Recursion, Lambda Abstractions and Genetic Programming

Tina Yu and Chris Clack

Department of Computer Science
University College London
Gower Street, London WC1E 6BT, U. K.
T.Yu@cs.ucl.ac.uk C.Clack@cs.ucl.ac.uk
http://www.cs.ucl.ac.uk/staff/t.yu http://www.cs.ucl.ac.uk/staff/c.clack

ABSTRACT

Module creation and reuse are essential for Genetic Programming (GP) to be effective with larger and more complex problems. This paper presents a particular kind of program structure to serve these purposes: modules are represented as λ abstractions and their reuse is achieved through an implicit recursion. A type system is used to preserve this structure. The structure of λ abstraction and implicit recursion also provides *structure abstraction* in the program. Since the GP paradigm evolves program *structure* and *contents* simultaneously, structure abstraction can reduce the search effort for good program structure. Most evolutionary effort is then focused on the search for correct program contents rather than the structure. Experiments on the Even-N-Parity problem show that, with the structure of λ abstractions and implicit recursion, GP is able to find a general solution which works for any value of N very efficiently.

1. Introduction

Genetic Programming (GP) [Koza, 1992] is increasing in popularity as the basis for a wide range of learning algorithms. The success of GP is generally attributed to its use of an evolutionary-based search strategy combined with a dynamic, tree-structure representation of the programs. Recently, there have been many attempts to enhance GP performance [Koza et al., 1996; Koza et al., 1997]. Among them, supporting modules in program representation has been shown to be beneficial [Koza, 1994; Rosca and Ballard,1996]. Module creation and reuse can enhance GP in its ability to scale to larger and more complex problems.

In this research, a structure which supports module creation and reuse is incorporated into the program representation. In this structure, modules are represented as λ abstractions (see Section 2.2) and module reuse is achieved through an implicit recursion (see Section 3.1). A type system (see Section 3.3) is used to preserve the structure during program evolution.

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

- Module creation is neither a random process nor a prefixed 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 approach is to generate modules dynamically, based on the recursion structures specified in advance by the users. This allows the exploration of beneficial program structures under the constraints of the users' specified conditions (see Section 3.2).

- Implicit recursion provides reuse without the possible side effect of infinite loops because there are no semantics of recursion 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 which would be used in directing genetic operation.

- The structure of λ abstraction and implicit recursion provides *structure abstraction* (see Section 6) in the program. Since the GP paradigm evolves program *structure* and *contents* simultaneously, structure abstraction can reduce the search effort for good program structure. Most evolutionary effort is then focused on the search for correct program contents rather than the structure.

We have evolved solutions to the Even-N-Parity problem [Koza,1992] using GP with the implicit recursion structure included in the program representation. The results indicate that with the structure of λ abstractions and implicit recursion, GP is able to find a general solution which works for any value of N very efficiently.

The paper is structured as follows: Section 2 provides background and related work; Section 3 presents our new

strategy; Section 4 describes the experiments; Section 5 summarizes the results; Section 6 analyzes the results and Section 7 concludes and outlines future work.

2. Background and Related Work

2.1 Recursion and Genetic Programming

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 must be satisfied for a recursive program to be effective:

• a terminating condition (base case);

• the recursive calls must be successively applied to arguments that converge towards the terminating condition.

A recursive program which fails to meet 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 infinite loops to halt. On the other hand, strict evaluation requires a function's arguments to be evaluated before the function body and can make such a program loop forever [Hudak,1989]. How program evaluation style affects GP performance has to be further studied.

To evolve recursive programs, GP faces two challenges:

1. **Handling infinite loops in a recursive program**: An evolved program may produce infinite loops as the two criteria are not always met. [Brave, 1996] adopted a finite limit on recursive calls in his tree search program. 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. Thus stopping any recursion after tree-depth number of iterations does not affect the behavior of the program. Consequently, Brave's approach is not a general solution for handling infinite loops. [Wong and Leung, 1996] employed a logic grammar to enforce the base-case structure in their "even-n-parity" programs. However, the convergence of recursive calls was not guaranteed. When infinite loops occur, they used an execution time limit to halt the program. [Clack and Yu, 1997] also imposed a recursion limit to evolve the "map" program. A "map" program applies the first argument (a function) to each element of the second argument (a list). Using the length of the input list as the recursion limit seemed to be a sensible decision.

Designing an appropriate fitness for infinite loops is a crucial task in GP, which uses fitness based selection for reproduction. A program which does not halt after the permitted number of recursion calls may still contain good partial solutions. Ideally, we would like to use them to generate new and hopefully better programs. [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 extra penalty is given to the program. [Clack and Yu, 1997], on the other hand, penalized recursion error proportionate to the length of the input list. When the recursion limit is reached, the program was terminated and returned with an empty list (no partial result was returned). The program was penalized not only for producing nothing but also for exceeding the recursion limit. This is a severe punishment. How these fitness measurements affect GP performance has to be further studied.

2. **Measuring the semantic elements of a recursive program to direct genetic operation**: The GP paradigm uses a syntactic approach to build programs; no semantic analysis is supported. A recursive program which contains a perfect base-case statement may not be selected for reproduction since semantics are not considered in the GP fitness function. [Whigham and McKay, 1995] has identified this problem and suggested the application of genetic operators in an environment where semantic analysis is supported.

2.2 λ Abstractions and Genetic Programming

In our work, we allow λ abstractions in our programs. 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 an equivalent C function.

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

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

However, λ abstractions are anonymous and cannot be invoked by name. The reuse of λ abstractions is performed by passing them as arguments to other functions. The following gives an example of the reuse of λ abstractions:

```
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
```

λ abstractions are modules that are created and reused in our programs. Similar to the structure of an Automatically Defined Function (ADF) [Koza, 1994], a λ abstraction has formal parameters and a function body. However, the determination of its structure and the overall program structure is

different from ADF. Koza adopted two approaches to defining the structure of a program with ADFs. The first one is to *statically* define it before the GP run. Once the structure is specified, every program in the population has the same structure. Genetic operations are customized to preserve the program structure. The second approach is to create various kinds of program structure *randomly* during the first generation. Program structure is then open to evolutionary determination [Koza, 1994 Ch. 21]. With 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 (Koza limited to 3,906 [Koza, 1994 pp. 529]) can be computationally expensive. This research provides a middle ground: modules are determined *dynamically* by GP but the possible selections are reduced to those allowed by the higher-order functions present in the function sets.

Higher-order functions are functions which take other functions as arguments. Each time one of these functions is used to create a program, the argument with a function type is created as a λ abstraction (more details are given in section 3). A priori knowledge about module creation and reuse is incorporated in functions and terminals to facilitate GP in determining the most effective program structure.

ADFs and λ abstractions are modules which are simultaneously evolved with the main program. Koza stated that ADFs provide two main functions in GP: 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, 1994]. These are also valid assessments about λ abstractions.

Two other approaches which also support module creation and reuse are Module Acquisition (MA) [Angeline 1994] and Adaptive Representation through Learning (ARL) [Rosca and Ballard, 1996]. In both approaches, program structures are *created* and *modified* dynamically during the 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.,1994] reported that the MA approach, where modules are created using randomly extracted program fragments, does not provide any 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].

2.3 Even-N-Parity Problem

The Even-N-Parity has been used by Koza as a difficult problem for GP to solve [Koza, 1992]. This program takes a list of N boolean inputs, returning True if an even number of inputs are True and False otherwise. This problem uses the following function and terminal sets:

• Function Set: {AND, OR, NAND, NOR},

These are standard logic functions and are logically complete.

• Terminal Set: {b_0, b_1,...,b_{N-1}},

N boolean variables.

The test cases consists of all the 2^N possible combinations of N inputs. As N is increased, the problem becomes exponential harder. Koza was able to use GP to solve the problem up to N=5. When N=6, none of his 19 runs found a 100%-correct solution [Koza, 1992].

Koza introduced ADFs as a mechanism for GP to construct modules for reuse. With ADFs, GP is able to solve this parity problem up to N=11 [Koza, 1994 Ch. 6].

[Wong and Leung, 1996] proposed a general solution which can handle any value of N using recursion. Their approach is to construct a logic grammar to enforce the base-case program structure. Type knowledge can also be incorporated in the logic grammar. When both sets of information are present in the grammar to guide the evolution, GP is able to find the solution more efficiently than using ADFs. Within 60 runs which use 8 fitness cases (Even-3-Parity), 16 runs found a solution that can work on any value of N. The generated programs, however, do not contain any subroutines, although an "xor" function can be extracted from the programs. The construction of subroutines within recursive programs is not yet implemented in their work. (They also demonstrated that their system required more computation effort when noisy fitness cases were employed [Wong and Leung, 1998]).

3. A New Strategy

This paper presents a structure of λ abstractions and implicit recursion to evolve program solutions for the Even-N-Parity problem. Implicit recursion facilitates GP to generate general solutions which work for any value of N. λ abstractions provide the module mechanism for GP to exploit the structure inherent in the Even-N-Parity problem. By combining implicit recursion with module mechanism, performance advantages are anticipated over previous work with the Even-N-Parity problem.

3.1 FOLDR: Implicit Recursion

The issues that explicit recursion has highlighted in GP foster the idea of implicit recursion. There are three popular higher-order functions which can provide recursion without explicit recursive calls [Clack, Myers and Poon, 1995]:

- MAP: applies the first argument, a monadic operator (a function which takes one argument), to each element of the second argument (a list) 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 operator (a function which takes two arguments), between each of the items in the list. With FOLDR, the empty list is substituted with the given terminating value and the resulting expression is associated to the right. With FOLDL, the given terminating value is prefixed to the expression and the resulting expression is associated to the left. For example:

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

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 Ch.4].

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

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

An important characteristic of using implicit recursion is that the programs do not produce infinite loops. The terminating condition is an empty list, which is incorporated into the higher-order functions. Moreover, there are no recursion semantics in the programs as the recursion is performed in the higher-order functions. Implicit recursion is therefore an ideal mechanism to support recursion in GP.

For the Even-N-Parity problem, we include FOLDR in our language to provide implicit recursion because FOLDR produces a single output value and so does the Even-N-Parity program. For different problem domains, other higher-order functions might be more suitable. Moreover, combining different higher-order functions is possible.

It might be possible to evolve the Even-N-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 paper, we would like to exploit the structure inherent in the

problem. The function argument is allowed to be discovered. Research is underway to compare GP performance using implicit recursion to solve the Even-N-Parity problem with and without λ abstractions as described in the following section.

3.2 λ Abstractions: Module Mechanism

In this work, the modules discovered by GP are represented as λ abstractions. Functions are allowed to take other functions as arguments. When an argument with function type is present, a λ abstraction is generated and is used as the actual value for that argument. By specifying primitive functions that require functional arguments, users can direct GP to build modules. For example, suppose the user includes FOLDR in the function set. Since the first argument of FOLDR is a function type, each time FOLDR function is selected to construct the program, a λ abstraction will be generated as the function argument. FOLDR reuses the created λ abstraction through implicit recursion.

λ abstractions 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. Argument naming in λ abstractions follows a simple rule: each argument is named with a hash symbol followed by an unique integer, for example #1, #2. This is an easy way to create unique arguments within a λ abstraction. This consistent naming style also allows crossover to be easily performed between λ abstractions with the same number of arguments. The following is an example of the FOLDR function with λ abstraction (in bold) created as its first argument:

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

3.3 Type System: Structure Preserving Engine

A type system is used to preserve the program structure of λ abstractions and implicit recursion in the programs. Initially, each primitive 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 information is specified using a type language. The abstract type syntax is given by:

$$\sigma :: \tau \qquad \textit{built-in type}$$
$$| \ \upsilon \qquad \textit{type variable}$$
$$| \ \sigma_1 \rightarrow \sigma_2 \qquad \textit{function type}$$
$$| \ [\sigma_1] \qquad \textit{list of elements all of type } \sigma_1$$
$$| \ (\sigma_1 \rightarrow \sigma_2) \qquad \textit{bracketed function type}$$

```
τ :: int | string | bool | generic_i
υ :: dummy_i | temporary_i
```

Every expression in the program may be annotated with a type:

- Constants such as `0` and identifiers such as `x` have a type pre-defined 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 have a type given as follows:

    if `exp1` has type ($\sigma_1 \rightarrow \sigma_2$)
    and `exp2` has type σ_1
    then (`exp1 exp2`) has type σ_2
    else there is a type error.

- λ abstractions have the following type:

    if `x` has type σ_1
    and `exp` has type σ_2
    then (λ `x exp`) has type $\sigma_1 \rightarrow \sigma_2$.

A function argument is denoted by the bracketed function type. For example, FOLDR is specified with the following type information:

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

The type information indicates that FOLDR can take 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 can take two arguments and returns one value. This function argument will be created as a λ abstraction of two arguments. Notice that FOLDR is a *polymorphic* function whose type signature contains *type variables*. The type system instantiates these type variables when the function is selected to construct the program. It is very important to assure that type variables are instantiated consistently so that the constructed program is type-correct.

By allowing type variables in our type language, the generality of functions in the function set are enhanced. For example, FOLDR can provide implicit recursion for many different types of arguments. More details about polymorphism and generality in GP can be found in [Yu and Clack, 1998].

The type system also performs type checking during the genetic operation of crossover and mutation. Thus, the structure of λ abstractions and implicit recursion can be preserved throughout the evolutionary process. The "point-typing" structure-preserving crossover [Koza, 1994 pp. 532] is performed in the programs: a point is first selected from the first parent program; depending on the source of the node (the main program or a λ abstraction body), a node with the same source is selected from the second parent program. If the crossover point in the first parent program is inside a λ abstraction, the crossover point in the second parent program has to be inside a λ abstraction which has the same number and type of arguments as the λ abstraction where the first point was selected. This restriction assures that the offspring will not contain any unbound global variables.

The crossover operation is also allowed to be performed on λ abstraction nodes; this is termed "λ modular crossover" as it results the swapping of a λ abstraction module in one program with a λ abstraction module in another program. This operation is similar to the modular crossover in [Kinnear, Jr., 1994]. However, with the type system, the problem of argument mismatching as mentioned in Kinnear's work does not occur in our implementation.

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. There follows an example of the λ modular crossover operation. The two parent programs are:

FOLDR (λ#1$^{\text{Int}}$(λ#2$^{\text{Int}}$(/ #1 #2))$^{\text{Int->Int}}$)$\underline{\text{Int-}}$
$\underline{\text{>Int->Int}}$ 10 [1,2,3]

FOLDR (λ#1$^{\text{Int}}$(λ#2$^{\text{Int}}$(+ #1 #2))$^{\text{Int->Int}}$)$\underline{\text{Int-}}$
$\underline{\text{>Int->Int}}$ 20 [1,2,3]

Note the two swapping λ abstractions have the same type, `Int->Int->Int`. The operation produces the following new program:

FOLDR (λ#1$^{\text{Int}}$(λ#2$^{\text{Int}}$(+ #1 #2))$^{\text{Int->Int}}$)Int-
>Int->Int 10 [1,2,3]

4. Experiments

4.1 Parameters and Primitives

As in [Wong and Leung, 1996], a population size of 500, a maximum generation of 50 and 60 runs are used in the experiment. In addition, a maximum tree depth of 4 for the main program is imposed. A λ abstraction is considered as one single node in a program parse tree as it performs one task just like the primitive functions in the function set. The maximum tree depth allowed for a λ abstraction is also 4. The crossover rate is 100%. The following are the primitives and their types used in the experiments:

Output Type: `bool`.

Argument Type: `[bool]`.

Terminal Set:

```
T = {L:: [bool]}
```

Function Set:

```
F = {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}
```

Table 1: Summary of Performance

Results	Implicit Recursion + λ Abstractions	Generic Genetic Programming	GP with ADFs
Programs	Even-N-Parity	Even-N-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	184,320,000

A FOLDR expression inside another FOLDR expression creates nested recursion. For example, the following is a program with nested recursion of depth 2:

```
FOLDR (+)(FOLDR (+) 0 [1,2,3]) [1,2,3]
= FOLDR (+) 6 [1,2,3]
= 12
```

Nested recursive programs require a considerable amount of time and space to evaluate. The depth of the nested recursion is therefore limited to 100, which we think is powerful enough to handle the Even-N-Parity problem.

4.2 Selection of Fitness Cases

The fitness cases of the Even-2-Parity and the Even-3-Parity are selected to evaluate the programs. There are $2^2 + 2^3 = 12$ fitness cases. A general Even-N-Parity program can handle any value of N, which may be either even or odd. The Even-2-Parity fitness cases help GP to learn to handle an input list with an even number of items while the Even-3-Parity fitness cases train GP to work on an input list with an odd number of items. With this set of fitness test cases, it is hoped that the generated programs can be general solutions which work for any value of N.

4.3 Handling of Run-Time Errors

A program which applies HEAD or TAIL to an empty list gives a run-time error. When such an error occurs, the system receives a default value for the expected type and continues evaluating the program so that a partial solutions can be returned for fitness evaluation. Meanwhile, a run-time error is flagged and solutions marked with this flag are penalized during the fitness evaluation.

4.4 Design of Fitness Function

The fitness function used is the same as that used by Koza [Koza 199, pp.160] except for the punishment of the run-time errors. Each program is evaluated against all of the fitness cases. When a correct result is produced for a fitness case, the program receives a 1; otherwise, it receives a 0. If run-time error has been flagged, fitness is reduced by 0.5. The fitness of a program is the sum of the fitness values for all of the fit-

ness cases. The maximum fitness value of a program in this experiment is 12.

4.5 Selection of Crossover Locations

A crossover location selection scheme which biases toward root crossover is used [Yu and Clack, 1998]. This selection scheme is designed to accommodate the premature convergence of the root nodes which was observed during the experiments and has been reported in [Gathercole and Ross, 1996]. The premature convergence of the root node can severely impair GP performance if the desired behavior of a program depends highly on the root node. The modified crossover location selection scheme provides program root nodes with more opportunities to be replaced with new nodes.

4.6 Crossover Operators

Crossover can be performed on λ abstraction nodes, fully applied or partially applied function nodes. A crossover location is first selected from the first parent program using the scheme described in Section 4.5. Its type and the depth of the node are used to select a crossover point in the second parent. The same selection scheme is used to find a node whose type "unifies" with the given type and whose depth is such that the new tree will satisfy the maximum tree depth. More details about the operation of type unification can be found in [Yu and Clack, 1998].

5. Results

60 runs were made and 57 of them found a solution. Moreover, all 57 are general solutions which work for any N number of inputs. To facilitate direct comparison, Koza's method [Koza, 1992 Ch. 8] is followed to measure the performance of this new strategy. Figure 1 shows the performance curves of the experiments.

The curve $P(M,i)$ shows the cumulative probability of success to solve the problem by generation i using a population size of 500. The curve $I(M, i, z)$ indicates the number of programs that have to be processed to produce a solution by generation i with probability z. In this work, the probability z is set to 99%. The curve of $I(M,i,z)$ reaches a minimum value

of 14,000 at generation 3 (marked on the figure). This means that if this problem is run through to generation 3, processing a total of $I(M,i,z) = I(500, 3, 0.99) = 14,000$ individuals (i.e. 500 x 4 generations x 7 runs) is sufficient to yield a solution of this problem with 99% confidence. Since 12 fitness cases are used to test the programs, the number of fitness cases to be processed is 14,000*12 = 168,000. Compared with other related works by Koza using ADFs [Koza, 1994, pp. 196] and by Wong and Leung using a Generic Genetic Programming (GGP) system [Wong and Leung, 1996], our performance excels. Table 1 summarizes the performance of these 3 different approaches in evolving Even-N-Parity program.
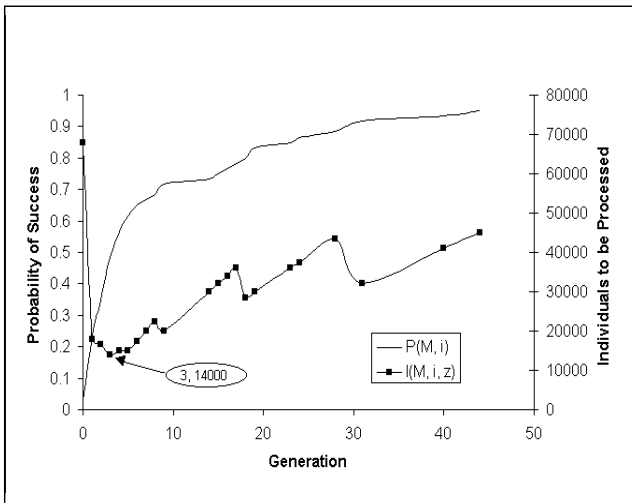

Figure 1: **Performance curves for the Even-N-Parity program with population size of 500.**

6. Analysis and Discussion

The results of the experiments indicate that by using the structure of λ abstractions and implicit recursion, GP is able to evolve very efficiently Even-N-Parity programs which work for any value of N. Koza's ADFs provide a mechanism for module creation and reuse which has helped GP learn the Even-N-Parity program up to N=11 but its performance details are not reported (Table 1 uses the performance information for Even-7-Parity on [Koza, 1994 pp.195]). The GGP system by Wong and Leung uses recursion to support program code reuse (however, no module creation mechanism is provided). The GGP system can learn the Even-N-Parity programs which work for any value of N more efficiently than the ADF approach.

The GP system presented in this paper supports both recursion and modules. With the structure of λ abstractions and implicit recursion, this system can learn the Even-N-Parity programs which work for any value of N and it can do this by processing a much smaller number of programs than the number required either by the ADF approach or by the GGP system (see Table 1). More than 50% of the 60 runs obtained a solution before generation 5 and two of them found a solution during generation 0 through random search under the constraints of the specified program structure. This

is an exceptional performance compared with any other previous work with the same problem.

Besides the benefits of recursion and modules, the authors believe there is one more factor which contributes to such an exceptional performance:

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

The ability of traditional 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 can facilitate GP in hierarchical processing by abstracting program contents. Our module mechanism of λ abstractions promotes the use of hierarchy further by supporting program *structure abstraction*. 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, this two-layer-hierarchy program structure grouping is protected from disruption by the type system; crossover can only change its contents but not its structure. In other words, GP uses the two-layer-hierarchy structure as one unit to exploit the most advantageous program structure. Figure 2 shows three program structure groupings that have identical structure. Note that they may have different contents since the three λ abstractions may be different.
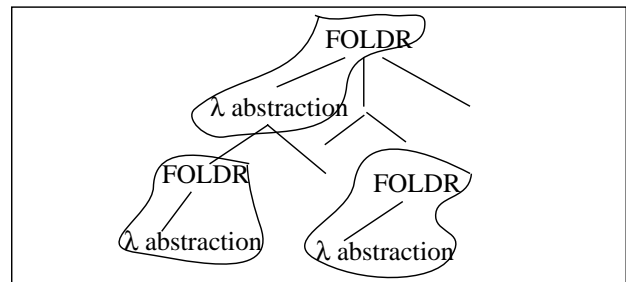

Figure 2: Program structure grouping for FOLDR.

The data collected from 10 test runs has been analyzed to see whether structure abstraction is beneficial to GP. During generation 0, various program structures are created. All those programs with more than two FOLDRs had fitness value 0, which means they cannot solve any of the 12 test cases. A few programs with no FOLDR had above average fitness. But all the programs which contain either 1 or 2 occurrences of FOLDR receive better than average fitness. All 57 correct Even-N-Parity programs generated from our experiments also contain either 1 or 2 occurrences of FOLDR (see Table 2). This suggests that the structure of the Even-N-Parity program is generally determined at generation 0. Most evolutionary processes search for the correct program contents to

fill in the program structure. Table 2 displays all 57 generated correct Even-N-Parity programs.

Table 2: Generated Correct Programs

Quantity	Even-N-Parity
22	nor (foldr *xor* (head L)(tail L)) False
9	foldr *xor* (nor (head L)(head L)) (tail L)
6	nor (foldr *xor* (head L)(tail L)) (foldr *exor* (head L)(tail L))
6	foldr *xor* (nand (head L)(head L))(tail L)
6	nand (foldr or (head L)(tail L)) (foldr *xor* (head L)(tail L))
5	nand (foldr *xor* (head L)(tail L)) True
2	foldr *xor* (foldr *xand* (head L)(tail L)) (tail (tail L))
1	nor (foldr *xor* (head L)(tail L)) (foldr *xor* (head L) (tail L))

Those functions in italics are generated functions represented as λ abstractions in the programs. They are anonymous functions in the programs but we provide them with names here for easy reference. The Truth Table for these generated λ functions is presented in Table 3. Note that those λ abstractions which compute *xor* might contain very different code. The values True and False in Table 2 indicate expressions which produce True or False under all conditions.

Table 3: Truth Table

x	y	*xor*	*exor*	*xand*
True	True	False	False	True
True	False	True	False	False
False	True	True	True	False
False	False	False	False	True

Much research work has asserted that type constraints can reduce the number of ways to construct programs and improve GP performance [Montana, 1995; Haynes, Wainwright, Sen and Schoenefeld, 1995; Haynes, Schoenefeld and Wainwright, 1996; Clack and Yu, 1997]. [Harris, 1997] used abstraction on user-defined types to enforce a hierarchy in the program parse tree, but in a specialized domain and with the hierarchy defined by the user's knowledge of that domain: this explicit program structuring method improves GP performance in his image template matching experiments. In this work, type constraints and higher-order functions are used in a general context to support structure abstraction and enhance GP performance. It is anticipated that type constraints can assist GP learning in other ways yet to be discovered.

7. Conclusion and Future Work

The structure of λ abstractions and implicit recursion provides GP with an effective mechanism to perform module creation and reuse. This work has demonstrated its power by evolving Even-N-Parity programs. We are continuing the investigation of its applicability to other problems. By incorporating this mechanism, GP is able to evolve a correct program by processing far fewer programs than the number required in any previous work. All evolved correct programs are general solutions which work well for any number of inputs. As was outlined in the introduction, three main factors have contributed to this result:

• Module creation is neither a random process nor a prefixed condition. Instead, modules are generated dynamically based on the recursion structures specified in advance by the users. This allows the exploration of beneficial program structures under the constraints of the users' specified conditions.

• Implicit recursion provides reuse without the possible side effect of infinite loops since there are no recursion semantics present in the program. This not only relieves GP from handling infinite loops in a program but also from measuring the semantic elements of recursive programs which would be used in directing genetic operation.

• The structure of λ abstractions and implicit recursion provides structure abstraction in the program. As the GP paradigm evolves program structure and contents simultaneously, abstraction of structure can reduce the search effort for good program structure. Most evolutionary effort is then focused on the search for correct program contents rather than the structure.

Further research is being directed in the following areas:

• Evaluation of the performance of a partial application crossover operator according to the type distribution in the program parse trees. Type information plays an important role in the crossover operation since they are used to assure only type-correct programs are generated. The distribution of type in the program parse tree nodes will be analyzed to see how it affects the partial application crossover operator.

• Enhancement of the recursion ability so that other forms of recursion, such as mutual recursion, can be expressed.

Acknowledgments

We like to thank Bill Langdon and Tom Westerdale for their valuable suggestions. We also thank Peter Bentley and Hava Lester for proof-reading this paper.

Bibliography

Angeline, P. J. 1994. Genetic programming and emergent intelligence. *Advances in Genetic Programming*, Kinnear, Jr., K.E.(ed.), MIT Press, Cambridge, MA, pp. 75-97.

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

Clack, C., Myers, C., and Poon, E. 1995. *Programming with Miranda*. Prentice Hall International.

Clack, C., and Yu, T. 1997. Performance enhanced genetic programming, *Proceedings of the Sixth International Conference on Evolutionary Programming*, Angeline, P.J., Reynolds, R., McDonnell, J., and Eberhart, R. (eds.), Springer-Verlag, Berlin, pp.87-100.

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

Harris, C. 1997. Strongly typed genetic programming to promote hierarchy through explicit syntactic constraints. *Late Breaking Papers at the Genetic Programming 1997 Conference*. Koza, J. R. (ed.). Stanford University Bookstore, Stanford, CA, pp. 72-80.

Haynes, T.D., Schoenefeld, D.A., and Wainwright, R.L. 1996. Type inheritance in strongly typed genetic programming. *Advances in Genetic Programming II*, Angeline, P.J., and Kinnear, Jr., K.E. (eds), MIT Press, Cambridge, MA, pp. 359-376.

Haynes, T.D., Wainwright, R., Sen, S., and Schoenefeld, D. 1995. Strongly typed genetic programming in evolving cooperation strategies. *Proceedings of the Sixth International Conference on Genetic Algorithms*, Eshelman, L. (ed.), Morgan Kaufmann Publishers, Inc. pp. 271-278.

Hudak, P. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, Vol. 21, No. 3, September, pp. 359-411.

Kinnear, Jr., K. E., 1994. Alternatives in automatic function definition: A comparison of performance. *Advances in Genetic Programming*, Kinnear, Jr., K.E. (ed.), MIT Press, Cambridge, MA, pp. 119-141.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Koza, J. R., Goldberg, D. E., Fogel, D.B., and Riolo, R.L. editors, 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. MIT Press, Cambridge, MA.

Koza, J. R., Goldberg, D. E., Fogel, D.B., and Riolo, R.L. editors, 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. MIT Press, Cambridge, MA.

Montana, D. J. 1995. Strongly typed genetic programming. *Evolutionary Computation*, Vol. 3:3, pp. 199-230.

O'Reilly, U., and Oppacher, F. 1995. The troubling aspects of a building block hypothesis for genetic programming. *Foundations of Genetic Algorithms*, Whitley, L.D., and Vose, M.D. (eds.), Morgan Kaufmann, San Francisco, CA, pp. 73-88.

Rosca, J. P., and Ballard, D. H. 1996. Discovery of subroutines in Genetic Programming. *Advances in Genetic Programming II*, Angeline, P.J. and Kinnear, Jr., K.E. (eds.), MIT Press, Cambridge, MA, pp.177-201.

Whigham, P. A., and McKay, R.I. 1995. Genetic approaches to learning recursive relations. *Progress in Evolutionary Computation,* Yao, X. (ed.), Lecture Notes in Artificial Intelligence, Vol. 956, Springer-Verlag, Heidelberg, Germany, pp. 17-27.

Wong, M.L., and Leung, K.S. 1996. Evolving recursive functions for the even-parity problem using genetic programming. *Advances in Genetic Programming II*, Angeline, P.J. and Kinnear, Jr., K.E.(eds.), MIT Press, Cambridge, MA, pp.222-240.

Wong, M.L., and Leung, K.S. 1997. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, Vol. 5:2, pp. 143-180.

Yu, T., and Clack, C. 1998. PolyGP: a polymorphic genetic programming system in Haskell. *Genetic Programming 1998: Proceedings of the Third Annual Conference Genetic Programming.* (to appear)