Stochastic Program Synthesis via Recursion Schemes

Jerry Swan
Department of Computer Science,
University of York
York, United Kingdom
dr.jerry.swan@gmail.com

Krzysztof Krawiec
Institute of Computing Science,
Poznań University of Technology
Poznań, Poland
krawiec@cs.put.poznan.pl

Zoltan A. Kocsis
School of Mathematics,
The University of Manchester
Manchester, United Kingdom
zoltan.kocsis@existence.property

ABSTRACT

Stochastic synthesis of recursive functions has historically proved difficult, not least due to issues of non-termination and the often *ad hoc* methods for addressing this. We propose a general method of implicit recursion which operates via an automatically-derivable decomposition of datatype structure by cases, thereby ensuring well-foundedness. The method is applied to recursive functions of long-standing interest and the results outperform recent work which combines two leading approaches and employs 'human in the loop' to define the recursion structure. We show that stochastic synthesis with the proposed method on benchmark functions is effective even with random search, motivating a need for more difficult recursive benchmarks in future. This paper summarizes work that appeared in [1].

CCS CONCEPTS

• **Theory of computation** → *Theory of randomized search heuristics*; *Functional constructs.*

KEYWORDS

program synthesis, algebraic data types, catamorphisms, pattern matching, recursion schemes

ACM Reference Format:
Jerry Swan, Krzysztof Krawiec, and Zoltan A. Kocsis. 2019. Stochastic Program Synthesis via Recursion Schemes. In *Proceedings of the Genetic and Evolutionary Computation Conference 2019 (GECCO '19 Companion).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3319619.3326758

1 INTRODUCTION

Synthesis of recursive programs is of long-standing interest in the Genetic Programming (GP) community [10], with a wide variety of proposed approaches (see e.g. [2, 5, 11, 12, 14]). Most of these proposals focus on *explicit* recursion, i.e. recursion expressed directly within the body of the synthesised code. In contrast, previous work by Yu et al. [15–17] demonstrated the utility of *implicit* recursion, where the control flow of the recursion is orchestrated by a specific template of individually handled *cases*. The implicit approach makes no recursive calls in the synthesized code: these are instead delegated to an external, fixed *combinator*: a stateless function that factors out the recursion pattern.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07.
https://doi.org/10.1145/3319619.3326758

The implicit approach has several advantages. It ensures that the recursion is well-founded, thus bypassing the issue of non-termination. The search space of the implicit case is likely to be smaller than that of the explicit approach, since the code for partitioning the recursion into base and alternative cases is provided by a template. The cases further constrain the list of fitness cases (examples) that can be used in GP-based search, thereby reducing the computational expense. Yu's method of implicit recursion was applied to the *List* data-type via its associated *fold* method. The *fold* method of *List* is a higher-order function that takes as argument a callback function used to accumulate information as the *fold* traverses the list. Although *fold* can express a surprisingly wide range of functions [6], it realises only one specific *recursion scheme*, i.e. it does not encompass all possible ways of performing recursion.

In our article [1] we describe a generalisation of this familiar *fold* on lists to all *inductively defined data types*. We use the resulting recursion schemes as a basis for inducing several widely-studied functions using stochastic heuristic search. This approach can automatically derive recursion schemes from the datatype declaration and produces programs that are guaranteed to issue valid recursive calls. When applied to a range of benchmarks, it robustly produces recursive programs that pass all tests and generalise well, and does so in a significantly smaller number of evaluations than the current state-of-the-art method (which we consider to be CTGGP, the call-tree-guided genetic programming heuristic proposed by Alexander and Zacher [3]).

2 METHOD

Our program synthesis heuristic uses algebraic data types and structural recursion to constrain the space of candidate solutions and to cope with the brittleness of recursion. Similarly to standard GP, the method learns inductively and thus requires *fitness cases* (tests), each of which is an input-output sample from the target function to be synthesized. The design of the method is dictated by the catamorphism recursion scheme (or any other general recursion scheme), which is essentially a list of non-recursive functions, each meant to handle one of many pattern-matching cases. We thus perform synthesis of the complete catamorphism-based implementation in two phases that follow.

In the first stage, the heuristic determines *case expressions* to be used to match against the arguments of the synthesized function. For recursive data types, the procedure requires a somewhat technical category-theoretic construction [7, 9], but can be automated nonetheless. In the second stage, once case expressions have been determined, the heuristic synthesizes a function for each case. These case-specific callback functions are supplied as arguments to the corresponding recursion scheme, which then represents a candidate solution that is evaluated on the fitness cases. The individual

case functions synthesized in this second phase are themselves *not* recursive, as the entirety of the recursion required for solving the synthesis task is captured by the underlying recursion scheme. Therefore, we could in principle synthesize the case callbacks using any type-aware variant of GP, or any other method capable of synthesizing programs from input-output examples.

To demonstrate the usefulness of our approach in real-world settings and for a fully-fledged programming language, we use ContainAnt [8], an online algorithm configurator/optimiser that can optimise any measurable property of code, given a set of components defined by a context-free grammar.

To search the space of such solutions, ContainAnt implements a range of strongly-typed metaheuristic search algorithms that guarantee candidate solutions to be consistent with the grammar. In this study, we employ ContainAnt's *Ant Programming* [4] heuristic, a variant of Ant-Colony Optimization in which the combinatorial structure traversed by the 'ants' is the tree of grammar productions. As a baseline, we use ContainAnt's implementation of random search, which draws each solution independently by randomly traversing grammar productions.

3 EXPERIMENTS

We compare our methods to CTGGP using its own Fib2, Fib3, OddEvens, Lucas and Pell benchmarks. We replicate all details of the CTGGP benchmark setup and use the best two out of the four configurations reported there as our baseline. These are grammatical evolution (referred to as plain grammatical evolution in [3]; **GE** in the following) and CTGGP combined with Scaffolding (**CTGGP** in the following). As an additional reference point, we also solve the benchmarks using PushGP [13] (**PushGP** in the following), the arguably most popular and continuously developed variant of stack-based GP. Our method employs two heuristics, ant programming (**Cata-AP**) and, as a further baseline, random search (**Cata-RS**). In both cases, we rely on the implementations of the ContainAnt library. The grammar defining the search space is identical for each algorithm, and is automatically extracted from source code by ContainAnt via reflective analysis of the corresponding class definitions.

Table 1: Experimental results. Results for GE and CTGGP from [3].

Benchmark	Mean number of evaluations				
	GE	CTGGP	PushGP	Cata-RS	Cata-AP
Fib2	53168	1081	288800	449	**418**
Fib3	117875	10347	278140	10301	**5722**
Lucas	105663	1622	275780	1116	**699**
OddEvens	539	255	14480	81	**26**
Pell	56240	1879	300000	1827	**544**

The results, partially showcased in Table 1, are unanimous: Cata-AP and Cata-RS synthesize optimal recursive programs in each run, and systematically use a lower number of evaluations than GE, CTGGP, and PushGP. Strikingly, this holds not only for the quite sophisticated ant programming heuristic, but even for random search, a memory-less trial-and-error. We find the superiority to CTGGP particularly important, as that method has been designed specifically with recursive programs in mind, and is state-of-the art in metaheuristic synthesis of recursive programs from examples, while Plain GE and PushGP are generic frameworks not meant to address this aspect natively. The main conceptual upshot of these findings is that the space of recursive programs of practical relevance turns out to be much smaller than widely assumed, as long as it is approached in a principled formalism, to the extent that even random search can synthesize them efficiently. In a broader context, it is likely that making more intense use of the conceptual framework offered by recursion scheme formalisms can help address other challenges inherent in program synthesis.

REFERENCES

[1] Jerry Swan, Krzysztof Krawiec, and Zoltan A Kocsis. 2019. Stochastic synthesis of recursive functions made easy with bananas, lenses, envelopes and barbed wire. *Genetic Programming and Evolvable Machines* (3 2019), 1–24. https://doi.org/10.1007/s10710-019-09347-3 Direct link: https://rdcu.be/bq8rF.

[2] Alexandros Agapitos and Simon M. Lucas. 2006. *Learning Recursive Functions with Object Oriented Genetic Programming.* Springer, Berlin, Heidelberg, 166–177. https://doi.org/10.1007/11729976_15

[3] Brad Alexander, Connie Pyromallis, George Lorenzetti, and Brad Zacher. 2016. *Using Scaffolding with Partial Call-Trees to Improve Search.* Springer International Publishing, Cham, 324–334. https://doi.org/10.1007/978-3-319-45823-6_30

[4] Mariusz Boryczka. 2002. *Ant Colony Programming for Approximation Problems.* Physica-Verlag HD, Heidelberg, 147–156. https://doi.org/10.1007/978-3-7908-1777-5_15

[5] Chris Clack and Tina Yu. 1997. *Performance enhanced genetic programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 85–100. https://doi.org/10.1007/BFb0014803

[6] Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.* 9, 4 (July 1999), 355–372. https://doi.org/10.1017/S0956796899003500

[7] Zoltan A. Kocsis and Jerry Swan. 2014. Asymptotic Genetic Improvement Programming via Type Functors and Catamorphisms. In *Semantic Methods in Genetic Programming*, Colin Johnson, Krzysztof Krawiec, Alberto Moraglio, and Michael O'Neill (Eds.). Ljubljana, Slovenia.

[8] Zoltan A. Kocsis and Jerry Swan. 2017. Dependency Injection for Programming by Optimization. *ArXiv e-print* (July 2017). arXiv:cs.AI/1707.04016

[9] Zoltan A. Kocsis and Jerry Swan. 2017. Genetic Programming + Proof Search = Automatic Improvement. *Journal of Automated Reasoning* (Mar 2017). https://doi.org/10.1007/s10817-017-9409-5

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

[11] Tessa Phillips, Mengjie Zhang, and Bing Xue. 2017. Genetic programming for solving common and domain-independent generic recursive problems. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, Jose A. Lozano (Ed.). IEEE, Donostia, San Sebastian, Spain, 1279–1286. https://doi.org/doi:10.1109/CEC.2017.7969452

[12] Shinichi Shirakawa and Tomoharu Nagao. 2010. *Graph Structured Program Evolution: Evolution of Loop Structures.* Springer US, Boston, MA, 177–194. https://doi.org/10.1007/978-1-4419-1626-6_11

[13] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40. https://doi.org/doi:10.1023/A:1014538503543

[14] P. A. Whigham and R. I. McKay. 1995. Genetic approaches to learning recursive relations. In *Progress in Evolutionary Computation*, Xin Yao (Ed.). Lecture Notes in Artificial Intelligence, Vol. 956. Springer-Verlag, 17–27. https://doi.org/doi:10.1007/3-540-60154-6_44

[15] Tina Yu. 1999. Structure Abstraction and Genetic Programming. In *Proceedings of the Congress on Evolutionary Computation*, Peter J. Angeline, Zbyszek Michalewicz, et al. (Eds.), Vol. 1. IEEE Press, Mayflower Hotel, Washington D.C., USA, 652–659. https://doi.org/doi:10.1109/CEC.1999.781995

[16] Tina Yu. 2005. A Higher-Order Function Approach to Evolve Recursive Programs. In *Genetic Programming Theory and Practice III*, Tina Yu, Rick L. Riolo, and Bill Worzel (Eds.). Genetic Programming, Vol. 9. Springer, Ann Arbor, Chapter 7, 93–108. https://doi.org/doi:10.1007/0-387-28111-8_7

[17] Tina Yu and Chris Clack. 1998. Recursion, Lambda Abstractions and Genetic Programming. In *Genetic Programming 1998*, John R. Koza, Wolfgang Banzhaf, et al. (Eds.). Morgan Kaufmann, Wisconsin, USA, 422–431.