Peter Nordin, Wolfgang Banzhaf and Frank D. Francone

This chapter describes recent advances in genetic programming of machine code. Evolutionary program induction using binary machine code is the fastest known Genetic Programming method. It is, in addition, the most well studied Genetic Programming system that uses a linear genome. Evolutionary program induction using binary machine code was originally referred to as *Compiling Genetic Programming System* (CGPS). For clarity, the name was changed in early 1998 to *Automatic Induction of Machine Code—Genetic Programming* (AIM-GP). AIM-GP stores evolved programs as linear strings of native binary machine code, which are directly executed by the processor. The absence of an interpreter and complex memory handling increases the speed of AIM-GP by about two orders of magnitude. AIM-GP has so far been applied to processors with a fixed instruction length (RISC) using integer and floating-point arithmetic. We also describe several recent advances in the AIM-GP technology. Such advances include enabling the induction of code for CISC processors such as the INTEL x86 as well as JAVA and many embedded processors. The new techniques also make AIM-GP more portable in general and simplify the adaptation to any processor architecture. Other additions include the use of floating point instructions, control flow instructions, ADFs and new genetic operators e.g. aligned homologous crossover. This chapter also discusses the benefits and drawbacks of register machine GP versus tree-based GP. This chapter is directed towards the practitioner, who wants to extend AIM-GP to new architectures and application domains.

12.1 Introduction

In less than a generation, the performance of computing devices has improved by several orders of magnitude. At the same time, their price has dropped dramatically. Today, a complete one-chip computer may be purchased for less than the price of one hour of work.

But the cost of software has not kept pace with falling hardware prices. From 1955 to the early 1990's, the proportion of system development costs attributable to software rose from 10% to 90%. Today, the demand for software greatly exceeds supply. Studies show that demand may outstrip supply by as much as three-to-one. This mismatch is sometimes referred to as a *software crisis*.

The reasons for this software crisis are not difficult to fathom. Hardware is mass-produced; the economies of scale and mass production exert constant downward pressure on hardware prices. Falling hardware prices have driven a rapid growth of demand for software.

But software is not mass-produced. It is still hand-crafted by a limited supply of programmers. While programming advances like structured programming, object-oriented programming, rapid application development and CASE tools have increased programming productivity to some extent, 99% of available CPU cycles are not used.

Further increases in hardware speed and capacity will lower the cost of system development to some extent. But it is likely that much of any such increase will result only in more unused CPU cycles. Serious reductions in system development costs in the future will likely focus on the largest remaining cost component of system development, software.

One possible approach to the software crisis is to have computers write computer programs, in other words, *automatic programming*. Such an ambitious goal would have been regarded as science fiction as recently as fifteen years ago. But today, there are a number of different approaches to automatic programming extant: Genetic Programming [Koza, 1992][Banzhaf et al., 1998], ADATE [Olsson, 1997], PIPE [Salustowicz et al., 1997] and others. All of these approaches generate computer programs by using CPU cycles instead of human programmers. In a real sense, automatic programming holds the promise that computer programs, like computer hardware, may someday be mass-produced.

But mass-production of computer programs faces significant obstacles. One of the most formidable is, ironically, limitations on available CPU time. Put simply, Genetic Programming and other automatic programming techniques are *very* computationally expensive. AIM-GP (Automatic Induction of Machine Code) addresses this obstacle by evolving programs using direct manipulation of native machine code. This results in a speedup of almost two orders of magnitude over other automatic programming systems.

AIM-GP has been the subject of extensive research and development for several years at the University of Dortmund in Germany and, during that time, has been the subject of many published articles. Originally limited to RISC chips and to a small number of inputs, AIM-GP has recently been extended to CISC environments such as the WINTEL (MS-Windows on Intel) platform and Java byte code. Today, AIM-GP is available on WINTEL machines in the commercial software package, Discipulus^{TM}, and in an academic research version for Java byte code.

AIM-GP has also been extended to include many new features such as floating-point arithmetic, greatly expanded input capabilities, conditional branching, flexible control over the function and terminal set, and an important new genetic operator, Homologous Crossover. These advances make AIM-GP more flexible and portable in general and simplify its adaptation to any processor architecture.

While such additional capabilities are simple to add in a typical Genetic Programming system, they pose a considerably greater challenge in AIM-GP, where all evolved programs must be made up of synctactically correct native machine code. The purpose of this chapter is to report the new techniques that made possible the recent advances in AIM-GP.

12.2 Why Evolve Machine Code?

All commercial computers are built around a CPU that executes native machine code. In fact, every task computers perform, including genetic programming, will in the end be executed as machine code.

It is, of course, possible to perform genetic programming using high-level data structures that represent computer programs. Most Genetic Programming systems, including all tree-based systems, represent evolving programs in this manner [Banzhaf et al. 1998:309-

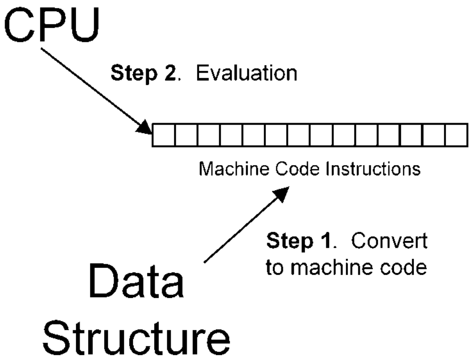

Figure 12.1
GP using high level data structures

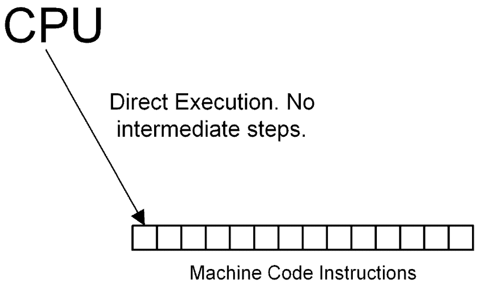
CPU
Direct Execution. No intermediate steps.
Machine Code Instructions

Figure 12.2
AIM-GP using direct binary manipulation

338][Koza 1992] [Keith and Martin, 1994]. Figure 12.1 demonstrates this approach. In such systems, high-level data structures that represent programs are converted into machine code at runtime (Step 1). Then, the resulting native machine code is evaluated for fitness (Step 2) This approach provides flexibility. The trade-off is that Step 1 is *very* computationally intensive.

But it is often advantageous evolve machine code directly in Genetic Programming. Figure 12.2 demonstrates the AIM-GP approach. In the direct binary approach of AIM-GP, there is no conversion step. The population of evolving programs is maintained, transformed, and evaluated in native machine code.

12.2.1 Advantages of Evolving Machine Code

In general the reasons for evolving machine code, rather than higher level languages, are similar to the reasons for programming by hand in machine code or assembler. The most important reason for using the direct machine code approach is speed. There are, however, other reasons also to evolve machine code directly:

1. The most efficient optimization is done at the machine code level. This is the lowest level for optimization of a program and it is also where the highest gains are possible. The optimization could be for speed, space or both. Genetic programming could be used to evolve short machine code subroutines with complex dependencies between registers, stack and memory.

2. High level tools could simply be missing for the target processor. This is sometimes the case for processors used for embedded control systems.

3. Machine code is hard to learn, program and master. This may be a matter of taste but it may be easier to let the computer itself evolve small machine code programs rather than writing machine code by hand.

4. Machine code genetic programming is inherently *linear*. That is, both the genome and the phenome are linear. Another reason to use a linear approach is that there is some evidence that the linear structure with side effects may yield a more efficient search for some applications, see section 12.7.

Some of these benefits may be achieved with a traditional tree-based Genetic Programming system evolving using a constrained crossover operator. However, there are additional reasons, in addition to speed, for working with binary machine code:

- A binary machine code system is usually memory efficient compared to a traditional Genetic Programming system. This is partly because knowledge of the program encoding is supplied by the CPU designer in hardware. Hence there is no need to define the language and its interpretation. In addition, the system manipulates the individual as a linear array of op-codes, which is more efficient than the more complex symbolic tree structures used in traditional Genetic Programming systems. Finally, CPU manufacturers have spent thousands of man-years to ensure that machine instruction codes are efficient and compact. Genetic Programming systems that use machine code instructions benefit directly from the manufacturers' optimization efforts.

- Memory consumption is usually more stable during evolution with less need for garbage collection etc. This could be an important property in real time applications.

- The use of machine code ensures that the behavior of the machine is correctly modeled since the same machine is used during fitness evaluation and in the target application.

On a more abstract level, machine code is the ”natural” language of all computers. Higher level languages such as C, Pascal, LISP and even Assembler are all attempts to make it easier for humans to think about programming without having to deal with the complexity of machine code. Traditional genetic programming systems ”think” about programming in a manner analogous to higher level computer languages. It may well be that these high-level, human techniques of thinking about programming are completely unnecessary when a computer is doing the programming. In fact, such higher level constructs may prevent the computer from programming as efficiently as it might.

12.3 Why is Binary Manipulation so Fast?

The approach of direct binary manipulation is between sixty and two-hundred times faster than comparable tree based interpreting systems. A partial explanation for its speed may be found in how an interpreter works.

To evaluate the expression $x = y + z$ in an interpreting system would normally require at least five different steps:

1. Load operand y from memory (e.g. stack)
2. Load operand z from memory
3. Look up symbol “+” in memory and get a function pointer
4. Call and execute the addition function
5. Store the resulting value (x) somewhere in memory

It is not difficult to calculate the best performance possible from an interpreting system for this evaluation. A memory operation normally takes at least three clock cycles for the CPU, even when there is a *cache hit*. The three memory operations listed above will, therefore, take nine clock cycles at best. Looking up the function pointer requires another memory access in an ideal hash table which means three additional clock cycles. Calling and executing a function usually takes from six to fifteen additional clock cycles depending on compiler conventions and type of function. All in all, this means that the best performance from an interpreting system for this simple function is about twenty clock cycles.

By way of contrast, Genetic Programming systems that work directly on the binary machine code execute the $x = y + z$ expression as a single instruction in one (1) clock cycle. AIM-GP should therefore be at least twenty times faster than an interpreting system. In fact, the speedup is by somewhat more than a factor of twenty. The remaining difference in performance is probably due to cache issues. All timing issues on modern CPUs are very sensitive to cache dependencies. Thus, any analysis that assumes cache hits must be viewed as the minimum speedup possible from the AIM-GP approach.

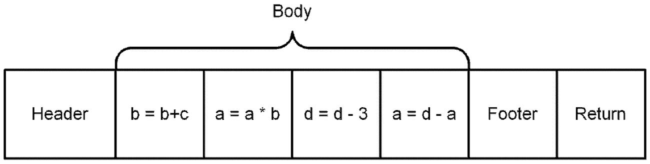

Figure 12.3
Structure of a program individual

12.4 Overview of AIM-GP

AIM-GP was formerly known as ”Compiling Genetic Programming Systems” or ”CGPS”. While AIM-GP is the best known system that works in a machine code type environment, there are others such systems. All such systems use a linear representation of the genome in contrast to the common tree-based Genetic Programming representation.

Genetic Programming systems that operate in a machine-code type environment may, in principle, be classified into three categories:

1. Approaches that evolve programs with a small virtual (toy) machine for research purposes [Cramer, 1985][Huelsberger, 1996].

2. Approaches that evolve programs using a simulation of a real machine or with a virtual machine designed for real applications [Crepeau, 1995].

3. Approaches that manipulate the native machine code of a real processor such AIM-GP. This third approach is, of course, the focus of the remainder of this chapter.

AIM-GP may be regarded as a large alphabet genetic algorithm operating on a variable-length, linear string of machine code instructions. Each individual consists of a header, body, footer, return instruction, and buffer. Figure 12.3 shows the structure of an evolved program in AIM-GP: The body contains machine code instructions contained in the terminal set of an AIM-GP run. All genetic operators are applied to the *body* of evolved programs in AIM-GP. For basic details regarding AIM-GP see [Nordin, 1997, Banzhaf et. al., 1998, Section 11.6.2-11.6.3].

AIM-GP evolves *imperative* programs. Imperative programs consist of instructions affecting a *state*. For example, instructions that assign values to variables or operating on the values contained in CPU registers are imperative instructions. Most commercial programming languages, such as C++, Pascal and Fortran, are imperative languages.

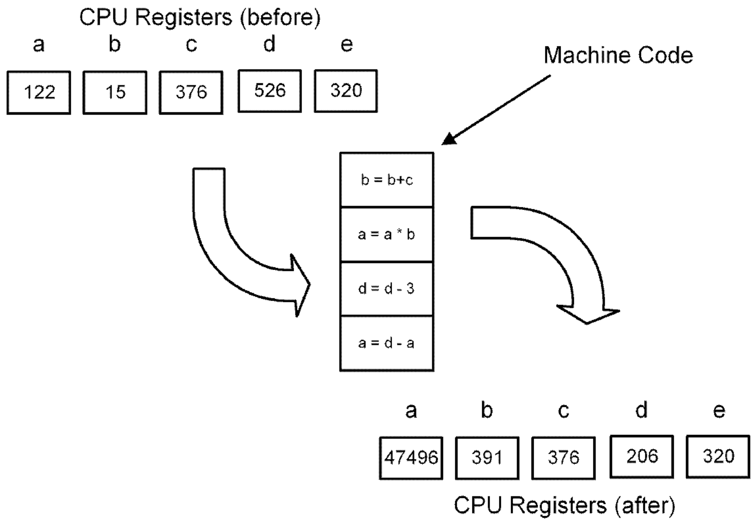

Figure 12.4
State transformations in CPU registers caused by AIM-GP program

Figure 12.4 is an example of how a sequence of AIM-GP instructions might transform the state of the registers of a hypothetical CPU. By way of contrast, traditional tree based Genetic Programming approaches are often inspired by *functional* programming approaches [Koza 1992]. More recently, there has been a trend in tree based Genetic Programming to view the tree more like a list of imperative instructions operating on state, than a function tree. This includes approaches using memory and *cellular encoding* [Gruau, 1995].

AIM-GP systems are now running on several different platforms and architectures. So far implementations exist for the following platforms using five different processor families:

- SUN-SPARC
- MOTOROLA POWER-PC
- INTEL 80X86
- Sony PlayStation
- Java Bytecode

The POWER-PC, Sony PlayStation and SPARC are all RISC architectures while INTEL 80X86 is a CISC architecture. The POWER-PC version works both on the Macintosh architecture and in a PARSYTEC parallel machine.

Java byte code occupies a spot by itself. It has a handful of instructions longer than a byte. So it could be seen as a CISC architecture even though it is possible to implement powerful systems using only the instructions of the fixed one-byte size. Furthermore, when running on the Java Virtual Machine, Java byte code is not precisely native machine code although it bears many similarities.

12.5 Making Machine Code Genetic Programming Work on CISC Processors

Many of the additions to AIM-GP reported here are the result of porting AIM-GP to CISC architectures. In particular, Instruction Blocks and Instruction Annotation made the transition straightforward and flexible.

12.5.1 The Importance of Extending AIM-GP to CISC Processors

The first AIM-GP systems was only able to evolve programs for *reduced instruction set computer* (RISC) architectures. A RISC processor has instructions of *equal length* and a relatively simple instruction grammar. By way of contrast, a CISC (Complex Instruction Set Computer) has instructions of varying length and, for lack of a better term, a messier instruction syntax.

The *PC de-facto-standard* is built on CISC and so are also many computers used for embedded applications. Other commonly-used computer architectures operate with variable length instructions, for example, the Motorola 68XXX and to some extent Java byte code. Being able to handle CISC processors is important for any Genetic Programming paradigm.

In addition to making AIM-GP work on the most common computer architectures, CISC processors also have large instruction sets with many special instructions. These special instructions are very useful in extending the capabilities of binary machine code induction systems like AIM-GP. For example, the INTEL X86 has a set of powerful string and loop instructions, which are never found on RISC machines:

- CMPS/CMPSB/CMPSW/CMPSD–Compare String Operands. These instructions can be used to compare strings for example in text search applications.

- STOS/STOSB/STOSW/STOSD Store String, LODS/LODSB/LODSW/LODSD Load String and MOVS/MOVSB/MOVSW/MOVSD–Move Data from String to String. Can be used when copying strings for instance in text data mining.

- LOOP/LOOP cc–Loop instructions allows for very compact and efficient loop constructs.

These single, powerful instructions, available only in CISC architectures, are important because such single instructions, when included in the function set, perform the same role as an external function call in a less complex and much faster manner.

12.5.2 Challenges in Moving AIM-GP to CISC Processors

Moving AIM-GP to CISC processors posed some real technical problems. Most approaches to AIM-GP perform crossover and mutation directly on the native machine code. It is easy to find where you are in RISC machine code since all instructions are the same length. Thus, if you know where the evolved program begins, you can easily find the instruction boundaries during crossover and mutation. For example, where all instructions are 32 bits long, the programmer always knows that a new instruction begins every 4 bytes.

CISC instructions are completely different. One encounters instructions that are 8 bits, 16 bits, 24 bits, 32 bits and more in length. Finding the instruction boundaries requires the programmer either to parse the entire evolved program to locate the instruction boundaries or to maintain information about program structure. Discipulus^{TM} (the commercial version of AIM-GP) maintains that information both implicitly (with Instruction Blocks) and explicitly (with Instruction Annotations). The next two sections of this chapter describe these innovations.

12.5.3 Instruction Blocks

To perform crossover and mutation in AIM-GP, we must know where the boundaries between instructions reside. Locating these boundaries can be difficult in CISC programs because of the variable instruction lengths. Figure 12.5 represents an evolved program comprised of variable length CISC instructions:

In Discipulus^{TM}, we imposed order on this scheme by combining one or more variable length instructions into fixed length *Instruction Blocks*. For example, in one scheme, we might fix the Instruction Block length at 32 bits. Each Instruction Block may contain any combination of instructions that are, taken together, 32 bits in length. Various examples of four different ways to put together an Instruction Block are illustrated in Figure 12.5. The Instruction Blocks may also contain NOPs (No Operation Instructions).

After grouping CISC instructions into Instruction Blocks, an Evolved program may be represented as in Figure 12.7:

Crossover With Instruction Blocks

The fixed length Instruction Blocks shown in Figure 12.7 simplify the crossover operator and memory management. They also make it straightforward to use new crossover methods such as aligned (homologous) crossover, as seen below Section 12.6.3.

Crossover with fixed length Instruction Blocks works only on the boundaries of the instruction blocks. This allows crossover to calculate and access each crossover point directly as shown in Figure 12.8. The size of the Instruction Blocks may be a settable parameter. In that case, the block size must be set so that the largest instruction used will fit in the block. However it should be small enough to allow the crossover operator to do useful recombination.

Mutation With Instruction Blocks As noted above, the crossover operator works blindly

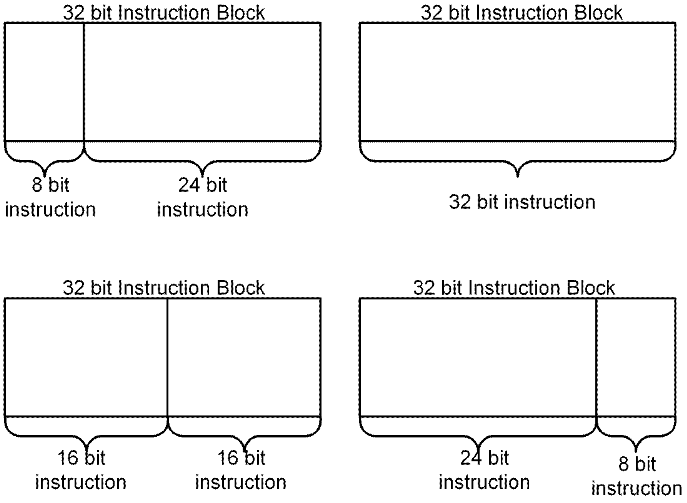

Figure 12.5
Four different 32 bit Instruction Blocks

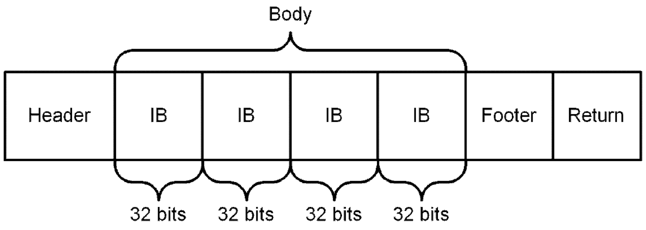

Figure 12.6
CISC AIM-GP evolved program

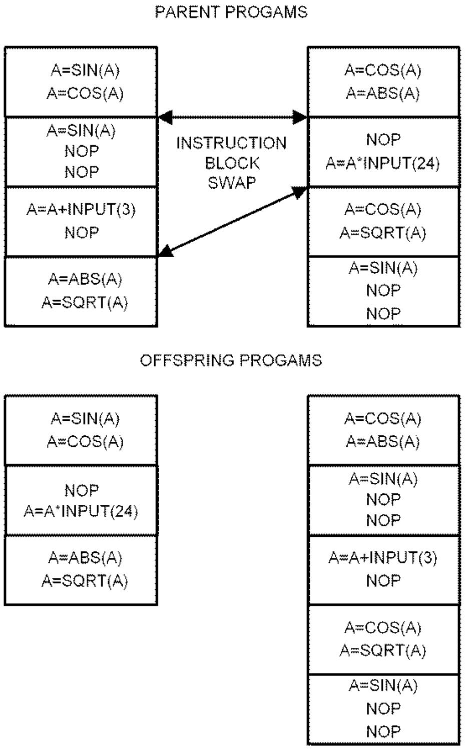

Figure 12.7
The Crossover Operator Using Instruction

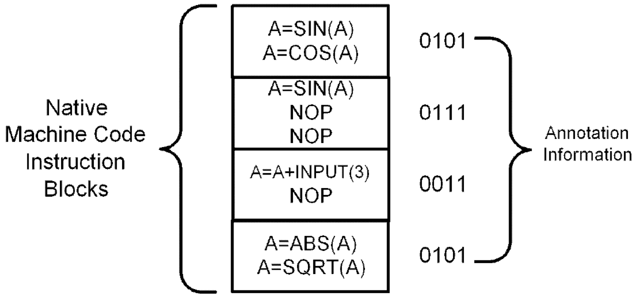

Figure 12.8
Blocks CISC AIM-GP evolved program with Instruction Blocks (”IB”)

in between the blocks. On the other hand, mutation operates inside the Instruction Blocks. Discipulus^{TM} implements three different types of mutation:

- *Block Mutation* replaces an existing Instruction Block with a new, randomly-generated Instruction Block.

- *Instruction Mutation* replaces a single instruction (inside of an Instruction Block) with a new, randomly-generated instruction.

- *Data Mutation* replaces one of the operands of a single instruction with another, randomly-generated operand.

12.5.4 Instruction Annotations

To perform Instruction Mutation or Data Mutation, AIM-GP operates inside the Instruction Blocks. Because there may be more than one instruction within an Instruction Block, the mutation operator needs to know where the boundaries of instructions reside. There are two ways to do this:

- The point where one instruction finishes and another one begins may be determined by ”decompiling” the binary code and determining the length of each instruction from a lookup table.

- The simpler way is to add extra information to each instruction in a separate array. This *annotation array* gives information about the position of instruction boundaries within an Instruction Block. The annotation information is a short binary string. Each binary digit corresponds to a *byte* in the block. If the binary digit is a 1 then a new instruction starts in this byte. If the binary digit is 0 then the previous instruction continues in this byte, see Figure 12.9.

When evolving Java byte code, annotation information is very useful. The Java virtual machine is a stack machine. *Current stack depth* is an example of annotation information kept with every instruction in our Java AIM-GP approach. We have also used annotation information to keep track of jump offsets in the Java system.

A word of caution is in order. If too much annotation information is used then the system probably contains a *compiler* translating annotation information into an executable. If this is the case then manipulating binary code might not be worth the extra complexity. So, there is a trade-off between annotation information, expressiveness and efficiency.

12.5.5 The Benefits of ”Glue”

Both Instruction Blocks and Instruction Annotations are different ways to *”glue”* multiple instructions together into functional groupings. Instruction Blocks comprise implicit ”glue” because their constant length is recognized by the crossover operator. Instruction

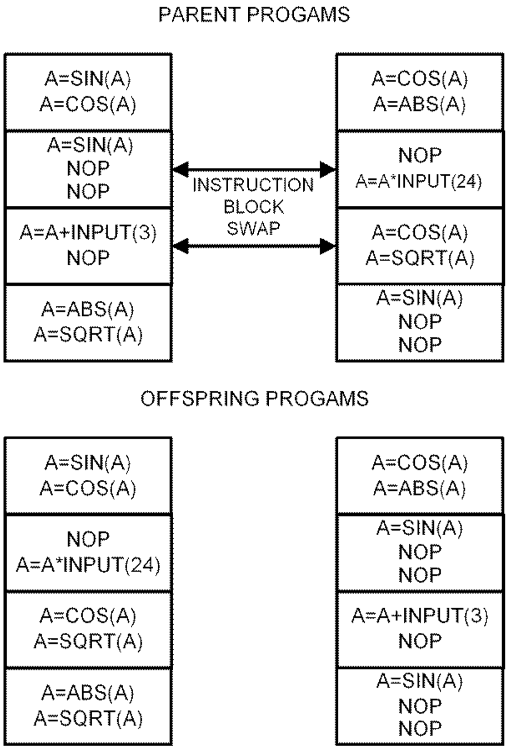

Figure 12.9
The homologous crossover operator with blocks

Annotations may be used as explicit ”glue”, delineating the start of compound instructions. Such glued instructions are very useful since they can be seen as small and very efficient user-defined functions. Compound instructions are also useful for special tricks such as ADFs, jumps and string manipulation.

Glued instructions also have benefits for applications on RISC architectures. In particular, the ability to glue instructions together can yield more efficient constructs than using functions calls for the same feature. Previously, special *leaf functions calls* in assembler, were used for user-defined-functions, ADFs, and protected functions [Nordin 1997]. However, a good deal of overhead is involved in a function call and it is also a more complex solution. A glued block does the same job and is usually more efficient than using function calls.

Finally, glued instructions held together in Instruction Blocks also appear to assist the Genetic Programming algorithm by making it easy to protect real *building blocks* against crossover. One building block observed by the authors that evolves repeatedly is a block comprised of an absolute value instruction followed by a square root instruction. Of course, the absolute value ensures that the square root function (which only accepts positive numbers) will return a number and not an error symbol. A block consisting of these two instructions is nothing but a protected function that evolves spontaneously through mutation over-and-over in AIM-GP runs that have Instruction Blocks.

12.6 Other AIM-GP Innovations

A number of other additions to the AIM-GP architecture are of note and are detailed in this section.

12.6.1 Memory Access and Large Input Sets

A CISC processor usually has fewer registers than a RISC processor. The CPU compensates for this by more efficient instructions for mixing values in memory with register operations. Such operations can be especially efficient if the *cache* is aligned. In that case, these operations are almost as fast as a register-to-register operation.

Previously, with the RISC approach, the maximum number of inputs available in AIM-GP systems was around fourteen variables. The fast and convenient memory access available on CISC machines makes it easy to expand the number of inputs efficiently. Currently Discipulus^{TM} may use up to sixty-four inputs although that number may be easily increased.

AIM-GP has with this version been used for data mining applications with wide input sets consisting of 40 columns or more. In such applications Genetic Programming seems to work well without any specific external *variable selection algorithm*. Instead Genetic Programming does an excellent job selecting relevant input columns and omitting irrelevant inputs from the resulting program.

12.6.2 Decompilation

Evolved machine code can be disassembled into compilable C-code. Decompilation is very useful for platform portability of the evolved programs. In Discipulus^{TM} we decompile to ANSI C programs. As a result, the decompiled programs may be compiled directly to most processors for which a C compiler is available.

The example below is a decompiled evolved program from Disciplus^{TM}. Even though this programs was evolved on a Pentium machine, the decompilation converts register and memory references in the machine code into values that may be used by most processors.

It this example, the f array in the program below, stands for the eight FPU registers available in the Pentium FPU while the v array represents the array of input values. Thus, the instruction $f[0]* = v[27]$ means that register 0 should be assigned the value of input number 27 multiplied by the preexisting value in register 0.

```
#define LOG2(x) ((float) (log(x)/log(2)))
#define LOG10(x) ((float) log10(x))
#define LOG_E(x) ((float) log(x))
#define PI 3.14159265359
#define E  2.718281828459

float DiscipulusCFunction(float v[])
{
  double f[8];
  double tmp = 0;

  f[1]=f[2]=f[3]=f[4]=f[5]=f[6]=f[7]=0;
  f[0]=v[0];

  l0: f[0]-=f[0];
  f[0]*=f[0];
  l1: f[0]-=0.5;
  l2: tmp=f[1]; f[1]=f[0]; f[0]=tmp;
  f[0]-=f[0];
  l3: f[0]*=f[0];
  l4: f[0]+=f[0];
  f[0]=fabs(f[0]);
  l5: tmp=f[0]; f[0]=f[0]; f[0]=tmp;
  f[0]*=f[0];
  l6: f[0]*=0;
  l7: f[0]-=0.5;
  l8: f[0]*=v[27];
  l9: f[0]*=f[0];
  l10: f[0]*=v[32];
  l11: f[0]*=v[4];
  l12: f[0]+=f[1];
  l13: f[0]+=f[0];
  f[0]=fabs(f[0]);
  l14: f[0]-=f[1];
  f[0]+=f[1];
  l15: f[0]*=v[61];
  l16:
  l17:
```

12.6.3 Homologous Crossover

One of the principal criticisms of standard Genetic Programming is that the crossover operator is too ”brutal”. It performs crossover by exchanging any sub-tree regardless of the context in which the sub-tree operated. The standard crossover operator exchanges subtrees with such little selectivity that crossover could be argued to be more of a mutation operator and Genetic Programming more like a hill-climbing algorithm with a population than a system working with recombination [Banzhaf et al. 1998:143-173]. The same argument can be made regarding the usual two-point string crossover in AIM-GP [Nordin 1997].

Natural crossover does not usually exchange ”apples” and ”pies”. Foot-genes are rarely crossed-over with nose-genes. In natural crossover, the DNA of the parents are aligned before a crossover takes place. This makes it likely that genes describing similar features will be exchanged during sexual recombination. Thus, biological crossover is *homologous* [Banzhaf et al.:48-54].

In nature, most crossover events are successful. That is, they result in viable offspring. This is in sharp contrast to Genetic Programming crossover, where 75% of the crossover events are what would be termed in biology *lethal*.

Homologous Crossover Mechanism. AIM-GP now contains a mechanism for crossover that fits the medium of Genetic Programming and that may achieve results similar to homologous crossover in nature. In nature, homologous crossover works as follows:

- Two parents have a child that combines some of the genomes of each parent.

- The natural exchange is strongly biased toward experimenting with features exchanging very similar chunks of the genome, specific genes performing specific functions, that have *small* variations among them, e.g., red eyes would be exchanged against green eyes, but not against a poor immune system.

Homologous crossover exchanges blocks at the same position in the genome allowing certain meaning to be developed at certain loci in the genome. Homologous crossover can be seen as an emergent implicit grammar where each position, *loci*, represents a certain *type of feature* in many ways similar to how *grammar based GP* systems work [Banzhaf et. al. 1997]. *Homologous Crossover Effects.* The authors have noted several effects of making the homologous crossover operator the dominant crossover operator. They are:

- Significant and consistent improvement in search performance.

- Less *bloat* or code growth. This makes sense if bloat is partly seen as a defense against the destructive effects of crossover. A reasonable hypothesis is that the homologous crossover exchanging blocks at the same position will be less destructive after some initial stabilizing of features at loci.

- Implementation efficiency. The execution of the evolved programs is so fast in AIM-GP that even the time to perform crossover becomes significant (20%). Homologous crossover is faster than standard crossover since it exchanges segments with the same size. Therefore, no blocks of machine code need to be shifted forward or back.

Tree Based Homologous Crossover. Homologous crossover is easy to formulate and implement in a linear imperative system such as AIM-GP since two evolved programs can be aligned in a manner analogous to DNA crossover in nature. With tree based systems it is not as easy to find a natural way to align the two parents. However, so-called *one-point crossover* is a very interesting development in tree-based Genetic Programming that, we speculate, may act in a manner similar to linear homologous crossover [Poli and Langdon, 1998]. In one-point crossover, the nodes of the two parents are traversed to identify nodes with the same position and shape (arity). In this way, the trees can be partly aligned.

Such a tree based system has an interesting property in that it allows the insertion of a sub-trees of any size without violating alignment. This is not as easy in a linear system such as DNA or AIM-GP. The only way to achieve the same effect in AIM-GP is to use *ADFs*. Using ADFs allows the homologous insertion of a block calling an ADF with arbitrary size, see Section 12.6.5. A mechanism like this is important since a new individual with very different alignment will have severe difficulties surviving in a population with a majority of differently aligned individuals. In this way alignment can be seen as a kind of speciation.

12.6.4 Floating Point Arithmetic

Many conventional Genetic Programming systems operate with floating-point arithmetic. Until recently, AIM-GP has used the ALU (Integer and Logic Unit). However, floating-point arithmetic has many benefits. One of them is access to efficient hardware, which implements common mathematical functions such as SIN, COS, TAN, ATN, SQRT, LN, LOG, ABS, EXP etc. as single machine code instructions. There are also a dozen well-used constants, such as PI, available.

Another substantial benefit of floating-point representations is portability of evolved code. All floating-point units adhere to a common standard about how to represent numbers and how certain functions (such as rounding) should be performed. The standard also describes what to do with exceptions (e.g. division by zero). All floating-point exceptions are well-defined and result in an error symbol (for instance INF) being placed into the result register. This causes fewer problems with protected functions because execution continues with the symbol in the register. When the function returns the symbol, this can be detected outside the evolved program and punished by a poor fitness evaluation.

Processor manufacturers have recently discovered the benefits of *conditional loads* in the FPU. Such instructions loads a value into a register if a certain condition holds. The calculation following the conditional load can then take very different paths depending on if the value was loaded or not. This way the instruction works as an efficient single instruction *if-statement*.

Even if floating point processors have many powerful new instructions, it is still important to select instructions with care. For instance the FPU of the INTEL processor has eight registers organized as a stack. But stack type instructions cause some problems in evolution. Best results in evolution have to date been obtained by omitting instructions that push or pop the stack. Instead it is more efficient to use the FPU registers as normal registers machine registers and load input directly into them.

Constants are more difficult to implement in AIM-GP floating point systems than in integer based systems. In integer systems there are *immediate data* available as part of the instructions. These immediate data may be used as constants in the individual and mutated to explore the search space of integer constants during evolution. In the floating-point instruction set, there are no constants in the instruction format. Instead constants must be initialized at the beginning of a run and then, during the run, loaded from memory much like the input variables.

Another possible feature when using CISC processors and floating-point units is the ability to use multiple outputs. The transfer of a function's result on a CISC floating-point application is communicated through memory. This technique enables the use of multiple outputs by assignment of memory in the individual. In principle there is no limit on the number of items in the output vector. A multiple output system is important in for instance control applications where it is desirable to control for instance several motors and servos.

12.6.5 Automatically Defined Functions

Even though the value of ADFs has been questioned in a register machine approach such as AIM-GP (see Section 12.7 below), it may have benefits in connection with homologous crossover. Previously ADFs have been implemented by calling a special subfuncion, a *leaf function*, which then in turn calls one of a fixed number of ADFs in the individual, see Figure 12.10. The reason for having an extra function in-between is that necessary boundary checks can be made in the leaf functions. Calling a function represents a considerable overhead. ADFs can be implemented more elegantly with blocks.

We need two blocks to realize ADFs. One block containing a $call$ and one containing a $return$ instructions. These blocks are then arbitrarily inserted into the individual. To work properly during evolution there must be control instructions in these blocks that check that there is no stack underflow or stack overflow. (The allowed calling depth need only be a few levels.) The blocks are initialised to call forward 5 or 10 blocks. A second check (also within the instruction blocks) is therefore needed to make sure that no call is made past the boundary of the individual. In this way no special ADF structure in the individual is necessary. Instead the subroutines are chaotically intermixed in a single individual. The benefits are a larger freedom for the system to control how many ADF's will be used and in what way. The block approach is also faster since multiple function calls are eliminated.

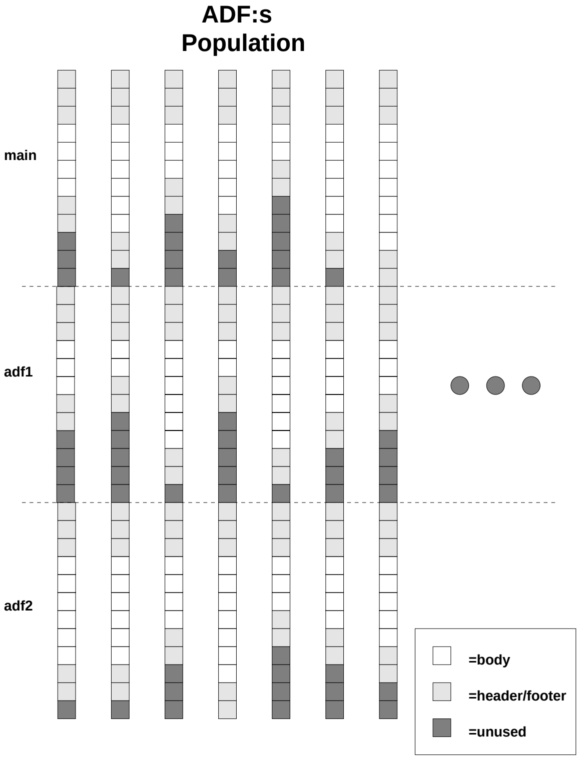

Figure 12.10
The structure of a population consisting of individuals with two ADF parts and a main part in AIM-GP

12.7 AIM-GP and Tree–Based GP

The greatest advantage of AIM-GP is the considerable speed enhancement compared to an interpreting system, as discussed above. An interesting question is whether the performance of the register machine system is comparable on a *per-generation or per-evaluation basis*.

AIM-GP has possible advantages over Tree based Genetic Programming other than speed. Consider that a four line program in machine language may look like this:

```
(1)   x=x-1
(2)   y=x*x
(3)   x=x*y
(4)   y=x+y
```

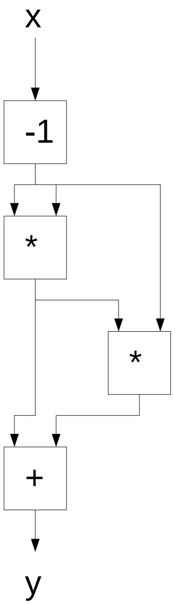

Figure 12.11
The dataflow graph of the $(x-1)^2 + (x-1)^3$ polynomial

The program uses two registers, x, y, to represent the function. In this case the polynomial is:

$$g(x) = (x-1)^2 + (x-1)^3 \tag{12.1}$$

The input to the function is placed in register x and the output is what is left in register y after all four instructions have been executed. Register y is initially zero. Note that the registers are variables that could be assigned at any point in the program. Register y, for example, is used as temporary storage in instruction number two ($y = x * x$) before its final value is assigned in the last instruction ($y = x + y$). The program has more of a graph structure than a tree structure, where the register assignments represent edges in the graph. Figure 12.11 shows a dataflow graph of the $(x-1)^2 + (x-1)^3$ computation. In that figure, the machine code program closely corresponds to this graph. Compare this to an equivalent individual in a tree–based Genetic Programming system as in figure 12.12. It has been argued that the more general graph representation of the register machine is an advantage compared to the tree representation of traditional GP. For this reason there is less need to use an explicit ADF feature in AIM-GP

In fact, the temporary storage of values in registers may be seen as a “poor man’s ADF” The reuse of calculated values can, in some cases, replace the need to divide the programs

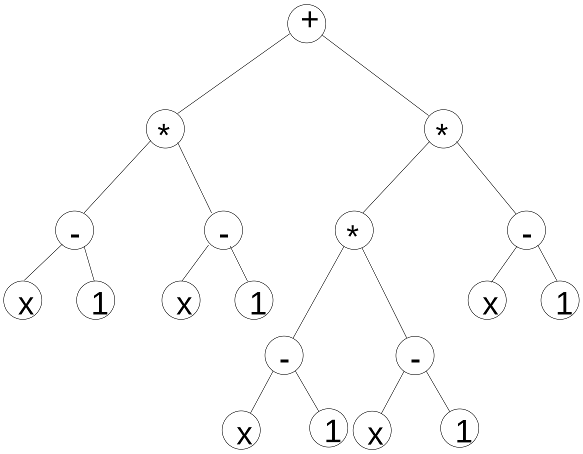

Figure 12.12
The representation of $(x-1)^2 + (x-1)^3$ in a tree–based genome

into subroutines or subfunctions. Reuse of useful instruction sequences is repeatedly observed in evolved AIM-GP programs.

To determine if this theoretical advantage of AIM-GP over tree-based Genetic Programming has any empirical support, we carefully tuned two Genetic Programming systems, one standard tree–based as well as one register based and evaluated their performance on a real world test problem. This problem is from the speech recognition domain and has been used previously as a benchmark problem in the machine learning community, with connectionist approaches. The problem consists of pre-processed speech segments, which should be classified according to type of phoneme.

The PHONEME recognition data set contains two classes of data: nasal vowels (Class 0) and oral vowels (Class 1) from isolated syllables spoken by different speakers. This database is composed of two classes in 5 dimensions [ELENA, 1995]. The classification problem is cast into a symbolic regression problem where the members of class zero have an ideal value of zero while the ideal output value of class one is 100.

The function set consisted, in both cases, of the arithmetic operator times, subtract, plus and the logical shift left (SLL) and logical shift right (SRL) operators. The selection method used was a steady state tournament of size four. Homologous crossover was not used. The population size was chosen to 3000 individuals and each experiment was run for 1000 generation equivalents.

Each system performed 10 runs on the problem and the average of the 10 runs was plotted. Figure 12.13 shows the average over 10 runs of the best individual fitness for the two systems.

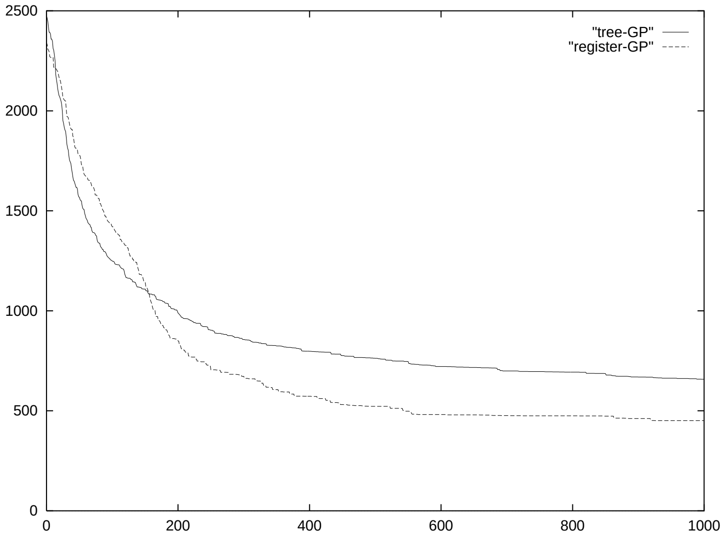

Figure 12.13
Comparison of fitness of the best individual with a tree and register based Genetic Programming system over 1000 generation equivalents

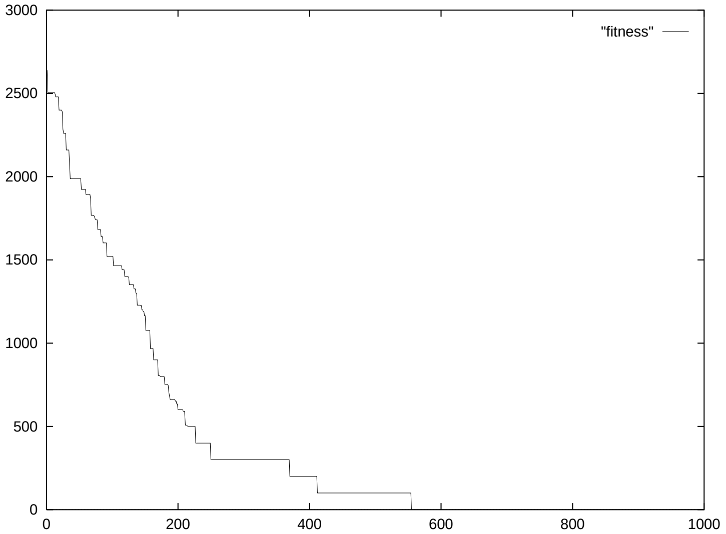

Figure 12.14
Evolution of best fitness over 1000 generation equivalents

The tree–based system starts out with a sharper drop in fitness but at generation 180 the register based system has a better fitness. The average of best fitness at termination after 1000 generation equivalents is 657.9 for the tree–based Genetic Programming and 450.9 for register based GP. This means that the average fitness advantage is 31% in favor of the register based system.

The results show that the register machine system, on this problem, converges to an equal or better fitness value than the tree–based system. These results suggest that AIM-GP could have advantages in addition to its superior speed.

12.8 Future Work

Many of the AIM-GP techniques currently in use are proven in practical applications. More thorough evaluations are planned. New additions to the system have opened-up completely new possibilities in several application areas:

- The introduction of blocks improves portability and we plan to exploit this by porting the system to embedded processors. Programming very complex tasks e.g. speech recognition is difficult to do in machine code with limited hardware resources. While AIM-GP has proven that it can evolve efficient solutions (as efficient short machine code programs) to such hard problems [Conrads et al., (1998)]. Applied in an inexpensive embedded processor such as the PIC, it could have many commercially applications.

- AIM-GP has previously been used in control domains such as on-line control on autonomous robots. We have begun work which will extend this domain to more complex walking robots. Autonomous robots need high processing capabilities in compact memory space and AIM-GP is therefore well suited for on-board learning.

- Genetic Programming differs from other evolutionary techniques and other “soft-computing” techniques in that it produces symbolic information (e.g. computer programs) as output. It can also process symbolic information as input very efficiently. Despite this unique strength genetic programming has so far been applied mostly in numerical or Boolean problem domains. We plan to evaluate the use of machine code evolution for text data mining of e.g. the Internet.

Other potential applications for AIM-GP are in special processors, such as:

- Video processing chips, compression, decompression (e.g. MPEG), blitter chips
- Signal processors
- Processors for special languages, for example, LISP-processors and data flow processors
- New processor architectures with very large instruction sizes

- Parallel vector processors
- Low power processors for example 4-bit processors in watches and cameras
- Special hardware, e.g. in network switching

12.9 Summary and Conclusion

We have presented additions to the AIM-GP making the approach more portable and enabling its use with CISC processors. Additions consist of blocks and annotations which enable safe use of genetic operators despite varying length instructions. Benefits of the CISC architecture are the large number of instructions in the instruction set, increasing the likelihood that the instructions needed for a specific application can be found. Complex instructions include LOOP instructions and special instructions for string manipulation. The use of the FPU further expands the directly possible instruction set by inclusion of important mathematical functions such as $SIN, COS, TAN, ATN, SQRT, LN, LOG, ABS, EXP$ etc. All these additions are important for the practical applicability of one of the fastest methods for Genetic Programming.

Acknowledgments

Peter Nordin gratefully acknowledges support from the Swedish Research Council for Engineering Sciences.

Bibliography

Banzhaf, W., Nordin, P. Keller, R. E., and Francone, F. D. (1998) Genetic Programming – An Introduction. On The Automatic Evolution Of Computer Programs And Its Applications. Morgan Kaufmann, San Francisco, USA and dpunkt, Heidelberg, Germany.

Conrads, M., Nordin P. and Banzhaf W. (1998) Speech Recognition using Genetic Programming. In Proceedings of the First European Workshop on Genetic Programming. W. Banzhaf, R. Poli, M. Schoenauer and T. Fogarty (eds.). LNCS Volume 1391, Springer, Berlin and New York.

Cramer, N.L. (1985) A Representation For Adaptive Generation Of Simple Sequential Programs. In Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications, J. Grefenstette (ed.), p.183-187.

Crepeau, R.L. (1995) Genetic Evolution of Machine Language Software, In Proceeding of the Genetic Programming workshop at Machine Learning 95, Tahoe City, CA, J. Rosca(ed.), University of Rochester Technical Report 95.2, Rochester, NY, p. 6-22.

ELENA partners (1995) Esprit Basic Research Project Number 6891, Jutten C., Project Coordinator. Document Number R3-B1-P.

Gruau, f. (1993) Automatic Definition Of Modular Neural Networks. Adaptive Behaviour, 3(2):151-183.

Huelsberger L. (1996) Simulated Evolution of Machine Language Iteration. In Proceedings of the first International Conference on Genetic Programming, Stanford, USA. J. Koza (ed.), Morgan Kaufmann, San Francisco, USA.

Koza, J. R. (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA.

Nordin, J.P. (1997) Evolutionary Program Induction of Binary Machine Code and its Application. Krehl Verlag, Muenster, Germany.

Olsson, J. (1998) The Art of Writing Specifications for the ADATE Automatic Programming System. In Proceedings of the Third International Conference on Genetic Progarmming. Morgan Kaufmann, San Francisco, USA.

Poli, R. and Langdon, W. B. (1998) On the Search Properties of Different Crossover Operators in Genetic Programming. In Proceedings of the Third International Conference on Genetic Progarmming. Morgan Kaufmann, Wisconsin, USA.

Salustowicz, R. and Schmidhuber J. (1997) Probabilistic Incremental Program Evolution. In Evolutionary Computation 5(2):123-141.