
Genetic Programming and Evolvable Machines, 2, 345–380, 2001

Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lambda Abstraction

TINA YU tiyu@chevron.com
Chevron Information Technology Company, 6001 Bollinger Canyon Road, San Ramon, CA 94583

Received January 8, 2001; Revised July 3, 2001

Abstract. We present a novel approach using higher-order functions and λ abstraction to evolve recursive and modular programs. Moreover, a new term "structure abstraction" is introduced to describe the property emerged from the higher-order function program structure. We test this technique on the general `even-parity` problem. The results indicate that this approach is very effective with the general `even-parity` problem due to the appropriate selection of the `foldr` higher-order function. Initially, `foldr` structure abstraction identify the promising area of the search space at generation zero. Once the population is within the promising area, `foldr` structure abstraction provides hierarchical processing for search. Consequently, solutions to the general `even-parity` problem are found very efficiently. We identify the limitations of this new approach and conclude that only when the appropriate higher-order function is selected that the benefits of structure abstraction show.

Keywords: genetic programming, hierarchical processing, recursion, structure abstraction, higher-order functions, lambda abstraction, polymorphism, type systems

1. Introduction

The pursuit of automatic evolution of computer programs spans four decades. In 1958 and 1959, Friedberg and colleagues developed a system that learned to write machine code computer programs using what looks like a modern mutation operations [13, 14]. In 1966, Fogel and colleagues evolved finite state machines as computer programs to predict a sequential series of symbols in light of an arbitrary payoff function [12]. Machine code and finite state machines are early development of computer programming languages. They are foundation to Computer Science and have influenced the development of modern programming languages.

Later, various high-level programming languages with more user-friendly syntax were developed. Among them, LISP S-expressions have been proposed as candidates for programs evolution by different researchers [15, 16, 19]. There were also works on evolving tree-structured programs using a subtree crossover operator [6, 9]. However, it was not until the work of Koza [24–26], which not only provides the techniques to evolve tree-structured programs but also demonstrates the versatility of his methods on many engineering problems, when the approach of evolving tree-structured LISP programs received much attention. Since then, the standard Genetic Programming (GP) paradigm defined in [26] has been extended

to incorporate programming language features by different researchers [4]. Moreover, many of the state of the art GP techniques have been used to create the first commercial GP software Discipulus, which evolves machine code to achieve great speed [11].

The interest in extending GP's ability to evolve recursive and modular programs stems from the wisdom in both the Artificial Intelligence (AI) and the Programming Languages communities. In AI, "divide and conquer" has long been recognized as an important problem solving approach [32]. Meanwhile, most programming languages support *module creation* as a mechanism for problem decomposition and *recursion* for code reuse. By enhancing GP with modularity and recursion, it has been shown that GP can be more effective with some problems (see Section 3).

Although there has been work in evolving recursive programs, its success has been hindered by non-terminating recursive calls (see Section 2.1). This paper addresses this issue by introducing *implicit recursion* provided by higher-order functions. In particular, a new approach using higher-order functions and λ abstraction is introduced to evolve recursive and modular programs. We test this method on the general even-parity problem. Moreover, the program evolution process is analyzed to understand how this approach assists GP search. Such information is very useful to the effective application of the technique to general problems.

The paper is organized as follows: Section 2 provides background of this work. At first, the difficulties of GP to evolve recursive programs are identified. After that, the concept of higher-order functions and λ abstraction is presented. In Section 3, related work is summarized. In Section 4, a new technique using higher-order functions and λ abstraction to evolve recursive and modular programs is presented. Moreover, a new term "structure abstraction" is introduced to describe the property produced by the higher-order function program structure.

Section 5 describes a GP benchmark problem, general `even-parity`. Section 6 presents the experimental results using the new technique to solve this benchmark problem. In Section 7, the evolution of program structure in the search space is investigated. Section 8 discusses the results of our study. In Section 9, the limitations of the new technique are summarized. Finally, Section 10 gives the conclusions.

2. Background

2.1. 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 therefore leads to more compact programs, hence can facilitate generalization [21].

Although a powerful reuse mechanism, recursion must be used carefully, otherwise it can cause the evaluation of a program loop forever. There are two important

criteria which are sufficient for a recursive program to halt:

1. have a terminating condition (base case);
2. the recursive calls are successively applied to arguments that 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* where arguments of a function are evaluated only if their values are needed, it is possible for programs containing code that leads to infinite loops to halt. This happens when the value of the code which generates infinite loops is not needed and therefore is not evaluated. On the other hand, *strict evaluation* requires the evaluation of a function's arguments before the function body and can cause the evaluation of such a program loop forever.

Due to the non-termination issue, evolving recursive programs has been very difficult for GP. In particular, there are three challenges that a GP system has to confront:

- **Determining if a program is non-terminating:** 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 [40]. 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 is 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.
- **Handling 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 decision. It is difficult because non-terminating programs can be of diverse content. 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 in Section 3.1). Moreover, the non-termination of a program can also be reflected in the fitness function to guide GP search. Ideally, a fitness function

should be designed to promote programs which not only produce the desired outputs but also terminate. However, such a fitness function is not easy to obtain [45].

- **Measuring the recursion semantics in an evolved program:** The standard GP paradigm defined in [26] uses a syntactic approach to build programs; no semantic analysis is supported. A recursive program which contains a perfect base-case statement is therefore not necessary to be selected for reproduction since program structure is not normally considered during GP search. Whigham and McKay have identified this problem and suggested that the application of genetic operators should be performed in an environment where semantic analysis is provided [41].

2.2. Higher-order functions and lambda abstraction

Higher-order functions are functions which take other functions as arguments or produce other functions as results. As an example, `twice` is a higher-order function which takes its first argument, a function `f`, and applies it twice to its second argument x:

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

The function argument `f` can be given in two different ways:

- as the name of a predefined function. For example, assuming that `add10` and `multiply10` are predefined, they can be used as arguments to `twice` and produce the following results.

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

- as a λ abstraction, which is an anonymous function definition. The syntax of a λ abstraction is as follows:

$$\lambda\ \mathtt{x.}\ \mathtt{e}$$

where $e ::=$	`c`	built-in function or constant
	| `x`	identifier
	| $\mathtt{e}_1\ \mathtt{e}_2$	application of one expression to another
	| λ `x. e`	λ abstraction

For example, `add10` can be implemented as a λ abstraction (λ x. + x 10). Given the λ abstraction as function argument to `twice`, the following result is generated:

```
twice(λ x. + x 10) 1 = (λ x. + x 10)((λ x. + x 10) 1)
                     = + ((λ x. + x 10) 1) 10
                     = + ( + 1 10 ) 10
                     = 21
```

The combination of higher-order functions and λ abstraction function arguments creates a form of modular program structure: a λ abstraction is an independent module that performs a specific task within a program. Moreover, the module may be reused in the higher-order function. In the example of `twice`, the λ abstraction function argument is reused twice in the higher-order function. Consequently, a higher-order function with a λ abstraction function argument provides a mechanism of module creation and reuse.

3. Related work

3.1. Evolving recursive programs

Koza investigated a problem-specific form of recursion to solve the Fibonacci sequence induction problem [26]. 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` was introduced into the function set. When a program containing the `SRF` function was evaluated for value of position `j`, the `(SRF k D)` statement computed the value for sequence position `k` provided `k` is between 0 and `j` − 1, otherwise, it returned 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 used GP to evolve programs with recursive ADFs to perform tree search (see Section 3.2 for the explanation of ADF) [7]. To evolve a recursive ADF, the name of the ADF was included in the function set used to evolve the ADF. However, an evolved recursive ADF might contain infinite-loops (see Section 2). To deal with this problem, Brave specified the depth of a tree as the limit of recursive calls. Usually such a limit affects the evolutionary 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 recursive call after the 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.

Wong and Leung proposed the use of recursion to evolve a general solution to the `even-parity` problem [44]. Their approach is to construct a logic grammar which includes a rule for recursive calls. In addition, the grammar enforces a termination condition in the program structure. However, the convergence of recursive calls in the program is not guaranteed, hence the evaluation of a program may become non-terminating. During program evaluation, an execution time limit is used to halt the program. A program which does not produce a result after the allowed evaluation time is regarded as a program producing a wrong result. No partial credits nor extra penalty is given to the program. Wong and Leung 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 ADF approach. Yet, we have devised a new strategy in evolving recursive programs which provides even better performance than Wong and Leung's on this problem. We will explain this work in Section 4.1.

Whigham designed two *directed mutation operators* to guide GP in evolving a recursive `member` function using his context-free grammar GP system [42]. In this system, each program is represented as a derivation tree. A directed mutation operator specifies that a subtree generated by one particular grammar rule is replaced by another subtree generated by a different rule. The two directed mutation operators he designed for recursion are for two different purposes. The first one is to repair programs that contain tautologies: the derivation tree `(eq x x)` is replaced with `(eq x (car y))`. The second one is to detect the pattern in a derivation tree where a recursive call should take place: the derivation tree `(eq x (car (cdr (cdr y))))` is replaced with `(member x (cdr y))`.

Using these two directed mutation operators, Whigham showed that the likelihood of evolving the recursive `member` function has improved. 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 we designed for the member function. Knowledge about the solution is nicely used to direct GP search. For problems which do not have an obvious recursion pattern, this approach may not be helpful.

3.2. Evolving modular programs

The original GP paradigm has no explicit support for 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).

3.2.1. Automatically defined functions. Koza defined an ADF as "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" [27]. When solving a problem that has considerable regularities in its solutions, ADFs provide GP with 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 programs more efficiently [18, 23, 27, 28]. However, on problems which do not have regularities, such as the two-boxes problem, ADFs do not provide any advantage [27].

The structures of GP 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 zero. Program structure is then open to evolutionary determination [27]. The last approach is to create all programs in generation zero with the same structure. Six architecture-altering operators are then used to evolve program structures during GP runs [29]. 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 the evolutionary process to determine the program structure among a wide range of possibilities can be computationally expensive. We have identified these shortcomings and provided a better way to define program structures based on the specification of higher-order functions in the function set. The details will be discussed in Section 4.

3.2.2. Module acquisition. MA is a method to support program module reuse by creating a library of subtrees extracted from the program trees in the population [1–3]. Unlike ADFs, which are locally defined for each program, modules in MA are globally defined and can be used by other programs in the population.

Two additional genetic operators are defined 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 replaces the subtree extracted from the program tree. Figure 1 shows the operation of the compression operator. Expansion is the opposite of compression: it replaces the name of a subroutine with its correspondent subtree.

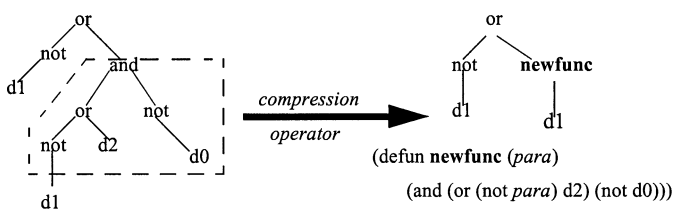

Figure 1. Compression operator in module acquisition.

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 very complex tasks. Consequently, the chance of breaking up desirable portions of a program during genetic operations may overwhelm 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 1, the compression operation reduces the number of nodes from 10 to 5 without altering the semantics of the program. 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.

3.2.3. Adaptive representation through learning. Similar to MA, the ARL method extracts program segments to create functions [35–38]. 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 programs for the next generation. Moreover, these dynamically created functions may be deleted from the function set when their usage does not 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 this program code. ARL addresses these two issues in the following ways: (1) local measurements such as parent-offspring differential fitness and block activation are used to detect good program code; (2) global measurement such as population entropy is 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, Rosca shows that ARL produces better programs than GP alone [36]. Recent research, however, reported that none of the selection heuristics they used under the ARL framework produced a better result than that produced by a simple near-random selection [10].

3.2.4. Automatically defined macros. Spector proposed the use of ADMs in GP to simultaneous evolve programs and their control structures [39]. 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 body of an ADM 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 an ADM. Since the evaluation of the ADM is carried out in the main program, the argument may or may not be evaluated depending on the calling environment, hence speed up program evaluation.

Another proposed model approach is the hierarchical GP (hGP) by Banzhaf and his colleagues [5]. Models in a hGP program is layered; models on one layer is only allowed to call models in an immediate lower layer. Moreover, these models are local to one program only. No global sharing is provided. They have implemented a hGP with the following restrictions:

- each program can have a maximum model level of 1;
- each program can have a maxima number of model of 1;
- mutation is only on the highest level.

Their experimental results show that this model approach improve the performance over the standard GP.

4. A new strategy

4.1. Higher-order functions: Implicit recursion

The issues of explicit recursion in GP highlighted in Section 2.1 foster the idea of implicit recursion. Higher-order functions can be used as vehicles to provide recursion semantics without explicit recursive calls. This is illustrated in Figure 2.

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

Figure 2. Explicit recursion versus implicit recursion.

By extracting the code in a recursive program and passing it as an argument to a higher-order function that applies the code iteratively to the inputs, the same recursion semantics is achieved without explicit recursive calls.

For example, the higher-order function `foldr` applies its function argument `f` iteratively to its two other inputs (`v` and a list `L`) to produce an output. The following describes this operation:

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

The `foldr` implicit recursion is also demonstrated in the following example, where the function + is applied 3 times to the inputs. Note that "+" is used as

the function argument for demonstration purpose. In practice, this function argument may be of any complex code.

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

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 inside the higher-order functions. Consequently, none of the recursion issues raised in Section 2 applies to implicit recursion. This makes implicit recursion an ideal mechanism for GP to evolve recursive programs.

4.2. Lambda abstraction: Module mechanism

Higher-order functions with λ abstraction function arguments provide a structure of modular programs. Within a GP program tree, a λ abstraction is represented as a subtree which can evolve during GP runs. Moreover, the λ abstraction can be reused within the higher-order function through implicit recursion. Consequently, λ abstractions provide a mechanism of module creation for reuse.

The λ abstraction module approach is similar to ADFs 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 [27].

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

- λ abstractions are anonymous hence cannot be invoked by names. The reuse of λ abstractions is carried out by passing them as arguments to a higher-order function which then reuses the λ abstraction.
- λ abstraction subtrees are constructed using the same function set as that used to create the main program. The terminal set, however, consists only of the arguments of the λ abstraction to be created; no global variables are allowed in this

work. Argument naming in λ abstractions follows a simple rule: each argument is named with a hash symbol followed by an unique integer, for example #l, #2. In this way, each argument within a λ abstraction has a unique name. Meanwhile, crossover can be easily performed between λ abstractions with the same number of arguments. The following is an example `foldr` function with a λ abstraction (in bold) created as its first argument:

```
foldr (λ#1 λ#2 (+ #1 #2)) 10 [1,2,3]
```

- The determination of the program structure with λ abstractions is different from that with ADFs. Instead of having the program structure predefined in advance or completely open to evolutionary determination (see Section 3.2), λ abstractions are created *dynamically* by GP according to the function arguments specified in the higher-order functions. When a higher-order function is selected to construct a 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. For example, an important partial solution to a general sorting algorithm is the comparison of two values. One can specify a two-input one-output λ abstraction to be the argument of the higher-order function used to evolve a sorting algorithm.

The program structures of MA and ARL are also created and modified dynamically during GP runs. However, unlike λ abstractions, MA and ARL program modules are modified less frequently. Once created, a module is 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 reported that the MA approach, where modules are created using randomly extracted program fragments, does not provide performance advantages [22]. A recent study, however, shows that this random encapsulation of program subtree scheme improves upon standard GP [34].

4.3. Structure abstraction

The combination of higher-order functions and λ abstractions function arguments generates a special structural pattern in the evolved program trees. As a function argument to a higher-order function, a λ abstraction subtree is constrained to sit underneath the higher-order function in the program tree hierarchy (see Figure 3). Consequently, a higher-order function and its λ abstraction function argument group into a two-layer-hierarchy 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 holds throughout the evolutionary process. Structure abstraction is an unique property produced by the higher-order functions and λ abstractions program representation.

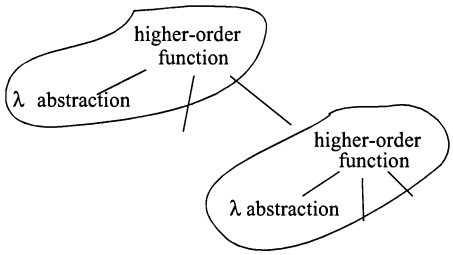

Figure 3. Structure abstraction in a program tree hierarchy.

4.4. Type system: Structure abstraction preserving engine

The structure abstraction grouping in the program trees has to be preserved during program evolution. This is achieved using a type system [46]. Initially, each function and terminal is specified with type information. Meanwhile, the input and output types of the program to be evolved are specified. This information is used by the type system to select type-matched functions and terminals to construct type-correct programs.

The type system support a type language with the following syntax.

$\sigma ::=$	τ	built-in type
$\mid$	υ	type variable
$\mid$	$\sigma_1 \rightarrow \sigma_2$	function type
$\mid$	$[\sigma]$	list of elements all of type σ
$\mid$	$(\sigma_1 \rightarrow \sigma_2)$	bracketed function type

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

ν::= $\texttt{dummy}_{\texttt{i}}$ | $\texttt{temporary}_{\texttt{i}}$

- Constants, such as 0, and terminals, such as `x`, are specified with a pre-defined type such as built-in type or a generic type variable.
- Functions are specified with function types. For example, the function `head` has the type [a] → a where `a` is a dummy type variable.
- Function arguments of a higher-order function are specified with the bracketed function type. For example, the higher-order function `foldr` is specified with the following type:

```
foldr :: (a→b→b)→b→[a]→b
```

  This type information indicates that the first argument is a function which takes two arguments and returns one value. During the creation of the initial population, each time `foldr` is selected to construct a program node, the type system ensures that a 2-input 1-output λ abstraction is generated as its function argument.

Notice that `foldr` is specified as a *polymorphic* function whose types contain *type variables* (e.g., `a` and `b`). The type system instantiates these type variables with type values each time the `foldr` function is selected to construct a program tree node. By supporting type variables in the type language, the generality of functions in the function set are enhanced. For example, `foldr` can be used to provide structure abstraction for many different types of arguments.

However, type variables may also make the search space larger than necessary. One example is when the type value of a type variable is known. In this case, replacing the type variable with the known type value can reduce the size of the search space (Although such a restriction does not necessarily make the evolutionary search easier). If no type variables are employed in the function and terminal sets, the type system performs monomorphic type checking, which is more restrictive than polymorphic typing in defining GP search space.

With one grammar to specify the syntax of λ abstraction (see Section 2.2) and one grammar to specify type information, this GP system seems to be similar to the logic-grammar based Generic Genetic Programming (GGP) system [44]. However, the two systems are different in the following ways:

- Our GP system uses a context-free grammar while GGP uses a logic grammar. In a logic grammar, each grammar symbol may include arguments, hence it allows more domain-dependent information to be expressed. As a result, the GGP system uses only one grammar to represent syntactic, type and semantic information, in contrast to the two grammars used here.
- Also, the logic grammar in GGP does not support module syntax. As a result, the system cannot create modular programs.

A survey of grammar based GP systems is provided in the Chapter 3 of [48].

4.5. Genetic operations

Within a program tree, each node is annotated with type information, which is used by the type system to generate type-correct programs during crossover and mutation. In particular, the "point-typing" method [26] is used to protect the higher-order function and λ abstraction structure grouping. With point-typing, crossover can only operate between nodes in the two main programs or between nodes in the same kind of λ abstraction (i.e., λ abstractions that represent the same function arguments to the same higher order function). In this way, structure abstraction can be preserved throughout the evolutionary process.

Genetic operations can also be performed on λ abstraction nodes. Each λ abstraction node is annotated with a type which indicates the number and type of its arguments. The type system limits the λ modular crossover to be performed between two λ abstractions which have the same number and type of arguments. Similarly, λ modular mutation replaces a λ abstraction with a newly created λ abstraction

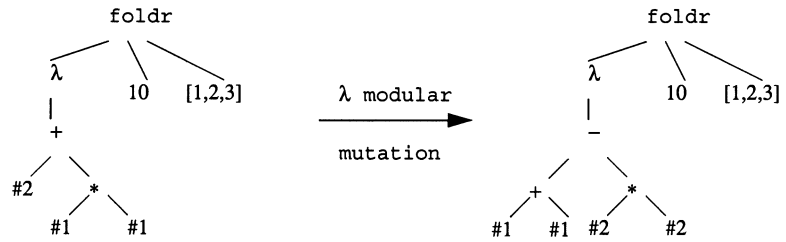

Figure 4. λ modular mutation operation.

that has the same number and type of arguments. An example of the λ modular mutation operation is shown in Figure 4.

λ modular operations add another "functional" flavor (besides polymorphism and higher-order functions) to the GP system: Functions are first-class in the programs, just like constants, variables and predefined functions; A λ abstraction node (representing a function) is therefore allowed to be replaced with another λ abstraction node. This principle will be applied later to view a λ abstraction function as one single node in a program tree. Details are given in Section 6.

5. The even-parity problem

The `even-parity` has been used by many researchers as a benchmark problem for GP to solve [8, 17, 26, 33, 44]. For N 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: $\{\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{N-1}\}$.
  These are the N Boolean inputs.

Rosca [37] and Poli, Page and Langdon [33] have identified two reasons why `even-parity` is a difficult problem for GP to solve:

- The problem solution is very sensitive to the value of the inputs. A single change of one of the N inputs would generate a different output.
- The function set does not contain the Boolean functions `xor` or `eq`. These two building blocks useful to the problem solution 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 [26]. His results also indicated 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. In this work, we will show that the new strategy described in Section 4 is capable to solve the general `even-parity` problem, i.e., it allows GP to generate a *general solution* which works for any value of N.

6. Initial experiments

The higher-order function selected to support structure abstraction for this problem is `foldr` (see Section 4.1 for its operation). Meanwhile, 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. These functions and terminals with their types are described in Table 1.

Table 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`	(a→b→b)→b→[a]→b
`head`	[a]→ a
`tail`	[a]→[a]
`L`	[bool]

6.1. Control parameters

As mentioned in Section 4.5, the GP system treats λ abstraction functions in the same way as other values in the program trees. A λ abstraction function is therefore considered to be one single node in the program tree, just like a constant, a variable or a function specified in the function set. For example, in Figure 5, the λ abstraction subtree inside the dashed line is considered as one single node. Consequently, this program is of tree depth 3.

The nested λ abstraction inside the dashed line is the result of polymorphic `foldr`, whose type variables can be instantiated with different type values. In Figure 5, the type variables of the `foldr` function inside the square box are instantiated in the following way:

`foldr` :: (a→b→b)→b→[a]→b
(bool→[bool]→[bool])→[bool]→[bool]→[bool]

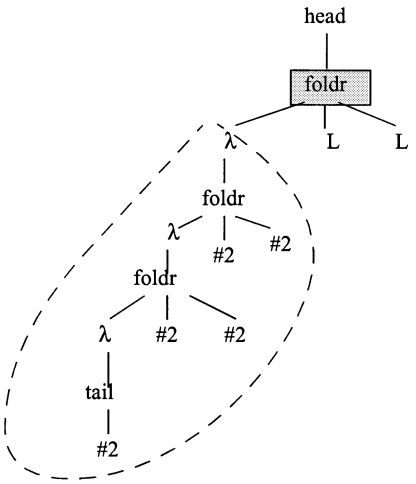

Figure 5. A program with nested λ abstractions.

That is `a` is instantiated to Boolean while `b` is instantiated to list of Boolean. The function argument has type (bool → [bool] → [bool]), which allows `foldr` to be used to construct the λ abstraction subtree. Consequently, nested λ abstractions are created. Since a λ abstraction is considered as one single node in the program tree, a program may contain a λ abstraction of any number of nodes (11 in Figure 5) and still satisfy the tree depth limit. With such a setup, only a small depth limit is necessary to obtain large program trees (we use 4 in this study). For other problems where nested λ abstractions do not occur, a larger tree depth limit might be required.

The nested `foldr` in the program trees not only produced nested λ abstractions but also nested recursion: `foldr` applies the λ abstraction function iteratively to every element of the input list. Nested recursive programs require a considerable amount of time and space to evaluate. It is therefore necessary to restrict the depth of the nested recursion. Once this limit is reached during program creation, `foldr` is excluded from being used further to construct λ abstractions. In this study, the recursion limit is 100. This means that each program tree can only have a maximum of 100 `foldr` nodes. It also means that the number of λ abstraction permitted in a program tree is 100.

Crossover is the only kind of genetic operations applied in this study. As described in the previous section, a crossover point can be in the main program body, in a λ abstraction body or be a λ abstraction node (λ modular cross-over). Moreover, the selection of crossover points is random, where each node has an equal opportunity to be selected. Since there are possibilities to generate very large program trees (due to the nested λ abstractions), this set of crossover operations seems to be able to provide all that is needed for evolutionary search (micro and macro mutations). Indeed, experimental results show that they are very effective with this problem. However, it is possible that other kinds of genetic operators, such as mutation and reproduction, are beneficial to the search.

Table 2 summaries the control parameters used in the experiments.

Table 2. Summary of control parameters

Parameter	Value
Population size	500
Maximum number of generation	51
Maximum main program tree depth	4
Maximum abstraction tree depth	4
Nested recursion limit	100
Crossover rate	100%

6.2. Test cases

The test cases of `even-2-parity` (2 Boolean inputs) and `even-3-parity` (3 Boolean inputs) are selected to evaluate the generated programs. Since each input can have value either True or False, there are $2^2 + 2^3 = 12$ 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 any odd or even number. The test cases of `even-2-parity` help GP to learn to handle an even number of inputs while the test cases of `even-3-parity` train GP to work on an odd number of inputs. With this set of test cases, it will be shown that the generated programs are general solutions which work for any value of `N`.

6.3. Fitness function

The fitness function used is the same as that used by Koza [26] except that an empty-list run-time error is penalized. When the function `head` or `tail` is applied to an empty list, the return value can not be defined. To handle such an error during program evaluation, a default value is returned and the evaluation continues. Meanwhile, this error is reflected in the program's fitness.

The fitness of a program is calculated as follows: when the program produces the correct result for a test case, it receives a 1; otherwise, it receives a 0. If the empty-list error is flagged during the program evaluation, its 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 correct general `even-parity` program receives a fitness of 12.

6.4. Results

Fifty runs were made and 40 of them found a correct solution. Moreover, all 40 are general solutions which work for any number of inputs. The complete list of

the 40 solutions is in Appendix A. Among them, the two most frequently generated programs are:

```
foldr xor (nor (head L) (head L)) (tail L)
foldr xor (nand (head L) (head L)) (tail L)
```

These two programs are equivalent to:

```
foldr xor (not (head L)) (tail L)
```

Tables 3, 4, and 5 give the evaluation results of this program on `even-2,3,4-parity` test cases (see Section 4.1 for the operation of `foldr`). With `even-3` and `even-4`, it becomes clear that

```
even-3-parity L = [odd-2-parity (tail L)] xor [not (head L)]
even-4-parity L = [odd-3-parity (tail L)] xor [not (head L)].
```

Using deduction, the following equation is derived:
`even-N-parity L = [odd-(N-1)-parity (tail L)] xor [not (head L)]`, when `N >2`.

Table 3. Even-2-parity test cases evaluation

I_1	I_2	foldr xor (not (head L)) (tail L)	Output
T	T	[T] xor F	T
T	F	[F] xor F	F
F	T	[T] xor T	F
F	F	[F] xor T	T

Table 4. Even-3-parity test cases evaluation

I_1	I_2	I_3	foldr xor (not (head L)) (tail L)	Output
T	T	T	[T xor T] xor F	F
T	T	F	[T xor F] xor F	T
T	F	T	[F xor T] xor F	T
T	F	F	[F xor F] xor F	F
F	T	T	[T xor T] xor T	T
F	T	F	[T xor F] xor T	F
F	F	T	[F xor T] xor T	F
F	F	F	[F xor F] xor T	T

Table 5. Even-4-parity test cases evaluation

I_1	I_2	I_3	I_4	foldr xor (not (head L)) (tail L)	Output
T	T	T	T	[T xor T xor T] xor F	T
T	T	T	F	[T xor T xor F] xor F	F
T	T	F	T	[T xor F xor T] xor F	F
T	T	F	F	[T xor F xor F] xor F	T
T	F	T	T	[F xor T xor T] xor F	F
T	F	T	F	[F xor T xor F] xor F	T
T	F	F	T	[F xor F xor T] xor F	T
T	F	F	F	[F xor F xor F] xor F	F
F	T	T	T	[T xor T xor T] xor T	F
F	T	T	F	[T xor T xor F] xor T	T
F	T	F	T	[T xor F xor T] xor T	T
F	T	F	F	[T xor F xor F] xor T	F
F	F	T	T	[F xor T xor T] xor T	T
F	F	T	F	[F xor T xor F] xor T	F
F	F	F	T	[F xor F xor T] xor T	F
F	F	F	F	[F xor F xor F] xor T	F

6.5. *Performance evaluation*

To facilitate direct comparison with others' work, we have adopted the most widely used measurement method within GP field, the "effort" requirement described in [26], to evaluate the performance of the new strategy. The results are presented in Figure 6.

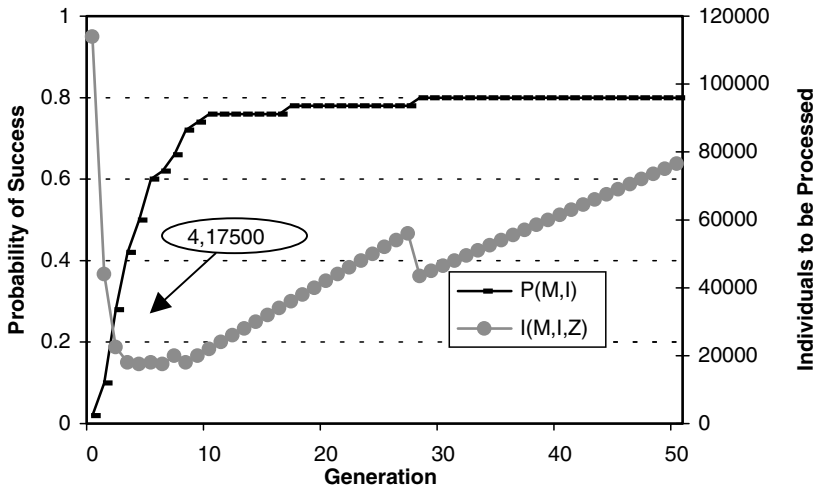

Figure 6. Performance curves for the general even-parity problem.

Table 6. Performance summary for the even-parity problem

Results	Higher-order function + abstractions	Generic genetic programming	GP with ADFs
Programs	General even-parity	General even-parity	Even-7-parity
Runs/success	50/40	60/17	29/10
Minimum I(M, i, z)	17,500	220,000	1,440,000
Number of fitness cases	12	8	128
Fitness cases processed	210,000	1,760,000	184,320,00

The "effort' curve, $I(M, i, z)$, indicates the number of program evaluations required at each generation to find a correct solution with 99% confidence. This value is calculated using the following formula given in [26]:

$$I(M, i, z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M, i))} \right\rceil, \quad \text{where } z = 99\%. \tag{1}$$

The smallest value on this curve is used to indicate the minimum effort required for GP to solve the given problem. According to the experimental results, the curve of I(M, i, z) reaches a minimum value of 17,500 at generation 4 (marked on the figure). Since 12 test cases were used to test each program, the number of test cases processed was 210,000.

Table 6 summarizes the experimental results with two other related works. The ADF module mechanism has been applied to solve the `even-11-parity` problem. However, performance details were not reported. The results of ADF work on the `even-7-parity` is therefore given instead. Another work is the Generic Genetic Programming (GGP) system which uses recursion to solve the same problem [44]. Although these two techniques have implementations (representation, fitness cases etc.) that are different from those used here, their goal is similar to this work: to improve both the accuracy of the solution and the efficiency of the evolutionary process. The technique of higher-order function and λ abstractions has achieved this goal better than the other two methods. However, would the technique be effective with other problems? This leads to the next step of our study: to investigate the applicability of the technique to general problems.

7. Investigation of the applicability of the new technique

There are different ways to investigate the applicability of a technique to general problems. One is to apply the technique to many different problems. However, this experimental-based method raises an important question: If a technique works for problem A, B, C and D, would it also work for problem E, F and G? Another approach is to give an in-depth analysis of how the technique solves a particular problem. Such understanding provides useful information for the general usage of the technique. This study takes the second approach by conducting a rigorous

analysis of the search space during GP program evolution. Based on the results, suggestions on the application of the technique will be discussed.

The search space of GP consists of all program trees that can be composed using the available functions and terminals. Since a program tree is defined by two elements (structure and contents), we will use these two elements to analyze the search space. In Section 7.1, the evolution of `foldr` program structure is used to identify the promising area of search space. This search space is farther analyzed in Section 7.2, according to the number of `foldr` structure in the programs. In Section 7.3, program contents of all solutions are studied. A second set of experiments is provided in Section 7.4 to verify the analysis.

7.1. Evolution of program structure

As a first step to understand the characteristics of the program search space, 25,000 programs were randomly generated. These programs contain different numbers of `foldr` structure abstraction groupings, which are summarized in Table 7.

Table 7. Results of the 25,000 randomly generated even-parity programs

No. of foldr structure abstraction groupings in the program	No. of programs generated	Percentage	No. of program with above average fitness
0	4,787	19%	4,787
1	9,233	37%	8,392
2	3,101	12.4%	685
More than 2	7,879	31.6%	3

As shown in the second column, the number of programs with 1 `foldr` is more than the number of any other type of program structure. A close examination of the function set tells us that `foldr` is only one out of seven functions that can be used to construct program tree nodes. With the tree depth limit of 4, it is highly possible that `foldr` is selected once during the creation of a program tree.

The fourth column in Table 7 gives the number of programs whose fitness is above average fitness of the 25,000 randomly generated programs. As shown, those programs which receive above average fitness are with either 2 or 1 or no `foldr` (only 3 exceptions). With a fitness proportionate selection scheme, this set of programs are favored and are more likely to be selected for reproduction. In Appendix A, the generated 40 correct programs have either 1 or 2 occurrences of `foldr`. This suggests that structure abstraction has helped to identify the program structure of good solutions at generation zero. Most evolutionary effort is devoted to search for the content to fill in the program structures.

To confirm this hypothesis, another experiment is conducted. In this experiment, 50 GP runs were made and the evolution of program structures (in terms of the

number of `foldr` structure abstraction groupings in the program) is recorded. The results are shown in Figure 7.

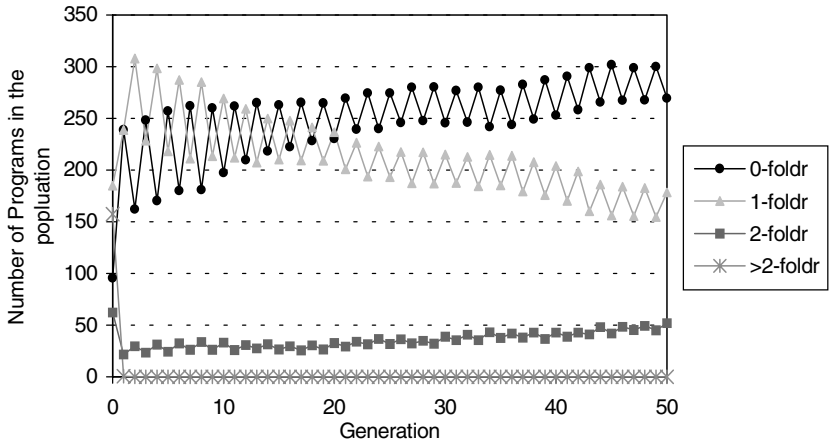

Figure 7. The evolution of `foldr` program structure grouping.

At generation zero, different kinds of program structures were created and their proportion is very similar to the results of the previous experiment (see Appendix B). However, once the GP evolution process begins, the number of programs with more than 2 `foldr` decreases dramatically. At generation 3, the population contained no program with more than two `foldr`. GP evolution became a competition process among programs with 0, 1 or 2 `foldr`.

These results confirm our hypothesis that structure abstraction has helped identifying the promising area of the search space (those programs with 1 or 2 or no `foldr`) at generation zero. Consequently, the evolutionary efforts required to find a program solution is reduced. However, once the population is within the promising area of the search space, how is program evolution performed? This is the focus of the following section.

7.2. Analysis of the promising area of search space

The promising area of the search space contains programs with either 0, 1 or 2 `foldr`. During the previous experiments, it has been observed that this set of programs has their polymorphic functions (`foldr`, `head` and `tail`) instantiated in a similar manner: their type variables are consistently instantiated to Boolean type, i.e., they are used as monomorphic functions:

$$\begin{aligned} \texttt{foldr} &:: (\text{bool} \rightarrow \text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \rightarrow [\text{bool}] \rightarrow \text{bool} \\ \texttt{head} &:: [\text{bool}] \rightarrow \text{bool} \\ \texttt{tail} &:: [\text{bool}] \rightarrow [\text{bool}] \end{aligned}$$

Using these monomorphic functions to create program trees, we have constructed the promising area of the search space. Meanwhile, the program structures are analyzed.

Program structure is the shape of a program tree. With λ abstractions, the program structure can become very complicated (see Figure 5 for example). We approach this problem by considering a λ abstraction as a single node. This node is labeled with the output of the λ abstraction. In other words, the contents of the λ abstraction are ignored in this phase. Once all program structures are identified, the λ abstraction contents will be studied in the next section.

The most complicated element of a program structure is higher-order functions, whose λ abstraction arguments can generate many different outputs. We therefore start the study with the `foldr` higher-order function program structure. A `foldr` program tree can only be constructed in some restricted way due to the specified type information. 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. As mentioned in Section 4.2, the terminal set used to construct a λ abstraction only contains arguments to the λ abstraction, i.e., two Boolean values in this case. Since there is no terminal with the type of Boolean list that can be used to construct the λ abstraction, those functions (`head`, `tail` and `foldr`) which require a 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 (`foldr, head, and, or, nand` and `nor`) and can be selected to build the internal nodes of the tree. However, there is only one terminal `L` which can be used to construct the leaves of the subtree. As specified, the type of `L` is 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 terminals, 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 8 shows the program structure for `foldr`. An * on the node indicates the node is optional. We partitioned the search space according to the number of `foldr` in program trees. Our study shows that in order to satisfy the tree depth limit of 4, a program tree can only have either 2, 1 or no `foldr`. These program trees will be listed in Figure 9, 10, and 11.

Figure 9 shows all the possible program structures containing 2 `foldr`. The program tree of Figure 9(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.) The program tree of Figure 9(b) represents 4,096 program structures. In total, there are 5,632 program structures containing 2 `foldrs` in the search space. Figure 10 shows program structures with 1 `foldr`. The program tree of Figure 10(a) represents 192 program structures. The program tree of Figure 10(b) represents

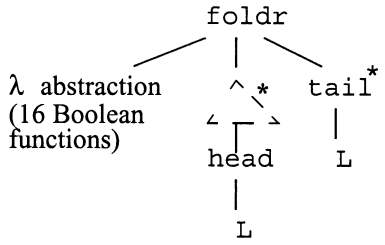

Figure 8. A `foldr` program tree structure.

1,024 program structures. The program tree of Figure 10(c) represents 96 program structures. The program tree of Figure 10(d) represents 512 program structures. In total, there are 1,824 program structures with 1 `foldr` in the search space.

Figure 11 shows program structures without `foldr`. Figure 11(a) represents 64 program structures. Figure 11(b) represents 64 program structures. Figure 11(c) represents 3 program structures and Figure 11(d) represents 16 program structures. In total, there are 147 program structures without `foldr` in the search space.

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

In general, any search space that contains programs with higher-order functions can be analyzed using the same methodology. There are three steps in this method:

- identify possible λ abstraction outputs;
- construct the higher-order function program structure;
- using the higher-order function structure as a building block to analyze the entire search space within the tree depth limit.

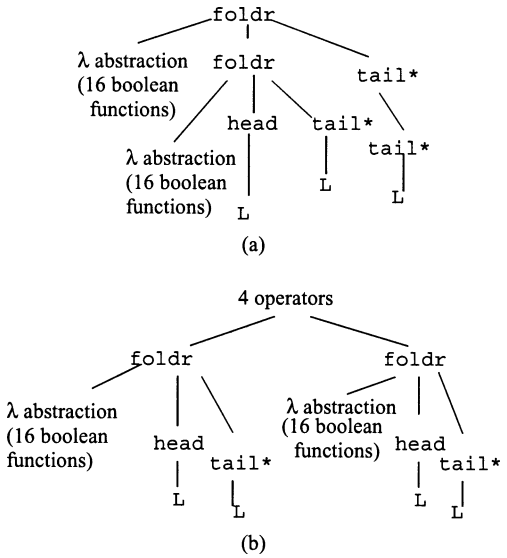

Figure 9. Program structures with 2 `foldr` higher-order functions in program trees.

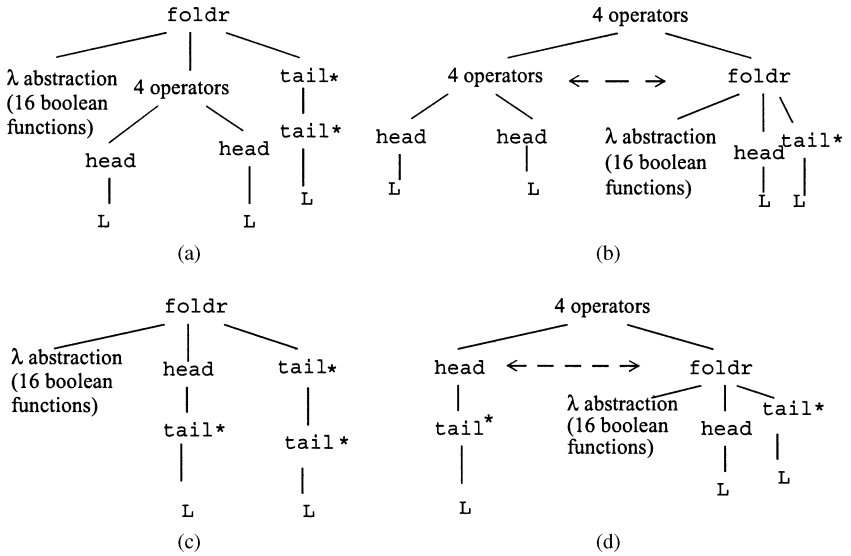

Figure 10. Program structures with 1 `foldr` higher-order function in program trees.

We are researching using a different higher-order function to evolve the general sorting algorithm.

7.2.1. Fitness distribution The 7,603 program structures in the search space are capable of solving the general `even-parity` problem to a different degree. With the designed test cases and fitness function (see Section 6.2 and 6.3), a program can have fitness value range from 0 to 12. Note that there is only one program in the search space that produces an empty-list error: `head (tail (tail L))`. This

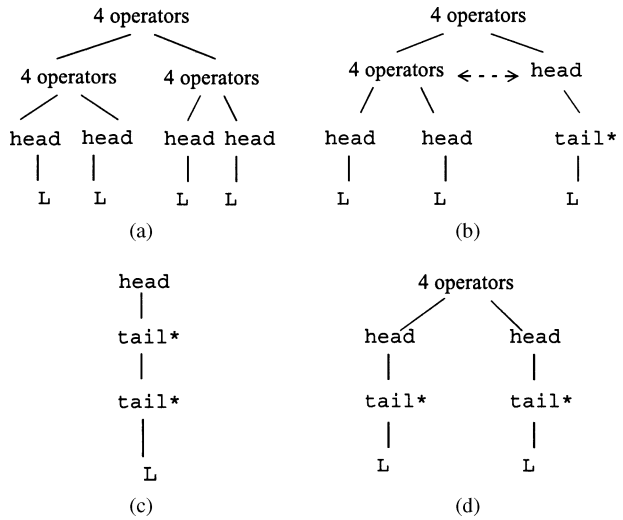

Figure 11. Program structures without `foldr` higher-order function in program trees.

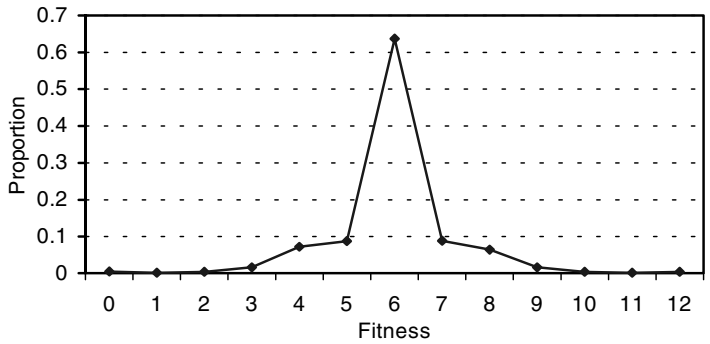

Figure 12. Fitness distribution in the search space.

program would receive negative fitness when evaluated against the test cases of `even-2-parity` (which has 2 inputs). However, its fitness is the summation of all 12 test cases fitness, which is 2.

Figure 12 shows the fitness distribution. More than 60 percent of program structures are of fitness 6. The number of program structures with other fitness falls dramatically off on either side of the peak. There we only 29 program structures which score 12 and are correct 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 intends 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.

The shape of the fitness distribution is very similar to those reported by other researchers on the same problem (`even-5-parity` and `even-6-parity`) [31, 38]. As analyzed by Langdon and Poli, with this kind of search space, a search algorithm finds either a solution or one that scores half marks. As a result, heuristic search algorithm, such as GP, will not outperform random search [30]. Yu and Miller showed that by increasing neutrality in the even-3-parity search space, the evolvability of the fitness landscape is increased [47].

7.3. Solutions in the promising area of search space

The 29 solutions structures in the promising area of search space are split into 11 groups (Tables 9–19). Programs in each group have the same structure but different λ abstraction values (Boolean functions). The probability to find each solution is calculated based on the program structure and its λ abstraction Boolean function values.

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 the previous section, the probability to find the program structure for each solution can be calculated. Meanwhile, the search of the 16 Boolean functions has been studied by Koza [26]. For each of the 16 Boolean functions (he called them Boolean rules), Koza did random search using a program tree with 31 nodes. The results are summarized in Table 8. We follow Koza and call them Boolean rules.

Table 8. 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

The easiest rules to find are rule 0 (always off) and rule 15 (always on). On average, they can be found by randomly generating 4.8 program trees. The most difficult one is rule 6 (`odd-2-parity/not-equal/xor`). It requires generating 846 program trees to find this rule. Similarly, rule 9 (`even-2-parity/equal`) requires generating 821 program trees before a correct one is found. We used this table to measure the probability to find each of the 16 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 l/4.8.

Tables 9–19 present the probability to find the 29 solutions. The caption gives the solution. Columns 1 and 2 are the Boolean rules for the λ abstractions. Columns 3 and 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 multiplying columns 3, 4 and 5. Table 19 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 9. foldr λ_1 (foldr λ_2) (head L) (tail L) (tail(tail L))

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	P(str)	$P(s)$
r3	r9	1/32	1/821	256/7603	1.28e-6

Table 10. foldr λ_1 (foldr λ_2 (head L) (tail L)) L

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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

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 [26].

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

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

Based on the study, a general even-parity solution can be found by randomly evaluating approximately 18,000 programs.

7.4. Further experiments

To verify the analysis in the previous sections, we conducted GP experiments to search solutions within the promising area of the original search space. This space contains programs trees which are constructed using monomorphic functions. The experimental setup is identical to that described in Section 6 except the 3 polymorphic functions (`foldr, head` and `tail`) are implemented as monomorphic.

Figure 13 gives the results based on 50 runs. All 50 runs find a correct solution. Among them, 7 runs find a solution at generation zero. The probability of success curve, P(M, i), starts as 14% at generation zero and reaches 100% at generation 27. The "effort" curve, I(M, i, z) is at its minimum at generation zero, i.e., a solution to the general `even-parity` problem can be found by evaluating 15,500 programs which are randomly generated at generation zero.

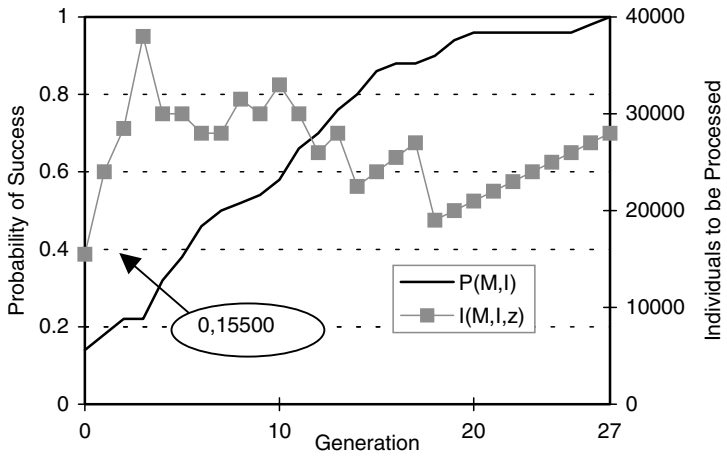

Figure 13. Performance curves for the general even-parity problem

Table 11. foldr λ_1 (foldr λ_2 (head L) L) (tail L)

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 12. foldr λ_1 (foldr λ_2 (head L) L) L

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 13. nand (foldr λ_1 (head L) L) (foldr λ_2 (head L) (tail L))

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 14. nand (foldr λ_1 (head L) (tail L)) (foldr λ_2 (head L) L)

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 15. nand (foldr λ_1 (head L) (tail L)) (foldr λ_2 (head L) (tail L))

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 16. nor (foldr λ_1 (head L) (tail L)) (foldr λ_2 (head L)(tail L))

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 17. nor (foldr λ_1 (head L) L) (foldr λ_2 (head L)(tail L))

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 18. nor (foldr λ_1 (head L)(tail L)) (foldr λ_2 (head L)L)

λ_1	λ_2	$P(\lambda_1)$	$P(\lambda_2)$	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 19. foldr λ_1 (fun (head L)(head L)) (tail L)

λ_1	fun	$P(\lambda_1)$	P(str)	$P(s)$
r6	nand	1/846	16/7603	2.48e-6
r6	nor	1/846	16/7603	2.48e-6

8. Discussion

As a subset of the original search space, programs with either 2, 1 or no `foldr` are the promising area within the search space. The analysis (Section 7.2 and 7.3) and experimental results (Section 7.4) indicate that within this search space, a solution to the general `even-parity` problem can be found by random generation of approximately 15,500 program trees. This result raises two important questions:

- Why structure abstraction allows the solution to be found so easily?
- Why random search outperforms GP search in this search space?

These two questions are addressed in the following subsections.

8.1. Impacts of structure abstraction

There are 7,603 program structures in the search space. With all the λ abstractions expended into all possible subtrees, this search space is undoubtedly very large. Moreover, the density of the solution in the search space is very low (0.38%). Yet, a solution to the problem can be found by random generation of 15,500 programs. This suggests that the difficulty of a problem is independent to the density of solutions in the search space. Instead, it relies on how easily these solutions can be found. The 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 (see Tables 9–19). This means that the module mechanism of λ abstraction, which allows partial solutions to be evolved, is very important for search. However, using partial solutions alone, Langdon and Poli have shown that the generation of an overall solution is still impossible [30]. 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 generation 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 generated and manipulated to form a bigger solution. This hierarchical processing is shown to be very effective in solving the general `even-parity` problem.

8.2. Random search versus GP search

The fitness distribution (Figure 12) shows that program structures with fitness 6 occupy 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. As explained in Section 7.2.1, this type of search space is very difficult for GP to outperform random search.

It has also been suggested that with this kind of search space, population-based search without mutation would perform worse than random search, as genetic drift in a smaller population means the population may loose one or more primitive [31]. The 100% crossover rate used in the experiments might have influenced the effectiveness of GP search.

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

9. Limitations

This study has shown that the new strategy of using higher-order functions and λ abstraction to evolve recursive and modular programs is very effective with the general `even-parity` problem. Initially, `foldr` structure abstraction identifies the promising area of the search space. Once the population is within the promising area, `foldr` structure abstraction provides hierarchical processing for search. As a result, the solution can be found very efficiently.

However, to use the method effectively, the following details have to be considered:

- Different problems have different recursion patterns, hence require different higher-order functions to be effective. For example, the recursion pattern of `even-parity` is carried around the input list. The selected `foldr` higher-order function provides effective implicit recursion for this problem. However, with the Fibonacci sequence problem, 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. On suggestion is a higher-order function that takes an integer index as its argument:

  `loop-n-times : (int → int) → int → int`

  In this way, the λ abstraction function (the first argument) can be executed many times, as specified by the index argument. This operation is defined and controlled in the `loop-n-times` higher-order function.
- Different problems have different module patterns (i.e., the partial solutions and their relationship to the overall solution), hence require different higher-order functions to be effective. For example, the module pattern in the `even-parity` was successfully captured by the `foldr` λ abstraction function argument. With different problems, such as the artificial ant [26], our study shows no such performance gain, possibly due to the higher-order function selected was not able to identify partial solutions to the problem [48].

Although the concept of using higher-order functions and λ abstraction to evolve recursive and modular programs can be applied to general problems, only the properly designed higher-order functions can receive the benefits.

Presently, we continue this research by using a different higher-order function to evolve quick sort [20]. The designed higher-order function used for this problem `partition` has the following type:

```
partition : ([int] → int → [int]) → [int] → [int] → [int]
```

The first argument to `partition` is a λ abstraction function. The λ abstraction is a module (sub-solution) that takes two integer inputs (a list and the pivot) and returns the same list input but in sorted order.

10. Conclusions

Implicit recursion supported by higher-order functions allows GP to evolve recursive programs without dealing with non-terminating recursive calls. Moreover, when combined with λ abstraction, higher-order functions present a unique program pattern, structure abstraction, which can make GP more effective on some problems. For the general `even-parity` problem, the `foldr` structure abstraction has helped to identify the promising area of search space at the generation zero. Once the population is within the promising area, `foldr` structure abstraction provided hierarchical processing for search. Consequently, the solution to the general `even-parity` problem was found very efficiently.

However, the benefits of structure abstraction can only be obtained with a suitable higher-order function to the problem. This is due to the fact that different problems have different recursion and module patterns. Hence, a higher-order function that is effective with one problem is not necessary advantageous to another. Similar to the No-Free-Lunch Theorem on search algorithms [43], the author believes that representation is also problem dependent. It is only when the appropriate selections are made that the power of a search algorithm shows.

In this work, we have presented a detailed analysis of how the `foldr` structure abstraction assists GP searching for the general `even-parity` solutions. Based on

Table 20. Generated correct general even-partity program

Number of programs	General even-parity program
20	foldr *r6* (nor(head L) (head L))(tail L)
12	foldr *r6* (nand (head L) (head L)) (tail L)
2	nand (foldr *r15* (head L)(tail L)) (foldr *r6* (head L) (tail L))
1	nand (foldr *r6* (head L)(tail L)) (foldr *r14* (head L)(tail L))
1	nand (foldr *r14* (head L) (tail L)) (foldr *r6* (head L) (tail L))
1	nor (foldr *r6* (head L)(tail L)) (foldr *r6*(head L)(tail L))
1	nor (foldr *r6* (head L)(tail L)) (foldr *r4* (head L)(tail L))
1	foldr *r3*(foldr *r9*(head L)(tail L)) (tail (tail L))

the information, the general principles of designing problem-specific higher-order functions are suggested. We will continue this research by applying the principles to design effective higher-order function for different problems.

Appendix A

The 40 generated correct programs can be divided into 9 groups (see Table 20). 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 [26] and present them in *italics*, i.e., *r1* to *r16* (Table 8). Note that those λ abstractions which compute the same Boolean function might contain very different code.

Appendix B

Table 21. Results of the initial population in 50 GP runs with population size 500

No. of foldr structure abstraction groupings in the program	Average	Percentage	Standard deviation
0	95.6	19%	2.59
1	184.8	37%	6.77
2	62.3	12.5%	4.45
More than 2	157.3	31.5%	4.10

Acknowledgments

I would like to thank Wolfgang Banzhaf and the reviewers for their comments and suggestions. I also thank Bill Langdon for his constant support of my work.

References

1. P. J. Angeline and J. Pollack, "The evolutionary induction of subroutines," The Fourteenth Annual Conference of the Cognitive Science Society, Lawrence Erlbaum: Bloomington, Indiana, 1992, pp. 236–241.
2. P. J. Angeline and J. Pollack, "Evolutionary module acquisition," in Proc. Second Annual Conf. Evolutionary Programming, D. B. Fogel and W. Atmar (eds.), Evolutionary Programming Society: La Jolla, CA, 1993, pp. 154–163.
3. P. J. Angeline, "Genetic programming and emergent intelligence," in Advances in Genetic Programming, K. E. Kinnear, Jr. (ed.), MIT Press: Cambridge, MA, 1994, pp. 75–98.
4. P. J. Angeline, "A historical perspective on the evolution of executable structures," Fundamenta Informaticae, vol. 36(1–4), pp. 179–195, 1998.
5. W. Banzhaf, D. Banscherus, and P. Dittrich, "Hierarchical genetic programming using local modules," Reihe cl 56/98. SFB 531, University of Dortmund, 1998.
6. A. S. Bickel and R. W. Bickel, "Tree structured rules in genetic algorithms," in Genetic Algorithms and Their Applications, Proc. Second Int. Conf. Genetic Algorithms, J. J. Grefenstette (ed.), Lawrence Erlbaum: Cambridge, MA, pp. 77–81, 1987.
7. 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, 1996, pp. 203–219.

8. K. Chellapilla, "Evolving computer programs without subtree crossover," IEEE Trans. Evolutionary Comput., vol. 1(3), pp. 209–216, 1997.
9. N. L. Cramer, "Representation for the adaptive generation of simple sequential programs," in Proc. Int. Conf. Genetic Algorithms and Their Applications, J. J. Grefenstette (ed.), Lawrence Erlbaum: Hillsdale, NJ, 1985, pp. 183–187.
10. A. Dessi, A. Giani, and A. Starita, "An analysis of automatic subroutine discovery in genetic programming," in Proc. Genetic and Evolutionary Comput. Conf., W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Hanavar, M. Jakiela, and R. Smith (eds.), Morgan Kaufmann: Los Altos, CA, 1999, pp. 996–1001.
11. Discipulus, Register Machine Learning Technologies, Inc. Littleton, CO, 1998.
12. L. J. Fogel, A. J. Owens, and M. J. Walsh, Artificial Intelligence through Simulated Evolution, Wiley: New York, 1966.
13. R. M. Friedberg, "A learning machine: Part I," IBM J. Research and Development, vol. 2(1), pp. 2–13, 1958.
14. R. M. Friedberg, B. Dunham, and J. H. North, "A learning machine: Part II," IMB J. Research and Development, vol. 3, pp. 282–287, 1959.
15. C. Fujiki, An evaluation of Holland's genetic operators applied to a program generator, Master's thesis, University of Idaho, Moscow, ID.
16. C. Fujiki and J. Dickinson, "Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma," in Genetic Algorithms and Their Applications, Proc. Second Int. Conf. Genetic Algorithms, J. J. Grefenstette (ed.), Lawrence Erlbaum: Hillsdale, NJ, 1987, pp. 236–240.
17. C. Gathercole and P. Ross, "Tackling the boolean even n parity problem with genetic programming and limited-error fitness," in Genetic Programming 1997, Proc. Second Ann. Conf., Stanford University, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), Morgan Kaufmann: Los Altos, CA, 1997, pp. 119–127.
18. 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, 1994, pp. 391–401.
19. J. F. Hicklin, Application of the genetic algorithm to automatic program generation, Master Thesis, University of Idaho, Moscow, ID, 1986.
20. C. A. R. Hoare, "Algorithm 63, Partition, Algorithm 64, Quicksort," Commun. ACM, vol. 4, p. 321, 1961.
21. P. Hudak, "Conception, evolution, and application of functional programming languages," ACM Comput. Surveys, vol. 21(3), pp. 359–411, 1989.
22. 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, 1994, pp. 119–141.
23. 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," in Genetic Programming 1996, Proc. First Ann. Conf., Stanford University, CA, J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo (eds.), MIT Press: Cambridge, 1996, MA, pp. 132–149.
24. J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in Proc. 11th Int. Conf. Artificial Intell., Detroit, MI, N. S. Sridharan (ed.), vol. I, Morgan Kaufmann: Los Altos, CA, 1989, pp. 768–774.
25. J. R. Koza, "Genetically breeding populations of computer programs to solve problems in Artificial Intelligence," in Proc. Second Int. Conf. Tools for AI, IEEE Computer Society Press: Herndon, Virginia, 1990, pp. 819–827.
26. J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press: Cambridge, MA, 1992.
27. J. R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press: Cambridge, MA, 1994.
28. 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, 1994, pp. 99–117.
29. J. R. Koza, "Evolving the architecture of a multi-part program in genetic programming using

architecture-altering operations," in Evolutionary Programming IV, Proc. Fourth Ann. Conf. Evolutionary Programming, San Diego, CA, J. R. McDonnell, R. G. Reynolds, and D. B. Fogel (eds.), MIT Press: Cambridge, MA, 1995, pp. 695–717.
30. 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.
31. W. B. Langdon, "Scaling of program fitness space," Evolutionary Comput., vol. 7(4), pp. 339–421, 1999.
32. N. J. Nillson, Principles of Artificial Intelligence, Morgan Kaufmann: Los Altos, CA, 1980.
33. 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," in Proc. Genetic and Evolutionary Comput. Conf., Orlando, Florida, W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Hanavar, M. Jakiela, and R. Smith (eds.), Morgan Kaufmann: Los Altos, CA, 1999, pp. 1162–1169.
34. S. C. Robert, D. Howard, and J. R. Koza, "Evolving modules in genetic programming by subtree encapsulation," in Proc. Fourth European Conf. Genetic Programming, Lake Como, Italy, J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (eds.), Springer: Berlin, 2001, pp. 160–175.
35. J. P. Rosca and D. H. Ballard, "Hierarchical self-organization in genetic programming," in Proc. Eleventh Int. Conf. Machine Learning, Morgan Kaufmann: Los Altos, CA, 1994, pp. 251–258.
36. 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.), MIT Press: Cambridge, MA, 1996, pp. 177–201.
37. J. P. Rosca, "Genetic programming exploratory power and the discovery of functions," in Evolutionary Programming IV, Proc. Fourth Ann. Conf. Evolutionary Programming, San Diego, CA, J. R. McDonnell, R G. Reynolds and D. B. Fogel (eds.), MIT Press: Cambridge, MA, 1995, pp. 719–736.
38. J. P. Rosca, Hierarchical Learning with Procedural Abstraction Mechanisms, Ph.D. Thesis, University of Rochester, Rochester, NY.
39. 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, 1996, pp. 137–154.
40. A. M. Turing, "On computable numbers, with an application to the entscheidungs problem," in Proc. London Math. Soc., Series 2, vol. 42, 1936–1937, pp. 230–265.
41. P. A. Whigham and R. I. McKay, "Genetic approaches to learning recursive relations," in Progress in Evolutionary Computation, Yao, X. (ed.), Springer-Verlag: Heidelberg, Germany, 1995, pp. 17–27.
42. P. A. Whigham, Grammatical Bias for Evolutionary Learning, Ph.D. Thesis, University of New South Wales, Australian Defence Force Academy, 1996.
43. D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," IEEE Trans. Evolut. Comput., vol. 1(1), pp. 67–82, 1997.
44. M. L. Wong and K. S. Leung, "Evolving recursive functions for the even-parity problem using genetic programming," in Advances in Genetic Programming II, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 222–240.
45. T. Yu and P. Bentley, "Methods to evolve legal phenotypes," in Fifth Int. Conf. Parallel Problem Solving from Nature, A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel (eds.), Springer: Amsterdam, 1998, pp. 280–291.
46. T. Yu and C. Clack, "PolyGP: a polymorphic genetic programming system in Haskell," in Genetic Programming 1998, Proc. Third Ann. Conf., University of Wisconsin, Madison, WI, 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.), pp. 416–421, Morgan Kaufmann: Los Altos, CA, 1998.
47. T. Yu and J. Miller, "Neutrality and the evolvability of boolean function landscape, " in Proc. Fourth European Conf. Genetic Programming, Lake Como, Italy, J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi and W. B. Langdon (eds.), Springer: Berlin, 2001, pp. 204–217.
48. T. Yu, An Analysis of the Impact of Functional Programming Techniques on Genetic Programming, Ph.D. Thesis, University College London, London, United Kingdom, 1999.