Using Scaffolding with Partial Call-Trees to Improve Search

Brad Alexander(✉), Connie Pyromallis, George Lorenzetti, and Brad Zacher

School of Computer Science, University of Adelaide, Adelaide 5005, Australia
bradley.alexander@adelaide.edu.au
http://www.cs.adelaide.edu.au/~brad

Abstract. Recursive functions are an attractive target for genetic programming because they can express complex computation compactly. However, the need to simultaneously discover correct recursive and base cases in these functions is a major obstacle in the evolutionary search process. To overcome these obstacles two recent remedies have been proposed. The first is Scaffolding which permits the recursive case of a function to be evaluated independently of the base case. The second is Call-Tree-Guided Genetic Programming (CTGGP) which uses a partial call tree, supplied by the user, to separately evolve the parameter expressions for recursive calls. Used in isolation, both of these approaches have been shown to offer significant advantages in terms of search performance. In this work we investigate the impact of different combinations of these approaches. We find that, on our benchmarks, CTGGP significantly outperforms Scaffolding and that a combination CTGGP and Scaffolding appears to produce further improvements in worst-case performance.

Keywords: Recursion · Genetic programming · Call-tree · Scaffolding · Grammatical evolution

1 Introduction

Recursive functions solve challenging problems by defining larger solutions in terms of sub-solutions. Recursive functions are often compact and expressive, which has made the evolution of recursive functions a popular target for genetic programming (GP) [5,6,9]. Unfortunately, the evolution of non-trivial recursive functions through GP has proven difficult in practice [1,6]. One cause of this difficulty is that fitness functions based solely on test cases are very sensitive to the correctness of the candidate code for the base case. Thus, when the base case's code is wrong, the fitness function will often give very low fitness to candidate solutions even when the major part of the code – the recursive case – is entirely correct [6]. This dependence results in the need to simultaneously evolve correct code for both the base and recursive cases [1,7] before a high fitness score is achieved.

Several approaches to improve search for recursive functions have been implemented. These have included: the use of niches to preserve diversity during search

© Springer International Publishing AG 2016
J. Handl et al. (Eds.): PPSN XIV 2016, LNCS 9921, pp. 324–334, 2016.
DOI: 10.1007/978-3-319-45823-6_30

[7] and narrowing search-spaces using templates that express common patterns of recurrence [10,11]. Moraglio et al. [6] defined a simple and general approach, called *Scaffolding*, which employed existing test cases to always return a correct result for recursive calls. This allowed the fitness of the recursive case to be gauged independently of the base case. Later, Alexander and Zacher [2] defined Call-Tree-Guided Genetic Programming (CTGGP), which used information in user-defined partial call trees to separately deduce recursive calls. More recently, Chennupati et al. [4] implemented Multi-core Grammatical Evolution on parallel platforms to speed up the evolution of parallel recursive integer and list functions.

In this paper we benchmark both Scaffolding and CTGGP alone and in combination. The question of which configuration of these frameworks gives the most benefit is interesting, because both Scaffolding and CTGGP showed significant improvement over unassisted GP search on their respective recursive benchmarks. A-priori, Scaffolding and CTGGP have a complementary focus: Scaffolding allows the base case to be evolved without impacting the search for the recursive case and, in contrast, CTGGP separately evolves the parameter to recursive calls. To date, there has been no research that has compared the two approaches either in isolation or in combination. In this paper we compare the performance of Scaffolding and CTGGP on a range of recursive benchmarks. We show that CTGGP performs better than Scaffolding in isolation and the combination of the two approaches marginally improves average performance, but consistently improves worst-case performance. We also describe improvements to the CTGGP system and briefly examine their impact.

The remainder of this paper is structured as follows. In the next section we describe the conceptual frameworks for Scaffolding and CTGGP. In Sect. 3 we describe an implementation of our framework combining Scaffolding and CTGGP. In Sect. 4 we describe our experimental parameters and results and, finally, in Sect. 5 we present our conclusions and ideas for future work.

2 Conceptual Frameworks

In this section we describe the concepts of Scaffolding and CTGGP. Both concepts are enhancements to Genetic Programming (GP). Here we define a GP search framework as a function $GPSearch$ which attempts to discover a target code fragment f. To define f's behaviour, $GPSearch$ requires test-cases in the form of a list of inputs to f: $in = [i_1, \ldots, \imath_n]$ and a corresponding list of desired outputs from f: $out = [o_1, \ldots, o_n]$. Given these definitions, the $GPSearch$ can be defined:

$$GPSearch(in, out) = f = \mathop{\mathbf{argmin}}_{f_x \in Genset} Error(f_x, in, out)$$

where $Genset$ is the set of candidate functions that is generated by $GPSearch$ and $Error$ is an error function that returns a measure of how much the outputs of f_x deviate from the desired outputs in out. In GP, $Genset$ is generated dynamically by an evolutionary process guided by the values of $Error$. In order to make

search tractable, the elements of *Genset* are usually constrained to belong to a grammar relevant to the problem domain of interest.

In this paper, all of our target benchmarks f are recursive integer functions of one argument that return one value. We constrain the members of *Genset* to this form using Grammatical Evolution (GE) [8] – a form of GP which maps a binary string genotype to code via a BNF grammar provided by the user. In the following, we define *Error* as:

$$Error(f_x, in, out) = \Sigma_{k=1}^{n} |f_x(in_k) - out_k|$$

The aim of our search is to converge to target f such that $Error(f, in, out) = 0$. Both Scaffolding and CTGGP extend the framework just described. We describe these extensions in turn.

2.1 Scaffolding

The motivation for Scaffolding stems from the observation that any candidate recursive function f_x applied to test cases *in* and *out* will receive a very poor fitness evaluation unless the base case is correct. For example, in the `fib` function shown in Algorithm 1 a change of the base case guard on line 2 to `x` $<$ `1` will cause the program to fail on most foreseeable test cases. Likewise a change of the base case body on line 3 to `return 1` also causes most test cases to fail. This happens despite the remaining code being entirely correct.

Algorithm 1. Correct C implementation of the Fibonacci function

```
1  int correctRecurse(int x) {
2    if (x >= orig_x) {
3      return recurse(x);
4    }
5    for (int i = 0; i < n; i++) {
6      if (inp[i] == x) return out[i];
7    }
8    return recurse(x);
9  }
```

This sensitivity to errors in the base case has a significant impact on search. Moraglio proposed Scaffolding [6] to help address this problem. Under Scaffolding, every recursive call is replaced by a call to a non-recursive function that returns the correct value for that call. The correct value for each call is mined from the original test cases in *in* and *out*. Thus, for example, Scaffolding would replace the call fib(2), where $in = [0, 1, 2, 4, 6]$ and $out = [0, 1, 1, 3, 8]$, with the call correct-fib(2) which would then return $out_2 = 1$. If the input to the call for the recursive function doesn't correspond to a value in *in*, then recursion is allowed to progress as

normal until a call with a value in *in* is encountered. After evolution has completed Scaffolding replaces all calls with the original function name.

The above scheme makes it possible to evaluate the correctness of the recursive case somewhat independently of the base case. The correctness of the base case still contributes to fitness to the extent that it is directly or indirectly exercised by *in*.

A final detail that must be addressed in Scaffolding is avoiding cases where the parameter of the recursive call matches the original parameter to the function. Without intervention, Scaffolding would find these cases to be correct while the actual program when run would make no progress. Scaffolding handles these cases by explicitly detecting them and avoiding replacement with the correct version of the call. This leads to infinite recursion and, thus, poor fitness.

2.2 CTGGP

CTGGP speeds up the GP search process by conducting a separate search for the parameters of the recursive call. Thus, if the target function is `fib` in Fig. 1, CTGGP would conduct a search for parameter expressions `x-1` and `x-2` separately from the rest of the code.

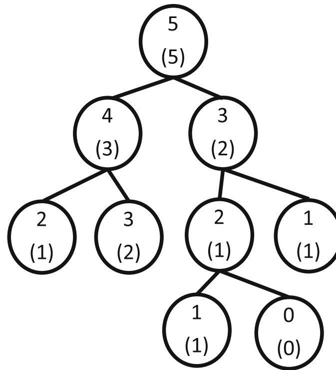

Fig. 1. An example partial call tree for a fib function

This search is guided by information embedded in a user-provided partial call tree. An example of a call for the `fib` function is shown in Fig. 1. The nodes of the call tree can contain up to two numbers. The top number is the parameter of the recursive call. The bottom number, in brackets, is the value returned by that call. Note that this return value is not required for every node in the tree. In addition, the user doesn't have to provide every call starting from the call at the root node. Moreover, the tree is allowed to be disjoint and overlapping calls from parent nodes can be shared, forming a directed-acyclic graph. The premise underpinning CTGGP is that in order to create the input and output

values required to drive a GP process the user will often sketch an approximation of a call tree in order to obtain these input and output values. Thus, by using CTGGP we are merely exploiting information that the user already has in hand.

The partial call tree can be mined for information to guide CTGGP. The values in *in* are the top values in each node. The values in *out* are the bottom values in each node. In addition, CTGGP uses the structure of the tree to create a list of tree-segments: *segs* which contains a list of entries $[s_1, \ldots, s_m]$ where each s_i is a pair $(pnt_i, [c_{i1}, \ldots, c_{im_i}])$ which defines the relationship between the parent: pnt_i and its child nodes in the call tree. As an example of the correspondence between the call tree and the values above the values extracted from the call tree shown in Fig. 1 are shown in Fig. 2

$$in = [5, 4, 3, 2, 3, 2, 1, 1, 0]$$
$$out = [5, 3, 2, 1, 2, 1, 1, 1, 0]$$
$$segs = [(5, [4, 3]), (4, [2, 3]), (3, [2, 1]), (2, []), (3, []), (2, [1, 0]), (1, []), (0, [])]$$

Fig. 2. Values extracted by CTGGP from the call tree in Fig. 1

The search process in CTGGP starts when the user specifies the call tree via a GUI interface. The interface application infers from the arity of tree fragments, with confirmation from the user, details of the grammars to be used. These details include the number of base cases and the inferred number of recursive calls [2]. From this step two grammars are produced. The first grammar, called *grammar1*, is used for searching for the parameters of the recursive calls. An example of this grammar for the call tree in Fig. 1, is shown in Fig. 3(a). A second grammar, called *grammar2*, is used for searching for the remainder of the function. The corresponding grammar2 is shown in Fig. 3(b). The candidate expressions generated from grammar1 are referenced in part (b) by the expressions **param1** and **param2**. The recursive calls in grammar2 are always denoted by the generic name "recurse".

Once the grammars have been defined, evolution in CTGGP proceeds in two concurrent phases of GE. Phase 1 produces individual expressions from grammar1 in order to produce parameter expressions for the recursive call(s). Phase 2 uses grammar2 to evolve the rest of the recursive function. During phase 2 evolution, the current best expressions from phase 1 are integrated into the candidate solutions. This simultaneous evolution is an improvement on our previous work [2] which ran phase 1 and phase 2 in sequence and thus required us to make an a-priori estimate of the amount of time phase 1 would require to find an acceptable solution.

In phase 2, individual candidate solutions f_x are evaluated by calling the $Error(f_x, in, out)$ function defined at the beginning of this section. In phase 1 the parameter expressions produced are evaluated by comparing the results of the candidate parameter expressions to the corresponding entries in *segs*. As an example of how this is done for the call tree in Fig. 1, consider a phase 1 search process for `fib` that produces the (correct) expressions: **param1** $= x - 1$ and

```
<expr_root> ::= <var> <op> <digit> |
  <digit> <op> <var>
<op> ::= - | * | + | /
<digit> ::= 0 | 1 | 2 | <big_digit>
<big_digit> ::= 3 | 4 | 5 | <bigger_digit>
<bigger_digit> ::= 6 | 7 | <huge_digit>
<huge_digit> ::= 8 | 9
<var> ::= x
```

(a)

```
<expr_root> ::=
  if(<var> < <digit>){
    return <lit>;
  }else{
    return <expr1> <op> <expr2>;
  }
<expr1> ::= <rec1> |
  (<rec1> <op> <lit>) |
  (<lit> <op> <rec1>)
<expr2> ::= <rec2> |
  (<rec2> <op> <lit>) |
  (<lit> <op> <rec2>)
<rec1> ::= recurse(param1)
<rec2> ::= recurse(param2)
<op> ::= - | * | +
<lit> ::= <digit> | <var>
... as per phase 1 grammar…
```

(b)

Fig. 3. Grammar1 (a) and grammar2 (b) generated by the call tree in Fig. 1.

param2 $= x - 2$. To test these we extract the first entry from *segs* from Fig. 2: $(5, [4, 3])$. This entry consists of the parent parameter 5 and the child parameters $[4, 3]$. We substitute the parent parameter 5 into each of **param1** and **param2**. These produce, respectively, the values 4 and 3. These are eliminated from the child parameter list leaving an empty list. The matching process then progresses other elements of *segs*. If all child parameter entries in *segs* are eliminated, then perfect fitness is given. If some items fail to match then fitness is penalised according to the distance between the output of the phase 1 expression and the closest match in the children of the relevant entry in *segs*.

The whole evolution progresses until either phase 2 evolution finishes with a perfect score or the maximum number of generations for phase 2 is reached.

3 Implementation and Experimental Setup

In this section we outline how Scaffolding is combined with CTGGP and how the experiments we use to measure their performance are set up.

To implement Scaffolding we need to embed code to access the correct answer to each recursive call in each candidate solution f_x. In our system this is done by implementing a function called `correctRecurse` that takes the place of the recursive calls: `recurse` in the phase 2 grammar. `correctRecurse` is implemented as part of the library code accessed by the candidate solution. The code for `correctRecurse` is shown in Algorithm 2. The variable `orig_x` is a global variable containing the value of the original call to `recurse`. The variables `in` and `out` are arrays representing *in* and *out* respectively. The variable `n` represents the length of both *in* and *out*. The if-statement on lines 2 to 4 checks to see if the parameter is the same as that of the original call. If so, it forces a call

Algorithm 2. Implementation of correctRecurse

```
1  int correctRecurse(int x) {
2    if (x >= orig_x) {
3      return recurse(x);
4    }
5    for (int i = 0; i < n; i++) {
6      if (inp[i] == x) return out[i];
7    }
8    return recurse(x);
9  }
```

back to `recurse` which, in this case, will usually result in infinite recursion and timeout thus giving a low fitness. The for-loop on lines 5 to 7 checks to see if `x` is part of *in*, if so, it will return the corresponding element of *out*. Otherwise the `recurse` function is called on line 8 with the possibility that it will, eventually, call `correctRecurse` again with a parameter that is in *in*.

The rest of the CTGGP framework remains the same as previously described. The framework itself is implemented in C++ and uses libGE version 0.26[1] to generate individuals and carry out search. Candidate functions are generated using C grammars and these are compiled into scaffold libraries using the Tiny-C-Compiler [3] (TCC). In phase 1 evolution, the test harness compares the output of the evolved parameter expressions with the elements of *segs*. In phase 2 evolution, which runs in a separate thread, the whole function is tested against the values in *in* and *out*. The phase 1 and phase 2 threads communicate via a C source file containing the parameter expressions generated by phase 1. This file is locked while being accessed so that our code remains thread-safe.

In our experiments we compare four different configurations on a range of target benchmarks. The four experiments are **Plain** - GE run against a grammar for each problem without Scaffolding or CTGGP; **Scaffolding** - GE run with Scaffolding but without separate evolution of parameters to recursive calls; **CTGGP** - GE running with CTGGP; and **Combined** combining CTGGP with Scaffolding as described above.

The target benchmarks are, for an integer parameter (n): *factorial* returns the factorial of n; *odd-evens* returns 0 if $n \bmod 2 = 0$ and 1 otherwise; *log2* finds $\lfloor \log_2 n \rfloor$; *fib* and *fib3* calculates the Fibonacci and Fibonacci-3 number for n; *lucas* calculates the nth Lucas number (this requires two base cases); and *pell* calculates the *nth* Pell number.

For the CTGGP runs, we use small call trees which provide information on the first five to six elements of the sequence. With our GUI these cases take less than 5 min each to draw. To enable a fair comparison for the non-CTGGP runs we simply integrate the phase1 grammar into the phase 2 grammar which defines a set of valid target programs. This setup allows the non-CTGGP

[1] This framework can be downloaded http://bds.ul.ie/libGE/.

experiments to take advantage of the specialised grammars generated with information from the structure of the call tree. This specialisation will positively influence their performance, making our relative estimates of the performance of CTGGP conservative.

In all experiments we use GE running on an underlying steady-state GA with tournament selection. The GE parameters remain same as the hand-tuned parameters from phase 2 from [2]. The replacement probability use is 0.25 and probabilities for crossover and mutation are, respectively, 0.9 and 0.01. In all experiments and both phases in the CTGGP experiments we use a population of 1000 running for 300 generations. The phases are set up to terminate early if an individual with perfect fitness is encountered. In all experiments using CTGGP phase 1 terminated faster than phase 2 so phase 2 statistics serve as the best indicator of the time the algorithm takes.

We ran our experiments on an AMD Opteron 6348 machine with 48 processors running at 2.8 GHz. When the load on the machine was light the average evaluation time per individual ranged from 2 ms to 20 ms depending on the complexity of the benchmark. All experiments were run for 50 trials.

4 Results and Discussion

Table 1 shows the results of our experiments. The columns show, respectively, the data for each experiment. The rows show the results for each benchmark – broken down into mean number of phase 2 evaluations ($\overline{\mathbf{x}}$), number of correct answers (**nc**) and worst case number of evaluations (**max**). Where not all runs in an experiment resulted in success the value of **max** simply indicates the time when the longest run terminated rather than maximum time-to-success. These cases are marked with an asterisk. We also mark bold an entry for $\overline{\mathbf{x}}$ if it is significantly better than corresponding value in the previous column (according to a log-rank test). As can be seen, on most benchmarks Scaffolding significantly outperforms plain GE. CTGGP significantly outperforms Scaffolding on all benchmarks – pointing to the advantage of utilising call tree information when it is available. In the last column Combined, only significantly outperforms CTGGP alone on the lucas benchmark. Exploring further, it can be seen that the value for $\overline{\mathbf{x}}$ is at least marginally better for combined than for CTGGP for all benchmarks. Moreover, the value of **max** is substantially lower for Combined and for CTGGP indicating that the combination may be a strategy for moderating worst-case performance.

As previously mentioned, in the latest implementation of CTGGP phase 1 and 2 evolution run concurrently. This means that on machines with spare processing capacity there is insignificant time overhead incurred from running phase 1. However, it is still interesting to observe how phase 1 and 2 interact over time. Figure 4 plots the best phase 1 fitness, best phase 2 fitness and and average phase 2 fitness against time for a long (75th percentile) run of the Combined framework on the fib3 benchmark.

Table 1. Mean number of evaluations ($\overline{\mathbf{x}}$), number of correct answers (**nc**) and worst case number of evaluations (**max**) for the four experimental configurations.

Problem		Plain	Scaffolding	CTGGP	Combined
factorial	$\overline{\mathbf{x}}$	4109	**2542**	**219**	157
	nc	50	50	50	50
	max	8942	7032	1403	366
oddeven	$\overline{\mathbf{x}}$	539	478	**269**	255
	nc	50	50	50	50
	max	2069	1845	1201	885
log2	$\overline{\mathbf{x}}$	21524	**9049**	**1404**	1206
	nc	50	50	50	50
	max	103783	22489	11527	3845
fib	$\overline{\mathbf{x}}$	53168	**31733**	**1189**	1081
	nc	40	49	50	50
	max	130923*	130107*	3698	3617
fib3	$\overline{\mathbf{x}}$	117875	**94818**	**12614**	10347
	nc	3	18	50	50
	max	125723*	124448*	84271	40771
lucas	$\overline{\mathbf{x}}$	105663	**35820**	**3081**	**1622**
	nc	8	49	50	50
	max	123455*	127936*	12288	7070
pell	$\overline{\mathbf{x}}$	56240	**28887**	**2127**	1879
	nc	41	49	50	50
	max	129823*	128358*	6186	4904

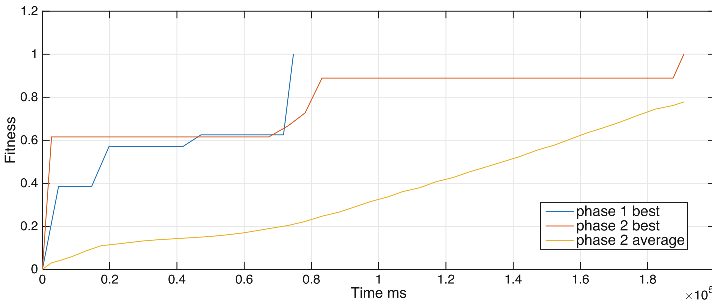

Fig. 4. Plot of fitness against time for the best individual in phase 1 and the best and average individual in phase 2 for the fib3 benchmark. (Color figure online)

As can be seen, phase 2 is able to make some progress, particularly in average fitness, while phase 1 is below perfect fitness - this indicates that even with incorrect parameter expressions search can progress. The speed of phase 2

search improves once phase 1 has produced the required parameters and terminated. This pattern of behavior is similar to that observed in other runs we have inspected.

5 Conclusions and Future Work

In this paper we have explored the effect of Scaffolding, CTGGP and a combination of these on GP search on a range of recursive benchmarks. We have shown that both Scaffolding and CTGGP significantly improve GP performance and there are indications that combining these is beneficial in terms of improving worst case performance. We have also shown that it is productive to run phase 1 and phase 2 evolution of CTGGP concurrently.

This work can be extended in several ways. We could further exploit the relationships between calling values and return values in the tree to help induct code that combines return values. We can extend the benchmarks for these experiments to include recurrences in loops. Finally we can conduct a more extensive study to confirm the effectiveness of the combined framework in reducing worst case times.

References

1. Agapitos, A., Lucas, S.: Learning recursive functions with object oriented genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) EuroGP 2006. LNCS, vol. 3905, pp. 166–177. Springer, Heidelberg (2006)
2. Alexander, B., Zacher, B.: Boosting search for recursive functions using partial call-trees. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 384–393. Springer, Heidelberg (2014)
3. Bellard, F., Tcc: Tiny C compiler (2003). http://fabrice.bellard.free.fr/tcc
4. Chennupati, G., Azad, R., Ryan, C.: Performance optimization of multi-core grammatical evolution generated parallel recursive programs. In: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1007–1014. ACM (2015)
5. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M.: Genetic Programming Darwinian Invention and Problem Solving. Morgan Kaufman, Burlington (1999)
6. Moraglio, A., Otero, F.E.B., Johnson, C.G., Thompson, S., Freitas, A.A.: Evolving recursive programs using non-recursive scaffolding. In: IEEE Congress on Evolutionary Computation, pp. 1–8 (2012)
7. Nishiguchi, M., Fujimoto, Y.: Evolution of recursive programs with multi-niche genetic programming (mnGP). In: Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, pp. 247–252 (1998)
8. Ryan, C., Collins, J.J., Neill, M.O.: Grammatical evolution: evolving programs for an arbitrary language. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, pp. 83–96. Springer, Heidelberg (1998)
9. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. Genet. Program. Evolvable Mach. **3**, 7–40 (2002)

10. Wong, M.L., Leung, K.S.: Evolving recursive functions for the even-parity problem using genetic programming. In: Advances in Genetic Programming, pp. 221–240. MIT Press (1996)
11. T. Yu and C. Clark. Recursion, lambda-abstractions and genetic programming. Cognitive Science Research Papers-University Of Birmingham CSRP, pp. 26–30 (1998)