A Functional Style and Fitness Evaluation Scheme for Inducting High Level Programs

Paul Walsh

Computer Science Department
University College Cork
Ireland
Email: paul@cs.ucc.ie
Phone: +353-21-365305

ABSTRACT

This paper presents a new technique, Functional Genetic Programming (FGP), for the induction of high level programs. This technique is based on the pure functional language FP, which allows the specification of functional programs that do not reference named objects and advocates a programming style that utilises higher-order functions. A number of fitness evaluation schemes are also investigated and results show that a step-wise fitness scheme out performs the evaluation of a single raw fitness measure. The results of a number of GP experiments are presented and results show that this technique can generate high level programs that are highly expressive and inherently parallel.

1 Introduction

It has been highlighted by Olsson, that when inferring a large expression E, the schema theorem requires that E is primarily composed of expressions $E_1, E_2, E_3, ..., E_n$ such that the fitness advantage of each E_i can be measured independently of each E_j with $j \neq i$, where each E_i can be viewed as a schema [Olsson 94]. Unfortunately, practically all recursive and iterative LISP programs consist of coupled E_i's, where an expression E_i cannot be evaluated until a base case expression E_j is evaluated. That is, for a recursive program, each successive call must converge towards a terminating condition. If each recursive call does not converge towards the terminating condition, then the base case will never be evaluated. Similarly, it is difficult for GP to produce iterative programs, and it has been claimed that the current form of GP is unlikely to ever become an effective tool for general purpose programming [Olsson 94]. Moreover, many inferred programs generated in GP do not use common programming language constructs such as iteration and recursion.

However, it has been shown that recursive and iterative programs can be generated by the use of higher-order functions in GP [Walsh 98]. Similar results have been reported using the pure functional language Haskell [Yu 98]. It is therefore the aim of this paper to extend the power of GP, by utilising the expressive power of higher order functions and the pure functional language FP to induct programs to solve a number of common programming problems that involve iteration and selection. To this end an FGP system is described and the practical scope of the FGP system is extended by the inclusion of an FP to C translator, which translates inducted FP code into C. This allows FP to be used as an intermediate form for the generation of general purpose C programs.

The remainder of this paper is organised in the following fashion. Section 2 of this paper highlights the advantages of a pure functional style for program induction and describes the architecture of a GP system that can infer general purpose programs in both FP and C. Section 3 describes a step-wise fitness function that allows a more efficient search of the FP program space and compares this with a standard GP fitness function. Section 4 describes a number of GP experiments and describes the performance of the system on a number of general purpose programming problems. Section 5 discusses the translation of FP programs to C. Section 6 presents a number of conclusions and discusses future directions of research.

2 Program Induction with FP

FP is a pure functional language that uses a highly expressive functional notation and the characteristic style of programming it encourages makes it a basis for a viable functional language for practical use. The syntax and semantics of FP are discussed in more detail in [Backus 78]. FP was chosen for program induction for the following benefits:

- Functional languages provide a powerful abstraction mechanism for describing a pattern of behaviour. This is achieved by the use of *higher order functions*, which are functions that can take functions as arguments or return functions as a result. Higher order functions in functional programming languages enable very regular and powerful algorithms to be constructed [Cunningham 97].

- Olsson has noted, that functional languages are particularly useful for program induction techniques as functional programs are often much smaller than the corresponding imperative programs. This is particularly important when using search strategies that search the subspace of all programs [Olsson 94]. Similarly, as FP programs do not require any explicit identifiers, the number of objects that must be expressed in the solution is reduced, which in turn reduces the problem search space.
- A reference to an explicitly defined argument complicates the formal manipulation of functions, since formal manipulation of user defined functions are more easily expressed if object references in the original program are abstracted out [Field 88]. Algebraic laws for FP have been constructed which can be applied to transform one FP program into another [Backus 78]. Such laws can be used to transform one program into an equivalent program that can be executed more efficiently.
- Pure functional programming languages encourage massively parallel execution. However many LISP implementations, including those used in GP, employ state changing constructs. This destroys the referential transparency of inducted programs and data dependency between statements in programs has proved limiting in the induction of parallel programs [Walsh 95].
- As FP is a loosely typed language, the stochastic search technique employed need not be constrained to the application of functions to a specific type. This simplifies the search procedure, whereas evolutionary search techniques that use typed languages, as in [Yu 98], must employ complex type preserving mechanisms.

3 Functional Genetic Programming

FGP is a technique that carries out a directed stochastic search on the set of all possible FP programs. This directed search is implemented by a typical GP algorithm implemented over the function set of FP primitive functions. Input/Output test cases are used to define the functionality of the program that is to be inferred. Programs inferred by the system are evaluated using an FP evaluation module. While functional languages are often criticised for being impractical, a key feature of the FGP system is the inclusion of an FP to C translator. This allows inducted subroutines to be linked with standard C programs. The C code generated is functional in style, which improves the modularity of C programs. The structure of the FGP system is shown in Figure 1.

3.1 Structure of Individuals

Individuals in FGP experiments in this paper consist only of primitive functions and the functional forms **constant**, **composition**, **conditional** and the **higher order** functions **apply-to-all** and **insert**, which are sufficient for generating many common general purpose programs. A unique feature of FGP is that there are no variables represented as there are no explicitly named objects in FP. Hence individuals are represented, for convenience, using trees that consist only of primitives and combining forms. The composition operator $\circ$ is inserted between functions and combining forms by parsing individual trees using a pre-order traversal. When the combining forms $|$ and α are encountered the compositional combining form $\circ$ is omitted. This representational scheme ensures that programs represented in this parse tree form remain syntactically correct, despite the disruptive effect of the crossover and mutation search operators. A sample individual is shown in Figure 2. The FP program corresponding to this is:

$$\iota \circ \mid + \circ \iota$$

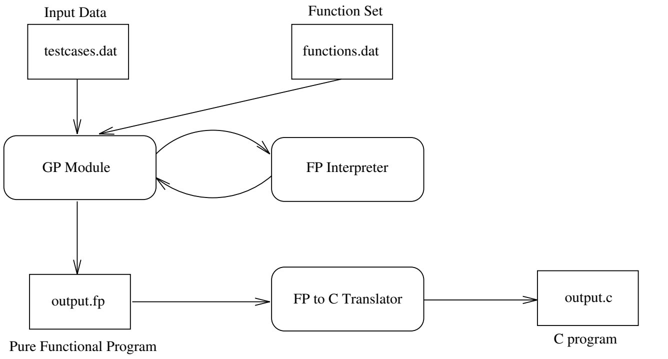

Figure 1 Architecture of the Functional Genetic Programming System.

However, a major limitation that was encountered with FP is the inability of FP programs to maintain state. This was particularly limiting for applications that require multiple instances of input arguments. This limitation was successfully addressed by extending the power of FP, without compromising the referential transparency of the system. A new set of functions are introduced which return the initial input arguments. These are constant functions and do not update any variables and are automatically defined for each application. For example, a program that transforms two input arguments will have two functions, $x1$ and $x2$, automatically defined by the system. These functions return the original unmodified values of the first and second input arguments respectively. Note that these argument "references" to input argu-

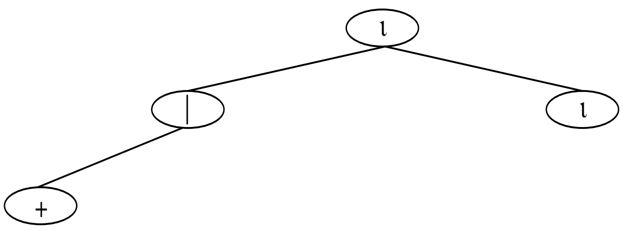
Figure 2 Sample FP program parse tree

Primitive	Function
+,-,*,/	Standard arithmetic operators, e.g. $+ :< 2, 3 >= 5$
$\leq, \geq, \neq, =$	Standard comparison operators $\leq :< 2, 3 >= T$
1,2,3,..	Sequence selector functions e.g. 2:$< x_1, x_2, .., x_n >= x_2$
xn	Input parameter selector function, where n is the n^{th} input parameter e.g. $x2 :< 3, 4 >= 4$
id	The identity function e.g. id:3=3
ι	Generation of a list of consecutive integers, e.g. ι:3=$< 1, 2, 3 >$
trans	Transposition of a matrix, e.g. trans:$<< 1, 2 >< 3, 4 >< 5, 6 >>$ $=<< 1, 3, 5 >< 2, 4, 6 >>$
%	Constant e.g. %4=4

Table 1 Primitive functions used in FGP

ments are functions, and as such they maintain the referential transparency of inducted programs. The notion of constant functions is also widely used in the object oriented programming paradigm. This innovation has proved useful for the efficient induction of functional programs, see Section 4 below.

3.2 Modifying Individuals

Crossover in FP parse trees is achieved by swapping sub-trees as in GP, but may result in individuals that contain combining forms that are undefined, $\perp$. However, any function that has $\perp$ as an input argument, automatically returns $\perp$, thus achieving closure. This notion of "undefined" has proved useful in the evaluation of individual programs created by the system. Programs that return $\perp$ as a result are not incorrect but are simply undefined over the current input arguments. Such programs may contain useful schema which may be employed in the generation of a correct solution.

3.3 Initial Population Structure

The initial population of individuals is created by randomly generating trees from the function set, F. A ramped half and half method is used so as to introduce a variety of different tree shapes into the initial population. Trees are generated by recursively inserting nodes from the functions set F for points less than the specified depth and by inserting null terminals for points that exceed the maximum depth.

3.4 Fitness

One of the most common return results from the vast majority of individuals created in FGP, in the population creation phase, is the undefined result $\perp$. The probability of this result being returned is high as the return values of a randomly selected sequence of functions are unlikely to be defined as input arguments to each other. Consider, for example, the evaluation of the following FP program:

$$
\begin{aligned}
&\iota \circ * \circ + :< 1, 2 > \\
&= \iota \circ * : 3 \\
&= \iota : \perp \\
&\perp
\end{aligned}
$$

The intermediate results in the evaluation of this example are the integer 3, $\perp$ and the final output $\perp$. While the final result of the FP program may not provide the correct solution, previous applications of combining forms may have produced a result that is close to the desired result. In the above example the application of the function + to the input argument $< 1, 2 >$ produced the result 3, which shows that the function + is not "undefined" over the input arguments. Consequently, to accommodate a more precise evaluation of individuals, three separate approaches were investigated, denoted as single, step-wise and exhaustive evaluation respectively.

In a single evaluation scheme the full expression is evaluated in its entirety and the final value is returned. In the above example, the single evaluation results in the return value $\perp$, and in this case the raw fitness is ∞, or an arbitrarily large value. In a step-wise evaluation scheme the result from the successive evaluation of each function of the expression is examined. The best result from each of these function applications is then taken as the result. In the above example this results in the subexpressions $+, * \circ +$ and $\iota \circ * \circ +$, which result in the return values 3, $\perp$ and $\perp$ respectively. In these cases the raw fitness measure is the least difference between the defined values and the target output value. In the exhaustive evaluation scheme, the complete combination of all subexpressions with in the original FP expression are evaluated. In the above example, this results in the expressions $+, * \circ +, \iota \circ * \circ +, \iota \circ *$ and ι. Here the return types are 3, $\perp, \perp, < 1, 2 >$ and $\perp$ respectively. Again the defined return values, in this case 3 and $< 1, 2 >$ are compared with the desired output and the best raw fitness measure is taken.

The motivation behind the step-wise and exhaustive evaluation schemes is the Building Block Hypothesis (BBH) [Holland 68]. Both these schemes attempt to locate building blocks which may lead the search mechanism to sections of the fitness landscape which are defined ($\neq \perp$), over the set of input arguments. Programs which contain useful schema will receive a higher score and will be propagated with exponentially increasing numbers in subsequent trials (generations). It should be noted that a high degree of diversity is maintained within these programs and that the search is not focused exclusively on these building blocks, as most programs will contain other undefined sub-expressions, which may be useful in future trials. There is some empirical evidence to support this hypothesis as a number of experiments were performed to investigate each of these evaluation schemes.

Each of these fitness evaluation schemes were tested on the dot product problem, see Section 4, and results were averaged over 10 runs, see Figure 3. It was found that all evaluations schemes found the correct solution, with the exhaustive

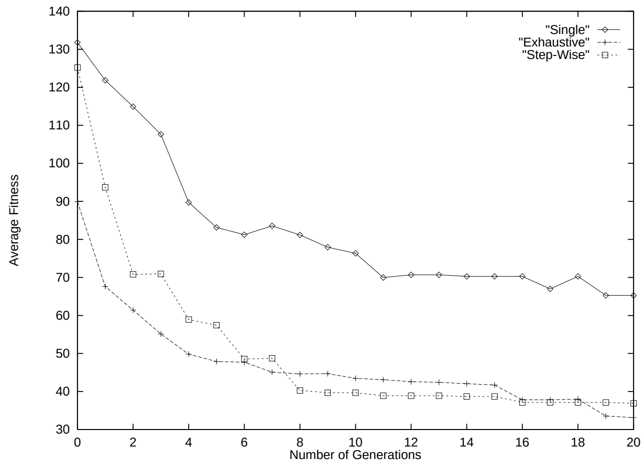

Figure 3 Time series plot of best fitness for the Single, Step-wise and Exhaustive evaluation schemes, averaged over 10 runs.

Objective :	Produce a functional program that minimises raw fitness
Terminal Set :	NULL
Function Set :	*, /, +, -, |, α, ι, $trans$, id, 1, 2, xn, $(>\to;)$, $(=\to;)$, $(<\to;)$
Fitness cases :	10 randomly generated test cases
Raw Fitness :	The number of fitness cases for which the functional program outputs the correct values
Standardized fitness	Same as raw fitness
Hits :	Same as raw fitness
Wrapper :	None
Parameters :	M = 500-2000, G=20
Success Predicate :	error=0

Table 2 A Koza-style tableau summarising the control parameters for the FGP experiements described in this paper.

evaluation method performing the best. The single evaluation scheme also found the correct solution as it was found that initial populations contained a number of short programs that did not result in $\perp$. For example, the FP programs $trans, \alpha \mid +$ and $\mid \ \alpha *$ were all generated in initial populations and are all defined over a pair of input vectors. Hence there was an increase in the number of these expressions in subsequent generations. This has the effect of focusing the search on the region of the search space that contains these functions. However, it was found that the search converges more quickly to a global optimum when the full evaluation of all building blocks is employed. This is at an increased overhead as there are Σ^{n-1} extra evaluations for an FP program containing n functions. However, step-wise evaluation requires the same amount of computation but performs significantly better than the single evaluation scheme. Consequently, the step-wise evaluation scheme is the most commonly employed fitness function in the following FGP experiments.

4 Results

FGP was applied to the task of generating functional programs for a number of common programming problems. These problems were chosen as they require some form of iteration or recursion. It should be noted that the complete function set described in Table 3 was used in *all* experiments. However, the search space may be reduced if some heuristic is used for the selection of functions suitable for a given problem. It is also possible to implement all of these algorithms in parallel as the insertion operation $\mid$ can be replaced by a synchronous parallel version, which would evaluate a problem of size n in $O(log_2 n)$ time and the apply-to-all operator α can be implemented in $O(C)$ time, where C is the constant time required to evaluate a single applied operator [Braunl 93].

4.1 Factorial

A common numerical function, that requires some form of iteration is the evaluation of the factorial of a number. The solution found by FGP is unique in that it uses the very nature of the problem to find a solution:

$$\mid * \circ \iota$$

Applying this to the input object 6, illustrates the expressive power of solutions generated by FGP, and the ability of FGP to find novel solutions:

$$\mid * \circ \iota : 6$$
$$= \mid * :< 1, 2, 3, 4, 5, 6 >$$
$$=< 1 * 2 * 3 * 4 * 5 * 6 >$$
$$= 720$$

4.2 Dot Product

Many high performance computing applications consist of a large number of matrix and vector operations. The following solution was generated by FGP:

$$\mid + \circ \alpha * \circ trans$$

Applying this to the input object $<< 1, 2, 3 >< 4, 5, 6 >>$:

$$\mid + \circ \alpha * \circ trans :<< 1, 2, 3 >< 4, 5, 6 >>$$
$$\mid + \circ \alpha * :<< 1, 4 >< 2, 5 >< 3, 6 >>$$
$$\mid + :< * :< 1, 4 > * :< 2, 5 > * :< 3, 6 >>$$
$$\mid + :< 4, 10, 18 >$$
$$< 4 + 10 + 18 >$$

This solution works for any pair of vectors, regardless of size, and the apply-to-all compositional form, α, can be executed in parallel using the synchronous SIMD model of parallel computation.

4.3 Exponentiation

Finding the power of a number raised to some exponent is a common numerical problem which requires some iterative or recursive mechanism. In order to increase the power of the FGP system the use of automatically named arguments, as described in Section 3.1 above, is introduced. A typical FP solution, without the use of automatically defined argument functions, is as follows:

$$| * \circ \alpha * dist \circ [hd, \alpha\%1 \circ \iota \circ tail]$$

However, extending FP to allow the use of automatically defined argument functions, the FGP system generated the following solution:

$$| * \circ \alpha x1 \circ \iota \circ x2$$

where $x1$ is a function that returns the value of the first argument and $x2$ is a function that returns the value of the second argument. Applying this program to the input arguments $< 2, 4 >$ illustrates the solution:

$$| * \circ \alpha x1 \circ \iota \circ x2 :< 2, 4 >$$

$$| * \circ \alpha x1 \circ \iota : 4$$

$$| * \circ \alpha x1 :< 1, 2, 3, 4 >$$

$$|* :< 2, 2, 2, 2 >$$

$$16$$

The use of automatically defined argument functions in FP is a unique feature and as the above example illustrates, it provides a novel way for FP programs to maintain input state, while preserving referential transparency, for a given problem.

4.4 Condition

One of the most fundamental constructs in any language is the conditional statement. The conditional statement $(\rightarrow;)$ is implemented in FGP. While current implementation limitations have not allowed the definition of a general conditional construct, a number of commonly used Boolean expression have been incorporated into conditional constructs. The following conditional constructs have been implemented for the experiments described in this paper:

$$(=\rightarrow;)(\neq\rightarrow;)(>\rightarrow;)$$

As the system allows any level of nesting of conditional statements, all possible comparison operations can be implemented using the above constructs. The system was tested with the basic problem of finding the greater of two numbers. The following solution was generated by the system:

$$(>\rightarrow 1; 2)$$

Applying this to the input argument $< 5, 9 >$ produces the following result:

$$(>\rightarrow 1; 2) :< 5, 9 >$$

$$2 :< 5, 9 >$$

$$9$$

4.5 Searching

Searching operations are common to many general purpose applications. The FGP system was tested with a simple searching problem of finding the greatest element in an arbitrarily long list of integers. The problem can easily be reformulated in terms of searching for the smallest element or searching for a specific element. For very large population sizes the system found a general solution to the problem. Convergence to a solution occurred more rapidly when an incremental or a large training set was used. The following solution was generated by FGP:

$$| (>\rightarrow 1; 2)$$

Applying this to the input argument list $< 3, 55, 23, 40, 2, 34 >$ yields the following result:

$$(>\rightarrow 3; (>\rightarrow 55; (>\rightarrow 23; (>\rightarrow 40; (>\rightarrow 2; 34)))))$$

$$= 55$$

5 FP to C

The FGP system can be applied to a wider set of applications by translating the FP solutions into the more widely used programming language C. Thus, FGP can hide the more cryptic features of the FP style to programmers who are more familiar with the imperative style of C. The following code segment shows the solution to the dot product of two vectors, as described in Section 4.2, translated to C.

```
#include <stdio.h>

fp_data dot (data)
fp_data data;
{
  fp_data plus(), fptimes(), trans();
  fp_data d1, d0, res;
  d0 = trans (data);
  d1 = fptimes (d0);
  res = plus (d1);
  return (res);
}
```

It can be seen from the sequential evaluation of this imperative program that the FP functions $trans$, $\alpha*$ and $| +$ are encapsulated by the functions `trans()`, `fptimes()` and `plus()` respectively, thus hiding the implementation details from the user.

6 Conclusion

In summary, this paper has introduced a technique for evolving *pure* functional programs. The advantages of using a pure functional programming language such as FP for program induction are:

- Pure Functional Programs are highly expressive, hence the search space of FP programs is smaller than that of conventional languages.
- As FP advocates a higher order functional style, programs requiring iteration and selection can be produced.
- The FGP system that utilises FP has the ability to find novel, unobvious solutions.
- FP programs can be directly translated to C, allowing a wider range of application.
- The generated pure functional programs contain inherent parallelism and are amenable to formal manipulation.

However, while a number of common programming problems have been addressed by the work presented in this paper, a number of issues remain:

- Only a fraction of the available primitive functions and compositional forms have been employed in the above experiments. The expressive power of individual programs in FGP can be vastly improved if a wider choice of these components are made available to the system.
- Advanced GP techniques such as ADF's, stochastic sampling and adaptive parsimony pressure could be incorporated into the system to facilitate the induction of more complex programs.
- The performance of FGP in a wider range of more complex problems must be investigated. The real benefit of a system such a FGP could be demonstrated by the induction of serial or parallel algorithms for novel programming problems that are encountered on-the-fly.
- The modification of FP to maintain the state of input arguments has proved useful. Similarly, as neither LISP nor FP were originally designed with GP in mind, it may be useful to design the syntax and semantics of a completely new language, specifically for inducting high level programs. At any rate, more use of higher order functions should be made in GP, regardless of the language used. It would also be useful to compare the performance of LISP based GP and FGP.

While the FP style of programming has been shown to be useful for genetic programming, it is not a panacea. However, it does show that iteration and recursion can be addressed to some extend by this programming style. Consequently, the use of higher order functions and pure functional programming in GP should be more widely investigated.

7 Acknowledgements

The author would like to acknowledge the assistance of Ed. Biagioni, Andy Valencia, Seamus Lankford, John O'Mullane and Trent McConaghy.

References

[Backus 78] Backus, J., Can Programming Be Liberated from the von Neumann Style? Communications of the ACM, Volume 21, Number 8, August 1978.

[Banzhaf 98] Banzhaf, W. et al. Genetic Programming: An Introduction, Morgan Kaufmann, 1998.

[Braunl 93] Braunl, T. Automatic Parallelization and Vectorization, Parallel Programming an Introduction, Prentice Hall, 1993, ISBN 0-13-336827-0.

[Cunningham 97] Cunningham, Conrad H., Functional Programming with Gofer, Technical Report UMCIS-1995-01, University of Mississippi, 1997.

[Field 88] Anthony J. Field and Peter G. Harrison, Functional Programming, Addison-Wesley 1988.

[Holland 68] Holland, J. H., Hierarchical descriptions of Universal Spaces and Adaptive Systems, Technical Reports 01252 and 08226, Ann Arbor: University of Michigan, 1968.

[Koza 93] Koza, John R., Genetic Programming, MIT Press 1993.

[Olsson 94] Olsson, Roland (1994) : Inductive Functional Programming Using Incremental Program Transformation, PhD Thesis, University of Oslo 1994.

[Walsh 95] Walsh, Paul, Automatic Conversion of Programs from Serial to Parallel using GP, Advances in Parallel Computing 11, E. Hollander (editor), North Holland Press (1995), ISBN 0-444-82490-1.

[Walsh 98] Walsh, Paul, Evolving Pure Functional Programs, Proceedings of the Third Annual Conference on Genetic Programming, Koza, John R., (editor), University of Wisconson. Morgan Kaufmann (1998).

[Yu 98] Yu, T., and Clack, C., Recursion, Lambda Abstraction and Genetic Programming, Proceedings of GP 98, 1998.