An Analysis of the Impact of Functional Programming Techniques on Genetic Programming

Gwoing Tina Yu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

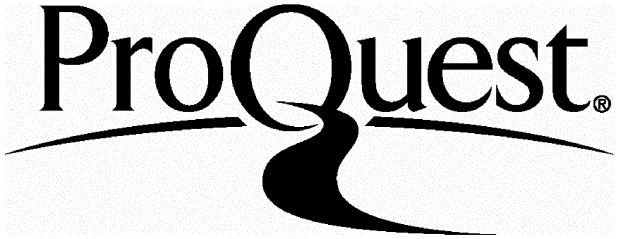
of the

University of London.

Department of Computer Science

University College London

1999



Abstract

Genetic Programming (GP) automatically generates computer programs to solve specified problems. It develops programs through the process of a "create-test-modify" cycle which is similar to the way a human writes programs. There are various functional programming techniques that human programmers can use to accelerate the program development process. This research investigated the applicability of some of the functional techniques to GP and analyzed their impact on GP performance.

Among many important functional techniques, three were chosen to be included in this research, due to their relevance to GP. They are *polymorphism*, *implicit recursion* and *higher-order functions*. To demonstrate their applicability, a GP system was developed with those techniques incorporated. Furthermore, a number of experiments were conducted using the system. The results were then compared to those generated by other GP systems which do not support these functional features. Finally, the program search space of the general `even-parity` problem was analyzed to explain how these techniques impact GP performance.

The experimental results showed that the investigated functional techniques have made GP more powerful in the following ways: 1) polymorphism has enabled GP to solve problems that are very difficult for standard GP to solve, i.e. `nth` and `map` programs; 2) higher-order functions and implicit recursion have enhanced GP's ability in solving the general `even-parity` problem to a greater degree than with any other known methods. Moreover, the analysis showed that these techniques directed GP to generate program solutions in a way that has never been previously reported. Finally, we provide the guidelines for the application of these techniques to other problems.

Acknowledgements

I would like to thank my supervisors, Chris Clack and Robin Hirsch, for their guidance and support. The department research student tutor, Mel Slater, has helped me out on many occasions. I would like to thank him for his generous support.

Bill Langdon has been my mentor during this work. Bill introduced me to the field of Genetic Programming at the beginning of my Ph.D. study. Since then, he has continuously helped me with my work. This thesis has improved a great deal by his detailed comments and suggestions. I would like to thank him for his time and his patience with me.

David Fogel, Peter Angeline and John Koza have provided timely assistance and encouragement on many occasions. They have made me feel what I am doing is important. I would like to thank them for being so generous to a new comer in the field.

Evolutionary computation is a friendly field. Since the beginning of this research, I have been lucky to receive support from many people. EP97 accepted my first paper, which was a great encouragement to me. Tom Westerdale has always made himself available when I needed inputs to my work. At GP97, Nic McPhee showed keen interest in my work. I was pleasantly surprised to see that he had developed my work at GP98. During my investigation of the GP schema theorem, Riccardo Poli has patiently answered my questions either through e-mail or in person. Peter Whigham, although he had never met me, has answered my questions many times through e-mail. I would like to thank them for being such good compatriots.

Ann and Paul Peterken have been my dearest English friends. They provided me a home in England during the three years of my study. I would like to thank them for their kindness.

Finally, I would like to thank my parents for letting me go and allowing me to grow into my own person. I also thank my brother for taking care of my parents during these years when I have been away from home. My hearty thanks go to my sister, who encouraged me to go to England at the time she needed me to stay with her in States the most. She has shown me the courage to deal with difficult situations. This thesis would not be possible without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Human computer programming involves a series of problem-solving activities. Firstly, the problem is analyzed and the parameters within the problem are defined. Secondly, a method to solve the problem is formulated. Finally, the solution is implemented and executed to solve the problem. These activities may be iterated during the programming process in order to solve the given problem.

Many modern programming techniques focus on the support of these problem solving activities. Two popular examples are *problem decomposition* and *contextual checking*. Problem decomposition is a method known as "divide and conquer"; it involves the subdivision of a problem into smaller problems and the use of the solutions to the smaller problems to construct the overall solution. Contextual checking is a procedure to ensure the internal consistency of the program; this checking is normally carried out by an independent agent. For example, lexical/syntactic analyzers check that programs conform to the programming language grammar while a type checker ensures that the types of data and functions are consistent with those specified by the programmers. The former technique provides guidelines to solve the problem while the later provides early warning of program errors. Both of them support a more effective program development process.

Modern functional programming languages provide a number of unique techniques to support program development. Three of these techniques are *polymorphism*, *higher-order functions* and *implicit recursion* - their details are described in Chapter 2. These techniques are unique in that either they are implemented in an unique way or they exist only in functional languages. Polymorphism in functional languages is implemented using a formal type system [Hindley 1969; Milner 1978] which is different from the generic function template approach used in imperative languages [Barnes, 1994; Stroustrup, 1991]. The implementation of higher-order functions in functional languages is based on the concept that functions are like data, hence are allowed to be passed as arguments to other functions. In contrast, impera-

tive languages use function pointers to implement higher-order functions [Kernighan and Ritchie, 1988]. Implicit recursion is uniquely supported in functional languages through higher-order functions such as `foldr`, `map` and `filter` (see Section 2.3.7). It is not supported in other types of programming languages. These three are powerful techniques, and have made program development an easier task [Hudak, 1989].

1.1 Genetic Programming as a Functional Programmer

The Genetic Programming (GP) [Koza, 1989, 1990, 1992, 1994a] paradigm is a problem-solving method based on a computational analogy to natural evolution. In this method, computer programs are automatically generated to solve the given problems. Initially, a population of programs is randomly created using functions and terminals appropriate to the problem domain. Each individual program is then measured in terms of how well it solves the problem. This measure becomes the fitness of the program. Next, the Darwinian principle of survival of the fittest is used to select programs for reproduction. The standard selection method is *fitness-proportionate selection*, where the probability of an individual to be selected is equal to its normalized fitness value [Koza, 1992, page 97]. The programs which are selected from the current population are manipulated by genetic operations of *crossover* and *mutation* to generate new offspring programs. The crossover operator creates offspring using two parental programs while the mutation operator generates new offspring by altering one individual program. The generated new offspring programs constitute the new population. Each individual in the new population is measured for fitness and the process is repeated for many generations. Typically, the best program that appears in the last generation is designated as the result produced by GP.

Consequently, there are three main phases in the GP problem solving method:

1. constructing the initial population of computer programs;
2. evaluating each program in the population and assigning it a fitness value according to how well it solves the problem;
3. selecting "fit" programs from the current population to create new programs using genetic crossover and mutation. The new programs form a new population.

Phases 2 and 3 are iterated until the specified termination criterion is met. The best program generated at the end of the process is the solution (or an approximate solution) to the problem.

The GP program development process can be described as a "create-test-modify" cycle (Figure 1.1) which is similar to the way humans develop their programs. Initially, programs

are created based on the knowledge given about the problem (given in functions and terminals). These programs are then tested on the problem. If the results are not satisfactory, modifications are made to improve the programs. This create-test-modify process is repeated until either a satisfactory solution is found or a specified condition is met.

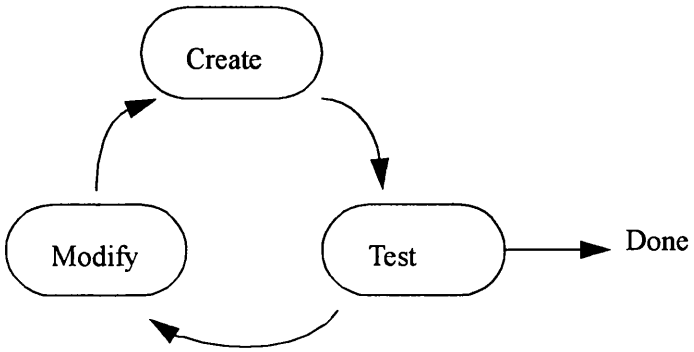


Figure 1.1: GP programs development process.

Depending on the problems that GP is to solve, various issues can arise. For example the problem solution may not be able to be represented in a way which satisfies the "closure" property [Koza, 1992, page 81]. Another example is that a problem might be much more easily solved using recursion, yet GP has not been very successful in evolving recursive programs [Brave, 1996; Wang and Leung, 1996]. These issues limit the applicability of GP. This thesis addresses these issues by introducing polymorphism, higher-order functions and implicit recursion to the GP paradigm. Extended with these techniques, GP is like a functional programmer who uses functional techniques to develop programs to solve the given problem.

Since polymorphism, higher-order functions and implicit recursion have benefited human programmers, we hypothesize that these functional programming techniques can also enhance GP's ability in solving suitable problems. This hypothesis will be tested on the general `even-parity` problem and the `map` and the `nth` programs.

1.2 Objectives

This work has three objectives. Firstly, to show that polymorphism, higher-order functions and implicit recursion are applicable to the GP paradigm. This has been achieved by implementing a GP system which incorporated these techniques and by generating programs using the system. Secondly, to demonstrate that these techniques can be beneficial to GP in solving suitable problems. This has been achieved by using the developed GP system to solve problems that are reported to be very difficult for the standard GP to solve, i.e. the general `even-parity` problem and the `map` and the `nth` programs. As will be shown, these techniques

have enabled GP to solve these problems very efficiently. Finally, to establish guidelines for the application of these techniques to other problems. This is partially achieved by analyzing program structures in the search space to identify how these techniques assist GP to find problem solutions. The guidelines are formulated accordingly.

1.3 Contributions

This research makes the following contributions:

1. It constructs a formal GP framework to evolve λ-calculus expressions.
   - A single language with sufficient computation power to solve a wide variety of problems is provided in GP. The language also provides a natural integration of a module mechanism via λ abstractions (see Chapter 4).
2. It demonstrates advantages provided by applying the following functional programming techniques to GP:
   - **polymorphism**: presents the concept of types in GP in great detail through the definition of and the differentiation between *untyped*, *dynamically typed* and *strongly typed* GP. The Strongly Typed Genetic Programming (STGP) [Montana, 1995] is formalized and extended to include various kinds of type variables and higher-order function types. Moreover, the impact of different type variables on GP search space is analyzed (see Chapter 5).
   - **implicit recursion**: provides recursion semantics in the evolved programs without explicit recursive calls. Previously, evolving recursive programs in GP has been *difficult*. This work not only identifies the issues that cause such a difficulty but also provides a solution, implicit recursion, to overcome the difficulty (see Chapter 6).
   - **higher-order functions**: supports an effective module mechanism for GP. In this approach, module creation is neither a random process nor determined in advance. Instead, it uses the knowledge (function type arguments) specified by the users in the higher-order functions to determine the most beneficial way to create modules. Most importantly, this work introduces a new term, *structure abstraction*, to describe the structure pattern emerging from the higher-order functions program representation. Structure abstraction not only enables GP to evolve a general solution to the `even-parity` problem but also achieves greater efficiency than any other previous work (see Chapter 6).
3. It identifies structure abstraction as a hierarchical processing engine for GP search. The

guidelines for the application of structure abstraction to other problems are outlined (see Chapter 7).

4. It presents a concept of constraint handling based on the general framework of evolutionary algorithms (see Appendix A). This general approach provides an easy way to compare and contrast different constraint handling methods, e.g. dynamic typing versus strong typing (see Chapter 5). Moreover, the seesaw effect demonstrated in the experiments gives a high level view of the impact of constraint handling on the evolutionary process, e.g. see the constraint handling for recursion error in Chapter 5.

1.4 Organization

Following this introductory chapter, Chapter 2 presents background in Genetic Algorithms, Genetic Programming and Functional Programming Languages. They are the foundation on which this work is based and from which it develops.

Chapter 3 summarizes related work. It is categorized into four areas: syntactic constraints using grammars, type constraints, modules and recursion. They are related to our work in evolving λ calculus and in investigating the applicability and benefits of the following functional techniques to GP: polymorphism, higher-order functions and implicit recursion.

Chapter 4 describes the GP system which was developed with the mentioned functional techniques incorporated. Each component of the system is presented, with the implementation details. The genetic algorithm used in the system and the programming language chosen to implement the system are explained. This chapter concludes with an example demonstrating the operation of the system.

Chapter 5 presents the application of polymorphism in GP. The concept of types in GP is introduced through the definitions of and differentiation between *untyped*, *dynamically typed* and *strongly typed* GP. The limitation of untyped GP in problem solving and the issues occurring in dynamically typed GP are then summarized. This is followed by a list of advantages that strongly typed GP can provide. The two different approaches of strongly typed GP, *monomorphic* and *polymorphic* GP, are then compared. This work advocates the use of polymorphic GP, which is implemented by using three different kinds of type variables. We analyze the impact of these type variables on GP search space. Finally, two polymorphic programs are evolved to demonstrate that polymorphism has enhanced GP applicability to problems which are very difficult for GP without polymorphism to solve.

In Chapter 6, higher-order functions are introduced to provide a better program representation for module creation and reuse. In this approach, a module is represented as a λ *abstrac-*

tion and its reuse is through *implicit recursion*. We first analyze GP issues in evolving recursive programs. Implicit recursion is then introduced as a solution to overcome these issues. We explain the λ abstraction module mechanism and compare it with other module approaches. This program representation is then used to evolve general solutions to the `even-parity` problem [Koza, 1992]. The experimental results show that this approach has enabled GP to find a solution with great efficiency. This chapter proposes that a program structure pattern, named "*structure abstraction*", is the cause of the superior performance.

Chapter 7 investigates the impact of structure abstraction on GP search. A formal definition of structure abstraction is first presented. This is followed by a detailed description of the application of structure abstraction to the general `even-parity` problem. The program structures and the solution structures in the search space are then analyzed. The results indicate that structure abstraction serves as an engine of hierarchical processing for GP search, hence allows the solution to be found very efficiently. Finally, the guidelines for the application of structure abstraction to other problems are provided.

Chapter 8 outlines our future work. Firstly, we will investigate the reason why type constrained GP is more efficient than standard GP on problems involving multiple types. Secondly, other forms of implicit recursion will be explored. Finally, we will apply structure abstraction to more problems.

The last chapter, Chapter 9, presents the summary and conclusion of this thesis. This is followed by the bibliography.

In Appendix A, a concept of constraint handling based on the general framework of evolutionary algorithms is presented. This approach provides an easy contrast and comparison between different constraint handling methods used in an evolutionary algorithm. Moreover, our experimental results demonstrate the importance of constraint handling in evolutionary algorithms, for if the search space is not constrained properly, the evolution of good solutions may be prevented.

Appendix B describes the method used to measure the performance of the GP system in various experiments. It is essentially a brief summary of Chapter 8 in [Koza, 1992].

Appendix C reports our experiments and their results on using a higher-order function program representation to solve the artificial ant problem [Koza, 1992].

Finally, Appendix D provides an analysis of program evolution for `even-parity`, `nth` and `map` programs. It investigates the impact that each component (fitness function, run-time error constraint handling, search algorithm) has on GP in generating these three programs by conducting a series of experiments. The results are analyzed and discussed.

Chapter 2

Background

The first part of this chapter provides a brief introduction to Genetic Algorithms (GAs) [Holland, 1992] from which GP is derived. In particular, we present the schema theorem [Holland, 1992] and the building block hypothesis [Goldberg, 1989], which were developed to explain how GAs search for problem solutions. Meanwhile, related criticisms are highlighted. This is followed by a summary of the distinctive features of GP and research conducted toward a GP schema theorem. The second half of this chapter presents background knowledge in functional programming languages, particularly in the area of λ calculus, polymorphism, higher-order functions and recursion. They are the functional techniques that this work applies to GP. This chapter provides background knowledge about how GP searches for problem solutions, thus enabling the analysis of the impact of functional techniques on GP performance.

2.1 Genetic Algorithms

GAs are search algorithms based on the mechanics of natural selection and natural genetics. Genetic material is packed in a fixed-length string to represent an individual. A population consists of many individuals. The natural selection scheme, survival of the fittest, combined with two reproduction mechanisms (the traditional GA uses one-point crossover and point mutation while other GAs use different operators) are used to evolve better individuals. The GA evolutionary process uses the following 3 artificial operators to mimic natural evolution:

- Fitness-proportionate selection: Darwinian survival of the fittest;
- One-point crossover: a sexual reproduction operator which generates offspring by interchanging sub-strings from two individuals;
- Point mutation: an asexual reproduction operator which generates offspring by modifying a single point in one individual.

These operators seem simple, yet they provide a powerful search algorithm which explores new search points with improved performance by exploiting historical information.

The schema theorem and the building block hypothesis were developed by [Holland, 1973; Holland, 1992] and [Goldberg, 1989] respectively to provide a theoretical framework for GAs. The schema theorem uses a mathematical formulation to explain why GAs are capable of searching for problem solutions. This theorem is further expanded with the building block hypothesis to explain how partial solutions are carried down generation by generation to build overall solutions. However, these two works have been widely criticized recently. The details of these two theoretical works and the related criticisms are presented in the following sections.

2.1.1 Schema Theorem

In [Holland, 1992], a schema is defined as a template describing a set of points, from the search space of a problem, that have certain specified similarities. Each schema is a string over an extended alphabet consisting of the original alphabet (0 and 1) and a hash symbol (the "don't care" symbol). For example, the schema "0##1" describes 4 similar points: "0001", "0011", "0101" and "0111". In the conventional GA, the number of individuals contained in the population is usually infinitesimal in comparison to the search space of the problem. However, each individual in the population represents 2^L schemata, where L is the length of the string. Consequently, the fitness-proportionate selection for reproduction, which explicitly operates only on the individuals present in the population, actually performs on a much larger number of schemata implicitly. However, this claim has been criticized by [Macready and Wolpert, 1996].

With the fitness-proportionate selection, the expected number of occurrences of every schema in the next generation can be estimated. Suppose that at a given time t, there are m individuals representing a particular schema H in the population. This is represented with the notation $m(H,t)$. At time $t+1$, the expected number of individuals representing the schema H in the population, represented as $E[m(H, t+1)]$, is given as the following:

$$E[m(H,t+1)] = \frac{f(H,t)}{\overline{f(t)}} m(H,t) \qquad (1)$$

where $f(H,t)$ is the average fitness of the individuals representing schema H at time t and $\overline{f(t)}$ is the average fitness of the population. Equation 1 says that the number of individuals representing a particular schema grows or shrinks at the ratio of the average fitness of the individu-

als representing the schema to the average fitness of the population. Put in another way, schemata with fitness values above the population average are expected to receive an increasing number of individuals in the next generation, while schemata with fitness values below the population average are expected to receive a decreasing number of individuals in the next generation.

Assuming that the fitness of a particular schema H remains above the average $\overline{f(t)}$ an amount $c \cdot \overline{f(t)}$ where c is a constant, the equation can be rewritten:

$$E[m(H,t+1)] = \frac{\overline{f(t)} + c\overline{f(t)}}{\overline{f(t)}} m(H,t) = (1+c) \cdot m(H,t) \qquad (2)$$

Only when the population is infinite and c is a stationary value, this equation represents a geometric progression. This means that a schema with above-average fitness will appear in the next generation at an approximately exponential increasing rate over those generations. Holland [1992] argued that this exponential increasing rate is the *optimal* way of schemata processing through his analysis of the two-armed bandit problem described in the following subsection.

The Two-Armed Bandit Problem

In the two-armed bandit problem, there is a slot machine which has two arms. Furthermore, one of the arms pays reward μ_1 with variance σ_1^2 and the other arm pays reward μ_2 with variance σ_2^2 where $\mu_1 > \mu_2$. The objective is to get the most reward by playing the arm with higher reward more frequently (the arm with a pay-off μ_1). But how do we know which arm to play in each trial, since we have no knowledge about which arm is associated with the higher reward? Ideally, we would like to make a decision which can provide not only a good reward but also information about which is the better arm to play for the next trial. There is a trade-off between these two wishes of the exploration for knowledge and the exploitation of that knowledge. Such a dilemma is a fundamental theme in adaptive systems. The two-armed bandit problem is therefore a good candidate to study the optimal trials in any adaptive system such as GAs.

In an experiment, presume that a series of trials have been conducted. The goal is to use the acquired knowledge to decide which is the better arm to play in the rest of the trials. This experiment has an expected loss, L, playing the wrong arm, as the following equation:

$$L(N,n) = |\mu_1 - \mu_2|[(N-n)q(n) + n(1-q(n))] \qquad (3)$$

N = total number of trials in the experiments.

n = number of trials that have been conducted on each of the two arms with a total of $2n$ trials.

$q(n)$= probability that the wrong decision is made after the $2n$ trials.

Equation *3* identifies two sources of loss:

- The first loss is a result of choosing the wrong arm, the arm associated with the lower payoff, after performing the $2n$ trials. This means that we choose n wrong arms during the $2n$ trials and also during the rest of the $(N-2n)$ trials. There are a total of $(n+(N-2n)) = (N-n)$ such trials.
- The second loss occurs when we select the correct arm, the arm associated with the better payoff, after the $2n$ trials. This means that we have issued n trials choosing the wrong arm during the $2n$ trials with a probability of $(1-q(n))$.

The objective is to allocate the N trials between the two arms so that the expected loss, $L(N,n)$, can be minimized. Holland [1973, 1992] has calculated that to allocate trials optimally (in the sense of *minimal expected loss*), an exponentially increasing number of trials should be given to the observed better arm, i.e. the arm which receives the reward μ_1 in the current trial.

In order to apply the result of the two-armed bandit analysis to GAs, where multiple schemata are competing simultaneously, Holland expanded his analysis to the k-armed bandit problem. Holland [1992] demonstrated that the optimal solution to allocate the trials of k competing arms is similar to the solution of the two-armed problem and argued that an exponentially increasing number of trials should be given to the observed best of the k arms.

This result of optimal allocation of trials to the k-armed bandit can be applied to GAs by viewing the process of GA search as a competition of k schemata. Two schemata A and B with individual positions a_i and b_i are competing with each other if for positions i = 1,2..., l, at least one of the i value has feature such that $a_i \neq \#$, $b_i \neq \#$ and $a_i \neq b_i$. For example, the following four schemata are competing at locations 2 and 3:

```
# 0 0 #
# 0 1 #
# 1 0 #
# 1 1 #
```

The four schemata are competing because they are defined over the same positions (2 and 3). They are competing with one another for precious population slots. In order to allocate the population slots optimally, exponentially increasing numbers of slots should be allocated to

the observed best schema, just as we give exponentially increasing trials to the observed best arm in the *k*-armed bandit analysis.

This analysis of the optimal allocation of trials has been criticized by [Fogel, 1995, page 134]. First, it assumes the independent sampling of schemata. In fact, most GAs employ coding strings such that schemata are not sampled independently [Davis, 1985]. Second, sampling schemata in a way to minimize expected losses does not guarantee the discovery of the global optimal solutions. Consider the two arms of a bandit, each having mean payoffs μ_1 and μ_2 and each having variances of σ_1^2 and σ_2^2, assuming also that $\mu_1 > \mu_2$ and $\sigma_1^2 << \sigma_2^2$. To minimize the expected loss (equation 3), all trials should be devoted to the first distribution because it has the larger mean. When the optimal solution is in the second distribution, it can never be discovered. For example, a schema A which represents 4 individuals each having fitness 10 would have an average fitness of 10. Another schema B which represents 4 individuals with fitness 0, 0, 0 and 20 respectively has an average fitness of 5. The minimizing expected losses approach would choose to sample schema A. However, the optimal solution (with fitness 20) is described in schema B, which will not be found by sampling with the principle of minimizing expected losses. "Identifying a schema with above-average performance does not, in general, provide information about which particular complete solution, which may be described by very many schemata, has the greatest fitness. The single solution with the greatest fitness (i.e. the globally optimal solution) may be described in schemata with below-average performance. The criterion of minimizing expected losses is quite conservative and may prevent successful optimization." [Fogel, 1995].

With the assumption that the exponential increasing of schemata with above-average fitness from one generation to another provides the optimal way to explore the search space in mind, we are now back to the schema theorem. Notice that the schema theorem states that the optimal schemata processing is achieved through the straightforward operation of fitness proportionate reproduction. Unfortunately, this selection operator doesn't introduce any new individual to the population. If the population size is the same as the search space, this won't be a problem (and this GA will find a solution in generation 0 through random search). To explore new and better individuals, variation operators such as crossover and mutation are needed. These variation operators disrupt schemata during their process and will impact the schema growth from generation to generation.

The probability of disrupting a schema *H* due to the one-point crossover is dependent on the *defining length* of the schema, represented in notation $\delta(H)$. The defining length of a schema is the distance between the outermost non-# symbols positions. For example: "1###" has defining length 0 while "#0#0" has defining length 2. A schema is disrupted when the

crossover is performed within the defining length among the total of l-1 possible crossover locations (l is the length of the schema). Assume p_c is the probability that the crossover is performed, and the location of crossover point is selected randomly, the probability that a schema survives from the disruption of crossover, represented as $P_s(c)$, is given in equation 4.

$$p_s(c) \geq 1 - p_c \cdot \frac{\delta(H)}{l-1} \tag{4}$$

The inequity in equation 4 is due to the fact that crossover within the defining length of a schema does not always disrupt the schema. For example, schema H is "11#####". When crossover takes place between the first and the second positions of two strings "1110101" (which schema H represents) and "0100000", the generated new string is "1100000". The result of the crossover does not disrupt schema H, although the crossover takes place within the defining length of the schema.

The next operator to consider is point mutation. The probability of disrupting a schema H due to mutation depends on the *order* of the schema, represented in notation *o(H)*. The order of a schema is the number of non-# symbols it contains. For example: "1#01" has order 3 while "#0#0" has order 2. Order is the number of places where mutation can effect a schema. Assume p_m is the probability that the mutation operation is performed and the mutation location is randomly selected within an individual, the probability that a schema survives from the disruption of mutation is the probability that all non-# symbol in the individual survive. This can be described in the following equation:

$$p_s(m) = (1 - p_m)^{o(H)} \tag{5}$$

Equation 6 gives the lower bound for the expected number of individuals representing schema H in the next generation under fitness-proportionate selection, one-point crossover and point mutation. It is the combination of equation 1, 4 and 5.

$$E[m(H,t+1)] \geq m(H,t) \cdot \frac{f(H)}{\overline{f(t)}} \cdot (1 - p_m)^{o(H)} \cdot \left[1 - p_c \frac{\delta(H)}{l-1}\right] \tag{6}$$

Equation 6 shows that the number of individuals representing a schema H grows or decays depending upon multiple factors: whether the schema fitness is above or below the population average fitness, whether the schema has short or long defining length and whether the schema has large or small order. Under untenable assumptions, (e.g. all schemata with above-average fitness increase their fitness values in a stationary rate c from generation to generation), the

schemata with above average fitness, short defining length and small order will appear in the next generation at an exponentially increased rate, which in turn provides the best way of schema processing, according to the analysis of the two-armed bandit problem.

2.1.2 Building Block Hypothesis

The schema theorem perceives the operation of GAs through the manipulation of schemata. The schemata with short defining length, low order and high-fitness are sampled most favorably because they have smaller disruption rates during crossover and mutation. In a way, GAs work by sampling these particular kind of schemata (the building blocks), recombining, and reassembling them to form individuals of potentially higher fitness. According to [Goldberg, 1989], the power of GAs is due to the ability to find good building blocks and to propagate them from generation to generation at a rate close to the optimal rate. This is called the building block hypothesis.

Dissenting arguments against the schema theorem, two-armed bandit analysis and the building block hypothesis have been expressed a number of times in the past and are addressed more profoundly in recent years. Since these topics are beyond the scope of this thesis, only related works are listed below for interested readers. Salomon [1998] also provides a comprehensive review of related issues.

- Both [Grefenstette and Baker, 1989] and [Muhlenbein, 1991] have criticized the schema theorem for not reflecting the real operation of GAs.
- Altenberg [1995] also dismissed the merit of the schema theorem in explaining GA performance.
- Fogel and Ghozeil [1998] pointed out that the schema theorem overlooked the misallocation of trials in the process of stochastic effects.
- The main theoretical proof of the incorrectness of the two-armed bandit analysis was first offered in [Macready and Wolpert, 1996; Macready and Wolpert, 1998].
- Later, Rudolph [1997] presented a counter example to demonstrate that the fitness proportionate selection method is not optimal in uncertain environments.
- Beyer [1995, 1997] showed that the power of crossover does not stem from the "combination" of "good properties" of the mates (as in the building block hypothesis) but rather from genetic repair diminishing the influence of harmful mutations.

2.2 Genetic Programming

Genetic Programming (GP) [Koza, 1992] is an extension of GAs. Similar to GAs, GP uses a central algorithm loop which applies the basic evolutionary-based genetic operators within a population of individuals to search for solutions. The most common way used to differentiate GP from GAs is that GP uses a dynamic tree representation and the representation is interpreted as a program. In addition, there are a number of emergent properties in GP that have distanced GP from GAs even further.

2.2.1 Genetic Programming Versus Genetic Algorithms

The following are emergent properties of GP that have been identified by [Angeline, 1994; Altenberg, 1994a; Altenberg, 1994b; Banzhaf, Francone and Nordin, 1997].

Dynamic Tree Representation

With the variable length program tree representation, GP evolves program *contents* and *structures* at the same time. The search space of GP is therefore more complicated than that of the traditional GAs, where only the contents of an individual are evolved. However, this property, strictly speaking, is not unique to GP. The tree representation is normally constrained to a maximum depth or a maximum number of tree nodes due to the virtual size of a computer's memory. In practice, the representation of the "dynamic" tree is implemented using part of a fixed-length bit string necessary to represent the tree. As the tree grows, more of the bit string is used. This approach has been used by others to implement their GA work [Shaffer, 1987; Goldberg, Deb and Korb, 1990; Jefferson *et al.* 1992].

Complex Representation Interpretation

In GAs, the interpretation function uses a positional encoding where the semantics of a bit are tied to its position as well as its content. As a result, the interpretation of a bit string is often a simple combination of the various positions, similar to a union of all independent features. In contrast, GP interprets a particular program tree without considering its position. For example, the interpretation of the function "if-then-else" in a genetic program is the same regardless of its position in the program tree. The arbitrarily complex association between an expression and its interpreted behavior allows for an big variety of dynamics to emerge naturally from the evolutionary process.

Syntax Preserving Crossover

The one-point crossover used in GAs is replaced with a subtree crossover in GP. This crossover operator preserves the syntax of the programs by swapping only complete subtrees between two parent programs. Many concerns have been raised regarding to its impact on the GP evolutionary process. Supposedly, this operator should provide GP the search power to find a good problem solution. However, [O'Reilly and Oppacher, 1995] has reported the opposite results: this crossover operator seems to destroy rather than facilitate the construction of problem solutions. To provide better performance, many alternative genetic operators have been proposed [O'Reilly and Oppacher, 1996; Angeline, 1997a; Chellapilla, 1997b; Harries and Smith, 1997; Poli and Langdon, 1998b].

Emergence of Introns

A tendency of GP program evolution is the growth of program length, commonly known as "bloat". Langdon and Poli [1997b] have argued convincingly that "such growth is inherent in using a fixed evaluation function with a discrete but variable length representation". Briefly, the fixed evaluation functions quickly drive search to converge, in the sense of concentrating the search on solutions with the same fitness as previously found solutions. In general, variable length allows many more long representations of a given solution than short ones of the same solution. Consequently, the longer representations occur more often and representation length tends to increase.

Angeline [1998] also pointed out an obvious, yet ignored, reason of bloat: subtree crossover promotes the recombining of larger program structures. When a set of 6 mutation operators is used to run three different problem experiments, Angeline showed that the program size does not grow as much as that using subtree crossover. This result is expected since 5 of these 6 mutation operators either modify a single program point or a program link, i.e. resulting no change of program size. The only exception is a subtree mutation which can potentially generate larger program trees. Consequently, program evolution using these 6 mutation operators does not cause the increase of program size as much as that using subtree crossover.

Program growth is also associated with the appearance of redundant code, called introns, in the evolved genetic programs. Although this redundant code has no effect on the semantics of the programs, its impacts on the GP evolutionary process have been reported in various research: according to [Nordin, Francone and Banzhaf, 1995; Haynes, 1996; Wineberg and Oppacher, 1996], introns are advantageous to GP as they protect fit building blocks from being destroyed by the crossover operator. In contrast, [Andre and Teller, 1997] reported that introns have a negative effects on GP. An important distinguishing characteristic is that,

unlike GAs where introns have to be designed into its representation [Levenick, 1991], introns in GP are emerged to its representation naturally through the dynamics of the representation.

2.2.2 Genetic Programming Schema Theorem

There are some theoretical works on defining the concept of *schema* for program trees and using them to define a *schema theorem* for GP [Koza, 1992; O'Reilly and Oppacher, 1995; Whigham, 1996b; Poli and Langdon, 1997; Rosca, 1997a]. Poli and Langdon [1998a] provided a good review of schema definition by different researchers. Basically, there are two main approaches in schema definition: Non-rooted tree and Rooted tree. The following is based on their work with more detailed explanations.

Non-Rooted Tree Approach

In this approach, a schema is a non-rooted program tree. The same schema are allowed to be present multiple times within a program parse tree. Since GP program trees can have many different sizes and shapes, multiple occurrences of the same schema can make the computation of the probability of schema disruption difficult. The formulation of schema theorems for GP using this approach is therefore inconclusive.

Koza [1992, page 117-118] made the first attempt to provide a schema definition for GP program trees. A schema H is represented as a set of S-expressions. For example the schema $H = \{(+\ 1\ x),\ (*\ x\ y)\}$ represents all programs including at least one occurrence of the expression *(+ 1 x)* and at least one occurrence of *(* x y)*. Koza's definition gives only the defining components of a schema not their position, so the same schema can be instantiated in different ways, and therefore multiple times, in the same program tree. Koza didn't provide a schema theorem for GP.

O'Reilly and Oppacher [1995] refined Koza's work and gave a schema theorem for GP based on fitness-proportionate selection and subtree crossover. Mutation is not considered in their work. A schema is an unordered collection (a miltiset) of subtrees and tree fragments. Tree fragments are trees with at least one leaf that is a "don't care" symbol ("#") which can be matched by any subtree. For example the schema $H=\{(+\ 3\ 4),(+\ 3\ 4),(-\ \#\ \#)\}$ represents all the programs including at least one occurrence of the tree fragment *(- # #)* and at least two occurrences of the subtree *(+ 3 4)*. The tree fragment *(- # #)* is present in all programs which include a '-' symbol. Like Koza's, this schema definition gives only the defining components of a schema not their position. Consequently, the same schema can be instantiated in different

ways, and therefore multiple times, in the same program. For example the program *(IF (+ 3 4) (+ 3 4) (- x 2))* instantiates the schema once while the program *(AND (+ 3 4) (+ 3 4) (+ 3 4) (- x y))* instantiates the schema three times because there are three ways to combine the two *(+ 3 4)* subtrees and one *(- # #)* fragment.

Based on the schema definition, the concept of *order* and *defining length* in GP are developed. The order of a schema is defined as the number of non-# nodes in the subtrees or tree fragments contained in the schema. For example, the schema *{(+ 3 4),(+ 3 4)}* has order 6 and the schema *{(- # #)}* has order 1. The defining length is defined using two factors: the schema and its instantiation program. The defining length of a schema is the number of links included in the subtrees and tree fragments *plus* the links which connect them together in its schema instantiation program. For example, the schema *{(+ 3 4),(+ 3 4)}* and its instantiation program *(IF (+ 3 4) (+ 3 4) (- x 2))* has defining length 4 (for schema) + 2 (for instantiation program) = 6. If a schema is instantiated in multiple programs in a population, the average number of links to connect the schema's subtrees for all the instantiated programs is used to calculate the defining length of the schema.

Using these definitions of schema, defining length and order, they developed a schema theorem for GP. In the theorem, the probability of disrupting a schema due to crossover is not a constant but varies depending on the shape, size and the composition of its instantiation programs. This is due to the fact that the defining length of a schema depends on the way a schema is instantiated inside the programs sampling it. O'Reilly and Oppacher argued that this variability of disruption from generation to generation makes the propagation and the use of building blocks (short, low-order relatively fit schemata) unattainable.

Whigham [1996b, Chapter 6] gave schema definition in his context-free grammar GP system. In his system, a program is represented as a derivation tree from a pre-defined grammar. The crossover and mutation operators are constrained to produce only valid derivation trees. A schema is a *partial derivation tree* rooted in some non-terminal symbol nodes. The schemata represented in a program are a collection of partial derivation tree (represented as production rules) organized into a single derivation tree (which represents the program). For example, consider the following derivation tree created using a context-free grammar:

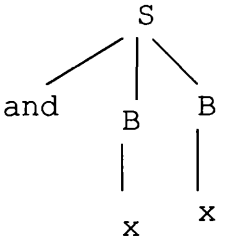


Figure 2.1: A derivation tree in a context-free grammar GP system.

```
S ::= andBB;
B ::= x;
B ::= y;
```

The program tree in Figure 2.1 can be created using the following derivation steps:

$$S \Rightarrow andBB \Rightarrow andxB \Rightarrow andxx$$

Meanwhile, the following schemata are represented in this derivation tree associated with these steps:

$$S \Rightarrow \ , B \Rightarrow \ , S \Rightarrow andBB, S \Rightarrow andxB, S \Rightarrow andBx, S \Rightarrow andxx, B \Rightarrow x$$

This definition of schema doesn't require a special symbol for *don't care* since every non-terminal (S and B in this example) in a partial derivation tree implicitly represents all legal strings that can be derived from the non-terminals. Also, a schema can appear multiple times in the same program since a schema derivation tree can be extended by applying one or more of the pre-defined grammar rules. For example, the schema $B \Rightarrow x$ is present twice in the program derivation tree. This is due to the absence of position information in the schema definition.

Whigham's definition of schema leads to a simple equation for crossover and mutation disruption of schemata without the need of defining length and order. However, as with that defined by O'Reilly and Oppacher, the disruption probabilities vary depending on the size of the derivation trees which the schema represents. To formulate his schema theorem, Whigham used the average disruption probabilities for all programs which a schema represents. This GP schema theorem differs from the one obtained by O'Reilly and Oppacher as the concept of schema is different. Whigham didn't draw any conclusion related to the building block hypothesis.

Rooted Trees Approach

In this approach, a schema is a rooted program tree or tree fragment. With this position restriction, a schema can only be instantiated at most once within a program tree. The study of the propagation of the components of the schema in the population is equivalent to analyzing the way the number of programs sampling the schema changes over time.

Rosca [1997a] formulated a schema definition, called *rooted tree-schema*, where a schema is a rooted and contiguous tree fragment. For example, the schema *H=(+ # x)* represents all the programs whose root node is a + and its second argument is *x* ("#" is don't care so it can be anything)[1]. He also gave a definition of *order* which is the number of functions

and terminals specified in the schema. The above schema example has order 2. His schema theorem doesn't use the concept of defining length. Instead, the disruption of a schema due to crossover is the *summation* of the disruption of all programs in the population that the schema represents. As an example, if there are 3 programs: *(+ x x)*, *(+ (- y 1) x)* and *(+ (- x y) x)* in the population which match the schema *H*, the disruption rate for the schema is the summation of the disruption of the three tree programs. The disruption of each program depends on 1) the size of the program, 2) its fitness and 3) the order of the schema represented. Rosca didn't provide conclusion regarding to building block hypothesis from his schema theorem.

Poli and Langdon [1997] defines a schema as a rooted tree fragment whose don't care symbol ("=") can only be matched by a *single* function or terminal. This makes a schema *H* represents only those programs which has the same shape as *H* and which have the same labels for the non-= nodes. The number of non-= symbols is called the *order* of a schema *H*, while the total number of nodes in the schema is called the *length N(H)* of the schema. The number of links in the minimum subtree including all the non-= symbols within a schema *H* is called the *defining length L(H)* of the schema. For example, the schema *(+ (1 = =) x)* has order 3 and defining length 2.

Using the concept of order, length and defining length, Poli and Langdon formulated a schema theorem for a GP system using *point mutation* and *one-point crossover*. Point mutation replaces a function in a tree with another function with the same arity or replaces a terminal with another terminal. One-point crossover works by selecting a common crossover point (the same position counting from the root of the tree) in the parent programs and then swapping the corresponding subtrees like the standard crossover. They have used the defined schema theorem to analyze the disruption of two groups of schemata: schemata representing programs with the same shape and size and schemata representing programs with different shape and size. The results indicate that these two groups of schemata interact with each other during GP run to optimize program *structure* and *contents*. According to their study, two conjectures have been made:

- During the early stage of GP run, the schema disruption rate is very high. The effect of fitness-proportional selection is counteracted by the crossover disruption.
- In the absence of mutation, after a while the population would start converging and the diversity of program shapes and sizes would decrease. During this phase competing schemata normally have the same size and shape, which is very much like GAs. There-

1. Rosca didn't use a "don't care" symbol since a schema is a rooted and contiguous tree. All nodes that extend the schema tree can be thought as "don't care" nodes.

fore, during this stage of GP run schemata with above average fitness, low order and short defining length (building blocks) would have a low disruption probability.

The purpose of formulating a schema theorem for GP is to provide an understanding of how GP searches for problem solutions. Yet, as the validity of the GA schema theorem is criticized, the usefulness of GP schema theorem is also under attack. In December of 1997, a heated debate went on the genetic programming list [GP-List, 1997] about the existence of building block, schema interpretation and the performance of crossover versus mutation. In fact, using a different approach, an enumeration of the program search space, [Langdon and Poli 1998a] has concluded that there is no building block in the artificial ant problem. Furthermore, they also demonstrated that the ability of GP to use building blocks to compose problem solutions is not exhibited in the `even-parity` problem [Langdon and Poli, 1998b]. We have used a similar approach to investigate the impact of a higher-order function program representation on the GP search process. This work will be described in Chapter 7.

2.3 Functional Programming Languages

This section summaries functional programming languages work which is related to this research. The λ calculus is presented first. This is followed by an overview of types and polymorphism. A polymorphic λ calculus is then given. Finally, two functional programming languages features, higher-order functions and recursion, are described at the end of this section.

2.3.1 Lambda Calculus

The λ calculus [Church, 1932-1933; Church, 1941] is usually regarded as the first functional language, although it was certainly not thought of as programming language at the time, given that there were no computers on which to run the programs [Hudak, 1989]. Modern functional languages can be thought of as the λ calculus (in various forms) with a lot of syntactic sugar.

λ expressions are expressions in the λ calculus. The abstract syntax of the *untyped* λ expressions is as the following:

`e ::= c`	built-in function or constant
`| x`	identifier
`| e1 e2`	application of one expression to another
`| λ x. e`	λ abstraction

In its purest form, the λ calculus does not have built-in functions or constants (`c` in the above syntax). In practice, however, every functional language provides some built-in functions and

constants, for example +. This extended version is therefore chosen as it represents modern functional languages more closely.

Expressions of the form λ x . e are called λ abstractions which represent function definition. Expressions of the form e_1 e_2 are called applications which represent the application of an expression to another expression. By convention, application is left-associative, so that (e_1 e_2 e_3) is the same as ((e_1 e_2) e_3). The following gives some examples of λ expressions:

```
a
+ 1 b
(λ x. + 1 x)
```

2.3.2 The Operational Semantics of the Lambda Calculus

This section provides the "calculation" part of the λ calculus: four conversion rules which describe how one λ expression is converted into another. The operation of *substitution* of the expression M for the *free variable* x in the expression *E*, denoted as E[M/x], is first explained since it is used by three of the four conversion rules.

The set of free variables of a λ expression E, which is represented in notion *fv(E)*, is defined by the following rules:

$$fv(c) = \{\ \}$$
$$fv(x) = \{x\}$$
$$fv(e_1 e_2) = fv(e_1) \cup fv(e_2)$$
$$fv(\lambda x\ e) = fv(e) - \{x\}$$

A variable x is free in E iff $x \in fv(E)$.

Substitution with the expression M of every free variable x in a λ expression E (denoted E[M/x]) is defined inductively by:

c [M/x] = c

x [M/x] = M

(e_1 e_2) [M/x] = e_1[M/x] e_2[M/x]

(λx. e) [M/x] = λx. e

(λy. e) [M/x] = λy. (e[M/x]) if $(x \notin fv(e)) or (y \notin fv(M))$

= λz. (e[z/y])[M/x] otherwise

The last rule is the most complicated one. It deals with name conflict and resolves it by making a name change. The following example demonstrates the application of this last rule:

```
(λy. + x y)[y/x] -> (λz. + x z)[y/x] -> λz. + y z
```

Now the terminologies are clear, the four conversion rules can be presented:

α-conversion: $\lambda x.\ e \leftrightarrow \lambda y.\ e[y/x]$, if $y \notin fv(e)$

β-conversion: $(\lambda x.\ e)M \leftrightarrow e[M/x]$

η-conversion: $(\lambda x.\ e\ x) \leftrightarrow e$, if $x \notin fv(e)$

δ-conversion: `evaluation of built-in functions`

The following gives examples of the application of these conversion rules:

$(\lambda x.\ +\ x\ 1) \overset{\alpha}{\leftrightarrow} (\lambda y.\ +\ y\ 1)$

$(\lambda x.\ +\ x\ 1)\ 6 \overset{\beta}{\leftrightarrow} (+\ 6\ 1)$

$(\lambda x.\ +\ 1\ x) \overset{\eta}{\leftrightarrow} (+\ 1)$

$+\ 1\ 2 \overset{\delta}{\leftrightarrow} 3$

Note that these four are *conversion rules* which allow the conversion to happen in either direction. When these rules are restricted to happen in one direction, these conversion rules are called *reduction rules*: α-reduction, β-reduction, η-reduction and δ-reduction. A λ expression can be reduced to another one by applying one of the four reduction rules.

Reduction Order

A λ expression is in normal form if it can not be reduced further using β or η rules. There are some λ expressions which do not have normal form, such as:

```
(λx. (x x))(λx. (x x))
```

where the only possible reduction leads to an identical term, thus the reduction process is nonterminating.

Furthermore, some λ expressions may or may not reach normal forms depending on the reduction order. For example, consider the following λ expression:

```
(λx. 3)(D D) where D is (λx. x x)
```

If we first reduce the application `(λx.3)(D D)` without reducing its argument `(D D)`, we got the result 3. However, if we first reduce the argument, we will never get any result as the process is nonterminating. This example raises an important question: can different reduction orders lead to different normal forms?

The Church-Rosser Theorem I [Church and Rosser, 1936] provides an answer for this question:

If $e_1 \leftrightarrow e_2$, *then there exists an expression e, such that* $e_1 \rightarrow e$ *and* $e_2 \rightarrow e$.

Corollary:

No lambda expression can be converted to two distinct normal forms.

Informally, the corollary says that all reduction sequences which *terminate* will result in the same normal form. The next question is which reduction order is mostly likely to terminate and to find the normal form?

The Church-Rosser Theorem II [Church and Rosser, 1936] provides an answer for this question:

If $e_1 \rightarrow e_2$, *and* e_2 *is in normal form, then there exists a normal order reduction from* e_1 *to* e_2.

These two Theorems promise that there is at most one possible result and normal order reduction will find it if it exists. So, what is normal order reduction?

A *normal order reduction* is a sequential reduction in which, where there is more than one reducible expression (called a *redex*), the *leftmost outermost* one is chosen first. In contrast, an *applicative order reduction* is a sequential reduction in which the *leftmost innermost* redex is reduced first. In the above example `((λx.3)(D D))`, normal-order reduces the λx-redex first while the applicative-order would reduce the `(D D)`-redex first. Intuitively, normal- order performs reduction on the body of a function first while applicative order performs reduction on the argument of a function first. In this example, normal order reduction produces a result while applicative order reduction will loop forever.

2.3.3 Typed Lambda Calculus

To introduce types into the λ calculus, the type language to be used has to be defined:

$\sigma ::= \tau$	basic type
$\mid \sigma_1 \text{ -> } \sigma_2$	function type

With the typed λ calculus, each λ expression is tagged with a member of the type language by superscripting, as in e^{σ}. Type σ_1 -> σ_2 denotes the type of all functions from value of type σ_1

to value of σ_2. Thus, the type of application $(e_1^{\sigma_1 \to \sigma_2} e_2^{\sigma_1})$ is σ_2. Modifying the λ calculus this way, a typed λ calculus can be derived:

e ::= c^{σ}	built-in function and constant
| x^{σ}	identifier
| $(e_1^{\sigma_1 \to \sigma_2} e_2^{\sigma_1})^{\sigma 2}$	application of one expression to another
| $(\lambda\ x^{\sigma_1}\ .\ e^{\sigma_2})^{\sigma_1 \to \sigma_2}$	λ abstraction

The typed conversion rules are as the following:

Typed-α-conversion: $(\lambda x^{\sigma_1}.e^{\sigma}) \leftrightarrow (\lambda y^{\sigma_1}.e^{\sigma}[y^{\sigma_1}/x^{\sigma_1}])$, if $y^{\sigma_1} \notin fv(e^{\sigma})$.

Typed-β-conversion: $((\lambda x^{\sigma_1}.e^{\sigma})M^{\sigma_1}) \leftrightarrow e^{\sigma}[M^{\sigma_1}/x^{\sigma_1}]$.

Typed-η-conversion: $(\lambda x^{\sigma_1}.e^{\sigma}\ x^{\sigma_1}) \leftrightarrow e^{\sigma}$, if $x^{\sigma_1} \notin fv(e^{\sigma})$.

Typed-δ-conversion: `evaluation of built-in functions`

The typed λ calculus supports a type system which is like that used in monomorphic typed languages. Modern functional languages do better than this by supporting polymorphism. The next section introduces the concept of polymorphism. A typed λ calculus which supports polymorphism will be discussed in Section 2.3.5.

2.3.4 Types and Polymorphism

Static Versus Dynamic Versus Strong Typing

In programming languages, *static typing* means that the type of every expression can be determined by static program analysis. This also means that before a program is executed, the type of every expression is known. By contrast, *dynamic typing* doesn't care about the type of an expression until program execution time. Dynamically typed languages use a run-time type checker and a run-time error handler to either report or repair type errors. Static typing is a desirable feature because it allows type errors to be detected at compile time, hence provides greater execution-time efficiency.

Although a useful feature, static typing can sometimes be too restrictive. Statically typed languages lose some flexibility and power of expression due to the premature constraint of the behavior of an object to a particular type. They exclude programming techniques that, although sound, are "incompatible with early binding of program objects to a specific type"

[Cardelli and Wegner, 1985]. For example, they exclude generic procedures, such as sorting, that represent algorithms which are applicable to a range of types.

Strong typing relaxes the restriction by allowing the exact type of an expression to be unknown at program compiling time. However, all expressions have to be type consistent. A program that is type consistent at compile time is guaranteed to be executed without run-time type errors. Note that every statically typed language is strongly typed, but the reverse is not necessarily true.

Kinds of Polymorphism

Strong typing can be implemented in programming languages in two different ways. Conventional typed languages, such as Pascal, are based on the idea that arguments of functions and procedures have an unique type. Such languages are called *monomorphic* languages. By contrast, *polymorphic* languages allow functions and variables to have more than one type.

Cardelli and Wegner [1985] have classified polymorphism as the following:

$$Polymorphism\begin{cases} universal\begin{cases} parametric \\ inclusion \end{cases} \\ ad\ hoc\begin{cases} overloading \\ coercion \end{cases} \end{cases}$$

Universally polymorphic functions work on a large number of types (all the types have a given common structure), whereas ad-hoc polymorphic functions only work on a finite set of different and potentially unrelated types. With universal polymorphism, a polymorphic function would operate on arguments of many types. However, this is not always true in ad-hoc polymorphism. An ad-hoc polymorphic function can be viewed as a small set of monomorphic functions and the set can contain only one element. In terms of implementation, a universally polymorphic function executes the same code for arguments of any admissible types, whereas an ad-hoc polymorphic function may execute different code for different type of argument.

There are two kinds of universal polymorphism: *parametric* and *inclusion*. In parametric polymorphism, a polymorphic function takes arguments of *any* type. The function performs the same kind of operation without considering the type of the argument. It is the purest form of polymorphism in the sense that any type is acceptable as its argument. With inclusion polymorphism, an object can be viewed as belonging to many different types which are related by inclusion. One particular instance of inclusion polymorphism is subtyping. With subtyping, a

type can be a subtype of another type. Furthermore, wherever a type may appear, its subtype may also appear. Functions that permit subtyping are polymorphic since the same operation can be applied to more than one type of argument.

There are two kinds of ad-hoc polymorphism: *overloading* and *coercion*. In overloading, the same name is used to denote different functions and the context is used to decide which function is denoted by a particular instance of the name. We may imagine that a preprocessing of the program will eliminate overloading by giving different names to the different functions; in this sense overloading is just a convenient *syntactic* abbreviation. On the other hand, a coercion is instead a *semantic* operation that is needed to convert the type of an argument to the type expected by a function, otherwise a type error would occur. Coercion can be provided statically, by automatically inserting the expected type in front of arguments and functions at compile time, or it can be determined dynamically by run-time tests on the arguments.

2.3.5 Polymorphic Lambda Calculus

One way to achieve *parametric* polymorphism in the typed λ calculus is to add *type variables* to the type language:

σ	::= τ	basic type
	| μ	type variable
	| σ_1 -> σ_2	function type

To accommodate this change, the typed β-conversion rule has to be extended to handle type variables:

Typed-β-conversion with type variables:

1. $((\lambda x^{\sigma_1} . e^{\sigma}) M^{\sigma_1}) \leftrightarrow e^{\sigma}[M^{\sigma_1}/x^{\sigma_1}]$

2. $((\lambda x^{\mu} . e^{\sigma}) M^{\sigma_1}) \leftrightarrow [\sigma_1/\mu](e^{\sigma}[M^{\sigma_1}/x^{\mu}])$

$[\sigma_1/\mu]$ is an operation which substitutes the type variable μ with type value σ_1.

Unfortunately, once type variables are introduced, it is no longer clear whether a λ expression is correctly typed. In fact, the general type-checking problem for this calculus is undecidable [Böhm, 1985; Pfenning, 1988].

Fortunately, [Hindley, 1969] and [Milner, 1978] independently discovered a restricted polymorphic type system that is almost as rich as that provided by the above calculus and for which type inference is decidable. The restriction of the type system is that the use of formal parameters in a function body must be monomorphic: all occurrences of a formal parameter

must have the same type. Since in practice, the class of programs that the type system rejects is not large, many functional programming languages, such as Haskell and Miranda, have already incorporated this type system to provide polymorphism in the language.

2.3.6 Higher-Order Functions and Partial Application

In functional languages, functions are treated as first-class values and can be stored in data structures, passed as arguments and returned as results for other functions. Functions which take other functions as arguments or produce other functions as results are referred as *higher-order functions*. The major philosophical argument for higher-order functions is that functions are values just like any others and should be treated just like other values. However, it's their pragmatic benefits which make higher-order functions appealing: they increase the use of abstraction. A function is an abstraction of common behavior over some values. Extend this use to functions increases the use of that kind of abstraction. As an example, the following higher-order function `twice` takes its first argument, a function `f`, and applies it twice to its second argument `x`.

```
twice f x= f(f x)
```

Now, we can use this higher-order function for many different function arguments (assumes `add10` and `multiply10` are defined before), hence providing another level of abstraction.

```
twice add10 1 = 21
twice multiply10 1 = 100
```

In modern functional languages, functions can be created in two ways: one is to name them using equations, such as the example function `twice`, and the other is to create them directly as λ *abstractions*, thus rendering them nameless. The following example defines a λ abstraction which takes one input and adds 1 to the value of the input. The result is the return value of the λ abstraction.

```
λ x. + x 1
```

In the λ calculus, functions can only be created using the second method. With higher-order functions and *currying* (named in honor of the mathematician Haskell Curry), a third way of function creation is provided. A function written in curried form allows its arguments to be partially applied. When only part of the arguments are provided to a function, a new function which expects the rest of the arguments is generated. This is called *partial application*. For example, the following is a λ abstraction defining a function taking two arguments.

```
λ x. λ y. ( + x y )
```

When only one argument, say 10, is given to the λ abstraction, a new function which expects one argument is generated.

```
λ y. ( + 10 y )
```

2.3.7 Recursion

Recursion is a general mechanism for program code reuse. When the name of a program appears in its program body, it is like making a new copy of the program code within the program. Recursion leads to more compact programs and can facilitate generalization.

Although a powerful reuse mechanism, recursion must be used carefully to be effective. There are two important criteria which are sufficient for a recursive program to terminate:

1. a terminating condition (base case);
2. the recursive calls are successively applied to arguments that eventually converge towards the terminating condition.

A recursive program which fails to meet either of the two requirements may or may not produce a result depending on the program evaluation style. With *lazy evaluation* (i.e. the normal order reduction described in Section 2.3.2) where arguments of a function are evaluated only if their values are needed, it is possible for programs containing infinite loops to halt. This happens when the code which contains infinite loops is not needed and therefore is not executed. On the other hand, *strict evaluation* (i.e. applicative order reduction described in Section 2.3.2) requires the evaluation of a function's arguments before the function body and can make such a program loop forever.

Implicit Recursion

A recursive function can also be implemented using implicit recursion. The relationship between explicit recursive calls and implicit recursion is demonstrated in Figure 2.2:

```
fun-name inputs
   apply code to inputs;
   recursive-call on inputs;
```

```
fun-name code inputs
   apply code to inputs
        recursively;
```

Figure 2.2: Explicit recursion versus implicit recursion

By extracting the code in a recursive function and making it an argument, the function becomes a higher-order function. Moreover, in this higher-order function, the code can be applied iteratively to the inputs, hence achieve the same recursion semantics as that using recursive calls.

This style of recursion implementation is only supported in functional languages, for example through the higher-order functions `map`, `foldr` and `filter`. Their operation is described as the following:

- `map`: applies the first argument, a monadic function (one which takes a single argument), to each element of the second list argument to produce a list of the results. For example:

```
map (+1) [1,2,3]
= [(+1 1),(+1 2),(+1 3)]
= [2,3,4]
```

- `fold`: places the first argument, a dyadic function (one which takes two arguments), between each of the items in the list. The `fold` function family contains two members: `foldr` and `foldl`. With `foldr`, the given terminating value (the second argument) is appended to the end of the expression and the resulting expression is evaluated with association to the right. For example:

```
foldr (+) 10 [1,2,3]
= 1 + (2 + (3 + 10))
= 1 + (2 + 13)
= 1 + 15
= 16
```

  With `foldl`, the given terminating value (the second argument) is prefixed to the expression and the resulting expression is evaluated with the association to the left. For example,

```
foldl (+) 10 [1,2,3]
= ((10 + 1) + 2) + 3
= (11 + 2) + 3
= 13 + 3
= 16
```

  Given the same arguments, `foldr` and `foldl` may or may not produce the same result.

More information about the differences between `foldr` and `foldl` functions can be found in [Clack, Myers and Poon, 1995, Chapter 4].

- `filter`: applies the first argument, a predicate operator (a function which returns True or False), to each element in the second list argument. It produces a list containing only the items which satisfy the predicate operator. For example:

```
filter (>1) [1,2,3]
= [2,3]
```

Implicit recursion is normally carried out by higher-order functions. This style of recursive programs do not generate infinite loops because the two required terminating conditions are always satisfied.

2.4 Summary

Background knowledge in GAs, GP and functional programming languages has been introduced in this chapter. The GAs/GP schema theorems and building block hypothesis, although criticized, provide us with a degree of understanding of the process of program evolution. Such understanding can assist us to analyze the impact of functional techniques on GP (see Chapter 6 and 7).

In the next chapter, related research will be presented. They are classified into four main categories: syntactic constraints using grammars, type constraints, modules and recursion.

Chapter 3

Related Work

This chapter summarizes research conducted in GP that is related to our work. This is classified into four categories: syntactic constraints using grammars, type constraints, modules and recursion. Those works use different approaches to tackle the same GP issues that this research is addressing. We identify their strength and weakness. They will be compared to our work in evolving λ calculus and in investigating the application and benefits of the following function programming techniques to GP, respectively: polymorphism, higher-order functions and implicit recursion.

3.1 Syntactic Constraints using Grammars

The traditional GP paradigm requires a "closure" property, which states that each "...function in the function set should be well defined for any combination of arguments that may be encountered" [Koza, 1992]. "Closure" allows unrestricted composition of the available functions and terminals in the program trees. Unfortunately, it also limits the applicability of GP because not all problems can be easily represented to satisfy this requirement.

Koza has realized this shortcoming and provided "constrained syntactic structures" to relax the closure property [Koza, 1992, Chapter 19]. With "constrained syntactic structures", a set of problem-specific syntactic rules are defined to specify which terminals and functions are allowed to be the child nodes of every function in the program trees. As an example, in the Fourier series problem [Koza, 1992, Chapter 19], the following syntactic rules are specified:

- The root of the tree is the function `&`.
- The only thing allowed below an `&` function is either an `&`, an `xsin` or `xcos` function.
- The only thing allowed below an `xsin` or `xcos` function is either a floating-point random constant or an arithmetic function (`+`, `-`, `*`, `%`).

Although this method worked adequately for the Fourier problem, it appeared as an ad hoc approach that would need to be modified for each new problem. Meanwhile, constraints represented in English are difficult to process during program evolution. An easier method to provide systematic specification of syntactic constraints is using grammars.

A grammar is a set of rules which specify the syntax of a language. A widely used notation is Backus-Naur form (BNF) or context-free grammar [Chomsky, 1956]. The λ calculus is specified in BNF. Other GP work which utilizes a context-free grammar or a logic grammar to specify syntactic constraints are summarized in the following subsections.

3.1.1 Context-Free Grammar Approach

Gruau [1995, Gruau and Whitley, 1996] presented a more general way to implement syntactic constraints in GP: syntactic constraints are represented as a context-free grammar and a computer algorithm is used to automatically generate program trees which conform to these constraints. The syntactic constraints are stored in a separate file which is a part of the GP parameters. Moreover, this file is parsed and translated by an enhanced GP algorithm. Using this approach, there is no need to re-implement a new GP algorithm each time the syntactic constraints are modified.

"A set of syntactic constraints is a context-free grammar. A valid GP tree is a parenthesized expression that can be generated by rewriting the axiom of the grammar, using the rewrite rules of the grammar" [Gruau, 1996]. For rules whose non-terminal[1] can be rewritten recursively, an upper bound of the number of recursive rewriting of this rule is specified. Implicitly, this bound specifies the size limit of the program tree. For example, in the following grammar, both <DNF> and <term> are non-terminals that can be rewritten recursively. The rules specify that <DNF> can only be rewritten at most 7 times while <term> can only be rewritten at most 4 times.

```
<axiom> ::= <DNF>
<DNF>[0..6] ::= or (<term>)(<DNF>) | <term>
<term>[0..3] ::= and (<literal>)(<term>) | <literal>
<literal> ::= <letter> | not (<letter>)
<letter> ::= A | B | C | D
```

1. A non-terminal in grammars is a symbol which can be rewritten using one of the grammar rules. A terminal, on the other hand, is a symbol which can not be rewritten.

To preserve the syntax of the programs, crossover can only be operated between two nodes which rewrite the same non-terminal symbol of the grammar. For example, with two program trees `(or (and A B) D)` and `(and (not C) D)`, crossover can be performed between the two `and` nodes since they rewrite the `<term>` non-terminal. Moreover, leaf nodes `A`, `B`, `C` and `D` can be exchanged between them since they rewrite the same `<letter>` non-terminal.

An unique feature of this work is that the grammar supports three types of data structures: list, set and array. The following is a modified version of the above grammar to generate programs with a list data structure:

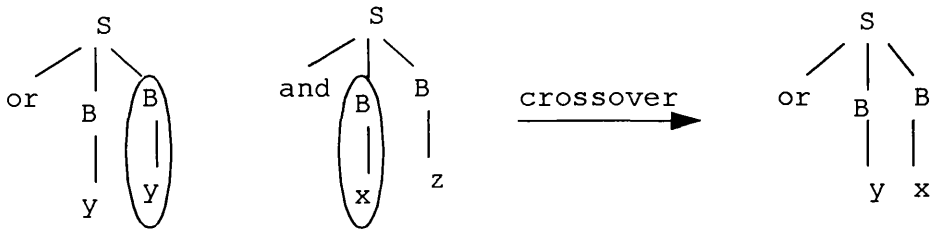
```
<DNF> ::= or list [1..7] of (<term>)
<term> ::= and list [1..4] of (<literal>)
```

The range of `list of` specifies interval for the number of elements of the list. When two subtrees are chosen for crossover and they are elements of the same type of list, the crossover will take place at the list level. For example, the crossover between trees t_1 = `or (and (C) (C) (C) and(C))` and t_2 = `or(and(A) (B) (B) (A))` can produce a new tree t_3 = `or (and (C) (C) (A) (B))`. In this way, these data structures in the program trees are preserved. Similar rules are applied to set and array data structures. With the support of data structures, Gruau's system essentially provide the implementation of both syntactic as well as type constraints of the GP program trees.

Whigham [1995, 1996a, 1996b] also investigated grammatically-based genetic programming system. However, his work emphasized on the use of grammars to bias GP learning. Similar to Gruau's, a context-free grammar is used to represent the language syntax and to guide genetic operators doing search. However, unlike Gruau's, Whigham's system used derivation trees to represent programs in the population. By keeping the information about the derivation of the program, crossover and mutation can be performed without the need of reconstructing the program. However, the reconstruction of programs is still needed for fitness evaluation.

Crossover in Whigham's system is done by swapping two derivation trees associated with the same non-terminal. For example, using the following grammar, two derivation program trees and their offspring are presented in Figure 3.1.

```
S ::= notB | andBB | orBB | x
B ::= x | y | z
```



(or y y) (and x z) (or y x)

Figure 3.1: Crossover operation in a grammatically-based GP system.

In addition, a *directed mutation* is designed to allow a more controlled search in the program search space. "A directed mutation specifies that one particular production used in a program derivation should be replaced by a second production" [Whigham, 1996b]. Directed mutation can be considered as a specialized mutation where the selection site for mutation is specified and the form of mutation is defined explicitly. An example is to specify that a derivation tree `B=>x` can only be replaced with a derivation tree `B=>z`. Directed mutation can also be used to specify larger derivation tree structures to be replaced. For example, one can specify that the derivation tree `S=>notB=>notx` to be replaced with `S=>andBB=>andzz`. Whigham has used this particular mutation operator to identify pattern in the programs which should be replaced with recursive calls. This work will be summarized in Section 3.4.

Freeman [1998] has also proposed the use of a context-free grammar to allow a linear representation of GP programs. Each individual program is an array of integer values which represent the *rule numbers* in the grammar. Crossover and mutation performed on such a representation produce some interesting effects. This style of program representation also provide more economic space usage. However, there is overhead of the reconstruction of program tree for fitness evaluation. This use of context-free grammar is to allow a linear representation of the program. There was no mention of constraints in her work.

3.1.2 Logic Grammar Approach

Wong and Leung [1995, 1996, 1997] employed a logic grammar to represent context-sensitive information and domain-dependent knowledge in their GP system. A logic grammar differs from a context-free grammar in that the logic grammar symbols, either terminals or non-terminals, may include *arguments*. An argument can be in one of the following 3 forms:

- a variable, represented by a question mark? followed by a string of letters/digits, e.g. `?x`;
- a function, represented as a function name followed by a bracketed *n*-tuple of terms;

- a constant, represented as a 0-arity function, e.g. `W`.

The following is an example logic grammar used in their system:

```
start -> [(*], exp(W), exp(W), exp(W),[)].

exp(?x) -> [(/ ?x 1.5)].
```

The programs are represented as derivation trees which show how the programs are derived from the logic grammar. For example, Figure 3.2 is a derivation tree representing a program using the above logic grammar. The genetic operators of crossover and mutation are modified so that only valid derivation trees are generated.

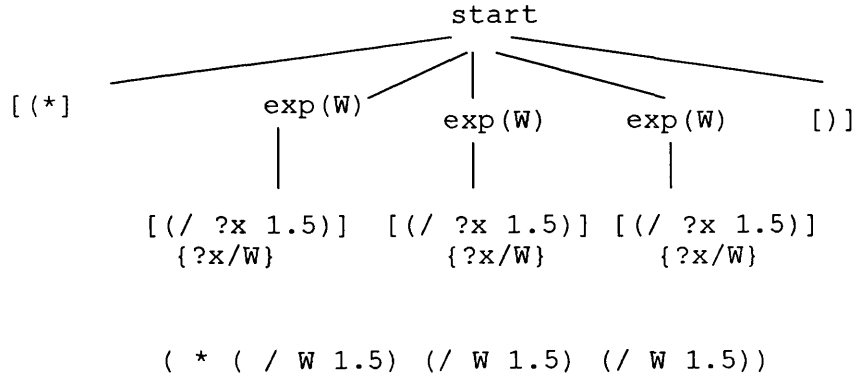


Figure 3.2: A derivation tree in the logic grammar GP system.

With the extra token of arguments to be associated with each terminal and non-terminal, a logic grammar allows more domain-dependant information to be expressed. For example, to evolve a general solution to the `even-parity` problem, arguments are used to specify that the type of the non-terminal `S-Exp` has to be a list, e.g. `S-Exp(List)` [Wong and Leung, 1996]. In addition, the grammar also specifies a semantic constraint that if the input argument `L` is an empty list, the program returns `True`. The purpose of this semantic constraint is to enforce the termination condition for recursive calls in the program (see Section 3.4 for more details).

A grammar which is capable of representing syntactic, type and semantic constraints is a powerful tool. Yet, it requires a powerful interpreter to process such a grammar. In our work, syntactic and type constraints are separated; each of which is specified with a grammar of its own. Type checking is performed by a type system which is also called upon by a syntactic checker during program generation and evolution. This system will be described in Chapter 4.

3.2 Type Constraints in Genetic Programming

Koza's "constrained syntactic structures" provides probably the most primitive implementa-

tion of typing for GP. For example, in the Fourier series problem described in the Section 3.1, three different types are defined by the three typing rules. The typing mechanism is implemented by labelling each program node with a symbol number that defines the level in the program tree where the node may exist. These special symbols were used to ensure that crossover and mutation do not violate the syntactic constraints.

Type constraints in GP have been extended by others as a method to reduce GP search space for problems which involve multiple types in order to find solutions faster. Although the relationship between the difficulty of a problem for GP to solve and the size of its search space has been questioned [Langdon and Poli, 1998a], the reported performance improvement due to type constraints (see the following subsections) can not be dismissed. If it is not a result of reduced search space, there must be another explanation which deserves investigation. Moreover, the ability to support multiple types has opened up a wide variety of applications for GP [GP-List 1998].

3.2.1 Strongly Typed Genetic Programming

Montana [1995] has developed Koza's "constrained syntactic structures" so that type constraints are given indirectly, through a type system, rather than directly (per-function). In his system, a table giving the types of all available terminals and functions is maintained. Thus, if a function takes an argument of type X then this implicitly constrains its child node in the program tree to produce a value of type X. A second table, type possibilities table, is generated before the GP runs. "Such a table tells for each i = 1,... MAX-TREE-DEPTH what are the possible return types for a tree of maximum depth i." [Montana, 1995]. This extra information constrains the choice of function to create nodes in the tree so that the tree can grow to its maximum depth, i.e. a function can be used to construct a node at tree depth `i` only if all its argument types can be generated by tree depth `i-1`. During the creation of the initial population, each program tree is grown top-down by choosing functions and terminals at random within the constraints of the types in the table. The crossover and mutation operators also have to consult the table. In this way, the population only consists of program trees that are type-correct.

Additionally, Montana has extended the type notion to include generic functions and generic data types. Generic functions are functions which accept arguments of any type. Generic data types are type variables used to specify that the generated programs can accept arguments of any type. These two features support *parametric polymorphism* and strong typing discussed in Section 2.3.4., he therefore called the system "Strongly Typed Genetic Programming".

However, we believe there are two key problems with Montana's work:

1. The implementation requires the creation of type possibilities table. The table is built started from entry `1` (leaf level) and moves up one level at a time. All terminals can be on the leaf level so their types are in entry `1`. To build entry `i`, each generic function has to have its argument type variables instantiated with types in level `i-1`. The instantiated return type of the generic function is then added into entry `i` in the possibilities table. This process is applied to every generic function to build one single entry and repeated for every entry to build the table. The computation time varies depending on the number of generic functions in the function set and the specified tree depth limit.
2. The type possibilities table approach also makes the implementation to support function types difficult. To evolve the higher-order function `mapcar` (which is the same as the `map` function described in Section 5.4.2) whose argument is of function type, Montana has adopted an ad-hoc manner: instead of passing the function argument during program evaluation, the function argument is provided as a member of the function set. Although this approach works for the `mapcar` program, the general support for function types is not provided in his system.

These two issues will be addressed in our type system described in Chapter 4.

The concept of using type constraints to reduce the size of the search space is also investigated by [Haynes, Wainwright, Sen and Schoenefeld, 1995]. They demonstrated that STGP outperformed standard GP for the problem of evolving cooperation strategies in a predator-prey environment. They concluded that the improved performance is due to the reduced search space attained by STGP. They also showed that the programs generated by STGP tend to be easier to understand.

STGP has also been applied to image processing applications. Harris [1997] used STGP to enforce a hierarchy in the program trees by defining a set of types based on the problem domain knowledge. Moreover, functions in the function set are specified with these types so that a hierarchy of program structure is maintained. This approach has been shown to be advantageous in solving an image template matching problem. In [Lucier, Mamillapalli and Palsberg, 1998], STGP is used as an optimization technique to speed up the evolution of an edge detector for mammogram images.

3.2.2 SubTyping

One kind of *inclusion polymorphism* (see Section 2.3.4), subtyping, is introduced to GP by [Haynes, Schoenefeld and Wainwright, 1996]. In their system, generic functions support sub-

typing: if an argument is of type A, it also accepts any values of subtypes of A. Subtyping is implemented using a table-lookup mechanism similar to that of Montana's STGP. However, during the selection of functions and terminals to construct program nodes, one extra check has to be made: the selected function has to have both its argument types and the subtypes satisfying the type constraints specified in the type possibilities table. Their type system supports subtyping with a non-branch type hierarchy, where each type is allowed to have a maximum of one supertype and one subtype. They demonstrated that it is essential for the functions to support subtyping in order to evolve the solution to the maximum clique problem [Kalmanson, 1986].

3.2.3 Higher-Order Function Types

We pioneered the investigation of using higher-order function types to provide a partial application style of program representation [Clack and Yu, 1997]. A higher-order function can return a function, i.e. the returned value has a function type. When a function is partially applied (only part of its arguments are provided), it returns a function which expects the rest of the arguments (see Section 2.3.6). By supporting this use of higher-order function, a new partial application style of program representation is introduced to GP.

In the standard GP system, the program representation is expressed in a full application manner, i.e. each function node represents a function where all its arguments are provided. For example, the program `(* (+ x y) y)` is represented as Figure 3.3 (a). Both of the function node + and * are full application nodes.

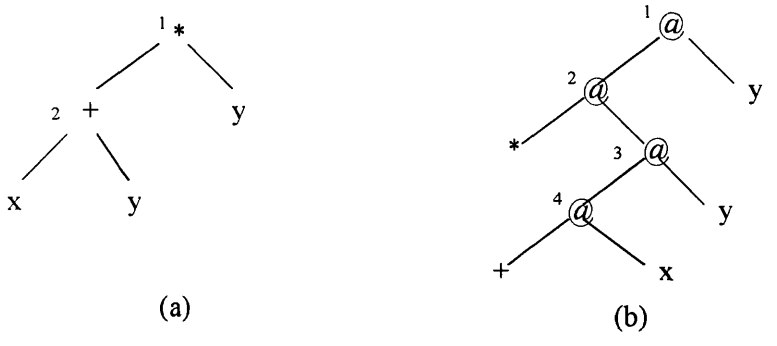


Figure 3.3: Standard GP versus partial application program representation.

In contrast, a partial application style of program representation allows a function node to represent a function where only part of its arguments are provided. For the previous example, the program tree is represented in Figure 3.3 (b). The @ symbol indicates a function application node. Node 4 is a partial application node where function + is only provided with one argu-

ment x. On the other hand, node 3 is a full application node where both arguments, x and y, are given to the function +. Node 3 in Figure 3.3 (b) corresponds to node 2 in Figure 3.3 (a).

A program representation which allows partial application to be expressed provides more locations for crossover and mutation, e.g. 9 versus 5 in the above example. However, to assure that the genetic operation generates valid program trees, the system has to be able to distinguish a partial application node from a full application node. This can be achieved by using type information: a partial application node is associate with a function type while a full application node is associated with a value type. We have implemented a type system which supports function types to create this new style of program representation for GP (see Chapter 5).

This work has been farther expanded by [McPhee, Hopper and Reierson 1998] who observed that this representation allows crossover to change a function in the program trees without affecting the arguments, e.g. crossover performed on the + node in Figure 3.2(b). Moreover, the probability to select leaf for crossover is reduced. In problems that are highly sensitive to the choice of functions at or near the root, such as MAX problem [Gathercole and Ross, 1996], this program representation can produce very different performance than that of the traditional GP. To investigate this particular effect of the new style of program representation, they conducted a series of experiments on different problems. The results showed that such a representation produced better performance than that produced by the standard GP representation on some problems but has no effect on the others. For problems whose program solutions have an obvious desired root, the partial application style of program representation is beneficial. On the other hand, for problems where the optimal solutions can be of different roots, both representations have similar performance. Further research is needed to gain more understanding of how this program representation impacts GP search.

3.2.4 Type Constraints using Sets

In the constrained genetic programming system developed by [Janikow 1996], type constraints are specified using two different sets: one set contains legal functions and arguments combinations while the other contains illegal functions and arguments combinations in the program trees. The first set is very similar to Montana's type possibilities table (see Section 3.2.1) while the second set provides a further restriction of the program search space. Moreover, these two sets can contain redundant and incompatible information as they will be transformed into *minimal normal forms* which are to be used during program generation and evolution to ensure that only valid programs are generated.

Constraints specified in the legal/illegal sets do not have to be type related. Janikow and

DeWeese [1998] provided an example of using the system on the single-typed multiplexer problem [Koza, 1992] where constraints specified are domain knowledge about the beneficial ways of constructing program structure. This is another approach of using one representation to express syntax, type and semantic constraints in one framework.

3.3 Modules in Genetic Programming

The original GP paradigm has no explicit support of module creation and reuse. To enhance GP's ability to scale up to larger and more complex problems, various module approaches have been proposed. These include Automatically Defined Functions (ADFs), Module Acquisition (MA), Adaptive Representation through Learning (ARL) and Automatically Defined Macros (ADMs).

3.3.1 Automatically Defined Functions

ADFs [Koza, 1993; Koza, 1994a; Koza, 1994b] are mechanisms devised by Koza to facilitate the creation and reuse of modules. An ADF is "a function (i.e., subroutine, procedure, module) that is evolved during a GP run and which may be called by the main program (or other calling program) that is being simultaneously evolved during the same run" [Koza, 1994a]. When solving a problem that has considerable regularities in its solutions, ADFs provide GP a mechanism to automatically decompose the problem into subproblems and then reuse the solution to the subproblem to solve the overall problem. On some problems, GP with ADFs can generate simpler program more efficiently [Koza, 1994a; Koza, 1994b; Handley, 1994; Koza, Andre, Bennett III and Keane, 1996]. On others, such as the two-boxes problem, ADFs do not provide any advantage [Koza, 1994, Chapter 4].

An ADF is an independent module which is evolved using separate function and terminal sets from the ones used to evolve the main program. To provide better modularity for the problem solutions, [Andre, 1994] has suggested that the function sets for ADFs should be grouped by functionality. In this way, each ADF is designated with a task defined by the function set. Moreover, each ADF can be specified with different parameters such as tree depth and crossover rate to suite its designated task. Consequently, the evolution of ADFs and the main program are similar but independent.

In the original ADF implementation, fitness evaluation is applied to the whole program and there is one fitness value for the whole program. During the fitness evaluation of the main program, when the name of an ADF is encountered, the evaluation process is suspended. A new process is created to evaluate the ADF. The outputs of the ADF is then used by the main

program to continue its fitness evaluation. The final result of the evaluation becomes the fitness of the overall program. With parallel distributed GP [Poli, 1996], non-parametrized reuse and automatically defined links are used to provide more efficient program evaluation.

The structures of programs with ADFs (the number of ADFs in the programs) can be defined in three different ways. The first approach is to statically define it before the GP run. Once the structure is specified, every program in the population has the same structure. Genetic operators are customized to preserve the program structure. The second approach is to have various kinds of program structure randomly created in generation 0. Program structure is then open to evolutionary determination [Koza, 1994, Chapter 21]. The last approach is to define the programs in generation 0 with the same structure. Six architecture-altering operators are then used to evolve program structures during GP runs [Koza, 1995]. With a predefined program structure, GP does not have the opportunity to explore more advantageous structures. When an unsuitable program structure is given, GP is doomed. On the other hand, leaving GP to determine the program structure among a wide range of possibilities can be computationally expensive. The architecture-altering operations require more computational effort for GP to solve problems [Koza, 1995]. We have identified these shortcoming and provided a better way to define program structures based on the specification of higher-order functions in function set. This work will be discussed in Chapter 6.

3.3.2 Module Acquisition

MA [Angeline and Pollack, 1992; Angeline and Pollack, 1993; Angeline 1994] is a method to support modules reuse by creating a library of subtrees extracted from the program trees in the population. Unlike ADFs, which are locally defined for each program tree and which can be called by one individual possibly many times, modules in MA are globally defined and can be used by other individuals in the population.

Two additional genetic operators are introduced in MA: *compression* and *expansion*. Compression creates subroutines from subtrees of individuals in the current population and introduces the subroutines into a "genetic library". A name is then given to the created subroutine and to replace the subtree extracted in the program tree. Figure 3.4 shows the operation of compression operator. Expansion is the opposite of compression: it replaces the name of a subroutine with its correspondent subtree.

The motivation behind MA is to solve the scaling problem in GP. With the dynamic nature of its representation, GP program trees become very large when learning to solve a very complex task. Consequently, the chance of breaking up desirable portions of the program during genetic operations overwhelms the chance of improving the program. By using

the compression operation, a large tree can be represented in a more compact way without changing its semantics. In Figure 3.4, this operation reduces the number of nodes from 10 to 5 without altering the semantics of the program.

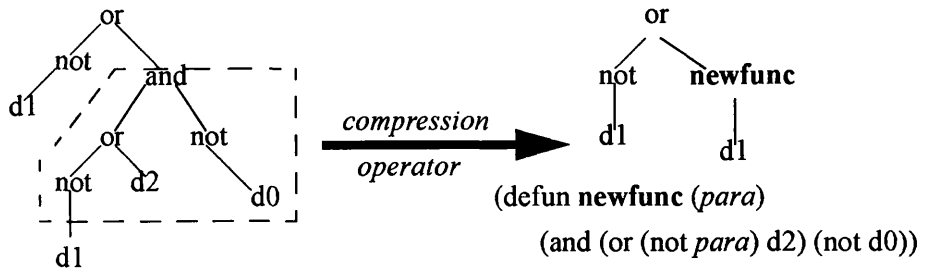


Figure 3.4: Compression operator in module acquisition.

Angeline argued that the reuse of the created modules is achieved automatically through the fitness-based reproduction without any additional intervention. Initially, when a new module is created there is only one member of the population which has a reference to it. However, if the program is comparatively fit, it will be selected for reproduction. Consequently, the sub-tree which contains the module name will be copied into several offspring. On the other hand, if the program is comparably unfit, the call of the module will less likely to be copied into the next generation. As a result, good modules are promoted while bad modules are killed off automatically by the dynamics of the evolutionary process. There is no explicit heuristics needed.

Unfortunately, while the compression operator supplies a method to create subroutines from the population, it also reduces the diversity of the population. For a genetic search to work, there must be sufficient genetic material in the population so that the combination of promising candidates can generate novel programs. The expansion operation remedies this shortcoming by restoring the genetic material for the compressed subtrees.

Kinnear Jr. [1994a] has summarized some important characteristics about MA:

- Modules, once defined, don't evolve. The genetic material in modules does come back into play through module expansion, but its identity as a module is lost at that point. It is highly unlikely that a similar module would be defined from that material at a later time.
- Modules can never be recursive, although they may call other modules to any arbitrary depth.
- As defined, modules use each argument exactly once. Since module definitions don't evolve, there is no possibility of generating multiple uses of a parameter through evolution.

- Module definitions contain the same function and terminal set as the original population.

3.3.3 Adaptive Representation through Learning

Similar to MA, the ARL method extracts program segments to create functions [Rosca and Ballard, 1994; Rosca and Ballard, 1996; Rosca, 1995; Rosca, 1996]. However, instead of being added to a genetic library, these functions are added into the GP function set to be used for the creation of the next generation. Moreover, these dynamically created functions may be deleted from the function set when their usage doesn't prove to be advantageous. Consequently, the size of the GP function set expands and shrinks during the GP run.

The motivation behind the dynamic creation and deletion of functions in GP is to promote the reuse of "good" program code. To achieve this goal, two issues have to be addressed: *what* is "good" program code and *when* to create and to delete these program code. The ARL approach addresses these two issues. Good program code is detected using local measurement such as parent-offspring differential fitness and block activation. In addition, global measurement such as population entropy are used to predict when the evolutionary search reaches local optima so that the modification of the function set can be performed to escape from it. Using these heuristics to detect good program segments for module creation, ARL produces better programs than GP alone [Rosca and Ballard, 1996]. A recent research, however, reported that ARL do not work too well [Dessi, Giani and Starita, 1999].

3.3.4 Automatically Defined Macros

Spector [1996] has proposed the use of Automatically Defined Macros (ADMs) in GP to simultaneous evolve programs and their control structures. The difference between ADFs and ADMs is that an ADF is evaluated in its local environment while an ADM is evaluated in the main program global environment. When the name of an ADM is called in the main program, a contextual substitution (macro expansion) is performed. The ADM body is then evaluated in the main program environment. With this style of evaluation, one can use ADMs to implement program control structure. This is done by passing a block of code as argument to ADMs. Since the evaluation of the arguments is carried out in the main program, depending on the calling environment, an argument may or may not be evaluated. ADMs therefore provide a mean of evolving programs and their control structures that perform multiple evaluation or conditional evaluation of blocks of code. For example, they allow speeding up the evaluation of Boolean functions.

ADMs are beneficial when the arguments perform side-effect operation which are sensi-

tive to their calling environment. For example, in the obstacle-avoiding robot problem [Koza, 1994a], GP is to evolve a program which performs side-effect functions such as `LEFT` and `MOP`. These function are sensitive to the robot's surroundings: the `MOP` function moves the robot in the direction it is currently facing (if no obstacle is ahead), mops the floor at the new location and returns a location of `(0, 0)`. Using ADMs, the `MOP` function is allowed to react differently according to the currently environment where the `MOP` is called upon. With ADFs, the `MOP` function performs the same way independent to where the `MOP` is invoked. To achieve the desired side-effect, an ADF requires an extra argument which specifies the current environment. The `MOP` in the ADF can then react to the environment accordingly. Experiments on the obstacle-avoiding robot problem shows that GP with ADMs performs better than GP with ADFs in this particular application.

3.4 Recursion in Genetic Programming

Koza has investigated a problem-specific form of recursion to solve the Fibonacci sequence induction problem [Koza, 1992, page 473]. The Fibonacci sequence can be computed using the recursive expression: $S_j = S_{j-1} + S_{j-2}$ where S_0 and S_1 are both 1. After these two elements of the sequence, each element of the sequence is computed using two previous values of the sequence, e.g. $S_2=S_1+S_0$; $S_3=S_2+S_1$, etc.

To allow a program to reference previously computed values in the sequence, a sequence referencing function `SRF` is introduced into the function set. When a program containing the `SRF` function is being evaluated for value of position `j`, the `(SRF k D)` computed the value for sequence position `k` provided `k` is between 0 and `j-1`, otherwise, it returns the default value `D`. The `SRF` function is useful for the Fibonacci sequence problem. However, it can not be used to generate general forms of recursive programs.

Brave [1996] has used GP to evolve programs with recursive ADFs to perform tree search. To evolve a recursive ADF, the name of the ADF was included in its function set. However, an evolved recursive ADF may contain infinite-loops (see Section 2.3.7). To deal with this problem, he specified the depth of the tree as the limit of the recursive calls. Usually such a limit affects the evolution process since a good program may never be discovered if its evaluation requires more than the permitted recursive calls. This shortcoming, however, does not apply to a tree search program since the maximum number of iterations required to search a tree is its tree-depth. Stopping any recursion after tree-depth number of iterations therefore does not affect the behavior of the program. However, this property is not present in general problems. Thus, Brave's approach is not a general solution to evolve recursive programs.

By using recursive ADFs, Brave showed that GP can find solutions to the tree search problem faster than that using non-recursive ADFs. Moreover, the program containing recursive ADFs is less complex and requires less computational effort to execute than the programs with non-recursive ADFs.

Wong and Leung [1996] proposed the use of recursion to evolve a general solution for the `even-parity` problem. Their approach is to construct a logic grammar (see Section 3.1.2) which includes a rule making recursive call. In addition, the grammar enforces a termination condition in the program structure. However, the convergence of recursive calls in the program is not guaranteed. When evaluating programs, they used an execution time limit to halt the program. They have shown that using such a grammar to guide evolution, GP is able to find the solution to the general `even-parity` problem more efficiently than Koza's ADFs approach. Yet, our approach of evolving recursive programs provides even better performance than Wong and Leung's in this problem. This work will be presented in Chapter 6.

Whigham has designed two *directed mutation operators* to guide GP to evolve a recursive `member` function using his GP system [Whigham, 1996b]. As described in Section 3.1.1, Whigham developed a context-free grammar GP system where each program is represented as a derivation tree. A directed mutation operator specifies that a subtree generated by one particular grammar rule should be replaced by another subtree generated by a different rule. The two directed mutation operators he designed for recursion serve for two different purposes. The first one is to repair programs that contain tautologies by replacing the derivation tree `(eq x x)` with `(eq x (car y))`. The second one is to detect the pattern in the derivation tree where a recursive call should take place: `(eq x (car (cdr (cdr y))))` is replaced with `(member x (cdr y))`.

These two directed mutation operators have improved the likelihood of evolving the recursive `member` function. According to Whigham, this is due to: "The first directed mutation seeds the population with building blocks that will create the intermediate step towards a recursive definition. The second directed mutation can exploit the increased bias towards the pattern used for recursion." Yet, these two mutation operators are problem-specific, i.e. they are designed for the `member` function. The knowledge about the solution is nicely used to direct GP search. For problems which do not have an obvious recursive pattern, this approach may not be appropriate.

3.5 Summary

This chapter has presented research in GP which is related to our work. The use of grammar

to specify syntactic constraints for GP has been advocated for some time. Whigham is a particularly strong believer of using grammar to support language bias and search bias for GP learning. We follow this avenue by evolving a grammar-based language, the λ calculus, whose syntax encompasses the definition of modules. This work will be described in the following chapter.

Type constraints, originally promoted for their ability to reduce search space, are receiving much attention recently due to a research result indicating that the size of the search space is not necessary the indication of the difficulty of problem for GP to solve [Langdon and Poli, 1998a]. In spite of this controversy, many researchers are interested in types simply because there are many applications require GP to be able to handle multiple types [GP-List, 1998]. Moreover, the reported performance improvement provided by type constraints is a fact that can not be neglected, although the cause behind it needs farther investigation. We will present the concept of types in GP with the implementation of polymorphism in Chapter 5.

Module creation and reuse has been acknowledged as an important problem solving method. Various work in this area has demonstrated their success in helping GP learning. We will present our novel approach of module creation and reuse in Chapter 6 and 7.

Recursion, although an important programming technique, has only been implemented in some restricted manner in GP. Our approach of using implicit recursion to evolve a general solution for the `even-parity` problem will be presented in Chapter 6.

Chapter 4

The Functional Genetic Programming System

This chapter describes a new GP system, implemented with various functional programming techniques incorporated. Firstly, the system evolves programs based on the syntax of the λ *calculus*. Secondly, the program representation can contain modules which are represented as λ *abstractions*. Finally, the program constructs are allowed to have multiple types. The type checking of the programs is performed by a *polymorphic type system*. The high-level system structure is presented in Section 4.1. This is followed by the detailed description of each component of the system (Sections 4.2-4.5). Section 4.6 discusses the implementation details such as the Genetic Algorithm and the programming language used in the system. In Section 4.7, an example is presented to demonstrate the operation of the system.

4.1 System Structure

The functional GP system has four major components: *creator*, *evaluator*, *evolver* and *type system*. Figure 4.1 illustrates the high-level structure of the system.

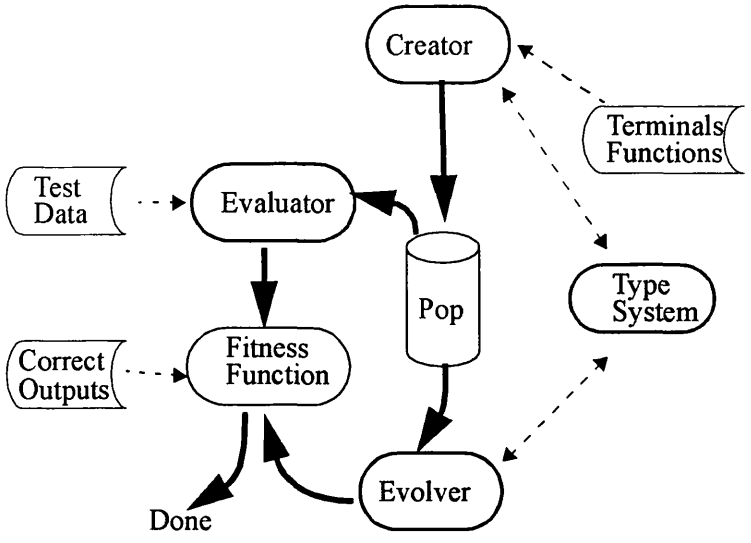


Figure 4.1: High-level system structure of the functional GP system.

Initially, the creator interacts with the type system to select type-matched functions and terminals to create a population of type-correct programs. Each program is then evaluated by the evaluator using test data as inputs to produce some outputs. Next, the outputs are passed over to the fitness function which assigns a fitness value to the program according to the correctness of the outputs. If the fitness value satisfies the requirement, the system stops and returns the program with the satisfactory fitness value as the solution. Otherwise, "good" programs are selected for the evolver to perform genetic operation and to create a population of new programs. This test-select-reproduction process is repeated until either a satisfactory program is found or the termination criterion is met.

Compared with the standard GP system, the functional GP system has an extra component: a type system. The type system is used by creator and evolver to ensure that only type-correct programs are generated. To use the type system, users have to specify input and output types for each function and terminal in the function and terminal sets. The type syntax and the details of the type system are provided in Section 4.5.

4.2 Creator

The programs created are represented as trees. The creator grows a program tree from the top node downwards. There is a user-specified type for the root node of the tree. The creator invokes the type system to select a function whose return type satisfies the required type to construct the root node. The selected function has arguments to be created at the next level in the tree: there will be type requirements for each of those arguments. If the argument has a function type, a λ abstraction will be created to represent the function argument (see Section 4.2.1). Otherwise, the type system randomly selects a function (or terminal-see next paragraph) whose return type satisfies the newly required type to construct the argument node. This process is repeated until the specified tree depth limit is reached.

When selecting functions to construct argument nodes, it is possible that there is no function in the function set whose return type can satisfy the required type. In this case, the creator stops growing the tree by selecting a terminal whose type satisfies the required type. Terminals are also randomly selected to construct leaf nodes. However, depending on the types defined in the function and terminal sets, program creation using this method of random selection of type-matched functions and terminals may not succeed. When the creation of a particular subtree fails, the creator backtracks to the root node of the subtree and regenerates a new subtree.

The Creator performs chronological backtracking which withdraws the most recently

made wrong choice, selects an alternative at that choice point and move ahead again. If all the alternatives at the last choice point have been explored already, then go further back until an unexplored alternative is found.

This dynamic approach to generate type-correct programs is different from the static approach used in STGP (see Section 3.2.1). In STGP, a type possibilities table is computed beforehand to specify all the types that are possible to be generated at each tree depth level. This table is then consulted by the function selection procedure to generate type-correct programs. In contrast, the functional GP type system uses a contextual instantiation method (see Section 4.5.3) to dynamically instantiate type variables. Hence, there is no need to construct type possibilities table.

The contextual instantiation is carried out by an unification algorithm (see Section 4.5.2). The linear-time implementation of the algorithm has been devised by [Paterson and Wegman, 1978]. However, the functional GP type system has not incorporated such an implementation. Compared to the table look-up method used in STGP to instantiate type variables, unification algorithm may or may not be more efficient. The evaluation of the two methods in terms of CPU time requires future study.

The biggest advantage of unification algorithm over table look-up is its generality: unification algorithm is capable of handling any type structure, e.g. function type. This is the area where table look-up method would require extra work.

However, there is an overhead associated with the functional GP type system: there is a possibility of backtracking when the tree can not be created successfully within the specified tree depth. To estimate the chance of its happening, 10,000 `map` programs (see Section 5.4.2) were generated using this method. Among them, 85% were successfully created without backtracking. More study is required to have better understanding about the frequency of backtracking during program generation.

In summary, functional GP and STGP adopt different approaches to implement their type systems. Yet they both achieve the same goal of generating type-correct programs.

4.2.1 Lambda Abstractions Creation

If the tree node being constructed is a function which has a function as one of its arguments, this function type argument is created as a λ abstraction. λ abstractions are local function definitions, similar to function definitions in a conventional language such as C. The following is an example λ abstraction together with similar C function.

```
(λ x. (+ x 1))                    (λ abstraction)
```

```
Inc (int x){return (x+1);}        (C function)
```

λ abstractions are created using the same function set as that used to create the main program. Thus a λ abstraction may contain another λ abstraction. The terminal set, however, consists only of the arguments of the λ abstraction to be created. In the current implementation, no global variables or constants are allowed. Argument naming in λ abstractions follows a simple rule: each argument is uniquely named with a hash symbol followed by an unique integer, e.g. #1, #2. This consistent naming style allows crossover to be easily performed between λ abstractions with the same number and type of arguments (see Section 4.4.3).

4.2.2 Curried Format Program Representation

The program trees are represented in a *curried* form (a function is applied to one argument at a time), thus allowing partial application to be expressed (see Section 3.2.3). The motivation behind this style of representation is to provide more locations for genetic operation so that more diverse new programs can be created. With more diverse programs in the population pool, it is hoped that GP can find solutions faster. However, this hypothesis has not been verified.

McPhee, Hopper and Reierson [1998] also investigated this style of program representation. Their work has been summarized in Section 3.2.3.

With a curried format program tree, each function application has two branches: a function and an argument. Figure 4.2 is the curried format program tree for the `IF-TEST-THEN-ELSE` function. The @ denotes an application node (see Section 3.2.3) and is a possible genetic operation location. The function `(IF(TEST-exp)(THEN-exp)(ELSE-exp))` has two branches: `(IF(TEST-exp)(THEN-exp))` and `(ELSE-exp)`. The first function branch, `(IF(TEST-exp)(THEN-exp))`, also has two branches: `(IF(TEST-exp))` and `(THEN-exp)`. The `(IF(TEST-exp)` also has two branches: `IF` and `(TEST-exp)`.

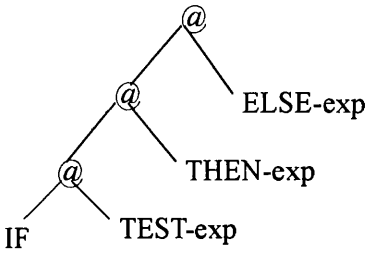


Figure 4.2: Curried format program tree for the IF-TEST-THEN-ELSE function.

The curried format program trees are generated in a depth-first-right-first manner, i.e. the argument subtree is created before the function branch subtree. In the `IF-TEST-THEN-`

ELSE program tree example, the ELSE-exp subtree is first generated, followed by the THEN-exp subtree, then the TEST-exp subtree. If any of the subtree creation fails, the creator will call the type system to select another function (other than IF-TEST-THEN-ELSE) to regenerate a new tree. This is the chronological backtracking procedure described in Section 4.2.

When the evolved program contains a λ abstraction, a λ node is used to indicate that the subtree represents a λ abstraction. For example, Figure 4.3 is the curried format program tree representing program `foldr (λ #1. + 10 #1) 20 [1,2,3]`:

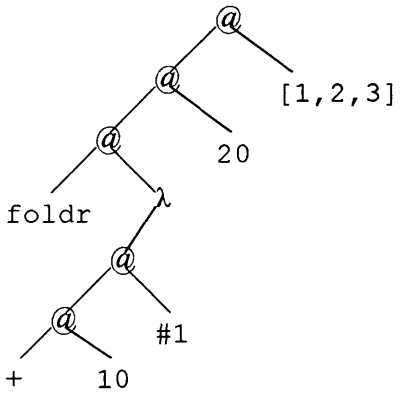


Figure 4.3: Curried format program tree with a λ abstraction.

4.3 Evaluator

The system generates expression-based programs (λ-expressions) which are different from the statement-based programs generated by the standard GP. A statement-based (or procedural) program can contain multiple assignment, such as the SET-V in the iterative summation problem [Koza, 1992, page 470]. Moreover, it can have explicit sequencing, such as the PROG2 in the artificial ant problem [Koza, 1992, page 150]. These two features make statement-based programs unsuitable for the implementation of strongly-typed GP (in the sense of 100% confidence of the type-safe of the evolved programs) due to the following reasons:

- Both assignment and sequencing statements are side-effect operators which do not return any value. This means that the type system has to be extended with a VOID type to support nodes which do not return a value. Consequently, the tree construction process can be effected since special rules must be applied to cater for this additional type.
- The assignment statement causes the search space to be larger than necessary. This is because GET-V (the opposite of SET-V) can legally be applied to a variable that has not yet had a value: this should be detected as an error, yet it is very difficult for the statement-based type system to detect this kind of error. As a result, the search space may

contain invalid program trees.

In contrast, expression-based program trees contain neither assignments nor sequencing statements. Consequently, the type system is less complex (`VOID` type is no longer needed) and more rigorous (more invalid programs are detected).

4.3.1 Program Syntax

The abstract syntax of the expression-based program is described as the following. This is the same as the syntax of untyped λ expressions described in Section 2.3.1.

e ::= c	built-in function or constant
| x	identifier
| e_1 e_2	application of one expression to another
| λ x. e	λ abstraction

Constants and identifiers are provided in the terminal set. Meanwhile, built-in functions are provided in the function set. Application of expressions and λ abstractions are constructed by the creator as described in Section 4.2.

However, this grammar only specifies part of the program syntax generated by the system. In addition, each node in the program tree is annotated with a type. The type syntax will be given later in Section 4.5.

4.3.2 Program Evaluation

A generated program is first converted into a λ abstraction before it can be evaluated. This is done by wrapping the program with λ notation and input variables, which have been made available to the program as members of the terminal set. For example, a program `(+ x y)` is converted into `λx. λy. (+ x y)`. This converted program is then applied to test data, one at a time, to produce outputs. The application of programs to test data is a process of syntax transformation. It involves a sequence of applications of β and δ reduction rules (see Section 2.3.2). These rules are briefly summarized in the following:

- β rule is the function application rule. It produces a new instance of the function body by substituting the arguments in the function with formal parameters. This rule can be expressed using the following notation:

  `(λx.E) M => E [M/x]`

  `E[M/x]` represents an expression `E` with `M` substituted for free occurrences of `x`.

- δ rules are rules associated with each function in the function set. For example, the `IF-TEST-THEN-ELSE` function has one rule to describe how it should be transformed.

The application of these two rules to a program is performed in a *normal order* sequence, i.e. the leftmost outermost expression is evaluated first. Intuitively, this means the body of a function is evaluated first and its arguments are evaluated when necessary. As mentioned in Section 2.3.2, if a program terminates, the order of evaluation should not make any difference; they should reach the same result. However, not all programs terminate. The Church-Rosser Theorem II says that normal order evaluation is the most likely to terminate [Church and Rosser, 1936]. This system therefore performs normal order evaluation rather than any other evaluation order. An example to apply the δ and β rules in normal order sequence is presented in the following:

```
(λ x (+ x (λ #1 (* #1 #1)) 5)) 10
β=> (+ 10 (λ #1 (* #1 #1)) 5)
β=> (+ 10 (* 5 5))
δ=> (+ 10 25)
δ=> 35
```

As mentioned in Section 4.2.1, the evolved programs can contain λ abstractions and a λ abstraction can itself have another λ abstraction inside. The evaluation of λ abstraction is to apply β and δ rules like that of main programs. Since the λ abstractions are created with a terminal set which contains only argument to the λ abstraction (see Section 4.2.1), no global variables would exist in a λ abstraction. Consequently, the name conflict described in Section 2.3.2 would never happen during program evaluation in this system.

4.3.3 Run-Time Error Handling

During programs evaluation, run-time errors may arise. An example run-time error is applying the `head` function to an empty list (the result has an undefined value). A program containing run-time errors may still have useful partial solutions. To allow these partial solutions to be returned for fitness evaluation, a default value is returned when a run-time error is encountered during program evaluation. In this way, the evaluator can continue its evaluation of the program and return with outputs. The fitness of a program is computed based on the correctness of the outputs and the run-time error penalty defined by users (see Section 5.4 for examples). Table 4.1 provides the default values that are used for run-time error handling.

Table 4.1: Defaults for run-time errors in the functional GP system.

Type	Default	Type	Default	Type	Default
int	0	bool	False	string	""

There are many other ways that run-time errors can be handled. Wong and Leung [1996] simply regarded a program with run-time errors as a program which produced wrong outputs. No partial credit nor penalty is considered for this kind of programs. In Section 5.4, a different way to handle a non-termination run-time error will be presented.

Section A.4 presents experiments on using five different methods to handle a division by 0 run-time error. The results show that the five different methods have different impact on GP evolutionary process. Consequently, some of them allow GP to find good and legal solutions faster than the others. It is important to handle constraints with care, or the evolution of problem solutions may be prevented.

4.4 Evolver

After programs are selected for reproduction (see Section 4.6.1), the evolver performs genetic operation (crossover and mutation) on them to generate new programs. A genetic operation can be performed on three different kinds of node: *full application nodes*, *partial application nodes* and λ *modular nodes*. We first describe the method used to select the genetic operation location in program trees. Next, the point typing method is explained. Finally, the genetic operations performed on the three different kinds of node will be discussed.

4.4.1 Selection of Genetic Operation Location

A program tree consists of many nodes. Potentially, a genetic operation can be performed on any of these nodes. This system adopts a selection scheme which biases genetic operations towards the root node. To select a node, a program tree is traversed in a depth-first, right-to-left manner. Moreover, the possibility of the node selection decreases exponentially every level, i.e. the root node has a 50% probability to be selected; the nodes on the next level shares a total of 25% probability to be selected and so on. In this way, the upper portion of the program trees can have more opportunities to be replaced with new nodes.

This location selection scheme is designed to prevent the premature convergence of the root nodes which has been observed during our experiments (`map` program in Section 5.4.2)

and has been reported in [Gathercole and Ross 1996]. In brief, due to the restriction of tree depth, the standard subtree crossover operator is not able to swap all possible pairs of subtrees between two parents and still produce "legal" trees. Instead, most of the genetic exchanges take place near the leaf nodes, with nodes near the root left unchanged. The premature convergence of the root node can severely impair GP performance if the behavior of a program depends highly on the program root node. This is the case with the `map` program. With this biased selection scheme, we are able to evolve the `map` program successfully (see Chapter 5).

Using a similar curried format program representation, [McPhee, Hopper and Reierson, 1998] has reported that because such a representation allows more opportunities for the root node to be modified, the premature convergence of the root node is less likely to happen. However, it did happen in our experiments. The following conjecture may explain these conflicting results:

- The two GP systems use different selection scheme for reproduction. In their system, a tournament selection with a tournament size of 10 is employed. In contrast, ours uses rank selection with exponential fitness normalization (see Section 4.6.1). With this implementation, the programs with high-rank fitness may have much higher probability to be selected for reproduction than those with low-rank fitness. Consequently, premature convergence may occur.

This is an interesting problem and deserves further investigation. We hope to conduct more research on this issue in the near future.

4.4.2 Point Typing Method

The system uses "point-typing" [Koza, 1994, page 532] during genetic operations to preserve program syntax. Initially, a node is selected from the first program tree. Depending on the source of the node (the main program or a λ abstraction), a node with the same source is selected from the second program tree to perform crossover. In other words, λ abstractions can only crossover with the same kind of λ abstractions, i.e. the λ abstraction that representing the same function argument for the same higher-order function. For mutation, a new subtree is generated using the same function and terminal sets as the replaced subtree. The point-typing assures that the produced new program has valid syntax, e.g. no undefined variables.

Point-typing is implemented by using the λ node in the program tree as an indicator of a λ abstraction subtree (see Section 4.2.2). If a λ node is encountered during the traversal of program trees, it indicates that a subsequent selected node is inside a λ abstraction. Otherwise, the selected node is a part of the main program. Since the program traversal process is done

recursively, the flag which specifies the current location of traverse is only set when inside a λ abstraction. It is turned off once the traversal leaves the λ abstraction. Based on the flag, the source of the selected node (main program or λ abstraction) can be easily identified. Another subtree with the same source can be chosen or generated to replace the subtree in the first program.

4.4.3 Genetic Operations

The genetic operation of crossover and mutation are only performed on internal nodes of the trees. This decision is strongly influenced by [Koza, 1992, page 114] who allocated higher probability distribution (90%) to internal nodes during crossover. "This distribution promotes the recombining of larger structure whereas an uniform probability distribution over all points would do an inordinate amount of mere swapping of terminals from tree to tree in a manner more akin to point mutation than to recombining of small structure or building blocks." [Koza, 1992, page 114]. As the ability of subtree crossover on using building blocks to compose problem solutions is discredited (see Section 2.2.2), this implementation will be modified accordingly in the future.

With the curried style program representation, there are three kinds of internal nodes: *full application*, *partial application* and λ *module* nodes. We describe genetic operations performed on these nodes in the following subsections.

Full Application Nodes

A full application node is annotated with a primitive type, i.e. a non-function type, to indicate that the function has all its arguments provided. For example, `int` and `[int]` are primitive types. Primitive types include "bracketed function type" (see Section 4.5) which indicates that the return value of a function is another function. A full application node can only crossover with another full application node with the same primitive type. Meanwhile, the mutation operator has to create a new subtree returning the same primitive type to replace the subtree rooted with the selected node. These two genetic operations can be applied to nodes in either the main program or a λ abstraction.

The mutation operator uses the same procedures that generate the main program to generate a new subtree (see Section 4.2). However, depending on the source of the mutation node, different terminal set is used. If the mutation node is inside a λ abstraction, the terminal set contains arguments to the λ abstraction. If the mutation node is in the main program, the same terminal set as that used to generate the main program is used. The function set that is used to

generate the main program is always used by the mutation operator to generate the new subtree.

Partial Application Nodes

A partial application node is annotated with a function type. The syntax of function type is described in Section 4.5. For example, `int->int` and `bool->int->int` are function types. A partial application node can only crossover with another partial application node with the same function type, i.e. node of type `int->int` crossover with another node of type `int->int`. Moreover, the mutation operator has to create a new subtree with the same function type to replace the subtree rooted with the selected node. These two operators can be applied to nodes in either the main program or a λ abstraction.

Since the creator (Section 4.2) is capable of generating program trees of any types, including function type, the operation of mutation on partial application nodes is identical to that on full application nodes.

Lambda Modular Nodes

Crossover on the λ nodes (see Figure 4.3) has the effect of swapping the definition of a λ abstraction definition in one program with a λ abstraction definition in another program, i.e. the whole of the λ abstraction in the first program is replaced by the whole of the λ abstraction in the second program. We call this "λ modular crossover". Similarly, λ modular mutation replaces a λ abstraction definition with a newly created λ abstraction definition. The creation of a λ abstraction is as described in Section 4.2.1.

In program trees, λ abstractions are viewed as structured building blocks. Genetic operations are therefore allowed to be performed on them like other program segments. The λ modular crossover is similar to the modular crossover in [Kinnear, Jr., 1994a]. However, with the benefit of the type system, the argument-mismatching mentioned in his work does not happen. Each λ node is annotated with a type (see Section 4.5) which indicates the number and type of argument defined in the λ abstraction. The type system assures that λ modular crossover is only performed between two λ abstractions definitions with the same number and type of arguments.

4.5 Type System

The functional GP system employs a type system to perform type checking so that invalid program trees are never created. Initially, the user has to specify the type of each function and

terminal. This type information is used by the type system and the creator to generate a population of type-correct programs (see Section 4.2). Each of the generated program trees has its nodes annotated with a type. During program evolution, this type information is used by the type system and the evolver to perform genetic operations. In this way, only type-correct programs are generated.

Type Syntax

The type syntax used to specify the of types of functions and terminals is as the following:

σ ::= τ	built-in type
| υ	type variable
| σ_1 -> σ_2	function type
| [σ]	list of elements all of type σ
| (σ_1 -> σ_2)	bracketed function type

τ ::= `int` | `string` | `bool` | $\texttt{generic}_i$

υ ::= $\texttt{dummy}_i$ | $\texttt{temporary}_i$

Each node in the tree is also annotated with a type specified in the above type syntax. A full application node is specified by a primitive type such as `int`. On the other hand, a partial application node is indicated by a function type σ_1->σ_2. To specify a higher-order function type, the bracketed function type is used. In particular, the function arguments and function values returned by a higher-order function are expressed using a bracketed function type. The usage of the three different kinds of type variables and their instantiation will be described in Section 4.5.1.

Annotating Expressions with Types

Every expression in the language (Section 4.3.1) is annotated with a type:

- Constants such as `0` and identifiers such as `x` have a type specified by the user;
- Functions also have pre-defined types. For example, the function `head` has the type `[a]->a`, where `a` is a dummy type variable;
- Applications of expressions have a type given as follows:

  if e_1 has type σ_1 -> σ_2 and e_2 has type σ_1

  then the application of e_1 to e_2, represented as e_1 e_2, has type σ_2 else there is a type error;

- λ abstractions have the following type:

  if x has type σ_1 and e has type σ_2

  then λ x. e has type σ_1 -> σ_2.

With the additional type annotation, the evolved programs are essentially the polymorphic λ calculus described in Section 2.3.5.

4.5.1 Type Variables and Instantiation

The system supports three kinds of type variables: *generic*, *dummy* and *temporary*. Their usages and their instantiation are described as follow.

Generic Type Variables

The generic type variables are used to indicate that the evolved programs are polymorphic, i.e. they can accept inputs or produce outputs of more than one type. For example, the `length` program (see Section 4.7) has type `[G1]->int`, where `G1` is a generic type variable. This specifies that the program takes as input a list of any type and returns an integer value. While the program is being evolved, this kind of type variable must not be instantiated: it therefore takes on the role of a built-in type.

Dummy and Temporary Type Variables

Dummy type variables are used to express polymorphism of functions in the function set and terminals in the terminal set. Table 4.2 gives some examples of functions and terminals with their type information. The `a` and `b` are dummy type variables; `bool` is a built-in type; `(a->b)` is a bracketed function type.

Table 4.2: Examples of functions and terminals with type information

Name	Type
if-test-then-else	`bool->a->a->a`
head	`[a]->a`
map	`(a->b)->[a]->[b]`
==	`a->a->bool`

When a polymorphic function or terminal is selected to construct a program tree node, its dummy type variables are instantiated to some other type values (and the type must not involve a dummy type, but it may be a generic or temporary type). Note that if a dummy type

variable occurs more than once in the selected function, the dummy type variable has to be instantiated to the same type. This is done through the process of *contextual instantiation* which will be discussed in Section 4.5.3. Table 4.3 shows how the dummy type variables are instantiated.

Table 4.3: Examples of dummy type variables instantiation

Required Type	Selected Function	Instantiated Type
[G1]	if-test-then-else	`bool -> [G1] -> [G1] -> [G1]`
[int]	map	`(T1->int) -> [T1] -> [int]`
bool	==	`T2 -> T2 -> bool`

Typically, the constraints imposed by the return type of the function will allow the dummy type to be instantiated to a known type. For example, the return type variable `a` of `if-test-then-else` function is instantiated to `[G1]`. This enforces the types of the second and the third argument to be of type `[G1]`. However, there are also situations when such constraints do not exist. For example, the dummy type variable `a` in the function `map` has no such constraint. In this case, the dummy type is instantiated to a new temporary type variable (`T1` in Table 4.3). A program tree may contain node which has a temporary type. However, dummy types would never exist in the program tree nodes.

Temporary type variables may become instantiated to other types at a later time during the growing of the program or during crossover or mutation (see Section 4.7 for examples). This delayed binding of temporary type variables provides greater flexibility and generality; essentially it supports a form of polymorphism within the program tree as it is being evolved.

Within a program tree, temporary type variables must be instantiated consistently to maintain the legality of the program. A global type environment is maintained for each program tree during the creation and evolution of the program. This environment records how each temporary type variable is instantiated. Once a temporary type variable is instantiated, all occurrences of the same variable in the same program are instantiated to the same type.

4.5.2 Unification Algorithm

The creation of type-correct programs is essentially a sequence of selecting type-matched functions and terminals to construct the programs (see Section 4.2). This important process of determining whether a type satisfies the required type is carried out by Robinson's unification algorithm [Robinson, 1965]. Briefly, the unification algorithm takes inputs of two types. In

this application, one type is the required node type and the other is the return type of a selected function or the type of a selected terminal. The algorithm determines whether they can "unify" with each other (unify will be explained in the next paragraph). If the two types unify, the algorithm returns their "most general unifier" (also explained in the next paragraph), otherwise it flags an error.

In this application, if the return type of the selected function unifies with the required node type, this function is used to construct the node. Otherwise, another function will be selected and type checked. Using the unification algorithm, functions and terminals can be randomly selected and typed checked as the tree grows. No preprocess (such as the generation of type possibilities table in STGP) is required. The linear-time implementation of this algorithm has been devised by [Paterson and Wegman, 1978]. However, the type system has not incorporated such an implementation.

A few terminologies need to be explained to understand the unification algorithm. They are listed in the following:

- A *substitution*, θ, is a finite set (possible empty) of pairs of the form (X_i, t_i) where X_i is a type variable and t_i is a type value or a type variable. In this application, the type variable X_i is either a dummy type or a temporary type while the type value t_i can be of any type specified in the type syntax (Section 4.5) except dummy type. For example: θ = {(a, int), (b, T2)}, where a and b are dummy types and T2 is a temporary type.
- The result of applying a substitution θ to a type A, denoted by Aθ, is the type obtained by replacing every occurrence of X_i in A by t_i, for each pair (X_i, t_i) in θ. For example: a -> a -> b {(a,int), (b,T2)} = int -> int -> T2. This process is called *contextual instantiation* described in Section 4.5.3.
- Two types A and B *unify* if there exists a substitution θ which makes A = B. For example, if A = T1->int and B = [string]->T2, A and B unify with θ={(T1, [string]), (T2, int)}. Note that θ can be empty. This is the case when A and B are identical.
- There may be more than one substitution which unifies two types. The *most general unifier* of two types A and B is a substitution θ that unifies A and B such that Aθ is more general than any other common instance of A and B. For example, if $A\theta_1$ = T1 -> int and $A\theta_2$ = int -> int, then $A\theta_1$ is more general than $A\theta_2$ and θ_1 is the most general unifier.

The following gives some examples which apply the unification algorithm, called *unify*, to different types:

```
unify(int->int, int->int)=(True,{})
unify(int, bool)=(False,{})
unify(int->int->T1, a->a->b)=(True,{(a,int),(b,T1)})
```

4.5.3 Contextual Instantiation

Type expressions which contain several occurrences of the same type variable, like in `a->a`, express contextual dependencies [Cardelli, 1987]. Whenever an occurrence of a type variable is instantiated, all the other occurrences of the same type variable must be instantiated to the same type value. This is done through the process of *contextual instantiation*: applying a substitution, which contains the instantiation of type variables, to the type expression. An example of it operations has been given in Section 4.5.2.

The process of contextual instantiation is applied in two places by the type system:

- During the selection of functions and terminals for tree construction. In this instance, all dummy type variables in the selected function or terminal have to be instantiated and bound to type values. Consequently, program trees would never contain a node with a dummy type.
- During the growing of a program tree. In this instance, a temporary type in a tree node may become bounded to another type value. Once this happens, all occurrences of this temporary type variable in the same program tree are instantiated and bound to the same type value.

Using the contextual instantiation method to instantiate type variables dynamically, the type system allows the program trees to be generated by random selection of type-matched functions and terminals.

4.6 Implementation

The functional GP system is implemented with two different genetic algorithms: generational replacement using fitness-proportionate selection and steady-state replacement [Syswerda, 1989; Reynolds, 1993] with rank selection. These two algorithms are described in Section 4.6.1. In Section 4.6.2, the implementation programming language Haskell and the advantages/disadvantages the language brings to the GP system are discussed.

4.6.1 Genetic Algorithms

Generational Replacement

Generational replacement is the traditional GP implementations: a population with a specified size is first created; new programs are generated from programs in the current generation to compose a new generation. The process of program evolution from generation to generation terminates when a satisfactory program is found or the maximum number of generations has reached.

The selection of programs for reproduction is based on the fitness-proportionate selection method. A random fitness value *f* is first generated (0.0 <= *f* <= total fitness of the population). The population is then enumerated, one at a time in the order that the program is generated, to accumulate their fitness value. This process stops when the accumulated value is equal or greater than the random fitness value *f*. The last program visited is selected as a parent to generate offspring to compose the next generation. This algorithm allows that the expected number of children a program has in the next generation approximately equals the ratio of its fitness to that of the average fitness in the population. The higher a program's fitness, the better chance it has to be selected for reproduction.

Steady-State Replacement

Initially, a population with a specified size is created. Within the population, every tree is unique. During program evolution, a newly generated program is checked for uniqueness before it is used to replace the programs with the lowest fitness score in the same population pool. As a result, the size of the population remains constant.

The advantage of steady-state replacement, compared with generational replacement, is that a program with a good fitness score is immediately available as a parent for reproduction rather than having to wait until the next generation. However, such an aggressive evolutionary approach may cause premature convergence to local optimum; the search for the global optima may never be achieved [Syswerda, 1991; Fogel and Fogel, 1995].

The selection scheme used with this GA is a rank selection combined with exponential fitness normalization [Cox, Davis and Qiu, 1991]. This means:

- The probability that a program is selected depends on its relative rank in the population;
- The probability of selecting the *n*-th best program is *parent-scalar* times the probability of selecting the (*n*-1)-th best individual. The *parent-scalar* is a parameter, valued between 0 and 1, provided by the user.

4.6.2 Programming Language

The system is implemented in Haskell 1.4 [Peterson and Hammond 1997] using the Glasgow Haskell Compiler version 2.02. Haskell is a *non-strict purely* functional programming language. Non-strict languages evaluate a program expression only when its value is needed; this is commonly referred as lazy evaluation. This is an advantage to the GP system because program trees normally contain "redundant expressions" (introns). Without the need to evaluate these redundant expressions, the GP system may take less time to evolve problem solution. The "purity" feature means no side-effect operation is allowed in the language. Consequently, each time a member of the population is updated, a new copy of the population may be made. Fortunately, the Glasgow Haskell compiler provided an updatable data structure (array) which can be used to implement GP population. This data structure supports effective methods in accessing and modifying the members of the population. Nevertheless, compared with other GP systems which are implemented in imperative languages, such as C and C++, the functional GP system is still slow.

Haskell is also a "typeful" programming language [Hudak, Peterson and Fasel, 1997]. The implementation of the GP system has benefited from Haskell's rich type system in the following ways:

- By declaring a user-defined type to be a derived type of an existing type class supported by the Haskell language, the code associated with the type class will be automatically generated for the user-defined type. For example, by specifying that `TypeExp` is derived from `Eq` and `Text` type classes, the "==" and "print" functions for `TypeExp` are automatically generated. The following shows the declaration of the `TypeExp` to be a derived type for `Eq` and `Text` type classes.

```
data TypeExp = IntNum | Boolean
               derived (Eq, Text)
```

- Haskell's pattern matching mechanism for type constructors (such as `InNum`, `Boolean` above) has made the implementation of functions easier. For example, the function `applySub`, which performs different operation based on the type construct of the argument, can be written as the following:

```
applySub typeExp = Case typeExp of {
                       IntNum -> ....;
                       Boolean -> ....; }
```

4.7 An Example

This section presents a worked example to demonstrate the operation of the functional GP system. Section 4.7.1 presents the creation of a single program. Section 4.7.2, 4.7.3 and 4.7.4 explain the genetic operations.

Problem Description: The objective is to evolve the `length` program, which takes a list of items of any type, and returns the number of items in the list. Note that the input has to be a finite list. For example, `length[1,9,3]=3; length['a','b']=2.`

Input Type: The input L has a generic type `[G1]`.

Output Type: The output has type `int`.

Terminal Set: {`L:: [G1]; zero:: int`} (where `L::[G1]` reads "L has type [G1]")

Function Set: {`head:: [a] -> a; tail:: [a] -> [a];`

`foldr:: (a -> b -> b) -> b -> [a] -> b; add1:: int -> int`}

Maximum Tree Depth: 3.

Maximum λ Abstraction Tree Depth: 3.

4.7.1 Program Creation

The following presentation is organized by headings which indicate the new technique used in creating the program tree. The generated program is presented in `courier` font with the type of each program construct (function or terminal) in superscript. Meanwhile, at each stage, the node that will be expanded in the next stage is underlined.

Contextual instantiation of dummy type variables: Based on the required return type, `int`, the type system randomly selects a function to construct the root node. For example, let the function be `foldr`. The contextual instantiation process takes place to instantiate dummy type variable `b` to `int`. The unconstrained dummy type variable `a` is instantiated to a new temporary type variable `T1`:

$$((\texttt{foldr}^{(\texttt{T1->int->int})\texttt{->int->[T1]->int}}\ \texttt{ARG1}^{(\texttt{T1->int->int})})^{\texttt{int->[T1]->int}}\ \texttt{ARG2}^{\texttt{int}})^{\texttt{[T1]->int}}\ \underline{\texttt{ARG3}}^{\texttt{[T1]}})^{\texttt{int}}$$

Depth-first-right-first tree expansion: The right branch of the program, `ARG3`, is first expanded. The type system selects the `tail` function:

$$((\texttt{foldr}^{(\texttt{T1->int->int})\texttt{->int->[T1]->int}}\ \texttt{ARG1}^{(\texttt{T1->int->int})})^{\texttt{int->[T1]->int}}\ \texttt{ARG2}^{\texttt{int}})^{\texttt{[T1]->int}}\ (\texttt{tail}^{\texttt{[T1]->[T1]}}\ \underline{\texttt{ARG4}}^{\texttt{[T1]}})^{\texttt{[T1]}})^{\texttt{int}}$$

Contextual instantiation of temporary type variables: The depth-first-right-first approach

makes ARG4 the next node to be expanded. Since the maximum tree depth is reached, the type system selects terminal L to construct a leaf node. After the temporary type variable T1 is instantiated to G1, the contextual instantiation process propagates this effect throughout the whole program:

$$(((\texttt{foldr}^{(G1\text{->}int\text{->}int)\text{->}int\text{->}[G1]\text{->}int}\ \underline{\texttt{ARG1}}^{(G1\text{->}int\text{->}int)})^{int\text{->}[G1]\text{->}int}\ \underline{\texttt{ARG2}}^{int})^{[G1]\text{->}int}\ (\texttt{tail}^{[G1]\text{->}[G1]}\ \texttt{L}^{[G1]})^{[G1]})^{int}$$

The next node to be expanded is ARG2 whose type is int. The function head is selected to construct this node. The dummy type variable a is instantiated to int to match the type of ARG2:

$$(((\texttt{foldr}^{(G1\text{->}int\text{->}int)\text{->}int\text{->}[G1]\text{->}int}\ \underline{\texttt{ARG1}}^{(G1\text{->}int\text{->}int)})^{int\text{->}[G1]\text{->}int}\ (\texttt{head}^{[int]\text{->}int}\ \underline{\texttt{ARG5}}^{[int]})^{int})^{[G1]\text{->}int}\ (\texttt{tail}^{[G1]\text{->}[G1]}\ \texttt{L}^{[G1]})^{[G1]})^{int}$$

Backtracking: The next node to be expanded is ARG5 which has to be a leaf node since the maximum tree depth is reached. Unfortunately, there is no terminal whose type unifies with the required type of [int]. The system backtracks and selects the function add1 to replace the head function:

$$(((\texttt{foldr}^{(G1\text{->}int\text{->}int)\text{->}int\text{->}[G1]\text{->}int}\ \underline{\texttt{ARG1}}^{(G1\text{->}int\text{->}int)})^{int\text{->}[G1]\text{->}int}\ (\texttt{add1}^{int\text{->}int}\ \underline{\texttt{ARG6}}^{int})^{int})^{[G1]\text{->}int}\ (\texttt{tail}^{[G1]\text{->}[G1]}\ \texttt{L}^{[G1]})^{[G1]})^{int}$$

The next node to be expanded is ARG6 which has to be a leaf node since the maximum tree depth is reached. The only terminal satisfying the required type of int is zero:

$$(((\texttt{foldr}^{(G1\text{->}int\text{->}int)\text{->}int\text{->}[G1]\text{->}int}\ \underline{\texttt{ARG1}}^{(G1\text{->}int\text{->}int)})^{int\text{->}[G1]\text{->}int}\ (\texttt{add1}^{int\text{->}int}\ \texttt{zero}^{int})^{int})^{[G1]\text{->}int}\ (\texttt{tail}^{[G1]\text{->}[G1]}\ \texttt{L}^{[G1]})^{[G1]})^{int}$$

λ Abstractions creation: The next node to be expanded is ARG1 which has a function type (G1->int->int). This specifies that this is a function which takes first argument of type G1 and second argument of type int. It returns a value of type int. This function argument is created as a λ abstraction. The procedure of λ abstraction creation is similar to that of the main program. The only difference is that it uses a different terminal set which contains only its arguments, i.e. #1:: G1; #2:: int. We skip the intermediate steps and present the final program which has its λ abstraction generated:

$$(((\texttt{foldr}^{(G1\text{->}int\text{->}int)\text{->}int\text{->}[G1]\text{->}int}\ (\lambda\#1^{G1}\ (\lambda\#2^{int}\ (\texttt{add1}^{int\text{->}int}\ \#2^{int})^{int})^{int\text{->}int})^{G1\text{->}int\text{->}int})^{int\text{->}[G1]\text{->}int}\ (\texttt{add1}^{int\text{->}int}\ \texttt{zero}^{int})^{int})\ ([G1]\text{->}int\ (\texttt{tail}^{[G1]\text{->}[G1]}\ \texttt{L}^{[G1]})^{[G1]})^{int}$$

This generated program is not only type-correct but also general since its argument L can be a list of any type of values.

4.7.2 Full Application Node Crossover

The following two programs are used as parents to demonstrate the crossover operation:

(((foldr ARG1)ARG2$^{\underline{int}}$)ARG3)

(head L)$^{\underline{T1}}$

The crossover operation is performed on nodes ARG2 and (head L). The underlined type T1 and int unify with each other. As a result, the temporary type variable T1 is instantiated to int. The crossover operator generates the following program:

(((foldr ARG1)(head L)^{int})ARG3)

4.7.3 Partial Application Node Mutation

The following is the parent program for the mutation operation:

(((foldr(λ#1(λ#2((* 10)$^{\underline{int->int}}$ (add1 #1)))))ARG2)ARG3)

The Creator (see Section 4.2) generates a subtree whose return type is the underlined type int->int:

(- (add1 #2))$^{\underline{int->int}}$

The new tree is used to replace the mutation node subtree and a new program is generated:

(((foldr(λ#1(λ#2((- (add1 #2))(add1 #1)))))ARG2)ARG3)

4.7.4 Lambda Modular Crossover

The first parent program is:

(((foldr(λ#1(λ#2 (add1 (+ #1 #2))))$^{\underline{(int->int->int)}}$)ARG2)ARG3)

The second parent program is:

(add1(((foldr(λ#1(λ#2 (add1 #1)))$^{\underline{(int->int->int)}}$)ARG2)ARG3))

The underlined type in the two parent programs unify with each other. The λ module cross-over replaces the λ abstraction in the first parent program by the λ abstraction in the second parent program and produce the following new program:

```
(((foldr(λ#1(λ#2 (add1 #1)))(int->int->int))ARG2)ARG3)
```

4.8 Summary

This chapter presents a functional GP system which (1) evolves programs based on the syntax of the λ *calculus*; (2) supports module creation through the use of λ *abstractions*, which can be reused through higher-order functions; and (3) performs type checking for the programs using a *polymorphic type system*. In addition, each component of the system and its operation are described in detail. This working system shows that these functional techniques are indeed applicable to GP. In the following three chapters, the impact of those functional techniques on GP will be further analyzed.

Chapter 5 first presents the concept of types and their implementation in GP. Advantages and drawbacks of different type checking approaches are compared. This is followed by two worked examples to demonstrate that polymorphism can enhance GP applicability to problems which require multiple types, type variables or function types. Chapter 6 provides a case of successful usage of λ abstractions and implicit recursion to provide module creation and reuse for GP. This approach has enabled GP to evolve a general solution for the `even-parity` problem very efficiently. Chapter 7 studies the program representation used in Chapter 6. A detailed analysis of the search space is conducted to provide guidelines for the application of this program representation to other problems.

Chapter 5

Polymorphism and Genetic Programming

This chapter presents the application of polymorphism in GP. Firstly, the concept of types in GP is presented by defining and differentiating *untyped*, *dynamically typed* and *strongly typed* GP. Secondly, the two different implementations of strongly typed GP, *monomorphic* GP and *polymorphic* GP, are introduced. In particular, we analyze the impacts of the three different type variables used in polymorphic GP. Finally, the experiments to generate two polymorphic programs: `nth` and `map` are presented. These two programs are chosen to demonstrate the power of polymorphism as they contain features which make them very difficult (if not impossible) for an untyped, dynamically typed or monomorphic typed GP to generate.

5.1 Types and Genetic Programming

The road from untyped to typed GP is motivated by two reasons. Firstly, to enhance the applicability of GP by removing the closure requirement. Secondly, to assist GP searching for problem solutions using type information. Koza made the first attempt to introduce types to GP by extending GP with "constrained syntactic structures" when he realized that not all problems have solutions which can be represented in ways that satisfy the closure requirement [Koza, 1992]. Supporters of this argument [Montana 1995; Haynes, Wainwright, Sen and Schoenefeld, 1995; Haynes, Schoenefeld and Wainwright, 1996; Clack and Yu, 1997] believe that it is important for GP to be able to handle multiple types and advocate excluding type-incorrect programs from the search space to speed up GP search. Another route to promote the use of types in GP is based on the idea that types provide inductive bias to direct GP learning. For example, [Wong and Leung, 1995] included type information in a logic grammar to bias the selection of genetic operation location during program evolution. McPhee, Hopper and Reierson [1998] also demonstrated that program representation which utilizes "function type" can bias recombination operations and benefit GP search on some problems.

These two paths, although with different purposes, are actually interrelated. Types exist in the real world naturally [Cardelli and Wegner, 1985]; by allowing problems to be represented in their natural ways, an inductive bias is established which selects solutions based on criteria that reflect experience with similar problems.

In its traditional style, the GP paradigm is not capable of distinguishing different types: the term *untyped* is used to refer to such a system. In the case when the programs manipulate multiple data types and contain functions designed to operate on particular data types, untyped GP leads to an unnecessary large search space, which contains both type-correct and type-incorrect programs. To enforce type constraints, two approaches can be used: *dynamically typed* GP and *strongly typed* GP. In dynamically typed GP, type checking is performed at program *evaluation* time. In contrast, strongly typed GP performs type checking at program *generation* time. The computation effort required for these two different type checking approaches is implementation dependent. It is not valid to claim that dynamic typing is more efficient than strong typing; nor the other way around. The details of these two type checking approaches are discussed in the following sections.

5.2 Dynamically Typed GP

Dynamic typing is a hard constraint method (see Appendix A): type-incorrect programs are not allowed to exist in the solution space; although the search space may contain illegal programs (see Figure 5.1).

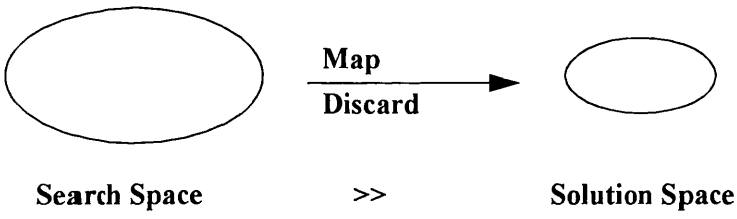


Figure 5.1: Search space versus solution space in dynamically typed GP.

The transformation of a type-incorrect into a type-correct program can be implemented using a "legal map" method (see Appendix A). For example, a value with an illegal type of "real" can be mapped into a value with legal type "integer". However, for more complex types such as list or matrix, a proper mapping scheme can be difficult to design. Montana implemented this dynamic typing approach in one of his experiments. When trying to add a 3-vector with a 4×2 matrix, the matrix is considered as an 8-vector, which is converted into a 3-vector by throwing away the last 5 entries. "The problem with such unnatural operations is that, while

they may succeed in finding a solution for a particular set of data, they are unlikely to be part of a symbolic expression that can generalize to new data." [Montana, 1995].

Another way to implement dynamic typing is to discard the type-incorrect program (see Figure 5.1). "The problem with this approach is that it can be terribly inefficient, spending most of its time evaluating programs that turn out to be illegal." [Montana, 1995]. In his experiments, Montana reported that within 50,000 programs in the initial population, only 20 are type-correct. Because of these issues, dynamic typing is not an ideal way to implement type checking in GP.

5.3 Strongly Typed GP

Similar to dynamic typing, strong typing is a hard constraint method. However, in addition to the solution space, the search space is restricted to contain only type-correct programs (see Figure 5.2). This is done through the "legal seeding" method (the initial population is seeded with solutions that do not conflict with the type constraints) and the "legal birth" method (crossover and mutation operators are designed such that they cannot generate illegal offspring) (see Appendix A).

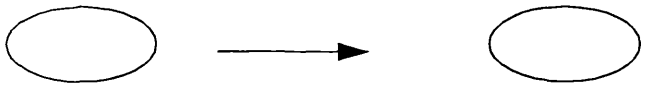


Figure 5.2: Search space versus solution space in strongly typed GP.

There are two approaches in implementing strong typing in GP: *monomorphic GP* and *polymorphic GP*. Monomorphic GP uses monomorphic functions and terminals to generate monomorphic programs. In contrast, polymorphic GP can generate polymorphic programs using polymorphic functions and terminals. In the first instance, inputs and outputs of a program need to be of the specified types. In the latter case, programs can accept inputs and produce outputs of more than one type. Polymorphic GP therefore can potentially generate more general solutions than those produced by monomorphic GP.

5.3.1 Generality and Polymorphism

The generality of polymorphic GP is achieved through the use of three different type variables (see Section 4.5.1). On the surface level, it seems that polymorphic GP is simply monomor-

phic GP plus type variables. Indeed, when functions and terminals are defined without any type variables, polymorphic GP becomes monomorphic GP. However, the use of these type variables can have impacts on the GP search space that is not obvious to the GP users. The purpose of this analysis is not to claim that the smaller search space produced by one particular GP system will make the solving of a problem easier. Instead, we identify such impacts so that users can be more careful when using these type variables to define functions and terminals. For some problems, polymorphic GP is necessary, e.g. `nth` and `map` programs (see Section 5.4). For others, monomorphic GP is more appropriate, e.g. the general `even-parity` problem (see Chapter 6 & 7). Misuse of these type variables can cause unnecessary overhead to the GP system. With the provided information, users can select polymorphism or monomorphism by incorporating or not incorporating type variables to suit their target problems.

Generic Type Variables

Generic type variables are used to specify that the inputs and/or outputs of the target programs have generic type. Similar to built-in types, generic types are not allowed to be instantiated to other types during program evolution. Consequently, generic type variables do not cause polymorphic GP to generate search space that is any different from that produced by monomorphic GP. However, by supporting generic types, polymorphic GP can generate generic programs. This is the most important advantage that polymorphic GP provides which is not available in monomorphic GP.

This advantage, however, can not be achieved by using generic type variables alone. To allow the evolved program to accept inputs of any type, the functions and terminals used to construct the program trees have to be able to handle arguments of multiple types, i.e. specified with dummy types. It is the combination usage of generic types and dummy types that enables polymorphic GP to evolve generic programs.

Dummy Type Variables

Dummy type variables allow a single function to represent multiple functions operating in the same way for different types of arguments. For example, the function `head` takes a list as input and returns the first element of the list as output. With polymorphic GP, one function with the type of `[a]->a` is sufficient to represent this functionality (where `a` is a dummy type). In contrast, with monomorphic GP, multiple functions, one for each type, have to be defined. For example, one for input of integer list, one for input of Boolean list and so on. Consequently, the function set in monomorphic GP can contain more entries than that in polymorphic GP even though the functionality provided by the two function sets are identical. The

same effect also applies to the terminal set.

With a smaller function set to represent the same problem, it is expected that the search space (which consists of programs that are composed using available functions and terminals) using polymorphic GP is smaller than that using monomorphic GP. Yet, this is not true. Since a dummy type variable can be instantiated to any type, a polymorphic function can be used in many different ways to construct GP program tree nodes. As a trivial example, assume the program to be evolved takes two inputs: a list of integer and a list of character. It returns True if the two lists have the same length and False otherwise. With a polymorphic GP system, the function `tail` is defined with type `[a]->[a]`. This function can be used to construct a program node which requires either integer list type or character list type. i.e. it represents both the `tail-int-list` and `tail-char-list` functions defined in a monomorphic GP system. In fact, the function `tail` represents more than just two functions since its dummy type can be instantiate to any type. Consequently, when solving the same problem, using polymorphic GP may generate not smaller but larger search space than that produced by monomorphic GP.

Because a single function can be used to represent multiple functions that perform the same operation, polymorphic GP requires less user effort to implement the function set compared to that required by a monomorphic GP system. Moreover, polymorphic GP allows functions or terminals with unknown types to be represented using dummy type variables. These are advantages that a polymorphic GP system has over a monomorphic GP system.

Temporary Type Variables

Temporary type variables can only be used in conjunction with the dummy type variables. A dummy type variable is instantiated to a temporary type when its type can not be decided. Temporary type variables may or may not be instantiated to other types later during the growth of the program trees (see Section 4.5.1). Consequently, a generated program may contain a node whose type is a temporary type variable.

The use of temporary types in polymorphic GP may potentially generate a larger search space than that produced by a monomorphic GP system (providing that both function sets are of the same functionality). The reason is that a temporary type is allowed to be instantiated to *any* type. For example, the polymorphic function `length` has type `[a]->int`. Its equivalencies in the monomorphic GP may be two monomorphic functions `length-int-list` and `length-char-list`. When the `length` function is selected to construct a program node, its dummy type a is instantiated to a temporary type, which can be instantiated to any type at a later time. Compared with monomorphic GP which only has the same function

defined for two types (`int` and `char`), the search space produced by polymorphic GP may be larger because the length function represents more functions than that as defined in the monomorphic GP.

5.3.2 Polymorphism in STGP

Generic functions in Montana's STGP provide a form of parametric polymorphism (see Section 3.2.1). Generic functions are parameterized templates that have to be instantiated with actual values before they can be used to construct tree nodes. The parameters can be type parameters, function parameters or value parameters. Generic functions with type parameters are polymorphic since the parameters can be instantiated with many different type values.

In STGP, the function set may contain generic functions. To be used in a program tree, a generic function has to be instantiated by specifying the argument and return types of the generic function. Instantiating a generic function can be viewed as making a new copy of the generic function with specified argument and return types. Instantiated generic functions are therefore monomorphic functions.

The implementation of polymorphism in STGP is different from that of the functional GP in the following areas:

- it requires the generation of a type possibilities table (see Section 3.2.1).
- it uses a table-lookup mechanism to instantiate type variables (see Section 3.2.1).
- it does not provide a systematic way to support function types (see Section 3.2.1).
- it does not uses *temporary type variables* to support polymorphism within a program tree during program evolution.

5.4 Experiments

The problems of evolving the `nth` and the `map` programs are chosen to demonstrate the power of polymorphism in GP. These two programs have the following characteristics that make them very difficult for a GP system without polymorphism to evolve:

- Both `nth` and `map` programs are required to manipulate multiple types. In the `nth` program, the two inputs are of different types (an integer and a list) while the `map` program takes two inputs of function type and list type respectively.
- Both `nth` and `map` are polymorphic programs. The type of the two programs are as the following (where `G1` and `G2` are generic types):

```
nth :: int->[G1]->G1
map :: (G1->G2)->[G1]->[G2]
```

- The `map` program requires the manipulation of function types. There has not been a GP system which can systematically handle function types up to now.

The ability of polymorphic GP to generate these two programs demonstrates that polymorphism is, indeed, applicable to GP. Moreover, polymorphism also enhances the applicability of GP to problems that are very difficult for the standard GP to solve.

Both `nth` and `map` are also recursive programs. To allow GP to evolve recursive programs, a simple method similar to [Brave, 1996] is used. In this method, the name of the program is included in the function set so that it can be used to construct program trees; hence making recursive calls. The results of this implementation of recursion will be analyzed in Section 5.4.3.

5.4.1 The Nth Program

Problem Description: The `nth` program takes two arguments, an integer `N` and a list `L`. It returns the `N`-th element of `L`. If the value of `N` is less than 1, it returns the first element of `L`. If the value of `N` is greater than the length of `L`, it returns the last element of `L`.

Input Types: The input `N` has type `int` and the input `L` has type `[G1]`.

Output Type: The output has type `G1`.

Terminal Set: `{L:: [G1], N:: int, one:: int}`

Function Set:
```
{head:: [a]->a, if-then-else:: bool->a->a->a,
 tail:: [a]->[a], less-eq:: int->int->bool,
 gtr:: int->int->bool, length:: [a]->int,
 minus:: int->int->int, nth:: int->[G1]->G1}
```

Genetic Parameters: The experiments were carried out using a steady-state replacement method described in Section 4.6.1. The population size is 3,000; parent scalar value (see Section 4.6.1) is 0.9965; maximum tree depth is 5; and crossover rate is 100%. Each run terminates when a correct `nth` program is found or 33,000 programs have been generated (3,000 by Creator and 30,000 by Evolvor).

Test Cases

Twelve test cases were used to evaluate the generated programs. Each test case gave `N` a different value from 0 to 11. The value `L`, however, is the same for all 12 test cases; it is a list

containing the characters `a` to `j`. Table 5.1 lists the 12 test cases and their expected outputs.

Table 5.1: The 12 test cases for evolving `nth` program.

Case No.	Value of N	Value of L	Expected Output
1	0	[a,b,c,d,e,f,g,h,i,j]	a
2	1	[a,b,c,d,e,f,g,h,i,j]	a
3	2	[a,b,c,d,e,f,g,h,i,j]	b
4	3	[a,b,c,d,e,f,g,h,i,j]	c
5	4	[a,b,c,d,e,f,g,h,i,j]	d
6	5	[a,b,c,d,e,f,g,h,i,j]	e
7	6	[a,b,c,d,e,f,g,h,i,j]	f
8	7	[a,b,c,d,e,f,g,h,i,j]	g
9	8	[a,b,c,d,e,f,g,h,i,j]	h
10	9	[a,b,c,d,e,f,g,h,i,j]	i
11	10	[a,b,c,d,e,f,g,h,i,j]	j
12	11	[a,b,c,d,e,f,g,h,i,j]	j

Fitness Function

A program tree is evaluated with each of the 12 test cases, one at a time. The produced output is compared with the expected output to compute the fitness value for each test case according to equation 7:

$$Fitness = 10 \cdot 2^{-d} - (10 \cdot rtError) - (10 \cdot reError) \qquad (7)$$

where d is the distance between the position of the expected value and the position of the value returned by the generated program; *rtError* is 1 if there is a run-time empty-list error (explained later); and *reError* is 1 if there is a run-time non-terminating recursion error (explained later). In this fitness function, the value 10 is chosen according to the number of elements in the input list. This number is designed to scale up the output fitness (the first part of the equation) for better precision due to computation round up. Since we give equal weight to each of these three components of the fitness function (range between 0 and 1), the same value is applied to scale up the two run-time errors fitness.

The first part of equation 7 specifies that a program which returns the value at the expected position receives a fitness 10. This fitness value decreases as the position of the returned value is farther away from the position of the expected value. If the returned value is

not a part of the input list (it's an "0" returned as default by the run-time error handler for recursion error), this fitness is 0. When no run-time error is encountered during program evaluation, a program can receives a maximum fitness value of 10 for each test case. The fitness of a program is the summation of the fitness for all 12 test cases, i.e. the maximum fitness of a program is 120.

The two kinds of run-time error and their handling methods need further explanation. The first kind of run-time error is to apply function `head` or `tail` to an empty list (which has an undefined value). When this error occurs during program evaluation, a default value is returned (see Section 4.3.3) and the evaluation of the program continues. In this way, partial solutions can be considered for partial credit. For example, a program which is expected to return the fifth element of the list may return the third element of the list due to this error. This program is given partial credit even a non-optimal program is generated. The second kind of run-time error is the non-terminating recursion error. The recursive calls in a evolved `nth` program may generate infinite-loop, hence makes the evaluation of such a program non-terminating. This error is handled by specifying a limit of the number of recursive calls allowed in the program (the length of the input list, 10, is used in this case). When this limit is reached, program evaluation halts and returns with a flag to indicate this error. Consequently, no partial credit is given to this kind of program, i.e. the first part of the equation 7 is 0. Both of these two errors are also penalized in the fitness function (see Equation 7). Referring to the constraints handling methods defined in the Appendix A, the combination of "legal map" and "phenotype penalty" are used to handle these two constraints.

Results

10 runs were made and four of them found an optimal solution. All correct programs were found before 12,000 programs were processed. The shortest program found was:

```
if-then-else (less-eq (length (tail L))(minus N N))
         (head L)
         (if-then-else (gtr (minus N N)(minus one N))
         (nth (minus N one) (tail L))
         (head L))
```

Analysis and Discussion

In this experiment, the fitness function specifies that the optimal solutions not only have to produce the desired output but also have to satisfy two different constraints: a non-terminating

error constraint and an empty-list error constraint. To meet the three criteria, the evolutionary process is directed in different directions. Consequently, the seesaw of evolutionary pressure can favor either of the criteria (see Appendix A).

In the fitness function, these three criteria are given equal importance (10 point each). However, the two run-time error constraints are handled in ways which implicitly generated more weight for them. These effects can be illustrated in Table 5.2 where four categories are defined for a generated `nth` program when evaluated with a test case.

Table 5.2: The 4 categories of `nth` programs with different run-time errors.

Run-time Errors	Output	Penalty	Fitness
non-terminating+empty-list	nothing	20	-20
non-terminating	nothing	10	-10
empty-list	a value	10	partial credit-10
none	a value	0	calculated fitness

It is clear that programs of the first two categories, which are non-terminating, have very low fitness value hence are very unwelcome in the evolutionary process. They soon will be gotten rid off from the population pool. The evolution is to find programs which produce the correct output for this problem and also satisfy the empty-list error constraint (see Appendix D).

5.4.2 The Map Program

Problem Description: The `map` program takes inputs of two arguments, a function `F` and a list `L`. It returns the list obtained by applying `F` to each element of `L`.

Input Types: The input `F` has type `G1->G2` and the input `L` has type `[G1]`

Output Type: The output has type `[G2]`

Terminal Set: `{L:: [G1], nil:: [a], F:: (G1->G2)}`

Function Set: `{head::[a]->a, if-then-else::bool->a->a->a,`

```
tail::[a]->[a], cons::a->[a]->[a], null::[a]->bool,
F:: G1->G2, map::(G1->G2)->[G1]->[G2]}
```

Genetic Parameters: The experiments were carried out using a steady-state replacement method described in Section 4.6.1. The population size is 5,000; parent scalar value is 0.999; maximum tree depth is 5; and crossover rate is 100%. Each run terminates when a correct `map` program is found or 55,000 programs have been generated (5,000 by Creator and 50,000 by Evolvor).

Test Cases

Two different lists were used for the argument `L` and one function for the argument `F`. The two lists were: (1) a list with 10 elements whose values were the characters `A` to `J` and (2) an empty list. The function `F` converts an alphabetic character input into a number, i.e. `A` to `1`, `B` to `2`, `C` to `3` and so on. Table 5.3 lists these two test cases and their expected outputs.

Table 5.3: The 2 test cases for evolving `map` program.

Case No.	Value of F	Value of L	Expected Output
1	atoi	[A,B,C,D,E,F,G,H,I,J]	[1,2,3,4,5,6,7,8,9,10]
2	atoi	[]	[]

Fitness Function

The fitness value is computed based on how close the return list is to the expected list. There are two elements in this criterion: 1) whether the returned list has the correct *length* and 2) whether the returned list has the correct *contents*. Equation 8 is the fitness function used to measure both elements for each test case.

$$Fitness = -2 \cdot |length(L_e) - length(L_r)| + \sum_{e \in L_e} 10 \cdot (2^{-dist(e, L_r)})$$

$$-(10 + 2 \cdot length(L_e)) \cdot rtError \quad -(10 + 2 \cdot length(L_e)) \cdot reError \qquad (8)$$

where L_e is the expected list and L_r is the list returned by the generated program; $dist(e,L_r)$ is the distance between the position of e in the expected list and in the returned list. If e does not exist in L_r, i.e. $e \notin L_r$, this value is ∞. The *rtError* is 1 if there is a run-time empty-list error. The *reError* is 1 if there is a run-time non-terminating recursion error. Similar to equation 7, the value 10 is used to scale up the output fitness for better precision.

The first item in equation 8 measures whether the returned list has the same length as the expected list. Each discrepancy is penalized with value of 2.

The second item in equation 8 measures whether the returned list has the same contents as the expected list. For each element in the expected list, the distance between its position in the expected list and in the returned list is measured. If the element is in the correct position in the returned list, a fitness value of 10 is given. This value decreases as the position of e in the returned list is farther away from the expected position. In the case that e does not exist in the returned list, this fitness is 0. The same measurement is applied to each element in the

expected list. For the first test case, a program which generates a list with the correct length and contents will receive a fitness value of 100. For the second test case, a program which generates a list with the correct length and contents will receive a fitness value of 0.

The third and the fourth items in equation 8 measure whether an empty-list or a non-terminating run-time error is encountered during program evaluation. The penalties of these two types of run-time error are the same. It is proportionate to the length of the expected output list (the same as the length of the input list). This decision is due to the recursion limit is set to be the length of the input list (see Section 5.4.1). It seems to be reasonable to penalize run-time errors using the same criterion. The handling method for these two run-time errors is the same as that described in Section 5.4.1.

When no run-time error is encountered during program evaluation, a program receives a maximum fitness value of 100 for the first test case and 0 for the second test case. The fitness of a program is the summation of the fitness for the two test cases, i.e. the maximum fitness of a program is 100.

Results

Ten runs were made and three of them found an optimal solution. All correct solutions were found before 35,000 programs were processed. The shortest program generated is as the following:

```
if-then-else (null L)
             (head (cons nil nil))
             (cons (F (head L)) (map F (tail L))))
```

Analysis and Discussion

The experiment results indicate that the `map` program is harder than the `nth` program for GP to evolve. Unlike the `nth` program where only one correct return value is required to get the maximum fitness, an optimal `map` program has to be able to process each of the 10 elements in the input list correctly. To meet this objective, one more criterion is added to the fitness function: the length discrepancy between the expected and the generated lists (see equation 8). Moreover, unlike that for the `nth` program, the fitness function for the `map` program gives different importance to the two objectives and the two constraints. In the first test case, 100 points are given to programs which return list with the correct contents; 20 points are given to programs which return list with the correct length; 30 points are given to programs which produce no non-terminating error and 30 points are given to programs which produce no empty-

list error. This is designed to direct GP searching for programs which return list with the correct contents. In the second test case, the two objectives are not considered in the fitness function while the two constraints are each given 10 points of importance in the fitness function. It is obvious that the purpose of the second test case is to train GP to handle empty list without producing any run-time errors. In addition, the handling methods for the two constraints during program evaluation also generates another evolutionary pressures. Due to such a complicated fitness assignment and constraint handling pressure, the evolutionary process is directed into many directions. Consequently, the generation of the correct `map` program is harder than the generation of the correct `nth` program.

To analyze the effect of these fitness assignment to the evolutionary process, we identify four categories for the generated `map` programs when evaluated with the first test case (Table 5.4) and the second test case (Table 5.5).

Table 5.4: The 4 categories of `map` programs for the first test case.

Run-time Errors	Output	Run-time Error Penalty	Length Penalty	Fitness
non-terminating+empty-list	nothing	60	20	-80
non-terminating	nothing	30	20	-50
empty-list	a value	30	calculated penalty	partial credit-30 - calculated penalty
none	a value	0	calculated penalty	calculated fitness - calculated penalty

Table 5.5: The 4 categories of `map` programs for the second test case.

Run-time Errors	Output	Run-time Error Penalty	Length Penalty	Fitness
non-terminating+empty-list	nothing	20	0	-20
non-terminating	nothing	10	0	-10
empty-list	empty list	10	0	-10
none	empty list	0	0	0

With these two test cases, programs with non-termination errors are heavily discriminated in the population (the first and the second categories, with fitness values -100 and -60 respectively). Consequently, they would be eliminated from the population first. After that, evolution is a process of competition among programs to meet the three other criteria (see

Appendix D).

5.4.3 Evolving Recursive Programs

The implementation of recursion used in these two experiments has enabled GP to evolve both `nth` and `map` recursive programs. In this approach, the name of the program is included in the function set so that it can be used to construct the programs, hence making recursive calls. However, such recursive calls may cause the program become non-terminating during program evaluation. To discourage its happening, the termination of the program is treated as a constraint that the evolved programs have to satisfy. This constraint is made as an additional criterion in the fitness function that the solution has to meet. Consequently, the fitness function directs GP to evolve programs which not only generate desired outputs but also make successful recursive calls.

However, this approach is not ideal because it complicates the dynamic of program evolution with other issues. The first issue is the handling of infinite loops. In these experiments, the maximum number of recursive calls allowed in a program is the length of the input list. This limit may prevent GP from discovering good program segments if the program takes more than the permitted recursive calls to evaluate. The second issue is the fitness penalty applied to programs with infinite loops. In the `nth` experiments, the penalty is the size of the input list while in the `map` experiments, a program with infinite loops is penalized by an amount which is proportional to the length of the expected list. It is not clear whether these decisions are appropriate or not. Finally, the most important issue is that a small change in a recursive program can lead to large variation of the program's functionality and fitness. This means that recursive programs are extremely deceptive. The fitness of a recursive program therefore does not necessarily reflect its proximity to a solution in the space of programs. Due to these issues, the evolution of recursive programs becomes very difficult for the fitness-based GP search algorithm.

To overcome these issues, an alternative to provide recursion in GP is proposed. In this approach, recursion semantics are provided implicitly by a higher-order function. The program evolution process is therefore relieved from dealing with any issues related to recursion semantics. This work will be presented in details in the following chapter.

5.5 Summary

This chapter has presented the application of polymorphism in GP. We first present the concept of types in GP. The three different approaches to handle types in GP are then summa-

rized. With untyped GP, programs containing multiple types can lead to an unnecessary large search space. Moreover, type information is not able to be used to guide GP learning. With dynamically typed GP, the search space for programs with multiple types is still not constrained. Moreover, designing a scheme to transform type-incorrect programs into type-correct programs can be a complicated task. In contrast, strongly typed GP not only enforces the search space to contain only type-correct programs but also allows type information to be used to bias the selection of genetic operation locations. It is therefore a preferred method to implement type checking for GP in evolving programs with multiple types.

The two implementations of strongly typed GP are polymorphic GP and monomorphic GP. On the surface level, polymorphic GP is monomorphic GP added with type variables. However, a deeper analysis shows that the use of different type variables can impact GP search space in ways that are not obvious to GP users. We provide this information so that users can be cautious when using these type variables to design the function and terminal sets.

When no type variables are used to define functions and terminals, polymorphic GP is essentially monomorphic GP. Depending on the target problem, users can select polymorphism or monomorphism by incorporating type variables or not incorporating type variables. Type variables hence provide the flexibility of one GP system to perform two different type constraints implementations.

Two polymorphic programs, `nth` and `map`, are evolved using our polymorphic GP system. The two programs require the manipulation of multiple types, type variables and function types. The ability of polymorphic GP to generate these two programs demonstrates that polymorphism is, indeed, applicable to GP. Polymorphism also enhances the applicability of GP to problems that are very difficult for the standard GP to solve.

Both `nth` and `map` are also recursive programs. An approach which includes the program name in the function set is used to allow the evolved programs to make recursive calls. However, such kind of programs may also generate infinite loop hence becomes non-terminating during program evaluation. A method which treats the termination of programs as a constraint is used to discourage its happening. This approach is shown to be successful in these two cases. However, it also brings out other issues that GP faces when evolving general recursive programs. In the next chapter, a more sophisticated approach which uses higher-order functions to provide recursion semantics will be presented.

Chapter 6

Recursion, Lambda Abstractions and Genetic Programming

The previous chapter has identified GP issues in evolving recursive programs. In this chapter, an alternative mechanism to provide recursion in the evolved programs is introduced. With this approach, recursion is provided implicitly by the higher-order function `foldr`. This higher-order function recursively applies its function argument to each pair of the elements in the input list. In addition, the function argument is represented as a function module. Consequently, program representation using the higher-order function `foldr` provides a mechanism of module creation (as function argument) and reuse (through implicit recursion).

We first analyze the difficulties of GP to evolve recursive programs. An alternative using implicit recursion to provide recursion semantics in the evolved programs is then presented. This is followed by the introduction of a new module mechanism using λ abstraction. A comparison of λ abstractions with other module approaches is then provided. Next, the general `even-parity` problem [Koza, 1992] and other work using recursion or modules to solve the problem are summarized. We then introduce our new technique which uses the higher-order function `foldr` to evolve a *general solution* to this problem. The experiments and their results show that this approach has enhanced GP performance on this problem to a great degree that has never been reported before. Most importantly, a new term "structure abstraction" is introduced to describe the property emerged from the program representation. Our analysis indicates that structure abstraction has helped GP to find good program structures, hence reduced GP effort in finding the overall problem solutions. Finally, the limitations of implicit recursion to general GP problem solving is provided.

6.1 Challenges in Evolving Recursive Programs

Recursion is a powerful mechanism for program reuse. However, when a recursive program is not implemented with care, it may produce infinite loops and become useless (see Section 2.3.7). In Section 5.4.3, the issues related to evolving recursive programs are briefly mentioned. This section provides more detailed analysis to identify the challenges that GP faces when evolving recursive programs. These difficulties have hindered the success of GP in evolving recursive programs (see Section 3.4).

6.1.1 Determining the Indication of Non-terminating Programs

When evaluating a recursive program, there is no way of knowing whether the program is going to terminate or not, i.e. the halting problem is undecidable [Hopcroft and Ullman, 1979]. Instead of letting the program evaluation process run for an unknown length of time, a decision has to be made about what indicates the program will not halt. This is not only a difficult decision (because of the theoretical impossibility) but also an important decision because it has impact on GP in searching for problem solutions. If a "fit" program is wrongly classified as a program which does not terminate, the program does not have a chance to contribute to the search of problem solutions. Furthermore, if a class of programs are wrongly classified as programs that do not terminate, GP search is directed to overlook a potentially beneficial area in the search space. Consequently, the optimal solution may never be found.

Various approaches have been used as the indication of non-terminating recursive programs. Brave [1996] used the depth of the tree as the limit of recursive calls in his tree search recursive programs (see Section 3.4). Once the tree-depth number of iterations is reached, the evaluation returns the value of the current tree node. Wong and Leung [1996] adopted an execution time limit to halt their recursive `even-parity` programs (see Section 3.4). A program which takes longer than the permitted time to execute is considered as a program that does not terminate. In the previous chapter, we imposed the length of the input list as the recursion limit when evolving the `nth` and the `map` recursive programs. Although these approaches have enabled GP to evolve recursive programs successfully, no detailed analysis is provided about how these methods direct GP search.

6.1.2 Handling the Non-terminating Programs

When a program is classified as non-terminating during its evaluation, a decision has to be made about how to handle such a program. This is yet another difficult and important deci-

sion. It is difficult because non-terminating programs can be of many diversified contents. Attempting to design a handling method which is appropriate for all non-terminating programs is a challenging task. The decision is also important because the consequences can impact how GP searches for problem solutions.

The handling method consists of two parts: 1) the return value of the program and 2) the penalty, if any, to be reflected in the fitness function. A non-terminating program may still contain good partial solutions. Ideally, these partial solutions should be credited so that they can be used to generate new and hopefully better programs. To achieve this goal, the return value for the non-terminating programs has to be defined. This value has influence on whether or not the partial solutions are credited and how they are credited (examples are given later). Moreover, the non-termination of a program can also be reflected in the fitness function to guide GP search. Ideally, the fitness function should be designed to promote programs which not only produce the correct outputs but also terminate. The analysis in the Appendix A has shown that such a goal is difficult as the evolution is directed in different directions.

Wong and Leung [1996] regarded a program which does not produce a result after the allowed execution time as a program producing a wrong result. No partial credits nor extra penalty is given to the program. Similarly, we return an empty list for non-terminating programs when evolving the `map` and the `nth` programs, hence no partial credits are given (see Section 5.4). However, we do penalize non-terminating programs. How these handling methods affect GP performance has to be further studied.

6.1.3 Measuring the Recursion Semantics in the Programs

The standard GP paradigm uses a syntactic approach to build programs; no semantic analysis is supported. A recursive program which contains a perfect base-case statement (see Section 2.3.7) is therefore not necessary to be selected for reproduction since program structure are not normally considered during GP search. Whigham and McKay [1995] have identified this problem and suggested the application of genetic operators to be performed in an environment where semantic analysis is supported.

6.2 Implicit Recursion

The issues that explicit recursion highlighted in GP can be solved by a particular functional programming technique - implicit recursion. Functional languages provide various higher-order functions to support implicit recursion. They are `map`, `fold` (includes `foldr` and

`foldl`) and `filter`. These higher-order functions provide recursion semantics in the programs without explicit recursive calls. The operation of these higher-order functions are described in Section 2.3.7.

An important characteristic of implicit recursion is that programs always terminate. This is because the terminating condition is incorporated into the higher-order functions. Moreover, there are no explicit recursive calls in the programs as the recursion is performed in the higher-order functions. Consequently, none of the issues raised in Section 6.1 applies to implicit recursion. This makes implicit recursion an ideal mechanism to support recursion in GP.

6.3 Lambda Abstractions Module Approach

This work also introduces a new module mechanism using λ abstractions. λ abstractions are local function definitions within programs. The creation of λ abstractions in the program trees has been described in Section 4.2.1. The role of λ abstractions in the programs is to provide a mechanism of module creation and reuse. For problem solutions which contain regularity, a module mechanism allows for the decomposition of the problem into smaller problems and the use of the solutions of the smaller problems to compose the overall solution. With such a goal in mind, various module mechanisms have been proposed to assist GP performing problem solving (see Section 3.3).

The λ abstractions module approach is similar to Automatically Defined Function (ADF) (see Section 3.3.1) in the following ways:

- a λ abstraction has formal parameters and a function body.
- λ abstractions are simultaneously evolved with the main program.

Consequently, λ abstractions provide the same two functions in GP as that provided by ADFs: first, they perform a top-down process of problem decomposition or a bottom-up process of representational change to exploit identified regularities in the problem. Second, they discover and exploit inherent patterns and modularities within a problem [Koza, 1994a].

However, λ abstraction module mechanism differs from ADFs in the following areas:

- λ abstractions are anonymous hence cannot be invoked by name. The reuse of λ abstractions is carried out by passing them as arguments to other functions which then reuse the λ abstraction. In the following example, the λ abstraction (λ x. (+ x 1)) is reused in the `twice` function:

```
twice f x = f (f x)
```

```
twice (λ x. (+ x 1)) 2
= (λ x. (+ x 1)) ((λ x. (+ x 1)) 2)
= + ((λ x. (+ x 1)) 2) 1
= + (+ 2 1) 1
= + 3 1
= 4
```

- The determination of the program structure with λ abstraction is different from that with ADFs. Instead of having the program structure predefined in advance or complete open to evolutionary determination (see Section 3.3.1), λ abstractions are created *dynamically* by GP according to the function arguments specified by the higher-order functions (see Section 4.2.1). Briefly, when a higher-order function is selected to construct the program node, its function argument is created as a λ abstraction. In this way, a priori knowledge about the creation and reuse of λ abstractions can be incorporated in the higher-order function to facilitate GP in determining the most effective program structure.

There are two other approaches which also support module creation and reuse: Module Acquisition (MA) and Adaptive Representation through Learning (ARL) (see Sections 3.3.2 and 3.3.3). In both approaches, program structures are created and modified dynamically during GP run. One important concept about MA and ARL is that modules are building blocks and should be protected from destruction. Modules in MA and ARL are therefore frozen for a period of time without any changes. The only way to modify a module is to Delete (in ARL) or to Expand (in MA) the module and to create a new one. Because of the less frequent modification, the quality of the modules becomes very important. Kinnear, Jr. [1994a] reported that the MA approach, where modules are created using randomly extracted program fragments, does not provide performance advantages. ARL adopts heuristics to detect good program segments for module creation. This approach produces better programs than GP alone [Rosca and Ballard, 1996].

6.4 The Even-Parity Problem

The `even-parity` has been used by many researchers as a benchmark problem for GP to solve [Koza, 1992; Wong and Leung, 1996; Chellapilla, 1997b; Gathercole and Ross, 1997; Poli, Page and Langdon, 1999]. This problem may have different number of inputs. For N number of inputs, the solution of the problem returns True if an even number of inputs are True. Otherwise, it returns False. This problem uses the following function and terminal sets:

- Function Set: {and, or, nand, nor}.

  These are standard logic functions and are logically complete in the sense that all Boolean functions can be built using these four functions.

- Terminal Set: $\{b_0, b_1, \ldots, b_{N-1}\}$.

  These are the N number of Boolean inputs.

Rosca [1995] and [Poli, Page and Langdon, 1999] have identified two reasons why the even-parity problem is difficult:

- The problem solution is very sensitive to the value of the inputs. A single change of one of the N number inputs would generate a different output.
- The function set used does not contain the Boolean functions xor or eq. These two useful building blocks to this problem have to be discovered by GP.

Moreover, as the value of N increases, the problem becomes more difficult. Koza has experimented with different values of N for this problem. Using the standard GP, he was able to solve the problem up to N=5. When N=6, none of his 19 runs found a 100%-correct solution [Koza, 1992]. His results also indicate that the number of program evaluation required for the standard GP to solve the problem increases by about an order of magnitude for each increment of N.

There are two other works which use modules or recursion to solve the even-parity problem:

- Using ADFs to support modules in GP, Koza has solved this parity problem up to N=11 [Koza, 1994, Chapter 6].
- Wong and Leung [1996] has evolved a general solution to this problem using recursion. In their approach, a logic grammar is used to enforce the terminating-condition in the programs. Moreover, type knowledge and semantics information are also incorporated in the logic grammar to guide GP search (see Section 3.1.2). Using this method, GP was able to find a solution more efficiently than that using ADFs. However, their method requires a significant amount of domain knowledge. Moreover, the explicit recursion approach (see Section 3.4) has the disadvantages that summarized in Section 6.1. Finally, the generated programs do not contain any subroutines, although an xor function can be extracted from the programs.

In the following section, we present a new strategy to evolve a *general solution* to the even-parity problem which works for any value of N, i.e. solving the general even-parity problem.

6.5 A New Strategy

To solve the general `even-parity` problem, we introduce a new strategy which uses the higher-order function `foldr` to support implicit recursion and module creation (as λ abstractions). Implicit recursion enables GP to generate general solutions which work for any value of N while λ abstractions provide a module mechanism for GP to exploit the structure inherent in the `even-parity` problem. This combination of implicit recursion and λ abstractions will be shown to provide great performance advantages over previous work with this problem.

6.5.1 FOLDR: Implicit Recursion

For this problem, the higher-order function selected to provide implicit recursion is `foldr`. This is because `foldr` produces a single output value and so does the general `even-parity` program. For different problem domains, other higher-order functions might be more suitable. Moreover, combining different higher-order functions or leaving GP to decide the most beneficial combination for each problem is possible.

6.5.2 Lambda Abstractions: Module Mechanism

It might be possible to evolve the general `even-parity` program using implicit recursion alone without λ abstractions. In this case, the function argument to `foldr` would be selected from the function set provided by the users. However, in this research, we would like to explore the structure inherent in the problem solutions. The function argument is therefore allowed to be discovered through the evolution of λ abstractions. The investigation of GP performance using implicit recursion to solve the general `even-parity` problem with and without λ abstractions will be conducted in the future.

The creation of λ abstraction has been described in Section 4.2.1. Briefly, each time `foldr` function is selected to construct a program tree node, a λ abstraction is generated as its function argument. The λ abstraction is then reused by the higher function `foldr` through implicit recursion. During programs evolution, both the main program and its λ abstractions are subject to genetic operations (see Section 4.4.3).

6.5.3 Type System: Structure Preserving Engine

A type system is used to preserve the structure of λ abstractions and implicit recursion in the program trees. The details of the type system are described in Section 4.5. In brief, `foldr`

function is specified with the following type:

```
foldr :: (a->b->b) -> b -> [a] -> b
```

This type information indicates that `foldr` takes three arguments: the first one is a function, the second one is a value and the third one is a list. It returns a single value. Additionally, the first argument is a function which takes two arguments and returns one value. During the generation of the initial population, each time `foldr` is selected to construct the program, a 2-input 1-output λ abstraction is generated as its function argument.

During program evolution, the program structure is also maintained by the type system. This is accomplished by the "point-typing" mechanism (Section 4.4.2). Thus, the structure of λ abstractions and implicit recursion can be preserved throughout the evolutionary process.

Notice that `foldr` is a *polymorphic* function whose types contains *type variables*. The type system instantiates these type variables each time `foldr` function is selected to construct a program tree node. By supporting type variables in the type language (see Section 4.5), the generality of functions in the function set are enhanced. For example, `foldr` can be used to provide implicit recursion for many different types of arguments (see Section 6.6.3). However, type variables may also make the search space larger than necessary (see Section 5.3.1). This work has identified that the `even-parity` can be solved without polymorphism (see Section 6.8.1). We will run the same experiment by replacing type variables with Boolean types in the next chapter.

6.6 Experiments

The following terminal and function sets are used in the experiments:

Input Type: The input `L` has type `[bool]`. (`L` is a list of N Boolean values)

Output Type: The output has type `bool`.

Terminal Set: `{L::[bool]}`

Function Set:

```
{head::[a]->a, tail::[a]->[a],
 and::bool->bool->bool, or::bool->bool->bool,
 nand::bool->bool->bool, nor::bool->bool->bool,
 foldr::(a->b->b)->b->[a]->b}
```

6.6.1 Test Cases

The test cases of the `even-2-parity` and the `even-3-parity` are selected to evaluate the generated programs. There are therefore $2^2 + 2^3 = 12$ test cases. This decision is based on

our observation that a general `even-parity` program should be able to handle any number of inputs, i.e. the value N can be an any odd or even number. The test cases of `even-2-parity` help GP to learn to handle an input list with an even number of elements while the test cases of `even-3-parity` train GP to work on input lists with an odd number of elements. With this set of test cases, it will be shown that the generated programs are general solutions which work for any value of N.

6.6.2 Fitness Function

The fitness function used is the same as that used by Koza [Koza 1992, page 160] except that an empty-list run-time error is punished (see Section 4.3.3 for run-time errors handling and Section 5.4 for examples). Each program is evaluated against all of the 12 test cases. When a correct result is produced for a test case, the program receives a 1; otherwise, it receives a 0. If the empty-list error has been flagged during program evaluation, fitness is reduced by 0.5. The fitness of a program is the sum of the fitness values for all of the 12 test cases. Thus, a perfect solution would receive a fitness of 12. Based on the fitness function, 4 categories are defined for the generated general `even-parity` programs (see Table 6.1).

Table 6.1: The 4 categories for the general even-parity programs.

Empty-list Run-time Error	Output	Fitness
flagged	wrong	-0.5
not flagged	wrong	0.0
flagged	correct	0.5
not flagged	correct	1.0

This fitness function was designed to assign more importance on output fitness than on run-time error fitness, i.e. 1.0 vs. 0.5. However, further investigation shows that most programs with run-time errors do not produce correct output, even with an error handler that encourages partial solutions (see Appendix D). Consequently, not many programs belong to the third category in Table 6.1. This also means that programs with run-time errors are mostly with low fitness (-0.5) and would be eliminated from the population very fast during the evolutionary process. These results and other related issues will be discussed in Appendix D.

6.6.3 Genetic Parameters

The experiments are conducted using generational replacement method described in Section

4.6.1. Following the GP setup defined in [Koza, 1992, page 114], a population size of 500 and a maximum generation of 51 (an initial random generation, called generation 0, plus 50 subsequent generations) are used. However, the crossover rate is 100%, i.e. no mutation nor reproduction is performed.

A λ abstraction is considered as one single node in the program trees as it performs one single task just like a function in the function set. For example, in Figure 6.1, the λ subtree inside the dash-line is considered as one single node. Consequently, the program is considered as having tree depth 3.

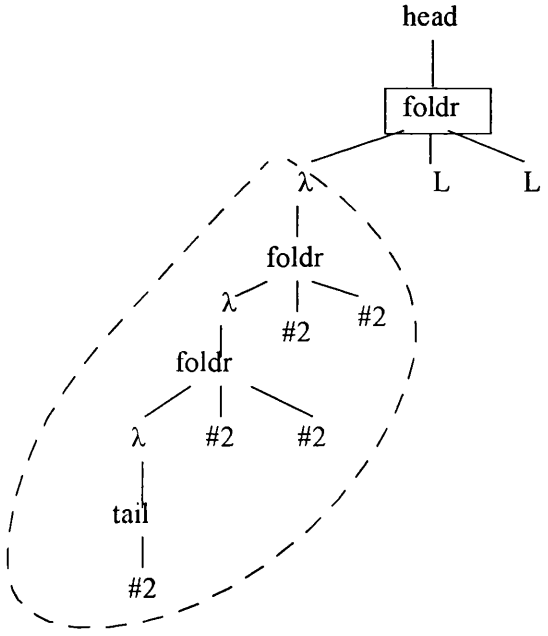


Figure 6.1: A program with nested λ abstractions.

The polymorphic feature defined by `foldr` function has enabled nested λ abstractions to be created in a program tree. For example, the boxed `foldr` function in Figure 6.1 has its type variables instantiated in the following ways:

```
foldr :: (a->b->b) -> b ->[a] -> b
         (Boolean->[Boolean]->[Boolean])->[Boolean]
         ->[Boolean]->[Boolean]
```

I.e., `a` is instantiated to Boolean while `b` is instantiated to list of Boolean. This makes its first argument to be a function of two arguments: one with Boolean type and the other with list of Boolean type. During the creation of the function argument (as a λ abstraction), `foldr` function may be selected to create a program node. This requires another λ abstraction to be created as the function argument for the selected `foldr` function. As a result, nested λ abstractions are created. Moreover, as mentioned before, a λ abstraction is viewed as one sin-

gle node in the program tree. Consequently, `foldr` function can be selected to construct a λ abstraction as many times as it happens (2 in Figure 6.1). With such a setup, there is no need to have large tree depth to obtain big size trees. The maximum tree depth is therefore set to be 4. The same maximum tree depth (4) is also applied to λ abstraction subtrees for the same reason.

However, a `foldr` function inside another `foldr` function creates nested recursion. Nested recursive programs require a considerable amount of time and space to evaluate. The depth of the nested recursion is therefore limited to 100. Once this limit is reached during program creation, `foldr` is excluded from being used to construct the λ abstraction. The same rule is also applied during genetic operations. As shown by the experiment results, this limit is powerful enough to handle the general `even-parity` problem. In fact, a further analysis has indicated that the `even-parity` can be solved with a limit of 3 `foldr` in a program tree (see Section 6.8.1).

6.7 Results

Sixty runs were made and 57 of them found a perfect solution. Moreover, all 57 are general solutions which work for any *N* number of inputs. To facilitate direct comparison with others' work, we have adopted the most widely used measurement method within GP field, the "effort" requirement described in [Koza, 1992, Chapter 8], to evaluate the performance of the new strategy. A detailed explanation of this method is given in Appendix B.

Figure 6.2 shows the performance curves of the experimental results.

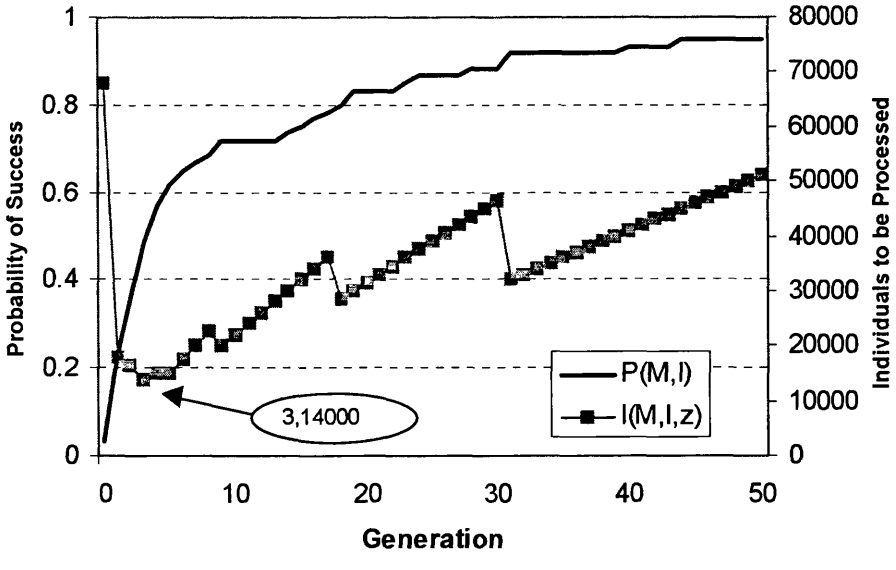


Figure 6.2: Performance curves for the general even-parity problem.

The curve P(M,i) shows the cumulative probability of success to solve the problem at each generation. It indicates that 2 runs found a solution during generation 0 through random search and more than 50% of the 60 runs obtained a solution before generation 5. Moreover, all 57 successful runs found solutions before generation 44.

The "effort" curve, I(M,i,z), indicates the number of program evaluations required at each generation to find a perfect solution with 99% confidence. This value is calculated using the following formula given by [Koza, 1992, page 194]:

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil \text{, where } z = 99\% \qquad (9)$$

The smallest value on this curve is used to indicate the minimum "effort" required for GP to solve the given problem. According to the experiment results, the curve of I(M,i,z) reaches a minimum value of 14,000 at generation 3 (marked on the figure). Since 12 test cases were used to test each program, the number of test cases processed was 168,000.

6.8 Analysis and Discussion

The results of our experiments indicate that by using the structure of λ abstractions and implicit recursion, GP is able to evolve the general solutions to the `even-parity` problem very efficiently. This is in comparison with the results reported by other researchers using the same measurement method to evaluate their GP system in solving the same problem. In particular, we compare our results with two other previous works in using modules or recursion to solve this problem. Table 6.2 summarizes the performance of related work. Note that the "effort" value, I(M,i,z), indicates the performance of the overall GP system in solving the problem. No detailed comparison of each components of the system is made in this work.

Table 6.2: Performance summary for the even-parity problem.

Results	Implicit Recursion + λ Abstractions	Generic Genetic Programming	GP with ADFs
Programs	general even-parity	general even-parity	even-7-parity
Runs/Success	60/57	60/17	29/10
Minimum I(M,i,z)	14,000	220,000	1,440,000
Number of Fitness Cases	12	8	128
Fitness Cases Processed	168,000	1,760,000	184,320,00

With the ADF module mechanism, Koza was able to use GP to evolve the `even-11-parity` programs. However, the performance details were not reported. We therefore show the performance of his work on the `even-7-parity` instead. We also show the performance of the Generic Genetic Programming (GGP) system [Wong and Leung, 1996] which uses recursion to evolve a general solution for the same problem.

As shown, the structure of λ abstractions and implicit recursion has enabled GP to evolve the general `even-parity` programs by processing much fewer programs than the number required either by the ADF approach or by the GGP system. Besides the benefits of recursion and modules, we believe that there is one more factor which contributes to such an exceptional performance:

Higher-order functions provide structure abstraction in the program trees. The type system protects this structure abstraction and helps GP to find good program structures during program evolution.

The ability of the standard GP to build good solutions from partial solutions hierarchically has been challenged [O'Reilly and Oppacher, 1995]. The module mechanisms of ADFs, MA and ARL are designed to facilitate GP in hierarchical processing by abstracting program *contents*. The λ abstractions module mechanism promotes the use of hierarchy further by supporting program *structure abstraction* (a formal definition will be provided in Chapter 7). As an argument to a higher-order function, a λ abstraction is constrained to sit underneath the higher-order function in the program tree hierarchy. During program evolution, the type system protects this two-layer-hierarchy program structure grouping from disruption; crossover can only change its contents but not its structure. Consequently, GP can use the two-layer-hierarchy structure as one unit to exploit the most advantageous program structure.

Figure 6.3 shows three structure abstraction groupings. Note that they may have different contents since the three λ abstractions may be different.

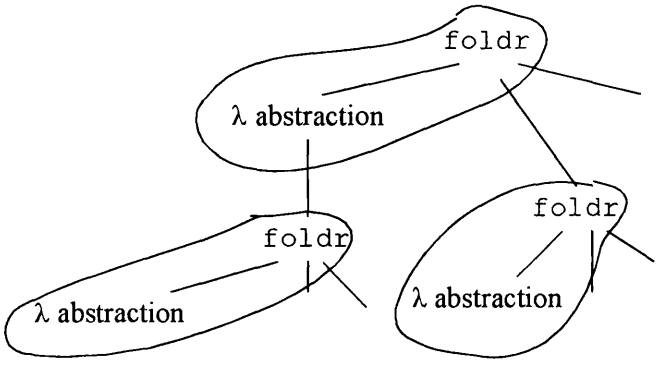


Figure 6.3: Structure abstraction grouping with foldr.

6.8.1 Program Structure Evolution with Structure Abstraction

To investigate the impact of structure abstraction on GP program structure evolution, we have conducted two sets of experiments. In the first experiment, 100,000 programs were randomly generated. These programs contain different number of `foldr` structure abstraction groupings. Table 6.3 summarizes the results.

Table 6.3: Results of the 100,000 randomly generated even-parity programs.

No. of foldr structure abstraction groupings	No. of programs	No. of program with above average fitness
0	10,068	1,589
1	41,104	2,885
2	13,516	429
more than 2	35,312	0

As shown in the third column, the programs which have above average fitness are either with 2 or 1 or no `foldr`. We have also examined all 57 generated correct programs and found them contain either 1 or 2 occurrences of `foldr` (see Table 6.4). This suggests that structure abstraction has helped identifying good structures for the general `even-parity` programs at generation 0. Most evolutionary effort is to search for the program contents to fill in the program structures.

To confirm this hypothesis, a second experiment is conducted. In this experiment, 10 GP runs were made and the evolution of program structures (in terms of the number of `foldr` structure abstraction groupings in the programs) is recorded. The results are shown in Figure 6.4.

At generation 0, different program structures were created (the proportion is very similar to the results of the first experiment). However, once the GP evolution process began, the number of programs with more than 2 `foldr` decreased quickly. At generation 3, the population contained no program with more than two `foldr`. GP evolution became a process of competition among programs with 0, 1 or 2 `foldr`.

These results confirm our hypothesis that structure abstraction helps identifying good program structures at generation 0. Consequently, the evolutionary efforts required to find program structures for the solutions is reduced. GP evolution becomes focused on the search of good program contents of the solutions. As a result, the solutions can be found more efficiently.

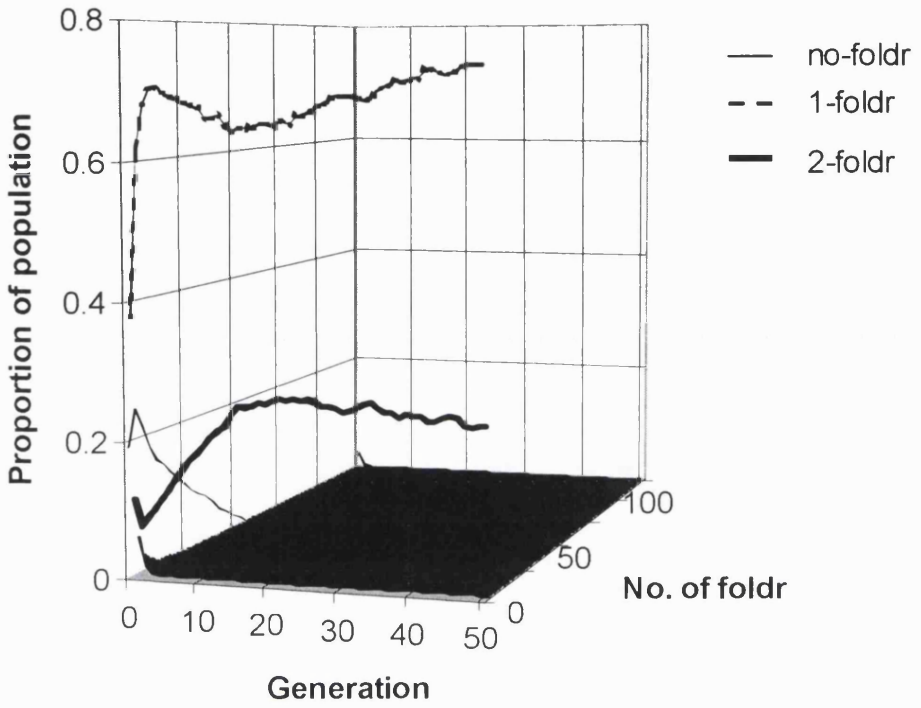


Figure 6.4: The evolution of program structure grouping.

We have also observed two interesting phenomena in these two experiments:

- Random search has generated more programs with one `foldr` than any other kinds of program structures. A close examination of the function set tells us that `foldr` is only one out of seven functions that can be used to construct a program tree's internal nodes. Since the Creator randomly selects type-matched functions to build program trees (see Section 4.2), unless the type variables of `foldr` is instantiated to a type other than Boolean (which the experimental records show that it is less likely), `foldr` function has a possibility of 1/7 to be selected to construct a program tree node. In conjunction with the tree depth restriction, it is highly possible that `foldr` is selected once during the creation of a program tree.
- All the generated programs (either by random search or evolutionary search) can be divided into two groups. In the first group, type variables of `foldr` are consistently instantiated to Boolean type, i.e. `foldr` is used as a monomorphic function. In the second group, type variable `b` of `foldr` is instantiated to Boolean type in some cases and to list of Boolean type in others, i.e. the polymorphic feature of `foldr` is utilized. Moreover, the first group consists of programs with 1 or 2 or no `foldr` while programs with more than two `foldr` belong to the second group. As indicated in our investigation, programs with 1 or 2 or no `foldr` are the promising area of the search space. Hence, by

specifying `foldr` as a monomorphic function, the `even-parity` problem can be solved more efficiently. This is one example where monomorphism is more advantageous than polymorphism in GP search (see Section 5.3.1).

6.8.2 The Generated Perfect Solutions

The 27 generated correct programs can be divided into 8 groups (see Table 6.4). In each group, the λ abstraction in the programs represents the same Boolean function. Those functions are anonymous in the programs but for easy reference, we use the names defined by Koza [Koza, 1992, page 228] and present them in *italics*, i.e. *r1* to *r16*. The Truth Table for these Boolean functions are presented in Table 6.5. Note that those λ abstractions which compute the same Boolean function might contain very different code. The values True and False in Table 6.4 indicate expressions which produce True or False under all conditions.

Table 6.4: Generated correct general even-parity programs.

Number of Programs	General Even-Parity Program
22	nor (foldr *r6* (head L)(tail L)) False
9	foldr *r6* (nor (head L)(head L)) (tail L)
6	nor (foldr *r6* (head L)(tail L))(foldr *r4* (head L)(tail L))
6	foldr *r6* (nand (head L)(head L))(tail L)
6	nand (foldr or (head L)(tail L))(foldr *r6* (head L)(tail L))
5	nand (foldr *r6* (head L)(tail L)) True
2	foldr *r3* (foldr *r9* (head L)(tail L)) (tail (tail L))
1	nor (foldr *r6* (head L)(tail L)) (foldr *r6* (head L) (tail L))

Table 6.5: Truth table for the λ abstraction in the generated programs.

x	y	r3	r4	r6	r9
False	False	False	False	False	True
False	True	False	True	True	False
True	False	True	False	True	False
True	True	False	False	False	True

6.8.3 Limitations of Implicit Recursion

We also analyze the limitations of implicit recursion provided by higher-order functions. Generally, implicit recursion can be achieved by any higher-order functions which contain recursion semantics. This means that the higher-order function contains the following two elements:

- one or more function arguments;
- the reuse of at least one of the function arguments.

However, a higher-order function which satisfies the two requirements does not necessary provide effective implicit recursion for the target problem. As being shown, the implicit recursion provided by the `foldr` higher-order function is very effective with the general `even-parity` problem. With `foldr`, the recursion is applied to a list structure, i.e. recursion is carried around a list of inputs. However, not all problem contain this recursion pattern. For example, in the Fibonacci sequence problem [Koza, 1992, page 473], the recursion is carried around a numerical value. In this case, a different higher-order function has to be designed to provide effective implicit recursion. Although the concept of using the implicit recursion provided by higher-order functions to evolve recursive programs can be applied to general problems, it's only the properly designed higher-order functions can receive the benefits.

6.9 Summary

Module creation and reuse are important for GP to be effective with problems whose solutions contain regularities. This chapter has presented a particular kind of program representation to supports module creation and reuse. In this representation, modules are represented as λ abstractions and their reuse is achieved through implicit recursion. A type system is used to preserve this representation during program evolution.

This style of module creation and reuse provides the following advantages:

- Module creation is neither a random process nor a predefined condition. A randomly generated module may or may not be beneficial to the problem to be solved. On the other hand, a hard-wired module template precludes the generation of more advantageous program structures. Our new approach is to generate modules dynamically, based on the function argument specified in the higher-order functions. This allows the exploration of beneficial program structures according to the higher-order functions defined by the users.

- Implicit recursion provides reuse without the possible side effect of infinite loops because there are no explicit recursive calls present in the program. This is an inherent feature of implicit recursion. Such a condition not only relieves GP from handling infinite loops in a program but also removes the need for GP to measure the semantic elements of recursive programs. Consequently, the fitness would reflect the ability of the program in solving the target problem more accurately, hence direct GP to search for problem solution more effectively.
- This style of program representation provides *structure abstraction* in the programs. Our investigation indicates that structure abstraction has helped identifying good program structures at generation 0. The evolutionary effort is hence reduced to search for program contents for the solutions.

We have evolved the general solutions to the `even-parity` problem using GP with this program representation. The results show that this style of module creation and reuse is very effective with this problem. As will be shown in the next chapter, the selected functions and terminals are very suitable to this problem. The detailed investigation of why such a representation is so effective with this problem is the topic of the next chapter.

Chapter 7

Structure Abstraction and Genetic Programming

In the previous chapter, a program representation which incorporates a higher-order function `foldr` was introduced to solve the general `even-parity` problem. This program representation contains a particular structure pattern named *structure abstraction* which we have identified to be the cause of its outstanding performance. However, when we use the same kind of program representation (with a different higher-order function) to solve the artificial ant problem [Koza, 1992], no performance advantage was found (see Appendix C). This prompts us to investigate why structure abstraction provided by `foldr` is particularly effective with the general `even-parity` problem.

This chapter starts by providing a formal definition of structure abstraction. This is followed by a detsiled description of the application of structure abstraction to the general `even-parity` problem. Next, a systematic analysis of program structures is presented. We investigate all perfect solutions in the search space. Based on the analysis, the effort required to find a solution to the problem is calculated. The result indicates that due to structure abstraction, the complexity of the general `even-parity` problem is reduced to that of a much simpler Boolean function with two arguments problem.

The GP experimental results are consistent with our analysis. Indeed, structure abstraction provides a mechanism of hierarchical processing to solve this problem, hence enables the solution to be found very efficiently. As there is a trend in developing problem-specific evolutionary algorithms [Leonhardi, Reissenberger, Schmelmer, Weicker and Weicker, 1998], we make the first-step in formulating guidelines for the application of structure abstraction to other problems.

7.1 Structure Abstraction in Program Evolution

The structure evolved by GP is a program tree where the internal nodes are labelled with

functions and the external nodes (leaves) are labelled with terminals. During program evolution, each subtree and leaf node acts as an independent unit and can be replaced by any other subtree and leaf node (based on the standard GP implementation defined in [Koza, 1992, page 114]). It is hoped that the evolutionary process will promote "good" subtrees/leaf nodes to compose the program which solves the target problem.

When modules are incorporated in the program representation, some additional dynamics are introduced to the evolutionary process. A module is a block of code represented as a single tree. It might or might not have any input arguments. It normally, but not always, produces outputs. With ADFs (see Section 3.3.1), a module is provided with a name. Like other functions and terminals, the names of ADFs can be used to compose the main program. In other words, an ADF can appear anywhere in the main program tree. During GP runs, both ADFs and the main program are evolved simultaneously.

Modules can also be incorporated into program representation through the use of higher-order functions (see Chapter 6). A higher-order function is a function which takes other functions as arguments or returns functions as outputs. When a higher-order function is included in the function set, its function arguments can be represented either as function names or λ abstractions in the evolved program trees. In this work, we choose λ abstractions to represent the function arguments (see Section 6.5.2).

Similar to ADFs, λ abstractions are modules which evolve during GP runs. However, unlike ADFs, λ abstractions can only be positioned in a certain way in the program trees. As an argument to a higher-order function, a λ abstraction is constrained to sit underneath the higher-order function in the program tree hierarchy (see Figure 7.1).

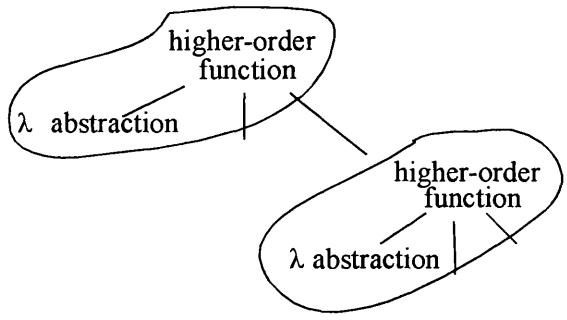


Figure 7.1: Structure abstraction in program tree hierarchy.

Consequently, a higher-order function and its function argument group into a two-layer-hier-

archy in the program trees. We term this two-layer-hierarchy "structure abstraction" as it groups into one structural unit during program creation and evolution. Although the contents of a grouping can change, i.e. λ abstractions also evolve, the two-layer-hierarchy structure grouping of higher-order functions and its function argument holds throughout the evolutionary process. Structure abstraction is a common property for any program representation using higher-order functions.

7.2 Structure Abstraction on Even-Parity Problem

The `even-parity` problem has been described in Section 6.4. We implemented structure abstraction for this problem the same way as that in the previous chapter. However, the polymorphic functions are specified as monomorphic by replacing all type variables with Boolean types. This change will be discussed in the following subsection.

7.2.1 Program Representation with Structure Abstraction

The higher-order function selected to support structure abstraction for this problem is `foldr`. The operation of `foldr` is described in Section 2.3.7. In brief, this function takes 3 arguments: the first argument is a function, the second argument is a value and the third argument is a list. It returns a single value. The function argument is represented as a λ abstraction in the program tree, hence provides the structure abstraction grouping in the program trees.

The inputs to the problem are given through a list named `L`, i.e. `L` is a list of N Boolean values. To allow the processing of each item in the input list, two more functions are included in the function set: `head` and `tail`. The function `head` takes a list as argument and returns the first element of the list. The `tail` function takes a list as argument and returns the list with its first element removed.

The functions `foldr`, `head` and `tail` are specified with types that are different from the previous chapter 6 (see Table 7.1). In Chapter 6, these functions are polymorphic. Our analysis has identified that the structure abstraction provided by polymorphic `foldr` helped identifying good program structures (those with 0, 1 or 2 `foldr`) at generation 0 hence allowed the solutions to be found very efficiently. In this chapter, we further our investigation on the impact of structure abstraction on GP search within the search space of these good program structures. Since this search space (programs with 0, 1 or 2 `foldr`) can be defined by using monomorphic `foldr` (see Section 6.8.1), this chapter uses monomorphic implementation of the functions to conduct this research.

Table 7.1: Functions and terminals with their types.

Name	Type
and	bool->bool->bool
or	bool->bool->bool
nand	bool->bool->bool
nor	bool->bool->bool
foldr	(bool->bool->bool)->bool->[bool]->bool
head	[bool]->bool
tail	[bool]->[bool]
L	[bool]

7.2.2 Type System

The structure abstraction grouping in the program trees is preserved by the type system described in Section 4.5. In brief, based on the function argument specified for `foldr` (a function of two Boolean inputs and one Boolean output), the type system would allow only such kind of λ abstractions to be generated during program creation. During crossover and mutation, the "point typing" method (see Section 4.4.2) is applied to maintain the structure abstraction grouping throughout the program evolutionary process.

7.3 Program Structures in the Search Space

The search space of GP consists of all program trees that can be composed using the available functions and terminals. The number of such trees increases rapidly as the number of tree nodes increases. This fast growth is due to 1) the substantial number of different tree structures and 2) the enormous number of permutations in labelling the internal and external nodes using the provided functions and terminals. Without any restriction on its program trees, the GP search space is infinite. In practice, however, most implementations either restrict the number of tree nodes or the tree depth to prevent the otherwise explosive computation. This work applied a tree depth restriction of 4 to evolve the solutions to the general `even-parity` problem.

We analyze program structures in the search space. The term "program structure" is used to describe a program where a λ abstraction module is considered as one partial solution unit. Hence, only the semantics (outputs) of the λ abstractions are considered in the program struc-

tures. The contents of the λ abstractions are ignored in this analysis. However, they will be discussed in the next section during the calculation of the effort required to solve the problem (see Section 7.4).

At first, the program structure of `foldr` is investigated because it is more complicated than the rest of the functions and terminals. A `foldr` program tree can only be constructed in some restricted ways due to the type constraints specified by the functions and terminals (see Table 7.1). The first argument of `foldr` is specified with a function type (bool->bool->bool). This function argument is constructed as a λ abstraction which takes 2 inputs of Boolean type and produces one output of Boolean type. The terminal set used to construct a λ abstraction is designed such that only arguments of the λ abstraction are included (see Section 4.2.1). Since both arguments are Boolean values, there is no terminal with the type of Boolean list that can be used to construct the λ abstraction. Consequently, those functions (`head`, `tail` and `foldr`) which require Boolean list type argument can never appear in a λ abstraction. Only the four Boolean operators (`and`, `or`, `nand` and `nor`) can be used to construct the internal nodes of λ abstractions. With two Boolean inputs and one Boolean output (each of them can be either True or False), there are total of 16 possible Boolean functions that a λ abstraction can generate.

The second argument of `foldr` is of type Boolean. There are six functions which return a Boolean value and hence can be used to built the internal nodes of the subtree. However, there is only one terminal `L` which can be used to construct the leaves of the subtree. As specified, `L` has type of Boolean list. To bridge between the leaf type (Boolean list) and the subtree root node type (Boolean), the `head` function is used. Consequently, the subtree (`head L`) occupied the fringes of the subtree representing the second argument.

The third argument of `foldr` is specified with Boolean list type. Among the functions and terminal, only `tail` and `L` return Boolean list type. Moreover, `tail` also requires its argument to be of Boolean list type. Consequently, the subtree representing the third argument can only be constructed using `tail` and `L`. Figure 7.2 shows the program structure for `foldr`. An * on the node indicates the node is optional.

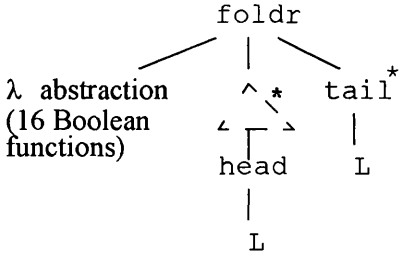


Figure 7.2: The `foldr` program tree structure.

The program structures in the search space is analyzed according to the numbers of `foldr` in the program trees. Figure 7.3 shows all the possible program structures containing 2 `foldr`. Figure 7.3(a) represents 1,536 program structures (Each of the two λ abstractions can generate 16 Boolean functions and there are 3 optional nodes in the tree). Figure 7.3 (b) represents 4,096 program structures. In total, there are 5,632 program structures containing 2 `foldrs` in the search space.

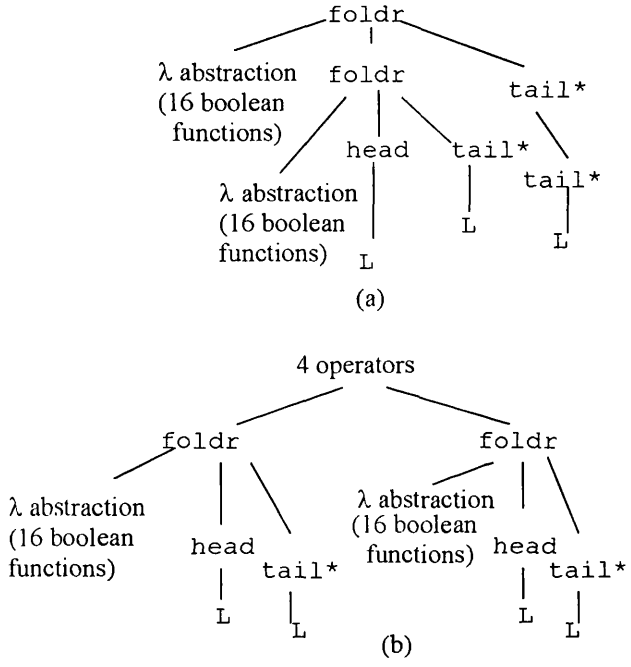


Figure 7.3: Program structures with 2 `foldrs`.

Figure 7.4 shows program structures with 1 `foldr`. The structure of Figure 7.4 (a) represents 192 programs. Figure 7.4 (b) represents 1,024 programs; Figure 7.4 (c) represents 96 programs and Figure 7.4 (d) represents 512 programs. In total, there are 1,824 program structures with 1 `foldr` in the search space.

Figure 7.5 shows program structures without `foldr`. Figure 7.5(a) represents 64 program structures. Figure 7.5(b) represents 64 program structures. Figure 7.5(c) represents 3 program structures and Figure 7.5(d) represents 16 program structures. In total, there are 147 program structures without `foldr` in the search space.

In summary, there are 7,603 program structures in the search space. Among them, 5,632 are with 2 `foldrs`; 1,824 are with 1 `foldr` and 147 are without `foldr`.

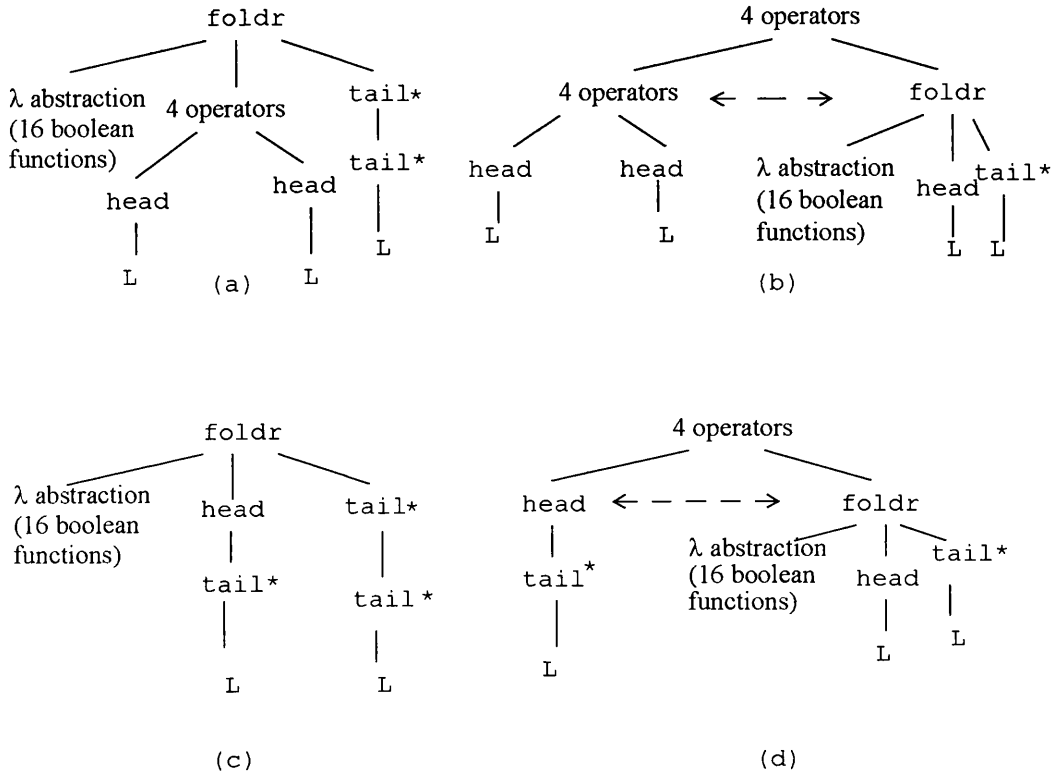


Figure 7.4: Program structures with 1 `foldr`.

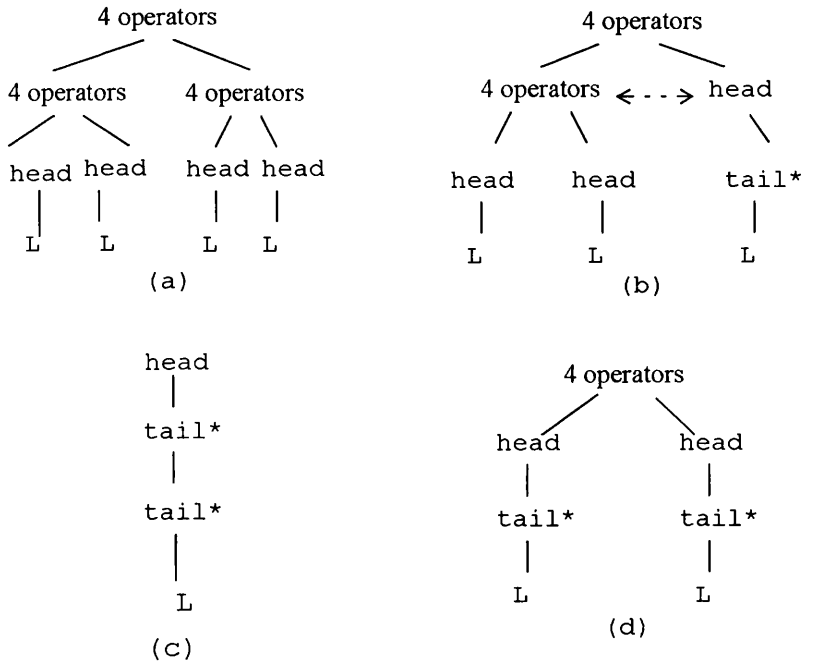


Figure 7.5: Program structures without `foldr`

7.3.1 Fitness Distribution

The 7,603 program structures in the search space are capable of solving the general `even-parity` problem in different degree. A general solution to this problem has to be able to handle any number of inputs, i.e. N can be of any value. To evaluate how well each program structure is in solving the problem, we use `even-2-parity` and `even-3-parity` as test cases (see Section 6.6.1). There are a total of $2^2 + 2^3 = 12$ test cases. A program structure receives one point for each correctly handled test case. The program which contains code causing empty-list run-time error is penalized with 0.5 (see Section 6.6.2). A perfect solution receives a fitness value of 12.

Note that there is only one program in the search space which produces empty-list run-time error: `head(tail(tail L))`. When evaluated against the test cases for the `even-2-parity` (which have 2 inputs), empty-list run-time error would occur. Since this is the only case where run-time error can happen, we believe that the constraint handling has very little impact on GP search.

Figure 7.6 shows the fitness distribution in the search space.

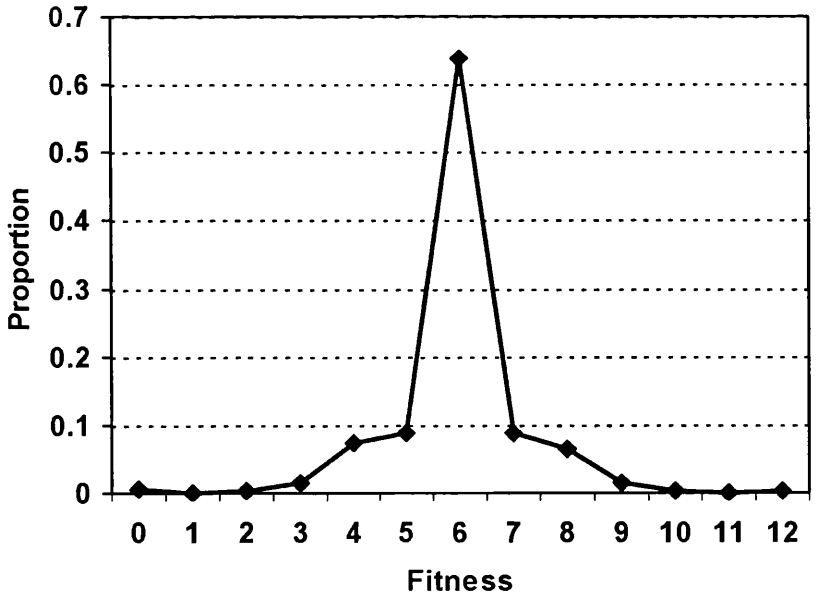


Figure 7.6: Fitness distribution in the search space.

More than 60 percent of the program structures are of fitness 6. The number of program structures with other fitness falls dramatically away either side of the peak. There are only 29 program structures which score 12 and are prefect solutions to the general `even-parity` problem. If the 16 Boolean functions are provided in the function set and they have equal probability to be selected to construct the programs, an unbiased random search could find

(on average) a solution with 262 fitness evaluations. However, this work intents to let GP discover these partial solutions to the problem. The Boolean functions are therefore evolved (as λ abstractions), not given directly in the function set. Consequently, this is a harder problem which requires more program evaluation to find the solutions.

7.4 Solutions in the Search Space

The 29 solutions in the search space are split into 11 groups (Tables 7.3 - 7.13). Programs in each group have the same structure but different λ abstraction values (Boolean functions). We will calculate the probability to find each solution based on the program structure and its Boolean functions.

To successfully find a solution, it requires to find both the correct program structure and the correct Boolean functions. Based on the analysis of program structures in Section 7.3, the probability to find each program structure can be calculated. Meanwhile, the search of the 16 Boolean functions has been studied by Koza [Koza, 1992, page 228]. For each of the 16 Boolean functions, Koza did random search using a program tree with 31 nodes. The results are summarized in Table 7.2. We use this table to measure the probability to find each of the 16 Boolean functions (Koza called them Boolean rules). For example, rule 15 can be generated by random creation of 4.8 programs. The probability of success to this rule is therefore 1/4.8.

Table 7.2: 16 Boolean rules found using random search.

Rule No.	Random Search	Rule No.	Random Search
15	4.8	13	31.9
00	4.8	11	32.0
10	7.8	04	32.0
05	7.8	03	32.0
14	28.8	02	32.1
08	28.8	12	32.2
07	28.9	09	821.0
01	29.0	06	846.0

Table 7.3 to Table 7.12 present the probability to find the 29 solutions. The caption indicates the solution. Columns 1 & 2 are the Boolean rules for the λ abstractions. Columns 3 & 4 are the probability to find each of the Boolean rules. Column 5 is the probability to find the program structure. Finally, column 6 is the probability to find the solution. It is the result of mul-

tiplying columns 3, 4 and 5. Table 7.13 is slightly different in that the solution only contains one λ abstraction. However, the method to measure the probability to find the solution is the same.

Table 7.3: foldr λ_1 (foldr λ_2 (head L) (tail L)) (tail (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r3	r9	1/32	1/821	256/7603	1.28e-6

Table 7.4: foldr λ_1 (foldr λ_2 (head L) (tail L)) L

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r3	r9	1/32	1/821	256/7603	1.28e-6
r6	r15	1/846	1/4.8	256/7603	8.29e-6

Table 7.5: foldr λ_1 (foldr λ_2 (head L) L) (tail L)

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r5	1/846	1/7.8	256/7603	5.10e-6
r9	r3	1/821	1/32	256/7603	1.28e-6

Table 7.6: foldr λ_1 (foldr λ_2 (head L) L) L

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r15	1/846	1/4.8	256/7603	8.29e-6
r6	r13	1/846	1/31.9	256/7603	1.25e-6

Table 7.7: nand (foldr λ_1 (head L) L) (foldr λ_2 (head L) (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r13	r6	1/31.9	1/846	256/7603	1.25e-6
r14	r6	1/28.8	1/846	256/7603	1.38e-6
r15	r6	1/4.8	1/846	256/7603	8.29e-6

Table 7.8: nand (foldr λ_1 (head L) (tail L)) (foldr λ_2 (head L) L)

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r13	1/846	1/31.9	256/7603	1.25e-6
r6	r14	1/846	1/28.8	256/7603	1.38e-6
r6	r15	1/846	1/4.8	256/7603	8.29e-6

Table 7.9: nand (foldrλ_1 (head L) (tail L)) (foldrλ_2 (head L) (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r6	1/846	1/846	256/7603	4.70e-8
r6	r14	1/846	1/28.8	256/7603	1.38e-6
r6	r15	1/846	1/4.8	256/7603	8.29e-6
r14	r6	1/28.8	1/846	256/7603	1.38e-6
r15	r6	1/4.8	1/846	256/7603	8.29e-6

Table 7.10: nor (foldrλ_1 (head L) (tail L)) (foldrλ_2 (head L) (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r0	r6	1/4.8	1/846	256/7603	8.29e-6
r4	r6	1/32	1/846	256/7603	1.24e-6
r6	r6	1/846	1/846	256/7603	4.70e-8
r6	r4	1/846	1/32	256/7603	1.24e-6
r6	r0	1/846	1/4.8	256/7603	8.29e-6

Table 7.11: nor (foldr λ_1 (head L) L) (foldr λ_2 (head L) (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r0	r6	1/4.8	1/846	256/7603	8.29e-6
r4	r6	1/32	1/846	256/7603	1.24e-6

Table 7.12: nor (foldr λ_1 (head L) (tail L)) (foldr λ_2 (head L) L)

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r0	1/846	1/4.8	256/7603	8.29e-6
r6	r4	1/846	1/32	256/7603	1.24e-6

Table 7.13: foldr λ_1 (fun (head L) (head L)) (tail L)

λ_1	fun	P(λ_1)	P(str)	P(s)
r6	nand	1/846	16/7603	2.48e-6
r6	nor	1/846	16/7603	2.48e-6

The probability to find a solution to the general `even-parity` problem is the summation of the probability to find each of the 29 solutions:

$$P(S) = \sum_{i=1}^{29} P(s_i) = 0.000111$$

Using the probability of finding a solution, the number of program evaluations required to find a solution can be calculated. This is the "Effort" defined in [Koza, 1992, page 194].

$$E = \frac{\log(1-z)}{\log(1-P)}, \text{ where } z = 99\%$$

$$E \approx -\frac{\log(1-z)}{P} \approx 18,000$$

When random search is performed without repetition, the effort required would be 9,000 fitness evaluation.

The results of our analysis have two important implications:

- Structure abstraction enables the general `even-parity` problem to be solved very efficiently. Related work using different techniques requires the evaluation of a much bigger number of programs before a solution can be found (see Table 7.14). Moreover, their solutions only work for a particular value of N and are not general solutions.
- Most of the effort used to find a solution is devoted to the search of the correct Boolean rules. This is based on the observation that $P(s) \approx P(\lambda_1) \cdot P(\lambda_2)$ ($P(str)$ does not effect $P(s)$ much compared to $P(\lambda_1) \cdot P(\lambda_2)$). This means that the complexity of the general `even-parity` problem is almost reduced to that of the problem of Boolean functions with two arguments.

Table 7.14: Various techniques used to solve the even-parity problem.

Techniques	N	Effort value
Standard GP [Koza, 1992]	5	6,528,000
Standard GP with ADFs [Koza, 1994]	7	1,440,000
GP with global ADFs [Aler, 1998]	6	627,200
EP with ADFs [Chellapilla, 1998]	9	586,000
Sub-machine Code GP [Poli, Page and Langdon, 1999]	22	418,600

7.5 Experiments and Results

We conduct experiments to verify our analysis. The implementations are basically the same as

that of the previous chapter except the following:

- The functions and terminals are *monomorphic* instead of polymorphic (see Table 7.1).
- Unlike polymorphic `foldr`, which allows a program to contain infinite number of `foldr` (see Section 6.6.3), the monomorphic `foldr` function can only appear at most twice in a program within the restricted tree depth 4 (see Section 7.3). Consequently, the nested recursion limit of 100 is not necessary.
- The subtrees representing λ abstractions are restricted to a depth limit of 5. This is the same as that used by Koza to run his Boolean rules experiments (see Section 7.4).
- Instead of selection nodes with bias towards the root nodes (see Section 4.4.1), we allow each external/internal node in a program tree to have an equal opportunity to be selected for crossover and mutation. Moreover, genetic operations are applied to full application nodes only. This is the normal GP implementation which allows the experiment results to reflect GP performance more accurately.

Figure 7.7 shows the performance curves based on 50 runs.

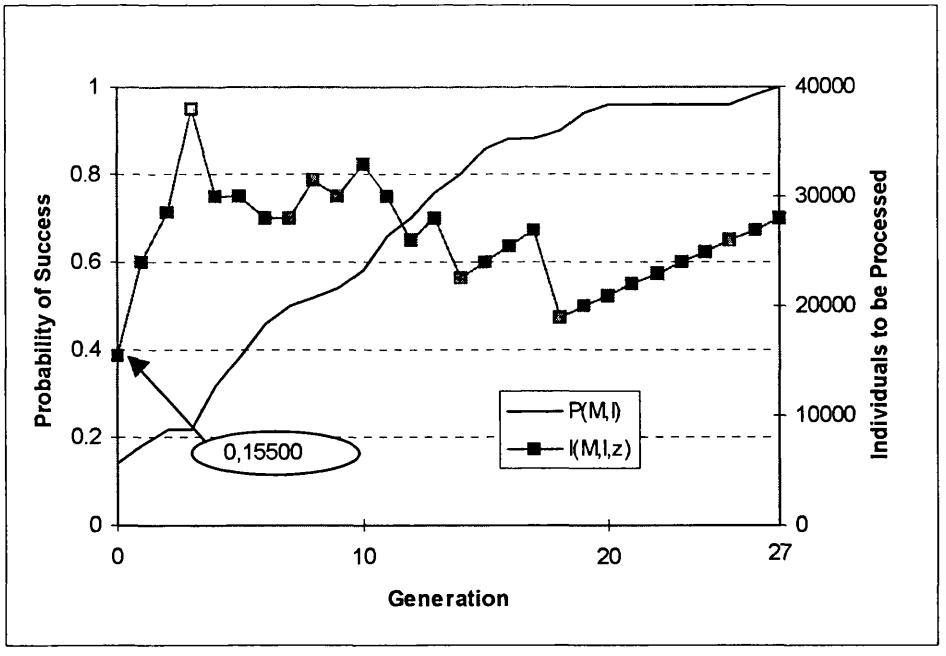


Figure 7.7: Performance curves for the general even-parity problem.

Due to the different implementations, the results are different from that in Figure 6.2. In particular, the probability of success at generation 0 is 3 times higher (7/50 vs. 2/50) in this experiment. This is a reasonable result based on our analysis in Section 6.8.1. With polymorphic implementation, it takes GP about 3 generations to identify the promising area of the

search space, i.e. that contains programs with 1, 2 or no `foldr`. In contrast, monomorphic implementation allows GP to start with a search space that contains only 1, 2 or no `foldr`. Consequently, the solutions can be found more easily at generation 0 through random search.

All 50 runs find a perfect solution. Among them, 7 runs find a solution at generation 0. The probability of success curve, P(M,i), starts as 14% at generation 0 and reaches 100% at generation 27. The "effort" curve, I(M,i,z), gives the number of program evaluations required at each generation to find a solution (see Appendix B). According to the experiment results, a solution to the general `even-parity` problem can be found by evaluating 15,500 programs which are randomly generated at generation 0. This is very close to our estimate of 18,000 program evaluations. Consequently, the 2 implications of our analysis are asserted.

7.6 Analysis and Discussion

With a program representation which supports structure abstraction, the general `even-parity` problem can be solved by random generation of 15,500 program trees. This result raises two important questions:

- Why structure abstraction makes the general `even-parity` an easy problem?
- Why random search outperforms GP search on this problem?

We address these two questions in the following subsections.

7.6.1 Impacts of Structure Abstraction

Structure abstraction has enabled the general `even-parity` problem to be solved very efficiently. Yet, this problem has a search space (the 7,603 program structures with their λ abstractions expanded with all possible subtrees) which is undoubtedly large. Moreover, the density of the solution is very low (0.38%). This suggests that the difficulty of a problem might be independent of the density of the solution in the search space. Instead, it relies on how easily these solutions can be found. Our analysis indicates that the effort required to find a solution to the problem is approximately the same as that to find the Boolean rules partial solutions. This means that the module mechanism of λ abstraction, which allows partial solutions to be evolved, is very important to the search of the solution. However, provided with partial solutions alone, [Langdon and Poli, 1998b] has shown that the generation of the overall solution is still not able to be achieved. An additional ingredient is the method to manipulate the partial solutions. The structure abstraction supported by `foldr` provides both ingredients, hence enables the solutions to be found easily:

- The bottom level of the structure abstraction hierarchy (λ abstraction) supports the evolution of partial solutions (Boolean rules).
- The top level of the structure abstraction hierarchy (`foldr` higher-order function) provides a mechanism for the manipulation of the partial solutions (reuse the Boolean rules) and the specification of the inputs order (as a list).

In other words, structure abstraction provides a mechanism of hierarchical processing in problem solving: partial solutions are evolved and manipulated to form a bigger solution. This hierarchical processing is shown to be very effective in solving the general `even-parity` problem.

7.6.2 Random Search Versus GP Search

The fitness distribution (Figure 7.6) shows that program structures with fitness 6 occupies more than 60% of the search space while program structures with other fitness are sparse. This means that the search space contains little gradient information. Like any other progressive search algorithms, GP relies on gradient information to perform search. Without such information, GP is not able to outperform random search on this problem.

However, structure abstraction does not always create this kind of search space. With different problems, we anticipate that structure abstraction will generate a search space which allows GP to shine.

7.7 Guidelines to Apply Structure Abstraction

Although an engine of hierarchical processing, structure abstraction must be applied properly in order to receive the benefits. This is the lesson learned from our experiences with the artificial ant problem (see Appendix C). We have made the first-step to formulate guidelines for the application of structure abstraction to other problems:

Design a higher-order function which provides the following:

- function arguments to allow partial solutions, represented as λ abstractions, to be evolved;
- methods to manipulate the partial solutions in constructing a bigger solution;
- specifications of the order of the inputs that the partial solutions can apply.

These guidelines will evolve as more experiences are gained.

7.8 Summary

By selecting an appropriate program representation (higher-order function `foldr`, functions `head` & `tail` and terminal `L`), we present a successful example of using an evolutionary algorithm to solve a difficult problem. The important property of such a program representation is "structure abstraction", which can be obtained by including a higher-order function in the function set. The analysis of the program search space indicates that the structure abstraction provided by `foldr` provides a mechanism of hierarchical processing for GP search. Consequently, the effort required to find a solution is reduced. The general `even-parity` problem presented is an example of good use of structure abstraction. As there is a trend in developing problem-specific evolutionary algorithms [Leonhardi, Reissenberger, Schmelmer, Weicker and Weicker, 1998], we have outlined the guidelines for the application of structure abstraction to other problems.

Chapter 8

Future Work

We have investigated the impact of three different functional programming techniques (polymorphism, implicit recursion and higher-order functions) on GP. The results show that these techniques are advantageous to GP problem solving. However, there are also open issues that need further study. This chapter highlights some of the areas that we will be following in the future.

8.1 Polymorphism

Type constraints have been applied to GP successfully by various researchers (see Section 3.2). One important motivation of this avenue of work is to enhance GP performance by the achieved reduced search space. Indeed, the reported results have confirmed that when solving problems involving multiple types, GP with type constraints performs better than the GP without (see Section 3.2). These results, however, do not provide any evidence that the performance advantage is due to the reduced search space. In fact, [Langdon and Poli, 1998a] has shown that there is no direct relationship between the size of the search space and the difficulty of a problem. Our investigation of constraint handling (see Appendix A) also shows that if the search space is not constrained properly, the search of a solution will become harder, not easier. These new results raise a couple of interesting questions for future work: why type constraints make GP search more efficient? Can we generalize these results to other problems?

8.2 Implicit Recursion

Implicit recursion can be achieved by any higher-order functions which provide recursion semantics (see Section 6.8). We have used the higher-order function `foldr` to solve the gen-

eral even-parity problem. However, there are other problems where a designed higher-order function may be required. For example, the Fibonacci sequence problem [Koza, 1992, page 473] does not have a structured input (list). Instead, the recursion is carried around a numerical value. A higher-order function which performs recursion based on a numerical input value would be more appropriate to this problem. This proposition needs farther investigation.

8.3 Higher-Order Functions

We have shown that the higher-order function program representation provides hierarchical processing for GP to solve the general even-parity problem. However, to generalize this result, this program representation needs to be tested on more problems. Moreover, the guidelines for the application of structure abstraction should evolve as more experiences are gained. Finally, to promote hierarchical processing farther, we will investigate structure abstraction with multiple-layer hierarchy.

8.4 Summary

The investigation of functional programming techniques on GP has opened a new potential research area. This chapter provides future research which stems from this work. Firs, polymorphism raises the question of why type constraints make GP search more efficient. Second, there are other forms of implicit recursion that can be explored. Finally, the claim of hierarchical processing provided by the higher-order functions program representation need to be tested against more problems. We hope to carry out these research in the near future.

Chapter 9

Summary and Conclusions

The similarity between humans and GP in their program development process has motivated this research. Indeed, Fogel has observed the "intelligence" demonstrated in the human problem solving process and proposed automating this process to create "artificial intelligence" [Fogel, 1962; Fogel, Owens and Walsh, 1966]. This is the fundamental theme of today's evolutionary algorithms.

Today, human programmers have applied and benefited from many modern functional programming techniques. We hypothesize that these techniques will also benefit evolutionary algorithms, such as GP. We have tested this hypothesis with three functional programming techniques (polymorphism, implicit recursion and higher-order functions). The results of our investigation are very encouraging. These techniques not only are applicable to GP but also have enhanced GP's applicability and efficiency.

9.1 Summary of Research

Chapter 4 presents a GP system which is developed with the functional programming techniques incorporated. Each component of the system is explained and its implementation is provided. This work demonstrates that the functional programming techniques of polymorphism, implicit recursion and higher-order-functions, are indeed applicable to GP.

In Chapter 5, we show that *polymorphism* enables GP to evolve the `map` and the `nth` programs. This has never been attempted by the standard GP because evolving these two programs requires the manipulation of multiple types and type variables. A previous non-standard GP [Montana, 1995], which does support these requirements, did evolve these two programs. However, that GP does not support function types. Our work has enhanced GP's applicability to problems that require multiple types, type variables or function types.

Recursion is an important programming technique because it provides an elegant way of

code reuse and produces programs more general than those do not use recursion. Yet, GP has not had much success in evolving recursive programs, due to various difficulties. In Chapter 6, we have not only identified these difficulties but also introduced *implicit recursion* to overcome these difficulties. Moreover, by combining implicit recursion with λ abstractions, GP is able to evolve a general solution to the `even-parity` problem very efficiently. This technique has enhanced GP performance on this problem to a degree that has never been reported by any previous work.

Higher-order functions provide a new kind of program representation for GP. This representation contains a special property which we named "structure abstraction". Chapter 7 identifies structure abstraction as an engine of hierarchical processing for GP search. It is this hierarchical processing which enables GP to solve the general `even-parity` very efficiently. This chapter provides guidelines for the application of structure abstraction to other problems.

Constraint handling can have strong impact on the evolution of problem solutions. Appendix A identifies eleven ways to handle constraints based on the general framework of the evolutionary algorithms. Five of these methods are experimented on a run-time error constraint using a GP system. The results show that an appropriate constraint method can help GP to search for solutions, whereas an inappropriate method would prevent the problem solutions to be found.

9.2 Summary of Contributions

This research makes the following contributions:

1. it constructs a formal GP framework to evolve λ-calculus expressions.
   - a single language with sufficient computation power to solve a wide variety of problems is provided in GP. The language also provides a natural integration of a module mechanism via λ abstractions (see Chapter 4).
2. it demonstrates advantages provided by applying the following functional programming techniques to GP:
   - **polymorphism**: presents the concept of types in GP in great detail through the definition of and the differentiation between *untyped*, *dynamically typed* and *strongly typed* GP. The Strongly Typed Genetic Programming (STGP) [Montana, 1995] is formalized and extended to include various kinds of type variables and higher-order function types. Moreover, the impact of different type variables on GP search space is analyzed

(see Chapter 5).

- **implicit recursion**: provides recursion semantics in the evolved programs without explicit recursive calls. Previously, evolving recursive programs in GP has been *difficult*. This work not only identifies the issues that cause such a difficulty but also provides a solution, implicit recursion, to overcome the difficulty (see Chapter 6).
- **higher-order functions**: supports an effective module mechanism for GP. In this approach, module creation is neither a random process nor determined in advance. Instead, it uses the knowledge (function type arguments) specified by the users in the higher-order functions to determine the most beneficial way to create modules. Most importantly, this work introduces a new term, *structure abstraction*, to describe the structure pattern emerging from the higher-order functions program representation. Structure abstraction not only enables GP to evolve a general solution to the `even-parity` problem but also achieve greater efficiency than any other previous work (see Chapter 6).

3. it identifies structure abstraction as a hierarchical processing engine for GP search. The guidelines for the application of structure abstraction to other problems are outlined (see Chapter 7).
4. it presents a concept of constraints handling based on the general framework of evolutionary algorithms (see Appendix A). This general approach provides an easy way to compare and contrast different constraint handling methods, e.g. dynamic typing versus strong typing (see Chapter 5). Moreover, the seesaw effect demonstrated in the experiments gives a high level view of the impact of constraint handling on the evolutionary process, e.g. see the constraint handling for recursion error in Chapter 5.

9.3 Conclusions

With the combination of its GA heritage and its unique interesting features (see Chapter 2), GP has attracted many researchers and practitioners as a promising paradigm to solve complex real-world problems. However, as efforts are made to push the field toward this goal, many fundamental questions arise. The journey of this research is an example of such a process, and it reflects the current state of art in the field.

Functional programming techniques, like other techniques used in GP, were proposed to make GP more powerful. Polymorphism extends previous work in applying type constraints to reduce search space (see Chapter 5). Intuitively, a smaller search space would allow a solu-

tion to be found faster. This conjecture has been confirmed by a number of researchers (see Section 3.2). However, a recent work has demonstrated clearly that the size of program search space is not necessarily an indication of the difficulty of a problem [Langdon and Poli, 1998a]. Moreover, our investigation of constraint handling methods indicates that if the search space is not constrained properly, the search for a solution will become harder, not easier (see Appendix A). These new discoveries raise a new question: Why does strong typing make GP search more efficient? This question touches the fundamental issue of how GP performs search.

Koza expects the search process of GP to be explained by following the GA schema theorem (he used the term "schema" to describe the GP sampling process). However, Chapter 2 shows that this approach is not able to describe the complex GP evolutionary process, where both the structure and the contents of the programs are evolved at the same time. The only attempt at describing the GP search process using a schema theorem is made by [Poli and Langdon, 1998a; Poli and Langdon, 1998c]. They described the evolutionary process of GP with one-point crossover as having 2 stages. In the first stage, competitions are between programs with different structures (program contents are ignored). Once the program structure in the population is settled, the second stage starts, and the competitions are between programs with the same structure but different contents. (This is very much like competition in GAs). Yet, as the validity of the GA schema theorem is under attack (see Chapter 2), this view of GP search process, which relies on the GA schema theorem, requires more work to be helpful [Poli, 1999]. The search for the interpretation of GP search process continues.

Higher-order functions, which provide module creation and reuse, were proposed to enhance GP's ability to scale up to larger and more complex problems. Indeed, this approach has enabled GP to solve the general `even-parity` problem very efficiently (see Chapter 6). Further investigation indicates that the power of the higher-order functions program representation stems from the hierarchical processing it provides (see Chapter 7). This hierarchical processing was hoped to be provided by the GP search operator, subtree crossover (i.e. building block hypothesis). Yet, [Langdon and Poli, 1998b] has demonstrated that provided with building blocks, the subtree crossover operator is not able to construct solutions for the `even-parity` problem using these building blocks. This raises other important questions: How does the subtree crossover operator perform search? Is it an appropriate search engine for GP?

We have considered these questions from the aspect of programming languages and their interpretation. A program's behavior depends on its constructs (functions and terminals) as well as the global environment in which its evaluation is conducted (the program input value).

Additionally, the behavior of each construct depends on its local environment: the argument values that are provided to the function and the value that a terminal is bound to. In GP program tree representation, the arguments to a function are results produced by the function's surrounding constructs. In biological terms, we say the genes (program constructs) have very high epistasis. The subtree crossover operator does not consider the surrounding environment of the two swapped subtrees. Consequently, the linkage between genes which produce good fitness are very likely to be destroyed. The syntax-based crossover operator is therefore more destructive than constructive in building fitter programs. Following previous wisdom [Olsson, 1995; Whigham, 1996b], we believe that semantics-based genetic operators, where the meaning of the functions and terminals are considered for the application of genetic operators, are better search strategies for GP.

Our investigation of the impact of functional programming techniques on GP has produced many positive results. Polymorphism has enhanced GP applicability to problems which require multiple types, type variables or function types, e.g. the `nth` and `map` problems. Implicit recursion introduces a more effective way for GP to evolve recursive programs. Higher-order functions generate a new program representation (structure abstraction) which provides hierarchical processing for GP search. The limitation of these techniques are also discussed. For example, polymorphism may become an overhead when the type variables are not used with care; the design of a suitable higher-order function to provide effective implicit recursion or hierarchical processing for the target problem might not be easy. Meanwhile, some fundamental questions are raised by these results. How does GP perform its search for problem solutions? What kind of genetic operators provide good search strategy for GP? These questions are vital to the advance of this field. Only when the operation of GP is understood it can be applied effectively to solve complex real-world problems.

Bibliography

[Aler, 1998] R. Aler. Immediate transference of global improvements to all individuals in a population in genetic programming compared to automatically defined functions for the even-5-parity problem. *Proceedings of the First European Workshop on Genetic Programming*. W. Banzhaf, R. Poli, M. Schoenauer and T. Fogarty (eds.). Paris, France, pages 14-15, 1998. Springer-Verlag.

[Altenberg, 1994a] L. Altenberg. Emergent phenomena in genetic programming. *Proceedings of the Third Annual Conference on Evolutionary Programming*. A. V. Sebald and L. J. Fogel (eds), pages 233-241, San Diego, CA, pages 233-241, 1994. World Scientific.

[Altenberg, 1994b] L. Altenberg. The evolution of evolvability in genetic programming. *Advances in Genetic Programming*, K. E. Kinnear, Jr. (ed.). MIT Press, Cambridge, MA, pages 47-74, 1994.

[Altenberg, 1995] L. Altenberg. The schema theorem and price's theorem. *Foundations of Genetic Algorithms* 3, L. D. Whitley and M. D. Vose (eds.), pages 23-49, Estes Park, Colorado, 1995. Morgan Kaufmann.

[Andre and Teller, 1997] D. Andre and A. Teller. A study in program response and the negative effects of introns in genetic programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 12-20, Stanford University, CA, 1997. MIT Press.

[Andre, 1994] D. Andre. Evolution of map making: learning, planning, and memory using genetic programming. *Proceedings of the First IEEE Conference on Evolutionary Computation*, Vol. 1, pages 250-255, Orlando, Florida, 1994. IEEE Press.

[Angeline and Pollack, 1992] P. J. Angeline and J. Pollack. The evolutionary induction of subroutines. *The Fourteenth Annual Conference of the Cognitive Science Society*, pages 236-241, Bloomington, Indiana, 1992. Lawrence Erlbaum.

[Angeline and Pollack, 1993] P. J. Angeline and J. Pollack. Evolutionary module acquisition. *Proceedings of t he Second Annual Conference on Evolutionary Programming*, D. B. Fogel and W. Atmar (eds.), La Jolla, CA, pages 154-163, 1993. Evolutionary Program-

ming Society.

[Angeline, 1994] P. J. Angeline. Genetic programming and emergent intelligence. *Advances in Genetic Programming*, K. E. Kinnear, Jr. (ed.). MIT Press, Cambridge, MA, pages 75-98, 1994.

[Angeline, 1997a] P. J. Angeline. Comparing subtree crossover with macromutation. *Evolutionary Programming VI: Proceedings of the Sixth Annual Conference on Evolutionary Programming.* P. J. Angeline, R. G. Reynolds, J. R. McDonnell and R. Eberhart (eds), pages 101-112, Indianapolis, Indiana, 1997. Springer-Verlag.

[Angeline, 1997b] P. J. Angeline. Subtree crossover: building block engine or macromutation?". *Genetic Programming 1997: Proceedings of the Second Annual Conference.* J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 9-17, Stanford University, CA, 1997. MIT Press.

[Angeline, 1998] P. J. Angeline. Subtree crossover causes bloat. *Genetic Programming 1998: Proceedings of the Third Annual Conference.* J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 745-752, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Bäck, Hammel and Schwefel, 1997] T. Bäck, U. Hammel and H. -P. Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, Vol. 1:1, pages 3-17, 1997.

[Bäck, 1996] T. Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, NY, 1996.

[Bal and Grune, 1997] H. E. Bal and D. Grune. *Programming Language Essentials.* Addison-Wesley, England, 1997.

[Banzhaf, 1994] W. Banzhaf. Genotype-phenotype-mapping and neutral variation - a case study in genetic programming. *Parallel Problem Solving From Nature, 3.* Y. Davidor, H-P Schwefel, and R. Manner (eds.), pages 322-332, Jerusalem, Israel, 1994. Springer-Verlag.

[Banzhaf, Francone and Nordin, 1997] W. Banzhaf, F. Francone and P. Nordin. On some emergent properties of variable size evolutionary algorithms. Position Paper a the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97.

[Barnes, 1994] J. G. Barnes. *Programming in Ada.* 4th edition. Addison-Wesley, Reading, MA, 1994.

[Bentley and Wakefield, 1997] P. J. Bentley and J. P. Wakefield. Finding acceptable solutions

in the pareto-optimal range using multiobjective genetic algorithms. *Soft Computing in Engineering Design and Manufacturing*. P. K. Chawdhry, R. Roy, and R.K. Pant (eds). Springer-Verlag London Limited, Part 5, pages 231-240. 1997.

[Beyer, 1995] H.-G. Beyer. Towards a theory of evolution strategies: on the benefits of sex-the ($\mu/\mu,\lambda$) theory. *Evolutionary Computation*, 3(1):81-111, 1995.

[Beyer, 1997] H.-G. Beyer. An alternative explanation for the manner in which genetic algorithms operate. *BioSystems*. 41:1-15, 1997.

[Böhm, 1985] H. J. Böhm. Partial polymorphic type inference is undecidable. *Proceedings of 26th Symposium on Foundations of Computer Science*. IEEE, pages 339-345, 1985.

[Brave, 1996] S. Brave. Evolving recursive programs for tree search. *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.). MIT Press, Cambridge, MA, pages 203-219, 1996.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, Vol. 17:4, pages 471-522, 1985.

[Cardelli, 1987] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*. Vol. 8, pages 147-172, 1987.

[Chellapilla, 1997a] K. Chellapilla. Evolutionary programming with tree mutations: evolving computer programs without crossover. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 431-438, Stanford University, CA, 1997. Morgan Kaufmann.

[Chellapilla, 1997b] K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*. 1(3): 209-216, September, 1997.

[Chellapilla, 1998] K. Chellapilla. A preliminary investigation into evolving modular programs without subtree crossover. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, D. Goldberg, H. Iba and R. L. Riolo (eds.), pages 23-31, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Chomsky, 1956] N. Chomsky. Three models for the description of languages. *IRE Transactions on Information Theory*. IT-2:3, pages 113-124, 1956.

[Church and Rosser, 1936] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of American Mathematical Society*. Vol. 39, pages 472-482, 1936.

[Church, 1932-1933] A. Church. A set of postulates for the foundation of logic. *Ann. Math.* 2, pages 33-34, 346-366, 839-864, 1932-1933.

[Church, 1941] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941.

[Clack, Myers and Poon, 1995] C. Clack, C. Myers and E. Poon. *Programming with Miranda.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[Clack and Yu, 1997] C. Clack and T. Yu. Performance enhanced genetic programming. *Evolutionary Programming VI: Proceedings of the Sixth Annual Conference on Evolutionary Programming*, P. J. Angeline, R. Reynolds, J. McDonnell and R. Eberhart (eds.), pages 87-100, Prude University, Indianapolis, Indiana, 1997. Springer-Verlag.

[Cox, Davis, and Qiu, 1991] A. L. Cox Jr., L. Davis and Y. Qiu. Dynamic anticipatory routing in circuit-switched telecommunications networks. *Handbook of Genetic Algorithms.* L. Davis, (ed.), Van Nostrand Reinhold, NY, pages 124-143, 1991.

[Davis, 1985] L. Davis. Applying adaptive algorithms to epistatic domains. *Proceedings of the International Joint Conference on Artificial Intelligence.* A. Joshi (ed.), Morgan Kaufmann, CA, pages 61-69, 1985.

[De Jong and Spears, 1990] K. De Jong and W. M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. *Parallel Problem Solving from Nature-Proceedings of the First Workshop.* H.-P. Schwefel and R. Manner (eds.), pages 38-47, Dortmund, Germany, 1990. Springer-Verlag.

[Dessi, Giani and Starita, 1999] A. Dessi, A. Giani and A. Starita. An analysis of automatic subroutine discovery in genetic programming. *Proceedings of the Genetic and Evolutionary Computation Conference*, W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Hanavar, M. Jakiela and R. Smith (eds.), Morgan Kaufmann, CA, page 996-1001, 1999.

[Eshelman, Caruna and Schaffer, 1989] L. J. Eshelman, R. Caruna, and J. D. Schaffer. Biases in the crossover landscape. *Proceedings of the Third International Conference on Genetic Algorithms*, J. D. Schaffer (ed.), pages 10-19, George Mason University, DC, 1989. Morgan Kaufmann.

[Fogel, Angeline and Bäck, 1996] L. Fogel, P. J. Angeline, and T. Bäck. *Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming.* MIT Press, Cambridge, MA, 1996.

[Fogel, Owens and Walsh, 1966] L. J. Fogel, A. J. Owens and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, New York, NY, 1966.

[Fogel and Fogel, 1995] G. B. Fogel and D. B. Fogel. Continuous evolutionary programming: analysis and experiments, *Cybernetics and Systems*, Vol. 26, pages 79-90. 1995.

[Fogel and Ghozeil, 1998] D. B. Fogel and A. Ghozeil. The schema theorem and the misallocation of trials in the presence of stochastic effects. *Evolutionary Programming VII: Proceedings of the 7th Annual Conference on Evolutionary Programming*, V. W. Porto, N. Saravanan, D. E. Waagen, and A. E. Eiben (eds.), pages 313-321, San Diego, CA, 1998. Springer-Verlag.

[Fogel, 1962] L. J. Fogel. Autonomous automata, *Industrial Research*, Vol. 4, pages 14-19, 1962.

[Fogel, 1995] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ: IEEE Press, 1995.

[Fogel, 1998] D. B. Fogel. *Evolutionary Computation: the Fossil Records*. Piscataway, NJ: IEEE Press, 1998.

[Freeman, 1998] J. J. Freeman. A linear representation for GP using context free grammars. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 72-77, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Gathercole and Ross, 1996] C. Gathercole and P. Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. *Genetic Programming 1996: Proceedings of the First Annual Conference*. J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo (eds.), pages 291-296, Stanford University, CA, 1996. MIT Press.

[Gathercole and Ross, 1997] C. Gathercole and P. Ross. Tackling the boolean even n parity problem with genetic programming and limited-error fitness. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 119-127, Stanford University, CA, 1997. Morgan Kaufmann.

[Gero and Kazakov, 1998] J. S. Gero and V. A. Kazakov. Evolving design genes in space layout planning problems, *Artificial Intelligence in Engineering*, 1998.

[Goldberg, 1989] D. E. Goldberg. *Genetic Algorithms, in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[Goldberg, Deb and Korb, 1990] D. Goldberg, K. Deb and B. Korb. Messy genetic algorithms revisited: studies in mixed size and scale. *Complex Systems* 4(4):415-444. 1990.

[GP-List, 1997] Genetic programming e-mail list: genetic-programming@cs.stanford.edu, 1997.

[GP-List, 1998] Genetic programming e-mail list: genetic-programming@cs.stanford.edu, 1998.

[Grefenstette and Baker, 1989] J. J. Grefenstette and J. E. Baker. How genetic algorithms work: a critical look at implicit parallelism. *Proceedings of the Third International Conference on Genetic Algorithms.* J. D. Schaffer (ed.), pages 20-27, George Mason University, DC, 1989. Morgan Kaufmann.

[Gruau and Whitley, 1995] F. Gruau and D. Whitley, A programming language for artificial development, *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming.* J. R. McDonnell, R. G. Reynolds, and D. B. Fogel (eds.). pages 415-434, San Diego, CA, 1995. MIT Press.

[Gruau, 1996] F. Gruau. On using syntactic constraints with genetic programming. *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pages 377-394, 1996.

[Handley, 1994] S. G. Handley. The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. *Advances in Genetic Programming.* K. E. Kinnear, Jr. (ed.). MIT Press, Cambridge, MA, pages 391-401, 1994.

[Harland, 1984] D. Harland. *Polymorphic Programming Languages Design and Implementation.* Ellis Horwood Limited, England. 1984.

[Harries and Smith, 1997] Exploring alternative operators and search strategies in genetic programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference.* J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), page 147-155, Stanford University, CA, 1997. Morgan Kaufmann.

[Harris, 1997] C. Harris. Strongly typed genetic programming to promote hierarchy through explicit syntactic constraints. *Late Breaking Papers at the Genetic Programming 1997 Conference.* J. R. Koza (ed.). pages 72-80, Stanford University, CA, 1997. Stanford University.

[Haynes, Schoenefeld and Wainwright, 1996] T. D. Haynes, D. A. Schoenefeld and R. L. Wainwright. Type inheritance in strongly typed genetic programming. *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pages 359-376, 1996.

[Haynes, Wainwright, Sen and Schoenefeld, 1995] T. D. Haynes, R. Wainwright, S. Sen, and

D. Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. Eshelman (ed.),pages 271-278, Pittsburgh, PA, 1995. Morgan Kaufmann.

[Haynes, 1996] T. Haynes, Duplication of coding segments in genetic programming. *Proceedings of the Thirteenth National Conference on Artificial Intelligence.* pages. 344-349, Portland, OR, 1996.

[Haynes, 1998] T. Haynes. Perturbing the representation, recoding, and evaluation of chromosomes. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 122-127, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Hindley, 1969] R. Hindley. The principle type scheme of an object in combinatory logic. *Transactions of American Mathematical Society,* Vol. 146, pages 29-60, 1969.

[Hinterding and Michalewicz, 1998] R. Hinterding and Z. Michalewicz. Your brains and my beauty: parent matching for constrained optimization, *Proceedings of the Fifth International Conference on Evolutionary Computation*, page 4-9, Anchorage, Alaska, 1998. IEEE Press.

[Holland, 1973] J. H. Holland. Genetic algorithms and the optimal allocation of trails. *SIAM Journal on Computation*, 2:88-105, 1973.

[Holland, 1992] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, 1992. First published by Michigan Press, 1975.

[Hopcroft and Ullman, 1979] J. Hopcroft and J. Ullman. *Introduction to automata theory, language and computation*. Addison-Wesley, 1979.

[Hudak, Peterson and Fasel, 1997] P. Hudak, J. Peterson and J. H. Fasel. *A gentle introduction to Haskell.* version 1.4. http://haskell.org/tutoral/index.html, 1997.

[Hudak, 1989] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, Vol. 21:3, pages 359-411, 1989.

[Igel and Chellapilla, 1999] C. Igel and K. Chellapilla. Fitness distributions: tools for designing efficient evolutionary computations. *Advances in Genetic Programming III*, L. Spector, W. B. Langdon, U.-M. O'Reilly and P. J. Angeline (eds.). MIT Press, Cambridge, MA, pages 191-216, 1999.

[Janikow and DeWeese, 1998] C. Z. Janikow and S. DeWeese. Processing constraints in

genetic programming with CGP2.1. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 173-180, University of Wisconsin, Madison, Wisconsin, 1998, Morgan Kaufmann.

[Janikow, 1996] C. Z. Janikow. A methodology for processing problem constraints in genetic programming. *Computers and Mathematics with Application*, Vol. 32 No. 8, pages 97-113, 1996.

[Jefferson, *et al*, 1992] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor and A. Wang. Evolution as a theme in artificial life: the genesys/tracker system. *Artificial Life II*, C. G. Langton, C. Taylor, J. Farmer and S. Rasmussen (eds.), pages 549-578, Santa Fe Institute, New Mexico, 1992. Addison-Wesley.

[Kalmanson, 1986] K. Kalmanson, *An Introduction to Discrete Mathematics and its Applications*. Addison Wesley, 1986.

[Keller and Banzhaf, 1996] R. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. *Genetic Programming 96: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, R. L. Riolo (eds), pages 116-122, Stanford University, CA, 1996. MIT Press.

[Kernighan and Ritchie, 1988] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, New York, 1988.

[Kinnear Jr., 1993] Evolving a sort: lessons in genetic programming. *Proceedings of the 1993 International Conference on Neural Networks*, Vol. 2, San Francisco, CA, 1993. IEEE Press.

[Kinnear Jr., 1994a] K. E. Kinnear Jr. Alternatives in automatic function definition: a comparison of performance. *Advances in Genetic Programming*. K. E. Kinnear, Jr.(ed.), MIT Press, Cambridge, MA, pages. 119-141, 1994.

[Kinnear Jr., 1994b] K. E. Kinnear Jr. Fitness landscapes and difficulty in genetic programming. *Proceedings of the 1994 IEEE World Conference Computational Intelligence*, pages 142-147, Orlando, Florida, 1994. IEEE Press.

[Koza, Andre, Bennett III and Keane, 1996] J. R. Koza, D. Andre, F. H. Bennett III and M. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. *Genetic Programming 1996: Proceedings of the First Annual Conference*. J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo (eds.). pages 132-149, Stanford University, CA, 1996. MIT Press.

[Koza and Andre, 1996] J. R. Koza and D. Andre. Evolution of iteration in genetic programming. *Evolutionary Programming V: Proceedings of the Fifth Annual Conference on Evolutionary Programming*. L. J. Fogel, P. J. Angeline and T. Bäck (eds). pages 469-478, San Diego, CA, 1996. MIT Press.

[Koza, 1989] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. *Proceedings of the 11th International Conference on Artificial Intelligence*. N. S. Sridharan (ed). Vol. I, pages 768-774, Detroit, MI, 1989. Morgan Kaufmann.

[Koza, 1990] J. R. Koza. Genetically breeding populations of computer programs to solve problems in Artificial Intelligence. *Proceedings of the Second International Conference on Tools for AI*. pages 819-827, Herndon, Virginia, 1990. IEEE Computer Society Press.

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

[Koza, 1993] J. R. Koza. Simultaneous discovery of reusable detectors and subroutines using genetic programming. *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest (ed), page 295-302, University of Illinois at Urbana-Champaign, IL, 1993. Morgan Kaufmann.

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

[Koza, 1994b] J. R. Koza. Scalable learning in genetic programming using automatic function definition. *Advances in Genetic Programming*, K. E. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA, pages 99-117, 1994.

[Koza, 1995] J. R. Koza. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations, *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. J. R. McDonnell, R. G. Reynolds, and D. B. Fogel (eds.). pages 695-717, San Diego, CA, 1995. MIT Press.

[Langdon and Poli, 1997a] W. B. Langdon and R. Poli. An analysis of the max problem in genetic programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 222-230, Stanford University, CA, 1997. MIT Press.

[Langdon and Poli, 1997b] W. B. Langdon and R. Poli. Fitness causes bloat. *Soft Computing in Engineering Design and Manufacturing*. P. K. Chawdhry, R. Roy and R. K. Pant (eds.). Springer-Verlag, pages 23-27, 1997.

[Langdon and Poli, 1998a] W. B. Langdon and R. Poli. Why ants are hard. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 193-201, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann,

[Langdon and Poli, 1998b] W. B. Langdon and R. Poli. Why "building blocks" don't work on parity problems. Technical report CSRP-98-17. The University of Birmingham, 1998.

[Langdon, 1996] W. B. Langdon. *Data Structures and Genetic Programming*. Ph.D. thesis, University College London, 27 September 1996.

[Lenat, 1984] D. Lenat. The role of heuristics in learning by discovery: three case studies. *Machine Learning: An Artificial Intelligence Approach,* Michalske, R., Carbonell, J., and Mitchell, T. (eds.), Chapter 9, Springer-Verlag, pages 243-306, 1984.

[Leonhardi, Reissenberger, Schmelmer, Weicker and Weicker, 1998] A. Leonhardi, W. Reissenberger, T. Schmelmer, K. Weicker and N. Weicker. Development of problem-specific evolutionary algorithms. *Fifth International Conference on Parallel Problem Solving from Nature*. A. E. Eiben, T. Bäck, M. Schoenauer and H.-P. Schwefel (eds.). pages 388-397, Amsterdam, Netherlands, 1998, Springer.

[Levenick, 1991] J. Levenick. Inserting introns improves genetic algorithm success rate: taking a cue from biology. *Proceedings of the Fourth International Conference on Genetic Algorithms,* R. K. Belew and L. B. Booker (eds), pages 123-127, University of California - San Diego, CA, 1991. Morgan Kaufmann.

[Lucier, Mamillapalli and Palsberg, 1998] B. J. Lucier, S. Mamillapalli and J. Palsberg. Program optimization for faster genetic programming. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.). pages 202-207, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Macready and Wolpert, 1996] W. G. Macready and D. H. Wolpert, On 2-armed gaussian bandits and optimization. Technical Report SFI-TR-96-03-009. Santa Fe Institute, Santa Fe, NM. 1996.

[Macready and Wolpert, 1998] W. G. Macready and D. H. Wolpert. Bandit problems and the exploration/exploitation trade-off. *IEEE Transactions on Evolutionary Computation*, Vol. 2 No. 1, pages 1-23, April, 1998.

[McDonnell, Reynolds and Fogel, 1995] J. R. McDonnell, R. G. Reynolds and D. B. Fogel.

Evolutionary Programming IV, Proceedings of the Fourth Annual Conference on Evolutionary Programming. MIT Press, 1995.

[McPhee, Hopper and Reierson, 1998] N. F. McPhee, N. J. Hopper and M. L. Reierson. Impact of types on essentially typeless problems in GP. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 232-240, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Maxwell, 1994] S. R. Maxwell. Experiments with a coroutine model for genetic programming. *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Vol. 1, pages 413-417, Orlando, Florida, 1994. IEEE Press.

[Maxwell, 1996] S. R. Maxwell. Why might some problems be difficult for genetic programming to find solutions? *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 125-128, Stanford University, CA, 1996. Stanford University Bookstore.

[Michalewicz, Dasgupta, Le Riche and Schoenauer, 1996] Z. Michalewicz, D. Dasgupta, R. G. Le Riche and M. Schoenauer, Evolutionary algorithms for constrained engineering problems, *Computers & Industrial Engineering Journal*, Vol.30, No.2, September, pages 851--870, 1996.

[Michalewicz and Michalewicz, 1995] Z. Michalewicz and M. Michalewicz. Pro-life versus pro-choice strategies in evolutionary computation techniques. Chapter 10, *Evolutionary Computation*, IEEE Press, 1995.

[Michalewicz and Schoenauer, 1996] Z. Michalewicz, M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems, *Evolutionary Computation*, Vol. 4 No. 1, pages 1-32, 1996.

[Michalewicz, 1995a] Z. Michalewicz. Genetic algorithms, numerical optimization and constraints, *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. Eshelman (ed.), pages 151-158, Pittsburgh, PA, 1995. Morgan Kaufmann.

[Michalewicz, 1995b] Z. Michalewicz. A survey of constraint handling techniques in evolutionary computation methods, *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. J. R. McDonnell, R. G. Reynolds, and D. B. Fogel (eds.). pages 135-155, San Diego, 1995. MIT Press.

[Milner, 1978] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, Vol. 17, pages 348-375, 1978.

[Mitchell, 1996] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, Cam-

bridge, MA. 1996.

[Montana, 1995] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, Vol. 3:2, pages 199-230, 1995.

[Muhlenbein, 1991] H. Muhlenbein. Evolution in time and space-the parallel genetic algorithm. *Foundations of Genetic Algorithms,* G. J. E. Rawlins (ed.). pages 316-338, Indiana University, Bloomington, Indiana, 1991. Morgan Kaufmann.

[Nordin, Francone and Banzhaf, 1995] P. Nordin, F. Francone and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications.* J. Rosca (ed.), pages 6-22, Tahoe City, CA, 1995.

[Olsson, 1995] J. R. Olsson. Inductive functional programming using incremental program transformation, *Artificial Intelligence*, volume 74, number 1, March 1995, Pages. 55-83.

[O'Reilly and Oppacher, 1995] U.-M. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. *Foundations of Genetic Algorithms.* L. D. Whitley and M. D. Vose (eds.), pages 73-88, Estes Park, Colorado, 1995. Morgan Kaufmann.

[O'Reilly and Oppacher, 1996] U.-M. O'Reilly and F. Oppacher. A Comparative Analysis of Genetic Programming. *Advances in Genetic Programming II,* P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pages 23-44, 1996.

[Paterson and Wegman, 1978] M. S. Paterson and M. N. Wegman. Linear unification. Journal of Computer and System Sciences. 16:158-167, 1978.

[Perkis, 1994] T. Perkis. Stack-based genetic programming. *Proceedings of the IEEE Conference on Evolutionary Computation*, D. B. Fogel (ed.), pages 148-153, Los Alamitos, CA, 1994. IEEE Press.

[Peterson and Hammond, 1997] J. Peterson and K. Hammond (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science. 1997.

[Peyton-Jones, 1987] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International. 1987.

[Pfenning, 1988] F. Pfenning. Partial polymorphic type inference and higher-order unification. *Proceeding 1988 ACM Conference on Lisp and Functional Programming.* pages

153-163. 1988.

[Poli, Page and Langdon, 1999] R. Poli, J. Page and W. B. Langdon. Smooth uniform crossover, sub-machine code GP and demes: a recipe for solving high-order boolean parity problem. *Proceedings of the Genetic and Evolutionary Computation Conference*, W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela and R. Smith (eds.), pages 1162-1169, Orlando, Florida. 1999. Morgan Kaufmann.

[Poli and Langdon, 1997] R. Poli and W. B. Langdon. A new schema theorem for genetic programming with one-point crossover and point mutation. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 278-285, Stanford University, CA, 1997. MIT Press.

[Poli and Langdon, 1998a] R. Poli and W. B. Langdon. A review of theoretical and experimental results on schemata in genetic programming. *Proceedings of the First European Workshop on Genetic Programming*. W. Banzhaf, R. Poli, M. Schoenauer and T. Fogarty (eds.), pages 1-15, Paris, France, 1998. Springer-Verlag.

[Poli and Langdon, 1998b] R. Poli and W. B. Langdon. On the search properties of different crossover operators in genetic programming. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.). pages 293-301, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Poli and Langdon, 1998c] R. Poli and W. B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation, *Evolutionary Computation Journal*, 6(3): 231-252, 1998.

[Poli, 1996] R. Poli. Parallel distributed genetic programming. Technical report CSRP-96-15. The University of Birmingham, 1996.

[Poli, 1999] R. Poli. Schema theorem without expectations for GP and GAs with one-point crossover in the presence of schema creation. *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*. A. S. Wu (editor). page 61-63. Orlando, Florida, 1999.

[Price, 1970] G. R. Price. Selection and covariance. *Nature*, 227:520-521, August 1, 1970.

[Radcliffe, 1992] N. J. Radcliffe. Non-linear genetic representations. *Parallel Problem Solving From Nature, 2.* R. Manner and B. Manderick (eds.), pages 259-268, Amsterdam, North Holland. 1992.

[Reynolds, Michalewicz and Cavaretta, 1995] R. G. Reynolds, Z. Michalewicz, and M. J. Cavaretta, Using cultural algorithms for constraint handling in GENOCOP, *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. J. R. McDonnell, R. G. Reynolds, and D. B. Fogel (eds.), pages 289-305, San Diego, CA, 1995. MIT Press.

[Reynolds, 1993] C. Reynolds. An evolved, vision-eased behavioral model of coordinated group motion. *From Animals to Animates II. Proceedings of the Second International Conference on Simulation of Adaptive Behavior.* J. A. Meyer, H. C. Roitblat and S. W. Wilson (eds.). Cambridge, MA, MIT Press. 1993.

[Robinson, 1965] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM.* Vol. 12:1, pages 23-49, January, 1965.

[Rosca and Ballard, 1994] J. P. Rosca and D. H. Ballard. Hierarchical self-organization in genetic programming. *Proceedings of the Eleventh International Conference on Machine Learning*. pages 251-258, 1994, Morgan Kaufmann.

[Rosca and Ballard, 1996] J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. *Advances in Genetic Programming II,* P. J. Angeline and K. E. Kinnear, Jr. (eds.), pages 177-201, MIT Press, 1996.

[Rosca, 1995] J. Rosca. Genetic programming exploratory power and the discovery of functions. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. J. R. McDonnell, R. G. Reynolds and D. B. Fogel (eds). pages 719-736, San Diego, CA, 1995. MIT Press.

[Rosca, 1996] J. Rosca. Towards automatic discovery of building blocks in genetic programming. *Working Notes for the AAAI Symposium on Genetic Programming*. E. V. Siegel and J. R. Koza (eds.). pages 78-85, MIT, Cambridge, 1996. AAAI.

[Rosca, 1997a] J. P. Rosca. Analysis of complexity drift in genetic programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo (eds.), pages 286-294, Stanford University, 1997. Morgan Kaufmann.

[Rosca, 1997b] J. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms.* Ph.D. Thesis. University of Rocheste, Rochester, NY 14627, 1997.

[Rosser, 1982] J. B. Rosser. Highlight of the history of the lambda calculus. *Proceedings 1982 ACM Conference on LISP and Functional Programming.* ACM, pages 216-225, 1982.

[Rudolph, 1997] G. Rudolph, Reflections on bandit problems and selection methods in uncertain environments. *Proceedings of Seventh International Conference on Genetic Algorithms*, T. Bäck (ed.). pages 166-173, Michigan State University, East Lansing, MI, 1997. Morgan Kaufmann.

[Salustowicz and Schmidhuber, 1997] R. Salustowicz and J. Schmidhuber, Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2), pages 123-141, 1997.

[Salomon, 1997] R. Salomon. Raising theoretical questions about the utility of genetic algorithms. *Evolutionary Programming VI: Proceedings of the Sixth Annual Conference on Evolutionary Programming*, P. J. Angeline, R. Reynolds, J. McDonnell and R. Eberhart (eds.), pages 275-284, Prude University, Indiana, 1997. Springer-Verlag.

[Salomon, 1998] R. Salomon. Short notes on the schema theorem and the building block hypothesis in genetic algorithms. *Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming.* V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben (eds). pages 113-124, San Diego, CA, 1998. Springer-Verlag.

[Schoenauer and Michalewicz, 1997] M. Schoenauer and Z. Michalewicz. Boundary operators for constrained parameter optimization problems, *Proceedings of the Seventh International Conference on Genetic Algorithms*, T. Bäck (ed.). pages 320-329, East Lansing, Michigan, 1997. Morgan Kaufmann.

[Shaffer, 1987] C. G. Shaffer. The ARGOT strategy: adaptive representation genetic optimizer technique. *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms.* J. J. Grefenstette (ed). Lawrence Erlbaun Assoicates. pp. 50-58, 1987.

[Sommerville, 1992] I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, MA, 1992.

[Spears and De Jong, 1991] W. M. Spears and K. De Jong. An analysis of multi-point crossover. *Foundations of Genetic Algorithms.* G. J. E. Rawlins (ed.). pages 301-315, Indiana University, Bloomington, Indiana, 1991. Morgan Kaufmann.

[Spector, 1996] L. Spector. Simultaneous evolution of programs and their control structures. *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pages 137-154, 1996.

[Stroustrup, 1991] B. Stroustrup. *The C++ Programming Language*, 2nd edition. Addison-Wesley, Reading MA, 1991.

[Syswerda, 1989] G. Syswerda. Uniform crossover in genetic algorithms. *Proceedings of the*

Third International Conference on Genetic Algorithms, J. D. Schaffer (ed.), pages 2-9, George Mason University, DC, 1989. Morgan Kaufmann.

[Syswerda, 1991] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. *Foundations of Genetic Algorithms.* G. J. E. Rawlins (ed.). pages 94-112, Indiana University, Bloomington, Indiana, 1991. Morgan Kaufmann.

[Teller, 1994a] A. Teller. Turing completeness in the language of genetic programming with indexed memory. *Proceedings of The 1994 First International World Congress on Computational Intelligence.* pages 136-146, Orlando, Florida, 1994. IEEE Press.

[Teller, 1994b] A. Teller. Genetic programming, indexed memory, the halting problem and other curiosities. *Proceedings of The 1994 Seventh Annual Florida AI Research Symposium,* pages 270-274, Pensacola, Florida, 1994. IEEE Press.

[Turing, 1937] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic 2,* pages 153-163, 1937.

[Whigham and McKay, 1995] P. A. Whigham and R. I. McKay. Genetic approaches to learning recursive relations. *Progress in Evolutionary Computation,* Yao, X. (ed.), pages 17-27, Heidelberg, Germany, 1995. Springer-Verlag.

[Whigham, 1995] P. A. Whigham. Grammatically-based genetic programming. *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, J. P. Rosca (ed.). pages 33-41, Tahoe City, CA. 1995.

[Whigham, 1996a] P. A. Whigham. Search bias, language bias and genetic programming. *Genetic Programming 1996: Proceedings of the First Annual Conference.* J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.). pages 230-237, Stanford University, CA, 1996. MIT Press.

[Whigham, 1996b] P. A. Whigham. *Grammatical Bias for Evolutionary Learning.* Ph.D. Thesis. University of New South Wales, Australian Defence Force Academy. 1996.

[Wineberg and Oppacher, 1996] M. Wineberg and F. Oppacher. The benefits of computing with introns. *Genetic Programming 1996: Proceedings of the First Annual Conference.* J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo (eds.), pages 410-415, Stanford University, CA, 1996. MIT Press.

[Winston, 1992] P. H. Winston. *Artificial Intelligence*, third edition. Addison Wesley, 1992.

[Wolpert and Macready, 1997] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67-82,

April 1997.

[Wong and Leung, 1995] M. L. Wong and K. S. Leung. An adaptive inductive logic programming system using genetic programming. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming.* J. R. McDonnell, R. G Reynolds, and D. B. Fogel (eds.). pages 737-752. San Diego, CA, 1995. MIT Press.

[Wong and Leung, 1996] M. L. Wong and K. S. Leung. Evolving recursive functions for the even-parity problem using genetic programming. *Advances in Genetic Programming II,* P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pages 222-240, 1996.

[Wong and Leung, 1997] M. L. Wong and K. S. Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation,* Vol. 5:2, pages 143-180, 1997.

[Yu and Bentley, 1998] T. Yu and P. Bentley. Methods to evolve legal phenotypes. *Fifth International Conference on Parallel Problem Solving from Nature.* A. E. Eiben, T. Bäck, M. Schoenauer and H.-P. Schwefel (eds.), page 280-291, Amsterdam, 1998. Springer.

[Yu and Clack, 1998a] T. Yu, and C. Clack. PolyGP: a polymorphic genetic programming system in Haskell. *Genetic Programming 1998: Proceedings of the Third Annual Conference.* J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 416-421, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Yu and Clack, 1998b] T. Yu and C. Clack. Recursion, lambda abstractions and genetic programming. *Genetic Programming 1998: Proceedings of the Third Annual Conference.* J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), pages 422-431, University of Wisconsin, Madison, Wisconsin, 1998. Morgan Kaufmann.

[Yu, 1999] T. Yu. Structure abstraction and genetic programming. *Proceedings of the 1999 Congress on Evolutionary Computation,* page 652-659, Washington DC, 1999. IEEE.

Appendix A

Methods to Evolve Legal Phenotypes

A.1 Introduction

Constraints form an integral part of every optimization problem, and yet they are often overlooked in evolutionary algorithms [Michalewicz, 1995b]. A problem with constraints has both an objective and a set of restrictions. For example, when designing a VLSI circuit, the objective may be to maximize speed and the constraint may be to use no more than 50 logic gates. It is vital to perform constraint handling with care, for, if evolutionary search is restricted inappropriately, the evolution of good solutions may be prevented.

In order to explore the relationship between constraints and evolutionary algorithms, this appendix presents an evolutionary framework in which the search space and solution space are separated. In this framework, a genotype represents a point in the search space and is operated on by the genetic operators (crossover and mutation). A phenotype represents a point in solution space and is evaluated by the fitness function. The result of the evaluation gives the fitness of the phenotype, and by implication, of the underlying genotype.

In the same way that phenotypes are evaluated for fitness, not genotypes, it is the phenotypes which must satisfy the problem constraints, not the genotypes (although their enforcement may result in the restriction of some genotypes). However, unlike the fitness evaluation, constraints can be enforced at any point in the algorithm to attain legal phenotypes. As will be described later, they may be incorporated into the genotype or phenotype representations, during the seeding of the population, during reproduction, or handled at other stages.

There are two main types of constraint: the *soft constraint* and the *hard constraint*. Soft constraints are restrictions on phenotypes that should be satisfied, but will not always be. Such constraints are often enforced by using penalty values to lower fitnesses. Illegal phenotypes (which conflict with the constraints) are permitted to exist as second-class, in the hope that some portions of their genotypes will aid the search for fit phenotypes [Michalewicz,

1995b]. Hard constraints, on the other hand, must always be satisfied. Illegal phenotypes are not permitted to exist (although their corresponding genotypes may be, as will be shown).

We identifies eleven methods to enforce constraints on phenotypes during various stages of evolutionary algorithms. Five methods are experimented on a run-time error constraint in a Genetic Programming (GP) system. The results are compared and analyzed.

The appendix is structured as follows: Section A.2 provides related work; Section A.3 classifies and describes the constraint handling methods; Section A.4 presents the experiments; Section A.5 gives the results; Section A.6 analyzes the results and Section A.7 concludes.

A.2 Related Work

A.2.1 Genetic Algorithms

Michalewicz and Schoenauer provide perhaps the most comprehensive reviews of implementations of constraint handling in genetic algorithms (GAs) [Michalewicz 1995b, Michalewicz and Schoenauer 1996]. They identify and discuss eleven different types of system. However, upon examination it is clear that their classification is based upon differences in implementation, and perhaps because of confusion of various multiobjective techniques, it fails to group constraint handling methods which employ similar underlying concepts. Nevertheless, the work of Michalewicz and colleagues provides some of the key investigations in this area. For example, [Michalewicz, 1995a] describes the application of five methods (three based on penalizing illegal phenotypes) to five test functions. Michalewicz and Schoenauer [1996] describe the use of behavioral memory and other penalty-based approaches in GAs to evolve different engineering designs. Schoenauer and Michalewicz [1997] described the use of a repair method in a GA to evolve legal phenotypes.

A.2.2 Evolution Strategies & Evolutionary Programming

In their original implementations, both ES and EP performed constraint handling during the creation of the initial populations. Schwefel's ES algorithm also used a 'legal mutant' constraint handling method, where the creation of an individual is simply repeated as long as the individual violates one or more constraints [Bäck, 1996]. The original EP, on the other hand, typically does not enforce constraints during the generation of new offspring. More recent research on constrained optimization problems in EP is described in [McDonnell, Reynolds and Fogel, 1995] and [Fogel, Angeline and Bäck 1996].

A.2.3 Genetic Programming

The traditional GP paradigm [Koza, 1992] does not distinguish genotypes from phenotypes, i.e. the search space is regarded as being the same as the solution space. An individual is represented as a program tree. This program tree represents both the genotype and phenotype of an individual as it is modified by the genetic operators and it is evaluated by the fitness function. Consequently, constraints in traditional GP are perceived as being applied to phenotypes and genotypes.

For example, program trees in GP are restricted by syntactic constraints: they must satisfy the syntax of the underlying language. Various other forms of syntactic constraints have been proposed [Gruau, 1996; Janikow, 1996]. In Chapter 4, we applied both syntactic constraints and type constraints in our GP system.

Banzhaf [1994] proposed an alternative paradigm for GP, where the search space is separated from the solution space. A mapping scheme is used to transform genotypes into legal phenotypes [Keller and Banzhaf, 1996].

A.3 Constraints in Evolutionary Algorithms

Just as evolution requires selection pressure to generate phenotypes that satisfy the objective function, evolution can have a second selection pressure placed upon it in order to generate phenotypes that do not conflict the constraints. However, using pressure in evolutionary search to evolve legal solutions is no guarantee that all of the solutions will always be legal (i.e., they are soft constraints).

Constraints can also be handled in two other ways: solutions that do not satisfy the constraints can be prevented from being created, or they can be corrected. Such methods can have significant drawbacks such as loss of diversity and premature convergence. Nevertheless, these two types of constraint handling ensure that all solutions are always legal (i.e., they are hard constraints). The following section identifies eleven methods which enforce '*hard constraints*' or '*soft constraints*'. These methods also fall within the three conceptual categories: *Prevention*, *Correction*, and *Pressure* (see Table A.1).

Table A.1: Classification of constraint handling methods

Prevention	HARD	C1, C2, C3, C10
Correction	HARD	C4, C5
Pressure	SOFT	C6, C7, C8, C9, C11

The detailed description will be presented in the subsection. (Note that this categorization encompasses the Pro-Life, Pro-Choice categorization of Michalewicz and Michalewicz [1995]. It is felt that the use of more neutral terminology is more appropriate for such technical classifications.)

A.3.1 Detailed Classification

Whilst previous classification of constraint handling methods within evolutionary search has identified implementation differences of existing systems [Michalewicz, 1995b], to date there has not been a general classification of constraint handling based on the underlying concepts of evolutionary algorithms.

Such a classification can be achieved, not only by examining the existing work of others, but also by examining the significant stages within evolutionary algorithms and identifying where it is possible to incorporate constraints. This allows all existing constraint handling methods to be clearly categorized and understood, and also identifies new, previously unexamined ways of tackling constraints in evolutionary search. Figure A.1 shows the most significant and commonly used stages within current evolutionary algorithms (GAs, GP, ES and EP).

After some careful consideration of these stages, it becomes clear that constraints can be incorporated at eleven different places within the design and execution of evolutionary algorithms (as shown on the right hand side of Figure A.1). These eleven methods should not be confused with Michalewicz's [1995b] list of different researchers' implementations (which coincidentally also contains eleven elements). The methods shown in Figure A.1 are categorized solely on their placement within the evolutionary algorithm, and can be used in combination or separately of each other. There follows a description of each method and its potential advantages and disadvantages:

C1: LEGAL SEARCHSPACE ***Design genotype representation.***

During the design of the evolutionary system, create a genotype representation that is only capable of representing legal solutions. Evolutionary search is then forced to consider only the space of legal solutions, where all constraints are satisfied. This method is frequently used, although designers who use it are often unaware that they are performing constraint handling of any kind. For example, in GAs, if the range of a problem parameter must be between 0 and 255, most designers would automatically use a binary gene consisting of eight bits - and this genotype representation would then ensure that the 0-255 range constraint was always satisfied.

Figure A.1: Constraint placement within stages of evolutionary algorithms.

`Design:`	search space *(contains genotypes)* solution space (contains phenotypes) operators	C1: LEGAL SEARCHSPACE C2: LEGAL SOLUTIONSPACE
`Initialize:`	(random) coded values ↓ genotypes	C3: LEGAL SEED
`Map:`	genotypes ↓ phenotypes	C4: GENETIC REPAIR C5: LEGAL MAP
`Evaluate:`	phenotypes ↓ fitness values	C6: GENOTYPE PENALTY C7: PHENOTYPE PENALTY
`Select parents:`	fitness values ↓ genotypes	C8: LEGAL SELECTION
`Calculate fertility of parents:`	genotypes, fitness values ↓ ↓ ↓ fertility # of offspring per parent	C9: LEGAL FERTILITY
`Generate offspring:`	parent genotypes *crossover/mutation* ↓ child genotypes	C10: LEGAL BIRTH
`Place offspring into population:`	genotypes *replace* ↓ genotypes	C11: ILLEGAL REPLACEMENT

C2: LEGAL SOLUTIONSPACE ***Design phenotype representation.***

During the design of the evolutionary system, create a new phenotype representation, so that only legal phenotypes can be defined. All genotypes are then mapped onto these phenotypes, which by definition, must always satisfy the constraints.

Often great care can go into the design of suitable phenotypes. For example, practitioners of floor-planning problems have two important constraints: room-spaces should not overlap, and no space should be left unaccounted for. To ensure that the computer always evolves solutions that satisfy these constraints, designers of these systems use phenotype representations which define the location of rooms indirectly, by defining the location and number of dividing walls [Gero and Kazakov, 1998].

C3: LEGAL SEED ***Seed with non-conflicting solutions.***

The initial population is seeded with solutions that do not conflict with the constraints and the

crossover and mutation operators are designed so that they cannot generate illegal solutions. Many constraints in GP are implemented using this method. For example, [Gruau, 1996] uses a context-free grammar to specify syntactic constraints of parse trees. In Chapter 4, we employ a type system to ensure that only type-correct programs are considered during evolution.

C4: GENETIC REPAIR *Correct illegal genotypes.*

If a new individual conflicts with a constraint, correct the genes that are responsible for the conflict to make it satisfy that constraint. For algorithms such as GP which make no explicit distinction between genotypes and phenotypes, this method modifies the solution, and the modification is inherited by its offspring.

This genetic engineering approach ensures that all solutions will satisfy all constraints, but may damage epistatic genotypes, discarding the result of careful evolution over many generations. In addition, the design of the repair procedure may be a non-trivial task for some problems.

C5: LEGAL MAP *Correct illegal phenotypes.*

Map every genotype of an individual to a phenotype that satisfies the constraints using some form of simple embryology. This forces all solutions to satisfy all constraints, and also does not disrupt or constrain the genotypes in any way, allowing evolutionary search to continue unrestricted. For algorithms such as GP which make no distinction between genotypes and phenotypes, this method modifies the solution before fitness evaluation, but the modification is not inherited by its offspring. (Also note that although this method is often used in combination with C2, the use of a phenotype representation which can only represent legal solutions is not a prerequisite for the use of Legal Map.)

Using a mapping stage to generate legal phenotypes is a very common approach to perform simple constraint handling. Goldberg [1989] describes perhaps the simplest: mapping the range of a gene to a specified interval. This permits constraints on parameter range and precision to be satisfied without the need to redesign the genotype representation and coding. More recently mapping stages have become more intricate and deserving of the term 'artificial embryology'. Researchers in GP have also reported that the use of an explicit genotype and mapping stage for constraint handling can increase diversity in populations [Banzhaf, 1994].

Type constraints in GP can be implemented using this method as an alternative to the Legal Seed method. A simple example is to map a value with an illegal type of 'real' into a value with legal type 'integer'. However, for other more complex types such as list or array, a proper mapping scheme may be difficult to design. This kind of type-constraint handling is

called 'dynamic typing' - in contrast to the 'strong typing' approach mentioned in the Legal Seed method.

C6: GENOTYPE PENALTY *Penalize illegal genotypes.*

Identify alleles or gene fragments within genotypes that seem to increase the chances of a solution conflicting the constraints, and reduce the fitness of any individual containing these fragments of genetic code. Although the identification of bad genes may discourage solutions from conflicting constraints, it will not guarantee that all solutions satisfy all constraints. In addition, with epistatic genotypes, this approach may result in the discouragement of other, epistatically linked, useful features within solutions. To date, research has investigated the automatic identification of 'good genes' during evolution to encourage the evolution of solutions with higher fitnesses [Gero and Kazakov, 1998]. However, we are unaware of any work which identifies bad genes for constraint handling.

C7: PHENOTYPE PENALTY *Penalize illegal phenotypes.*

When a phenotype conflicts a constraint, reduce its fitness. This soft constraint discourages all phenotypes that conflict the constraints, but does not force evolutionary search to generate legal solutions. In effect, the use of a penalty value becomes an additional criterion to be considered by the evolutionary algorithm, and multiobjective techniques should be used to ensure that all criteria are considered separately (otherwise one or more criteria may dominate the others) [Bentley and Wakefield, 1997]. This is one of the most commonly used methods for constraint handling in evolutionary algorithms. (Indeed, it is the only one explicitly mentioned in [Goldberg, 1989].)

C8: LEGAL SELECTION *Select only legal parents for reproduction.*

During reproduction, only select parent solutions which satisfy the constraints. This method should be used with a fitness-based replacement method to ensure that evolution is guided to evolve fit solutions in addition to legal solutions. (If all solutions are illegal, parents which violate the fewest constraints to the least extent should be selected.) However, the exclusion of potential parents may discard beneficial genetic material and so could be harmful to evolution. Other than the work described in this paper, only one recent investigation has been made on this method [Hinterding and Michalewicz, 1998].

C9: LEGAL FERTILITY *Increase the no. of offspring for legal parents.*

Having selected the parent genotypes (based on their fitnesses) this method allocates a larger fertility to parents which better satisfy the constraints. This method can be thought of as an implicit multiobjective method, allowing independent selection pressure to be exerted for high fitness and legal solutions. Being a 'soft constraint', there are no guarantees that all solutions will always satisfy the constraints. In addition, if legal parents are favored excessively, it

is possible that the diversity of the population could be reduced. To our knowledge, this idea has not been previously used for constraint handling.

C10: LEGAL BIRTH ***Stop illegal offspring from being born.***

If a new solution conflicts a constraint, discard it, and try generating another solution using the same parents. This brute-force method, which is sometimes used in GAs [Michalewicz, 1995b] forces all solutions to satisfy the constraints, but may discard useful genetic material (and may also be prohibitively slow).

Another implementation is to modify crossover and mutation operators to incorporate the constraint bias so that only legal offspring can be generated.

C11: ILLEGAL REPLACEMENT ***Replace illegal solutions with legal offspring.***

When replacing individuals with new offspring in the population, always replace the solutions that conflict constraints. (If all solutions satisfy the constraints, either replace randomly or replace the least fit.) This method should be used with a fitness-based selection method to ensure that evolution is guided to evolve fit solutions in addition to legal solutions. However, the replacement of potential parents discards potentially beneficial genetic material and so may be harmful to evolution. This method requires the use of a steady-state GA [Syswerda, 1989].

A.4 Experiments with a Run-Time Constraint in GP

This section describes experiments conducted to compare five of the constraint handling methods described above in a GP system. The experiments are focused on one particular kind of constraint in GP: the run-time error constraint.

GP evolves computer programs as problem solutions. Thus, in most cases the genetic material is in some sense executable. When run-time errors occur during the execution of a program, its behavior is undefined. A fundamental constraint is therefore imposed on GP: no programs can contain run-time errors.

Unlike other types of constraint, the run-time error constraint has a special property: when it occurs the fitness cannot be calculated. (When the behavior of the program is undefined, the evaluation of its fitness cannot be performed.) Illegal phenotypes are therefore not allowed to exist. This means that soft constraint methods (where illegal phenotypes can exist as second-class) can only be used in conjunction with a phenotype correction method - they cannot be used on their own. In the experiments, the Legal Map method is used to serve this purpose.

A constraint can be handled using many different methods, yet some are more suitable

than others. For the run-time error constraint, its prevention (in methods C1, C2, C3 and C10) is extremely hard because these errors are only evident during program execution. In addition, genetic repair (method C4) requires the corrected material to follow the genotype syntax (so that it can be inherited) which is not appropriate (or easy to implement) for this constraint. Consequently, none of the hard constraint methods are suitable for this problem except for the Legal Map method (C5), which corrects illegal phenotypes (and the corrections are not inherited by offspring).

Soft constraint approaches, on the other hand, are appropriate for this problem. The experiments investigate four of these methods (C7, C8, C9, and C11). (Method C6 which penalizes illegal genotypes by identifying bad genes was not investigated because of the substantial time required for its implementation).

In summary, the experiments investigate one hard constraint-handling method (Legal Map) and four soft constraint methods (Phenotype Penalty, Legal Selection, Legal Fertility, and Illegal Replacement) to enforce the zero-division run-time error constraint. The zero-division constraint was chosen as it is the most frequently observed run-time error, potentially occurring in any numerical problem tackled by GP.

Table A.2: Tableau of the simple symbolic regression problem

Objective:	Find the symbolic function $x^4 - x^3 + x^2 - x$ using 9 pairs of sample points.
Terminal Set:	x
Function Set:	+, -, *, /
Fitness Cases:	9 data points (x_i,y_i) where x_i is the input value between -1.0 and 1.0 and y_i is the desired output value
Fitness:	9/(9+*total_error*), where *total_error* is $\sum_{i=1}^{9} \lvert y_i - R_i \rvert$ and R_i is the result of phenotype execution given input x_i
Hits:	$\sum_{i=1}^{9} p_i$ where $p_i = \begin{pmatrix} 1, \lvert y_i - R_i \rvert \le 0.01 \\ 0, otherwise \end{pmatrix}$
Parameters:	PopSize = 500, MaxTest = 25500, TreeSize = 25, Crossover = 60%, Mutation = 4%, Copy = 36%, Runs = 20
Success predicate:	9 hits (GP stop a run when the success predicate is met)

The experiments use GP to solve a symbolic regression problem, which is to find a function, in symbolic form (with numeric coefficients) that fits a given finite sample of data. It is "data-

to-function" regression. The goal is to find the target function of $x^4-x^3+x^2-x$, given a data sample of nine pairs (x_i, y_i), where x_i is a value of the independent variable and y_i is the associated value of the dependent variable. Table A.2 provides the features of this problem.

A.4.1 Implementation of Constraints

To allow the use of the Illegal Replacement method, the GP system uses a steady-state replacement scheme [Syswerda, 1989] where a population with a constant number of individuals is maintained. Unless otherwise stated, parents are selected using fitness proportionate selection, and offspring replace individuals with the worst fitness in the population. The five constraint handling methods were implemented as follows:

C5: Legal Map. When a run-time error occurs during the execution of a phenotype, the value 1 is returned and the execution continues. For example, if the phenotype is $5+x/x$ and $x = 0.0$, Legal Map changes the phenotype to: 5+1. Corrected phenotypes are marked with a run-time error flag to allow this method to be used in conjunction with the following four.

C7 & C5: Phenotype Penalty with Legal Map. Phenotypes that have to be corrected are penalized by multiplying their total_error values by 2. Legal phenotypes that do not have to be corrected are not penalized.

C8 & C5: Legal Selection with Legal Map. During the selection of parents for reproduction, only programs without run-time errors are selected randomly.

C9 & C5: Legal Fertility with Legal Map. If both parents are legal, three offspring are generated from them. If one parent is legal, two offspring are generated, and if neither of the parents is legal, only one offspring is generated from them.

C11 & C5: Illegal Replacement with Legal Map. One offspring (legal/illegal) is generated to replace a randomly selected illegal individual. If there is no illegal individual left in the population, the normal replacement scheme is used.

A.5 Results

Twenty runs were performed for each constraint handling method. Each run was terminated when a program which produced nine hits was found (i.e., when the evolved function produced output sufficiently close to the desired output for all nine data points) or when 25,500 programs had been processed. If the former occurs, the run is termed *successful*. Table A.3 summarizes the experiment results.

Table A.3: Summary of experiment results

Method	Success/Runs	Average Number of Programs Processed in Successful Runs
Legal Map	18/20	3,983
Phenotype Penalty & Legal Map	18/20	4,841
Legal Selection & Legal Map	5/20	18,284
Legal Fertility & Legal Map	20/20	3,984
Illegal Replacement & Legal Map	3/20	6,998

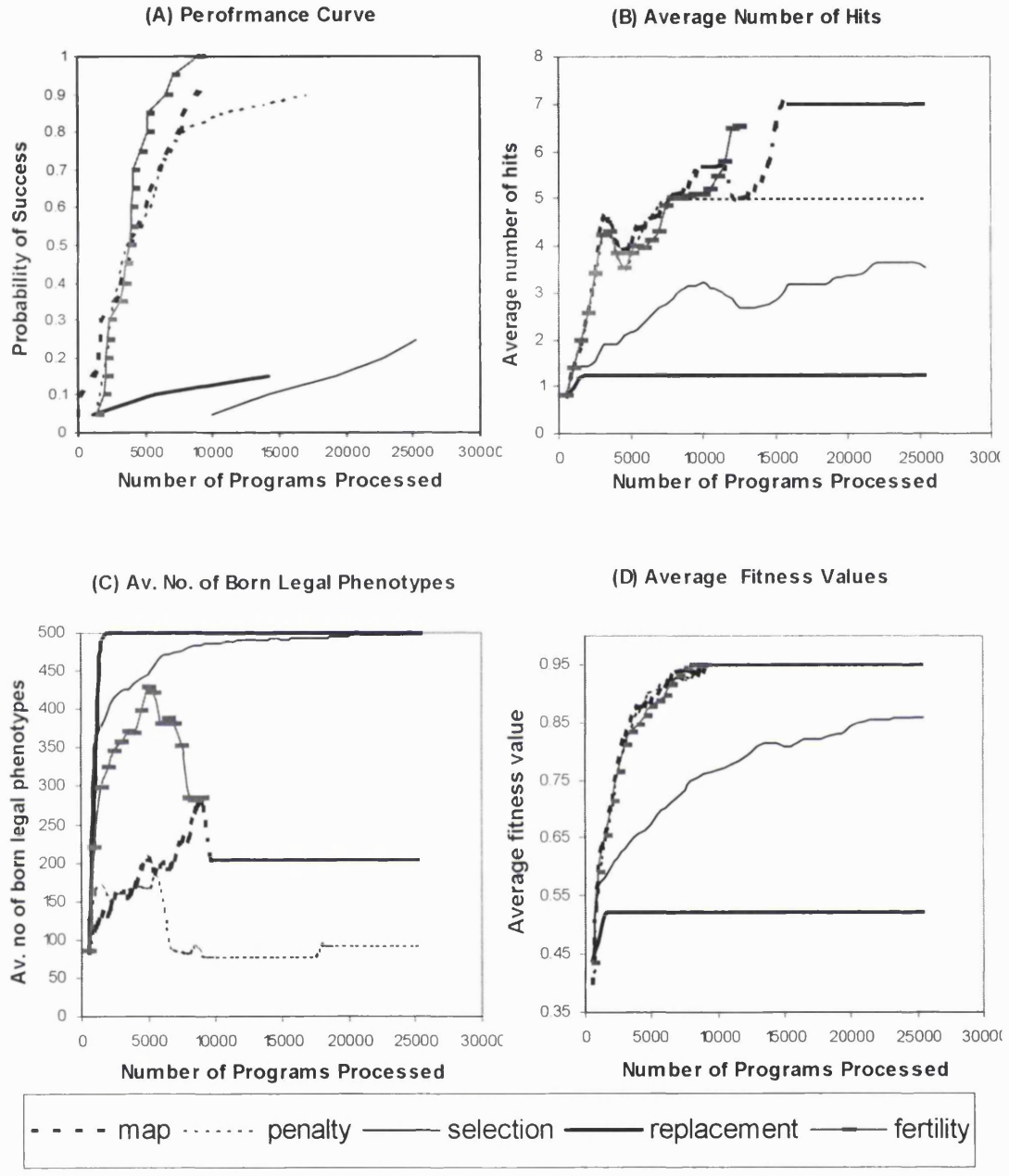
The experiments show that Legal Map, Phenotype Penalty with Legal Map and Legal Fertility with Legal Map find a phenotype with nine hits in most of the runs (18/20 and 20/20). For the successful runs, the average number of programs tested is around 4,000. In contrast, Legal Selection with Legal Map and Illegal Replacement with Legal Map methods do not perform well. Most of the runs are unsuccessful and in the small number of successful runs, they have to test a larger number of phenotypes to find one with nine hits.

Figure A.2(A) provides the probability of success of each method based on the experiments. The Legal Map, Phenotype Penalty with Legal Map and Legal Fertility with Legal Map methods all perform comparably. Their success curves increase stability from the beginning. Most of the runs found a phenotype with nine hits before 10,000 phenotypes had been tested. However, Legal Selection with Legal Map did not achieve this. Its best success rate was 25% with a requirement of processing 25,000 phenotypes. The success probability of Illegal Replacement with Legal Map was also very low. Even when 14,000 phenotypes were processed, there was less than a 20% probability that this method would find a phenotype with nine hits.

It is clear that three of the methods provide good success rates in evolving phenotypes with nine hits, see Figure A.2(B)[2]. However, the results also show that these same methods were the worst at evolving phenotypes which satisfied the run-time error constraint. As shown in Figure A.2(C), the two methods with the lowest success rates: Legal Selection with Legal Map and Illegal Replacement with Legal Map were able to evolve considerably more legal phenotypes than the other methods. Only one method: Legal Fertility with Legal Map, had a high success rate and evolved larger numbers of legal phenotypes.

2. Note that the data shown in Figure A.2(B) were generated in separate runs.

Figure A.2: Result summary charts.



A.6 Analysis and Discussion

The experiments with the run-time error constraint demonstrate a common dilemma in all constrained optimization problems: both the objective and constraints need to be satisfied, and evolving phenotypes which fulfill one of them can sacrifice the evolution of phenotypes which fulfill the other. Using an evolutionary algorithm to find solutions for such problems is therefore difficult because evolutionary search is directed in different directions. The experiments investigated five different ways in which a GP system could be made to evolve both fit

and legal programs. The results show, however, that each method exerted a different level of evolutionary pressure for the constraint and objective. It is clear that such different levels of pressure can effect the degree to which both criteria are met.

In the implementation described above, the Legal Map method (a hard constraint) is the neutral placement in the spectrum (i.e., the control method) as it *repairs* phenotypes without the addition of a second selection pressure for the constraint. The other four (soft constraint) methods use this same phenotype repair scheme with an added *pressure* to reduce the number of illegal programs evolved.

Figure A.2(C) shows the average number of born-legal phenotypes in the population using these methods. Our control, the Legal Map method, enforces no pressure for the constraint and the average number of legal individuals remains around 200 throughout the runs. In contrast, the Illegal Replacement method shows that a very strong pressure is exerted on the GP system to evolve legal programs. After the processing of only 2,500 phenotypes, all illegal phenotypes have been replaced and the population contains only legal phenotypes. The Legal Selection method also exerts a strong pressure for the constraint. Since only legal phenotypes are selected for reproduction, programs which satisfy the constraint are propagated quickly: after 15,000 phenotypes are processed, only legal phenotypes exist in the population. The Fertility method exerts pressure for constraints by generating more offspring for legal parents than for illegal parents. Compared to the control method, Legal Map, all twenty runs of this method show a consistent increase of legal phenotypes in the population. (The downward trend after 5,000 individuals have been processed, evident in Figure A.2(C), is a distortion of the graph caused by a single run, and is not considered significant.)

Not all of the methods exert such consistent pressures for the constraint, however. The Penalty method generates a strong fitness-driven evolutionary process (illegal phenotypes have their *total_error* values doubled to reduce fitness values, so pressure for the constraint drops as individuals become fitter.). As Figure A.2(C) shows, this results in the number of legal phenotypes being gradually reduced to satisfy fitness (objective) requirement. It seems likely that the use of fixed penalty values might prevent this effect.

Figure A.2(D) shows the average fitness in the population using these methods. Driven to satisfy only the fitness (objective), the Map method raises population fitness consistently through fitness proportionate selection. Similarly, the strong fitness-oriented pressure of the Penalty method and the Fertility method raises population fitness consistently. The Selection method also raises the average fitness as it replaces the worst individuals with newly created offspring. However, the average fitness stays below 0.87 because by only selecting legal phenotypes for reproduction, the genetic diversity is dramatically reduced. (Figure A.2(C) shows

that only 15% of initial population were legal). Because of this reduced diversity, combined with the strong pressure for constraints, the population tends to converge prematurely. This is why only 5 out of the 20 runs for the Selection method were successful. The same effect is evident for the Replacement method. Again, genetic diversity is lost as a large number of illegal phenotypes are replaced. Populations converged when around 2000 phenotypes had been processed. Only 3 out of the 20 runs were successful.

In summary, the combination of pressure for the run-time error constraint and fitness directs evolutionary search to find a legal phenotype which produces nine hits. While some of the results may be due to the type of constraint tackled and the implementation of the constraint handling methods, these experiments show that the Fertility method seems to provide the best balance of evolutionary pressure on both criteria. It raises the average fitness value and at the same time reduces the number of illegal phenotypes in the population. As a result, the average number of hits in the population is raised consistently (see Figure A.2(B)) and successful phenotypes are found in all 20 runs.

A.7 Conclusions

This appendix presented a framework to allow the classification of constraint handling methods within various stages of evolutionary algorithms. Such methods impose either hard constraints or soft constraints, and all use prevention, correction, or pressure to enforce the constraints. Eleven methods were identified, including some which had not been explored previously.

Five of these eleven methods were tested on a run-time error constraint in a GP system. The results show that depending on the problem, the methods used and their implementation, the seesaw of evolutionary pressure can either favor constraints or objectives. For this particular problem, of the methods examined, the Legal Fertility method provided a good balance between these two criteria, and led GP to find phenotypes which satisfied both objective and constraints.

Appendix B

Measurement Method

This appendix describes the method used to measure GP performance in this thesis. This method is created by [Koza, 1992 Chapter 8] and has been used by most GP researchers as a standard to evaluate GP performance in their experiments. To facilitate a direct comparison of the performance of our GP system to others', we have adopted the same method in this work.

In this method, the performance of a GP system is measured by the *number of programs* that must be processed in order to satisfy the success predicate of the problem with a specified probability (e.g. 99%). This number is then used to indicate the efficiency of the GP system in solving the problem: the smaller the number is, the more effective the system is.

GP is a probabilistic search method. This means that there is no guarantee that a given run will yield a solution that satisfies the success predicate of the problem. In some runs, premature convergence or non-convergence might happen. In other runs, good solutions might emerge quickly. To measure the performance of a particular GP system, one approach is to make *multiple independent runs* on a problem. The best-of-run program from all of the independent runs can then be designated as the result of the group of the runs.

To measure the amount of computational resources required by a GP system in solving a problem, one can first calculate the number of independent runs (R) that is needed to yield a success with a certain probability (say 99%). Once this number is found, it is then multiplied by the amount of processing required for each run (E) to get the total amount of processing required in solving the problem (I) (i.e. $I = R \times E$).

The following section describe how the values of R, E and I are obtained.

B.1 The Value *R*

To obtain the value R, the value of cumulative probability of success $P(M,i)$ for each generation i using population size M has to be calculated first.

$P(M,i) = \left(\sum_{n=0}^{i} S_n \right) / T$, where T is total number of runs and S is the number of successful runs in generation i (see examples in column 3 of Table B.1).

Once the value of $P(M,i)$ is found, the probability to satisfy the success predicate by generation i at least once in R runs is can be obtained by:

$$1 - [1 - P(M,i)]^R .$$

As $P(M,i)$ is the cumulative probability of success in generation i, $(1\text{-}P(M,i))$ is the cumulative probability of failure in generation i. The cumulative probability of failure in R runs is then:

$$[1 - P(M,i)]^R .$$

The cumulative probability of success in R runs is therefore:

$$1 - [1 - P(M,i)]^R .$$

If we want to satisfy the success predicate with a probability of, say $z = 1 - \varepsilon = 99\%$, then it must be that:

$$1 - [1 - P(M,i)]^R = 99\%.$$

Using this equation, the number of independent runs required (R) to satisfy the predicate by generation i with a probability of z can then be computed:

$$1 - [1 - P(M,i)]^R = z$$
$$[1 - P(M,i)]^R = 1 - z$$
$$\log[1 - P(M,i)]^R = \log(1 - z)$$
$$R \log[1 - P(M,i)] = \log(1 - z)$$
$$R = \left\lceil \frac{\log(1 - z)}{\log(1 - P(M,i))} \right\rceil$$
$$R = \left\lceil \frac{\log \varepsilon}{\log(1 - P(M,i))} \right\rceil$$

where the square brackets indicate the ceiling function for rounding up to the next highest integer.

B.2 The Value *E*

The amount of processing required for each run (E) is the product of the following two factors ($E = M \times G$):

- the number of individuals (M) in the population, and
- the number of generations (G) executed in each run.

This is based on the following assumptions:

- Evaluation of the program is the only processing required;
- Every program is evaluated exactly once in each generation. This is not always true especially for the generation where the optimal solution is found. In this case, GP normally abort the process and returns with the optimal solution. Consequently, the rest of the program in the generation will left without being evaluated.
- All programs take equal amount of processing to evaluate.

B.3 The Value *I*

Once both the values of R and E are obtained, the value of I can be found. The smallest value of I is used to indicate the minimum "effort" required for GP to solve the given problem; hence, the efficiency of the GP system.

Table B.1 provides an example to demonstrate how the values R and I are obtained with M=500 and G=51.

Table B.1: Procedures to obtain R and I

i	*no. of successful run*	*P(50,i)*	*R(0.99)*	*I(50,i,0.99)*
0	0	0.0	∞	∞
1	2	0.25	17	1,700
2	3	0.625	5	750
3	1	0.75	4	800
4	0	0.75	4	850
5	1	0.875	3	900
6	0	0.875	3	950

B.4 The Code

We enclosed the code to calculate the value I.

```
/* cc -lm process.c */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define MAXGEN 51
#define TOTALRUN 60
#define POPSIZE 500
main(argc, argv)
int argc;
char *argv[];
{
  FILE *fp, *fopen();
  double inputs[MAXGEN][5];
  int a;
  int i=0;

  if (argc == 1)
    {
      printf("Wrong number of arguments.\n");
      exit;
    }
  if ((fp=fopen(*++argv, "r")) == NULL)
    {
      printf("Can't open %s \n", *argv);
      exit;
    }

  while (i < MAXGEN )
    {
      inputs[i][0]=0.0;
      inputs[i][1]=0.0;
      inputs[i][2]=0.0;
      inputs[i][3]=0.0;
      inputs[i++][4]=0.0;
    }

  while (!feof(fp))
```

```
    {
      fscanf(fp,"%d", &a);
      inputs[a][0]++;
    }

  inputs[0][1] = inputs [0][0];
  i = 1;
  while (i < MAXGEN )
      inputs[i][1] = inputs[i-1][1]+inputs[i++][0];

  i = 0;
  while (i < MAXGEN )
      inputs[i][2] = inputs[i++][1]/TOTALRUN;

  i = 0;
  while (i < MAXGEN )
    {
      if (inputs[i][2] != 1.0)
inputs[i][3] = ceil(-2.0 / log10 (1.0 - inputs[i][2]));
      i++;
    }

  i = 0;
  while (i < MAXGEN )
    {
      inputs[i][4] = inputs[i][3]* POPSIZE * (i+1);
      i++;
    }

  i = 0;
  while (i < MAXGEN )
    {
      printf("generation %d I is %.0f %.0f %0.3f %.0f %.0f\n", i,
inputs[i][0],     inputs[i][1],     inputs[i][2],     inputs[i][3],
inputs[i][4]);
    i++;
    }
}
```

Appendix C

Structure Abstraction on Artificial Ant Problem

The artificial ant problem [Koza, 1992] has been shown to be difficult for different search algorithms (GP and hill climbing) [Langdon and Poli, 1998a]. In particular, they showed that "current GP techniques are not exploiting the symmetries of the problem." [Langdon and Poli, 1998a]. We have conducted experiments using a higher-order function program representation and hoped it would allow GP to exploit the symmetries. However, the results show no performance improvement over that reported by other researchers using other techniques. We report our experiments and the results. Meanwhile, we are continuing the design of a better higher-order function which can provide the hierarchical processing described in Chapter 7.

Table C.1 explains the problem and the parameters used in the experiments.

Table C.1: The artificial ant problem.

Test Case	The Santa Fe trail
Fitness	Food eaten
Wrapper	Program repeatedly executed for 600 time steps
Parameters	popSize=500; generation=50; crossoverRate=100%; treeDepth=5

Table C.2 gives the functions and terminals used in the experiments.

Table C.2: Functions and terminals for artificial ant problem.

Name	Type
if-food-ahead	state->(state->state)->(state->state)->state
move	state->state
right	state->state
left	state->state

The higher-order function `if-food-ahead` needs more explanation. The first argument is the current state that the ant is in. If there is food in front of the ant, the second argument, a λ abstraction function module, is executed. Otherwise, the third argument, another λ abstraction function module, is executed. This function returns a new state.

During crossover, a λ abstraction can only crossover with another λ abstraction representing the same function argument in the program trees. This means that the first argument of a `if-food-ahead` function can only crossover with the first argument of another `if-food-ahead` function. In this way, GP can use the λ abstraction module mechanism to exploit the symmetries present in the problem, if they exist.

We have made 6 runs and none of them find a solution which can eat all of the 89 pieces of food in the trail, although the average fitness in the population increases gradually. Figure C.1 shows the average fitness in the population.

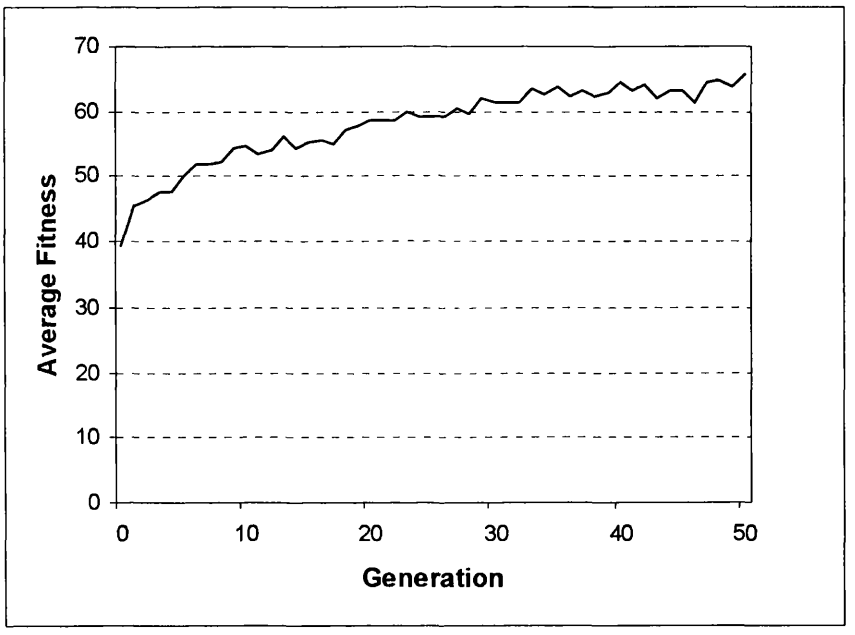


Figure C.1: Average fitness in the population for the artificial ant problem.

Appendix D

An Analysis of Program Evolution

In Chapter 5 and 6, three sets of experimental results were used to support the claim that functional techniques have enhanced GP applicability and efficiency. Polymorphism was endorsed by `nth` and `map` programs, where the incorporation of type variables and function types has enabled GP to evolve these two polymorphic programs. The advantages of implicit recursion and higher-order functions were demonstrated in the general `even-parity` problem. According to the experimental result, which was measured using the method developed by Koza (see Appendix B), the performance of the functional GP system was superior to any other known systems on this problem. These experimental results were sufficient to provide a holistic justification about the applicability and effectiveness of these techniques on GP.

However, the implementation of these techniques has also complicated the GP system with other issues. For example, the system is required to handle two different kinds of run-time error (see Section 4.3.3). To understand the impact that each component (fitness function, run-time error constraint handling, search algorithm) has on the overall good performance, a series of experiments were conducted. The goal of these experiments was to determine whether the undoubted successes in evolving the `nth`, `map` and `even-parity` programs should be interpreted as the result of:

1. random sampling of the set of type-correct programs;
2. sampling by genetic search in the set of type-correct programs that satisfy the run-time error constraints;
3. genetic search guided by the fitness function towards a correct solution.

D.1 Experimental Setup

To achieve this goal, we have designed and implemented the following four experiments:

1. Programs are randomly generated by Creator only, no genetic operation is applied.

2. Programs are generated using Creator (initial population) and Evolver (subsequent generations). However, the fitness function is not used for selection, i.e. programs are randomly selected for reproduction.
3. Programs are generated using Creator, Evolver and a fitness-based selection. The fitness function only considers the satisfaction of the run-time error constraints. The correctness of the outputs is not used to direct GP search.
4. Programs are generated using Creator, Evolver and a fitness-based selection. The fitness function evaluates both the correctness of the outputs and the satisfaction of the run-time error constraints.

In each set of the four experiments, the following data are recorded:

- The number of correct solutions found;
- The mean value of each component of the fitness function.

To allow for a statistically meaningful analysis, the first experiment is implemented by random generation of 100,000 programs. For each of the second, third and fourth experiments, 10 runs are performed. Moreover, each run continues until the end, even if a solution has been found. In terms of implementation, these three experiments are basically the same, except for the fitness function used for selection. With the second experiment, a flat fitness (1.0) is given to every evolved program. With the third experiment, the run-time error fitness is used as the program's fitness. With the fourth experiment, the combination of run-time error fitness and output fitness is used as the program's fitness. The following subsections report the results and our analysis.

D.2 The General Even-Parity Problem

This problem has been described in Section 6.4. The fitness function for this problem consists of two parts: the first part evaluates the satisfaction of the run-time error constraint (run-time error fitness) and the second part evaluates the correctness of the output (output fitness). Moreover, output fitness is weighted as twice as important as run-time error fitness, i.e. 1.0 Vs. 0.5 (see Section 6.6.2).

The first experiment took 70 minutes to complete. Within the 100,000 randomly generated programs, 3 of them are correct solutions. Hence, the "effort" required to find a solution is 33,333 programs. The average fitness of the programs is 4.31351 where -1.6882 is the average run-time error fitness and 6.00171 is the average output fitness.

For the second, third and fourth experiments, each run took about 9 minutes to complete. The results of these three experiments are presented in Figure D.1

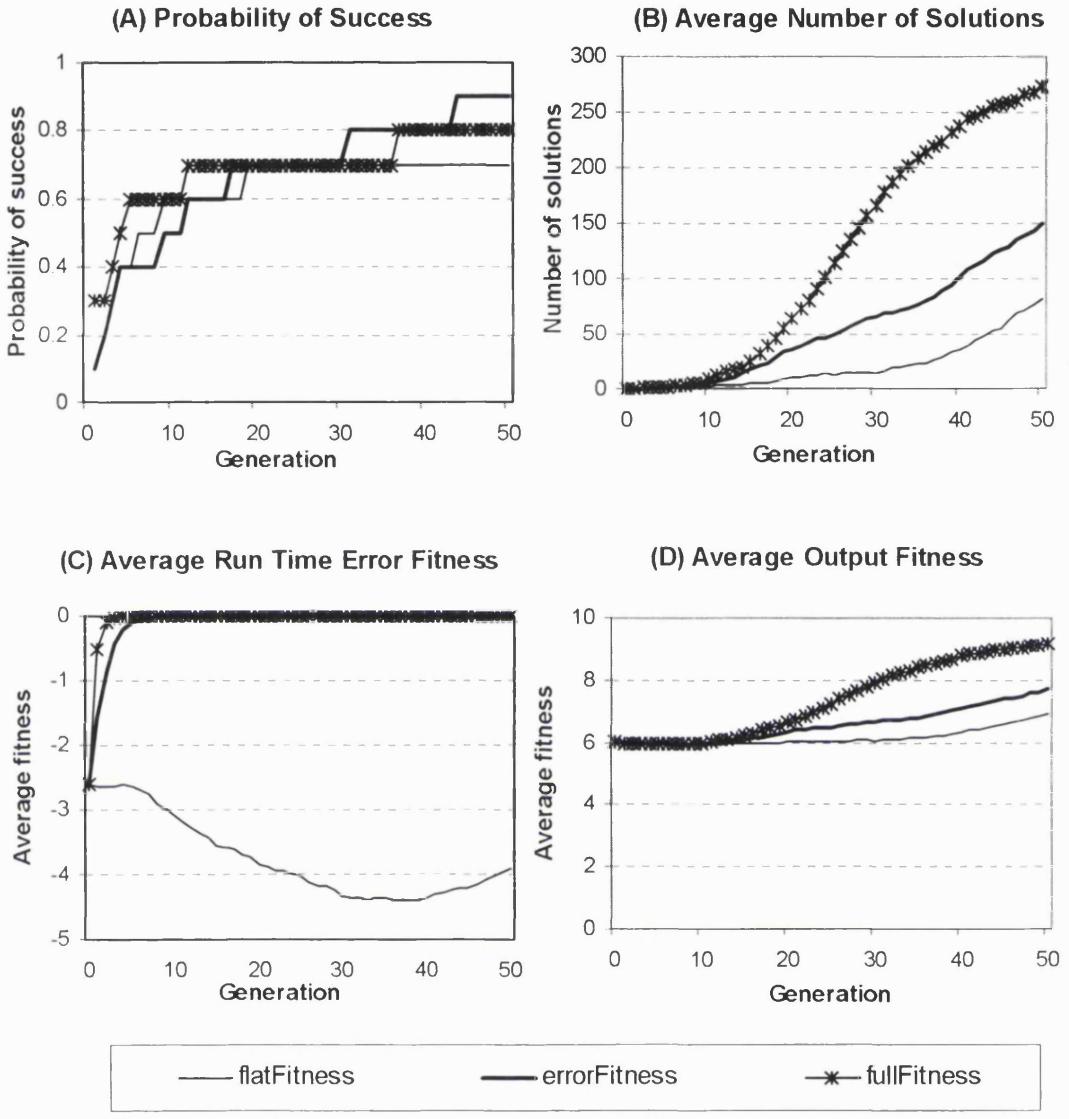


Figure D.1: Experimental results for the general even-parity problem.

Figure D.1 (A) shows the probability of success of the three experiments. With the application of genetic operations (flatFitness), 7 out of the 10 runs have found solutions (the probability of success is 70%). Added with a fitness function (that evaluates the satisfaction of run-time error constraint) to select "fit" programs for reproduction, 9 out of the 10 runs have found solutions (the probability of success is 90%). Extending this fitness function to evaluate the correctness of the output, 8 out the 10 runs have found solutions (the probability of success is 80%). Although errorFitness has the highest probability of success, it requires more generations to find a solution. Consequently, its required "effort" to find a solution is higher than the other two approaches (see Figure D.4(A)).

Figure D.1 shows that without the guidance of a fitness function (flatFitness), genetic

operations search solutions in an ad-hoc manner. In Figure D.1(C), its average run-time error fitness gets worse as the generations progress. The average output fitness, although it does not decrease, improves very slowly during the run (see Figure D.1(D)). As a result, the number of correct solutions found in the population increases slowly (see Figure D.1(B)).

When genetic operations are combined with fitness-based selection, evolution happens. In the third experiment, evolution is directed toward solutions that do not violate the run-time error constraint (errorFitness). Consequently, the average run-time error fitness improves very fast. At generation 6, the population contains no programs with run-time error (see Figure D.1(C)). Genetic operations are applied randomly within the set of programs that satisfy the run-time error constraint. Figure D.1 (D) shows that these operations have improved the average fitness of the population. Consequently, the number of correct solutions found in the population increases faster than that using genetic operations alone (see Figure D.1(B)).

With a fitness function which guides GP to evolve solutions that not only satisfy the run-time error constraint but also produce correct outputs (fullFitness), programs with run-time error were eliminated even faster than the errorFitness approach (see Figure D.1(C)). This suggests that programs without run-time error also produce better outputs. After generation 4, genetic operations are solely guided by output fitness. As shown in Figure D.1 (D), this guidances has made genetic operations to improve output fitness faster than that without guidance (errorFitness). Consequently, the number of correct solutions in the population increases much faster than the errorFitness approach (see Figure D.1(B)).

The results of the third and the fourth experiments show that a genetic search guided by fitness provides a systematic way to find solutions. During the search process, solutions are improved gradually and consistently. At the end, even if a perfect solution is not found, an approximate solution is guaranteed.

D.3 The Nth Program

The `nth` program is explained in Section 5.4.1. The fitness function for this program has three parts: the first part evaluates the correctness of the output (output fitness), the second part evaluates the satisfaction of the empty-list run-time error constraint (empty-list error fitness) and the last part evaluates the satisfaction of the recursion run-time error constraint (recursion error fitness). With one more component, this fitness function is more complicated than that for the `even-parity` problem. Moreover, unlike the `even-parity`, the three components of the fitness function are given equal weight. However, due to the ways that the two run-time errors are handled, more weight is given to the run-time errors (see Section

5.4.1).

The first experiment took 90 minutes to complete[1]. Within the 100,000 randomly generated programs, none of them is a correct program. The average fitness is -121.266 where 7.342464 is the average output fitness; -41.9595 is the average empty-list error fitness and −86.6488 is the average recursion error fitness.

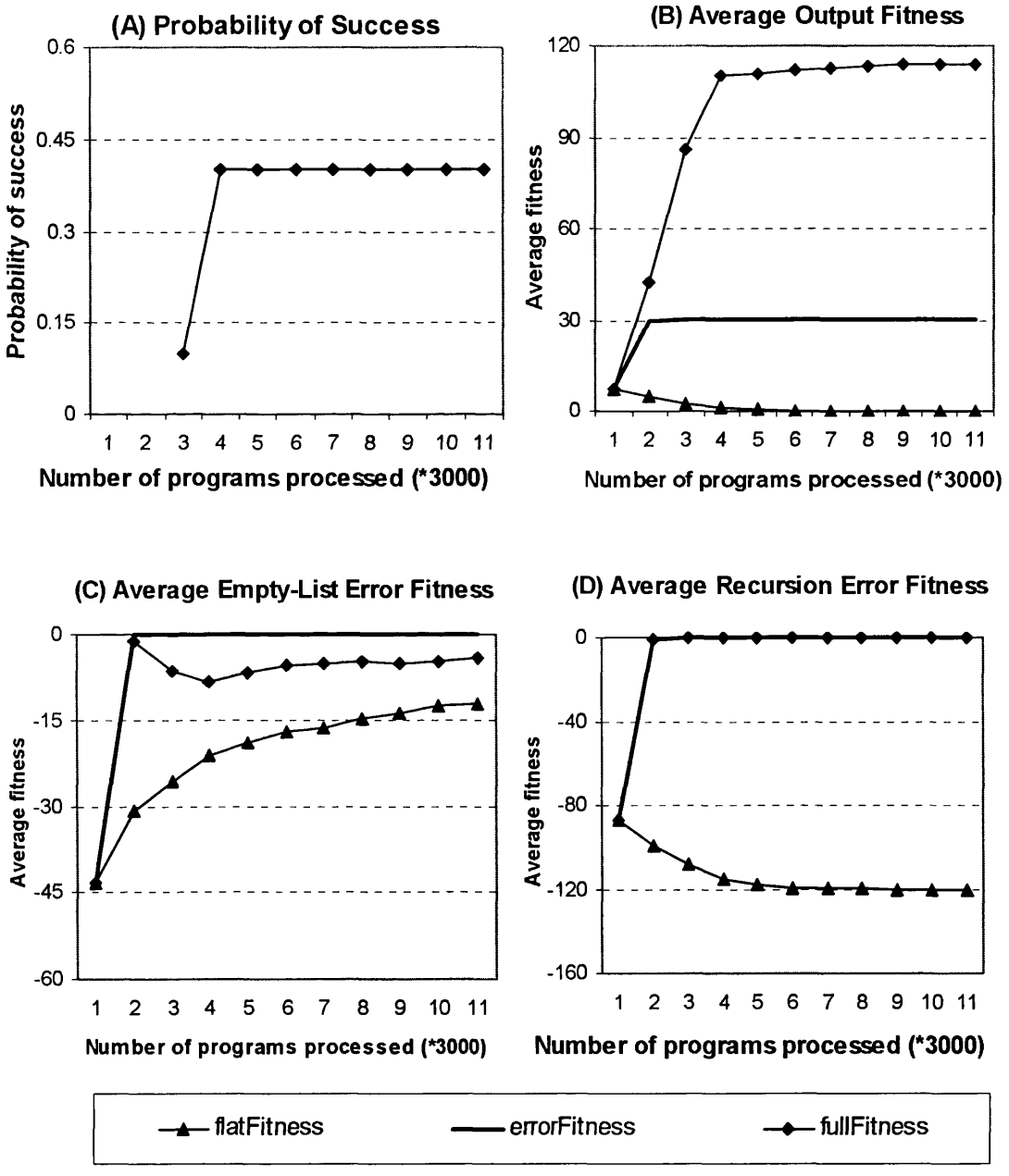


Figure D.2: Experimental results for the nth program.

1. The experiments were conducted through the internet which took longer than normal due to network delay. Also, the experiments used my old code which implemented population as a list (an inefficient data structure for updating), hence required more time to run.

Each run for the second, third and fourth experiments took about 60 minutes to complete[1]. Since this experiment used a steady-state replacement, there is no concept of "generation". Instead, the data are reported every 3,000 (the size of the population) programs were processed. The results of these three experiments are presented in Figure D.2.

For these three sets of experiments, only the one which applied genetic operations with full fitness has found correct `nth` programs. Within 10 runs, 4 of them found a solution. The probability of success is 40%. Moreover, all the successful runs found a solution before 12,000 programs were processed (see Figure D.2 (A)).

Similar to the results from the `even-parity` experiments, genetic operations applied on randomly selected programs do not provide a consistent improvement of the solutions. Although the average empty-list error fitness increases (see figure D.2 (C)), the average output fitness and recursion error fitness decrease as the number of programs processed increases. Consequently, no solution was found at the end of the 10 runs.

When genetic operations were guided by a fitness function to evolve programs which satisfy the two run-time error constraints (errorFitness), both the average empty-list and recursion error fitness improve very fast during the runs. After 3,000 genetic operations, all programs with run-time errors were eliminated from the population (see Figure D.2 (C) & (D)). Unfortunately, this "biased" selection also eliminated potential good programs from the population. As shown in Figure D.2 (B), applying genetic operations randomly on the set of programs which satisfy the run-time error constraints does not improve output fitness (due to premature convergence). Consequently, no solution was found at the end of the 10 runs.

Using the combination of run-time error and output fitness to determine the selection of programs for reproduction, correct `nth` programs have emerged during the runs. At the beginning of the evolutionary process, programs with recursion error were eliminated from the population very quickly (see Figure D.2 (D)). The evolution became a competition between programs with good output fitness and good empty-list error fitness. This result confirms our analysis in Section 5.4.1. Between the processing of 6,000 and 8,000 programs, the average output fitness increases while the average empty-list error fitness decreases, i.e. evolution is driven by the output fitness. After that, both output and empty-list fitness improve consistently (see Figure D.2 (B) & (C)). As a result, 4 of the 10 runs have found a correct `nth` program.

D.4 The Map Program

The `map` program is explained in Section 5.4.2. The fitness function for this program has four

parts: the first part evaluates the correct length of the output (length fitness), the second part evaluates the correct contents of the output (output fitness), the third part evaluates the satisfaction of the empty-list run-time error constraint (empty-list error fitness) and the last part evaluates the satisfaction of the recursion run-time error constraint (recursion error fitness). With one more component, this fitness function is more complicated than that used to evolve `nth` program. Moreover, unlike the `nth`, the four components of the fitness function are weighted with different importance. With the addition of an extra pressure caused by the two run-time error handling methods, the evolutionary process of this program becomes very complicated. Consequently, the generation of the correct `map` program is harder than the generation of the correct `nth` program (see Section 5.4.2).

The first experiment took 90 minutes to complete[1]. Within the 100,000 randomly generated programs, none of them is a correct program. The average fitness is -82.6504 where -19.9724 is the average length fitness; 0.0976 is the average output fitness; -27.9946 is the average empty-list error fitness and -34.7787 is the average recursion error fitness.

Each run in the second, third and fourth experiments took about 120 minutes to complete[1]. The results are presented in Figure D.3.

Among these GP runs, only those which applied genetic operations with full fitness function have found correct `map` programs. There are 3 such successful runs within 10 trials. The probability of success is thus 30%. All correct solutions were found before 35,000 programs were processed.

Applying genetic operations on randomly selected programs does not generate better programs than those created using random search (initial population). Figure D.3 (A), (B), (C) & (D) show that the four average fitness stay pretty much the same as the initial population through out the run.

When genetic operations were directed to evolve programs which satisfy the two run-time error constraints, both the average empty-list and recursion error fitness improved (see Figure D.3 (C) & (D). However, similar to the results of `nth`, the average length and output fitness do not improve at all through out the runs. Consequently, no solution was found at the end of the 10 runs.

Using the combination of run-time errors, length and output fitness to select programs for reproduction, correct `map` programs have emerged during the runs. Because evolution is pulled toward different directions by the four different criteria, the 4 fitness graphs have very interesting shapes (see Figure D.3 (A), (B), (C) & (D)).

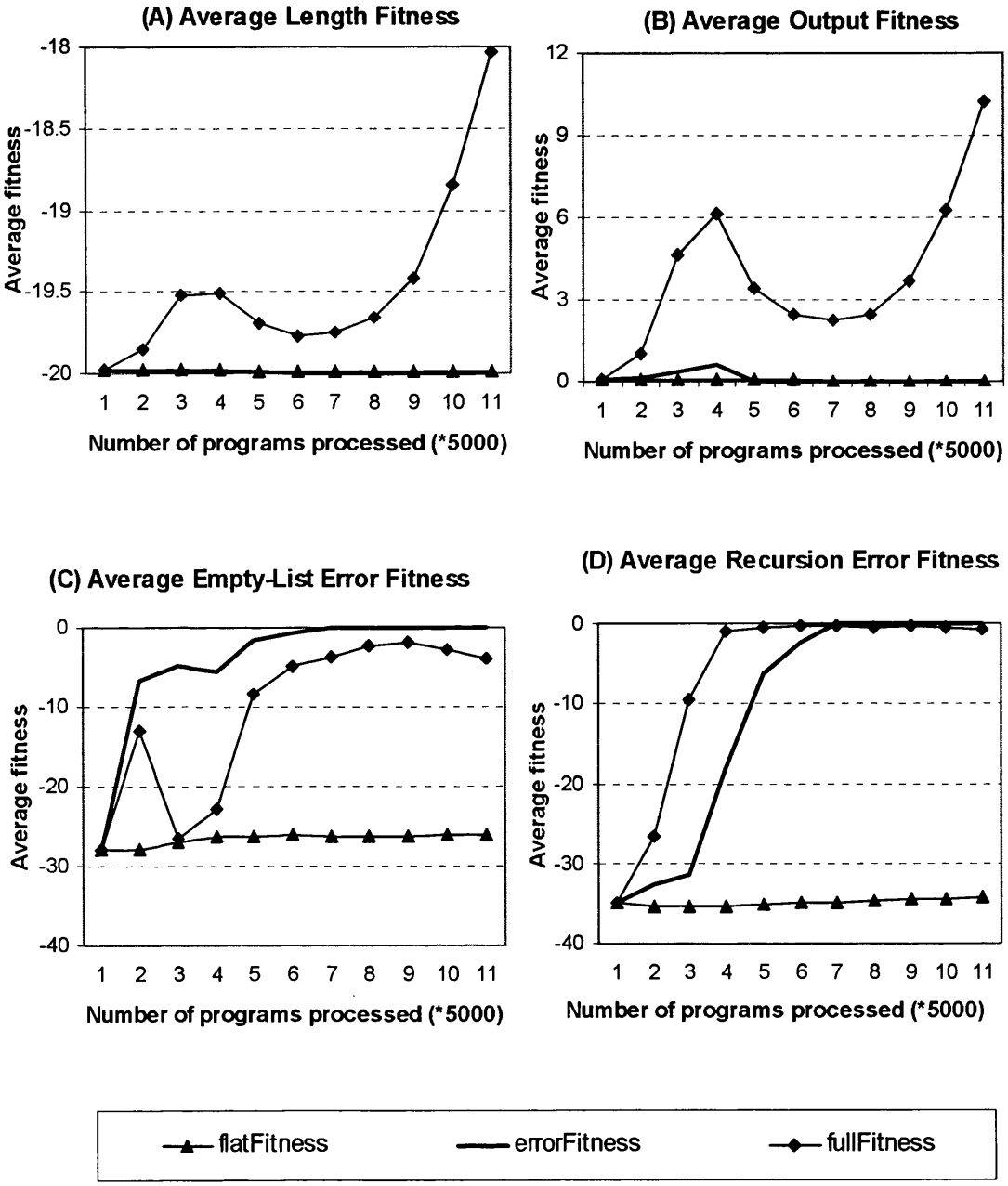


Figure D.3: Experimental results for the map program.

At the very beginning of the evolution (before 20,000 programs were processed), recursion error fitness dominates evolution because it is weighted the most in the fitness function (see Section 5.4.2). The average output and length fitness also improve (see Figure D.3 (A) & (B)), which indicate that programs without recursion error also produce more accurate outputs. The empty-list error fitness, however, does not increase consistently during this period of time.This result can be explained by the sea-saw effect described in Appendix A.

After most of the programs with recursion error were eliminated from the population,

empty-list error fitness becomes the dominant factor to drive evolution. Between the processing of 20,000 and 35,000 programs, its fitness improved very fast. However, both output and length fitness decreased.

After 35,000 programs were processed, the population contained no program with recursion errors. Since the empty-list error fitness was reasonably good (-5), the evolution became focused on improving output and length fitness. As shown in Figure D.3 (A) & (B), their fitness values increase very fast. At a result, 3 of the 10 runs have generated correct `map` programs.

D.5 Discussion

Based on the experimental results, we will address the questions posed at the beginning of the appendix. Random sampling of the set of type-correct programs has found correct general `even-parity` program, but not `nth` nor `map` programs. Moreover, the effort required to find the correct `even-parity` using random search is 33,333 (see Section D.2), which is more than that required by genetic operations or evolutionary search (see Figure D.4 (A)).These results suggested that the good performance of functional GP on these 3 problems can't be achieved by random sampling of the set of type-correct programs.

Genetic operations on randomly selected programs did not provide better performance than random search. As shown in Figure D.4 (B), (C) and (D), this approach generates average fitness that is about the same as that of the initial generation throughout the runs. In fact, it even caused the fitness to decrease with the `even-parity` problem. This can be explained by perceiving genetic operations as performing random search within the population. Instead of the whole search space of all possible solution to explore, genetic search is restricted within the population. For some problems, such as `nth` and `map`, this constraint does not have an effect on the performance. For others, such as `even-parity`, this lack of diversity causes genetic search to perform worse than random search.

Genetic operations guided by a fitness function towards programs without run-time errors have successfully eliminated all programs with run-time errors from the population. This leads to the random application of genetic operations within the set of programs that satisfy the run-time error constraints. With the `even-parity` problem, this set is the promising area of search space, i.e. programs without run-time error also produce better outputs. Hence, the random genetic operations have improved population fitness (see Figure D.3 (B)). However, this is not the case with `nth` or `map` programs. The restricted set has reduced program diversity and prevented the improvement of population fitness (see Figure D.4 (C) &

(D)).[2] Based on these results, we conclude that the good performance of functional GP on these 3 problems can't be achieved by applying genetic operations randomly in the set of type-correct programs that satisfy run-time error constraints.

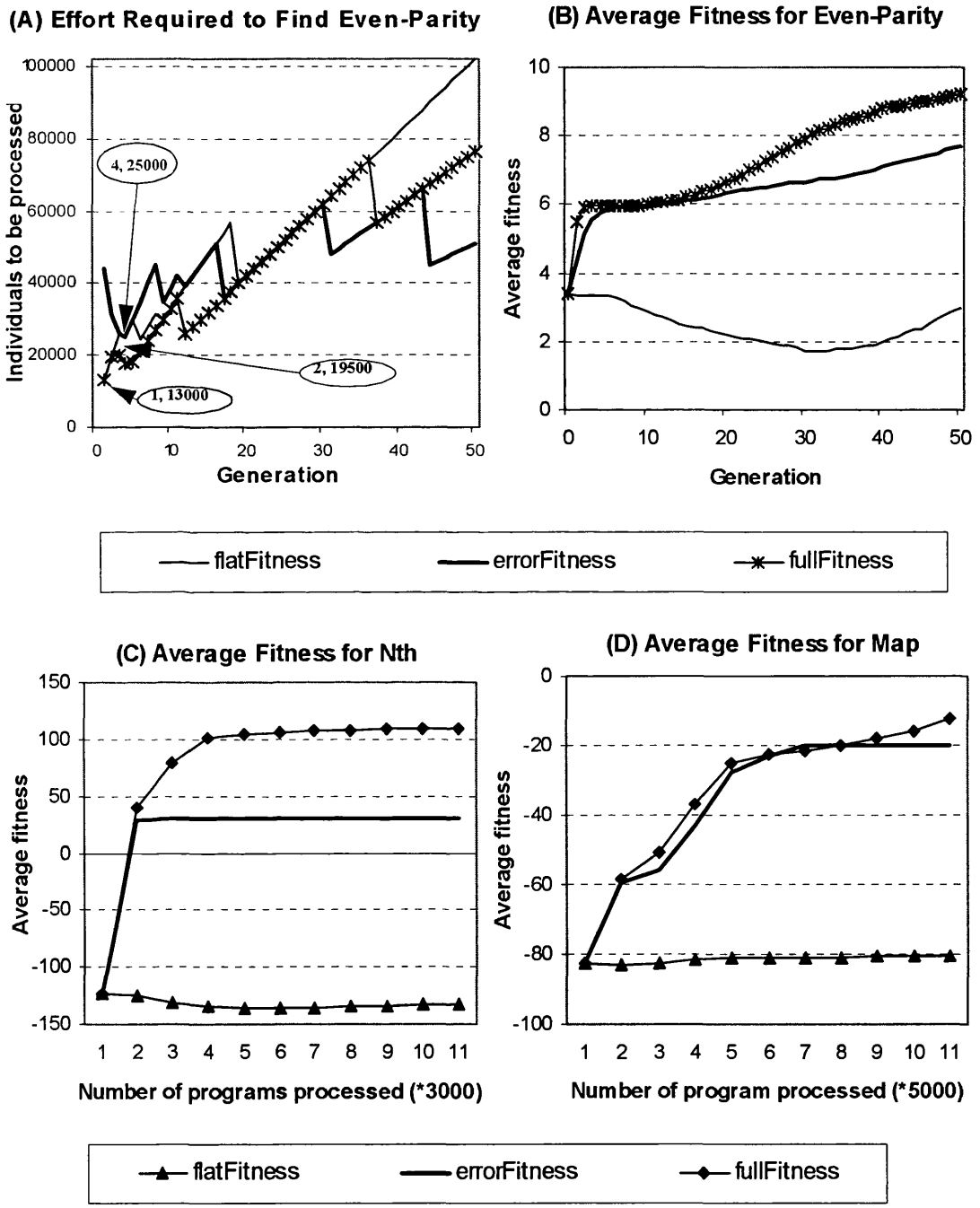


Figure D.4: Summary of performance.

The good performance of functional GP on `nth` and `map` programs is a result of genetic

2. With the `map` experiments, programs with *both* run-time errors are completely eliminated from population after 35,000 programs are processed.

search guided by the fitness function towards a correct solution. Although the two fitness functions are designed with different emphasis, evolution complies the law of "survival of the fittest" and enables programs which most satisfy the fitness function to emerge.

With `nth` program, the fitness function and run-time error handling methods are designed to eliminate programs with recursion error from the population first (see Table 5.2). Indeed, this is exactly what has happened in our experiments. This leads the evolution to become a process of competition between programs with good empty-list fitness and good output fitness. Initially, output fitness drives the evolution (population input fitness improves very fast). After a while, both empty-list error fitness and output fitness improve consistently. The experimental results indicate that this fitness function has directed genetic operations to find correct `nth` programs successfully.

With `map` program, the fitness function and run-time error handling methods are designed to eliminate programs with recursion error from the population first (see Table 5.4 & 5.5). Indeed, this has also happened in our experiments.This makes the evolution become a process of competition between programs with good empty-list fitness, good length fitness and good output fitness.This competition process, however, is complicated as the three fitness values seem to relate to each other in ways that cause the see-saw effects to appear in various stages of the evolutionary process (see Section D.4). Nevertheless, this fitness function has directed genetic operations to find correct `map` programs successfully.

The experimental results for the `even-parity` problem suggested a similar conclusion yet the search space is so small that the performance advantage provided by genetic operations with full fitness (13,000) over other method (25,000 by errorFitness and 19,500 by flat Fitness) is not as obvious as that in the other two problems (see Figure D.4 (A)). Nevertheless, the fitness function has successfully directed evolution to find correct `even-parity` programs.

The fitness function and run-time error handling method are designed to give more weight to output fitness (see section 6.6.2). However, the experimental results show that most programs with run-time error also can't produce good outputs, i.e. not many programs belong to the third category in Table 6.1. Consequently, all programs with run-time error have bad fitness (-5) and are eliminated from the population first. The evolution is then solely directed by output fitness. As shown in our experimental results, this fitness function has directed genetic operations to find correct `even-parity` programs successfully.