Chapter 7

A HIGHER-ORDER FUNCTION APPROACH TO EVOLVE RECURSIVE PROGRAMS

Tina Yu[1]
[1] *Chevron Information Technology Company*

Abstract We demonstrate a functional style recursion implementation to evolve recursive programs. This approach re-expresses a recursive program using a non-recursive application of a higher-order function. It divides a program recursion pattern into two parts: the recursion code and the application of the code. With the higher-order functions handling recursion code application, GP effort becomes focused on the generation of recursion code. We employed this method to evolve two recursive programs: a STRSTR C library function, and programs that produce the Fibonacci sequence. In both cases, the program space defined by higher-order functions are much easier for GP to search and to find a solution. We have learned about higher-order function selection and fitness assignment through this study. The next step will be to test the approach on applications with open-ended solutions, such as evolutionary design.

Keywords: recursion, Fibonacci sequence, strstr, PolyGP, type systems, higher-order functions, recursion patterns, filter, foldr, scanr, λ abstraction, functional programming languages, Haskell

1. Introduction

In August of 2000, I met Inman Harvey at the Seventh International Conference on Artificial Life in Portland, Oregon. "I just finished my Ph.D in genetic programming last year," I told Inman at the dinner table. "Great, I have a challenge for you. Can you evolve (faster than random search) the STRSTR program?"

He was referring to the C library function which scans the first appearance of one character string in another character string. If the first string does not exist in the second string, STRSTR returns an empty string. For example [1]

```
strstr (''example'',''test example'') = ''example''
strstr (''example'',''example test'') = ''example test''
strstr (''example'',''test'') = '' ''
```

This program clearly needs recursion or iteration, a subject which I spent half of my Ph.D to investigate. Although I was eager to undertake the challenge, many other projects had higher priorities at that time. It was not until early this year when I got the chance to work on this problem.

In this chapter, I present my results of using a higher-order function approach to evolve the STRSTR program. Additionally, I will show that programs generating the Fibonacci sequence can be evolved using higher-order functions.

This chapter is organized as follows: Section 2 explains higher-order functions and reviews previous work on using higher-order functions to evolve computer programs. In Section 3, the PolyGP system is described. Section 4 presents Genetic Programming (GP) (Koza, 1992) experiments to evolve STRSTR. The experiments to generate programs producing the Fibonacci sequence are given in Section 5. In Section 6, we discuss our results and review other approaches to evolve recursive programs. Finally, section 7 concludes the chapter.

2. Higher-Order Functions and Program Evolution

Higher-order functions are functions which take other functions as inputs or return functions as outputs. This ability to pass functions around as inputs and outputs can be used to express patterns of recursion. A recursion pattern has two components: operations (recursion code) and application of the operations. By extracting the operations into a function and passing it to a higher-order function, the operations can be carried out by the higher-order function.

For example, if the pattern of recursion is performing a series of operations on every element of a list, the operation can be extracted as a function *f* which is then passed as an argument to the higher-order function *map*, which applies it to every element of the list:

```
map f [] = []
map f list = cons (f (head list)) (map f (tail list))
map (+1) [1,2,3,4,5] = [2,3,4,5,6]
```

[1]In this study, STRSTR returns a character string itself instead of the pointer to the character string.

Consequently, a recursive function can be re-expressed using a non-recursive application of a higher-order function (Field and Harrison, 1988).

In a previous work, we have adapted this programming style to evolve recursive EVEN-PARITY programs (Yu, 1999). Semantically, EVEN-PARITY takes a list of `Boolean` inputs and returns `True` if an even number of inputs are `True` and `False` otherwise. Experienced circuit design engineers might be able to identify one or two familiar methods to obtain recursion. One example is applying XOR to each pair of the `Boolean` inputs and then negating the result as the final output.

When combined with the higher-order function *foldr* (with polymorphic types), the PolyGP system (described in Section 3) discovered 8 different recursion patterns; each of which operates differently by applying different `Boolean` function (XOR, NOR, NAND) to the `Boolean` input pairs (Yu,1999, Chapter 6). This work not only shows that higher-order functions provide a feasible way to evolve recursive programs, but also demonstrates the power of GP for discovering solutions that are beyond human capability.

Higher-order functions are not restricted to express recursion patterns for *list* data structures. Other data types, such as *tree* and *integer*, can have higher-order functions defined over them to carry out the recursive operations. In Section 5, we will show such an example. In that case, a higher-order function is defined over an integer value. A set of operations are performed repeatedly until the integer value reaches zero. We have applied this higher-order function to evolve programs generating the Fibonacci sequence successfully.

Higher-order functions are not expressly limited to programs with recursion patterns. Non-recursive programs can also incorporate higher-order functions to create modular programs. As an argument to a higher-order function, a function becomes a self-contained module (a λ abstraction) in a program. This module has its own identity and can only exchange materials with the same kind of modules in another program during evolution. Consequently, higher-order functions provide the ability to explore the regularity in a given problem during GP evolution. This module mechanism has been incorporated with GP to evolve financial technical trading rules based on S&P500 index (Yu et al., 2004). Those results demonstrated that modular GP rules give higher returns than the returns of non-modular GP rules.

3. The PolyGP System

PolyGP (Yu, 1999) is a GP system which is able to evolve programs containing higher-order functions. The programs have the following syntax:

$exp :: c$	constant
$\mid x$	identifier
$\mid f$	built-in function
$\mid exp1\ exp2$	application of one expression to another
$\mid \lambda x.exp$	lambda abstraction

Constants and identifiers are given in the terminal set while built-in functions are provided in the function set. Application of expressions and λ abstractions are constructed by the system.

Each program expression has an associated type. The types of constants and identifiers are specified with known types or type variables. For example, the input variable *str1* has type `[char]` and constant *True* has `Boolean` type.

```
str1::[char]
True::Bool
```

Each function in the function set is also specified with its argument and return types. For example, the function *and* takes two `Boolean` type inputs, and returns a `Boolean` type output.

```
and::Bool→Bool→Bool
```

Higher-order functions have brackets around their function arguments. For example, *filter* takes two arguments: one is a function and the other is a `[char]` type value. The function argument has type `(char→Bool)`, which indicates that it is a function which takes one input of `char` type and return a `Boolean` value. The output of *filter* is a `[char]` value.

```
filter::(char→Bool)→[char]→[char]
```

Using the specified type information, a type system selects type-matching functions and terminals to construct type-correct program trees. A program tree is grown from the top node downwards. There is a required type for the top node of the tree. The type system selects a function whose return type matches the required type. The selected function will require arguments to be created at the next (lower) level in the tree; there will be type requirements for each of those arguments. If the argument has a function type, a λ abstraction tree will be created. Otherwise, the type system will randomly select a function (or a terminal) whose return type matches the new required type to construct the argument node. This process is repeated many times until the permitted tree depth is reached.

Lambda Abstraction and Higher-order Functions

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){return (x+1)} (C function)
```

However, λ abstractions are anonymous and can not be invoked by name. The application of λ abstractions is done by passing them as arguments to a higher-order function. The following shows the above defined λ abstraction is applied by the higher-order function *twice*:

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

The procedure to create λ abstraction trees is similar to that used to create the main program tree. The only difference is that their terminal set consists not only of the terminal set used to create the main program, but also the input variables to the λ abstraction. Input variable naming in λ abstractions follows a simple rule: each input variable is uniquely named with a hash symbol followed by an unique integer, *e.g.* #1, #2. This consistent naming style allows crossover to be easily performed between λ abstraction trees with the same number and the same type of inputs and outputs. Figure 7-1 gives the program tree with higher-order function *twice* and λ abstraction described in the above example.

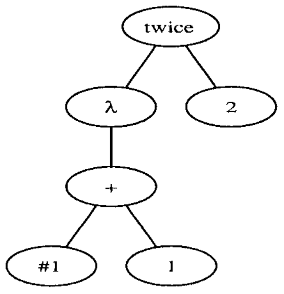

Figure 7-1. The program tree with higher-order function *twice and λ abstraction.*

4. Evolving STRSTR Programs

To evolve STRSTR program, the first step is to select higher-order functions that facilitate the evolution of recursion patterns. Functional programming languages, such as Haskell , have a rich set of higher-order functions in their libraries. From the Haskell library (Jones, 2002), we selected two higher-order functions: *filter* and *scanr.*

The function *filter* applies a predicate to a list and returns the list of those elements that satisfy the predicate.

```
filter::(a→Bool)→[a]→[a]
filter (/= 'p') ['a','p','p','l','e']=['a','l','e']
```

The function *scanr* first applies its function argument (*f*) to the last item of the list argument and the second argument (*q0*). Next , it applies *f* to the penultimate item from the end of the list argument and the result from the previous application. This operation continues until all elements in the list argument is processed. It then returns the list of all intermediate and final results.

```
scanr::(a→b→b)→b→[a]→[b]
scanr f q0 []     = [q0]
scanr f q0 (x:xs) = f x q:qs
           where qs@(q_)= scanr f q0 xs
scanr cons [][''apple'']=
           [''apple'',''pple'',''ple'',''le'',''e''].
```

In addition, the library function *isPrefixOf* is handy for implementing STRSTR. It checks if the first argument is a prefix of the second argument.

```
isPrefixOf::[a]→[a]→Bool
isPrefixOf [''app''] [''apple'']= True
```

With the 3 library functions, STRSTR function is defined as:

```
strstr str1 str2 =
      head (filter (isPrefixOf str1) (scanr cons [] str2))
```

Here, *scanr* produces all sub-strings of the input *str2.* The function *filter* checks each of the sub-strings and returns the list of the sub-strings where *str1* is the prefix. The function *head* then returns the first sub-string in the list. This STRSTR implementation works fine as long as *str1* occurs in *str2.* When this is not the case, *filter* would return an empty list, which will cause *head* return a run-time error. To avoid such an error, a function *headORnil* is defined:

```
headORnil [] = []
headORnil list = head list
```

The recursive STRSTR defined using higher-order functions is therefore:

```
strstr str1 str2=
  headORnil (filter (isPrefixOf str1) (scanr cons [] str2)
```

Figure 7-2a is the defined STRSTR program tree. As explained, the role of higher-order functions in a recursive program is to apply the recursion code to data inputs. The recursion code, however, is defined by programmers. In the case where GP is the programmer, we have to provide terminals and functions for GP to evolve the code. Figure 7-2b shows the areas of the code which are generated by GP. In particular, the triangle with a λ root is the recursion code for *filter* to apply. The recursion code for *scanr* is inside the other triangle which is also evolved by GP.

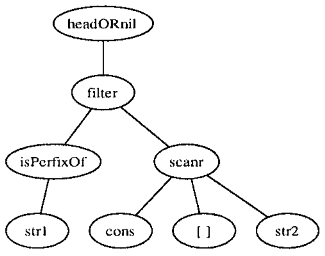

Figure 7-2a. The defined recursive STRSTR program tree.

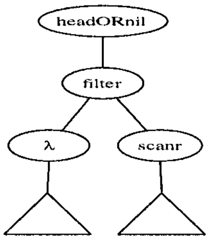

Figure 7-2b. The STRSTR program tree structure; the code inside the two triangles will be evolved by GP.

Experimental Setup

Table 7-1 gives the function set for GP to evolve STRSTR. Among them, three are higher-order functions: *filter* and *scanr* are selected from the Haskell library while *fold2lists* is defined for GP to evolve a function operating like *isPrefixOf*. *fold2lists* is an extension of *foldr*. Instead of applying recursion code on single list, *fold2lists* applies recursion code over two lists. When an empty list is encountered, *fold2lists* returns different default value, depending on which one of the two lists is empty.

```
fold2lists f default1 default2 [] list2 = default1
fold2lists f default1 default2 list1 [] = default2
fold2lists f default1 default2 (front1:rest1)(front2:rest2) =
f front1 front2 (fold2lists f default1 default2 rest1 rest2)
```

The second column of Table 7-1 specifies the type of each function. We used special types such as `input` and `output` to constrain the functions on certain tree nodes, so that the top two layers of the program trees have the same structure as that shown in Figure 7-2b.

For example, we specify the return type of STRSTR to be `[output]`. The only function which returns this type is *headORnil*, which will always be selected as the program tree root. The single argument of *headORnil* has type `[[output]]` and the only function that returns this type is *filter*, which will always be selected as the argument node below *headORnil*. Although there are other ways to constrain tree structures, typing is convenient since the PolyGP system has a powerful type system to perform type checking for the program trees.

Table 7-1. Function Set

function	*type*
headORnil	$[\![output]\!] \to [output]$
filter	$([char] \to Bool) \to [\![char]\!] \to [\![output]\!]$
scanr	$(char \to [char] \to [char]) \to [output] \to [output] \to [\![char]\!]$
cons	$char \to [char] \to [char]$
fold2lists	$(char \to char \to Bool \to Bool) \to Bool \to Bool \to$ $[input] \to [char] \to Bool$
and	$Bool \to Bool \to Bool$

Table 7-2 gives the terminal set. The variable *str1* will always be selected as the fourth argument to *fold2lists*. Similarly, *str2* and [] will always be selected as either the second or the third argument to *scanr*. At a first look, it seems that the program trees are so constrained that the generation of STRSTR programs would be very easy. However, after careful examination, you will find that all that have been specified are the skeleton of STRSTR program: the higher-order functions and the inputs list which the recursion code will apply. The core of a recursive program, the recursion code must be discovered by GP.

Table 7-2. Terminal Set

terminal	*type*	*terminal*	*type*
str1	$[input]$	str2	$[output]$
true	$Bool$	false	$Bool$
[]	$[output]$		

The GP parameters are given in Table 7-3 while the three test cases used to evaluate GP programs are listed in Table 7-4. For this problem, three test cases are sufficient as they include all possible scenarios: the first string appears at the beginning of the second string; the first string appears in the middle of the second string and the first string does not exist in the second string.

Table 7-3. GP Parameters

population size	500	max generation	100
maximum tree depth	5	crossover rate	50%
mutation rate	40%	copy rate	10%
selection method	tournament of size 2	number of runs	100

The fitness function is defined as follows:

$$f = \sum_{i=1}^{3} \text{diff}(R_i, Ei) + \begin{cases} \text{length}(R_i) - \text{length}(E_i), & \text{length}(R_i) \geq \text{length}(E_i) \\ 5 * (\text{length}(E_i) - \text{length}(R_i)), & \text{otherwise} \end{cases}$$

R is the output returned by a GP program and E is the expected output; *diff* computes the number of different characters between the two outputs. If the two outputs have different length, *diff* stops computing when the shorter output ends. The length difference then becomes a penalty in the fitness calculation. Note that a program which returns an output shorter than the expected length is given a penalty five times higher than a program which returns an output longer than the expected length. This is based on my observation that the most frequently produced shorter output is an empty list. Such programs obtain a reasonably good fitness by satisfying the easiest test case: case number 3. However, they are very poor in handling the other two test cases. Once the population converges toward that kind of program, some important terminal nodes (*e.g.* `False`) become distinct and crossover or mutation are not able to correct them. To avoid such premature convergence, programs which generate shorter outputs than the expected outputs are penalized severely. A program which satisfies all 3 test cases successfully has fitness 0.

Table 7-4. Three Test Cases

case no.	*str1*	*str2*	*expected output*
1	"sample"	"sample test"	"sample test"
2	"sample"	"test sample"	"sample"
3	"sample"	"test"	""

Results

The program space turns out to be very easy for GP to search: all 100 runs find a solution before generation 31. The "computation effort" required to find a solution is given in Figure 7-3a. The minimum number of programs GP has to process in order to find a solution is 20,000.

The computational effort was calculated using the method described in (Koza, 1992). First, the cumulative probability of success by generation *i* using a population size *M* (*P(M,i)*) is computes. This is the total number of of runs that succeeded on or before the *i*th generation, divided by the total number runs conducted. Next, the number of individuals that must be processed to produce a solution by generation *i* with probability greater than *z* (by convention, *z*=99%) is computed using the following equation:

$$I(M,i,z) = M * (i+1) * \left\lceil \frac{log(1-z)}{log(1-P(M,i))} \right\rceil$$

The hardware CPU time used on a Pentium 4 machine to complete the 100 runs is 40 minutes, which is longer than our other GP experimental runs. This is because each program has 3 recursions. In particular, *fold2lists* is inside of *filter*. This nested recursion takes machines a long time to evaluate.

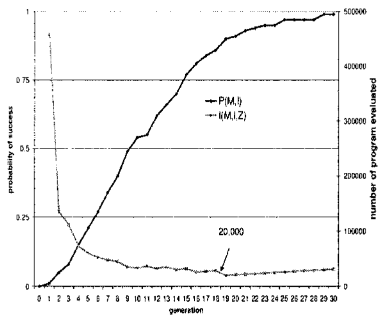

Figure 7-3a. Computation effort required to generate a STRSTR programs.

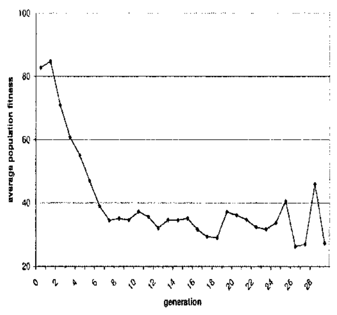

Figure 7-3b. The average population fitness during program evolution.

After editing, all evolved STRSTR programs look the same (see Figure 7-4). The *scanr* function generates a list of sub-strings from input *str2*. The *filter* function then removes the sub-strings whose initial characters do not match the input *str1*. The *headORnil* function then returns the first item in the resulting list. If the resulting list is empty, *headORnil* returns an empty list.

To investigate if a random search can do as well a job as GP does for this problem, we made 100 random search runs, each of which generated 20,000 programs randomly. None of them found a solution. We also evaluated the average population fitness of the 100 GP runs (see Figure 7-3b). They show the average population fitness improves as the evolution progress (the data after generation 18 are insignificant as they are based on a very small number of

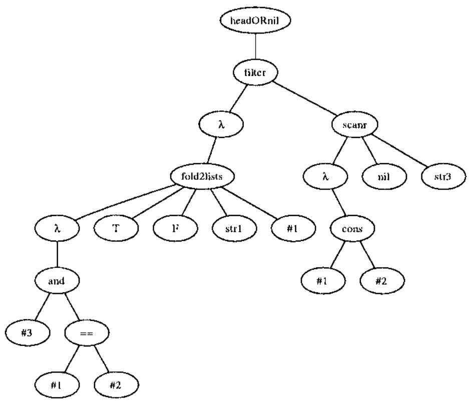

Figure 7-4. The shortest STRSTR program generated by GP.

runs); the improvement is particularly evident during the first 8 generations. In other words, GP search leads the population converge toward fitter solutions and finds an optimal at the end. All the evidences indicate that fitness and selection have positive impact on the search. GP is a better search algorithm than random search to find STRSTR programs in this program search space.

5. Fibonacci Sequence

Fibonacci sequence is defined as the following:

$$f(n) = \begin{cases} 1 & \text{, if n=0 or n=1} \\ n_{i-1} + n_{i-2} & \text{, otherwise} \end{cases}$$

To generate the first *n* values of the sequence, a program has to compute the two previous sequence values recursively for *n* time. The recursion, recursion pattern in this case is therefore applying some operations over an integer value. A higher-order function *foldn* is designed for this pattern of recursion:

```
foldn::([int]→[int])→int→input→[output]→[output]
foldn f default 0 list = cons default list
foldn f default 1 list = cons default (foldn f default 0 list)
foldn f default n list = f (foldn f default n-1 list)
```

Here, *list* is an accumulator to store the sequence values generated so far. The recursion code (*f*) is applied on the accumulator to compute the next sequence value. As mentioned previously, the role of higher-order functions in a recursive program is to apply the recursion code, which are generated by GP. In Figure 7-5, the left triangle is the recursion code area. The right triangle is the default value. Both of them are generated by GP . Similar to the previous experiment, we use special types `input` and `output` to constrain the functions and terminals on certain tree nodes so that the evolved program trees have the specified structure.

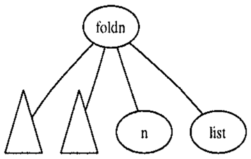

Figure 7-5. The program tree structure; the area inside the two triangles are generated by GP.

Experimental Setup

The function and terminal sets are given in Table 7-5 and Table 7-6 respectively. We specify `[output]` as the program return type, hence enforce *foldn*, the only function that returns this type, to be the program tree root . This function has four arguments; the third one will always be the variable *n* and the fourth one will always be the variable *list*. Initially, accumulator *list* is an empty list. It grows as the sequence values are generated. *RandomInt* is a random number generator which returns a random integer value in the range of 0 and 3. The GP parameters are listed in Table 7-3.

Table 7-5. Function Set

function	*type*
foldn	$([int] \rightarrow [int]) \rightarrow int \rightarrow input \rightarrow [output] \rightarrow [output]$
plus	$int \rightarrow int \rightarrow int$
minus	$int \rightarrow int \rightarrow int$
head	$[int] \rightarrow int$
tail	$[int] \rightarrow int$
cons	$int \rightarrow [int] \rightarrow [int]$

Table 7-6. Terminal Set

terminal	*type*	*terminal*	*type*	*terminal*	*type*
n	*input*	list	$[output]$	randomInt	*int*

Each evolved GP program is tested on *n* value of 8. The expected return list is therefore `[34,21,13,8,5,3,2,1,1]`. The fitness function is basically the same as the one in the previous experiments. One exception is that there is a run-time error penalty of 10 for programs applying *head* or *tail* to an empty list. The fitness function is therefore:

$$f = \text{diff(R,E)+10*rtError+} \begin{cases} \text{length(R)-length(E)} & \text{,length(R)} \geq \text{length(E)} \\ \text{5*(length(E)-length(R))} & \text{,otherwise} \end{cases}$$

where *R* is the return list while *E* is the expected list. The run-time error flag *rtError* is 1 if a run time error is encountered during program fitness evaluation. Otherwise, it is 0.

Results

This program space is slightly harder than the STRSTR program space for GP. Among 100 runs, 97 found a solution; all of them are general solutions work for any value of *n*. The computation effort required to find a solution is given in Figure 7-6a. The minimum number of programs evaluated by GP to find a solution is 33,000. The hardware CPU time on a Pentium 4 machine to complete the 100 runs is 7 minutes. After editing, all programs become the same as that shown in Figure 7-7.

The left most branch in the program tree is the recursion code that the higher-order function *foldn* applies to a list. It is a function, specified by λ, with one argument (#1). The argument is an accumulator containing the Fibonacci sequence values generated so far. The function adds the first two elements of the list together and then concatenates the result to the accumulator. This operation is repeated until the input *n* becomes 0, when the default value 1 is returned. It is a general solution that produces the first *n* values of the Fibonacci sequence.

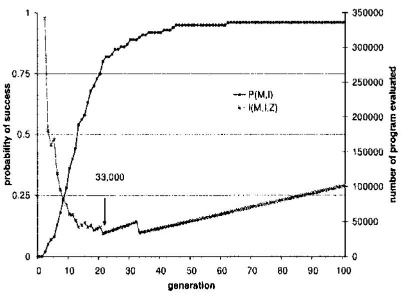

Figure 7-6a. Computation effort required to evolve a program generating Fibonacci sequence.

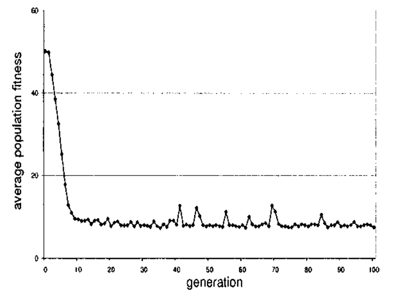

Figure 7-6b. The average population fitness during program evolution.

We also made 100 random search runs, each generates 33,000 programs randomly. Similar to the results of the STRSTR experiments, none of them found a program capable of producing the Fibonacci sequence. The average

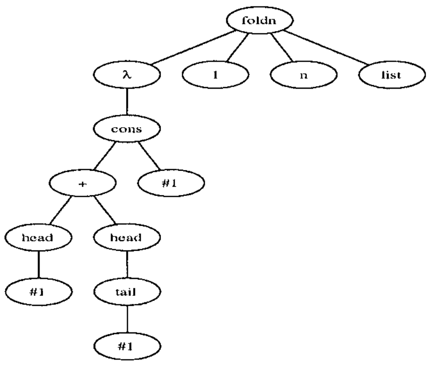

Figure 7-7. The shortest program generated by GP.

population fitness of the 100 GP runs in Figure 7-6b indicates that GP search guided by fitness and selection has led the population converging toward fitter programs and found an optimal at the end. This supports the case that GP is a better search algorithm for this program space.

6. Discussion

Recursion is a powerful programming technique that not only reduces program size through reuse but also improves program scalability. Evolving recursive programs, however, has not been easy due to issues such as non-termination and fitness assignment (Yu, 1999, Chapter 3). By re-expressing recursive programs using non-recursive application of a higher-order function, the produced recursive programs always terminate. It is therefore a promising approach to evolve recursive programs.

In a previous work, we have shown that when using higher-order function *foldr* (with monomorphic types) to define recursive EVEN-PARITY, the problem difficulty is greatly reduced. In fact, random search is sufficient to find a solution in this program space (Yu, 1999, Chapter 7). In this chapter, we study two other recursive programs using a similar approach. Both program spaces defined by higher-order functions are not difficult for GP to find a solution. Random search, however, could not find a solution. Further analysis of population average fitness confirms that GP search is indeed superior than random search in these two problem spaces.

One important characteristic of this approach is that GP effort is mostly on evolving the recursion code (λ abstractions). The application of the code is handled by higher-order functions. It is important to note that GP has no knowledge about how the recursion code is applied. The relationship between the code and its application is learned through the iterative process of programs evaluation, correction and selection. Our experimental results indicate that GP is able to acquire such knowledge to evolve recursion code that work with the higher-order function to produce correct outputs.

Although incorporating designed/selected higher-order functions is an effective way to evolve recursive programs, it has its shortcoming: domain knowledge are not always available to design/select the appropriate higher-order functions. A more general approach would be to let GP evolve the higher-order functions suitable for a given problem. In this way, problems with poorly-defined scope can also benefit this technique.

Koza and his colleagues proposed Automatic Defined Recursion (ADR) as a way for GP to evolve recursive programs (Koza et al., 1999). An ADR tree has 4 branches: condition, body, update and ground. Since an ADR can call itself inside its body and the update branch may be ill-defined during program evolution, an ADR may never terminate. It is therefore necessary to set an ADR execution limit when evolving recursive programs. They have employed ADR with architecture-alternating operations to successfully evolve programs generating the Fibonacci sequence. However, the solution is not general and does not work for input n beyond 12.

Through incremental program transformation, Olsson showed that recursive programs can be developed by his ADATE system (Olsson, 1995). Instead of relying on fitness-based selection and genetic operation, his system applies four transformation operations to induce recursive programs. He gave some example programs, such as a sorting algorithm, which were successfully generated using this approach.

7. Conclusions

Functional implementation of recursive programs is not well understood nor utilized in the GP community. The implementation does not make explicit recursive calls. Instead, recursion is carried out by non-recursive application of a higher-order function. This chapter explains this style of recursion implementation and demonstrates one way to incorporate it in a GP system to evolve recursive programs.

In this GP system, higher-order functions are included in the function set. Recursion occurs when a higher-order function appears in a program tree node. We applied this GP system to evolve two recursive programs. In the first case, a challenge by Inman Harvey, multiple recursions are involved. We selected two Haskell library functions and designed one higher-order function for these recursion patterns. In the second case, a higher-order function operating over an integer value is designed. In both cases, the GP system is able to evolve the recursive programs successfully by evaluating a small number of programs. Random search, however, is not able to find a solution.

These results clearly endorse GP ability to evolve recursive programs that random search can not. Yet, the success is linked to the problem-specific higher-order functions. When domain knowledge is available, like the two problems we studied, identify such higher-order functions is not hard. However, when this is not the case, it becomes unclear if GP is able to compose the recursive code to work with a general purpose higher-order function. An alternative approach is to have GP evolve the problem-specific higher-order functions. In this way, the GP system is more general and can be applied to problems that do not have a well-defined scope. This is the area of our future research.

Are we ready to tackle real-world problems using this approach? Maybe. We have learned quite a deal about higher-order functions selection and fitness assignment. However, both problems we studied have known solutions, which help the selection of higher-order functions. Most real-world problems are open-ended in the sense that there is no known optimum. However, this does not preclude the possibility of applying the method. In particular, in the area of evolutionary design where creativity is essential to problem solving, an imperfect higher-order function might still be able to deliver good solutions.

8. Acknowledgements

I thank Lee Spector and Inman Harvey for their comments and suggestions.

References

Field, Anthony J. and Harrison, Peter G. (1988). *Functional Programming.* Addison-Wesley Publishing Company.

Jones, Simon Peyton (2002). Haskell 98 language and libraries, the revised report. Technical report, Haskell Org.

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

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman.

Olsson, Roland (1995). Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81.

Yu, Gwoing Tina (1999). *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming.* PhD thesis, University College, London, Gower Street, London, WC1E 6BT.

Yu, Tina, Chen, Shu-Heng, and Kuo, Tzu-Wen (2004). Discovering financial technical trading rules using genetic programming with lambda abstraction. In U-M O'Reilly, T. Yu, R. Riolo and Worzel, B., editors, *Genetic Programming Theory and Practice II*, pages 11–30.