Andrew Head on *Parallelizing Program Synthesis for the Desktop*

**About Me**

I'm [Andrew Head](#), a second-year PhD student from EECS. I study at the [Berkeley Institute of Design](#). Through my research, I design and evaluate user interfaces and algorithms to generate programming help. To do this, I draw on design techniques from human-computer interaction and an understanding of programmers' cognitive needs and development practices. Broadly, my research interests intersect human-computer interaction, programming languages, and software engineering.

**What engineering problem do I want to help solve?**

I'm interested in speeding up program synthesis on desktop computers. Program synthesis is the process of generating code that satisfies a human-written specification. The study of this topic has gained some popularity in the last ten years. Recently, synthesis techniques have started to take advantage of highly-optimized formal verification methods, which has led to dramatic performance improvements.

Since program synthesis is cheaper to perform, it is possible to use it select, new applications. Synthesis can be employed to help end users write data extraction scripts by just pointing out example data (Mayer et al. 2015). It can also be used to provide tailored help to programmers working on auto-graded assignments (Singh et al. 2013). That's where this relates back to my research. *I'm interested in using program synthesis to help aid programmers get adaptive help and write code faster. Before this can happen, program synthesis still has to be made much faster.*

**How well do synthesis strategies perform today?**

From outside the programming languages community, the performance of contemporary program synthesis strategies doesn't look amazing. Paraphrasing a review by Akiba et al. (2003), modern synthesis techniques can produce:

- bit-vector manipulation programs up to size 16 in 45 minutes (Gulwani et al. 2011)
- up to 9 vector instructions in 0.12 seconds (Barthe et al. 2013)
- 18 instructions with more than an hour of search time (Eldib & Wang 2013)
- up to 15 instructions in at most 15 minutes (Udupa et al. 2013)
up to 15 instructions in at most 1000 seconds (Alur et al. 2013)

These results are not yet standardized: authors don't always report the hardware on which they test program synthesis, or attempt to synthesize from the same specifications. Alur et al. (2013) proposed a set of benchmarks to evaluating the synthesis performance. They focus on synthesizing bit-vector manipulation and linear arithmetic routines. Alur et al. piloted the benchmarks on three approaches to synthesis, implemented without substantial fine-tuning. The graph below compares the time to solution for the three methods (Enumerative, Stochastic, and Symbolic) on the three classes of benchmarks:

The "Boolean" problems seem to be hard for all three strategies: nearly all strategies take 2-5 minutes to synthesize the last three parity operations and both Morton operations. While the enumerative method performs the best in general, it takes longer than half a second for four out of the six integer benchmarks, and two minutes or longer for three out of the six integer benchmarks. This is still far away from interactive performance. To be practically useful in real-time programming practice, program synthesis techniques should be able to generate non-trivial programs such as those from Alur et al.'s linear arithmetic "integer benchmarks." They should be able to do this in much less than one second.

What are the techniques of successful implementations? What haven't they done yet?

In 2013, Moskal and Swamy authored a paper that documented an ACM programming contest on program synthesis (Akiba et al. 2013). The winning team produced λBV programs of 51 symbols in under five minutes. This performance, when compared to academic work, was very good.

The winning team, Unagi, employed several strategies for synthesis that essentially allowed
**very efficient parallelized exhaustive search.** When building candidate programs, they pruned the search space of expressions that belonged to the same equivalence class. After exhaustive program generation, several different search strategies were run in parallel. Different strategies ran across 32 threads on the EC2 cloud computing platform. Certain areas of the search space were visited before others based on the expected format of the programs. Search followed a simulated annealing approach: partial programs were ranked based on which ones satisfied the most input/output pairs of sampled behavior, and these rankings were used to inform the on-going search. Over the course of the competition, Unagi utilized over 3,000 hours of cloud computing time.

Unagi's work demonstrates the promise of program synthesis for generating complex programs through rapid inspection of a large search space. Their success depended on cloud computing and parallelizing search using several distinct strategies concurrently. I want to adapt Unagi's approaches to performing synthesis on the Akiba et al. problem set, using the hardware available on a desktop computer. In the near future, development tools may need to synthesize programs locally if they want to produce synthetic programs at no charge. It remains to be seen how to perform efficient exhaustive program generation and memoized, concurrent search on a multicore desktop computer with limited memory and storage. While such a solution will almost certainly be slower than Unagi's, it may produce an acceptable solution to begin evaluating program synthesis in modern development tools.

**References**


