MATHEW CARR – MSc. Project Design
Using GP to evolve code in a compiler

Summary of Proposal

Statement of Problem

Compilers are computer programs that translate one language to another [1]. The purpose of a software compiler is to transform an input program in the form of some high-level, human readable language to an output program in the form of object code or machine code executable by some real processor or virtual machine.

High level languages are designed to allow the user to specify the solution to a problem in terms of the problem space, rather than in terms provided by or implied by the choice of target machine. It can be very difficult to modify or maintain programs written in assembly language or machine code.

Almost all executable software is produced by means of a compiler. It is very rare for machine code to be directly manipulated by a human programmer, except for the development of software for uncommon architectures where a compiler may not be available, or in the design of programs which meta-manipulate machine code, such as the assembler function of a compiler. As such, a considerable amount of research has been invested in improving the quality and efficiency of compiler software.

Where compilers are available, they may not be fully able to automatically exploit specialised functionality available to advanced architectures. This functionality includes complex vector mathematics, and operations on large homogeneous data sets. These functions are highly valuable in applications such as image and sound processing. However, to be used effectively, the programmer must manually design the software to use these functions and invoke it explicitly. This places a dependency on the architecture, making it harder to reuse the same software elsewhere.

The code generation phase of compilation generates object code in the target architecture given an input program in the form of some intermediate representation. It is during this phase that the properties of the target architecture become a significant factor in the selection of optimisations. Therefore, the quality of the output object code is highly dependent on the sophistication of the techniques used during the code generation phase.

Genetic Programming (GP) [2] is an evolutionary computation technique that can automatically solve problems without requiring the user to know or specify the form or structure of the solution in advance [3]. GP typically works upon, and produces, tree structures, such as the S-Expressions used in the LISP language or other graph-based structures.

GP can be used to automatically induce computer programs, given a high level statement of the problem to be solved and a method for determining the suitability of a given candidate solution. GP searches the space of possible computer programs through the use of a model of evolution through natural selection.

First, a pool of random candidate programs is generated. Then, the suitability of each of these programs is calculated. A new 'generation' of candidate programs is produced by repeatedly selecting several highly suitable candidate programs to participate in 'crossover' or 'mutation' operations. The crossover operation combines parts of two source programs to produce a 'child' program. The mutation operation produces a new program by randomly altering some component of a source program. The suitability of the new generation of candidate programs is then calculated and the process is repeated until a candidate program of sufficient suitability is evolved.

Linear Genetic Programming (LGP) is an adaptation of GP techniques for use in working on linear structures. As such, it is suitable for the direct manipulation of assembly language or machine code instructions [8]. It is proposed that LGP may provide a means to accelerate or augment the code generation phase of compiler software.

The translation of a program provided in the form of an intermediate representation into the form of a low level machine code program can be expressed as the search in the space of possible machine code programs for a program with semantics similar to those expressed by the intermediate representation. The degree of similarity required depends on the program, there may be optimisations may render certain computations irrelevant. LGP provides the means to search such a space.

It has been observed that GP techniques are routinely successful where in situations where a number of properties are evident [3, page 111]. The problem identified in this document exhibits some of these traits:

Finding the size and shape of the ultimate solution is a major part of the problem: In this problem, the length of the (optimal) solution program is not specified in advance. The optimal solution program may not be shortest in terms of instruction as not all instructions may have the same 'cost' in some measure.

Significant amounts of test data are available in computer readable form: In this problem, this is the case. All input data is already in computer readable form, as is the output data form.

There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions: In this problem, developing an interpreter for a machine code language is not a difficult task. If the target architecture were a real, physical processor, then the processor itself may be used to execute the programs directly, quickly and exactly.

Small improvements in performance are routinely measured (or easily measurable) and highly prized: All factors affecting the suitability of a program, including program length, memory length and required processor time are all easily measurable. Additional suitability measures are proposed in the project design.

Previous Research

Koza has produced significant research into GP techniques in the induction of mathematical expressions, computer programs and other structures [2, 4, 5, 6]. This series provides the basis for a significant amount of research into GP. Koza does not attempt to provide formal proof of the ability of GP as a problem solving technique, but provides a large amount of promising experimental data. In this series, it is argued that GP techniques provide the means for human-competitive intelligence, due to the power of GP in program induction.

Huelsburgen [7] showed that GP techniques can be used to generate programs for a simplified virtual machine. In this paper, GP was used to evolve a program capable of arithmetic operations and iteration. The techniques used were shown to be superior to a random search of the program space.

Nordin and Banzhaf [8, 9, 10, 11] have demonstrated an approach for the direct induction of binary machine code programs using LGP. This approach is suitable for many applications, including image processing, sound processing, robot control and plant control. In this approach, programs are manipulated directly in the form of native machine code of the SPARC processor. The fitness measure is application specific and based upon considering the results of program execution on the environment.

The approach described in this document is similar to this method, though a simplified theoretical target architecture is considered, an interpreter is used as opposed to direct execution, and the fitness measure is based upon considering both the output of the program and the effects of execution of individual instructions and the syntactic structure of the program.

Design

Summary

The aim of this project is to investigate Linear Genetic Programming (LGP) techniques and determine their applicability to the task of code generation within a software compiler.

To this effect, software will be produced to allow experiments to be performed that use LGP to transform an input program in the form of a parse tree and symbol table pair into the form of an instruction string.

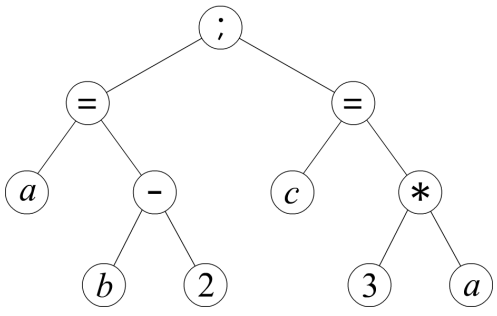
The effectiveness of the methods will be evaluated by developing a series of typical, simple programs with a range of complexities and sizes and instructing the evolutionary system to develop appropriate solution programs. The resources required to produce suitable solutions will be measured.

Parse Tree and Symbol Table

The source language considered in these experiments is a simple high level imperative language, similar to C. To begin, this language will only feature sequences of statements containing arithmetic operations between integer constants and named variables. The only data type used within the language is the 32-bit signed integer. The exact syntax of this language is not important, however, as this project will manipulate the parse tree intermediate representation produced during compilation. Where a parse tree is shown in this document, high level language source code will be shown to ease reading.

It is assumed that the parsing and lexical analysis stages of compilation have been completed previously, and the complete results of these operations are available for use.

For the purpose of these experiments, an input program is given as a parse tree and symbol table pair. A parse tree is a directed, pointed acyclic tree. A visualisation of a possible input program parse tree is shown below, together with the high level language program it encapsulates.




```
a = b-2;
c = 3*a
```

The interior nodes of the graph denote language constructs such as sequencing (represented here by a semicolon character), assignments, branches and loops; and arithmetic operators such as addition, subtraction, division and multiplication. The leaf nodes of the graph denote variables (represented here by a node showing the symbolic name of the variable) or constants. Constants are treated no differently than variables, other than the setting of a flag in the symbol table.

The program expressed by this parse tree can be executed in a top-down manner. Any node can be invoked to return the numeric value of the sub tree in the current state at the time of invocation. To execute the program, the value of the root node is requested in the context of some given input system state. A system state in this context refers to the values of all symbolic variables. This system state is global and shared throughout execution of the tree. As such, the state may be altered during execution (for example, as a result of the assignment operation), and this altered state subsequently inspected. For sequencing nodes, the left hand side child is evaluated first, followed by the right hand side child and this value returned. These semantics are similar to the comma operator in C.

This tree has the effect of first calculating the value of the expression `b-2` and assigning this value to the variable a, and then calculating the value of the expression `3*a` and assigning this value to c.

We may evaluate the program considering one instantiation of an input system state to produce a single output system state. That is, numerically. We may alternatively evaluate the program symbolically, across all input states, to provide an algebraic representation of the system state to system state mapping given by the program.

A symbol table contains the information about the variables referred to in the input program. The symbol table is stored internally as a mapping from unique keys (unsigned integers) to structures containing data defining the nature of the variable.

This data includes:

- The symbolic name of the variable.
- The scoping of the variable in the context of the program. The parse tree may represent an isolated subroutine apart from calling program. In this case, certain variables may exist only within the scope of the subroutine.
- The constant nature of the variable.

For the purposes of this project, we consider only variables of signed 32-bit integer type.

Target Architecture

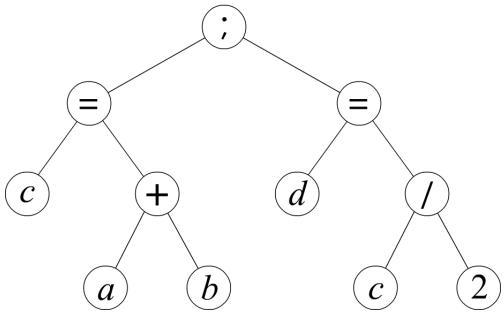
The target architecture for the evolution of programs is a simple register machine consisting of a number of general purpose registers, capable of storing 32-bit signed integers and available to the inserted program, and a symbolically accessible memory: storage for 32-bit signed integer values that may be addressed by a symbolic name for read or write access. Integer indexed memory is possible but not currently investigated. Software has been created that allows for non-interactive or

interactive (user-stepped) interpretation of programs written in the low level language for this architecture.

A simplified, RISC-like low level language is specified for the purposes of this project. An instruction in the low level language consists of an operation and a number of operands whose quantity and nature are defined by the choice of operation.

For example, an instruction performing addition may appear as ADD 2, 0, 1. This instruction consists of the operation (represented by the readable mnemonic 'ADD'), the number of the register into which the value of the result of the addition should be placed (the 'destination register'), and the numbers of the registers from which the values to be used in the addition should be read (the 'source registers'). This instruction stores into register 2 the value produced by adding up the values currently stored in registers 0 and 1. Instruction strings are stored in memory as an array of 32-bit signed integers for ease of manipulation.

Possible programs performing the operation '*Calculate the integer mean of the values of the variables a and b, and store the result in variable d*' are shown below in high level source language form, parse tree form and as an instruction string written in the low level language.




```
c = a + b;
d = c / 2
```

```
LOADS 0,   a    // load the value of variable a into register 0
LOADS 1,   b    // load the value of variable a into register 0
ADD   0, 0, 1   // add the values of registers 0 and 1 and store the result in register 0
LOADV 1,   2    // load the direct value 2 into register 1
DIVP  0, 0, 1   // divide (protected division) the value of r0 by the value of r1 and store the result in r0
STORS 0,   d    // store the value of register 0 into variable d
HALT            // end program
```

Note that the version written in the low level language does not refer to variable c at any time during its execution: the semantics of the program did not specify its inclusion. An optimising compiler may produce machine code that removes the need to directly store or consider the intermediate calculation; certain architectures provide a 'barrel shift' that can augment a standard

arithmetic instruction with a bitshift operation. This may only occur if it is determined that the variable c is not required subsequently. The required information may be obtained by inspecting the scope of variable c, defined in the symbol table.

In this project, Linear Genetic Programming (LGP) will be used to attempt to induce programs that have the same semantics as an input program in the form of a parse tree and symbol table pair. It will do this by first producing a random population of candidate programs in the low level language, calculating a value representing the suitability of each program, then repeatedly combining and modifying programs with high suitability with the aim of iteratively producing successively more suitable programs until a program that is sufficiently suitable is generated. The calculation of the suitability value, referred to henceforth as 'fitness' to mirror genetic programming literature, will include the degree of correlation to the semantics of the input parse tree program, and several other measures reflecting the 'sanity' present in the candidate programs.

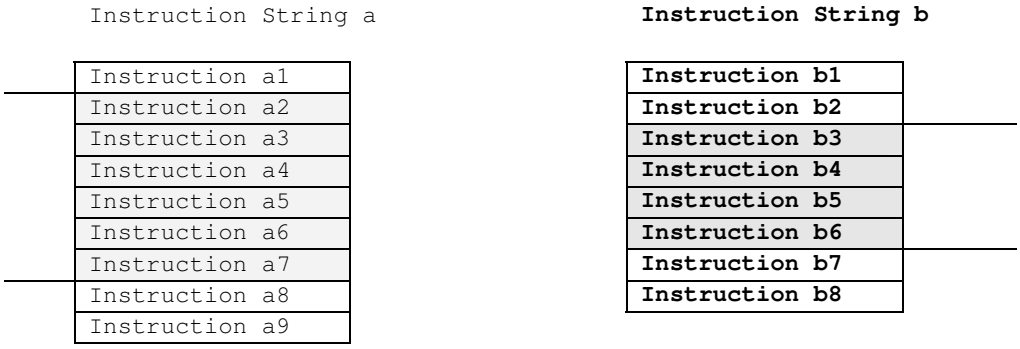
Genetic Operations

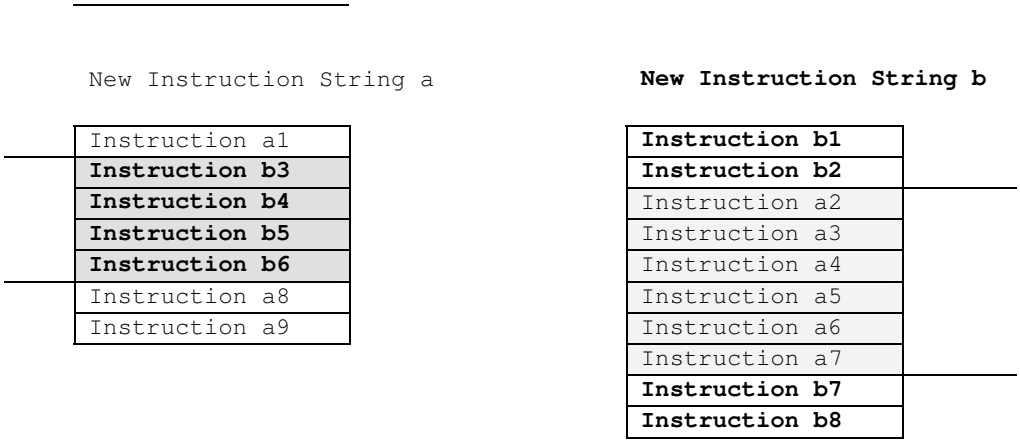
Tournament selection will be used to select programs from the population for application to the available operations. In tournament selection, N programs are randomly selected from the population to participate in the tournament, then from this set the M programs with the highest fitness values are passed to the operator.

The population model used in this system is a steady state model where the number of candidate programs in the population will remain constant throughout. For each new program produced by a genetic operation, a tournament is used to choose a random program with low fitness to be removed from the population.

There are two operations available to the evolutionary system to manipulate candidate programs. These actions are selected randomly according to rate parameters given to the evolutionary system.

The crossover operation combines the contents of two source instruction strings to produce two new instruction strings, which are then inserted into the population. Transition points are randomly placed within the two instructions. The new strings are constructed by copying instructions from first string until the first transition point, then copying instructions from the second string starting at the first transition point until the second transition point, then copying the remaining instructions from the first string starting from the second transition point until the end.





The effect of the mutation operation is to alter an existing instruction at random from the chosen instruction string. One component of the instruction is chosen at random from the available components (the number and nature of which will depend on the current operation). If an operand component is chosen, an operand of compatible type is chosen in its place. If the operation component is chosen, then the all components of the instruction are reinitialised.

In addition, the mutation operation may insert a new random instruction or remove an existing instruction chosen at random from the string.

Fitness Case Construction

The first stage is to establish the semantics of the input program, as these will be used to determine the degree of semantic correlation between the parse tree and a candidate low level language instruction string.

As stated previously, it is assumed that the parsing and lexical analysis stages of compilation have been fully completed. Therefore, we can assume that the symbol table contains sufficient information to determine which variables referred to within the input program are of importance.

We can draw from the symbol table the following attributes of any given variable:

- If the variable exists purely within some limited scope, solely as a holder of intermediate calculations, then candidate programs are permitted to freely read and write to this variable at any time.
- If the variable exists in a scope higher than that of the parse tree, then candidate programs are permitted to read the value of this variable, but they are not permitted to change it.
- If the variable has a value of use associated with it before execution begins.

The symbol table contains flags determining whether the value of each variable must change (as with d in the previous example), must not change (as with a, b and, implicitly, all variables other than c), or if it doesn't matter (as with c).

Intuitively, it appears that it may be possible to execute the low level instruction string in a symbolic fashion to produce an exact statement of the state to state mapping that the program performs. This statement could then be compared with a statement describing the semantics of the parse tree to determine the degree of semantic correlation. However, I believe that this approach would become unworkable for all but the simplest of cases.

Instead, a sampling of the possible values of the input variables is considered. This is analogous to compiler testing. It is believed that through a representative sampling of the input values, enough data will be available to construct a sufficient expression of the state mapping defined by the program. The number of samples will affect the accuracy of the expression, and therefore the accuracy of the output program. If there are too few samples, then the resulting program may exploit properties specific to the sample set. Increasing the number of samples will increase the time required to calculate the fitness of a candidate program.

For each sample set of input values, a fitness case is built by evaluating the input parse tree program. Each fitness case contains the values of the symbolic variables before execution, after execution, and a full record of all the intermediate evaluations that took place during evaluation. In addition, the number of calculations that were required, in total, to produce the result value is recorded as a heuristic measure of the complexity of calculation.

Fitness Evaluation

The fitness value for a candidate is calculated as the total of the degree of semantic correlation and additional values representing the ‘sanity’ of the candidate program, over all fitness cases.

To calculate the degree of semantic correlation, each candidate program is tested against each fitness case in turn. A virtual machine instance is created and reset; the input symbolic variable value set is copied from the fitness case into the variable machine symbolic variable memory, and the execution started. When execution terminates, the final values of the target variables in the memory of the virtual machine are compared against those from the fitness case. If the values of all target variables are exactly the same, no penalty is applied. If the value of a variable differs, a constant penalty is given together with a variable amount of penalty as a function of the error.

During the execution of the candidate program, a line by line record of the execution is produced. This record contains, for each executed arithmetic instruction, the operation that was performed, the register locations and values of the operands used and the register location and value produced as a result. For symbolic loads, only the destination register location and value is stored. It is possible to perform some analysis of the program without this record, but this may become complicated if conditional or jumping instructions are applied. With the introduction of the fitness cases as fixed points in the input space, it makes sense to continue in this vein by analysing the exact actions taken as a result of these input sets.

The complete record allows for the construction of a timeline showing the values of the registers after each instruction execution. If the registers are not reset to a known value before execution begins, this record shows which registers hold determinate values (which may be assumed to be of some use), or indeterminate garbage values (which will not be the same between executions, and therefore should not be relied upon in the output program).

Fitness bonuses and penalties are activated during analysis of the behaviour of the candidate program. These bonuses are designed to 'coax' the evolution of candidate programs towards those which a human programmer would consider productive.

The following productive behaviour is rewarded:

- Reading the value of a symbolic variable.
- Writing to the value of a variable that may change during execution.
- Writing to the value of a variable that must change during execution.
- Reading from a register whose value is determinate at the time of reading.
- Performing a calculation that results in a value that was encountered during construction of the corresponding fitness case. An added bonus is applied if the low level instruction correlates to the construct in the parse tree that was used.

The following counterproductive behaviour is penalised:

- Writing to a register and never subsequently reading it.
- Writing to a symbolic variable (other than one designated as an output) and never subsequently reading it.
- Writing to a register twice in succession without reading it in the interval.
- Reading from a register containing an indeterminate value.
- Performing a calculation upon indeterminate values.
- Writing an indeterminate value to any register.
- Writing an indeterminate value to any symbolic variable.

It is hypothesised that the crossover operation will combine programs that are correct 'up to a point' with genetic material from elsewhere to produce child programs that provide further functionality than either of their parents. It is also hypothesised that the mutation operations will act to 'repair' programs by removing or rewriting counterproductive instructions in programs, hence increasing their suitability.

The specification of a large number of fitness modifiers is intended to provide a more gradual fitness landscape. If only the error in the values of target variables is considered, the mapping from input candidate program space will be discontinuous. In such a space, many programs will share the same fitness value, and the evolutionary system will be able to offer little improvement.

With the above modifiers, there is the risk that the system may produce a program that calculates a useful value at some point during execution, and then attempt to improve such a program by repeating the segment that triggers the reward, resulting in a program that does nothing more than

calculate the same (albeit useful, or even necessary) value multiple times. Given time, such programs may dominate the candidate program population. To prevent this, the system can be configured to allow these rewards to be awarded only finitely many times per action, or per expected appearance of a result value. The complexity heuristic is designed so that a candidate program that correctly implements a multiple stage calculation is deemed to be more suitable than a candidate program that performs a simple calculation multiple times.

A ‘hit’ is recorded for a given fitness case if the resulting value for each variable is equal between the resulting state of the virtual machine memory after execution has terminated and the final state of the parse tree evaluation. If a ‘hit’ is recorded for each fitness case in the training set, the candidate program is tested against each fitness case in the test set. If a ‘hit’ is recorded for all fitness cases in the test set, the candidate program is judged to be a satisfactory solution: the evolutionary system terminates and returns the candidate program.

Review Against Plan

Progress

I have successfully implemented the program as described in the Design section of this document. The evolutionary system prototype software is capable of evolving low level instruction string programs that are semantically equivalent to short sequences of statements in the high level source code language. However, this process is time consuming and processor intensive due to the evaluation of many thousands of candidate programs against hundreds of fitness cases. No guarantee may be made that the process will succeed at all, due to the probabilistic nature of the linear genetic programming system.

Changes in Experiment Design

The design described in this document succeeds an earlier design which was later determined to be infeasible. In this previous design, the virtual machine system would not be able to directly recall the value of symbolic variables, and would instead be required to access their respective locations from an indexed memory. This was considered to be a difficult operation to induce, as candidate programs would have to first construct the constant values of the cells containing the variables to use in calculation, and then possible reconstruct these constant values to store the result of the calculation.

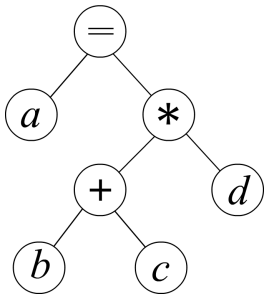
The virtual machine also featured a homogeneous instruction layout, where each instruction consisted of an operand and three signed integer arguments. These arguments would be interpreted to ensure closure and correct otherwise meaningless instructions (for example, the use of modulo to correct a negative or out of range register argument). This homogeneous instruction layout would enable the use of a novel form of instruction crossover that can act arbitrarily within instructions rather than being restricted to manipulating chunks of whole instructions. The crossover was free to cross over any amount of material between any two points in any two programs, with the requirement that the operation may not result in the ‘role’ of a value being changed, such as an operation becoming an operand or vice versa. This crossover was discarded as it was thought to be too destructive and unpredictable: it had the possible dual effect of copying complete instructions between programs, as well as producing new instructions at the boundaries of the copied material.

The removal of the intra-instruction crossover lead to a redesign of the virtual machine and accompanying low level language to a simpler architecture without indexed memory. By allowing only the mutation operator the ability to produce new genetic material, the evolutionary system could be directly restricted to only producing specific types of instructions. Without this restriction, many candidate programs were able to manipulate the register containing the program counter. The result of this was, that the most ‘suitable’ candidate programs in many populations were those that that immediately halted or jumped to an illegal address, as these actions were ‘preferable’ to random programs that attempted to access registers containing or perform arithmetic operations on indeterminate values.

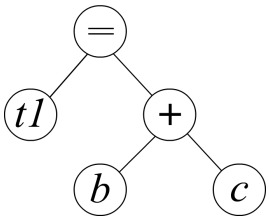
Expected Future Considerations

Due to the difficulty encountered inducing longer programs, I will investigate methods whereby the longer program may be instead considered as a series of shorter programs, which are then combined after completion. It may be easier to induce suitable low level candidate programs for these smaller programs, resulting in reduced requirements in processing time and memory usage.

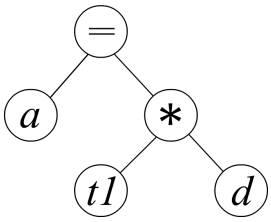
The granularisation of the input parse tree may be done automatically by splitting the program at each node in the parse tree (a sequence point in the high level source language) and extracting the nodes at that point into a new program.




```
a = (b+c) * d;
```




```
t1 = b+c;
```




```
a = t1 * d;
```

In this example, the additional variable t1 has been introduced to hold the value of the intermediate calculation b+c. This value is then used in to calculate the final value of a.

Candidate solutions for the smaller parse trees can be evolved individually, and this process may be parallelised. The resulting program combined by simple concatenation to produce a low level program satisfying the semantics of the original parse tree.

The resulting program would refer to numerous additional variables which were not present in the original parse tree. It is possible that this program may be used as an embryo to a final evolutionary system, with the intent that the system gradually improve the program through genetic operations until a final set of criteria is satisfied, such as the elimination of some or all of the automatically produced intermediate symbolic variables.

The two methods may be compared by considering the candidate solutions produced by each, together the processing time and other requirements necessary to produce them.

References

[1] Louden, K. C. (1997) *Compiler Construction: Principles and Practice*. PWS Publishing Company

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

[3] Poli, R., Langdon, W. B., and McPhee, N. F. (2008) *A Field Guide to Genetic Programming*. http://www.gp-field-guide.org.uk

[4] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA.

[5] Koza, J. R., Andre, B., Bennett III, F. H., and Keane M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman.

[6] Koza, J. R., Keane M. A., Streeter, M. J. Mydlowec, W., Yu, J., and Lanza, G (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

[7] Huelsbergen, L. (1996) Toward Simulated Evolution of Machine-Language Iteration, In *Conference on Genetic Programming 1996*, pages 315-320. Bell Labs.

[8] Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K. E. editor, *Advances in Genetic Programming*, chapter 13, pages 311-331. MIT Press.

[9] Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. Ph.D thesis, der Universitat Dortmund am Facherich Informatik.

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

[11] Nordin, P., Banzhaf, W., and Francone, F. (1999) Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Advances *in Genetic Programming 3*, chapter 12, pages 275-299, MIT Press.