Gauging Research in Program Synthesis with a Massive Open Online Contest

Takuya Akiba[1] Kentaro Imajo[2] Hiroaki Iwami[2] Yoichi Iwata[1] Toshiki Kataoka[1] Naohiro Takahashi[3,4] Michał Moskal[5] Nikhil Swamy[5]

University of Tokyo[1] Google[2] Keio University[3] AtCoder[4] Microsoft Research[5]

icfpc-unagi@googlegroups.com, {michal.moskal,nswamy}@microsoft.com

Abstract

Program synthesis from input/output examples has received much attention in the recent literature. Its success can be explained in large part by significant strides made in automated theorem proving technologies, particularly in the development of satisfiability solvers. In an effort to understand the scalability of this style of program synthesis, we ran a three-day programming contest featuring thousands of expert programmers. The results of the contest were astounding: the winning teams produced program synthesizers that exceeded the performance of synthesis techniques reported in several recent research papers. In one sense, this is a "reality check" for program synthesis researchers. In another, this is cause for great optimism—program synthesis from examples has the potential to scale well beyond what we thought was possible, if only we are willing to fine tune and employ strategies that were previously thought infeasible.

A note from the authors: The first six authors are all members of "Unagi—The Synthesis", the team that won the ICFP programming contest 2013. When describing their role in the contest, we refer to them collectively as "Unagi". The last two authors organized the contest and are referred to using the first-person plural. All the authors stand by the content of this report.

1. Introduction

A resurgence in program synthesis Automated program synthesis has always been alluring. Manually developing software is expensive and error prone, so programming computers to write programs instead is an attractive prospect. An active topic of research since the 1970s and '80s, program synthesis is a diverse field. A sampling of some recent successes of program synthesis include works like Spiral [23], which generates low-level code from a high-level digital signal processing transformation; Autobayes [9], which synthesizes code to learn the free parameters of a probabilistic model based on the observed data; GenProg [11], which uses genetic programming [22] to automatically fix defects in a program; and Sketch [24], a tool which allows a programmer to develop code with "holes" which are automatically filled by a program synthesis tool. Program synthesis also has applications in other fields, such as in biology. For example, Koksal et al. [21] show how to synthesize executable models of cell fate development in *C. elegans* from experimental data about that earthworm's cells.

On particular sub-area of program synthesis that has caught our attention is a technique we call *example-driven* (or ED) synthesis. ED-synthesizers generate programs from a few examples of the desired program's input/output behavior. When applicable, ED-synthesis is attractive since it provides a natural way for a user to interact with the synthesis tool. As such, ED-synthesis has seen significant adoption, notably in the widely publicized "Flash Fill" automatic scripting feature of Microsoft Excel (see http://research.microsoft.com/~sumitg/flashfill.html). Several recent papers in this area have also been well-received at the top ACM programming languages conferences [10, 15, 27]. In the last few months, two papers exploring ED-synthesis have even received "best-paper" awards [4, 8] and another has been featured as a "research highlight" in the Communications of the ACM [14].

The success of recent work on ED-synthesis can be explained in part by advances in automated symbolic reasoning, particularly in satisfiability-modulo-theories (SMT) solvers. As a case in point, consider Gulwani et al.'s 2011 work [15], which develops a tool called Brahma that can synthesize sequences of tricky "bit-twiddling" instructions from user-provided input/output examples. For example, to turn off the rightmost 1-bit in a bit-vector, the user may provide to Brahma the following set of examples: `{01100 -> 01000, 00000 -> 00000}`. Brahma responds with a program such as `f(x) = x & (x - 1)`, a function that agrees with the user's examples. To effectively search the space of programs that matches the user's specification, Brahma cleverly encodes the synthesis problem as a first-order logic formula which can be effectively solved by an SMT solver like Z3 [6]. When successful, Z3 provides a model for the formula from which Brahma can read off a solution to the original synthesis problem.

Brahma is able to synthesize programs that contain sequences of up to 16 instructions, where each instruction is drawn from a set of around a dozen bit-manipulating primitive operations. Brahma's running time on these examples range from 1 second to 45 minutes. In comparison, two other non-SMT-based tools that Gulwani et al. compare against fail to synthesize any program longer than 6 instructions long within their timeout of 1 hour. As such, Brahma represents an enormous stride over the prior work. Quoting the paper: *"The winning advantage comes from the fact that we ride upon the recent engineering advances made in SMT solving technology [...] as opposed to explicitly performing an exhaustive enumeration over an exponential search space"*.

Following Brahma, all the other papers cited earlier ([4, 8, 10, 15, 27] and several others besides) make use of SMT solvers to effectively solve synthesis problems that were out of reach previously. Impressed by the work of our colleagues (full disclosure: Gulwani and the authors of Z3 are our colleagues at Microsoft Research), we got to wondering: is this as good as it gets? Synthesizing 16 straight-line instructions in under an hour is undoubtedly impressive, but typical human-authored functions are significantly larger. How far can ED-synthesis go?

The experience of various solver competitions suggests that a very good way to answer such a question is through a contest. Over the years, various competitions have been used to accelerate research on solving techniques and tools for various hard problems—first-order theorem proving (the CASC competition started in 1996 [25]), Boolean satisfiability solving (SAT Competition started in 2002 [18]), satisfiability modulo theories (SMT-COMP started in 2005 [3]), and more recently model checking [5] and software verification [16]. Such competitions are widely regarded as successful in raising the profile of the sub-field as well as improving quality, performance, availability, and interoperability (via a common input format) of solvers. The typical setup involves researchers being asked to submit their tools, which are then run on a number of benchmark problems. The performance of solvers is compared according to (very detailed) rules set in advance and the winner (usually one per category of problems) is announced, given some small prize and, most importantly, bragging rights. Further afield, contests such as the DARPA Grand Challenge, the ImageNet Challenge, and the NetFlix Prize have spurred research in areas such as robotics, image recognition and machine learning.

A golden opportunity to do something similar for program synthesis came by way of the ACM International Conference in Functional Programming's Programming Contest (ICFP-PC). Held annually since 1998, the ICFP-PC features teams of 1–20 programmers working for 72 hours to complete a challenging programming task using programming languages and tools of their choosing. Over the years, this contest has gained a reputation of identifying the best hackers in the world, and in recent years has regularly featured more 1000 programmers (self-organized into around 300 teams) competing for the top prize—bragging rights occasionally accompanied by a modest cash prize funded by the ACM. That's many thousands of hours of expert-programmer time, an enormous resource that the contest organizers must put to use each year, mostly for recreation but occasionally also for scientific value.

The contestants of the 2013 ICFP Programming Contest were set an ED-synthesis task very like the problem addressed by tools like Brahma. Following tradition, the contest task was a secret[1] until the contest opened on August 8, 2013. Our goal was to compare what an army of expert programmers could achieve in 72 hours, relative to the state of the art in the research literature.

The results of the competition bowled us over. The best team (and others in the top 5) automatically synthesized programs that contained up to 51 instructions in less than 5 minutes, and they only had 72 hours to develop the synthesizer. Unlike the straight-line code over 32-bit vectors synthesized by Brahma and related tools, the contestants synthesized programs involving 64-bit vectors that also included conditionals and bounded loops; on the other hand, programs synthesized by Brahma also had more constants. While preparing for the contest, we implemented several several variations of the SMT-based synthesis algorithms like Brahma, but none of them scaled beyond 17 instructions within the 5-minute window.

The rest of this paper describes the way we set up and ran the contest, the strategies used by some of the highly ranked teams, and compares the outcomes of the contest more closely with the related work in ED-synthesis. In summary, the best teams were aware of the SMT-based program synthesis literature, but after some initial experiments, chose not to use it. Instead, *Unagi—The Synthesis*, a group of 6 Japanese programmers (currently undertaking or having recently completed their undergraduate studies) developed a custom nearly exhaustive search procedure that used a combination of offline and online search, parallelized to use 1000 hours of compute-time on the Amazon EC2 cloud and won the contest by a comfortable margin. We take away the following points from our experience:

- Talented young hackers are not to be underestimated!
- Custom-built domain-specific approaches to program synthesis problems can provide a big boost to scalability. This is not surprising. Green and Barstow make this observation as far back as 1978 [12]. Additionally, Gulwani et al.'s more recent efforts on automatic text processing [14] also employ domain-specific reasoning independent of SMT solvers.
- There is still a long way to go before we hit the ceiling of ED-synthesis. As one begins to scale up to and beyond the size reached by *Unagi* and the other contestants, it may actually be feasible in the near future to synthesize genuinely non-trivial pieces of code by example.
- The path towards scaling program synthesis may well be through cloud computing. Exhaustively enumerating and storing terabytes of program snippets is perfectly viable in the cloud, and it may be the best way to integrate serious program synthesis into the simple, syntactic code-completion features available today in program development environments (IDEs), where fast response times are crucial. Effectively parallelizing the search to use the vast computing resources that are now readily available in the cloud is also key.
- Programming contests at the scale of the ICFP-PC garner the the attention of thousands of expert programmers for a few days. This is a very valuable human resource! Finding ways to harness this resource to pose and answer scientific questions is a both an opportunity and a challenge facing our community. Turning ED-synthesis into a game was relatively easy, at least in hindsight. What other questions might we use the ICFP-PC to answer in the future?

A few disclaimers. The main purpose of this paper is to illustrate how a large, open competition can be used to gauge the current state and future potential of a research topic. We focused specifically on ED-synthesis for a few reasons: (1) it has been studied extensively in the literature; (2) with a little careful thought, it can be formulated as a game—an essential feature of a competition; (3) many instances of the game can itself be synthesized—essential for a massive competition. We make no independent claims about the scope of ED-synthesis in general, the practical usefulness of contest problems, or what conclusions one might draw about the diverse field of program synthesis from the results of the competition. While we do give descriptions of the some of the techniques used by the winning teams, a precise description and evaluation of these algorithms is beyond the scope (and page limits) of this paper.

2. Running the contest

Inspired by the coding duels at `http://pex4fun.com` [26], the contest was designed as a guessing game played between a game server and a player that went as follows:

Game: I have a secret program `A`, and I want you to guess it.

Player: Can you tell me what `A(16)`, `A(42)` and `A(128)` are?

Game: Sure, `A(16) = 17`, `A(42) = 43`, and `A(128) = 129`.

Player: Is it the program `B0`, where `B0(x) = x + 1`?

Game: No dice, `A(9) = 9`, but `B0(9) = 10`.

Player: What are `A(11)` and `A(12)` then?

Game: Since you ask so nicely: `A(11) = 11`, `A(12) = 13`.

Player: Is it the program `B1`, where
`B1(x) = if0 ((x & 1) ^ 1) then x else x + 1`

Game: That's right! You score one point.
I have a secret program `A'`, and want you to guess it.

Player: Argh!!!

[1] In an effort to also draw some of the experts in program synthesis to the competition, we dropped a hint a few weeks earlier mentioning just that *"[the] programming task will involve an element of program synthesis"*.

$$
\begin{array}{lcl}
|0| = |1| = |x| & = & 1 \\
|(\texttt{lambda}\ (x)\ e)| & = & 1 + |e| \\
|(\texttt{op}\ e_0\ \ldots\ e_n)| & = & 1 + |e_0| + \ldots + |e_n| \\
|(\texttt{if0}\ e_0\ e_1\ e_2)| & = & 1 + |e_0| + |e_1| + |e_2| \\
|(\texttt{fold}\ e_0\ e_1\ (\texttt{lambda}\ (x\ y)\ e_2))| & = & 2 + |e_0| + |e_1| + |e_2|
\end{array}
$$

Figure 1. Defining the size of a program

Such an interaction constitutes one round of the game. Of course, the player has to guess a program that is *semantically equivalent* to the secret, e.g., the program `B2` where `B2(x) = if0 ((x & 1) ^ 0) then x + 1 else x`, would also have been a correct guess, since it computes the same function as `B1`.

In this section, we summarize the rules of the game, various measures we put in place to ensure that the game was both fun and fair, and the technological challenges we had to overcome in order to stage a game of this scale.

2.1 Rules of the game

Each round of the game lasted a maximum of 5 minutes, i.e., the contestants had at most 5 minutes to guess the program. Many rounds of the game could be played in parallel, if the contestant so desired. A maximum of 1,820 rounds could be played within the 72 hours of the contest, allowing a maximum score of 1,820 points.

The game rules specified that the secret programs were all programmed in λBV, a small programming language in which to express functions over 64-bit vectors. λBV's syntax was based on S-expressions for ease of parsing. An example program is: `(lambda (x) (and x (plus x 1)))`. λBV included the following primitive operators (all bit-wise): negation (`not`); shift left by one bit (`shl1`); shift right by one (`shr1`), four (`shr4`), or sixteen bits (`shr16`); conjunction (`and`); disjunction (`or`); exclusive or (`xor`); and addition (`plus`). The only constants in the language were 0 and 1. Additionally, the language included a conditional form to branch after testing whether a value is 0 (`if0`), and a `fold` operator to loop over each byte of a bit-vector from right to left with an accumulator. The semantics of λBV programs is quite straightforward and was not formally specified—contestants could experiment with the game server to confirm that their interpretation of the semantics matched ours.

The 1,820 rounds of the game were classified into five broad classes. When the contest began, only the following three classes were made known—the remaining two were presented as bonus problems later (cf. §2.4).

1. **Fold-free problems** Anticipating that synthesizing programs with loops would pose a significant challenge, we restricted 560 rounds of the game by revealing to the contestant that the secret program made no use of the `fold` operator.
2. **Top-fold problems** 460 rounds revealed to the contestant that the secret program began with an occurrence of the `fold` operator, and contained no further occurrence of `fold`.
3. **General-fold problems** 400 rounds revealed to the contestant that the secret program contained one or more occurrence of `fold`, but with no further constraints.

In each case, the contestants were also told the size of the secret program (i.e., the number of sub-terms it contains, as defined in Figure 1), as well as the set of all operators in it. The fold-free problems ranged in size from 3 to 30; with 20 programs in each size class; the top-fold programs ranged in size from 8 to 30 with 20 programs in each size class; while the general-fold problems ranged in size from 11 to 30, also with 20 in each size class. In total, these three classes represented 1,420 rounds of the game.

As an illustration, if a secret program in the fold-free class was `(lambda (x) (and x (plus x 1)))`, we would reveal to the contestant that the program was of size 6 and that it only contained the operators `and` and `plus`. For fold-free programs of size 3, this information was sufficient to uniquely determine the program—this was by design; we wanted to start contestants off on easy problems so that they could score a few points and be encouraged to continue participating. However, the problems increased quickly in difficulty—for programs larger than size 12, the information we revealed about the set of operators it contained was usually useless, since it typically contained all the operators in the language (aside from information about the (non-)occurrence of `fold`).

2.2 The logistics of running the game

Contestants interacted with the game through a programmatic web-interface that we designed and implemented. The first task for contestants was to implement the client side of this interface—fairly easy given support for HTTP requests and common marshalling formats available as libraries in most languages. However, ensuring that our game server stayed running throughout the duration of the contest required some effort on our part.

Deciding equivalence A key step in staging the game involves checking whether a user successfully guessed the secret program. This involves proving that the guess and secret are semantically equivalent. However, classic results from the theory of computation make it clear that deciding whether two programs are semantically equivalent is impossible, at least in the general case.

With this in mind, we carefully designed λBV with restrictions to ensure that program equivalence is decidable. In particular, λBV programs only have bounded loops that can be unrolled completely. Proving the equivalence of loop-free programs is decidable, although still not easy to do efficiently.

Our approach was to make use of Z3 to also solve the problem equivalence problem. Specifically, after unrolling all loops, we encoded λBV programs as a formula in Z3's input language, making use of its support for reasoning about bit-vectors. Given two such programs encoded as formulas, equivalence of the programs amounts to proving equivalence of the formulas. Z3 first performs a number of rewriting, and then usually resorts to "bit blasting" the formulas into logical circuits over propositional variables and uses its SAT-solving capabilities to find satisfying assignments to those variables.

SAT-solving is NP-complete, so the problem is decidable. But, the SAT instances that must be solved for deciding the equivalence of λBV programs may have upwards of a million clauses. Efficiently solving these SAT instances is a computationally intensive task that is also key to staging the game: with only a five minute window for each round of the game, and anticipating that a round would involve several guesses, we needed to ensure that validating a guess would not take more than a few seconds. From some experiments, we concluded that Z3 could effectively solve the majority of λBV equivalences in less than 20 seconds. Our experiments were validated over the course of the competition: Z3 successfully decided around 350,000 program-equivalence queries, taking 1.25 seconds on average, although it did exceed 20 seconds in around 421 cases, a success rate of 99.88%. Our servers also handled an additional 400,000 requests for training problems and evaluating secret programs on user-provided inputs.

Elastic scaling in the cloud Since validating a guess could consume as much as 20 seconds of processor time, a major concern for us organizing the game was to ensure that we had sufficient resources to efficiently handle requests from the 300+ teams that were competing. Our approach to managing this was to make use of the elastic cloud computing platform provided by Windows Azure.

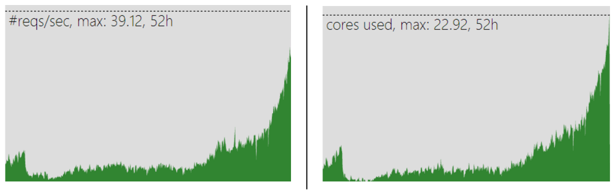


Figure 2. The graphs show the requests-per-second (left) and the number of processor cores used by our game servers (right) during the last 52 hours of the contest. Near the end of the contest, our game server was processing nearly 40 requests per second on 23 processor cores.

This involved reserving capacity for up to 128 processing cores in the cloud, with additional 128 for swapping test and production deployments. We used 4-core instances communicating via Azure table storage. Initially, we started with only 2 instances (for redundancy, 1 would be in fact enough), but as the number of requests and server load crossed pre-defined thresholds, we configured the cloud service to bring more cores online.

As it turned out, we had much more capacity than we actually needed. The graph at the left of Figure 2 show the frequency of requests we received for the last 52 hours of the competition. Requests spiked first at the 24 hour mark (the end of the "lightning division" of the contest), and then peaked at nearly 40 requests per second at the very end of the contest. The graph at the right shows the number of utilized processor cores (or rather CPU seconds per real-time second), peaking at almost 23 cores. In fact, playing it safe, we had usually 2-3 times more cores running than was needed—towards the end we had 64. Still, the final bill for the compute resources was well below 500 USD.

2.3 Balancing fairness and fun

When refereeing an academic conference, one can generally assume that the participants are acting in good faith. An informal system of academic incentives and reputations ensures that no malicious actors are actively trying to compromise the academic peer-review process. Staging a competition like the ICFP-PC is a different kettle of fish. The contest attracts many anonymous participants, the web servers are widely publicized and can be freely accessed from anywhere, and winning the contest brings enough fame (or notoriety) that someone may attempt to subvert the game. In past years of the ICFP-PC, the game server has been subject to denial-of-service attacks. Keeping things fair, while still ensuring the game was fun, required some careful design.

Pre-registration Prior to the start of the contest, teams had to register their intention to participate by creating an account on `EasyChair.org`, a popular conference management web-site. The purpose of pre-registration was left unspecified (which drew some criticism on various public forums). Once the contest began, we provided each pre-registered team with an authentication token which gave them access to the game servers. Informally, keeping the purpose of pre-registration secret helped to minimize the problem of teams pre-registering repeatedly in an attempt to accumulate multiple authentication tokens.

Throttling user requests Any well-intentioned team should spend most of the 5-minutes available in each round in searching for a solution, rather than bombarding the server with requests. Having assigned unique tokens to each team, we were able to rate-limit the requests that any team could make. Each token entitled a team to make up to 5 requests in any 20-second window and use up to 20 CPU seconds in any 60 second window. Additionally, these limits would be automatically tightened should the servers get over-loaded. This, however, did not happen during the contest.

Timeouts We gave the game server 20 seconds to check the validity of a guess. As mentioned earlier, our experiments with Z3 suggested that this timeout would be exceeded only very rarely. However, knowing that timeouts would occasionally happen, an adversarial team may attempt to craft particularly convoluted guesses designed to cause Z3 to timeout. To deter such a strategy, the game rules specified clearly that when the server exceeds the 20-second timeout, the guess is considered to be invalid and no point is scored. In the rare cases where timeouts occurred, this led to some frustration with the contestants. To mitigate these concerns, after the contest concluded, we re-ran Z3 on those timed out guesses and gave additional credit to each team if their guesses were in fact correct. This had no impact on the final ranking of teams.

Minimizing collusion Teams may have attempted to subvert the game by colluding. If all teams were assigned identical problems in each round, one team could partially solve a round, and then run out of time to complete it. They could then hand off the partial solution to another team for completion. To prevent this, each team was assigned 1,420 randomly chosen problems from a set of around 100,000 problems. The distribution of these 1,420 problems was arranged so that every team received the same number of problems in each category and size class.

Training problems Each team could also request randomly chosen training problems in a particular class and size category. In a training round, the secret program was revealed to the contestant at the start of the round. Of course, training rounds scored no points,

Problem generation Generating 100,000 secret programs is in itself not an easy problem. We followed a relatively simple approach for the first three categories of problems. We randomly generated several thousand programs in each size and class category, ensured that these programs had no dead sub-expressions in them, and then weeded out those that failed various simple tests (e.g., those whose results appeared to be constant on a few randomly chosen input points), ensuring that we retained around 1,000 programs in each size and class.

An element of chance Our simple problem generation strategy left (what we thought was) an exciting element of chance in the game. The class and size of a secret program was not necessarily a measure of the semantic complexity of the function it computed—a syntactically large secret program could be optimized to a much smaller program. This was designed to keep the problem interesting for as many teams as possible. We did not expect the majority of teams to be able to effectively win rounds where the secret function to be guessed needed more than around 12–15 terms to be computable in λBV—except for the very best teams, this turned out to be true. However, our problem generation strategy meant that average teams could still score points in the more difficult problem categories by finding relatively small solutions.

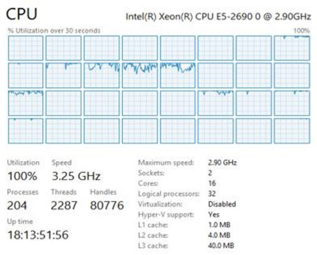
Of course, we did not want the whole contest to devolve into a game of chance. So, we also had two classes of bonus problems that were designed differently—we describe this next.

2.4 Bonus problems

It is typical for the ICFP contests to modify the rules in the middle of the contest. Maintaining this tradition, with roughly 24 hours remaining in the contest, we released 200 bonus problems; with around 18 hours to go we released an additional 200 bonus problems. Our expectation was that the winning team would be clearly identified by their performance on these bonus problems, with no element of chance. As such, each team was assigned exactly the same 400 bonus problems, but each problem was assigned a randomly chosen identifier for each team, minimizing their ability to correlate problem instances between teams and their ability to collude.

Unlike the problems in the first three classes, each bonus problem was chosen in a way that ensured that there were no simple solutions to the problem. To do this, we decided to randomly mine difficult program *nuggets*. Specifically, our approach to generating hard problems was to start by randomly generating a fold-free program P of size n. We then used Z3 to check that there exists no λBV program of size k or smaller that computes the same function as P. If the check succeeds, then we consider P a *k-nugget*, meaning that it can only be solved using a program of size k or greater. From such nuggets, more complex programs can be built. Given three nuggets, P_1, P_2, P_3, the program `(if0 (and` P_1 `1)` P_2 P_3`)` is also likely to hard to reduce further (so long as the parity of P_1's low-order bit is not constant, which is easy to prove with Z3). By repeating this process, nearly arbitrarily large programs can be built.

Proving that a program cannot be optimized to use less than, say, 10 terms is quite computationally intensive. In effect, one has to enumerate all programs up to size 9, which can number in the 10s or even 100s of millions. Doing this consumed several days of CPU time, running Z3 on all 32 cores of a workstation, searching for nuggets, as the CPU-utilization screenshot alongside shows.


CPU
Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz
% Utilization over 30 seconds
100%

Utilization 100%
Speed 3.25 GHz
Processes 204
Threads 2287
Handles 80776
Up time 18:13:51:56

```
Maximum speed:       2.90 GHz
Sockets:             2
Cores:               16
Logical processors:  32
Virtualization:      Disabled
Hyper-V support:     Yes
L1 cache:            1.0 MB
L2 cache:            4.0 MB
L3 cache:            40.0 MB
```

Our first set of 200 bonus problems were fold-free programs of size between 19 and 25, composed of 5-nuggets. The second set of 200 bonus problems were fold-free problems between size 31 and 43 and were built from 9-nuggets. We also provided training problems in these categories, so contestants could reverse engineer that these programs had a particular branching structure.

2.5 Discussion

The task that the contestants faced most closely resembles the problem of reverse engineering or deobfuscating a program—an example of this problem in the program synthesis literature is the work of Jha et al. [19]. A related problem also studied in the synthesis literature is *super-optimization*, which has the additional constraint (not imposed on our contestants) that the synthesized program be minimal in some regard, e.g., have the fewest number of instructions to carry out the required functionality. Neither reverse engineering nor super-optimization is a perfect characterization of the problems tackled by ED-synthesis researchers. However, the problem of synthesizing bit-vector programs from examples is known to be hard, and has been studied extensively in the literature. As such, the contest task was not without precedent.

The reader might also wonder about the specific problem generation strategies we used. For example, rather than generating random problems, one might have considered selecting some set of "useful" problems. However, keeping the nature of the contest in mind, useful problems are an unworkable choice, since a contestant might then be able to win the contest just by guessing some common programs, rather than actually solving the synthesis problem. Besides, it's unclear how one might devise an automated strategy for generating thousands of useful programs.

The question of generating useful problem instances is a particularly interesting direction for future research, since randomly generated problem instances may be artificially difficult. For example, in the SAT Competition there are different tracks for application, crafted, and random instances, with different solvers, using different techniques, claiming top spots. The sizes of instances in the categories are also radically different—in 2011 the smallest unsolved crafted instance had 141 variables, whereas typical application instances sported millions [18]. We hope for the creation of such a set of useful benchmarks in different categories for ED-synthesis. For what it's worth, we have contributed 320 bonus problem instances from the ICFP competition to researchers who are organizing a synthesis competition at FLOC 2014 (see http://www.sygus.org/).

3. Outcomes of the contest and winning strategies

Unagi—The Synthesis, a 6-member Japanese team, won the contest by scoring 1,696 out of a maximum of 1,820 points. The second place team, *F5 Attackers*, a 5-member Japanese team, scored 1,608. Third place was taken by *Hack the Loop*, 10 Russian programmers, who scored 1,499 points. The next 15 teams were bunched closely together, with only 100 points separating them all. As such, *Unagi* was the clear winner. We also gave a judges' prize to a team called *Kuma-* for a particularly short solution in Ruby developed while hacking an experimental garbage collector to improve performance of the language runtime. A "lightning prize" was also given to team *ITF* for having the highest score 24 hours after the contest began.

Our own colleagues who research program synthesis were excluded from the competition due to a conflict of interest. However, we were informed that other prominent researchers in the program synthesis field did participate in the competition. Their strategy was to use a combination of exhaustive enumeration and SMT solving, but they did not finish higher than rank 90.

In the remainder of this section, we focus primarily on the techniques employed by *Unagi* (as well as some of the other teams who ranked in the top 25) to win the competition. Our description is based on the following sources of information.

- A short survey that all teams were required to complete at the end of the competition.
- A more detailed questionnaire that we requested all the teams ranked in the top-25 to complete.
- The logs maintained by our game server which records the sequence of interactions with the game made by each team.
- Discussions while preparing this report among all the authors of this report, specifically addressing various details of *Unagi*'s solution.

At a high level, *Unagi* employed three strategies: offline exhaustive enumeration, heuristic online search, and a technique that we will call "stitching" conditionals. For the latter two strategies, *Unagi* deployed up to 64 variants of the heuristics in parallel, dynamically selecting the best one for each problem. We describe their strategies by recalling a few of their interactions with the game.

3.1 Offline exhaustive enumeration

- At `2013-08-11T06:40:56`, *Unagi* begins a round involving a size 17, fold-free program. They know beforehand that the secret program contains the following operators: `{xor, shr1, shr4, not, or, plus, if0}`—only `shr16` and `and` are excluded. The game server's clock for this round begins to tick—*Unagi* have five minutes to guess the program.
- At the same time, *Unagi* request that the secret program be evaluated on 256 input points. The server responds almost immediately with the output of the secret program on these input points. The provided inputs include some common bit patterns, e.g, and all powers of 2 and bit-wise negations of the powers of 2. The input set also contained randomly generated numbers, conjoined with $1, 3, 7, 15, \ldots$, as well as the bit-wise negation of those numbers. This set of inputs was fixed for all rounds.

- Around four seconds later, at 2013-08-11T06:41:00, *Unagi* guesses that the secret program is:

```
(lambda (x)
  (if0 (shr1 x)
       (or x (shr4 (not (shr1 (not 0)))))
       (xor x (shr4 (not x)))))
```

  The secret program is in fact:

```
(lambda (x)  (xor (shr4 (not
  (or (shr1 (if0 (plus (shl1 (shr1 x)) 0) (not 0) 1))
    x))) x))
```

  It takes the game server a bit more than 1 second to prove that the guess is semantically equivalent to the secret. *Unagi* synthesizes a program of size 16 in less than 4 seconds and scores another point.

How did they do it? The primary strategy used by *Unagi* was offline exhaustive enumeration of programs. Knowing the operators in the secret program and the size of that program, they were able to exhaustively enumerate all programs up that size. Various pruning strategies were used to reduce the number of programs. In each round, *Unagi* enumerated around 100 million programs offline. In rounds where the size of the program was at most 15, this enumeration was guaranteed to be exhaustive.

Size- and operator-based pruning. *Unagi* enumerated programs in increasing order of size, generating programs of size $k+1$ by adding operators to each program P of size k or less. If they were able to determine that there was no way to produce a term of the final desired size and operator set from P, then it was discarded from further consideration. For example, knowing that a secret program size of 15 contains both `or` and `if0`, a program P of size 11 that has no occurrence of `or` and `if0` cannot be a sub-term of the secret program (since adding both operators to P would increase its size by at least 5). Interestingly, knowing that the target program has size less than or equal to 15 allowed *Unagi* to exhaustively enumerate all programs up to that size. But, for larger examples, knowing that the size was, say, 30 did not provide as much information—in such cases, they were only able to exhaustively enumerate programs that were up to size 12 or 13, still numbering in the 100s of millions.

Semantic pruning rules. *Unagi* also implemented scores of rules that pruned the search space by exploiting algebraic properties of the operators to normalize the generated programs, e.g., `not` is an involution; `and` and `or` are commutative and associative; etc.. More sophisticated rewrite rules were also used to normalize the generated programs, e.g., `(not (if0 x (not y) z))` was normalized to `(if0 x y (not z))`. When enumerating programs, if any rewrite rule could be applied to a program, it was discarded.

Having enumerated programs offline, they evaluated all of them on the pre-chosen set of 256 input points. Then, after querying the game server for the outputs of the secret program, they simply responded immediately with one of the previously enumerated programs that matched the outputs.

3.2 Online heuristic search

- At 2013-08-11T17:20:51, *Unagi* attempts a top-fold problem of size 23. They know that the secret program does not contain any occurrence of `and`, `shr16` or `not`. As usual, they request the output of the secret program on their pre-chosen 256 inputs.
- 4 minutes and 53 seconds later, they guess the following program of size 14, and score one point (it takes us 1.2 seconds to check the guess).

```
(lambda (x)
  (fold x 0
    (lambda (y z)
       (shl1 (or z (if0 (shr4 (shl1 y)) 0 1))))))
```

For larger problems, size- and operator-based pruning is not effective. So, *Unagi* also used various heuristics to optimize an online search, i.e., searching for nearly the full five minutes to find a matching program.

Output equivalence classes. One incomplete but apparently effective heuristic involved treating all programs that produce the same results on the 256 inputs to be in the same equivalence class. When generating a larger program from some already enumerated smaller programs, only one element of each equivalence class of the smaller programs was chosen. This may lead to incompleteness, since programs in an equivalence class may not truly be functionally equivalent.

Syntactic heuristics. By studying the training problems, *Unagi* learned that the secret programs often had a particular structure, e.g., the expression trees are biased towards being unbalanced. So, they aimed to enumerate unbalanced trees first. We think it is unlikely that this particular heuristic was very effective, however in other cases syntactic heuristics were essential to solving the problem. For example, the class of top-fold problems inherently restricts the search space to those programs that begin with a `fold` operator. By examining the training problems, *Unagi* were able to discover other valuable hints, e.g., that the initial accumulator in top-fold problems was always zero.

3.3 Stitching conditionals

- At 2013-08-11T22:28:33, *Unagi* attempts a bonus problem of size 43 and evaluate the secret program on their favorite 256 inputs.
- Over the course of the next 3:45 minutes, they guess incorrectly 5 times, each time obtaining a counterexample from the game server. Finally, they guess correctly with the program below (of size 51!):

```
(lambda (x)
  (if0 (and 1 (shr1 x))
       (if0 (shr1 (shr4 x))
            (if0 (and 1 (shr1 (shr1 x)))
                 (or 1 (not x))
                 (if0 (and x 1) 0 1))
            (if0 (shr16 (not x))
                 (not (xor x 1))
                 (or 1 (not x))))
       (if0 (and x 1) 0
            (if0 (not (or x (shr4 x))) 0 1))))
```

  Each incorrect guess leads them closer to the final answer. For example, their last incorrect guess is shown below: it only differs from the final answer in the guards and arrangement of some of the conditionals.

```
(lambda (x)
   (if0 (and 1 (shr1 x))
        (if0 (and x (shr1 (shr1 x)))
             (or 1 (not x))
             (if0 (shr1 (shr4 x))
                  (if0 (and x 1) 0 1)
                  (if0 (shr16 (not x))
                       (not (xor x 1))
                       (or 1 (not x)))))
        (if0 (and x 1) 0
             (if0 (not (or x (shr4 x))) 0 1))))
```

This interaction illustrates one of *Unagi*'s major online search strategies. Rather than trying to find a single large program that matches the input/output relation, *Unagi* searched for a set of smaller programs that together covered the relation. Having found this set, they aimed to find other small terms that would serve as the switching logic—we call this technique *stitching*. When combined with the other techniques, this often allowed *Unagi* to find solutions to extremely large problems in a structured way. From our inspection of the logs, it seems evident that stitching was a key technique that enabled solving many of the larger problems, especially the bonus problems.

Semi-directed search strategies. Underlying *Unagi*'s stitching algorithm lay a variety of semi-directed search strategies based on simulated annealing and hill-climbing. Indeed, their interactions with the game server indicate that they often arrived at the solution gradually—starting with a term that was almost functionally equivalent to the secret and then making small changes to it syntactically until they arrived at the solution—as is evident in the difference between that last incorrect guess and the final solution in the interaction above. *Unagi* simply used the number of input/output pairs that were properly classified as a metric to rank partial solutions.

3.4 Parallel deployment of heuristics in the cloud

In just three days, *Unagi* had developed a wide range of sophisticated heuristics, each tunable by various parameters, with no single heuristic dominating the others. The final element that made *Unagi* victorious was their effective use of parallelism and large-scale compute power.

Unagi deployed their solution on Amazon's EC2 cloud. Their solution consisted of many strategies and heuristics running simultaneously for each round. They played up to four rounds in parallel, with 32 threads running independent strategies in each round. They even went to the extent of using the Tokyo location of EC2 for development, while deploying their contest-ready solution in the Richmond, Virginia data center of EC2 for proximity to our own Windows Azure servers in the Northeast United States. Over the course of the contest, *Unagi* estimated using around 3000 hours of compute time on EC2, which cost them less than 80 USD (easily covered by the 1000 USD prize money that they later received from ACM SIGPLAN).

Unagi programmed in 7 different languages, reflecting the preferences of the various team members. 1000 lines of Java in 10 variations and 1000 lines of C# in 10 variations ran the main search algorithms. 2000 lines of C++, 1400 lines of PHP, and 300 lines of Shell managed the parallel deployment and various utilities. 700 lines of Ruby and 100 lines of Haskell were used for other tools.

4. Gauging recent work on ED-synthesis

Unagi and many other contestants studied many of the recent research papers in program synthesis, including several of the papers cited here. Several teams even implemented prototype solutions using the SMT-solving strategies described in current research, but most well-placed teams abandoned the use of SMT solvers for the contest itself, relying instead on various ad hoc search strategies. Some of *Unagi*'s strategies described in §3 have also been discovered independently by other researchers. For example, the stitching technique has also been reported in several recent papers [1, 17, 20].

In this section, we briefly review the results from several recent program synthesis papers and conclude with a summary of lessons learned from the *Unagi* experience. Our treatment is necessarily brief, and focuses primarily on research in component-based, inductive program synthesis—Gulwani provides a useful survey [13].

Synthesis of loop-free programs [15]. As mentioned previously, this paper was among the first to propose using SMT solvers as an alternative to ad hoc synthesis strategies. Gulwani et al. develop a technique to encode synthesis problems as constraints in an SMT solver, given the *multi-set of operators* in the secret program to be synthesized, and a set of input/output examples. The multi-set of operators is more information than we provided to the ICFP-PC contestants. We also provided the size of the secret term, but given the multi-set of operators, one can easily recover the size. Gulwani et al. report being able to synthesize programs up to size 16 in 45 minutes—this performance is comparable to what we achieved with our test solution, which was significantly below what the winning teams managed.

From Relational Verification to SIMD Loop Synthesis [4]. This best paper from PPoPP '13 makes use of inductive synthesis to vectorize computations in a loop so they can take advantage of the SIMD hardware present in modern CPU architectures. One key innovation of the paper is independent of program synthesis—it relates to proving that the optimization is sound. However, the core synthesis problem is solved using a combination of a enumerative search technique combined with an SMT solver. The input to the synthesis tool is a logical specification of the pre- and post-condition of the program to be synthesized. The tool enumerates programs that match the specification on some subset of the inputs, then uses an SMT solver to generate counterexamples, from which the synthesized program is generalized to cover more inputs. This enumerative technique is augmented with various pruning heuristics and scales to the generation of up to 9 vector instructions in just 0.12 seconds. Our experience with *Unagi* suggests that smarter enumerative search techniques, when run at cloud scale, can scale several orders of magnitude further.

An SMT based method for optimizing arithmetic computations in embedded software code [8]. This best paper by Eldib and Wang from FMCAD '13 makes use of an SMT-based synthesis technique to expand the dynamic range of arithmetic instructions while excluding overflow. The authors use the SMT solver Yices [7] in their experiments and are able to synthesize programs of up to 7 instructions in a few seconds. Their technique is a combination of enumerative search with SMT solving. It works by fixing a syntactic skeleton of the program, using an SMT solver to fill in the skeleton. If this fails, the size of the skeleton is increased, and the process is repeated. Although the experiments reported in the paper show only that the technique scales to 7 instructions, the authors claim in private communication that they were able to scale it to 18 instructions using more than an hour of search time. Unfortunately, we are unable to independently verify these claims since, at the time of writing, Eldib and Wang were unable to provide the data and tools mentioned in their paper.

TRANSIT: specifying protocols with concolic snippets [27]. Not all researchers use SMT solvers for synthesis. Udupa et al. develop a purely enumerative exhaustive search technique for inductive synthesis, and implement it in a tool called TRANSIT. The main idea they use to prune the search space is similar to *Unagi*'s "output equivalence class" technique described in §3.2. Given a set of input/output examples, TRANSIT enumerates programs in increasing order of size. At each size class, TRANSIT buckets programs into equivalence classes, where the elements of an equivalence class have the same behavior on the inputs in the provided examples. Only a single representative of an equivalence class is retained when iterating to the next size class. When a candidate program synthesized by TRANSIT does not satisfy the desired specification, a counterexample is added to the set on input/output examples, and the enumerative process is repeated from scratch. The tool produces programs of up to 15 instructions in at most 15 minutes.

Syntax-guided synthesis [2]. This recent paper by Alur et al. has an objective similar to our objective in this paper: they seek to develop a set of standardized benchmarks for program synthesis, and evaluate three existing example-driven inductive program synthesis tools on those benchmarks. The authors of this paper include researchers who are known for their work on several different program synthesis tools in the past—it's great to see them come together to evaluate their tools on common ground, and to set up a framework against which future tools can also be evaluated. They find that an enumerative solver similar to the one used in TRANSIT is the most efficient of three strategies they consider—the other two being an SMT-based approach, and a stochastic approach. As in TRANSIT, the enumerative solver is able to find up to 15 instructions in at most 1000 seconds. Surprisingly, in contrast with Gulwani et al.'s experience with Brahma and with our own experience evaluating an SMT-based test solution to the ICFP-PC, Alur et al. fail to scale their SMT-based solution to programs of size 15, even on the same benchmarks on which Brahma succeeds—this suggests that the precise encodings of synthesis problems in an SMT solver are of crucial importance. Ideally, we would have liked to evaluate *Unagi*'s solution head-to-head with Alur et al.'s tools. However, at the time of writing, we have not yet been able to reproduce their results. We plan to continue our discussions with them and hope to be able to conduct such a head-to-head comparison in the near future.

5. Conclusions

If a synthesis problem is really worth solving (e.g., super-optimizing fragments of performance-critical code, or providing synthesis services in an program editor at user-interaction speed), then throwing widely available, large-scale computing resources at the problem seems perfectly reasonable. Motivated by the contest, *Unagi* recognized this and quickly found creative ways to deploy a sophisticated parallel search algorithm in the cloud.

While many in the research community in program synthesis have realized that domain-specific search strategies are likely to outperform generic SMT-based solutions, few have made the transition to parallelizing their algorithms and deploying them at cloud scale. Whereas the state of the art in ED-synthesis scales to around 17 instructions in an hour, *Unagi* were able to synthesize programs up to 51 instructions long in just 5 minutes. Given that the search space is exponential in the size of the program, *Unagi* were able to search a vastly larger space in just a small fraction of the time—besides, they only had 72 hours to do it.

This suggests that mobilizing resources at the scale of the cloud does not stand to just improve performance by a small multiplicative factor—the improvements in scale can truly be transformational. *Unagi* and the other contestants have shown that through the clouds, only the sky is the limit!

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[3] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, pages 1–35, 2012.

[4] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to simd loop synthesis. In *PPoPP*. ACM, 2013.

[5] D. Beyer. Competition on software verification - (sv-comp). In *TACAS*, 2012.

[6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[7] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[8] H. Eldib and C. Wang. An SMT based method for optimizing arithmetic computations in embedded software code. In *FMCAD*, 2013.

[9] B. Fischer and J. Schumann. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13:483–508, 5 2003.

[10] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In *PLDI*. ACM, 2012.

[11] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[12] C. Green and D. Barstow. On program synthesis knowledge. *Artif. Intell.*, 10(3):241–279, Nov. 1978.

[13] S. Gulwani. Synthesis from examples: Interaction models and algorithms. *2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 0:8–14, 2012.

[14] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*. ACM, 2011.

[16] M. Huisman, V. Klebanov, and R. Monahan. On the organisation of program verification competitions. In *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE)*, volume 873, 2012.

[17] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*. ACM, 2010.

[18] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[19] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*. ACM, 2010.

[20] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426, 2013.

[21] A. S. Koksal, Y. Pu, S. Srivastava, R. Bodik, J. Fisher, and N. Piterman. Synthesis of biological models from mutation experiments. In *POPL*. ACM, 2013.

[22] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[23] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb 2005.

[24] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*. ACM, 2006.

[25] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[26] N. Tillmann, J. D. Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *SEE*, 2013.

[27] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, 2013.