Utilizing Frequently Occurring Subprograms in Program Synthesis

Jelle Römer[1]

[1]Delft University of Technology

Introduction

Program synthesis generates programs attempting to solve a **problem**, which is a set of pairs of *program inputs* and their *desired outputs*, i.e. test cases. It does this using a **grammar**, a set of building blocks from which programs are created.

I explore two-phase program synthesis: alternating between ”waking” (generating candidate programs) and ”sleeping” (discovering frequently occurring subprograms to refactor the grammar).

- Programs with partial success likely share useful subprograms
- Adding discovered patterns to grammar reduces program size/depth
- Inspired by DreamCoder's wake-sleep approach [1]

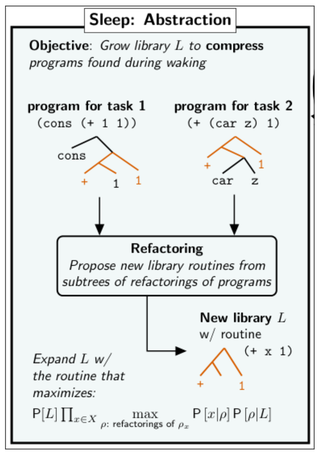


Figure 1. Visualization of refactoring from DreamCoder [1]

Methodology

Each loop contains a wake phase and a sleep phase. Its key components:

- `iterator`: decides the order of synthesized programs, in my case Breadth First Search with a maximum depth of 10 for the program's abstract syntax trees.
- `selector`: only keeps programs where more than 20% of test cases pass.
- `updater`: using previously selected programs, finds the most frequently occurring subprogram and inserts it into the grammar using a clingo model [2].

Methodology

Using that loop, this is how each problem is approached:

- Run the iterator without sleep phases, measure how many iterations that requires.
- Based on that amount, intersperse 5, 10, 15, and 20 evenly-distributed sleep phases.

In other words, if it took 1000 iterations to find the solution without any sleep phases, the sleep phases are introduced every:

- 1000 / 5 = 200 iterations
- 1000 / 10 = 100 iterations
- 1000 / 15 = 66 iterations
- 1000 / 20 = 50 iterations

When running with sleep phases, every test case is attempted to see what percentage of them pass. When running without sleep phases, *short circuiting* is turned on, which means each candidate program is discarded as soon as one test case fails, because partial solutions are not used.

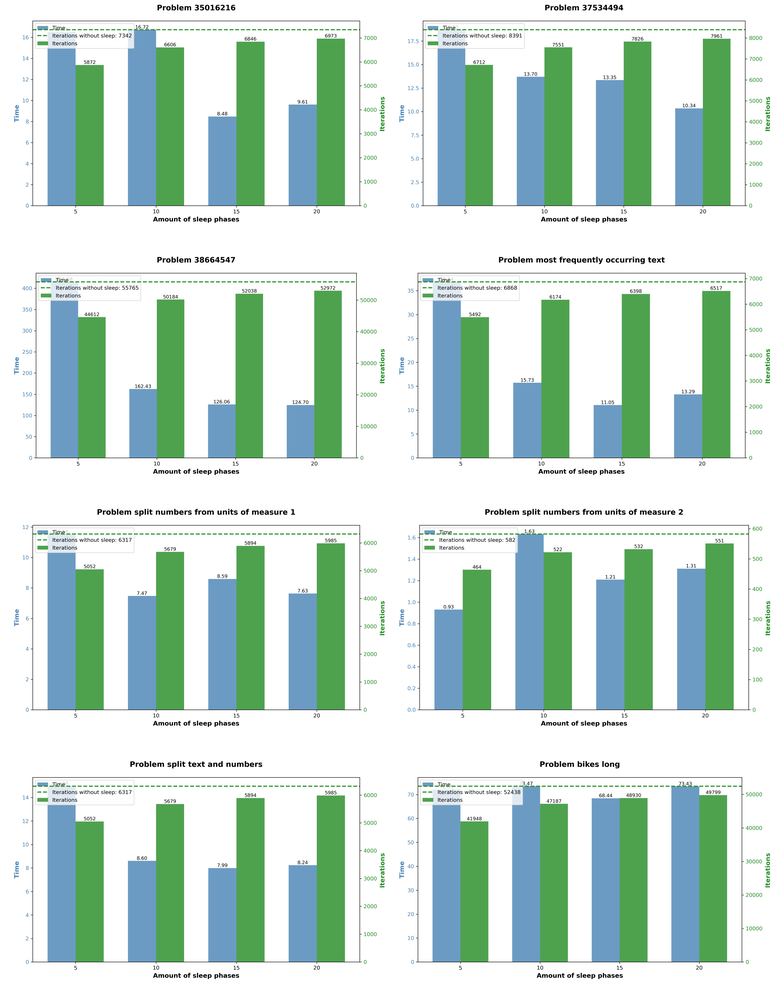
Conclusions

Iterations: The more sleep phases, the more iterations needed.
Time: Sweet spot usually at 15 or 20 sleep phases.
Trade-off: Fewer sleep phases means longer per-phase analysis time (which is why 5 is often the slowest).

Further research

- Test other iterators.
- Vary iterator depths.
- **Try uneven sleep phase distribution.** I evenly distributed the sleep phases. Perhaps more sleep phases early on -> more subprograms early on -> they have more influence on the total search. Perhaps more sleep phases later on -> there are more high-scoring programs to analyze -> better subprograms.
- **Detect functional equivalence to avoid redundancy.**

Results



References

[1] Kevin Ellis, L. Morales, Armando Solar-Lezama, and Joshua Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.

[2] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo, 2017.

[3] Saswat Padhi, Udupa Abhishek, Andi Fu, Elizabeth Polgreen, and Andrew Reynolds. Benchmarks for sygus competition. `https://github.com/SyGuS-Org/benchmarks`, 2019.