Genetic Reasoning
Evolving Proofs with Genetic Search

Peter Nordin[*] Wolfgang Banzhaf[†]
Fachbereich Informatik
Universität Dortmund
44221 Dortmund, Germany

January 21, 1996

Abstract

Most automated reasoning systems relies on human knowledge or heuristics to guide the reasoning or search for proofs. We have evaluated the use of a powerful general search algorithm to search in the space of mathematical proofs. In our approach automated reasoning is seen as an instance of automated programming where the proof is seen as a program (of functions corresponding to rules of inference) that transforms a statement into an axiom. Genetic programming is a technique for automated programming that evolves programs with a genetic algorithm. We show that such a system can be used to evolve mathematical proofs in complex domains i.e. arithmetics and program verification. The system is not restricted to evaluations of classical two-valued logic but can be used with for instance Kleene's three valued logic in order to detect paradoxes that can occur in real life reasoning applications.

[*] email:nordin@ls11.informatik.uni-dortmund.de
[†] email: banzhaf@ls11.informatik.uni-dortmund.de

1 Introduction

We present an approach to reasoning that uses a genetic search heuristic to navigate and search in the space of true statements. An algorithm inspired by natural selection and survival of the fittest is used to search for proofs.
To use a genetic process as the architecture for mentally related activities could, at first, be considered awkward. As far as we know today, genetic information processing is not directly involved in information processing in brains, though the idea of genetics as a model of mental processes is not new. William James, the father of American psychology, argued just 15 years after Darwin published *The Origin of Species*, in 1874, that mental processes could operate in a Darwinian manner (William James 1890). He suggested that ideas "compete" with one another in the brain leaving only the best or fittest. Just as Darwinian evolution shaped a better brain in a couple of million years, a similar Darwinian process operating within the brain might shape intelligent solutions to problems on the time scale of thought and action. This allows "our thoughts to die instead of ourselves".
Evolutionary Algorithms mimic aspects of natural evolution, to optimize a solution towards a defined goal. Darwin's principle of natural selection and survival of the fittest, which is thought to be responsible for the evolution of all life forms on earth, has been employed successfully on computers over the past 30 years. Different research subfields have emerged such as , Evolution Strategies (Schwefel 1975,1995), Genetic Algorithms (Holland 1975) and Evolutionary Programming (Fogel, Owens & Walsh 1966), all mimicking various aspects of natural evolution. In recent years, these methods have been applied successfully to a spectrum of real-world and academic problem domains.
A mathematical function can for instance by optimized by a genetic algorithm that keeps a population of solution candidates which are reproduced by selection, modified by mutation and recombination during evolution until a sufficiently good solution is found.
A comparatively young research topic in this field is Genetic Programming (GP). Genetic Programming uses the mechanisms behind natural selection for *evolution of computer programs*. The search space is here the space of all computer programs. This contrasts other evolutionary algorithms which often optimizes real numbers or vectors of real numbers. In GP the structures optimized is a symbolic representation of a computer program. Instead of a human programmer programming the computer, the computer can modify, through genetic operators, a population of programs in order to finally generate a program that solves the defined problem. The GP technique, like other adaptive and learning techniques, has applications in problem domains where analytical solutions are incomplete and insufficient to the human programmer or when there isn't enough time or resources available to allow for human programming.
Methods related to GP were suggested as far back as in the 1950s (Friedberg 1958). For various reasons these experiments never were a complete success even

if partial results were achieved (Cramer 1985). A breakthrough came when Koza formulated his approach based on program individuals as tree structures represented by LISP S-expressions (Koza 1992). His hierarchical subtree exchanging genetic crossover operator guaranteed syntactic closure during evolution.
GP differs from other Evolutionary techniques and other "soft-computing" techniques in that it produces symbolic information (i.e. computer programs) as output. It can also efficiently process symbolic information as input. Despite this unique strength, has genetic programming mostly been applied in numerical or boolean problem domains.
In this paper we exploit GP's strength of processing purely symbolic information by searching in the domain of proofs.
Genetic Programming is thus a method for automated programming. A formal proof of a statement could be seen as a computer program and a theorem prover as an application of automated programming. The proof program is a list of inference functions transforming a statement to an axiom (or to a statement known to be false, i.e. a contradiction). Rules of inference are here seen as functions from theorems to theorems, like in for instance the programming language ML. The inference rules are rules that matches a part of a formula and rewrites it as something equivalent, or equally true. The formula $X + 0$ could for instance be replaced by X, as one of the axioms of Peano arithmetic tells us. This rule describes a function from theorem to theorem. In the same way the reverse is true and X could be replaced by $X + 0$, (but this is considered as an other function.)
This simplest form of theorem prover systematically applies rules of inference to construct all possible valid logical deductions. This was what the pioneering AI research tried in the 1950s. Most notably the Logic Theory Machine of Alan Newell and Herbert Simon (Newell and Simon 1956). In practise can such a method only find very short proofs. The combinatorial explosion will quickly exhaust any computer resources. Different more efficient variants of representation and search methods has been introduced like the *resolution* method pioneered by, for instance, Robinson in the early 1960s, (Robinson 1965), (Bundy 1983). These methods were more adapted to machine reasoning then human reasoning and were more efficient when implemented. Still they needed to by governed by strategies and heuristics optimizing the order in which clauses were resolved etc. Resolution theorem provers help against the combinatorial explosion but they do not eliminate it. They can still only produce proofs of modest length.
The disappointment of some of the reasoning system lead to the conclusion that more human knowledge needs to be put into the reasoning process, or as Bledsoe put it (Bledsoe 1977):

> The word "knowledge" is a key to much of this modern theorem-proving. Somehow we want to use the knowledge accumulated by humans over the last few thousand years, to help direct the search for proofs

This knowledge is included as heuristics, weights and priorities in the theorem prover. If it is an *interactive theorem prover* it can have its heuristics modified by a human during execution. Regarding search algorithm most systems rely on a hill-climbing algorithm, back-tracking or a best-first heuristic (Winker and Wos 1978).
In our research we are investigating another approach. Instead of using explicitly added heuristics to guide the search we apply a more powerful and robust general search algorithm. The hypothesis is that the robustness of genetic search could free the reasoning system from some of the burdens of carrying specialized heuristics. The search could then be more autonomous and act more "intelligent" when it produces solution to problems with less a priori knowledge.

2 Genetic Programming

Genetic programming (Koza 1992) uses an evolutionary technique to *breed* programs. First a goal in the form of a goodness criteria is defined. This, so called, fitness function could for instance be the error in a symbolic regression function. The population – a set of solution candidates – is initialized with random contents, (random programs). Each "generation" the most fit individual programs are selected for reproduction. These highly fit individuals have offspring trough recombination (crossover) and mutation. Various methods exists for selection and reproduction but the idea is that the better individuals, and their offspring, gradually replace the worse performing individuals [1].
The individual solution candidate is represented as a tree (the genome). This tree can be seen as the parse tree of the program in a programming language. Recombination is normally performed by two parents which exchange subtrees ,see figure 1.

A typical application of GP is symbolic regression. Symbolic regression is the procedure of inducing a symbolic equation, function or program which fits given numerical data. Genetic programming is ideal for symbolic regression and most GP applications could be reformulated as a variant of symbolic regression. A GP system performing symbolic regression takes a number of numerical input/output relations, called fitness cases, and produces a function or program that is consistent with these fitness cases. Consider for example the following fitness cases:

```
f(2) = 6
f(4) = 20
f(5) = 30
f(7) = 56
```

[1]Genetic programming has some similarities with, "beam search" (Lowerre and Reddy 1980), (Rosenbloom 1987), if the population is regarded as the memory buffer and the fitness as a stochastically assigned priority.

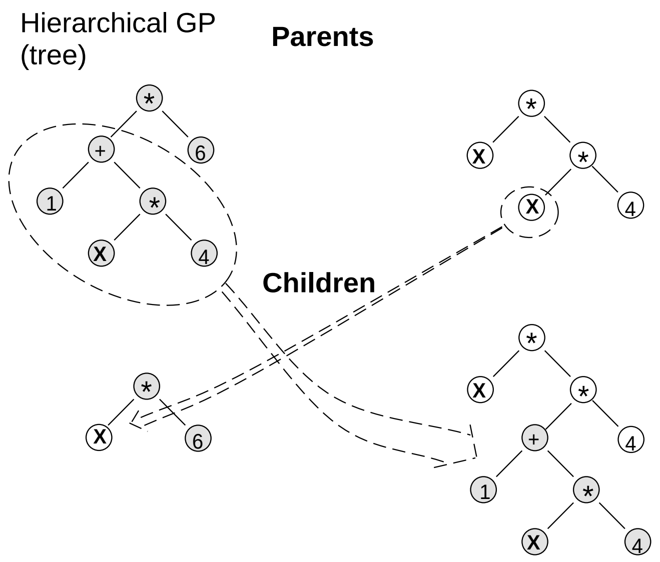

Figure 1: hierarchical crossover in Genetic programming.

One of the infinite number of perfect solutions would be $f(x) = x^2 + x$. The fitness function would, for instance, be the sum of difference between an individual's (function's) actual output and the output specified by the fitness cases. The *function set*, or the function primitives could in this case be the arithmetic primitives $+, -, \cdot, /$, as also seen in figure 1.
The two most important decisions to make before training a genetic programming system is to choose a good fitness function and to choose the right *function set*.
The fitness function should allow for a gradual improvement during evolution and it should, like other objective functions, give meaningful feedback to the GP induction system.
The function set should contain relevant primitives to the problem domain. Each function in the function set should also be syntactically closed – they should be able to gracefully accept all possible inputs in the problem domain.

3 Genetic Reasoning

In order to apply GP to reasoning and automated theorem proving (ATP) we need to design the appropriate fitness function, function set and choose a theorem representation. Our goal is to handle statements about arithmetics in a

logic as powerful as first order logic.
The function set is made up of function representing rules of inference. Such a function could for instance be the rule $X + O$ *can be replaced by* X. All function we use are unary-functions – they take one statement as input and produce another equivalent statement. This means that the tree representation of the individuals in GP collapses into a linear list representation[2] and that recombination will exchange linear segments of the individual genome, see figure 2.

The actual statement that should be proven true or false is represented by a

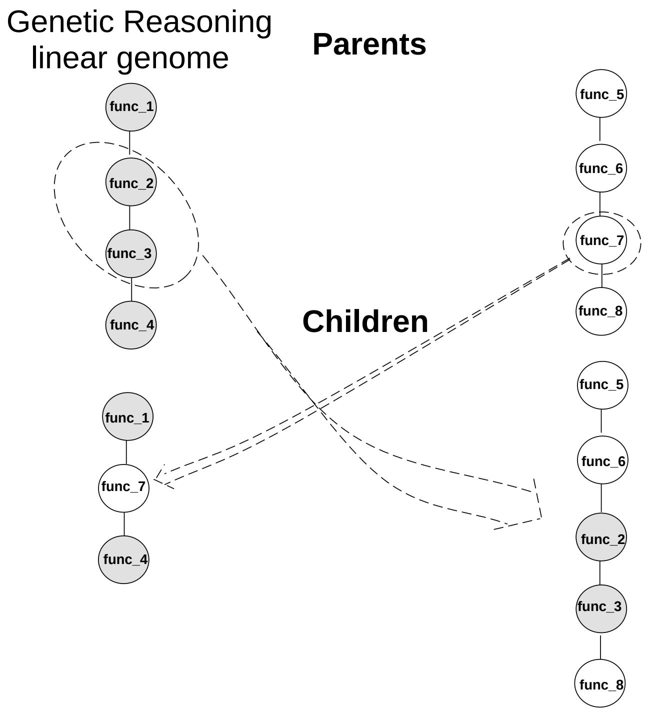

Figure 2: Crossover in Genetic Reasoning.

tree. Universal quantification is indicated by leaving variables free. Existential quantification is represented by a Skolem function, as common in several approaches to ATP. The natural numbers are built into our system in the form of the *zero (0)* symbol and the successor function. Figure 3 shows how the (false) statement $3 = 2 + 0$ would be represented.

The inference functions in the genome are then applied in turn to this structure.

[2]Note that in this application the genome structure is linear while the fitness case input is a tree structure. This is sometimes the other way around in other GP applications.

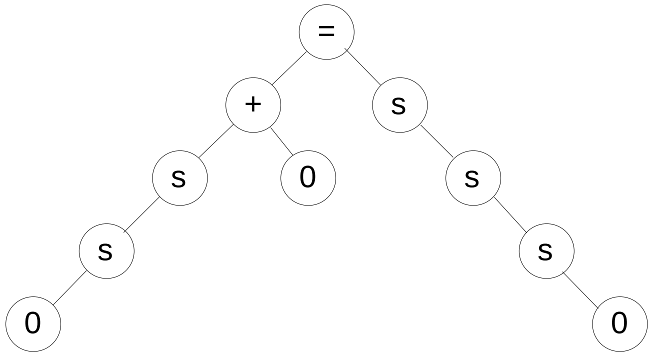

Figure 3: Representation of the statement $3 = 2 + 0$.

When all inference function have been applied – when the individual "program" has terminated – then we have another tree structure representing an equivalent statement. Let us say that we call the rule $X + O$ can be replaced by X, $func_1$. If this function is part of the genome it will try to match a subtree in the statement and, if it finds a match, replace it with X, see figure 4.
In figure 4 the function matches a sub-tree in the statement and the statement

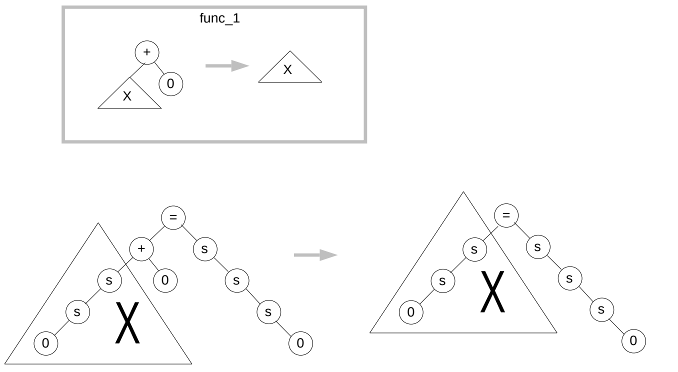

Figure 4: Application of $func_1$ to the statement $3 = 2 + 0$.

can be transformed. With this transformation the size of the statement structure is reduced, but there exists an equal number of functions in the function set that increase the size of the structure. If a function *does not* match any sub-tree in the statement then the structure is left untouched. This procedure provides syntactic closure and does also give the opportunity to temporary store unused material in the genome. The phenomena of unused genetic material is called introns in biology and may play an important role in the efficiency of a genetic search, see (Nordin, Francone and Banzhaf 1995b), (Nordin and Banzhaf 1995a).

3.1 The Fitness Function

The fitness function is very simple in our genetic reasoning system. It is just the number of nodes in the statement structure (figure 3). The two simplest and shortest statements are the Boolean constants **t** and **f**, each represented by only one node. These truth values are short hand for an axiom respectively a contradiction. The genetic system will thus try to simplify any expression down to the statement of either true or false, represented by the nodes **t** and **f**.
Genetic search has been proven to perform well and robustly in a wide variety of highly multi-modal search domains were local optima easily can trap a more hill climbing related approach. So, the pressure towards simplifying the statement does not mean that the system will try to constantly shorten the structure. The concept of a population of solution candidates helps the search to avoid local optima. The selection criteria from generation to generation does not monotonically select the best individuals but probabilistically reproduces individual with a large variation in fitness.

4 The Logic of Genetic Reasoning

The logic of the genetic reasoning system is similar to the one of the automated reasoning system Nqthm (Boyer and More 1979). It is a quantifier free first order logic with equality. The rules of inference are from propositional logic and the equality. Mathematical induction is an important part of the system. This principle is added explicitly as it cannot be expressed efficiently in first order logic.
Functions defining all boolean arithmetic operations are built-in ($\wedge$,$\vee$,$\neg$,$\rightarrow$).
The boolean constants **t** and **f** representing an axiom respectively a contradiction are also built-in. There are if-then-else functions as well as equality.
The natural numbers and arithmetics are defined by the peano axioms and the symmetry relation.
The deduction theorem is also predefined.
It is possible to add functions defining abstract data types and lemmas to support a specific application. In the register machine example below axioms describing this fictional processor are added.

4.1 The Evolutionary Algorithm and its Implementation

It is in principle possible to use any variant of GP as the basis for genetic reasoning. We have used a steady-state algorithm with tournament selection. The size of the population has been between 100 – 1000 individuals.
The GP system is implemented on a SUN-20 in PROLOG. The built-in features of PROLOG, such as pattern matching and list handling, simplifies implementation significantly

5 Results

Our reasoning system has so far been applied to two different domains: proving simple statement in arithmetics and reasoning about, for instance, halting of machine code programs. Both these applications relies to a great extent on mathematical induction as the proof method. All examples are such that the proof could not be obtained by simplification only. The system could not just hill-climb towards a solution, instead various steps of expansions needed on the way, to be able to finally reach a constant false or true statement. These expansions where not defined by a lemma or heuristic but were the result of the genetic search process.

5.1 Arithmetic Problems

The arithmetic problems that we started our evaluation with were selected using two criteria. The statement should be hard to prove without induction and it should be impossible to prove by just transformations to shorter statements. The induction principle might in it self require proofs that cannot be obtained by monotonic transformations and reductions. A typical statement used is:

There is no natural number bigger than three, that when added two to it, is equal to four. This statement is represented by the genome (tree structure) in figure 5.

This kind of statement can be proven (false) with a few hundred generation equivalents and a population size of 200 individuals. This calculation takes about 10 minutes on our SPARC-20.

5.2 Termination Proofs of a Program for a Register Machine

A register machine is a machine that operates with instructions that manipulates a limited set of registers. All CPUs in commercial computers are register machines. The instructions of a register machine might looks as: $a = b + 12$ which should be interpreted as: add 12 to the content of register b and place the result in register a. The processor also contains instructions for control of program flow, for instance jumps as well as conditional instructions. The axioms

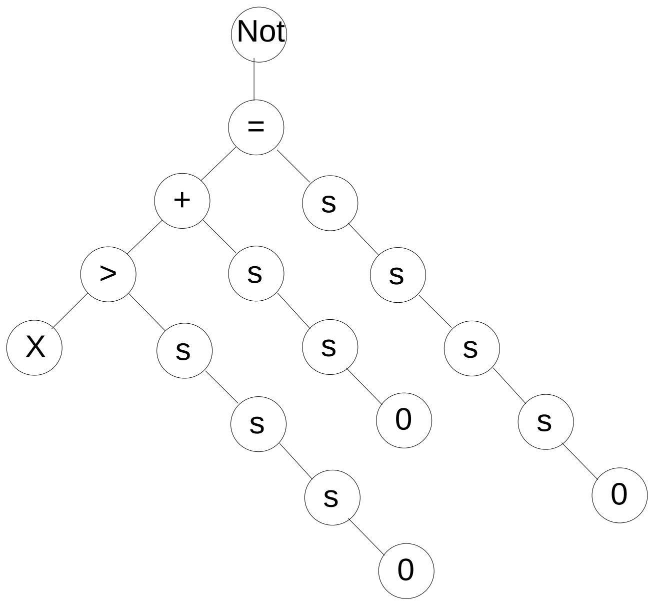

Figure 5: An example statement representation.

defining the processor and the current processor program is added to the system. We then use the genetic reasoning system to determine the correctness of the machine code programs, which often means the proof of termination. This approach demonstrates one of the strengths of the genetic reasoning system as termination proofs almost exclusively requires induction to be part of the system.

The machine code application is slightly more complex than the purely arithmetic statements and the verification of a short program, 2-10 instructions, takes about one hour with a population size of 1000 individuals.

The correctness proofs of programs has many applications, data security, high robustness in programs (i.e. satellite technology), simplification of machine code programs, (Boyer and Yu 1992). GP is often accused of producing non-robust solutions. A reasoning system could judge evolved solutions to prove if they are complete or not. Much like a human programmer that first almost by intuition might put a program solution together and when it works he (hopefully) studies it and reasons in his mind to see if it will really hold for all input and if it is safe, if it can be simplified etc. The termination proofs could also be used with a normal GP system to detect infinite loops in individual programs during evolution. Normally a few hundred generation equivalents has been needed to reach the true and false constants in our program verification experiments.

5.3 Multi-valued Logic

Other reasoning techniques, such as resolution, relies in a classical two valued logic. Most of these systems search for a contradiction to disprove a statement and to conclude its negation. Genetic reasoning on the other hand can be used with any logic. It is just a matter of defining the right transformation functions in the function set. We have implemented Kleene's three valued logic (Kleene 1950) in the system to better deal with paradoxes. In Kleene's tree valued logic a formula is either true false or a paradox (catch-22). With this logic our system can answer statement of paradoxical character. We have for instance included the definition of the genetic reasoning system as a primitive in the computer language we are reasoning about. This can give rise to true paradoxes that are hard to resolve in classic logic. We have also tried the use of a fourth truth value which is needed in the case when the genetic reasoning system does not find a proof of the statement. This fourth truth value represents a n "unknown" truth.

6 Future Work

In our experiments we have so far been concentrated on the problem of proving a theorem. We are, however, convinced that the genetic reasoning method has application is less rigid areas of reasoning and machine learning, such as planning in robotics. We plan to continue and extend our robot experiments on the Khepera robot platform with the application of genetic reasoning (Nordin and Banzhaf 1995c).
We also plan to port the system to C which will give an acceleration of as much as 100 times. This would allow us to try more difficult problems using large population sizes.

7 Summary and Conclusions

We have demonstrated that automated reasoning could be seen as an instance of automated programming. In this spirit we a have evaluated the use of a robust genetic search algorithm to search the spaces of proofs. The system has been able to avoid local minima in its search and found proofs of statements from complex domains such as arithmetics and program verification. The system uses no heuristic or human knowledge to guide its search, instead it relies on the performance of the search algorithm. We believe that this technique can have applications in many automated reasoning, and machine learning domains for instance robot planning.

Acknowledgement

Thanks to Steffo Weber for invaluable advise in the process of writing this paper. We would also like to thank Tom Haynes for giving us pointers to his experiments with GP and theorem proving (Knight and Haynes 1994). One of us (P.N.) acknowledges support by a grant from the Ministerium für Wissenschaft und Forschung des Landes Nordrhein-Westfalen under contract I-A-4-6037-I

References

[1] James W. (1890) The principles of psychology Vol.1. Originally published: Henry Holt, New York1890.

[2] Schwefel, H.-P. (1995) *Evolution and Optimum Seeking*, Wiley,New York

[3] Holland, J. (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.

[4] Fogel, L.J., Owens, A.J., Walsh, M.J.,(1966) Artificial Intelligence through Simulated Evolution. Wiley, New York

[5] Friedberg, R.M. (1958) A Learning Machine - Part I,*IBM Journal of Research and Development* IBM, USA 2(1), 2-11.

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

[7] Koza, J. (1992) *Genetic Programming*, MIT Press, Cambridge, MA

[8] Newell, A., Shaw J.C., and Simon H. (1957) Empirical Explorations of the Logic Theorem Machine: A case study in Heuristic, in *Proceedings of Western Joint Computer Conference* Vol. 15.

[9] Robinson J.A., (1965) A Machine Oriented Logic Based on the Resolution Principle, In *J.ACM*, Vol. 12, No. 1, pp. 23-41.

[10] Bundy A., (1983) The Computer Modeling of Mathematical Reasoning, Academic Press, London, pp. 74-77.

[11] Bledsoe W. W., (1977) Non-Resolution Theorem Proving, In *Artificial Intelligence*, Vol. 9, pp 2-3.

[12] Winker S., Wos L.,, (1978) Automated Generation of Models and Counterexamples and its application to Open Questions in Ternary Boolean Algebra, In Proceedings of 8th international symposium Multiple-Valued Logic, Rosemont, Ill., IEEE and ACM, pp. 251-256, New York

[13] Lowerre, B.T., and Reddy, R.D. (1980) The Harpy Speech Understanding System. In *Trends in Speech Recognition.* Lea, W.A. (Ed.) Englewood Cliffs, Prentice-Hall, New York.

[14] Rosenblom, P. (1987) Best First Search. In *Encyclopedia of Artificial Intelligence*, Shapiro, S. (Ed) Vol. 2, Wiley, New York.

[15] Nordin, J.P. , Banzhaf W.(1995c) Controlling an Autonomous Robot with Genetic Programming. In proceedings of: *1996 AAAI fall symposium on Genetic Programming*, Cambridge, USA.

[16] Nordin, J.P. ,F. Francone, Banzhaf W. (1995) Explicitly Defined Introns in Genetic Programming. In *Advances in Genetic Programming II*,(In press) Kim Kinnear, Peter Angeline (Eds.) , MIT Press USA.

[17] Nordin J.P. and Banzhaf W. (1995a) Complexity Compression and Evolution, in *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995*, L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA

[18] Boyer R.S., and Moore J.S. (1979) Proving Teorems about LISP-Functions, In *J.ACM*, Vol. 22, pp 129-144.

[19] Boyer R.S. and Yu Y. (1992) Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor, In *Automated Deduction - CADE-11* Kapur D. (Ed), pp. 416-430.

[20] Kleene, S.C. (1950) Introduction to Metamathematics, Van Nostrand, New York.

[21] Nordin, J.P. , Banzhaf W.(1995c) Controlling an Autonomous Robot with Genetic Programming. In proceedings of: *AAAI fall symposium of Genetic Programming*, Cambridge, USA.

[22] Knight L., Haynes T. (1994) A GP Theorem Prover, technical report CS 7213, University of Tulsa