Herb.jl: A Fast and Efficient Program Synthesis Library

Tilman Hinnerichs[1][0000−0003−4155−0239], Reuben Gardos Reid[1][0000−0002−3491−9686], Pamela Wochner[1][0000−0003−4066−8614], Issa Hanou[1][0000−0001−5873−5139], Sebastijan Dumancic[1][0000−0003−0915−8034]

[1]Technische Universiteit Delft, Delft, Netherlands
{t.r.hinnerichs, r.j.gardosreid, p.wochner, i.k.hanou, s.dumancic}@tudelft.nl

Abstract. Herb.jl is a modular library for program synthesis written in Julia. It decomposes synthesizers into reusable components and provides standardized benchmarks, enabling rapid prototyping, fair comparison, and easier combination of ideas.

Keywords: Program Synthesis · Automated Programming · Neuro-Symbolic AI.

1 Introduction

Wouldn't it be great if computers programmed themselves? Program synthesis [7] aims to do exactly that — given a description of the intended behavior and a target language, it generates a program with the desired behavior. It has exciting applications in areas ranging from automated software development to robotics [9, 10, 6].

Despite decades of progress, program synthesis research faces recurring issues common to research software: implementations are often domain-specific, difficult to adapt, and hard to compare fairly [4, 8, 12, 5]. Further, benchmarks and methods are inconsistently defined, making reproducibility and evaluation tricky. These obstacles slow down research and progress within the field and force unnecessary re-implementation of ideas.

We present Herb.jl[1], a unifying program synthesis library written in Julia that tackles these issues head-on. Herb.jl decomposes synthesizers into modular, reusable components, allowing researchers to easily re-implement old and novel ideas, and fairly compare approaches. We provide a standardized but easily extensible formulation for all core concepts in program synthesis, i.e., grammars, specifications, interpreters, constraints, and search methods. Herb.jl also includes a collection of human-readable benchmarks and ready-to-use implementations of common synthesizers, giving a solid starting point to enable rapid prototyping of novel synthesis methods.

[1] https://herb-ai.github.io/

2 Preliminaries

Herb.jl aims to make formulating and solving a program synthesis problem as easy as possible. Here, we use Herb.jl to introduce a simple program synthesis problem, defined by a specification of the intended behavior of the program in the form of input/output examples

```
problem = Problem([
  IOExample(Dict(:x => 0), 1), # when input x = 0, output 1
  IOExample(Dict(:x => 1), 3), # when x = 1, output 3
  IOExample(Dict(:x => 3), 7)
])
```

and a target language in the form of a context-free grammar

```
grammar = @cfgrammar begin
  Number = |(1:2)           # A `Number` can be 1 or 2
  Number = x                # or an input `x`
  Number = Number + Number # or the addition of two `Number`s
end
```

3 Overview and Design

Herb.jl is organized into Julia subpackages, each handling a specific aspect of synthesis—from defining specifications and grammars (`HerbSpecification`, `HerbGrammar`), to interpreting semantics (`HerbInterpret`), enforcing constraints (`HerbConstraints`), and searching program spaces (`HerbSearch`). Standard interfaces make these components interchangeable: users can swap grammars, interpreters, or search strategies with minimal changes. Additional modules include `HerbCore` for shared interfaces, and `HerbBenchmarks` for standardized problem sets.

A key design principle is the explicit separation of syntax and semantics, allowing program enumeration to operate purely on abstract syntax trees before execution. We implement two families of search methods: top-down and bottom-up search. Here, the enumeration order is customizable, making it straightforward to implement search strategies such as depth-first [7], but also state-of-the-art enumeration, such as cost-based search [2, 11, 3], or genetic search [7, 6, 13]. To illustrate Herb.jl's flexibility, the companion `Garden.jl` repository includes clean, performant implementations of existing synthesizers built directly from Herb.jl's building blocks [1, 3].

4 Conclusion and Future Work

Although Herb.jl already provides modular components that allow us to implement many approaches, we are continually expanding its toolkit to capture an ever-wider range of ideas.

References

1. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 319–336 (2017). https://doi.org/10.1007/978-3-662-54577-5_18, https://doi.org/10.1007/978-3-662-54577-5_18
2. Ameen, S., Lelis, L.H.S.: Program synthesis with best-first bottom-up search. J. Artif. Intell. Res. **77**, 1275–1310 (2023). https://doi.org/10.1613/JAIR.1.14394, https://doi.org/10.1613/jair.1.14394
3. Barke, S., Peleg, H., Polikarpova, N.: Just-in-time learning for bottom-up enumerative synthesis. Proc. ACM Program. Lang. **4**(OOPSLA), 227:1–227:29 (2020). https://doi.org/10.1145/3428295, https://doi.org/10.1145/3428295
4. Chen, A., Wong, C., Sharif, B., Peruma, A.: Exploring code comprehension in scientific programming: Preliminary insights from research scientists. In: 33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025. pp. 350–354. IEEE (2025). https://doi.org/10.1109/ICPC66645.2025.00043, https://doi.org/10.1109/ICPC66645.2025.00043
5. Cropper, A., Dumancic, S., Evans, R., Muggleton, S.H.: Inductive logic programming at 30. Mach. Learn. **111**(1), 147–172 (2022). https://doi.org/10.1007/S10994-021-06089-1, https://doi.org/10.1007/s10994-021-06089-1
6. Gulwani, S.: Dimensions in program synthesis. In: Bloem, R., Sharygina, N. (eds.) Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23. p. 1. IEEE (2010), https://ieeexplore.ieee.org/document/5770924/
7. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1-2), 1–119 (2017). https://doi.org/10.1561/2500000010, https://doi.org/10.1561/2500000010
8. Kanewala, U., Bieman, J.M.: Testing scientific software: A systematic literature review. Inf. Softw. Technol. **56**(10), 1219–1232 (2014). https://doi.org/10.1016/J.INFSOF.2014.05.006, https://doi.org/10.1016/j.infsof.2014.05.006
9. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. Communications of the ACM **55**(2), 103–111 (2012)
10. Kuniyoshi, Y., Inaba, M., Inoue, H.: Learning by watching: Extracting reusable task knowledge from visual observation of human performance. IEEE transactions on robotics and automation **10**(6), 799–822 (1994)
11. Matricon, T., Fijalkow, N., Lagarde, G.: Eco search: A no-delay best-first search algorithm for program synthesis. In: Walsh, T., Shah, J., Kolter, Z. (eds.) AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA. pp. 19432–19439. AAAI Press (2025). https://doi.org/10.1609/AAAI.V39I18.34139, https://doi.org/10.1609/aaai.v39i18.34139
12. Muggleton, S.H., Raedt, L.D., Poole, D., Bratko, I., Flach, P.A., Inoue, K., Srinivasan, A.: ILP turns 20 - biography and future challenges. Mach. Learn. **86**(1), 3–23 (2012). https://doi.org/10.1007/S10994-011-5259-2, https://doi.org/10.1007/s10994-011-5259-2

13. Sobania, D., Schweim, D., Rothlauf, F.: A comprehensive survey on program synthesis with evolutionary algorithms. IEEE Trans. Evol. Comput. **27**(1), 82–97 (2023). https://doi.org/10.1109/TEVC.2022.3162324, https://doi.org/10.1109/TEVC.2022.3162324