Evolving Pure Functional Programs

Paul Walsh
Computer Science Department
University College Cork
Ireland
paul@cs.ucc.ie

ABSTRACT

This paper introduces a new technique, Functional Genetic Programming (FGP), for the evolution of pure functional programs. This technique is based on the purely functional language FP, which unlike LISP, allows the specification of functional programs without the use of named objects. This allows the evolution of programs that are highly amenable to parallel implementation. This paper describes FGP and presents the results of a number of GP experiments and results show that this technique can generate functional programs that are highly expressive and that are inherently parallel.

1 Introduction

Computer programming has been dominated by von Neumann model of computation, whereby a single processor is connected, via a bus, to storage unit. Conventional programming languages, despite the variety of many complex dialects, operate on this word at a time model of computation, where data is stored and fetched in variables using assignment statements. This extensive use of store and fetch statements, limits the speed of conventional computing systems to the speed at which these operations can be carried out. Moreover, most conventional languages are imperative, whereby the attention of the programmer is focused on how the problem is to be solved, not the problem to be solved.

A more powerful approach is to use a functional language whereby programs are constructed from *pure* functions. Pure functions are defined as those that are referentially transparent, where each function denotes a single value, with no global variables and no side-effects. Therefore in a pure functional program there are no destructive assignment statements. However, although many functional languages, such as LISP, are pure in that they are referentially transparent, dialects have evolved that include constructs for performing destructive assignments which have destroyed the referential transparency and simplicity of the original language.

Moreover, in many functional languages such as LISP, user defined functions are defined in terms of transformations on explicitly named objects. In other words, user defined functions have a function name and an explicitly named argument. For example, the following LISP function consists of a function identifier square, an argument identifier x, and a function body:

```
(defun square (lambda (x) (* x x)))
```

This reference to an explicitly defined argument limits the parallel execution of programs, as a number of data dependencies may arise between statements that reference named objects. However, implicit parallelism is a useful feature of pure functional languages, as functional programs are not written with regard to the von Neumann fetch and execute model of computation. The referential transparency of pure functional programs implies that there are no explicitly defined variables and consequentially there are no shared variables between functions. Therefore functional programs can readily be mapped onto both SIMD and MIMD type architectures [Braunl 93].

2 Functional Programming

FP is a purely functional language that avoids the use of named variables [Backus 78]. An FP system consists of three fundamental components: objects, primitives and combining forms.

Objects are not explicitly defined in FP as objects only appear as run time arguments to functions and the results returned from these functions. The are three basic objects that may be passed to or returned from functions: $\perp$, atoms and lists. The object $\perp$ represents an 'undefined' object and signifies an error condition. An atom in FP is any basic type data type such as integer, real, character or Boolean. A sequence in FP is any enclosed symmetrical list and is *bottom-preserving* so that the sequence is undefined if any of the sequence components are undefined.

A set of predefined functions, known are primitives are defined within the particular implementation of FP. The implementation of FP described in this paper considers the primitives described in Table 1.

User defined functional programs can be constructed using combining forms. There are six basic combining forms defined in FP. The experiments described in this paper use three

Primitive	Function
+,-,*,/	Arithmetic Operators e.g. $+ :< 1,2 >= 3$
iota	Generation of a list of consecutive integers e.g. $iota : 3 =< 1,2,3 >$
trans	Transposition of a matrix e.g. $trans :<< 1,2 >< 3,4 >< 5,6 >>$ $=<< 1,3,5 >< 2,4,6 >>$

Table 1 Primitive functions used in FGP

of these combining forms, namely **composition**, **apply-to-all** and **insert**.

The notion of a sequence in conventional languages is formed in FP by the composition functions. The composition of two functions f and g is written $f \circ g$. For example

$$iota \circ + :< 1,2 >$$

$$iota : + < 1,2 >$$

$$= iota : 3$$

$$=< 1,2,3 >$$

Apply-to-all, denoted α, applies a specified function to every element in a given sequence. For example

$$\alpha + :<< 1,2 >< 3,4 >>$$

$$=< + :< 1,2 > + :< 3,4 >>$$

$$=< 3,7 >$$

Insert, defined as both left-insert and right-insert, is used to reduce an FP sequence by inserting or applying a specified function to a sequence of objects. In the GP experiments described in this paper the right insert operator, denoted $|$, is considered. For example:

$$| + :< 1,2,3 >$$

$$=< 1 + 2 + 3 >$$

$$= 6$$

Functional programs can be written in FP by combining primitive functions, using these combinational forms, to operate on object domain argument.

3 Functional Genetic Programming

The main goal of the FGP system is to evolve pure functional programs that avoid the use of explicitly named objects. This allows the evolution of programs, using GP [Koza 93], that are highly amenable to parallel implementation. Previous work in the evolution of parallel programs, [Walsh 95, Walsh 96, Ryan 97] has focused on the transformation of existing sequential programs into parallel programs. While this area of research has proved fruitful, the effectiveness of such transformations depends ultimately on the runtime characteristics of the original sequential program. This has proved to be a non-trivial task, as most sequential programs are written in an imperative style language and may contain a large number of data dependencies between statements within the program.

With FGP, imperative style programming is eschewed by using the expressive power of FP and data dependency issues are avoided completely by the elimination of all explicit object references. The problem of automatic parallelization of sequential programs is further complicated by the requirement to prove that the transformed program is functionally equivalent to original sequential program. With FGP this issue does not arise as new functional programs are generated to solve a given problem, not to mimic a previous solution.

Individuals in FGP experiments in this paper consist only of primitive functions and the combining forms α and $|$, which are sufficient for evolving solutions for the experiments described. 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 post-order traversal.

The individual FP program corresponding to this is:

$$iota \circ | + \circ iota$$

This individual may be evaluated against a sample fitness case, 3, as follows:

$$iota \circ | + \circ iota : 3$$

$$= iota \circ | + :< 1,2,3 >$$

$$= iota :< 1 + 2 + 3 >$$

$$= iota : 6$$

$$=< 1,2,3,4,5,6 >$$

The advantage of using this representation scheme is that it allows the representation of hierarchical structures, such as α and $|$. Crossover 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.

Fitness is evaluated in FGP by incrementally evaluating the application of each primitive function on the input objects, so that intermediate results are considered when evaluating fitness. Those combining forms that do not result in $\perp$, are applicable and possibly relevant to the input objects and are rewarded by the fitness function. Combining forms that result in $\perp$ will result in a lower fitness score but may prove useful in a later context. For example the following individual may be evaluated against a sample fitness case, 3, as follows:

$$+ \circ | + \circ iota : 3$$

$$= +\circ \mid + :< 1,2,3 >$$
$$= + :< 1+2+3 >$$
$$= + : 6$$
$$= \perp$$

The intermediate results in the evaluation of this example are the vector $< 1,2,3 >$, the integer 6 and the final output $\perp$. Each of these intermediate results are evaluated by the fitness function for a fitness measure. 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.

FGP uses an elitist breeding strategy, where only the top 10-20% individuals are permitted to mate [De Jong 75]. Premature convergence is avoided by the use of a steady state approach [Syswerda 89]. When a fitness measure is evaluated for an individual, it is compared against individuals in the current population. If the individual has scored high enough to enter the population, the lowest scoring member of the population is removed. Individuals are also compared with the current population so as to avoid the insertion of duplicate individuals into the population, thus avoiding premature convergence.

4 Results

FGP was applied to the task of generating functional programs to find the factorial of a given number and the dot product of a pair of vectors. These problem were chosen as they are relatively computationally intensive, they contain inherent parallelism and they illustrate the expressive power of FP.

4.1 Factorial

A basic function in many applications is evaluating the factorial of a number. The fitness function for this problem was the best measure of the output error for each successive application of an individual's combining forms. In initial experiments it was found that as individuals were rewarded for the correct application of primitives to the input arguments, individuals rapidly grew in size. This were avoided by punishing individuals whose length exceeded a maximum limit.

The solution generated by FGP is unique in many respects, considering the standard imperative approach to finding the factorial of a number. The following is a standard procedure for finding the factorial of a number:

```
fact=1;
for i=1 to n do
  fact=fact*i
```

In a functional language that explicitly name objects, such as LISP, factorial may be implemented as:

```
(defun fact
(lambda (x)
  (cond (eq x 0) 1)
   (* x (fact (sub 1 x))))))
```

This function defines an input argument x and makes three explicit references to that input argument x. This makes formal manipulation of this function more difficult and any parallel implementation of this function must contend with data dependencies brought about by the use of this variable x.

It is also worth considering a standard functional approach to finding the factorial of a number [Backus 78], which is similar in style to the above LISP implementation:

$$eq0 \rightarrow 1; * \circ [id, ! \circ sub1]$$

Where $\rightarrow$ is the conditional operator. If the input argument is equal to 0 then 1 is returned, otherwise a list is composed consisting of 1 subtracted from the input argument and the identity of the input argument. The multiplication function is then applied to this list.

This is a refinement to the above lisp implementation as no variables are used. However, the following 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:

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

It is possible to implement this solution in parallel as the inserting operation $\mid$ can be replaced by a synchronous parallel version, which would reduce a list of objects of size n in only $log_2 n$ steps.

4.2 Dot Product

Many high performance computing applications consist of a large number of matrix and vector operations. Finding the dot product of a pair of vectors is a common operation and a conventional von Neumann approach to this problem is as follows:

```
begin
sum:=0;
for i:=1 to n do
  sum:=sum+a[i]*b[i];
end
```

However, the following solution was generated by FGP using the parameters shown in Table 2:

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

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

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

$$|+ \circ \alpha * :<< 1,4 >< 2,5 >< 3,6 >>$$

$$|+ :< * :< 1,4 > * :< 2,5 > * :< 3,6 >>$$

$$|+ :< 4,10,18 >$$

$$< 4 + 10 + 18 >$$

$$32$$

This solution has the following major features:

1. It operates on conceptual units related to the problem, not on words stored in memory.
2. It works for any pair of vectors, regardless of size.
3. It does not use variables or named arguments (unlike LISP or Haskell).
4. The apply-to-all compositional form, α, can be executed in parallel using the synchronous SIMD model of parallel computation.

5 Future Work

This paper serves as an introduction to the automatic generation of functional programs. The problems used in the above experiments are well suited to functional programming and are cited as examples of this programming style [Backus 78, Field 88]. However, more work needs to be done in expanding the application of this approach to other areas, such as the standard GP benchmark problems of symbolic regression and the multiplexer problem [Koza 93].

Areas that would benefit from this approach are those that require parallel execution. The dot product example above could be extended to the evaluation of matrix operations, which are a staple of high performance computing applications. Similar techniques could also be applied to the evolution of parallel programs in other models of computation, such as ability driven dataflow machines.

More work needs to be done in improving the representation of the syntactic structure of individual programs in FGP. A more constrained syntactic structure would avoid the creation and evaluation of individuals that result in the undefined object, $\perp$, thus improving the probability of success for a given generation.

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. The selection of these components could left to the programmer and could depend on the particular application domain.

6 Conclusion

In summary, this paper has introduced a technique for evolving *pure* functional programs. The advantage of evolving pure functional programs are:

- Pure functional programs contain inherent parallelism.
- Pure functional programs can be executed on both MIMD and SIMD models of parallel computation.
- Pure functional programs are amenable to formal manipulation.
- There are no hidden states and complex transition rules.
- Pure Functional Programs are highly expressive.
- The use of names variables and associated side effects is avoided.
- The system has the ability to find novel, unobvious solutions.

References

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

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

[De Jong 75] De Jong, K. A., An Analysis of the Behavior of a class of genetic adaptive systems, (Doctoral dissertation, University of Michigan), Dissertation Abstracts International 36(10), 5140B. (University Microfilms No. 76-9381).

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

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

[Ryan 97] Evolving Parallel Programs, Conor Ryan and Paul Walsh, Genetic Programming 1997, Koza, John R., (editor), Stanford University. San Francisco, Morgan Kaufmann (1997).

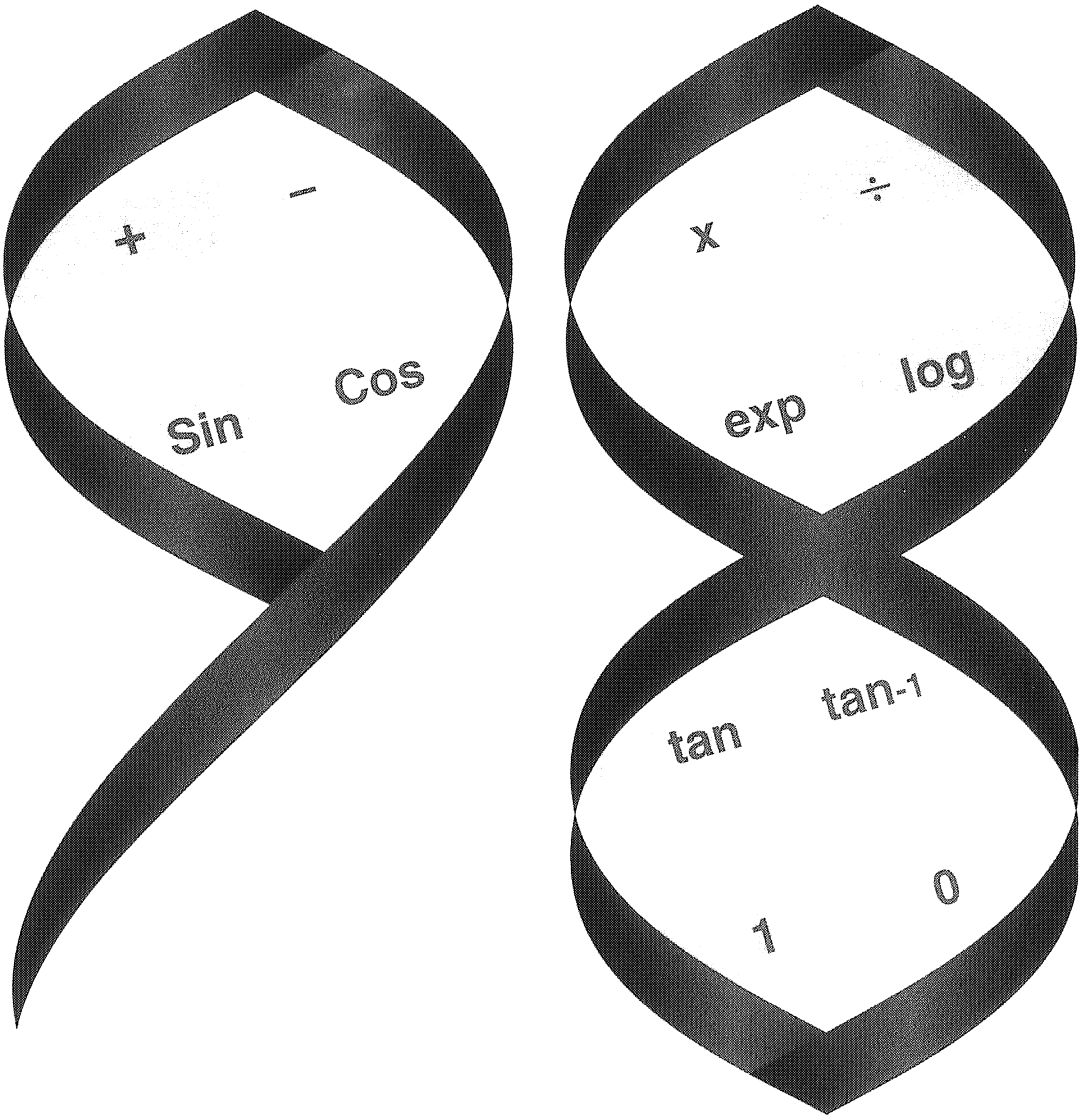
[Syswerda 89] Syswerda G., Uniform crossover in genetic algorithms, In ICCA3, 1989.

[Walsh 95] Paul Walsh and C. Ryan, Automatic Conversion of Programs from Serial to Parallel using GP, Advances in Parallel Computing 11, E. Hollander (editor), North Holland Press (1995).

[Walsh 96] Paul Walsh and Conor Ryan : Paragen : A novel technique for the Autoparallelization of Sequential Programs using GP. In *Genetic Programming 1996* Ed. John Koza et. al. Cambridge : MIT Press.

CONFERENCE PROCEEDINGS

Genetic Programming



edited by
John R. Koza
Wolfgang Banzhaf
Kumar Chellapilla
Kalyanmoy Deb
Marco Dorigo
David B. Fogel
Max H. Garzon
David E. Goldberg
Hitoshi Iba
Rick L. Riolo

Proceedings of the Third Annual Genetic Programming Conference

July 22–25, 1998
University of Wisconsin, Madison, Wisconsin