially holds all the non-blank input and grows by one element whenever a new memory element is referenced. In the construction below, one memory position may have to hold an arbitrarily large integer. This arbitrarily large number can be simulated with a linked list in a similar way.

Before the construction and actual proof, it will help to sketch what is going on. We are going to make a GP+IM function that will be the Δ_G for the machine we are constructing. The array of integers that this function has access to, through indexed memory, will serve as the tape for the machine we are constructing. The current "state" of the machine we are constructing will be stored in Memory[0]. The index of the array position that is the "current tape position" will be kept in Memory[1]. In a sense Memory[1] will track the "tape head" for us. Every other element in the array will effectively be a "tape element." Since the memory elements 0 and 1 are not part of the "tape" there is a little bit of work done in the construction below (and in the sample code) to make sure that when the "tape head" for the GP+IM machine moves left or right, that Memory[1] skips from 2 to -1 or from -1 to 2, respectively. With the exception of Δ_G, the constructions below will be picking a subset of the integers to stand for the different tape symbols and states that the machine we are constructing can have.

The construction of Q_G

Suppose that the magnitude of Q_T is Q_{mag}. We can assign Q_{mag} distinct integers to be the set Q_G. (e.g. $\{101 \ldots Q_{mag} + 100\}$ This set will be the set of allowable states for the GP+IM machine.

The construction of q_G

The members of Q_T can be put into a one-to-one relation with the integers in the set Q_G**.** There is a unique member of Q_T that is the start state for the target machine. Using the one-to-one relation, q_G is chosen to be the integer from Q_G that corresponds to the element of Q_T that is q_T. This integer will be the start state for the GP+IM machine.

The construction of F_G

Using the same one-to-one relation, the subset of integers from Q_G that correspond to the subset of Q_T which is F_T will be chosen as the set F_G. These integers will be the final states of the GP+IM machine.

The construction of Γ_G

Suppose the size of Γ_T is Γ_{mag}. Γ_{mag} distinct integers (none of which are in Q_G) can be chosen to stand for the tape symbols. (e.g. the set $\{Q_{mag} + 101 \ldots Q_{mag}+100+\Gamma_{mag}\}$) The integers in Γ_G will be the legal tape symbols for the GP+IM machine.

The construction of B_G

A one-to-one relation can be established between the elements of Γ_T and the integers of Γ_G. B_G will be chosen to be the integer in Γ_G that corresponds to the B_T that is a member of Γ_T. This new integer will be the blank symbol for the GP+IM machine.

The construction of Σ_G

Using this same relation between Γ_T and Γ_G, the subset of integers from Γ_G that correspond to the subset of Γ_T which is Σ_T will be chosen as the set Σ_G. Σ_G will be the legal input tape symbols for the GP+IM machine.

The construction of Δ_G

Δ_T is a transition function, so it can always be organized as follows:

State#	TapeSymbol		NewState#	Write	Move
s1	t1	==>	s1	t0	LEFT
s1	t0	==>	s1	t0	RIGHT
s2	t1	==>	s1	t0	LEFT
s2	B	==>	s2	t1	RIGHT
s3	t1	==>	s3	t0	RIGHT
s3	t0	==>	s1	t0	LEFT
(and so on ...)					

This list can have at most $|Q_T \times \Gamma_T|$ lines in it. Since Q_T and Γ_T are finite by definition, this list will always be finite in length. For example, suppose Q_G becomes the set {101, 102, 103 ...199} and Γ_G becomes the set {200, 201, 202, ...299} (where B_G = 299). Then the above list representing an example Δ_T can be rewritten for Δ_G as:

```
(IF (AND (= (Read 0) 101)
         (= (Read (Read 1)) 201))
 THEN
    (AND (Write 101 0)
           (AND (Write 200 (Read 1))
                  (IF (= (Read 1) 2)
                    THEN  (Write -1 1)
                    ELSE  (Write (Sub (Read 1) 1) 1))))
 ELSE
    (IF (AND (= (Read 0) 101)
             (= (Read (Read 1)) 200))
      THEN
        (AND (Write 101 0)
               (AND (Write 200 (Read 1))
                      (IF   (= (Read 1) -1)
                       THEN (Write 2 1)
                       ELSE  (Write (Add (Read 1) 1) 1))))
      ELSE
        (and so on ...)
```

In general, if there are **k** cases in the list for the Δ_T then **k** nested IF-THEN-ELSE's can be constructed that perform the same actions on the GP+IM machine that the Δ_T function performs on the target machine. The i^{th} transition in Δ_T can be written as <x,y> ==> <z,u,v>, where x and z are members of Q_T , y and u are members of Γ_T , and v is member of {Left, Right}. After the constructions described above, x,y,z, and u all have integer representations for each transition in Δ_T. So the i^{th} IF-THEN-ELSE in the constructed nesting for Δ_G, corresponding to the i^{th} transition of Δ_T, will be like figure 1 if the tape head moves left in the target machine and like figure 2 if the tape head moves right in the target machine. Some of the code in figures 1 and 2 is used to force Memory[1] to decrement from 2 to -1 and increment from -1 to 2, in order to skip over the two non-tape elements: Memory[0] and Memory[1].

After the **k** nested IF-THEN-ELSE's there is a last ELSE that must contain something. The GP+IM function will only reach that point when none of the **k** transition preconditions are satisfied. This is exactly the case in which the TM halts. So for the construction of Δ_G the action of this last ELSE should be to halt the GP+IM machine. This can be done as shown in figure 3. Remember that the GP+IM machine is a Repeat-Until loop that has as its termination condition (the Until clause) some particular state of the memory. It is this condition that this last ELSE will satisfy and that is why the GP+IM machine will halt.

```
(IF (AND (= (Read 0) x)
         (= (Read (Read 1)) y))
 THEN
    (AND (Write z 0)
         (AND (Write u (Read 1))
              (IF  (= (Read 1) 2)
               THEN   (Write -1 1)
               ELSE   (Write (Sub (Read 1) 1) 1))))
 ELSE .....
```

Figure 1.

```
(IF (AND (= (Read 0) x)
         (= (Read (Read 1)) y))
 THEN
    (AND (Write z 0)
         (AND (Write u (Read 1))
              (IF  (= (Read 1) -1)
               THEN   (Write 2 1)
               ELSE   (Write (Add (Read 1) 1) 1))))
 ELSE .....
```

Figure 2.

```
(ELSE (Write 1 1))))
```

Figure 3.

In figure 3 the tape head is set to point to itself. This is a head position that will never happen during the computation so this state (Memory[1] == 1) will be the test for completion of the Repeat-Until loop for this GP+IM machine. (i.e. Repeat <some GP+IM function> Until Memory[1] ==1)

The terminal set (legal list of constants and variables) for this proof is the set of constants which are the integers in the union of the sets Q_G, Γ_G, and {-1,0,1,2}.

A GP+IM function has been constructed which is the nested IF-THEN-ELSE's just described. The GP+IM machine will start with the integer q_G in Memory[0] and Memory[1] holding the index of the correct initial position in the array for the machine's "tape head" (discussed below).

A TM can be said to accept its input if it halts in one of the final states. This means that the target machine we are trying to duplicate accepts its input when and only when it is started on that input and halts after a finite number of transitions in one of the states F_T. For the machine we are constructing, it accepts its input when and only when it is started on that input and halts after a finite number of transitions with the value in Memory[0] being a member of the set of integers F_G.

Equivalence Claim

For any input placed in the GP+IM memory array in the manner prescribed below, the GP+IM machine will accept the input when and only when the target machine accepts the isomorphic input on its tape.

The word isomorphic will appear quite often during the next two pages. There are three similarities going on in this construction and proof. The first is the similarity between the states of the target machine and the states of the machine we are constructing. These two sets are not identical, but there is an obvious one-to-one mapping between them. The second is a similar relation between the tape symbols for the target machine and the tape symbols for the machine we are constructing. The third is the behavioral similarity between what Δ_T does for the target machine and what Δ_G does for the machine we are constructing. So the term isomorphic here will be understood to mean equivalence under each of these mappings.

If the initial tape for the TM is

-	-	A	B	A	B	A	A	A	-	-

then memory must be set so that ...

	999	999	101	100	101	100	101	101	101	999	999	
....	i-4	i-3	i-2	i-1	i	i+1	i+2	i+3	i+4	i+5	i+6	....

... Memory[1] is set to i. This sets the initial "tape head" for the GP+IM machine to be the same as the target machine with equivalent tape (memory) to either side of the starting head position. (under the isomorphism that A==> 101, B==>100, Blank==>999) Though this proof avoids discussions of practicality, it is worth noting that the infinite storage needed for the infinite Blanks can be avoided by noticing that, by default, every new memory element never accessed before must contain a Blank (use some kludge for the initial input).

To prove this **Equivalence Claim** I need to show that the GP+IM function sets Memory[1] to 1 if and only if the Δ_T of the target Turing Machine we are duplicating has no legal transitions(i.e. halts) **and** that the GP+IM machine will be in a final state when and only when the target Turing machine is in a final state. In showing that both machines will always be in the same situation from (Q x Γ), we will have shown that, in particular, either both machines are in a final state, or neither are. To make the language of the proof simpler, we will assume that the two machines are being run synchronously. That is, they each make one transition at the beginning of each new unit of time. After **i** units of time, both machines will have made exactly **i** transitions (unless either has already halted).

<u>Inductive proof that all transitions preserve the isomorphic equality of the two machines.</u>

Base Case: Initially the target machine and the GP+IM machine and their tapes are in the isomorphic states by construction.

Inductive Hypothesis: Suppose that during the last **n** units of time, each machine has made exactly **n** transitions and that their states and their tapes are still isomorphically equivalent.

Proof that the **n+1** transition leaves both machines in isomorphically equivalent states:

1] Suppose that **i**th transition of the finite number of transitions in the list Δ_T will be the appropriate transition for the target machine at the beginning of time unit **n+1** and that this transition is <x,y> ==> <z,u,v> where x and z are members of Q_T , y and u are members of Γ_T , and v is member of {Left, Right}.

2] Since there is one IF-THEN-ELSE in Δ_G for each transition in Δ_T (by construction) and since the IF-THEN-ELSE tests are mutually exclusive (we will assume deterministic Turing machines without loss of generality), the IF-THEN-ELSE whose test succeeds will be an instantiation of figure 1 or figure 2 (depending on the whether the transition dictated by that situation moves the head left or right).

3] The concept of "adjacentness" on the target machine tape becomes, in this construction isomorphism, the same linear ordering except that -1 and 2 are "next to" each other and 0 and 1 are excluded (by construction).

4] (AND (= (Read 0) x) (= (Read (Read 1)) y)) is true when and only when <x,y> is the current situation. (by construction and definition of AND and READ)

5] (Write z 0) puts the integer z (which stands for the corresponding state by construction) into memory position 0, which is where the state number is kept. This effectively changes the GP+IM state to be z.

6] (Write u (Read 1)) puts the integer u (which stands for the corresponding new tape symbol) into memory position (Read 1). (Read 1) is the index of the memory element of the current tape symbol (by construction), so this puts the new tape symbol u into the current tape position.

7]
```
(IF      (= (Read 1) 2)
 THEN    (Write -1 1)
 ELSE    (Write (Sub (Read 1) 1) 1))))
```

This subtracts 1 from (Read 1) unless (Read 1) == 2, in which case it moves the tape to -1. This is exactly isomorphic to moving the tape head left one space (by construction and #3).

8]
```
(IF      (= (Read 1) -1)
 THEN    (Write 2 1)
 ELSE    (Write (Add (Read 1) 1) 1))))
```

This adds 1 to (Read 1) unless (Read 1) == -1, in which case it moves the tape to 2. This is exactly isomorphic to moving the tape head right one space (by construction and #3).

9] The meaning of <x,y> ==> <z,u,v> is that when we are in state x and y is the current tape symbol, we should move to state z, write u on the tape, and do v {Left, Right} From #2, #4,#5,#6,#7,#8 this is done by the GP+IM function (Δ_G).

Therefore all transitions preserve isomorphic equivalence between the two machines.

<u>Proof that both machines halt on the same transition.</u>

10] At the end of each transition the two machines are in isomorphically equivalent states from (Q x Γ) (proved above).

11] (Write 1 1) This sets Memory[1] to be 1, which is the condition for halting for the GP+IM machine (by definition of Write and by construction).

12] Memory[1] does not start equal to 1 (by construction) and could only be changed to 1 by a Write (definition of indexed memory).

13] In all **k** IF-THEN-ELSE transitions, Memory[1] is only incremented, decremented, set to -1, and set to 2 (by construction).

14] Increment and decrement never leave Memory[1] equal to 1 (From #7 and #8).

15] There is a (Write 1 1) in the last ELSE and nowhere else (by construction).

16] The last ELSE is executed when and only when there are no legal transitions in the target machine (by construction and #2,#4).

17] The target machine halts when and only when it has no legal transition to do (definition of Turing machines).

18] Memory[1] = 1 (i.e. GP+IM machine halts) when and only when the target machine halts (from #10 through #17).

Therefore on any input, the two machines will halt after the same number of transitions.

<u>Proof that the GP+IM machine accepts an input when and only when the target Turing machine accepts that input.</u>

After every transition, the two machines are in equivalent states from (Q x Γ) (proved above).

The target machine halts after the exact same number of transitions as does the GP+IM machine (proved above).

Therefore if the machines have halted, they will be in isomorphically equivalent states from (Q x Γ). In particular the target machine will be in state $Q_{T,i}$ and the GP+IM machine will be in state $Q_{G,j}$ (i.e. Memory[0] = $Q_{G,j}$) and $Q_{T,i}$ will be the pair to $Q_{G,j}$ in the one-to-one mapping used in the construction of the GP+IM machine.

Therefore $Q_{T,i}$ is a final state if and only if $Q_{G,j}$ is a final state.

Therefore the target machine accepts an input when and only when the GP+IM machine accepts the isomorphically equivalent input.

This proves the **Equivalence Claim.**

It has just been shown that given an arbitrary TM, we can construct an algorithm in the GP+IM paradigm that will, given the same input and initial tape head position, act exactly as the TM acts. That is one definition of Turing completeness. Therefore the language of genetic programming when combined with indexed memory is a Turing complete language.

VI. Discussion

The non-terminal set used in the proof is overly simple. The Δ_G function in the proof is orderly in a way that we know will, in practice, never occur in a randomly generated or evolved function. This does not mean that in order to insure Turing completeness in an experiment one has to use these exact non-terminals, hold one's breath, and hope for nested IF-THEN-ELSE's. As long as the non-terminal set **could** be used to construct a Δ_G function like the one shown above, the system under evolution is Turing complete. That is, every member of the population is a TM and there is no TM outside the language of the population individuals. **Repeat (ADD 1 (Read 61)) Until Memory[15]=0** is a TM, but not the most interesting of Turing machines, since it either halts immediately or runs on forever. But most Turing machines are like that. This notion is similar to saying that Shakespeare is expressible in some language **X**. That may be good to know, but that doesn't guaranty that any particular line from language **X** will be quality literature. Similarly, the language of a typewriter is expressive enough to capture Shakespeare. However, Shakespearean prose will, in practice, never arise from a monkey playing with the keys. Whether evolutionary computation will suffer from this phenomenon in the pursuit of evolving algorithms is still an open question.

When the time comes to evolve a population of GP+IM machines, a difficulty will arise. Each member of the population will be of the form **Repeat Evaluate <GP+IM function> Until <Some Test>**. To test the fitness of an individual will involve initializing its memory to something and then looping until the test is true. The fitness of the individual might be determined by examining its memory or through the result of some side-effecting terminals done before the test becomes true. The difficulty is that for most tests there is a danger that the test will never be true; some machines will run on forever. Worse, the Halting problem tells us that we cannot always be sure whether the program under examination will ever halt. Does this mean that we cannot really evolve GP+IM machines in practice ? The short answer is that know no one really knows yet. [Teller, to appear]

What this proof shows is that any TM can be reproduced in the GP+IM scheme. When you try to evolve them you may have to abort a test machine if it runs for a long time without terminating. This makes it hard or impossible to evolve some TMs. However, in practice the computers we all work on are not Turing complete, because they do not have infinite memory. Still, we find them to be a good approximation to a TM and even feel comfortable saying that they are capable of universal computation. In the same way, aborting a GP+IM machine that has run for a long time cuts off part of the search space but does not, in practice, remove the universality of GP+IM's expressiveness.

VII. Conclusions and Future Work

This paper has proven that the combination of GP and indexed memory is a system with a sufficient complexity to support any computation. This was shown by hypothesizing an arbitrary Turing machine and constructing a machine of the form:

Repeat
  Evaluate < GP+IM function >
Until
  < Some specific state of the memory >

and then proving that this machine had behavior equivalent to that of the target Turing machine.

The discussion stressed that the highly improbable nature of the GP+IM function under consideration was to keep the proof simple. The Turing completeness of the system does not depend on a convenient organization of the function. Rather, it is a property of the language in which that function is written. Similarly, an extension or change of the non-terminals or terminals used in the proof will maintain the Turing completeness of the system, as long as there is still a way to build each case of Δ_T into the GP+IM function.

This paper went to some length to emphasize the theoretical nature of this work. The fundamental completeness of the GP+IM system will be reassuring to some, but this proof is no substitute for a thorough investigation into how to make this possibility a practical reality.

Acknowledgments

The list of people who patiently listened to me rant and rave about this proof is too long to include here. However, Eric Siegel, Paul Teller, and Zander Teller deserve special mention for their help in making this paper readable.

Bibliography

Hopcroft, John et al. **Introduction to Automata Theory, Languages, and Computation**. Addison-Wesley Publishing Co. 1979

Koza, John R. **Genetic Programming: On the Programming of computers by means of natural selection**. The MIT Press. 1992.

Teller, Astro "The Evolution of Mental Models" Chapter 9. **Advances in Genetic Programming.** editor Kinnear, Kim. MIT Press. 1993

Teller, Astro "Genetic Programming, Indexed Memory, the Halting Problem, and Other Curiosities" Currently unpublished manuscript. Available on request.