On Turing complete T7 and MISC F-4 program fitness landscapes

W. B. Langdon and R. Poli
Department of Computer Science, University of Essex, UK
Technical Report CSM-445

19 December 2005

Abstract

We use the minimal instruction set F-4 computer to define a minimal Turing complete T7 computer suitable for genetic programming (GP) and amenable to theoretical analysis. Experimental runs and mathematical analysis of the T7, show the fraction of halting programs is drops to zero as bigger programs are run.

1 Introduction

Recent work on strengthening the theoretical underpinnings of genetic programming (GP) has considered how GP searches its fitness landscape [Langdon and Poli, 2002; McPhee and Poli, 2001; McPhee *et al.*, 2001; McPhee and Poli, 2002; Rosca, 2003; Sastry *et al.*, 2003; Greene, 2004; Poli *et al.*, 2004; Mitavskiy and Rowe, 2005; Daida *et al.*, 2005]. Results gained on the space of all possible programs are applicable to both GP and other search based automatic programming techniques. We have *proved* convergence results for the two most important forms of GP, i.e. trees (without side effects) and linear GP [Langdon and Poli, 2002; Langdon, 2002a; Langdon, 2002b; Langdon, 2003b; Langdon, 2003a]. As remarked more than ten years ago [Teller, 1994], it is still true that few researchers allow their GP's to include iteration or recursion. Indeed there are only about 50 papers (out of 4631) where loops or recursion have been included in GP. These are listed in Appendix B. Without some form of looping and memory there are algorithms which cannot be represented and so GP stands no chance of evolving them.

We extend our results to Turing complete linear GP machine code programs. We analyse the formation of the first loop in the programs and whether programs ever leave that loop. Mathematical analysis is followed up by simulations on a demonstration computer. In particular we study how the frequency of different types of loops varies with program size. In the process we have executed programs of up to 16 777 215 instructions. These are perhaps the largest programs ever (deliberately) executed as part of a GP experiment. (beating the previous largest of 1 000 000 [Langdon, 2000]). Results confirm theory and show that, the fraction of programs that produce usable results, i.e. that halt, is vanishingly small, confirming the popular view that machine code programming is hard.

The next two sections describe the T7 computer (see Figure 1) and simulations run on it, whilst Sections 4 and 5 present theoretical models and compare them with measurement of halting and non-halting programs. The implications of these results are discussed in Section 6 before we conclude (Section 7).

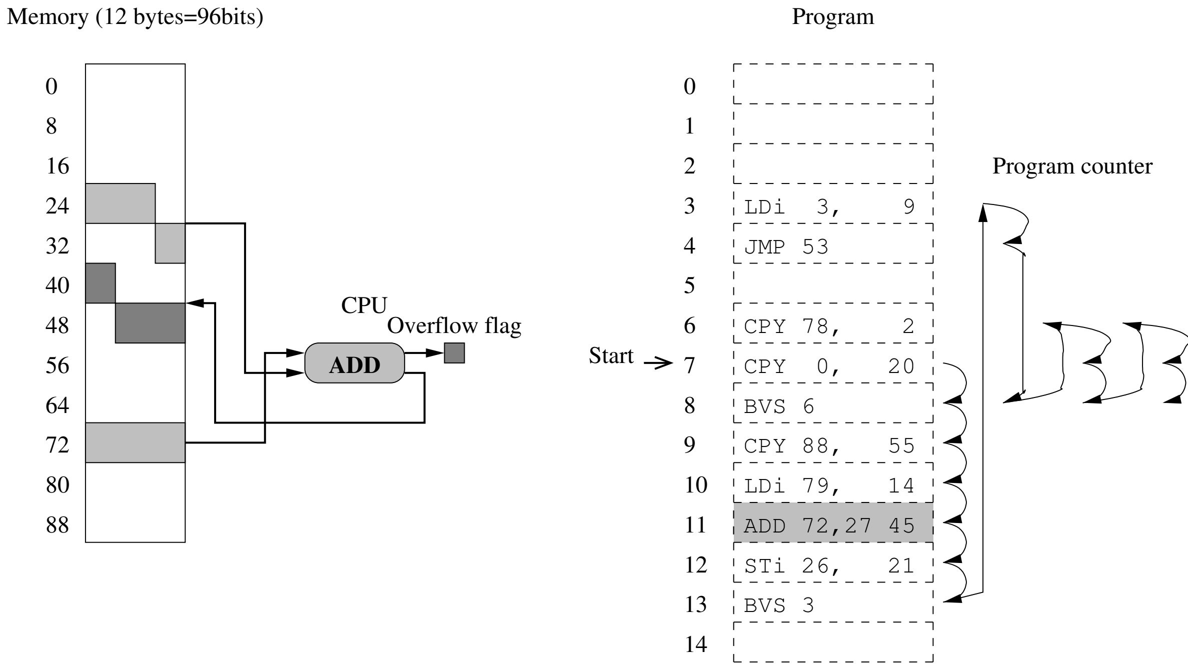

Figure 1: The T7 computer and sample program

2 T7 an Example Turing Complete Computer

To test our theoretical results we need a simple Turing complete system. Our seven instruction CPU (see Table 1) is based on the Kowalczy F-4 minimal instruction set computer `http://www.dakeng.com/misc.html` cf. Tables 2–4 in Appendix A. T7 consists of: directly accessed bit addressable memory (there are no special registers), a single arithmetic operator (ADD), an unconditional JUMP, a conditional Branch if oVerflow flag is Set (BVS) jump and four copy instructions. COPY_PC allows a programmer to save the current program address for use as the return address in subroutine calls, whilst the direct and indirect addressing modes allow access to stack and arrays.

In these experiments 8 bit data words are used, while a number of program addresses word sizes are used. In fact, in each run, the address words size is chosen to be just big enough to be able to address every instruction in the program. E.g., if the program is 300 instructions, then BVS, JUMP and COPY_PC instructions use 9 bits. These experiments use 12 bytes (96 bits) of memory (plus the overflow flag).

3 Experimental Method

There are simply too many programs to test all of them. Instead we gather representative statistics about those of a particular length by randomly sampling programs of that length. Then we sample those of another length and so on, until we can build up a picture of the whole search space.

To be more specific, one thousand programs of each of various lengths (30. . . 16 777 215 instructions) are each run from a random starting point, with random inputs, until either they reach their last instruction and stop, an infinite loop is detected or an

Instruction	*#operands*	*operation*	v set
ADD	3	A + B→C	v
BVS	1	#addr→pc if v=1	
COPY	2	A→B	
LDi	2	@A→B	
STi	2	A→@B	
COPY_PC	1	pc→A	
JUMP	1	addr→pc	

Each operation has up to three arguments. These are valid addresses of memory locations. Every ADD operation either sets or clears the overflow bit v. LDi and STi, treat one of their arguments as the address of the data. They allow array manipulation without the need for self modifying code. cf. Table 4. (LDi and STi data addresses are 8 bits.) To ensure JUMP addresses are legal, they are reduced modulo the program length.

Table 1: T7 Turing Complete Instruction Set

individual instruction has been executed more than 100 times. (In practise we can detect almost all infinite loops by keeping track of the machine's contents, i.e. memory and overflow bit. We can be sure the loop is infinite, if the contents is identical to what it was when the instruction was last executed.) The programs' execution paths are then analysed. Statistics are gathered on the number of instructions executed, normal program terminations, type of loops, length of loops, start of first loop, etc.

4 Terminating Programs

The introduction of Turing completeness into GP raises the halting problem, in particular how to assign fitness to a program which may loop indefinitely [8]. We shall give a lower bound on the number of programs which, given arbitrary input, stop, and show how this varies with their size.

The T7 instruction set has been designed to have as little bias as possible. In particular, given a random starting point a random sequence of ADD and copy instructions will create another random pattern in memory. The contents of the memory is essentially uniformly random. I.e. the overflow v bit is equally likely to be set as to be clear, and each address in memory is equally likely. (Where programs are not exactly a fraction of a power of two long, JUMP and COPY_PC addresses cannot completely fill the number of bits allocated to them. This introduces a slight bias in favour of lower addresses.) So, until correlations are introduced by re-executing the same instructions, we can treat JUMP instructions as being to random locations in the program. Similarly we can treat half BVS as jumping to a random address. The other half do nothing. We will start by analysing the simplest case of a loop formed by random jumps. First we present an accurate Markov chain model, then Section 4.2 gives a less precise but more mathematical model. Section 4.3 considers the run time of terminating programs.

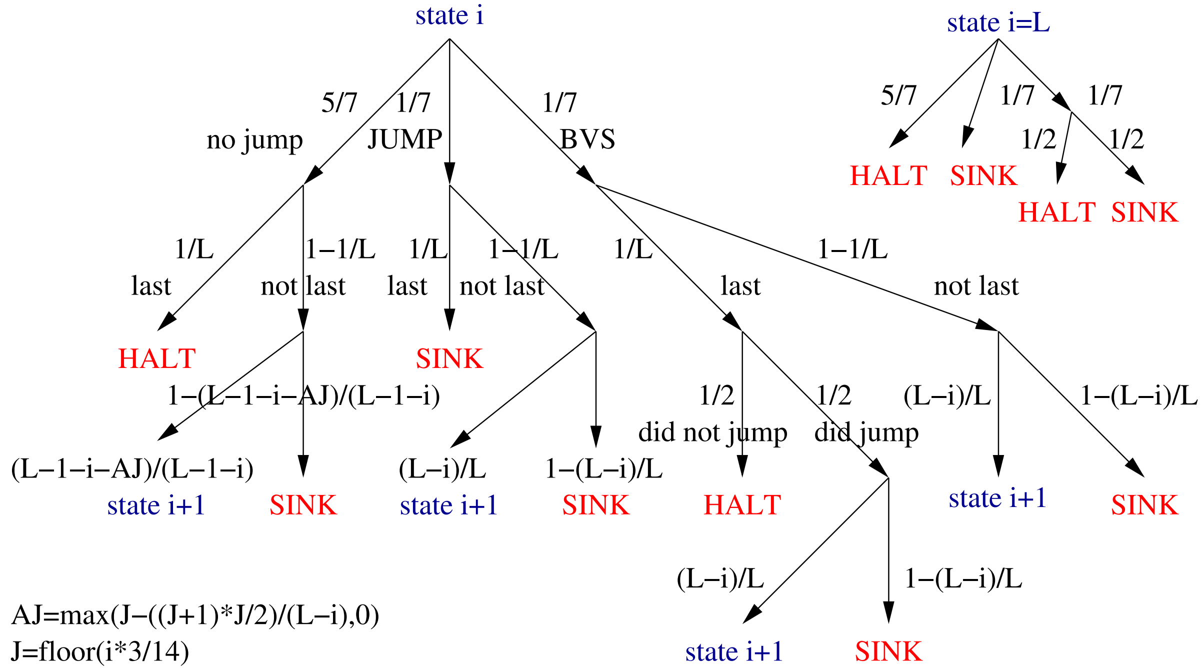

Figure 2: Probability tree used to create Markov model of the execution of random Turing complete programs. HALT indicates a terminating program, while SINK means the start of a loop.

4.1 Markov Chain Model of Non-Looping Programs

The Markov chain model predicts how many programs will not loop and so halt. This means it, and the following segments model, do not take into account those programs which are able to escape loops and do reach the end of the program and stop. As a program runs, the model keeps track of: the number of new instructions it executes, if it has repeated any, and if it has stopped. The last two states are attractors from which the Markov process cannot escape. State i means the program has run i instructions without repeating any. The next instruction will take the program from state i either to state $i+1$, to SINK or to HALT. In our model the probabilities of each of these transitions depends only on i and the program length L, see Figure 2. We construct a $(L+2) \times (L+2)$ Markov transition matrix T containing the probabilities in Figure 2. The probabilities of reaching the end of the program (HALT) or the looping (SINK) are given by two entries in T^L. Figure 3 shows our Markov chain describes the fraction of programs which never repeat any instructions very well.

4.2 Segment Model of Non-Looping Programs

As before, we assume half BVS instructions cause a jump. So the chance of program flow not being disrupted is 11/14. Thus the average length of uninterrupted random sequential instructions is $\sum_{i=1}^{L/2} i\,(11/14)^{i-1}\,3/14$. We can reasonably replace the upper limit on the summation by infinity to give the geometric distribution (mean of $14/3 = 4.67$ and standard deviation $\sqrt{14^2/3^2 \times 11/3} = 8.94$).

For simplicity we will assume the program's L instructions are divided into $L/4.67$ segments. Two thirds end with a JUMP and the remainder with an active BVS (i.e. with the overflow

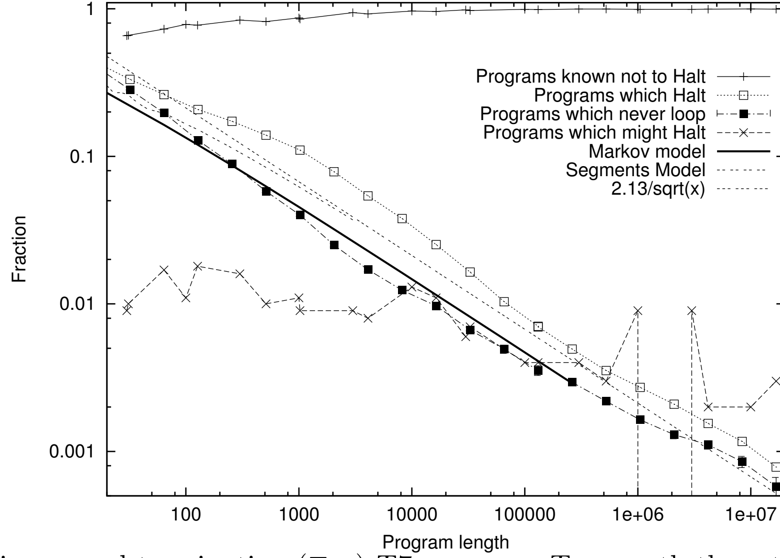

Figure 3: Looping + and terminating (□ ■) T7 programs To smooth these two curves, 50 000 to 200 000 samples taken at exact powers of two. (Other lengths lie slightly below these curves). Solid diagonal line is the Markov model of programs without any repeated instructions. This approximately fits ■, especially if lengths are near a power of two. The other diagonal line is the segments model and its large program limit, $2.13\,\text{length}^{-\frac{1}{2}}$. The small number of programs which did not halt, but which might do so eventually, are also plotted ×.

bits set). The idea behind this simplification is that if we jump to any of the instructions in a segment, the normal sequencing of (i.e. non-branching) instructions will carry us to its end, thus guaranteeing the last instruction will be executed. The chance of jumping to a segment that has already been executed is the ratio of already executed segments to the total. (This ignores the possibility that the last instruction is a jump. We compensate for this later.)

Let i be the number of instructions run so far divided by 4.67 and $N = L/4.67$. At the end of each segment, there are three possible outcomes: either we jump to the end of the program (probability $1/N$) and so stop its execution; we jump to a segment that has already been run (probability i/N) so forming a loop; or we branch elsewhere. The chance the program repeats an instruction at the end of the ith segment is

$$= \frac{i}{N}(1-\frac{2}{N})(1-\frac{3}{N})\dots(1-\frac{i}{N})$$

I.e. it is the chance of jumping back to code that has already been executed (i/N) times the probability we have not already looped or exited the program at each of the previous steps. Similarly the chance the program stops at the end of the ith segment is

$$\begin{aligned}\frac{1}{N}(1-\frac{2}{N})(1-\frac{3}{N})\dots(1-\frac{i}{N}) &= \frac{1}{N^i}\frac{(N-2)!}{(N-i-1)!} = \frac{(N-2)!}{N^{N-1}}\frac{N^{N-1-i}}{(N-i-1)!} \\ &= (N-2)!N^{1-N}e^N\text{Psn}(N-i-1,N)\end{aligned}$$

Where $\text{Psn}(k,\lambda) = e^{-\lambda}\lambda^k/k!$ is the Poisson distribution with mean λ.

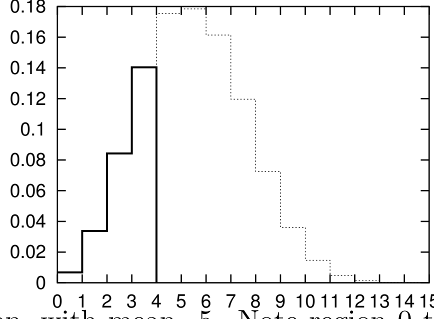

Figure 4: Poisson distribution, with mean=5. Note region 0 to mean-2 corresponding to the segments model of non-looping programs.

The chance the program stops at all (ignoring both the possibility of leaving the first loop and of other loops for the time being) is simply the sum of all the ways it could stop

$$\sum_{i=1}^{N-1} (N-2)!N^{1-N}e^N \mathrm{Psn}(N-i-1,N) \quad = \quad (N-2)!N^{1-N}e^N \sum_{j=0}^{N-2} \mathrm{Psn}(j,N)$$

For large mean (N) $\sum_{j=0}^{N-2} \mathrm{Psn}(j,N)$ approaches 1/2 (see Figure 4). Therefore the chance of long programs not looping is (using Gosper's approximation $n! \approx \sqrt{(2n+1/3)\pi}\, n^n e^{-n}$ and that for large x $(1-1/x)^x \approx e^{-1}$)

$$\begin{aligned}
& (N-2)!N^{1-N}e^N \left(\sum_{j=0}^{N-2} \mathrm{Psn}(j,N) \right) \\
\approx \;& 1/2(N-2)!N^{1-N}e^N \\
\approx \;& \sqrt{\pi(2N-11/3)} \left(\frac{N-2}{e}\right)^{N-2} N^{1-N}e^N \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (N-2)^{N-2}e^{-N+2}N^{1-N}e^N \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (N-2)^{N-2}N^{-(N-2)}N^{-1}e^2 \\
= \;& 1/2\sqrt{\pi(2N-11/3)} \left(\frac{N-2}{N}\right)^{N-2} N^{-1}e^2 \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (1-2/N)^{N-2}N^{-1}e^2 \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (1-2/N)^{N/2}(1-2/N)^{N/2}(1-2/N)^{-2}N^{-1}e^2 \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ e^{-1}e^{-1}(1-2/N)^{-2}N^{-1}e^2 \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (1-2/N)^{-2}N^{-1} \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (N(1-2/N))^{-2}N^1 \\
= \;& 1/2\sqrt{\pi(2N-11/3)}\ (N-2)^{-2}N^1 \\
= \;& 1/2\sqrt{2\pi(N-11/6)}\ (N-2)^{-2}N^1 \\
\approx \;& 1/2\sqrt{2\pi}\sqrt{N}\left(1-\frac{11}{12N}\right) N^{-2} \left(1+\frac{4}{N}\right) N^1
\end{aligned}$$

$$
\begin{aligned}
&= 1/2\sqrt{2\pi}N^{-0.5}\left(1-\frac{11}{12N}\right)\left(1+\frac{4}{N}\right)\\
&= 1/2\sqrt{2\pi/N}\left(1-\frac{11}{12N}\right)\left(1+\frac{4}{N}\right)\\
&\approx 1/2\sqrt{2\pi/N}\left(1+\frac{48-11}{12N}\right)\\
&\approx 1/2\sqrt{2\pi/N}\left(1+\frac{37}{12N}\right)
\end{aligned}
$$

That is (ignoring both the possibility of leaving the first loop and of other loops for the time being) the probability of a long random T7 program of length L stopping is about $1/2\sqrt{2\pi 14/3L}\left(1+\frac{37\times 14}{36L}\right) = \sqrt{7\pi/3L}\,(1+259/18L)$. As mentioned above, we have to consider explicitly the 3/14 of programs where the last instruction is itself an active jump. Including this correction gives the chance of a long program not repeating any instructions as $\approx 11/14\sqrt{7\pi/3L}\,(1+259/18L)$. Figure 3 shows this $\sqrt{\text{length}}$ scaling fits the data reasonably well.

4.3 Average Number of Instructions run before Stopping

The average number of instructions run before stopping can easily be computed from the Markov chain. This gives an excellent fit with the data (Figure 5). However, to get a scaling law, we again apply our segments model.

The mean number of segments evaluated by programs that do halt is:

$$
\frac{\sum_{i=1}^{N-1} i/N \ \prod_{j=2}^{i}(1-j/N)}{\sum_{i=1}^{N-1} 1/N \ \prod_{j=2}^{i}(1-j/N)}
$$

Consider the top term for the time being

$$
\begin{aligned}
&\sum_{i=1}^{N-1} i/N \ \prod_{j=2}^{i}(1-j/N)\\
&= 1/N\sum_{i=1}^{N-1} i\exp\left(\sum_{j=2}^{i}\log(1-j/N)\right)\\
&< 1/N\sum_{i=1}^{N-1} i\exp\left(\sum_{j=2}^{i}-j/N\right)\\
&= 1/N\sum_{i=1}^{N-1} i\exp\left(-\frac{i(i+1)-2}{2N}\right)\\
&= 1/N\sum_{i=1}^{N-1} i\exp\left(-\frac{i^2}{2N}\right)\exp\left(-\frac{i}{2N}\right)\exp\left(\frac{+2}{2N}\right)\\
&= 1/Ne^{\frac{1}{N}}\sum_{i=1}^{N-1} i\exp\left(-\frac{i^2}{2N}\right)\exp\left(-\frac{i}{2N}\right)\\
&< 1/Ne^{\frac{1}{N}}e^{-\frac{1}{2N}}\sum_{i=1}^{N-1} i\exp\left(-\frac{i^2}{2N}\right)
\end{aligned}
$$

$$
\begin{aligned}
&= e^{\frac{1}{2N}} 1/N \sum_{i=1}^{N-1} i \exp\left(-\frac{i^2}{2N}\right) \\
&\approx e^{\frac{1}{2N}} 1/N \int_{1/2}^{N-1/2} x e^{-x^2/2N} dx \\
&= e^{\frac{1}{2N}} 1/N \left[-N e^{-x^2/2N}\right]_{1/2}^{N-1/2} \\
&= e^{\frac{1}{2N}} \left[e^{-x^2/2N}\right]_{N-1/2}^{1/2} \\
&= e^{\frac{1}{2N}} \left(e^{-1/8N} - e^{-(N-1/2)^2/2N}\right) \\
&\approx e^{\frac{3}{8N}}
\end{aligned}
$$

Dividing $e^{\frac{3}{8N}}$ by the lower part (the probability of a long program not looping) gives an upper bound on the expected number of segments executed by a program which does not enter a loop:

$$
\begin{aligned}
&\approx \frac{e^{3/8N}}{1/2\sqrt{2\pi/N}\left(1+\frac{37}{12N}\right)} \\
&\approx 2(1+3/8N)(1-37/12N)\sqrt{N/2\pi} \\
&\approx (1+9/24N-74/24N)\sqrt{2N/\pi} \\
&= (1-65/24N)\sqrt{2N/\pi} \\
&\approx 0.8\sqrt{N}
\end{aligned}
$$

Replacing the number of segments N ($N = 3L/14$) by the the number of instructions L gives $14/3 \times \sqrt{2/\pi}\sqrt{3/14}\sqrt{L} = \sqrt{28/3\pi}\sqrt{L} \approx 1.72\sqrt{L}$. Figure 5 shows, particularly for large random programs, this gives a good bound for the T7 segments model. However the segments model itself is an over estimate.

Neither the segments model, nor the Markov model, take into account de-randomisation of memory as more instructions are run. This is particularly acute since we have a small memory. JUMP and COPY_PC instructions introduce correlations between the contents of memory and the path of the program counter. These make it easier for loops to form.

4.4 How random is memory?

Our models assume that the contents of memory is random. Figure 6 shows this assumption is valid initially. However as random instructions are executed the number of bits set tends to wonder away from its initial setting, cf. Figure 7. Where legal program addresses do not fit exactly into the power of two allocated to them, the upper bits of random addresses are more likely to be zero than 1/2. This means COPY_PC instructions tend to inject more zeros into memory. This leads to the slight asymmetry seen in one plot in Figure 6. Figure 8 shows that while memory appears random at a given time, its contents is correlated from one time to the next. A simple model based on the chance of random instructions over writing addresses stored in memory predicts, in large programs, an address will survive on average ≈ 5 instructions.

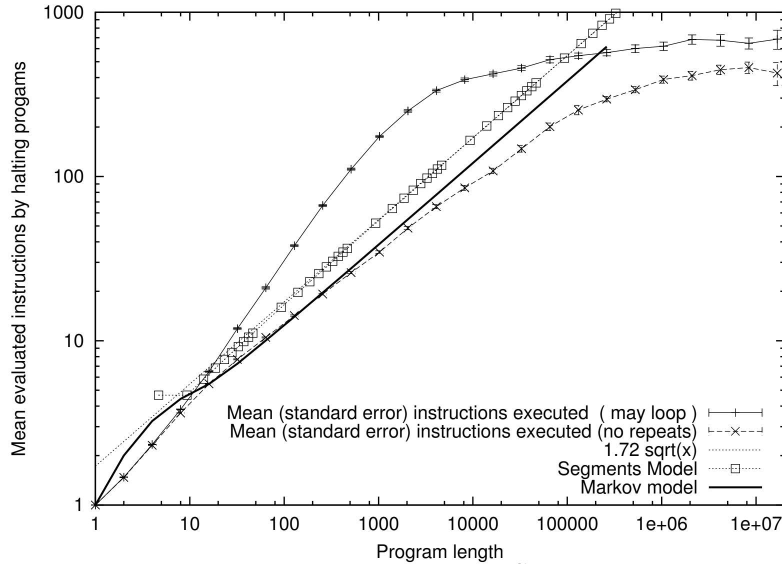

Figure 5: Instructions executed by programs which halt. Standard deviations are approximately equal to the means, suggesting geometric distributions. As Figure 3, larger samples used to increase reliability. Models reasonable for short programs. However as random programs run for longer, COPY_PC and JUMP derandomise the small memory, so easing looping.

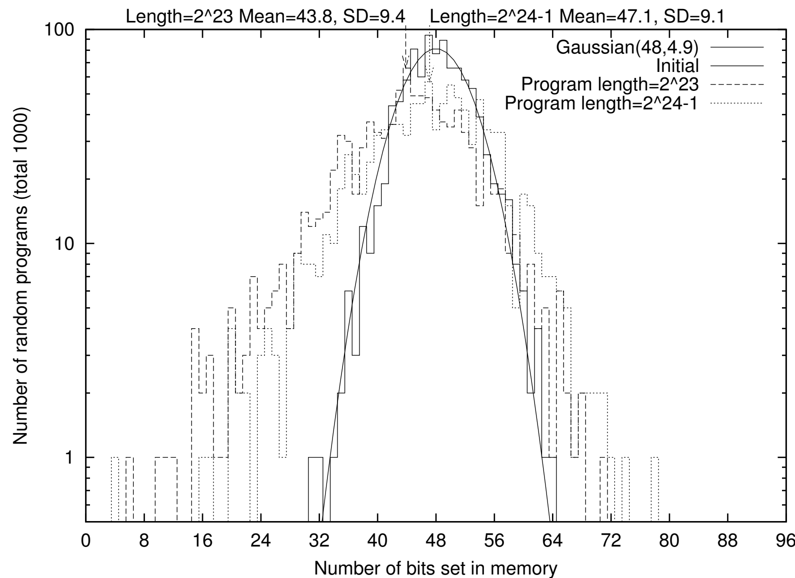

Figure 6: We count the number of bits set in memory when a random program terminates or first repeats an instruction. (Remember the longer random programs tend to execute more such instructions). Random instructions tend to derandomise the memory only a little. However note programs of length 2^{23} (dashed line) never use the most significant bit of their 24 bit addresses. This gives and a slightly asymmetric distribution, not visible for programs of length $2^{24} - 1$ (dotted line).

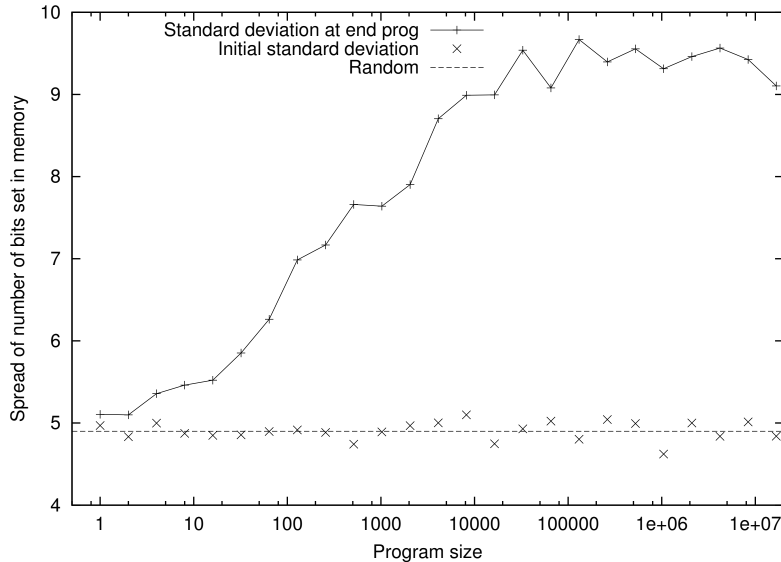

Figure 7: Spread of number of bits set in 1000 random programs, cf. Figure 6. Longer random programs tend to run more instructions before either stopping or repeating an instruction. This causes the memory to derandomise only a little. This plot shows, for large random programs, the spread in number of bits set in memory is up to twice that which would be expected of random initial fluctuations.

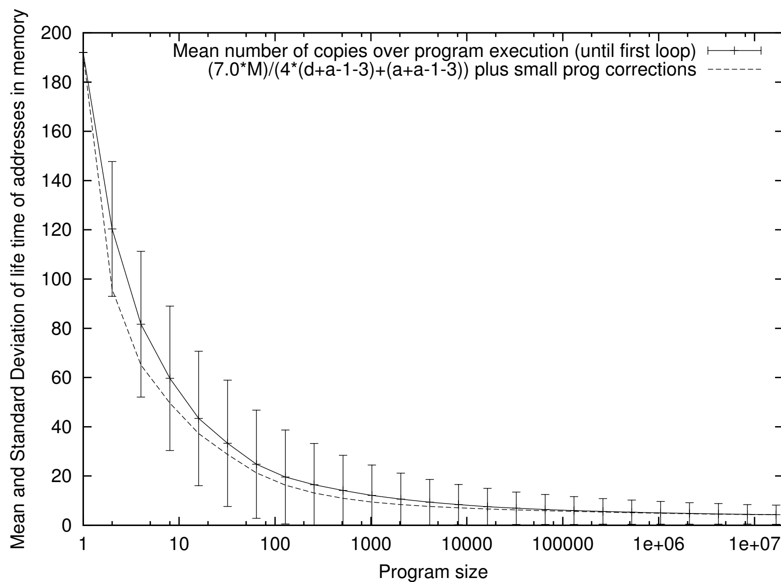

Figure 8: Life time of address sized patterns within memory. The picture for small random programs is made more complex by various uninteresting artifacts, however generally as the programs get longer the address size increases. Longer patterns are more liable to disruption by random updates, so reducing their life time. For larger programs, addresses are mostly removed rapidly from memory. Again mean and standard deviation are nearly equal, suggesting a geometric distribution. Which in turn suggest perhaps ≈ 20 random instructions are needed to be reasonably sure that an address will be deleted.

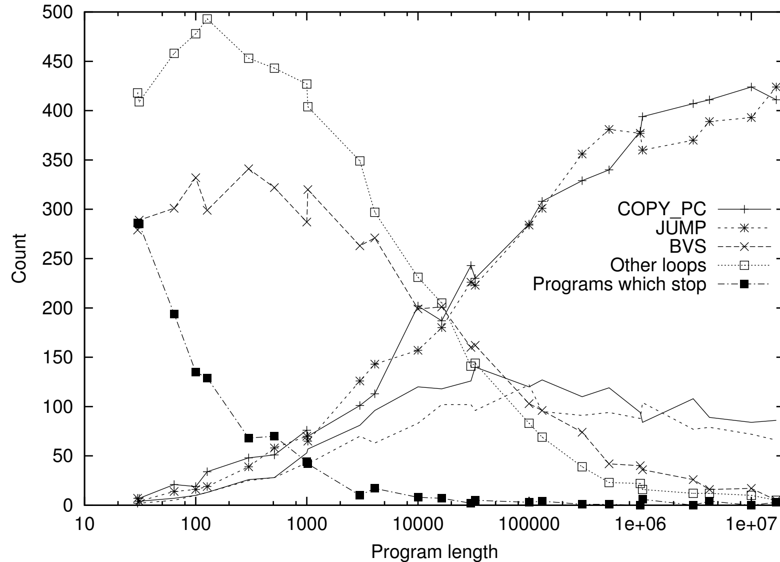

Figure 9: Types of first loop in T7 random programs. In large programs most loops have a last branch instruction which is either COPY_PC (+) or JUMP (∗) with unsullied target addresses. In a further 20–30% of programs, the lower three bits of the target address may have been modified (plotted as solid and dashed lines). Since BVS (×) jumps to any address in the program, it is less likely to be responsible for loops as the program gets larger. Mostly, the fraction of unclassified loops (□) also falls with increasing program size.

5 Loops

5.1 Code Fragments which Form Loops

If a BVS or an unconditional JUMP instruction jumps to an instruction that has been previously obeyed, a loop is formed. Unless something is different the second time the instruction is reached (e.g. the setting of the overflow flag) the program will obey exactly the same instruction sequence as before, including calculating the same answers, and so return to start of the loop again. Again, if nothing important has changed, the same sequence of instructions will be run again and an infinite loop will be performed. Automated analysis can, in most cases, detect if changes are important and so the course of program execution might change, so enabling the program to leave the loop.

We distinguish loops using the instruction which formed the loop. I.e. the last BVS or JUMP. There are two common ways JUMP can lead to a loop: either the program goes to an address which was previously saved by a LOAD_PC instruction or it jumps to an address which it has already jumped to before. I.e. the two JUMP instructions take their target instruction from the same memory register. A loop can be formed even when one JUMP address is slightly different from the other. Therefore we subdivide the two types of JUMP loops into three sub-classes: those where we know the address register has not been modified, those where the least significant three bits might have been changed, and the rest. See Figure 9.

5.2 Number of Instructions Before the First Loop

While BVS loops are longer and so might be expected to appear later in a program's execution, those we see, appear at about the same time as other types of loop, cf. Figure 10. However,

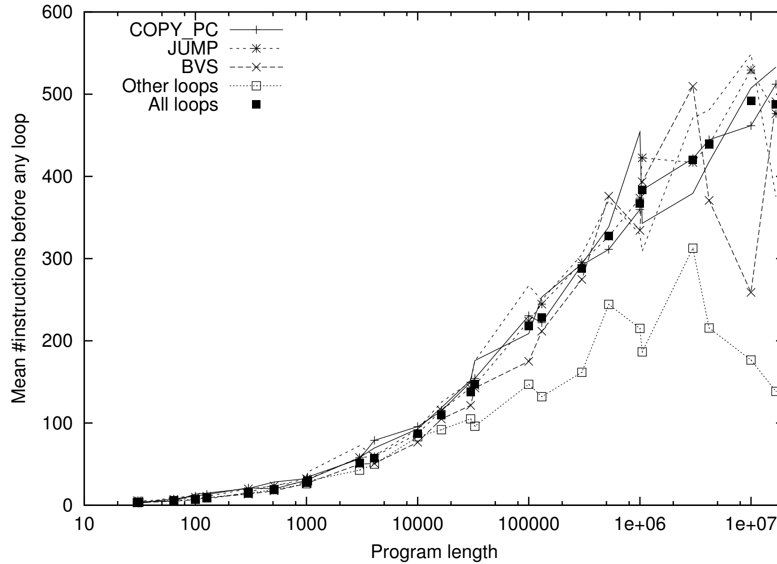

Figure 10: Number of instructions before the first loop in T7 programs.

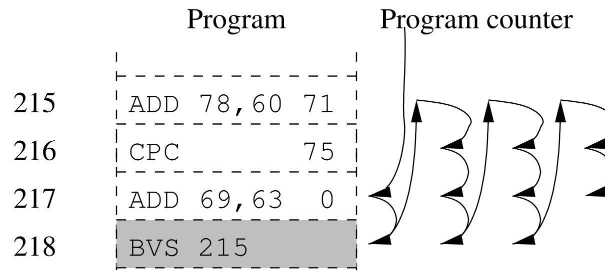

Figure 11: Example of a randomly formed BVS loop. As long as the overflow bit is set by ADDing the contents memory 69 and memory 63 the program will loop but, once the loop is formed, they do not change.

especially in long programs, BVS loops are bigger than the others, so we see far fewer of them (cf. Figure 9). That is, the competition inherent in selecting the first loop tends to make the observed mean of each type of loop behave like the fastest.

5.3 COPY_PC and JUMP Loops

As Figure 9 shows, almost all long programs get trapped in either a COPY_PC or a JUMP loop. The COPY_PC instruction lends itself readily to the formation of loops, since once its address has been saved, it can be used by a subsequent JUMP instruction. Less obvious is the JUMP JUMP loop. This is formed by jumping to a (possibly random) address taken from a memory location followed some time later by another JUMP using the same memory location. Since the first JUMP is just before the first repeated instruction a JUMP-JUMP loop appears to be one instruction shorter than a COPY_PC-JUMP loop. Both require an address in memory not to be disrupted before the second JUMP needs it. As before this gives rise to a geometric distribution (cf. Figure 14). Figures 11–13 give examples of the common types of loop. We can approximately model the lengths of both types of loops. In both cases very short loops are predicted. We would expect, since there is less chance to disrupt memory, tight loops to be more difficult to escape.

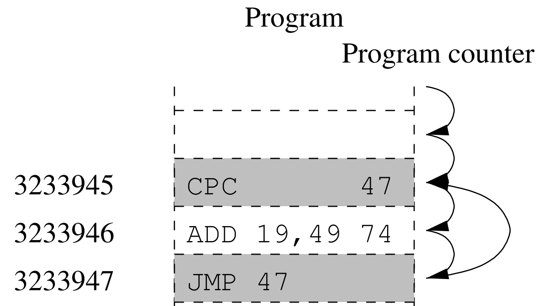

Figure 12: Example of a randomly formed loop created by a COPY_PC instruction setting up an address for a subsequent JUMP. Since the intervening ADD instruction does not modify the contents of memory 74, the loop will continue indefinitely.

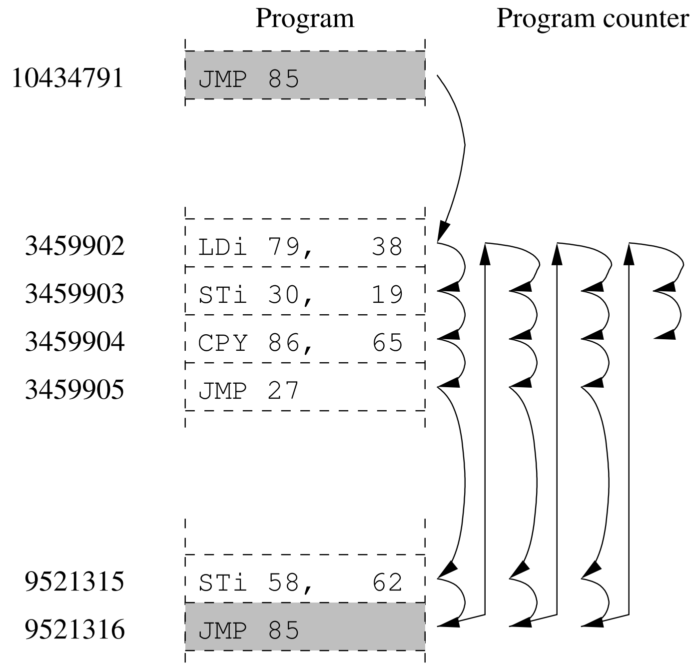

Figure 13: Example of a randomly formed JUMP-JUMP loop. The first JUMP 85 does not change memory but uses it to reposition the program counter (PC). However it is followed by another JUMP 85 which causes the PC to loop back on itself. In this example, there are two JUMPs (we name loops from the last jump which caused it to close). For the loop to persist it is necessary that the contents of both memory 27 and memory 85 are not changed.

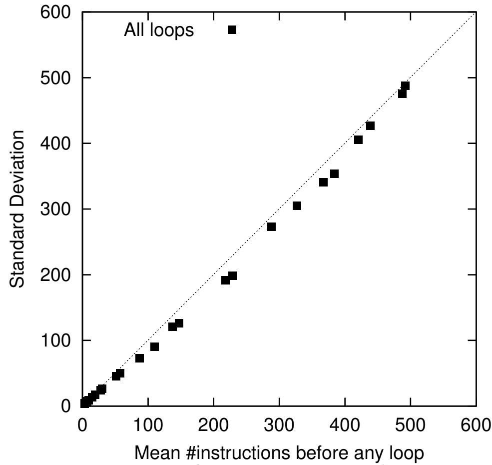

Figure 14: Mean and standard deviation of the number of T7 instructions before the first loop. In a geometric distribution the mean and standard deviation are almost equal (dotted line).

Let M = #bits = 96, A = size of program address, D = data size = 8. Assume the chance of a loop containing i instructions = chance appropriate JUMP × (chance loop not already formed and memory not disturbed)$^{i-1}$. It is very difficult to calculate the probability of another loop forming before the one of interest. Instead we will just model the random disruption of the address stored in memory by a COPY_PC instruction. We assume every memory update is with random data and (conservatively) assume changing a single bit totally destroys the address of the start of the loop. There are seven instructions, four of which write D bits and COPY_PC which writes A bits (BVS and JUMP do not change memory). If the overlap between an address and the written area is one bit, there is a 50% chance of not changing the address. If the overlap is two bits, then this is reduced to 25% and so on. Since $\sum 2^{-i}$ quickly approaches 1, we may approximate the effect of random write patterns exactly matching the original contents of memory by treating each edge of the address as if it was one bit smaller. I.e. as if the actual address size was $A-2$ bits, rather than A. Thus the chance of a random ADD modifying an address held in memory is about $((A-2)+D-1)/M$ and a random COPY_PC it is about $((A-2)+A-1)/M$.

Thus the chance of a random instruction modifying the address is $(4(A{+}D{-}3){+}2A{-}3))/7M$ $= (6A+4D-15)/7M$. The chance of not modifying it is $1-(6A+4D-15)/7M$.

Therefore the chance of a COPY_PC-JUMP loop being exactly i instructions long is about:

$$= \frac{1}{7M}\left(1-(6A+4D-15)/7M\right)^{i-1}$$

Note this is a geometric distribution, with mean $7M/(6A+4D-15)$. For the longest programs A = 24, suggesting the mean length will be 161/672=4.17. In fact, we measure 4.74 ± 0.16. Figure 15 shows considering the disruption of addresses in memory by random instructions yields approximate models of the length of most loops, particularly in large random programs. However (note we are only considering the first loop) this simple model does not fully capture the competition between different loops. The mean length for JUMP-JUMP loops will be one less (lower curve in Figure 15). In other words, the vast majority of programs in the whole search space (which is dominated by long programs) fall into loops with fewer than 20 instructions.

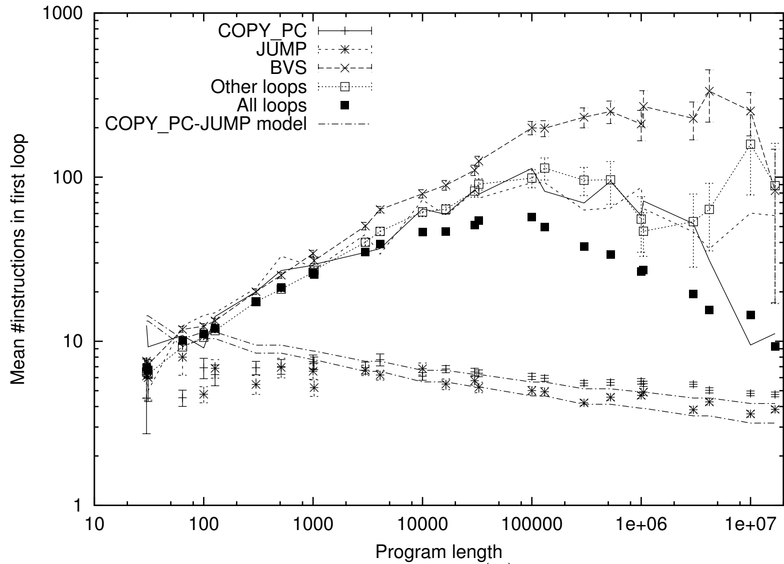

Figure 15: Length of the first loop. Note the average (■) initially tracks "other" (□), BVS (×) etc. but as the number of these types of loop, cf. Figure 10, falls the mean begins to resemble the length of first loops created by unsullied COPY_PC (+) and JUMP (∗) instructions. The theoretical curves lie close by.

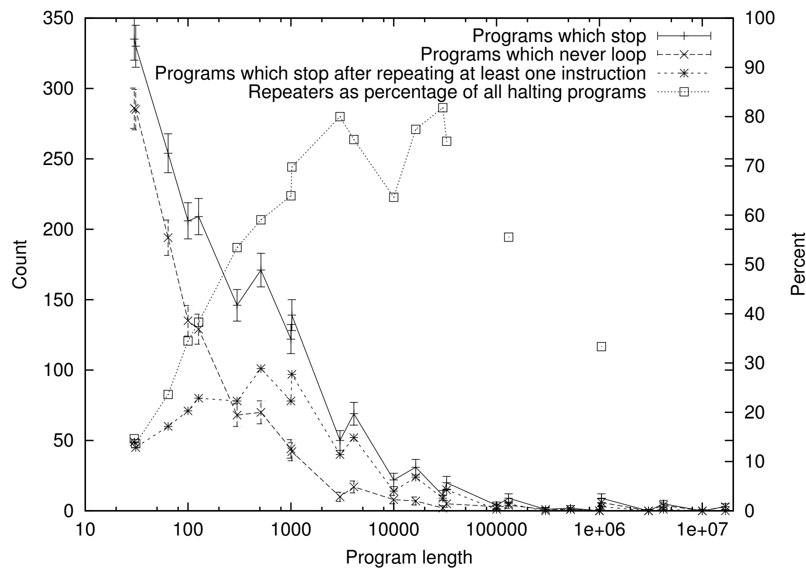

Figure 16: Number of T7 programs (out of 1000) which escape the first loop and subsequent loops and then terminate successfully. Error bars are standard errors, indicating the measured differences (∗) and percentages (□) become increasingly noisy with programs longer than 3000. Nevertheless the trend for terminating programs to loop but escape from the loop can be seen.

6 Discussion

Of course the undecidability of the Halting problem has long been known. More recently work by Chaitin [Chaitin, 1988] started to consider a probabilistic information theoretic approach. However this is based on self-delimiting Turing machines (particularly the "Chaitin machines") and has lead to a non-zero value for Ω [Calude *et al.*, 2002] and postmodern metamathematics. Our approach is firmly based on the von Neumann architecture, which for practical purposes is Turing complete. Indeed the T7 computer is similar to the linear GP area of existing Turing complete GP research.

While the numerical values we have calculated are specific to the T7, the scaling laws are general. Given time and resources it would be nice to perform similar experiments with more memory and on different computers. Obviously include a halt instruction will change the proportions radically but leads to many random programs terminating but having run only a handful of instructions. Our results are also very general in the sense that they apply to the space of all possible programs and so are applicable to both GP and any other search based automatic programming techniques.

Section 4 has accurately modelled the formation of the first loops in program execution. Section 5 shows in long programs most loops are quite short but we have not yet been able to quantitatively model the programs which enter a loop and then leave it. However we can argue recursively that once the program has left a loop it is back almost where it started. That is, it has executed only a tiny fraction of the whole program, and the remainder is still random with respect to its current state. Now there may be something in the memory which makes it to easier to exit loops, or harder to form them in the first place. For example, the overflow flag not being set. However we would expect the flag to be randomised almost immediately. Also initial studies, c.f. Section 4.4, indicate the rest of the memory remains randomised. That is having left one loop, we expect the chance of entering another to be much the same as when the program started, i.e. almost one. Thus the program will stumble from one loop to another until it gets trapped by a loop it cannot escape. As explained in Section 5, we expect, in long programs, it will not take long to find a short loop from which it is impossible to escape.

Real computer systems lose information (converting into heat). We expect this to lead to further convergence properties in programming languages with recursion and memory.

7 Conclusions

Our models and simulations of a Turing complete linear GP system based on practical von Neumann computer architectures, show that the proportion of halting programs falls towards zero with increasing program length. However there are exponentially more long programs than short ones. This means in absolute terms the number of halting programs increases with their size (cf. Figure 17) but, in probabilistic terms, the Halting problem is decidable: von Neumann programs do not terminate with probability one.

In detail: the proportion of halting programs is $\approx 1/\sqrt{\text{length}}$, while the average and standard deviation of the run time of terminating programs grows as $\sqrt{\text{length}}$. This suggests a limit on run time of, say, 20 times $\sqrt{\text{length}}$ instruction cycles, will differentiate between almost all halting and non-halting T7 programs. E.g. for a real GHz machine, if a random program has been running for a single millisecond that is enough to be confident that it will never stop.

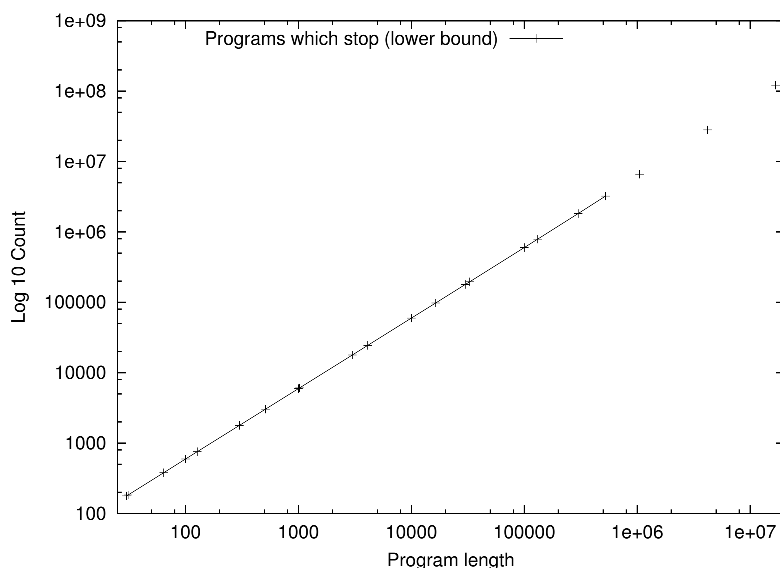

Figure 17: The number of Halting programs rises exponentially with length despite getting increasingly rare as a fraction of all programs.

Acknowledgements

I would like to thank Dave Kowalczy. Funded by EPSRC grant GR/T11234/01.

References

[Calude *et al.*, 2002] Cristian S. Calude, Michael J. Dinneen, and Chi-Kou Shu. Computing a glimpse of randomness. *Experimental Mathematics*, 11(3):361–370, 2002.

[Chaitin, 1988] Gregory J. Chaitin. An algebraic equation for the halting probability. In Rolf Herken, editor, *The Universal Turing Machine A Half-Century Survey*, pages 279–283. Oxford University Press, 1988.

[Daida *et al.*, 2005] Jason M. Daida, Adam M. Hilss, David J. Ward, and Stephen L. Long. Visualizing tree structures in genetic programming. *Genetic Programming and Evolvable Machines*, 6(1):79–110, March 2005.

[Greene, 2004] William A. Greene. Schema disruption in chromosomes that are structured as binary trees. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 1197–1207, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[Kowalczyk, 2005] Dave Kowalczyk. Programming array access F-4 MISC, October 2005. Personal communication.

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

[Langdon, 2000] W. B. Langdon. Quadratic bloat in genetic programming. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[Langdon, 2002a] W. B. Langdon. Convergence rates for the distribution of program outputs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 812–819, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Langdon, 2002b] W. B. Langdon. How many good programs are there? How long are they? In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain, 4-6 September 2002. Morgan Kaufmann. Published 2003.

[Langdon, 2003a] W. B. Langdon. Convergence of program fitness landscapes. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1702–1714, Chicago, 12-16 July 2003. Springer-Verlag.

[Langdon, 2003b] W. B. Langdon. The distribution of reversible functions is Normal. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practise*, chapter 11, pages 173–188. Kluwer, 2003.

[Maxwell III, 1994] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, volume 1, pages 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[McPhee and Poli, 2001] Nicholas Freitag McPhee and Riccardo Poli. A schema theory analysis of the evolution of size in genetic programming with linear representations. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 108–125, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[McPhee and Poli, 2002] Nicholas Freitag McPhee and Riccardo Poli. Using schema theory to explore interactions of multiple operators. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 853–860, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[McPhee *et al.*, 2001] Nicholas Freitag McPhee, Riccardo Poli, and Jonathan E. Rowe. A schema theory analysis of mutation size biases in genetic programming with linear representations. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1078–1085, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[Mitavskiy and Rowe, 2005] Boris Mitavskiy and Jonathan E. Rowe. Boris Mitavskiy and Jonathan E. Rowe. A schema-based version of Geiringer's theorem for nonlinear genetic programming with homologous crossover. In Alden H. Wright, Michael D. Vose, Kenneth A. De Jong, and Lothar M. Schmitt, editors, *Foundations of Genetic Algorithms 8*, volume 3469 of *Lecture Notes in Computer Science*, pages 156–175. Springer-Verlag, Berlin Heidelberg, 2005.

[Poli *et al.*, 2004] Riccardo Poli, Nicholas Freitag McPhee, and Jonathan E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, March 2004.

[Rosca, 2003] Justinian Rosca. A probabilistic model of size drift. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practice*, chapter 8, pages 119–136. Kluwer, 2003.

[Sastry *et al.*, 2003] Kumara Sastry, Una-May O'Reilly, David E. Goldberg, and David Hill. Building block supply in genetic programming. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practice*, chapter 9, pages 137–154. Kluwer, 2003.

[Teller, 1994] Astro Teller. 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 June 1994. IEEE Press.

A Turing Equivalence of T7 and F-4

Table 2 describes the Turing complete minimal instruction set computer (MISC) F-4. Table 3 shows each F-4 instruction can be expressed by the T7. Table 4 shows the T7 can achieve Turing completeness (since it can do everything the F-4 can do) without requiring programs to modifying their own code.

Table 2: F-4 MISC 16 bit instruction set

Instruction		*opcode*	*operand*	*operation*	*clock cycles*	v set
ADDi	imm	00 01	16 bit value	imm+A→A	3	v
ADDm	addr	00 02	16 bit address	(addr)+A→A	4	v
ADDpc		00 04		PC+A→A	3	v
BVS	addr	00 08	16 bit address	(addr)→PC if v=1	3	
LDAi	imm	00 10	16 bit value	imm→A	3	
LDAm	addr	00 20	16 bit address	(addr)→A	3	
LDApc		00 40		PC→A	3	
STAm	addr	00 80	16 bit address	A→(addr)	3	
STApc		01 00		A→PC	3	

Based on `http://www.dakeng.com/misc.html`. Indirect memory access, e.g. as needed for arrays or stacks, requires the program to modify addresses held with itself [Kowalczyk, 2005]. See Table 4

Table 3: Expressing the F-4 MISC with the T7 Instruction Set

Any program written for the F-4 can be rewritten using the new instruction set. I.e. the new instruction set is also Turing complete.

A memory location in the theoretical machine is used to hold the contents of the F-4 MISC accumulator (A).

Constants which F-4 keeps in the program (using immediate, or imm, mode) are loaded into individual designated memory locations in the new machine before the program is started.

F-4 Instruction		*T7 Implementation*			
ADDi	imm	ADD	const	A	A
ADDm	addr	ADD	addr	A	A
ADDpc		COPY_PC			temp
		ADD	temp	A	A
BVS	addr	BVS	addr		
LDAi	imm	COPY	const		A
LDAm	addr	COPY	addr		A
LDApc		COPY_PC			A
STAm	addr	COPY	A		addr
STApc		JUMP	A		

Table 4: Expressing F-4 MISC Array Access with the T7 Instruction Set

To show T7 can provide F-4's functions without the need for programs to modify their own address we code `array[i] = array [j]+27;` in both F-4 and T7 instruction sets. The F-4 code and explanation is from [Kowalczyk, 2005]. For simplicity, in this example, we will assume the T7, like the F-4, data and address word sizes are both 16 bits.

F-4 Instructions		*T7 Instructions*	
LDAm	"base of array"	COPY	"zero"→0
STAm	"temp1"	COPY	"i"→4 //multiply by 16
LDAm	"i"	ADD	"base of array",0→0
ADDm	"temp1"		
STAm	"temp1"		

So now "temp1" (T7 addresses 0..15) is the location of "array[i]" in memory. We do the same thing for array[j]+27:

LDAm	"base of array"	COPY	"zero"→16
STAm	"temp2"	COPY	"j"→20 //multiply by 16
LDAm	"j"	ADD	"base of array",16→16
ADDm	"temp2"	ADD	"27",16→16
ADDi	27		
STAm	"temp2"		

Note "zero" and "27" are T7 memory locations containing 16 bits values 0 and 27. temp1 and temp2 are now the array word addresses. All that's left is to copy one to the other. To do this, on the F-4 it is necessary to, load the word for the address and write it to the operand byte only a few instructions ahead:

F-4 Instructions		*F-4 comments*	*T7 code*	
LDAm	"temp2"	← Address of array[j]+27	STi 16→0	
STAm	^Operand1	← This is PC + 6		
LDAm	"temp1"	← Address of array[i]		
STAm	^Operand2	← This is PC + 4		
LDAm	"Operand1"	← This operand word was modified directly by previous lines; it's the "From address of array[j]+27"		A
STAm	"Operand2"	← This operand word was also modified by previous lines; it's the "To address of array[i]"		

practical F-4 implementation would have a number of small routines like the above in the lower memory segments for things like copying, frequently uses instructions, etc... If the memory location of the copy operation is not absolute (e.g., not known at compile time), then you can use LDApc and ADDi to reference the operands since their location is relative to the first two load and store instructions.

B Iteration and Recursion in Genetic Programming

B1 Iteration

[1] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[2] Kenneth E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA, 28 March-1 April 1993. IEEE Press.

[3] Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[4] Lorenz Huelsbergen. Toward simulated evolution of machine language iteration. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315–320, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[5] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[6] Gregory Scott Hornby. *Generative Representations for Evolutionary Design Automation.* PhD thesis, Brandeis University, Dept. of Computer Science, Boston, MA, USA, February 2003.

[7] Gregory S. Hornby. Creating complex building blocks through generative representations. In Hod Lipson, Erik K. Antonsson, and John R. Koza, editors, *Computational Synthesis: From Basic Building Blocks to High Level Functionality: Papers from the 2003 AAAI Spring Symposium*, AAAI technical report SS-03-02, pages 98–105, Stanford, California, USA, 2003. AAAI Press.

[8] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, volume 1, pages 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[9] Nelishia Pillay. Using genetic programming for the induction of novice procedural programming solution algorithms. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 578–583, Madrid, Spain, March 2002. ACM Press.

[10] Evan Kirshenbaum. Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories, December 17 2001.

[11] Socrates A. Lucas-Gonzalez and Hugo Terashima-Marin. Generating programs for solving vector and matrix problems using genetic programming. In Erik D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 260–266, San Francisco, California, USA, 9-11 July 2001.

B2 Bounded iteration

[12] J. R. Koza. Recognizing patterns in protein sequences using iteration-performing calculations in genetic programming. In *1994 IEEE World Congress on Computational Intelligence*, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[13] John R. Koza. Automated discovery of detectors and iteration-performing calculations to recognize patterns in protein sequences using genetic programming. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 684–689. IEEE Computer Society Press, 1994.

[14] John R. Koza and David Andre. Evolution of iteration in genetic programming. In Lawrence J. Fogel, Peter J. Angeline, and Thomas Baeck, editors, *Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming*, San Diego, February 29-March 3 1996. MIT Press.

B3 Recursion

[15] Scott Brave. Evolution of planning: Using recursive techniques in genetic planning. In John R. Koza, editor, *Artificial Life at Stanford 1994*, pages 1–10. Stanford Bookstore, Stanford, California, 94305-3079 USA, June 1994.

[16] Scott Brave. Using genetic programming to evolve recursive programs for tree search. In S. Louis, editor, *Fourth Golden West Conference on Intelligent Systems*, pages 60–65. International Society for Computers and their Applications - ISCA, 12-14 June 1995.

[17] Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[18] Daryl Essam and R. I. Bob McKay. Adaptive control of partial functions in genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 895–901, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[19] Bob McKay. Partial functions in fitness-shared genetic programming. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 349–356, La Jolla Marriott Hotel La Jolla, California, USA, 6-9 July 2000. IEEE Press.

[20] Paul Holmes. The odin genetic programming system. Tech Report RR-95-3, Computer Studies, Napier University, Craiglockhart, 216 Colinton Road, Edinburgh, EH14 1DJ, 1995.

[21] Akira Ito and Hiroyuki Yano. The emergence of cooperation in a society of autonomous agents – the prisoner's dilemma game under the disclosure of contract histories –. In Victor Lesser, editor, *ICMAS-95 Proceedings First International Conference on Multi-Agent Systems*, pages 201–208, San Francisco, California, USA, 12–14 June 1995. AAAI Press/MIT Press.

[22] Mykel J. Kochenderfer. Evolving hierarchical and recursive teleo-reactive programs through genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 83–92, Essex, 14-16 April 2003. Springer-Verlag.

[23] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 20-25 August 1989.

[24] John R. Koza. Integrating symbolic processing into genetic algorithms. In *Workshop on Integrating Symbolic and Neural Processes at AAAI-90*. AAAI, 1990.

[25] John Koza, Forrest Bennett, and David Andre. Using programmatic motifs and genetic programming to classify protein sequences as to extracellular and membrane cellular location. In V. William Porto, N. Saravanan, D. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming*, volume 1447 of *LNCS*, Mission Valley Marriott, San Diego, California, USA, 25-27 March 1998. Springer-Verlag.

[26] John R. Koza, Forrest H Bennett III, and David Andre. Classifying proteins as extracellular using programmatic motifs and genetic programming. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 212–217, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[27] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.

[28] John R. Koza, Matthew J. Streeter, and Martin A. Keane. Automated synthesis by means of genetic programming of complex structures incorporating reuse, hierarchies, development, and parameterized toplogies. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practise*, chapter 14, pages 221–238. Kluwer, 2003.

[29] Geum Yong Lee. Genetic recursive regression for modeling and forecasting real-world chaotic time series. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 17, pages 401–423. MIT Press, Cambridge, MA, USA, June 1999.

[30] Masato Nishiguchi and Yoshiji Fujimoto. Evolutions of recursive programs with multi-niche genetic programming (mnGP). In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 247–252, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[31] 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 July 1995. Morgan Kaufmann.

[32] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[33] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, March 1995.

[34] Roland Olsson. Population management for automatic design of algorithms through evolution. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 592–597, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[35] Juergen Schmidhuber. Optimal ordered problem solver. Technical Report IDSIA-12-02, IDSIA, 31 July 2002.

[36] Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[37] Lee Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[38] P. A. Whigham and R. I. McKay. Genetic approaches to learning recursive relations. In Xin Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Artificial Intelligence*, pages 17–27. Springer-Verlag, 1995.

[39] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[40] Man Leung Wong and Kwong Sak Leung. Learning recursive functions from noisy examples using generic genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 238–246, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[41] Man Leung Wong. Applying adaptive grammar based genetic programming in evolving recursive programs. In Sung-Bae Cho, Nguyen Xuan Hoai, and Yin Shan, editors, *Proceedings of The First Asian-Pacific Workshop on Genetic Programming*, pages 1–8, Rydges (lakeside) Hotel, Canberra, Australia, 8 December 2003.

[42] John R. Woodward. Invariance of function complexity under primitive recursive functions. In Boris Mirkin and George Magoulas, editors, *The 5th annual UK Workshop on Computational Intelligence*, pages 281–288, London, 5-7 September 2005.

[43] John Woodward. Evolving turing complete representations. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 830–837, Canberra, 8-12 December 2003. IEEE Press.

[44] John R. Woodward. *Algorithm Induction, Modularity and Complexity*. PhD thesis, School of Computer Science, The University of Birmingham, UK, 2005.

[45] Taro Yabuki. *Representation Schemes for Evolutionary Automatic Programming*. PhD thesis, Department of Frontier Informatics, Graduate School of Frontier Sciences, The University of Tokyo, Japan, 2004. In Japanese.

[46] Tina Yu. Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genetic Programming and Evolvable Machines*, 2(4):345–380, December 2001.

[47] Tina Yu. A higher-order function approach to evolve recursive programs. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, pages 97–112. Kluwer, Ann Arbor, 12-14 May 2005.

[48] Tina Yu and Chris Clack. Recursion, lambda-abstractions and genetic programming. In Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf, editors, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 26–30, Paris, France, 14-15 April 1998. CSRP-98-10, The University of Birmingham, UK.

[49] Tina Yu and Chris Clack. Recursion, lambda abstractions and genetic programming. 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 422–431, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[50] Gwoing Tina Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. PhD thesis, University College, London, Gower Street, London, WC1E 6BT, 1999.

B4 Recursive Signal Processing, Control and Modelling of Time Series

[51] Anna I. Esparcia-Alcazar. *Genetic Programming for Adaptive Signal Processing*. PhD thesis, Electronics and Electrical Engineering, University of Glasgow, July 1998.

[52] Ken C. Sharman and Anna I. Esparcia-Alcazar. Genetic evolution of symbolic signal models. In *Proceedings of the Second International Conference on Natural Algorithms in Signal Processing, NASP'93*, Essex University, UK, 15-16 November 1993.

[53] D. P. Searson, G. A. Montague, and M. J Willis. Evolutionary design of process controllers. In *In Proceedings of the 1998 United Kingdom Automatic Control Council International Conference on Control (UKACC International Conference on Control '98)*, volume 455 of *IEE Conference Publications*, University of Wales, Swansea, UK, 1-4 September 1998. Institution of Electrical Engineers (IEE).

[54] W. Panyaworayan and G. Wuetschner. Time series prediction using a recursive algorithm of a combination of genetic programming and constant optimization. In Alwyn M. Barry, editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 101–107, New York, 8 July 2002. AAAI.

[55] Witthaya Panyaworayan and Georg Wuetschner. Time series prediction using a recursive algorithm of a combination of genetic programming and constant optimization. In R. Matousek and P. Osmera, editors, *8th International Conference on Soft Computing MENDEL, 2002*, pages 68–73, Institute of Automation and Computer Science, Technical University of Brno, Brno, Czech Republic, 5-7 June 2002.

[56] Riccardo Poli. Evolution of recursive transistion networks for natural language recognition with parallel distributed genetic programming. In David Corne and Jonathan L. Shapiro, editors, *Evolutionary Computing*, volume 1305 of *Lecture Notes in Computer Science*, pages 163–177, Manchester, UK, 11-13 April 1997. Springer-Verlag.

[57] James Cunha Werner and Terence C. Fogarty. Genetic control applied to asset managements. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 192–201, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

In Section B4, rather than directly evolving recursive programs, genetic programming (possibly indirectly, e.g. via embryogenies, morphogenesis or development) is used to evolve recursive structures, e.g. for signal processing or feedback control.