Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code

Peter Nordin
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Systemanalyse
D–44221 Dortmund
nordin@ls11.informatik.uni-dortmund.de

Wolfgang Banzhaf
Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Systemanalyse
D–44221 Dortmund
banzhaf@ls11.informatik.uni-dortmund.de

Abstract

The majority of commercial computers today are register machines of von Neumann type. We have developed a method to evolve Turing-complete programs for a register machine. The described implementation enables the use of most program constructs, such as arithmetic operators, large indexed memory, automatic decomposition into subfunctions and subroutines (ADFs), conditional constructs i.e. if-then-else, jumps, loop structures, recursion, protected functions, string and list functions. Any C-function can be compiled and linked into the function set of the system. The use of register machine language allows us to work at the lowest level of binary machine code without any interpreting steps. In a von Neumann machine, programs and data reside in the same memory and the genetic operators can thus directly manipulate the binary machine code in memory. The genetic operators themselves are written in C-language but they modify individuals in binary representation. The result is an execution speed enhancement of up to 100 times compared to an interpreting C-language implementation, and up to 2000 times compared to a LISP implementation. The use of binary machine code demands a very compact coding of about one byte per node in the individual. The resulting evolved programs are disassembled into C-modules and can be incorporated into a conventional software development environment. The low memory requirements and the significant speed enhancement of this technique could be of use when applying genetic programming to new application areas, platforms and research domains.

1 Introduction

A von Neumann machine is a machine whose program resides in the same storage location as the data used by that program. The name is after the famous Hungarian/American mathematician John von Neumann, and almost all computers today are of the von Neumann type. The fact that the program could be considered as just another kind of data makes it possible to write programs manipulating other programs and programs manipulating themselves. The memory in a such a machine could be viewed as an indexed array of integers and a program is thus also a collection of integer numbers. A program manipulating another program's binary instructions is just a program manipulating an array of integers. There are no higher level languages that directly support this meta manipulating with the appropriate tools and structures. In the few high level languages where it is, possible it comes as a side effect of the general freedom of memory manipulations. The language we have been using is C because it is efficient and gives the possibility to manipulate the memory where a program is stored.

In genetic programming, the goal of the system is to evolve algorithms or programs in a given language (Koza 1992). Most genetic programming systems use a tree structure for the representation of programs. The most used tree representation form is the S-expressions common in LISP. This representation guarantees evolved programs to be syntactically correct after the genetic operators are applied. In the original model (Koza 1992), the only genetic operator, apart from selection, is the subtree exchanging crossover.

A few experiments have also been performed with fixed length character string representation showing the general difficulties of that approach (Cramer 1985).

This paper describes the first successful method to

evolve a complex register machine with a genetic algorithm. Our register machine system is complex enough to evolve code for an existing hardware machine, making it possible for the processor to, through evolution, self-modify its binary code using no interpretation or intermediate code.

The implementation uses a bit string representation which is necessary when evolving binary code directly. We call this approach compiling genetic programming (CGP).

2 Compiling Genetic Programming

We have earlier demonstrated how to use an evolutionary algorithm to evolve binary machine language programs (Nordin 1994). This algorithm evolved fixed length binary strings and used uniform crossover, and mutation. The crossover was prevented from damaging certain bit fields within the 32 bits of instructions, which guaranteed that only valid machine code instructions where generated through crossover. The mutation operator had a similar protection feature.

We tried this simpler system on a number of problems. In (Nordin 1994) we also compared the performance of this Compiling Genetic Programming System (CGPS) to that of a neural network. We evaluated the two machine learning paradigms on a natural language task, where the goal is to find spelling heuristics to determine whether a word is a noun or a non-noun. The CGPS clearly outperformed the neural network in this example.

In these earlier experiments we only used arithmetic and logic operators between two registers and we only used a single procedure structure. This guaranteed that there were no unwanted effects during execution, no loops or out of bound jumps, etc. The evolved programs were functions limited to one input and one output parameter.

The system described below is a significantly more complete machine learning and induction system, capable of evolving Turing-complete programs. In contrast to the previous system, the new system merits:

- use of several machine registers
- dynamic allocation of population memory
- variable length programs
- multiple input parameters to functions
- unlimited memory through indexed memory
- automatic evolution of subfunctions
- if-then-else structures
- jumps
- use of loop structures such as: for,while,repeat
- recursion, direct and through subfunctions
- protected functions, e.g. division
- string functions and list functions
- linking *any* C-function for use in the function set

In addition, we wanted these goals to be met by a program written mostly in C-language. We also wanted the program to use *unrestricted crossover* at instruction boundaries. By unrestricted crossover we mean that the crossover acting on the strings of instructions should be able to work blindly, in the function body, without checking what kind of instructions were moved, and where. The advantages is that the implementation will be very efficient, because the algorithm will only consist of a loop moving a sequence of integers, something a computer does very well. The implementation will be simple and easily extendable because there will be a minimum of interaction and interference between parts of the program. Equally important is the fact that the program will be more easily ported to different platforms because the architecturally specific parts can be restricted to a minor part of the program. The crossover mechanism for instance, does not have to be affected.

The method of binary machine code manipulation should not be confused with the task of translating a genetic programming system into assembler code, which is how compiler work Instead this is a new way of making large-scale efficient meta manipulating programs.

2.1 Why Use a Compiling Approach

The use of a CGPS has several advantages over interpreting systems:

- execution speed
- compact kernel
- compact representation
- simple memory management

The efficiency improvement in compiling genetic programming comes from the deletion of all interpreting steps. Instead of using a structure defined by the programmer to be run through an interpreter, the algorithm here manipulates and executes the binaries directly. The efficiency benefit comes mostly from deletion of the interpreting steps, but also from the simplicity of manipulating the linear integer array that constitute the binary machine code program.

The lack of any interpreting steps makes the kernel of the algorithm more compact, because no definition of the interpreted language is needed. The machine code binary format is compact in itself which is im-

portant when working with large populations. The inherent linear structure of a binary code program forces the design of a system that uses simple memory management. The less frequent use of pointers and garbage collection requires a straight forward handling of memory that is an advantage in real-time applications for instance.

The approach of binary manipulating machine learning algorithms, building on the idea behind a von Neumann machine, is applicable on most computers, from the biggest super-computers to the small invisible systems in our cars, cameras and washing machines.

3 Implementation

In this section we describe the implementation of the CGPS. We start by presenting some basic facts about machine code representation of functions, and then continue with the description of the CGPS implementation.

3.1 The Structure of a Machine Code Function

Machine language is the sequence of integers that constitute the program that the processor is executing.By binary machine code, we mean the actual numbers stored (in binary format) in the computer. Assembly language is a higher level text-based language with a simple translation to machine language.

A function call in machine language has to perform three different subtasks:

- Program control transfer and storing the return address
- Save processor registers
- Transfer parameters

The most important instruction for functions and procedures is the `call` instruction. It is present in all kinds of processors and it works as a jump instruction that saves the memory address of the location it was jumping from. This will allow a `return` instruction to return back to this memory location when execution of the function is complete. Most `call` instructions save the return address in a special memory segment called the stack, but some architectures, such as the SPARC architecture, save it internally in a register in the processor. A `call` instruction is not sufficient to make a complete function call. The contents of the registers must be saved somewhere before the actual instructions of the function are executed. This assures that the called function will not interfere with the results in the registers of the calling function, and gives the called function the liberty to manipulate these registers itself. The most common place to store the contents of the calling functions registers is the stack. Some architectures, such as the SPARC-architecture, stores it inside the processor by a special *save* instruction. When the execution of the function is complete, the registers of the calling function have to be restored to allow this function to continue processing in the context it was working in before the function call.

The last task a processor has to perform in a complete function call is to transfer the input parameters to the function. Again, this is most commonly done by storing these values in the stack but it can also be done by using special registers inside the processor.

The structure of a function on the machine code level could be thought of as an array of integers divided into three parts: the header, the body and the footer. The header of a function does one or both of the save and transfer steps mentioned above. The header is fairly constant and is not manipulated by the genetic operators of the program. It is defined at an early initialization phase of the system.

The body of the function does the actual work that the function is supposed to carry out. When the body of the function is entered, all its parameters are accessible to it, and the registers from the calling function are saved. The genetic operators are only allowed to operate in the body of the function structure. The footer restores the registers, transfers the resulting values and executes the `return` instruction.

3.2 Self Modifying[1] Code in C language

So far we have only discussed concepts of machine code and assembler. Our implementation is, however, written in C. The intention is to write the genetic operators and kernel in a high-level language and only keep the manipulated structures in binary code.

In a computer, all data are represented as integer numbers. To make programming an easier task, there are additional data types and structures in a high-level program, for instance: strings, pointers, characters, bitmaps, etc. But all of these structures are translated into integers. To translate an object of a certain type into an object of another type, to cast it, is thus only an operation in the high level language. At the machine code level they are represented by the same integer.

[1] In this paper the term *self-modifying* means a program that manipulates its binaries, as common in a low-level programming context

Our approach is built on the casting of a pointer to an array and a pointer to a function. The pointer to the array is used when manipulating the program structures and the pointer to the function is used when the individual is executed. In the machine it is the same integer, but we write the cast in the code to satisfy the compiler. *This cast is the core principle of binary manipulating machine language.* We change an integer array, and then we cast it into a function pointer and execute it.

Below we give an example of how the pointer to *sumarray* is casted by *(function_ptr)* to a function pointer and called with the value of two and three.

```
typedef unsigned int(* function_ptr) ();

unsigned int sumarray[]={2178940928, 2953183257,
                         2179465216, 2177359880, 16777216};

a=((function_ptr) sumarray)(2,3);
```

These three lines of C-code will compute the sum of two and three and put the result in variable "a" by calling a function defined by the integers in the *sumarray*. The first line of the code declares the function pointer type. The second line declares an integer array containing integers for the instructions in the *sum* function. The last line converts the address of the *sumarray* from, a pointer to an integer array, to a pointer to a function, and then calls this function with the arguments two and three. The result from the function call is placed in variable "a".

This C-code illustrates the basic principles of compiling genetic programming. The *sumarray* can be changed just as any other array as long as its content is a valid function.

4 The SPARC Architecture

We have chosen to work with the SPARC architecture and SUN workstations because it is one of the most widely used architectures in the research community, and its hardware supports safe multi-tasking. Other simpler versions (i.e. without ADFs) of the CGPS exist for other architecture platforms. For a discussion about portability and advantages of different architectures see (Nordin 1994).

We will divide our introduction into two parts: one describing the registers, and one describing the instructions of the processor.

4.1 Registers

The key to understanding genetic programming with a register machine is understanding the use of registers and their role in function calls. The most important of the different kinds of registers are the so called windowed registers. It is between the windowed registers that almost all arithmetic and logic operations take place. The windowed registers are divided into four classes: in-registers, out-registers, local registers and global registers.

There are eight registers in each of these classes. When a function is called in a register machine, the registers from the calling function must be saved. In our SPARC implementation, a *save instruction* is executed which copies the contents of the *out* registers into a new set of corresponding *in* registers. Register *out0* is copied into *in0*, *out1* copied into *in1*, etc. Here *in* is a new input register owned by the called function. The values in the old *in* registers are kept or saved. The processor has an internal storage for a few sets of registers such as this and if this limit is exceeded the hardware and system software will save the contents in memory. For the user, this mechanism can be thought of as a small internal stack. It is important to note that a save instruction copies the contents of the calling functions *out registers* into the called functions *in registers*, while the restore instruction copies the contents of the returning function's *in* registers back into the calling function's *out* registers. When a function wants to call another function and pass some variables, it places the parameters in its out registers and makes a call. After the function call, the returning values can be found again in the out registers.

The local registers are local to the function working for the moment and is used for temporary storage. A fresh set of local registers are provided when a new function is called, but there is no special transfer of values into these registers. The global registers are always the same. They keep their meaning and content across function calls, and can thus be used to store global values. There are no alternative sets of these registers as there are with the in, out and local registers.

Registers *in6, in7, out6, out7* are by convention used to store stack and frame pointer as well as the return address in a function call and should not be used directly by the algorithm. The registers left to be used for arithmetic operations are *global register one, in0-in5, out0-out5, local0-local7.*

4.2 Instructions

SPARC is a RISC architecture with a word length of 32-bits. All of the instructions have this size. The 32 bits define the instructions, operations, operators and constants to be used. We use the following subset of SPARC machine code instructions in CGPS:

- ADD, Addition
- SUB, Subtraction
- MUL, Integer multiplication
- SLL, Shift left
- SRL, Shift right
- XOR, Exclusive or
- AND, Logical And
- OR, Logical Or
- Call local register 0-7

The first eight instructions are arithmetic and logic instructions. They contain the information of the destination register, the two operands and the arithmetic or logic operation to be used. One of the two operands can be a small constant, the other has to be another register. A single 32 bit instruction thus contains the information of four nodes in a conventional genetic programming system, one node for the operator, two nodes for the two operands and one node for the destination of the result. This approach is thus quite memory efficient, using only four bytes of memory to store four nodes.

We use eight different call instructions, one for each of the local registers. The call instruction jumps to addresses given by the content of the local registers in the function. The local registers are thus reserved for this usage in our implementation, and have to be initialized in the header of every function, to make sure that they point to a function in use. These instructions are the basis for external functions, automatically defined subfunctions, controls structures and memory manipulations. The number of these instructions used in the initialization is determined by the number of ADFs and external functions. The maximum number of external functions and ADFs are in this implementation limited to eight, but this figure could easily be extended.

4.3 Leaf Procedures

The above described properties of the call and save instructions make it possible to use two kinds of procedure. The first type of procedure is a full procedure that uses the *save* and *restore* instructions, and the procedure is consequently allowed to use and manipulate *in, out* and *local* registers. The *save* and *restore* instructions, however, consumes some processor cycles. If there is room inside the processor's "internal stack", the time consumption is moderate, but if storage in memory is needed it will use several processor cycles. The second type of procedure is called a leaf procedure, because it cannot call another procedure and is thus a leaf in the procedure structure of a program. A leaf procedure does *not* perform a save operation and works with the same set of registers as the calling procedure. To avoid interfering with the content of the calling procedure, it only manipulates the out registers, which are assumed to be destroyed by the compiler across function calls. One of the elegant consequences of this technique is that the calling procedure does not have to know what kind of procedure it is calling. This means that linking of procedures works normally. The difference in the leaf procedure is that it only manipulates the out registers, it does not use *save* or *restore*, and it has a special `return` instruction that looks for the return address in "out" register seven instead of "in" register seven.

In some of the examples in Compiling Genetic Programming below, we use a hybrid of the normal procedure and the leaf procedure, where we manipulate the "in" registers in a leaf procedure. This type of leaf-procedure is used for the efficient and safe implementation of, for instance, recursion, indexed memory, loops, if statements, etc.

5 The Evolutionary Algorithm

The evolutionary algorithm works directly on the individuals represented by a string of bits, where every 32 bits defines an instruction. The genetic operators are selection, crossover and mutation.

5.1 Crossover

Crossover is only allowed between the instructions at a locus that is a multiple of 32. The *crossover* operator selects two such points in two individuals and then swaps the instructions between them. This approach enables us to cut and splice blocks of instructions into the individual without the risk of generating programs with invalid syntax. The crossover operator is also prevented from manipulating the header and footer of the function. Only the body of the function can be affected. If ADFs are used, then one subfunction is selected from every individual and the crossover is performed between these two selected subfunctions. The principle of *unrestricted crossover* is crucial to our design choices in the system. Our goal is that the crossover operator acting on the strings of instructions should be able to work blindly without checking what kind of instructions are moved and where. It is easy to find examples of instructions and combinations of instructions where these properties do not hold. The normal *call* and *branch* instructions contain an offset which is added to the program counter. If such an instruction is moved to another memory position then the call or branch will point to another

memory location, where there might not be any valid function. The SPARC architecture does not have calls with absolute addresses, which still would work after crossover. Instead, we use calls to an address specified by a register. The value in the register will be the same even if the instruction is moved by crossover. Machine language is quite flexible and the restriction of instruction use is possible without abandoning the goal of Turing-completeness.

5.2 The Mutation Operator

The mutation operator selects an instruction at random and checks whether it has a constant part or if it is only an operation between registers. If it has a constant part, a bit in this constant is mutated an. It can also mutate the index defining the source and destination registers of the operation. If the instruction does not have a constant part, the instruction's type, source and destination registers are subject to mutation. The bits in calls and jump instructions are not mutated but may be swapped for other instructions. This division of the mutation operator in steps ensures that valid are instructions are created.

6 Initialization

The memory used for the individuals in the arrays is a linear array of integers. The array is divided into blocks determined by the system parameter *maximum length*. A fixed maximum length is thus reserved for every individual. If there are automatically defined functions, then this memory is allocated for every subfunction according to the maximal number of ADFs. The program and its subfunctions, then vary in length within these boundaries.

The advantages of this paradigm is that we have simple memory management, without garbage collection, etc. There is, furthermore, a constant, easily calculated, memory usage once the system is initialized. This could be a requirement in real-time systems and in systems with limited memory resources.

The initialization of the header consists of two parts; one that is fixed and one that depends on the number of ADFs and external functions. The fixed part of the header is a save instruction. The second part of the header initializes the local registers. The local registers are used to store jump addresses to ADFs and external functions by load instructions that load the appropriate addresses into these registers. The function body is initialized by randomly selecting instructions from the function set, including the call instructions using local registers. If an instruction is an arithmetic instruction then input and output registers are chosen for operands and destination, according to the parameter for *maximum number of registers* supplied by the user.

7 Additional features of the system

7.1 Automatically Defined Subfunctions

Automatically defined functions (ADF) are modularisations or subfunctions within an individual that spontaneously emerge during evolution. An ADF call has the same structure as a function call. It consists of instructions for transferring the function input parameters into the *out* variables and a call instruction.

The linear array of integers that constitute an individual is divided into a number of equal size segments. The number of segments corresponds to the *maximum number of ADFs* parameter. Every such memory section is an ADF. The ADF is organized just as the main functions with a header, a footer, and a function body.

In the header of the ADF the local registers are initialized to contain the addresses of the other ADFs that could be called from this ADF. When the recursion option of the system is switched off the local registers are only loaded with the addresses of ADFs higher up in the hierarchy. Remaining unused local registers are initialized with the value of a dummy function. With this scheme it is possible to allow *unrestricted crossover* between individuals and between subfunctions, because the local registers will always be initialized in the header of each subfunctions to a correct address. The "call local register" instructions can thus be freely copied and moved in the population and between ADFs at different levels.

7.2 Recursion

When recursion is used the local registers are not only initialized to the values of ADFs higher up in the hierarchy, but also to the current function themselves, enabling the function to call itself. whether we use recursion or not, which makes the implementation less complex.

In any approach we use for recursion, there will always be the problem of infinite chains of instructions calls. The halting problem makes it impossible to know in advance which recursive functions will stop and which will not. The solution to this problem is to have a global variable that is incremented for every time a function is called (Brave 1994). If a certain limit value is reached, then the execution of the individual is abandoned. The code for this checking is placed in the

header of each function

7.3 Leaf Procedures as Program Primitives.

Loops are implemented in a manner similar to recursion. A leaf procedure is used which performs a test of a variable. Depending on the outcome of the test, a loop branch is either performed or not. The test could, for instance, be whether the last performed arithmetic instruction had the value zero as a result or not. This is tested by checking the zero flag in the processor. Many loop structures could be formed by checking other integer conditions from the last instruction. The branch in a loop, is made with a `return` instruction. Recall that the `return` instruction is nothing but a "longjump" instruction, jumping to the address of a register to which a constant is added. This constant could be positive or negative. The normal `return` instruction jumps back to the address given by the content of register *out7* or *in7*. When we use this instruction in a loop we decrement *out7* before the jump and the program control jumps back to an address before the call to the "loop leaf procedure". The loop leaf procedures, also decrement and check a global variable, giving *the number of maximum loop iteration.* The fact that the different call addresses so far in the execution of the individual are all stored and accessible to the leaf procedure makes it possible to construct a large number of control structures. With this method it is possible to define efficient external leaf procedures that implement ordinary jumps, all kinds of loops, indexed memory, if then else structures, and protected functions.

7.4 External functions

An external function is a function that is not part of the processors instruction set, i.e. string and list functions. It is possible to incorporate any C-function into the function set by a simple chain of compilations and linkings. Any C-module could be compiled and linked into the system. This feature means that the CGPS is as general as any other GP systems and the terminal set, for instance, can be filled with complex functions such as bitmap manipulations, file processing etc. Many external functions[2] receives a pointer to its input data and returns a pointer to the output. [3]

[2]The external procedures could also be the primitives of any Turing-machine showing the turing-completeness of the system.

[3]The external procedures could also be the primitives of any Turing-machine showing the turing-completeness of the system.

7.5 C-language Output

The current system has a disassembler that translates the generated binary machine code into C-language modules. The feature enables the system to be used as a utility to a conventional software development environment. The following example shows the output from the disassembler system:

```
unsigned int fn1(a,b)
register unsigned int a,b;

{
  a = a + b;
  a = a * 2;
  return(a);
}


unsigned int fn0(a,b)
register unsigned int a,b,c;
{
  while( a > 0 )
  {
    c = fn1(c,b);
    a = a - 1;
  }
  a = c + 0;
  return(a);
}
```

The disassembler could also be requested to produce assembler code.

8 Applications

The compiling method will be specially suited for applications of genetic programming within areas that require:

- high execution speed demands
- real-time learning
- large population sizes
- low end architectures, consumer electronics
- well defined memory behavior

These properties could be of use when applying genetic programming to new application areas, platforms and research domains.

9 Speed Evaluation

The original GP systems were written in LISP, a language well suited for meta-manipulations. Several other implementations have been presented in other high level interpreting language, for instance Mathematica. Recently a few implementations of interpreting GP system in C has been presented (Tackett 1994).

Tackett has evaluated his elegantly written SGPS C-system and measured its performance against a LISP system. His conclusion is that the interpreting C-language GP system is about 25 times faster.

We have in turn evaluated the execution time differ-

ences between the SGPS and our compiling genetic programming system. The tests were performed on a SPARC station IPX and the problem was symbolic regression of polynomials. The size constraints were adjusted to be as equivalent as possible when the SGPS worked with tree structures and the CGPS with binary strings. The function set contained the addition, multiplication and subtraction operators. Ten fitness cases were used with a population size of 1000 individuals. By speed we mean the number of populations executed per minute.

The result showed that the SGPS system proceeded about 8-11 generations per minute while the compiling system proceeded between 700-1200 populations per minute[4]. For some experiments, the speed enhancement for the CGPS was a 100 times. On average, the compiling system was 60 times faster than the interpreting C language system. This means that the speed up compared to LISP implementations is in the magnitude of 1500-2000 times.

These figures should, however, be seen as an indication only, considering that the two systems have different representations and features.

10 Summary

In this paper we have presented a new technique for genetic programming and the evolution of Turing-complete programs for a register machine. The technique has a number of attractive properties. The use of register machines and binary string representation enables the use of the lowest level language of the processor in a computer system, resulting in a speed enhancement of the genetic programming system by a factor of 1500 compared to implementations in an interpreting language. The memory requirements are low and constant during evolution of both the kernel and the population. A population of a million individuals can reside in 40MB of RAM memory, and the kernel only needs 30KB. We believe that this new genetic programming technique could be of use to expedite the emergence of new real-life application areas and open up research in new domains.

Acknowledgments

The authors would like to thank John Koza, and Walter Tackett for the use of their GP systems, from the GP-archive. Thanks also to Sami Khuri for valuable comments to this paper. This research has been supported by the Ministry for Wissenschaft und Forschung (MWF) of Nordrhein-Westfalen, under grant I-A-4-6037.I .

[4]The variation in the CGPS seemed to be depending on the frequency of multiplication instruction in the solutions.

References

J. Koza (1992) *Genetic Programming*, Cambridge, MA: MIT Press.

N.L. Cramer (1985). A representation for adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pp183-187

J.P Nordin (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA: MIT Press.

W.A. Tackett (1994) Recombination, Selection and the Genetic Construction of Computer Programs. Dissertation, Faculty of the Graduate School, UCLA, CA.

S. Brave (1994) Evolution of a Recursive Program for Tree Search Using Genetic Programming. Stanford University, Stanford, CA.