# Simultaneous Synthesis of Multiple Functions using Genetic Programming with Scaffolding

Iwo Błądek and Krzysztof Krawiec
Poznan University of Technology, 60965 Poznań, Poland
{iwo.bladek,krawiec}@cs.put.poznan.pl

## ABSTRACT

We consider simultaneous evolutionary synthesis of multiple functions, and verify whether such approach leads to computational savings compared to conventional synthesis of functions one-by-one. We also extend the proposed synthesis model with scaffolding, a technique originally intended to facilitate evolution of recursive programs, and consisting in fetching the desired output from a test case, rather than calling a function. Experiment concerning synthesis of list manipulation programs in Scala allows us to conclude that parallel synthesis indeed pays off, and that engagement of scaffolding leads to further improvements.

## Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

## Keywords

genetic programming, scaffolding, multisynthesis, problem decomposition, Scala

## 1. MULTISYNTHESIS

A problem of synthesizing a single program realizing certain task is a search problem defined by a *contract* $C$, which defines the desired behaviour of function $f$, and a set of instructions $I$ of which $f$ is to be constructed. We consider simultaneous synthesis (*multisynthesis*) of multiple functions $f_1, \ldots, f_n$ specified as a $n$-tuple of contracts $C_1, \ldots, C_n$, each defining the desired behavior of the corresponding function $f_i$, and an $n$-tuple of instruction sets $I_1, \ldots, I_n$ of which the $f_i$s are to be constructed, such that $\forall_{i=1,\ldots,n} \forall_{j \neq i} f_j \subset I_i$, i.e. functions may call each other. A solution to a multisynthesis problem is an $n$-tuple of correct programs $(p_1, \ldots, p_n)$ that meet the corresponding contracts in the above-defined sense, $\forall_i p_i \equiv_{C_i} f_i$. Availability of recursive calls (i.e., including $f_i$ in $I_i$) is not an essential characteristics of a multisynthesis problem, so we leave them out from the formulation.

We say that the $i$th synthesis problem *depends* on the $j$th synthesis problem if and only if the function $f_i$ to be synthesized, in order to meet the contract $C_i$, needs to call $f_j$ (i.e., $f_j$ is necessary for $f_i$ to be synthesized). If the $i$th problem is independent on all remaining problems, it becomes a conventional synthesis problem and can be solved separately. However, an important motive for this paper is that independence does not necessarily mean that a problem *should* be solved separately, because solving it with the help of the remaining functions may be easier and lead to shorter and more legible code, as demonstrated by everyday practice in software engineering. Note also that existence of a dependency may be unknown for a given pair of functions in a multisynthesis task.

**Recursive scaffolding** [2] has been proposed as a technique to facilitate evolutionary synthesis of recursive programs. The main problem with evolving recursive functions with genetic programming (GP) is the abrupt deterioration of fitness for recursive programs that are structurally only slightly different from the optimal solution. We generalize scaffolding to multisynthesis problems: a **scaffolded multisynthesis** algorithm replaces the calls to a function $f_i$ (all or some of them) with 'calls' to the corresponding contract $C_i$. We consider two variants:

**1. Test-based scaffolding** (applicable to contracts specified by tests). Given an invocation $f_i(x)$, we seek a test of the form $(x, y)$ in contract $C_i$. If such a test is found, we return $y$; otherwise, we call $p_i(x)$, i.e., the implementation of $f_i$ known at the given moment of search, whether it is correct or not.

**2. Oracle-based scaffolding**. This variant is applicable in scenarios where contracts are 'executable', i.e., can serve as oracles. Given an invocation $f_i(x)$, we apply the corresponding oracle $C_i$ to $x$ and return the result. On the face of it, oracle-based scaffolding may seem a contradiction in terms: why would one attempt to synthesize a program $p_i$ if the corresponding oracle $C_i$, i.e., basically a *program*, is already available? There are, however, plausible usage scenarios for this case; consider for instance reverse engineering of existing (but closed) software or hardware, which can be executed but cannot be used directly due to technical or legal issues.

**Experiment.** Our experimental environment is a subset of Scala, a hybrid object-oriented and functional programming language. Using the FUEL framework[1] and related

[1]https://github.com/kkrawiec/fuel



*GECCO '16 July 20-24, 2016, Denver, CO, USA*



ACM ISBN 978-1-4503-4323-7/16/07.

DOI: http://dx.doi.org/10.1145/2908961.2908992

Table 1: The benchmarks. `T` stands for generic type.

| Benchmark | Dependant function | Helper functions | Number of tests |
|---|---|---|---|
| `last` | `last(list:List[T]):T` | `at,size` | 20 |
| `patch` | `patch(list:List[T],d:Int,p:List[T],u:Int):List[T]` | `drop,take` | 39 |
| `slice` | `slice(list:List[T],d:Int,u:Int):List[T]` | `drop,take` | 19 |
| `splitAt` | `splitAt(list:List[T],i:Int):(List[T],List[T])` | `drop,take` | 24 |

libraries, we evolve Scala *expressions*, without the possibility of creating values (`val`) or variables (`var`), and exclude certain sophisticated functional programming mechanisms. We consider four selected methods of the immutable `List` class (Table 1). Out of over 170 methods available in that class, we select triples of functions such that at least one of them (*dependent function*) can be implemented by calling the other two (*helper functions*). Each contract $C_i$ that defines program's desired behavior is specified by a sets of tests (numbers of tests are given in Table 1). Solutions are not allowed to contain cyclic calls between contained functions, which was motivated by the observation that such calls would rarely form a sensible solution. We consider two scenarios of synthesizing the three requested functions:

In **sequential scenario (SEQ)**, functions are synthesized one by one, in three evolutionary processes that follow each other. Once a given function is synthesized, a GP process for the next function is started with the remaining computation budget. A candidate solution comprises one function, its fitness is the number of passed tests, and selection operator is tournament of size 7. In $\text{SEQ}_{opt}$, functions are synthesized in a problem-specific 'optimal' order, i.e., the helper functions followed by the dependent function (e.g., for SLICE: `drop`, `take`, `slice`). In $\text{SEQ}_{exp}$, we assume that the optimal ordering of functions' synthesis is unknown, and split the computation budget of 25 runs between all $3! = 6$ permutations of functions by drawing a permutation randomly (with uniform distribution) for each run. The total success rate approximates the overall expected probability of synthesizing a correct candidate solution.

In **parallel scenario (PAR)**, there is one evolutionary run and each candidate solution is a triple of candidate programs. In evaluation phase, the programs are verified on corresponding tests, which yields three objectives. We consider a single- and a multiobjective variant of this scenario. In PAR, the fitness is the sum of fractions of tests passed by particular functions, i.e., a scalar value ranging in $[0, 3]$, which is then subject to tournament of size 7. In $\text{PAR}_{nsga}$, the three objectives remain separate and form the basis of selection using the NSGA-II algorithm [1] with default parameter settings (rank-based tournament size set to 2). PAR configurations occur in two variants, without (PAR and $\text{PAR}_{nsga}$) and with oracle-based scaffolding ($\text{PAR}^s$ and $\text{PAR}^s_{nsga}$). In the former configurations, the current implementation of the function is used to obtain the return value.

**Results.** In Table 2, we present the number of correctly synthesized constituent functions in best-of-run individuals averaged over benchmarks. As expected, $\text{SEQ}_{opt}$ dominates all other configurations. The knowledge on the optimal ordering of functions to be synthesized is clearly helpful. This scenario assumes an omniscient designer and is thus only of theoretical importance, so we do not rank it. $\text{SEQ}_{exp}$ features the lowest fraction of correctly synthesized functions, and the parallel scenarios fare better in that respect. Among them, the multiobjective variants fare on average better than the scalar ones, which corroborates the common observation that multifaceted characterization of candidate solutions improves population diversification, lowers the risk of premature convergence, and increases the odds of success. Oracle-based scaffolding makes $\text{PAR}^s_{nsga}$ better than $\text{PAR}_{nsga}$, while deteriorating (albeit only slightly) the performance of PAR. The overall best performing variant is $\text{PAR}^s_{nsga}$, which confirms the merit of combining parallel scaffolded synthesis with multiobjective evaluation.

Table 2: Average number of synthesized functions.

| Method | `last` | `patch` | `slice` | `splitAt` | Average |
|---|---|---|---|---|---|
| $\text{SEQ}_{opt}$ | 2.23 | 2.13 | 1.36 | 2.69 | 2.10 |
| $\text{SEQ}_{exp}$ | 1.11 | 1.73 | 0.97 | 1.56 | 1.34 |
| PAR | 2.00 | 1.50 | 0.72 | 1.80 | 1.51 |
| $\text{PAR}^s$ | 1.90 | 1.47 | 0.59 | 1.94 | 1.48 |
| $\text{PAR}_{nsga}$ | 2.03 | 1.47 | 0.53 | 2.17 | 1.55 |
| $\text{PAR}^s_{nsga}$ | 2.18 | 1.40 | 1.05 | 1.88 | 1.63 |

**Conclusions.** We hypothesize that the main factor that makes parallel multisynthesis more effective is that the presence of multiple contracts transforms the fitness landscape. The considered scenarios are essential to meet the demands of real-world programming. Synthesizing programs in isolation inevitably leads to reinventing the wheel, i.e., repetition of code fragments that could be reused by calling other functions. This issue can be addressed by synthesizing the programs in the 'right' order and allowing the functions synthesized later to call the already synthesized functions. One cannot however assume that the knowledge on this optimal ordering is always available. Moreover, the order of synthesis dictated by dependencies between functions does not necessarily lead to the highest chance of success. These issues incline us to favor the parallel variant, which fared well on the considered benchmarks, especially when equipped with scaffolding and multiobjective evaluation.

The anticipated follow-ups of this work will target elitism, other ways of defining best-of-run solutions, other domains than list-manipulating programs, and scalability with the number of functions.

**Acknowledgments.** The authors acknowledge support from grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland.

## 2. REFERENCES

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197, apr 2002.

[2] A. Moraglio, F. Otero, C. Johnson, S. Thompson, and A. Freitas. Evolving recursive programs using non-recursive scaffolding. In *Proceedings of the 2012 IEEE Congress on Evolutionary Computation*, pages 2242–2249, Brisbane, Australia, 10-15 June 2012.