Applying Genetic Programming to PSB2: The Next Generation Program Synthesis Benchmark Suite

Thomas Helmuth[1*] and Peter Kelly[1]

[1]Computer Science, Hamilton College, Clinton, New York, USA.

*Corresponding author(s). E-mail(s): thelmuth@hamilton.edu;
Contributing authors: pxkelly@hamilton.edu;

Abstract

For the past seven years, researchers in genetic programming and other program synthesis disciplines have used the General Program Synthesis Benchmark Suite (PSB1) to benchmark many aspects of systems that conduct programming by example, where the specifications of the desired program are given as input/output pairs. PSB1 has been used to make notable progress toward the goal of general program synthesis: automatically creating the types of software that human programmers code. Many of the systems that have attempted the problems in PSB1 have used it to demonstrate performance improvements granted through new techniques. Over time, the suite has gradually become outdated, hindering the accurate measurement of further improvements. The field needs a new set of more difficult benchmark problems to move beyond what was previously possible and ensure that systems do not overfit to one benchmark suite.

In this paper, we describe the 25 new general program synthesis benchmark problems that make up PSB2, a new benchmark suite. These problems are curated from a variety of sources, including programming katas and college courses. We selected these problems to be more difficult than those in the original suite, and give results using PushGP showing this increase in difficulty. We additionally give an example of benchmarking using a state-of-the-art parent selection method, showing improved performance on PSB2 while still leaving plenty of room for improvement. These new problems will help guide program synthesis research for years to come.

Keywords: automatic program synthesis, benchmarking, genetic programming

1 Introduction

Automatic general program synthesis, with the aim of automatically generating programs of the type humans write from scratch, has long been a goal of artificial intelligence and machine learning. Yet, for many years there were no common benchmark problems for evaluating general program synthesis[1] systems; existing problems were either easy toy problems or were situated in specific domains where solution programs were composed of a small set of domain-specific instructions. In 2015, the General Program Synthesis Benchmark Suite (PSB1) introduced 29 *programming by example* problems that can be used to benchmark program synthesis systems that take their specifications as input/output examples [1]. Since then, more than 100 research papers have benchmarked 10+ program synthesis systems using PSB1, producing numerous insights into program synthesis.

Of the systems that have adopted PSB1, most fall within the field of genetic programming (GP), including PushGP [1], grammar-guided GP [2], grammatical evolution [3], and linear GP [4]. However, non-evolutionary program synthesis methods have also been applied to PSB1, including those based on delayed-acceptance hillclimbing [5] and Monte Carlo tree search [6]. We expand on the details of these methods and the results they have achieved using PSB1 in Section 2, but to summarize, many of these systems have improved performance and demonstrated new techniques.

When PSB1 was first introduced, the initial PushGP benchmarking runs were able to solve 22 of the 29 problems, with an average success rate of 23 successful runs out of 100 [1]. The best-performing PushGP results have now solved 25 problems, with an average success rate of 42/100 [7, 8]. Some of the most drastic improvements have come on some of the most informative problems in PSB1, such as Double Letters ($6 \rightarrow 50$ successes between [1] and [7]), Replace Space with Newline ($51 \rightarrow 100$), Syllables ($18 \rightarrow 64$), Vector Average ($16 \rightarrow 97$), and X-Word Lines ($8 \rightarrow 91$).

Thus, for PushGP and other synthesis systems, the problems of PSB1 have become less useful over time. In particular, the very high performance achieved on some PSB1 problems leaves little room for exhibiting improvement; a few other problems have never been solved and are likely too difficult to be solved any time soon. Additionally, peculiarities in some of the problems in PSB1 make them less ideal as benchmarks, either because of how synthesis systems move through their search space or how slow they are to run. Finally, some decisions about the specification of problems in PSB1 make them difficult to implement, potentially preventing wider adoption.

[1] Also known as automatic programming or software synthesis.

With these drawbacks in mind, we have created a second Program Synthesis Benchmark suite, which we refer to as PSB2. PSB2 consists of 25 problems curated from programming challenges, programming katas, and college courses. In order to facilitate the implementation of PSB2 in new synthesis systems, we provide a reference implementation of each problem, datasets that can be sampled for each problem, and libraries for downloading and sampling the datasets.[2] Just like PSB1, the problems in PSB2 require a wide range of programming techniques, data types, and control flow structures to solve. However, they are markedly harder to solve than problems in PSB1, with our initial results solving 13 of the 25 problems for an average success rate of 10 successful runs out of 100. These more difficult problems will drive program synthesis research toward solving more realistic program synthesis tasks.

The purpose of benchmark problems is to allow us to empirically show what changes to a system produce improvements that may transfer to real-world problems. To achieve this goal, they must be sufficiently difficult, unlike toy problems that have been used as benchmarks in the past. They must also be representative of the types of tasks we want our system to perform. However, we also want benchmarks to be easier and faster to run than an actual real-world problem in order to produce benchmarking results with reasonable amounts of computation. Given that automatic program synthesis is still in its fledgling stages, we see the problems in PSB2 as a stepping stone toward solving more realistic problems.

PSB2 also addresses calls from the GP community to produce and adopt realistic benchmarks. GP community discussions calling for better benchmarks [9–11] inspired the creation of PSB1; these calls also highlighted the need to periodically update and replace benchmark problems in order to keep advancing the field without over-optimizing to a single set of problems. More recently, a call to refocus the efforts of GP on automatic programming stated, "We are in no doubt of the need for the further principled development of additional benchmarks that can be used in a targeted manner to push the boundaries along different dimensions such as scalability, generalisation, and adaptation, and to facilitate comparison across a range of very different approaches to automatic programming" [12]. The creation of PSB2 aims to push the boundaries of program synthesis research and give synthesis systems a fresh set of problems to explore. Of course, there is no need to entirely throw out the problems of PSB1; we could imagine some of the harder problems continuing to provide useful data, and newer systems may need to start on the easier problems as a jumping off point.

PSB2 was originally introduced at the 2021 Genetic and Evolutionary Computation Conference, where it was nominated for Best Paper in the Genetic Programming track [13]. This article expands on that work by extending the discussion of best practices for benchmarking program synthesis using GP, presenting new experiments, and adding a discussion of related program synthesis

[2]Reference implementation, datasets, and other resources can be found on this paper's companion website: https://cs.hamilton.edu/~thelmuth/PSB2/PSB2.html.

techniques. In the new experiments, we use PushGP with down-sampled lexicase selection [7, 8], improving the number of problems solved from 13 to 17, and additionally conduct a control study using random search to ensure that program synthesis is necessary to solve these problems.

The remainder of this paper is structured as follows: in the next section, we discuss research that has used PSB1. In Section 3, we highlight lessons learned about program synthesis benchmarking from PSB1. Sections 4 and 5 describe the sources of PSB2's problems and describe the problems in detail. We then give general guidance on benchmarking with PSB2, and give details of the parameters we used in our experiments in Sections 6, 7 and 8. Sections 9, 10, and 11 present initial results using PushGP, a control study using random search, and a benchmarking study showing improved performance using down-sampled lexicase selection. Finally, Section 12 situates PSB2 in the larger field of program synthesis, discussing how specifications are defined in various subfields.

2 Past Research Using PSB1

PSB1 has been used in a variety of research projects on automatic program synthesis, many of them using GP as the synthesis system. A recent survey gives details of the methods that have been used to solve the problems in PSB1, as well as analyzing the difficulties of the problems [14].

The paper that introduced PSB1 [1] used PushGP, a GP system based on the stack-based Push programming language; a variety of papers using PushGP have made use of PSB1 since [7, 8, 15–20]. Code-building GP is a stack-based GP system borrowing some inspirations from Push that constructs programs in a host language; it solved some of the PSB1 problems, producing solution programs Synthesized in Python [21].

General program synthesis requires the manipulation of multiple data types; stack-based GP systems have handled this requirement well, but so have other GP systems that handle strong typing of programs. In particular, grammar-based approaches such as grammar guided GP (G3P) [2, 22–24] and grammatical evolution (GE) [3, 25–27] have made good progress at solving the problems in PSB1. Many of these use the type-based grammar design patterns introduced to flexibly handle problems with different type requirements [2]. Another use of these grammars trains a sequence-to-sequence variational autoencoder to embed programs in a continuous space and then uses an evolutionary algorithm to optimize programs in this space [28]. Finally, a linear GP system with tag-based memory has also been explored using PSB1 [4, 29, 30].

As for non-GP systems, an approach using delayed-acceptance hillclimbing for inductive synthesis proved competitive with GP on PSB1, including producing the only known solutions to the Collatz Numbers problem [5]. A comparison was made between Flash Fill [31], MagicHaskeller [32], PushGP, and G3P, finding that the non-GP methods fared much worse but ran much

faster than the GP methods [33]. Finally, Monte Carlo tree search was used to generate Java bytecode programs using a few of the problems in PSB1 [6].

3 Lessons Learned

While PSB1 has been successfully used in a variety of research, it was a first attempt at a general-purpose program synthesis benchmark suite. The research community has grown from using it, both in terms of improving program synthesis methods as well as learning lessons about how to best define program synthesis benchmarks. Here, we discuss some of those lessons and how they have influenced our creation of PSB2.

One major issue with PSB1 is that every system that uses it needs to implement all of the problems from scratch. This hurdle likely decreased wider adoption. Additionally, there may be inconsistencies between implementations in different systems, leading to less comparable results; one known such inconsistency is that some systems use new randomized data for each run, while others use the same dataset for every run. Four years after its initial release, the authors of PSB1 created large datasets of the inputs and correct outputs for each problem [34]. These datasets can be sampled for each program synthesis run, meaning there is no need for each system to implement each problem. We have copied this model and provide datasets for each problem in PSB2, as well as software libraries to make them easier to use (see Section 6).

A handful of the problems in PSB1 require programs to produce Boolean outputs, as such functions are common in programming exercises. A trend noted across completely different program representations is that solutions to these Boolean-output problems often do not generalize to unseen data [1, 22]. A simple explanation for this phenomenon is that it is relatively easy for a solution program to produce the correct answers for the wrong reasons when there are only two possible answers, thus overfitting to the training data. It is much harder to perfectly answer training data for the wrong reasons when the output is an integer or string, for example. Because of this issue, we have selected fewer Boolean-output problems for PSB2, including only one representative problem.

PSB1 was designed to emulate the textbook problems it was curated from as closely as possible. For example, many problems from the original textbook required the program to "print" its answers. PSB1 suggested that synthesis systems develop methods for emulating an output buffer and common printing instructions in order to mimic these problems. However, this approach was infeasible for some synthesis systems, which instead simply returned string outputs. As PSB2 is less loosely coupled with its problem sources, we decided to have all programs return their outputs instead of "printing" them. Another wrinkle related to outputs is that some problems require a solution to return multiple outputs. While multiple outputs may prove difficult in some systems, it is generally feasible in all; we have included 4 multi-output problems in PSB2.

PSB1 recommended different training and test set sizes, as well as program evaluation budgets, for each problem. This led to difficulties and confusion in both implementation and reporting results. For PSB2, we recommend using a fixed setting for each parameter across problems, as well as for system-specific parameters, such as maximum sizes for program size control.

Forstenlechner et al. [22] discussed understanding and refining the problems in PSB1, making some general recommendations about both synthesis systems and benchmark problems. One suggestion put forth is using larger and more targeted training sets, to better guide synthesis and increase generalization. We take this recommendation and use large training sets (200 examples) that have a variety of specific edge cases purposefully included. Most of their other suggestions relate to specific system settings, such as the length of evolution; these parameters are not prescribed by PSB2, and can be chosen by the researcher.

Our approach to progressing the field of general program synthesis is an incremental one by design: researchers have improved synthesis systems on the problems of PSB1, so now we wish to find improvements on the moderately more difficult problems of PSB2. However, we could imagine other approaches that could be used to progress the field besides an incremental approach, and other researchers may choose to use these instead. For example, we could create significantly more challenging problems, beyond the capabilities of any current synthesis system. Then, we could work on methods to solve those problems directly, with the hope that working on a more difficult task will allow for the creation of new techniques that may not be obvious with a more incremental approach. The drawback here is that until some method actually solves the problem, there is not much gradient for the research community to follow in order to learn which methods are more promising. Thus we prefer an incremental approach, with benchmarking and comparison of existing methods using moderately more difficult problems.

4 Problem Selection and Sources

Below we describe the four sources we used as inspiration for the problems included in this suite. Each of these sources contains problems for humans to use to improve their programming skills, whether for experienced programmers or students in class. As such, these sources contain problems representative of the types of programming that we expect humans to perform.

Code Wars (CW) - A website full of user-created programming challenges, called *coding kata*. The aim of the site is for users to spend small amounts of time programming every day to hone their coding skills.

Advent of Code (AoC) - An Advent calendar of coding problems created every year in December. These problems can be used for any number of things, like training, interview prep, or coursework. Problems tend to become harder throughout the month.

Homework Problems (HW) - These problems come from programming homework given in our undergraduate programming courses. The problems come from two courses: an introductory programming course, and a program languages course. These problems do not have citations, since we created them for our courses.

Project Euler (PE) - A website containing hundreds of problems in an archive. Users are free to submit answers to validate their solutions. Most problems tend to be mathematically focused and often require efficient and elegant solutions.

These sources contain a large number and variety of problems; we considered over 75 problems from these sources and implemented and tested over 50 of them, filtering out problems that seemed too easy, too difficult, or inaccessible. While curating the suite, we did not include any problems on which standard PushGP produced a success rate over 60% in initial experiments, to ensure that problems are sufficiently difficult to allow for improvement. While other GP systems surely will find some of these problems easier or harder than PushGP, as they have with PSB1, we hope a combination of subjective curation and objective results using PushGP will give a good balance of problems that are moderately more difficult than the problems in PSB1 for any system. In order to be transparent about our curation process, we have created a table containing all of the problems we considered, including the reason for accepting/rejecting each problem, initial results if we implemented the problem, and a link to the source of the problem.[3]

We aimed to include problems that require a large variety of data types and control flow structures to solve, with a balance between data types across problems. Most of the problems require some type of iteration and/or conditional statements. Required data types include integers, floats, Booleans, characters, strings, vectors of integers, and vectors of floats. In order to produce large datasets, we aimed to select problems that have at least 1 million possible unique inputs.

5 Problem Descriptions

Below is a list of the English language descriptions of the 25 benchmark problems in PSB2. Each problem (besides those from our courses) has a citation of its source with a link to the original problem. The types of the input(s) and output(s) for each problem are given in Table 1. Because the type(s) of control flow that are necessary to solve a problem can be a large influence on how difficult it is, we give our best prediction as to which types of control flow will be necessary to solve each problem in Table 2. Note that all problems that require vector/string processing (considering individual elements or characters) must also require iteration over those elements/characters, but we include it as a separate category to distinguish it from problems that simply need iteration

[3] https://docs.google.com/spreadsheets/d/e/2PACX-1vQKO1D2sZA9KosXpOJNuiDW6yDQEZnMrwzNeMJJU25MbZhU6odQ0jGkJN5lgbRgspsmmum65WLbEI2B/pubhtml?gid=0&single=true

Table 1 For each problem, the types of the inputs and outputs, and the limits imposed on the inputs.

Problem	Inputs	Outputs
Basement	vector of integers of length $[1, 20]$ with each integer in $[-100, 100]$	integer
Bouncing B.	float in $[1.0, 100.0]$, float in $[1.0, 100.0]$, integer in $[1, 20]$	float
Bowling	string in form of completed bowling card, with one character per roll	integer
Camel Case	string of length $[1, 20]$	string
Coin Sums	integer in $[1, 10000]$	4 integers
Cut Vector	vector of integers of length $[1, 20]$ with each integer in $[1, 10000]$	2 vectors of integers
Dice Game	2 integers in $[1, 1000]$	float
Find Pair	vector of integers of length $[2, 20]$ with each integer in $[-10000, 10000]$, integer in $[-20000, 20000]$	2 integers
Fizz Buzz	integer in $[1, 1000000]$	string
Fuel Cost	vector of integers of length $[1, 20]$ with each integer in $[6, 100000]$	integer
GCD	2 integers in $[1, 1000000]$	integer
Ind. of Subs.	2 strings of length $[1, 20]$	vector of integers
Leaders	vector of integers of length $[0, 20]$ with each integer in $[0, 1000]$	vector of integers
Luhn	vector of integers of length 16 with each integer in $[1, 9]$	integer
Mastermind	2 strings of length 4 made of `B, R, W, Y, O, G`	2 integers
Middle Char.	string of length $[1, 100]$	string
Paired Digits	string of digits of length $[2, 20]$	integer
Shopping List	vector of floats of length $[1, 20]$ with each float in $[0.0, 50.0]$, vector of floats of length $[1, 20]$ with each float in $[0.0, 100.0]$. Both vectors must be the same length	float
Snow Day	integer in $[0, 20]$, float in $[0.0, 20.0]$, float in $[0.0, 10.0]$, float in $[0.0, 1.0]$	float
Solve Boolean	string of length $[1, 20]$ made of characters from `{t, f, |, &}`	Boolean
Spin Words	string of length $[0, 20]$	string
Square Digits	integer in $[0, 1000000]$	string
Subs. Cipher	3 strings of length $[0, 26]$	string
Twitter	string of length $[0, 200]$	string
Vector Dist.	2 vectors of floats of length $[1, 20]$ with each float in $[-100.0, 100.0]$	float

without vector/string processing. For more precise details of each problem, see the reference implementation.[4]

1. **Basement (AoC)** Given a vector of integers, return the first index such that the sum of all integers from the start of the vector to that index (inclusive) is negative. [35]

2. **Bouncing Balls (CW)** Given a starting height and a height after the first bounce of a dropped ball, calculate the *bounciness index* (height of first bounce / starting height). Then, given a number of bounces, use

[4] https://github.com/thelmuth/Clojush/releases/tag/psb2-v1.0

Table 2 For each problem, the types of control flow that a program likely needs to include to solve the problem. **Iteration** refers to some sort of iteration, looping, or recursion. **Vector** includes an type of vector/string processing, considering individual elements or characters. **Conditional** refers to conditional statements of some kind, whether through standard *if* statements or other means.

Problem	Iteration	Vector	Conditional
Basement	x	x	x
Bouncing B.	x		
Bowling	x	x	x
Camel Case	x	x	x
Coin Sums			
Cut Vector	x	x	x
Dice Game			
Find Pair	x	x	x
Fizz Buzz			x
Fuel Cost	x	x	
GCD	x		x
Ind. of Subs.	x	x	x
Leaders	x	x	x
Luhn	x	x	x
Mastermind	x	x	x
Middle Char.			x
Paired Digits	x	x	x
Shopping List	x	x	
Snow Day	x		
Solve Boolean	x	x	x
Spin Words	x	x	x
Square Digits	x		
Subs. Cipher	x	x	
Twitter			x
Vector Dist.	x	x	
Total	20	16	16

the bounciness index to calculate the total distance that the ball travels across those bounces. [36]

3. **Bowling (CW)** Given a string representing the individual bowls in a 10-frame round of 10 pin bowling, return the score of that round. [37]

4. **Camel Case (CW)** Take a string in `kebab-case` and convert all of the words to `camelCase`. Each group of words to convert is delimited by `"-"`, and each grouping is separated by a space. For example: `"camel-case example-test-string"` → `"camelCase exampleTestString"`. [38]

5. **Coin Sums (PE)** Given a number of cents, find the fewest number of US coins (pennies, nickles, dimes, quarters) needed to make that amount, and return the number of each type of coin as a separate output. [39]

6. **Cut Vector (CW)** Given a vector of positive integers, find the spot where, if you cut the vector, the sum of the numbers on both sides are

either equal, or the difference in sums is as small as possible. Return the two resulting subvectors as two outputs. [40]

7. **Dice Game (PE)** Peter has an n sided die and Colin has an m sided die. If they both roll their dice at the same time, return the probability that Peter rolls strictly higher than Colin. [41]

8. **Find Pair (AoC)** Given a vector of integers, return the two elements that sum to a target integer. [42]

9. **Fizz Buzz (CW)** Given an integer x, return `"Fizz"` if x is divisible by 3, `"Buzz"` if x is divisible by 5, `"FizzBuzz"` if x is divisible by 3 and 5, and a string version of x if none of the above hold. [43]

10. **Fuel Cost (AoC)** Given a vector of positive integers, divide each by 3, round the result down to the nearest integer, and subtract 2. Return the sum of all of the new integers. [44]

11. **GCD [Greatest Common Divisor] (CW)** Given two positive integers, return the largest integer that divides each of the integers evenly. [45]

12. **Indices of Substring (CW)** Given a text string and a target string, return a vector of integers of the indices at which the target appears in the text. If the target string overlaps itself in the text, all indices (including those overlapping) should be returned. [46]

13. **Leaders (CW)** Given a vector of positive integers, return a vector of the leaders in that vector. A leader is defined as a number that is greater than or equal to all the numbers to the right of it. The rightmost element is always a leader. [47]

14. **Luhn (CW)** Given a vector of 16 digits, implement Luhn's algorithm to verify a credit card number, such that it follows the following rules: double every other digit starting with the second digit. If any of the results are over 9, subtract 9 from them. Return the sum of all of the new digits. [48]

15. **Mastermind (HW)** Based on the board game Mastermind. Given a Mastermind code and a guess, each of which are 4-character strings consisting of 6 possible characters, return the number of white pegs (correct color, wrong place) and black pegs (correct color, correct place) the codemaster should give as a clue.

16. **Middle Character (CW)** Given a string, return the middle character as a string if it is odd length; return the two middle characters as a string if it is even length. [49]

17. **Paired Digits (AoC)** Given a string of digits, return the sum of the digits whose following digit is the same. [50]

18. **Shopping List (CW)** Given a vector of floats representing the prices of various shopping goods and another vector of floats representing the percent discount of each of those goods, return the total price of the shopping trip after applying the discount to each item. [51]

19. **Snow Day (HW)** Given an integer representing a number of hours and 3 floats representing how much snow is on the ground, the rate of snow fall, and the proportion of snow melting per hour, return the amount of snow on the ground after the amount of hours given. Each hour is considered a discrete event of adding snow and then melting, not a continuous process.

20. **Solve Boolean (CW)** Given a string representing a Boolean expression consisting of `T`, `F`, `|`, and `&`, evaluate it and return the resulting Boolean. [52]

21. **Spin Words (CW)** Given a string of one or more words (separated by spaces), reverse all of the words that are five or more letters long and return the resulting string. [53]

22. **Square Digits (CW)** Given a positive integer, square each digit and concatenate the squares into a returned string. [54]

23. **Substitution Cipher (CW)** This problem gives 3 strings. The first two represent a cipher, mapping each character in one string to the one at the same index in the other string. The program must apply this cipher to the third string and return the deciphered message. [55]

24. **Twitter (HW)** Given a string representing a tweet, validate whether the tweet meets Twitter's original character requirements. If the tweet has more than 140 characters, return the string `"Too many characters"`. If the tweet is empty, return the string `"You didn't type anything"`. Otherwise, return `"Your tweet has X characters"`, where the X is the number of characters in the tweet.

25. **Vector Distance (CW)** Given two n-dimensional vectors of floats, return the Euclidean distance between the two vectors in n-dimensional space. [56]

6 Implementing PSB2

While Section 5 provides English-language descriptions of the 25 benchmark problems, these are not sufficient to implement each problem in a new system. Here we discuss the system-agnostic details for implementing these problems in new synthesis systems.

For reasons discussed in Section 3, we have created datasets consisting of large numbers of inputs and correct outputs for every problem [57].[5] The

[5]Our datasets follow the model of other machine learning datasets such as Penn ML Benchmarks [58, 59] and the UCI ML Repository [60].

dataset for each problem consists of a small number of hand-chosen inputs, often addressing edge cases for the problem, and 1 million randomly-generated inputs falling within the constraints of the problem's inputs; the only exception being the Coin Sums problem, which has 10,000 possible integer inputs. We recommend that each different program synthesis run use a different set of training data, composed of every one of the hand-chosen inputs and a random sample of the randomly-generated inputs. The alternative method of using the same fixed set of inputs for every run could happen to use a particularly lucky (or unlucky) set of inputs, and across multiple experiments, could lead to undesirable tuning to the particular data; using different randomized inputs for each run avoids these issues. Our datasets will allow those implementing PSB2 to simply sample the provided data, greatly decreasing the barrier to using PSB2. The PSB2 datasets can be found permanently on Zenodo.[6] For more information about distributions of inputs in randomly-generated inputs, see the reference implementation, which was used to generate the datasets.[7]

In order to make it even easier to download and sample the PSB2 datasets, we have created libraries that implement this functionality in Python[8] and Clojure[9]. Each library can be easily installed using the host language's package management system. Both libraries provide one function that downloads, caches, and samples the dataset to produce training and test sets for a given problem.

When using our provided datasets, one could sample different sizes of training and unseen test sets to fit a given experiment. Our recommendation, which we use in the experiments, is to use 200 example cases for the training set (including all hand-chosen inputs and the remaining inputs sampled randomly) and 2000 for the unseen test set. However, some synthesis methods may need smaller or larger training sets, and PSB2 can flexibly adapt to such systems.

Program synthesis methods that have been applied to PSB1 have used varying methods for constraining the instruction set and other program syntax. For example, some have used grammars [2, 3, 23, 25–28] while others have used data-type categorized subsets of an instruction set [1, 18]. We do not want to constrain what a reasonable approach to selecting instructions may look like for any given program synthesis system. However, we also warn against cherry-picking a small subset of instructions suspected of being useful for a particular problem. Part of the difficulty of general program synthesis is that a system must manage a large set of potentially useful instructions, finding those relevant to a particular problem. We recommend employing a large set of general-purpose instructions when using PSB2 to benchmark program synthesis to best replicate the conditions of a real-world scenario.

[6] https://zenodo.org/record/4678739
[7] https://github.com/thelmuth/Clojush/releases/tag/psb2-v1.0
[8] https://github.com/thelmuth/psb2-python
[9] https://github.com/thelmuth/psb2-clojure

7 Best Practices for Benchmarking

There are many ways to use a suite of benchmark problems, ranging from illustrating a synthesis method's behavior to tuning algorithms and hyperparameters [61]. However, many uses require fair comparisons between systems or methods in order to benchmark performance or other characteristics. Here we make recommendations for benchmarking best practices in order to ensure fair comparisons.

In order to compare performance, we need to ensure that the compared systems or methods are given similar computational resources. Arguments could be made for a variety of budgets to limit execution, including processing time, number of GP generations, program fitness evaluations (across all training cases), program executions (per case), and number of individual instructions executed. All of these methods have flaws, and all might be useful in certain circumstances. However, we generally recommend using a fixed program execution budget to limit the number of generated program executions in a single program synthesis run. This type of budget is more general than using a budget of generations or fitness evaluations, as it allows systems to use different population sizes or training set sizes yet still budget similar computation. On the other hand, a budget of processing time or instruction executions may be misleading when running experiments on different hardware or different synthesis systems. *We recommend using a budget of 60 million program executions*; we allocate these to 200 training cases used to evaluate a population of 1000 individuals for 300 generations in our experimental GP runs, but other allocations of the same executions would be reasonable.

In order to produce sufficient data on the performance of a stochastic search such as GP, one needs to conduct multiple runs/trials of the system to get an idea of the distribution of results. While we do not want to set a fixed number of recommended trials, more are certainly better when considering the significance of the collected results. We expect that around 30 trials would be the minimum acceptable for showing significant differences, though larger numbers of trials such as 100 or more would be more likely to produce significant differences in statistical tests; we use 100 runs per problem in our experiments.

When evaluating the performance of a synthesis system on PSB2, we recommend using *success rate* (the number of synthesis runs that produce a solution) as the primary measure of performance, as was recommended in PSB1 [1]. For the synthesis of software, generating programs that pass most, but not all, of the training cases is not sufficient; for this reason success rate is a better measure of performance than other metrics such as mean best fitness or mean number of training cases passed. In particular, a solution must not only pass all cases in the training set, but also all of the cases in the test set, to ensure that it generalizes to unseen data. This avoids considering programs that overfit the training data, such as by memorizing the correct output to each input, as solutions. We recommend sampling 2000 (or more) random inputs to create the unseen test set.

In order to determine whether differences in performance are statistically significant, experimenters should use an appropriate significance test. In the case of success rates measured in numbers of success and failure runs, a chi-squared significance test is appropriate. When comparing multiple methods, it is necessary to apply a correction to account for multiple comparisons, such as the Holm–Bonferroni correction.

8 Experimental Methods and System Parameters

In this section we will discuss the system-specific parameters and choices that must be decided in order to use PSB2. In contrast with the previous sections on general implementation and benchmarking considerations, the choices here may differ considerably for different program synthesis systems. For our experiments, we used PushGP; we will describe in general the decisions that must be made and give our specific choices.

PushGP evolves programs in the language Push, a stack-based programming language built specifically for use in genetic programming [62, 63]. Every data type has its own stack, and each Push instruction acts by pushing and popping various elements on and off the stacks. The output of each problem is typically the top element on a particular stack. The interpreter executes programs that are themselves placed on an `exec` stack, allowing `exec` instructions to manipulate control flow as well as the program itself as it runs. We provide a reference implementation in Clojure of the PushGP system used to produce our results, which includes each problem in PSB2.[10] This reference implementation is the same implementation of PushGP used in recent research using PSB1, e.g. [7, 16, 17].

We discuss the general design of program synthesis instruction sets in Section 6. For our PushGP experiments, we use the general process recommended in PSB1, where, for each problem, we identify which data types (corresponding to stacks) are relevant and include all implemented instructions that use those stacks [1]. In Tables 3 and 4, we present the data types we chose to include for each problem, and the total number of instructions in the instruction set. Every problem includes the `exec`, `integer`, and `Boolean` types, since they are required for a variety of control flow instructions such as iteration and conditionals. These large instruction sets contain a wide range of general-purpose Push instructions, including some new instructions implemented since PSB1, avoiding the cherry-picking of clearly useful instructions. For the full list of instructions for each data type, see Appendix A.

Research utilizing PSB1 in using transfer-learned instruction sets showed that the composition of the instruction set matters a great deal to problem-solving performance [18]. While we do not use fully transfer-learned instruction sets here, we do make use of one simple take-away: that including larger proportions of input instructions and literals/ERCs improves performance. An

[10] https://github.com/thelmuth/Clojush/releases/tag/psb2-v1.0

Table 3 Instructions and data types used in our PushGP implementation of each problem. The column "# Instructions" reports the number of instructions, terminals, and ephemeral random constants (ERC) used for each problem. The middle columns show which data types were used for each problem. For example, the Basement problem used all instructions relevant to exec, integers, Booleans, and vectors of integers. The last column lists the literals and ERCs used for the problem. Here, char literals are represented in Clojure syntax, starting with a backslash, and strings are surrounded by double quotation marks. The "Problems" row simply counts how many problems use each data type. The "Instructions" row shows the number of Push instructions that primarily use each data type; some use multiple types but are only counted once.

Problem	# Instructions	exec	integer	float	Boolean	char	string	vector of integers	vector of floats	Literals and ERCs (besides inputs)
Basement	117	x	x		x			x		`[], -1, 0, 1, integer ERC`
Bouncing B.	127	x	x	x	x					`0.0, 1.0, 2.0`
Bowling	161	x	x		x	x	x			`\-, \X, \/, \1, \2, \3, \4, \5, \6, \7, \8, \9, 10, integer ERC`
Camel Case	151	x	x		x	x	x			`\-, \space, visible character ERC, string ERC`
Coin Sums	86	x	x		x					`0, 1, 5, 10, 25`
Cut Vector	116	x	x		x			x		`[], 0`
Dice Game	125	x	x	x	x					`0.0, 1.0`
Find Pair	120	x	x		x			x		`-1, 0, 1, 2, integer ERC`
Fizz Buzz	118	x	x		x		x			`"Fizz", "Buzz", "FizzBuzz", 0, 3, 5`
Fuel Cost	117	x	x		x			x		`0, 1, 2, 3, integer ERC`
GCD	79	x	x		x					`integer ERC`
Ind. of Subs.	184	x	x		x	x	x	x		`[], "", 0, 1`
Leaders	114	x	x		x			x		`[], vector ERC`
Luhn	117	x	x		x			x		`0, 2, 9, 10, integer ERC`
Mastermind	123	x	x		x	x	x			`0, 1, \B, \R, \W, \Y, \O, \G`
Middle Char.	151	x	x		x	x	x			`"", 0, 1, 2, integer ERC`
Paired Digits	149	x	x		x	x	x			`0, char digit ERC, integer ERC`
Shopping List	161	x	x	x	x				x	`0.0, 100.0, float ERC`

Table 4 Continuation of Table 2

Problem	# Instructions	exec	integer	float	Boolean	char	string	vector of integers	vector of floats	Literals and ERCs (besides inputs)
Snow Day	131	x	x	x	x					`0, 1, -1, 0.0, 1.0, -1.0`
Solve Boolean	153	x	x		x	x	x			`true, false, \t, \f, \l, \&`
Spin Words	152	x	x		x	x	x			`4, 5, \space, visible character ERC, string ERC`
Square Digits	151	x	x		x	x	x			`"", 0, 1, 2, integer ERC`
Subs. Cipher	151	x	x		x	x	x			`"", 0`
Twitter	153	x	x		x	x	x			`0, 140, "Too many characters", "You didn't type anything", "your tweet has " , " characters"`
Vector Dist.	160	x	x	x	x				x	`[], 0`
Problems		25	25	5	25	11	12	7	2	
Instructions		29	33	45	21	21	47	34	34	

explanation of this result is that most Push instructions decrease stack sizes by consuming arguments and producing fewer return values, so increasing inputs and literals creates more data on which instructions can act. We boost the presence of input instructions and literals in the instruction set, making input instructions fill 15% of the instruction set and literals/ERCs fill 5% of the instruction set. The additional input instructions are evenly distributed between each input for problems with multiple inputs, and literals/ERCs are similarly evenly distributed for each listed in the last column of Tables 3 and 4.

For problems with multiple outputs, different synthesis systems will need to make choices specific to the language of the synthesized programs. Initial experiments in PushGP show that it achieves better results on multi-output problems when using one output instruction per output. These output instructions are included in the instruction set for such problems and will always appear in solution programs. An example of this is for Coin Sums, which has 4 outputs. We provide four corresponding output instructions, each of which takes the top integer from the integer stack and stores it in a write-only register for that output; further calls to an output instruction will overwrite this output register.

In order to define each problem for GP, we not only need the inputs and correct outputs for each problem, but also how to calculate the error function based on the correct output and a program's output. Here we describe the error functions we employed in our experiments, which we recommend for any GP system implementing PSB2; other non-GP program synthesis systems may require entirely different metrics. For each output data type, we use the following standard error functions for problems outputting that data type:

- **Integer or float:** absolute value of the difference between program output and correct output.
- **Boolean:** 0 for correct and 1 for incorrect output.
- **String:** Levenshtein string edit distance between the program output and correct output.
- **Vector of integers:** add the difference in length between the program's output vector and the correct vector times 1000 to the absolute difference between each integer and the corresponding integer in the correct vector.

The only exception is for the Indices of Substring problem, where we used Levenshtein distance to compare vectors of integers, since it makes more sense when comparing vectors of indices. In PushGP, some evolved programs will not return values of a problem's output data type; we give a penalty error value specific to the problem when this occurs.

As has been shown to be effective at improving generalization, we use an automatic simplification procedure on every evolved Push program that passes all of the training cases before testing it on the test set [16].

Unlike for PSB1, we aimed to keep all system-specific parameters constant between problems, increasing ease of use for both implementation and reporting of results. These parameters were chosen based on prior experience and reasonable performance; we leave optimizing parameter settings as an open research question. Other systems may choose to use different system-specific parameters.

Our PushGP system uses linear Plush genomes that are initialized by generating lists of random instructions from the instruction set [15]. We list the important parameters used in our experiments below:

- Maximum initial genome size: 250 genes
- Maximum genome size: 500 genes
- Population size: 1000
- Maximum generations per run: 300
- Maximum steps of the Push interpreter when executing one program: 2000
- Parent selection: lexicase selection [64, 65]
- Genetic operator: Uniform Mutation with Additions and Deletions (UMAD), used to make 100% of children [17].
- UMAD addition rate: 0.09

As described in Section 7, using the exact same population size and generations is not necessary for comparisons between systems; instead, we recommend

Table 5 Results from 100 PushGP runs on each problem. "Succ." gives the number of runs that successfully find a program that passes every training case and perfectly passes a set of 2000 unseen test cases. "Gen." gives the proportion of solutions on the training data that generalize to unseen data. "Size" gives the size of the smallest automatically simplified solution that generalized to unseen data in any of our runs. "Time" is the average number of seconds taken per generation.

Problem	Succ.	Gen.	Size	Time
Basement	1	1.00	16	250
Bouncing Balls	0	0.00	15	311
Bowling	0	-	-	206
Camel Case	1	1.00	20	95
Coin Sums	2	1.00	22	213
Cut Vector	0	-	-	194
Dice Game	0	-	20	287
Find Pair	4	1.00	13	763
Fizz Buzz	25	0.96	15	281
Fuel Cost	50	1.00	9	305
GCD	8	0.67	9	198
Indices of Substring	0	-	19	241
Leaders	0	-	-	302
Luhn	0	-	-	239
Mastermind	0	-	-	126
Middle Character	57	0.86	10	547
Paired Digits	8	1.00	12	250
Shopping List	0	-	-	714
Snow Day	4	1.00	11	263
Solve Boolean	5	1.00	18	373
Spin Words	0	-	-	443
Square Digits	0	-	13	435
Substitution Cipher	60	0.98	9	395
Twitter	31	0.74	22	527
Vector Distance	0	-	-	667

using a maximum budget of 60 million program executions regardless of other settings.

9 Experimental Results

In order to give a baseline performance of the 25 problems in PSB2, we conducted 100 PushGP runs on each problem using the experimental methods described in Section 8. Additionally, in Section 10 we conduct a control study using randomly generated programs.

We present success rates of our runs in Table 5. Out of the 25 problems in PSB2, 13 were solved by PushGP. Of these 13, 3 of them had 50 or more successes (Fuel Cost, Middle Character, and Substitution Cipher) and 2 others had 25 or more successes (Fizz Buzz and Twitter). The remaining 8 had fewer than 10 solutions, showing that they are solvable by GP but leave a lot of room for improvement.

The second column in Table 5 gives the generalization rate of all evolved solutions for problems on which PushGP produced at least one program that

solved every training case. The generalization rate is calculated as the number of solution programs that pass the unseen test set divided by the number of solution programs that pass the training set. For most problems with training set solutions, those solutions tended to generalize well with rates of 0.95 to 1.0. The three problems with lower generalization rates, GCD, Middle Character, and Twitter all had rates over 0.5. However, Bouncing Balls found 2 solutions on the training data, but neither of them generalized to the test, which resulted in 0 successful runs and a 0.00 generalization rate.

Another way of approximating the difficulty of these problems is by looking at the size of the smallest solution program found for each problem. Smaller solutions are easier for a program synthesis system to generate, simply because they require assembling fewer instructions in the right order. Our results are particular to Push program solutions, but should correlate with the sizes of programs needed to solve these problems in other systems. In order to find each size, we took each solution program and automatically simplified it to produce a smaller equivalent program [16, 66]. Of these simplified programs, in Table 5 we report the smallest simplified solution size out of all simplified solutions to each problem (including experiments we conduct in Section 11). We see that the smallest solution size is 9 instructions for three problems, and one other has a size of 10; three of these four problems also had the highest success rates in PSB2. Many others have larger smallest solution sizes, though we note that with the small sample sizes of solutions for some problems, smaller solutions may exist. In comparison, [1] reported that 8 of the problems in PSB1 had a smallest Push solution size less than 9, the minimum for PSB2. Along with success rates, these sizes of smallest solutions give evidence that the problems in PSB2 are more difficult than those in PSB1.

The last column of Table 5 gives the average number of seconds per generation over all of the PushGP runs for the problem. Note that these runs were conducted on two different computing clusters, each of which is composed of heterogeneous machines, so these measurements should only be considered as rough approximations of running time. To that end, we note that all problems have generational running times within one order of magnitude of each other, meaning there are not any exceptionally slow or fast problems.

10 Control Study: Random Search

We have shown GP capable of solving many of the problems in PSB2; however, are program synthesis techniques necessary to solve these problems? In order to answer this question, we conduct a control study that uses purely random search. To do so, we generate a large number of random Push programs using the same initialization procedure we use to create initial GP populations, with the same range of initial program sizes. We test each random program on training data to see if it solves the given benchmark problem.

To make a fair comparison with our GP study, we run the same number of program executions with one "run" of random search. Our GP runs evaluate

```
(boolean_yank exec_yankdup in1 boolean_dup
 in1 integer_gte integer_flush in2 in1 exec_y
  (in1 integer_yankdup integer_mod integer_swap))

(-1 exec_k (-8 exec_do*count -5 integer_flush in2 in1
 exec_y (in2 integer_yankdup integer_mod integer_swap)))
```

Fig. 1 The two simplified Push solutions generated by random search to the GCD problem.

300,000 individuals, using a population size of 1000 over 300 generations. While we could simply run 300,000 randomly generated programs on each of the 200 training cases for a problem, since the relative performance of each program does not matter (since we are not using it to guide search), we only need to know if each program perfectly passes all cases or not. As such, we can instead execute each program on one case at a time, stopping and moving on to the next program when it fails a single case. Since most random programs will fail the first case they are tested on, we can evaluate nearly 200 times as many random programs this way, equivalent to the 200 cases used for evaluating GP programs. Thus nearly 60 million random programs can be tested with the same execution budget as one GP run. We repeated this process 100 times per problem, to give random search the same computation as GP.

Out of the 25 PSB2 problems, random search found only 2 solutions, both to the GCD problem. The automatically simplified versions of those solution programs are presented in Figure 1. Both programs have similar structure, especially near the end, starting with `exec_y`, a version of the Y combinator. In both programs, `exec_y` recursively calls the parenthesized code block, which performs the heart of the Euclidean algorithm for finding a greatest common denominator. The random generation of these two solutions is surprising; while GCD is tied for the smallest simplified solution sizes in PSB2, all solutions so far need at least 9 instructions, and it is not among the 5 problems solved most often in our initial PushGP results (see Table 5). However, GCD does have the smallest instruction set of the PSB2 problems, with 79 instructions being significantly smaller than the instruction set for most other problems (see Table 3). This smaller instruction set reduces the search space a bit, but we would still expect it unlikely to randomly generate these solutions in the huge combinatorial space.

Giving random search almost 200 times as many programs to consider as GP, it has not produced a single solution to any of the PSB2 problems besides GCD. This is no great surprise, since even the problems with the smallest possible solutions need to line up at least 9 instructions in the right order to solve the problem, producing a large combinatorial space when considering that every problem uses a set of at least 79 instructions or more. These results show that some type of program synthesis method, such as GP, is almost always necessary to solve the problems in PSB2.

11 Down-sampled Lexicase Selection

In order to demonstrate the utility of PSB2 for benchmarking GP, we have conducted a set of PushGP runs using down-sampled lexicase selection [7, 8, 29, 30] to compare with standard lexicase selection. These runs additionally assess the current state-of-the-art for PushGP, since down-sampled lexicase has given better results than standard lexicase on the PSB1 problems.

In down-sampled lexicase selection, the population each generation is only evaluated on a random subsample of the available training cases, with the subsample being resampled each generation. We used a down-sampling level of 0.25, meaning only 50 out of 200 training cases are used each generation. By down-sampling, less information is available for evaluating the performance of each individual; however, each individual is executed fewer times. As such, we can allow GP to run for more generations (or use a larger population size) while maintaining the same execution budget. In our experiments here, we increased the maximum number of generations from 300 to 1200, a change proportional to the down-sampling level, maintaining the execution budget of 60 million programs. In previous work using PSB1, the drawback of having less information to assess each program was more than compensated for by increasing the number of individuals considered during search, and here we assess whether that holds true for PSB2.

The results comparing down-sampled lexicase selection to standard lexicase selection can be found in Table 6. Down-sampling improves results compared to lexicase selection across the board, solving 17 of the 25 problems. In addition, the success rates are the same or improved on every problem, with eight of those improvements being significant using a chi-squared test. These results show clear indications that down-sampled lexicase selection is an improvement over standard lexicase selection on the PSB2 problems, aligning with previous results [7, 8].

These results exemplify how PSB2 can be useful for benchmarking, in particular that different methods can produce significantly different results when using the PSB2 problems. At this point, 18 out of 25 problems have been solved at least once using PushGP: 17 of those with down-sampled lexicase selection, and 1 additional in our initial exploratory PushGP runs (the Leaders problem). While we have no guarantees that the other 7 problems can be solved by any program synthesis system, they provide useful targets for future research.

12 Related Work in Program Synthesis

We expect that PSB2 can be adopted by GP program synthesis practitioners and others who used PSB1. Here, we would like to examine the place of PSB2 within the larger community of program synthesis, to see both where it can be adopted and where there is room for cross-pollination of ideas. Program synthesis encompasses a large and varied field; here, we cite representative examples of these approaches without conducting a full literature review of the field.

Table 6 Number of successful runs comparing lexicase selection to down-sampled lexicase selection. Underlined values indicate significant improvement of down-sampled lexicase over lexicase using a chi-squared test with a 0.05 significance level. Lexicase never produced better results than down-sampled lexicase. "Problems solved" counts the number of problems each method solved at least once.

Problem	Down-sampled	Lexicase
Basement	2	1
Bouncing Balls	3	0
Bowling	0	0
Camel Case	4	1
Coin Sums	39	2
Cut Vector	0	0
Dice Game	1	0
Find Pair	20	4
Fizz Buzz	74	25
Fuel Cost	67	50
GCD	20	8
Indices of Substring	4	0
Leaders	0	0
Luhn	0	0
Mastermind	0	0
Middle Character	79	57
Paired Digits	17	8
Shopping List	0	0
Snow Day	7	4
Solve Boolean	5	5
Spin Words	0	0
Square Digits	2	0
Substitution Cipher	86	61
Twitter	52	31
Vector Distance	0	0
Problems solved	17	13

Programming by example (PBE) generally includes any method where the specification of the intended program is given by input/output examples, as is the case with PSB2. Most GP methods, including those mentioned in Section 2, use PBE. That said, many non-GP PBE synthesis methods have been proposed [67–73]. Many of these use non-evolutionary search over the space of possible programs, while others use satisfiability modulo theories (SMT) solvers to generate programs. In either case, the systems in this space are typically limited to generating programs using small domain-specific languages over one or two datatypes (such as manipulating strings or vectors), both limitations that would make them difficult to apply to every problem in PSB2. However, we encourage the exchange of ideas between these fields, and hope that some non-GP PBE methods can be applied to PSB2 soon.

Natural language specifications, for example a docstring of a desired function, give a natural but often loose specification of the intended functionalty. However, program synthesis based on natural language specifications could prove very useful if the desired programs can be accurately synthesized.

Recently, advances in language modeling have led to methods based on producing abstract syntax trees [74, 75], using component-based synthesis [76], or that simply produce sequences of source code strings [77, 78]. Some of these methods combine natural language specifications with PBE [74, 76]. While some of these methods have only been benchmarked on creating small, domain-specific programs [74–76], those producing source code generate high-level code [77, 78]. These methods could be tested directly on PSB2's problem descriptions given in Section 5.

Some methods define the problem to be solved using a *formal specification*, such as a logic formula or a reference implementation [79–83]. These methods also require either a *sketch* of the desired program with holes to be filled in [79] or a problem-specific syntax defined by a grammar [80–83]. For example, Syntax-Guided Synthesis (SyGuS) problems require formal specifications as well as a context-free grammar describing allowable syntax [81]. These methods have been tested on domain-specific benchmark tasks similar to those often used for PBE, such as those manipulating one of the following: integers, strings, bit vectors, or Boolean circuits. One interesting approach combines GP and sketching, where GP creates the partial program and then a SMT solver is used to fill in the holes [70]. Another combines GP with SMT solvers to decompose problems into smaller parts [83]. Any of these methods would require the creation of formal specifications of the PSB2 problems, instead of using examples of the desired behavior.

13 Conclusions

We have presented PSB2, the second generation of general program synthesis benchmark problems. We discussed the past research that has used PSB1, the lessons learned from years of its use, and why we need a new benchmark suite. We then provided the sources and problems that make up PSB2, giving details of how to implement and use it in new systems. Finally, we presented experimental results showing the increased difficulty of the problems of PSB2 compared to PSB1, and offer comparisons to other synthesis techniques.

We anticipate that using other GP systems (such as those we mention in Section 2 that have used PSB1) to produce initial results on PSB2 will provide a useful comparison to the results with PushGP we have presented here. Research that has compared these different GP methods on the problems in PSB1 has shown them to perform better or worse on different problems, likely due to their different GP representations and instruction sets [3, 14, 33]; we would expect to see differences in abilities on the problems in PSB2 as well. Additionally, we encourage the application of non-evolutionary automatic program synthesis methods to these problems, to better gauge the strengths and weaknesses of these different methods.

PSB2, like PSB1 before it, focuses on core programming methods and data types. As program synthesis systems become more complex and encounter real-world scenarios, they may need to utilize more complex resources such

as concurrency or database manipulation. Additionally, they may need to manipulate problem-specific data types or library-specific functions (a proof-of-concept of one approach to these problems was given using code-building GP [21]). We imagine that new benchmark problems that address these challenges will need to be developed in the future.

The lessons learned from PSB1 will make it easier to implement PSB2 in new program synthesis systems, increasing adoption in the community and furthering the field. PSB2 will provide a new target for program synthesis systems, stretching their capabilities and moving the field toward the types of problems that may be encountered in real-world program synthesis applications.

Acknowledgments

The authors would like to thank Lee Spector, Grace Woolson, and Amr Abdelhady for discussions that helped shape this work.

Appendix A Push Instructions

Our experiments using PushGP use instruction sets based on the data types relevant to the problem, as discussed in Section 8. Tables A1 and A2 contain the Push instructions for each of the types referenced in Tables 3 and 4. Experiments using PSB2 do not need to match this instruction set exactly, and indeed for non-Push program synthesis systems, there may be wildly different instruction sets. However, we include the exhaustive list of instructions that we used in order to give the full details of the system and allow comparisons with instruction sets in other systems.

References

[1] Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM, Madrid, Spain (2015). https://doi.org/10.1145/2739480.2754769

[2] Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A grammar design pattern for arbitrary program synthesis problems in genetic programming. In: EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming. LNCS, vol. 10196, pp. 262–277. Springer, Amsterdam (2017). https://doi.org/10.1007/978-3-319-55696-3_17

[3] Hemberg, E., Kelly, J., O'Reilly, U.-M.: On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In: GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM, Prague, Czech Republic (2019). https://doi.org/10.1145/3321707.3321865

Table A1 The Push instructions included in our experiments.

Data Type	Instructions
exec	exec_do*count, exec_do*range, exec_do*times, exec_do*vector_float, exec_do*vector_integer, exec_do*while, exec_dup, exec_dup_items, exec_dup_times, exec_empty, exec_eq, exec_flush, exec_if, exec_k, exec_noop, exec_pop, exec_rot, exec_s, exec_shove, exec_stackdepth, exec_string_iterate, exec_swap, exec_when, exec_while, exec_y, exec_yank, exec_yankdup, tag, tagged
integer	integer_abs, integer_add, integer_dec, integer_div, integer_dup, integer_dup_items, integer_dup_times, integer_empty, integer_eq, integer_flush, integer_fromboolean, integer_fromchar, integer_fromfloat, integer_fromstring, integer_gt, integer_gte, integer_inc, integer_lt, integer_lte, integer_max, integer_min, integer_mod, integer_mult, integer_negate, integer_pop, integer_pow, integer_rot, integer_shove, integer_stackdepth, integer_sub, integer_swap, integer_yank, integer_yankdup
float	float_abs, float_add, float_arccos, float_arcsin, float_arctan, float_ceiling, float_cos, float_dec, float_div, float_dup, float_dup_items, float_dup_times, float_empty, float_eq, float_floor, float_flush, float_fromboolean, float_fromchar, float_frominteger, float_fromstring, float_gt, float_gte, float_inc, float_log10, float_log2, float_lt, float_lte, float_max, float_min, float_mod, float_mult, float_negate, float_pop, float_pow, float_rot, float_shove, float_sin, float_sqrt, float_square, float_stackdepth, float_sub, float_swap, float_tan, float_yank, float_yankdup
Boolean	boolean_and, boolean_dup, boolean_dup_items, boolean_dup_times, boolean_empty, boolean_eq, boolean_flush, boolean_fromfloat, boolean_frominteger, boolean_invert_first_then_and, boolean_invert_second_then_and, boolean_not, boolean_or, boolean_pop, boolean_rot, boolean_shove, boolean_stackdepth, boolean_swap, boolean_xor, boolean_yank, boolean_yankdup
char	char_allfromstring, char_dup, char_dup_items, char_dup_times, char_empty, char_eq, char_flush, char_fromfloat, char_frominteger, char_isdigit, char_isletter, char_iswhitespace, char_lowercase, char_pop, char_rot, char_shove, char_stackdepth, char_swap, char_uppercase, char_yank, char_yankdup
string	string_butlast, string_capitalize, string_concat, string_conjchar, string_contains, string_containschar, string_dup, string_dup_items, string_dup_times, string_empty, string_emptystring, string_eq, string_first, string_flush, string_fromboolean, string_fromchar, string_fromfloat, string_frominteger, string_includes, string_indexof, string_indexofchar, string_last, string_length, string_lowercase, string_nth, string_occurrencesofchar, string_parse_to_chars, string_pop, string_removechar, string_replace, string_replacechar, string_replacefirst, string_replacefirstchar, string_rest, string_reverse, string_rot, string_setchar, string_shove, string_sort, string_split, string_stackdepth, string_substring, string_swap, string_take, string_uppercase, string_yank, string_yankdup

[4] Lalejini, A., Ofria, C.: Tag-accessed memory for genetic programming. In: GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 346–347. ACM, Prague, Czech Republic (2019). https://doi.org/10.1145/3319619.3321892

[5] Rosin, C.D.: Stepping stones to inductive synthesis of low-level looping

Table A2 Continuation of Table A1

Data Type	Instructions
vector of integers	vector_integer_butlast, vector_integer_concat, vector_integer_conj, vector_integer_contains, vector_integer_dup, vector_integer_dup_items, vector_integer_dup_times, vector_integer_empty, vector_integer_emptyvector, vector_integer_eq, vector_integer_first, vector_integer_flush, vector_integer_indexof, vector_integer_last, vector_integer_length, vector_integer_nth, vector_integer_occurrencesof, vector_integer_pop, vector_integer_pushall, vector_integer_remove, vector_integer_replace, vector_integer_replacefirst, vector_integer_rest, vector_integer_reverse, vector_integer_rot, vector_integer_set, vector_integer_shove, vector_integer_sort, vector_integer_stackdepth, vector_integer_subvec, vector_integer_swap, vector_integer_take, vector_integer_yank, vector_integer_yankdup
vector of floats	vector_float_butlast, vector_float_concat, vector_float_conj, vector_float_contains, vector_float_dup, vector_float_dup_items, vector_float_dup_times, vector_float_empty, vector_float_emptyvector, vector_float_eq, vector_float_first, vector_float_flush, vector_float_indexof, vector_float_last, vector_float_length, vector_float_nth, vector_float_occurrencesof, vector_float_pop, vector_float_pushall, vector_float_remove, vector_float_replace, vector_float_replacefirst, vector_float_rest, vector_float_reverse, vector_float_rot, vector_float_set, vector_float_shove, vector_float_sort, vector_float_stackdepth, vector_float_subvec, vector_float_swap, vector_float_take, vector_float_yank, vector_float_yankdup

programs. In: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence. AAAI '19, vol. 33. AAAI Press, Palo Alto, California USA (2019)

[6] Lim, J., Yoo, S.: Field report: Applying monte carlo tree search for program synthesis. In: International Symposium on Search Based Software Engineering, pp. 304–310 (2016). Springer

[7] Helmuth, T., Spector, L.: Explaining and exploiting the advantages of down-sampled lexicase selection. In: Artificial Life Conference Proceedings, pp. 341–349. MIT Press, Online (2020). https://doi.org/10.1162/isal_a_00334. https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_00334

[8] Helmuth, T., Spector, L.: Problem-Solving Benefits of Down-Sampled Lexicase Selection. Artificial Life, 1–21 (2021) https://arxiv.org/abs/https://direct.mit.edu/artl/article-pdf/doi/10.1162/artl_a_00341/1960075/artl_a_00341.pdf. https://doi.org/10.1162/artl_a_00341

[9] McDermott, J., White, D.R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., O'Reilly, U.-M.: Genetic programming needs better benchmarks. In: GECCO '12: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 791–798. ACM, Philadelphia, Pennsylvania, USA (2012). https://doi

.org/10.1145/2330163.2330273

[10] White, D.R., Mcdermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.-M., Luke, S.: Better GP benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**(1), 3–29 (2013). https://doi.org/10.1007/s10710-012-9177-2

[11] Woodward, J., Martin, S., Swan, J.: Benchmarks that matter for genetic programming. In: GECCO 2014 4th Workshop on Evolutionary Computation for the Automated Design of Algorithms, pp. 1397–1404. ACM, Vancouver, BC, Canada (2014). https://doi.org/10.1145/2598394.2609875. http://doi.acm.org/10.1145/2598394.2609875

[12] O'Neill, M., Spector, L.: Automatic programming: The open issue? Genetic Programming and Evolvable Machines **21**(1-2), 251–262 (2020). https://doi.org/10.1007/s10710-019-09364-2. Twentieth Anniversary Issue

[13] Helmuth, T., Kelly, P.: PSB2: The second program synthesis benchmark suite. In: 2021 Genetic and Evolutionary Computation Conference. GECCO '21. ACM, Lille, France (2021). https://doi.org/10.1145/3449639.3459285. https://dl.acm.org/doi/10.1145/3449639.3459285

[14] Sobania, D., Schweim, D., Rothlauf, F.: Recent developments in program synthesis with evolutionary algorithms. arXiv (2021) https://arxiv.org/abs/2108.12227 [cs.NE]

[15] Helmuth, T., Spector, L., McPhee, N.F., Shanabrook, S.: Linear genomes for structured programs. In: Genetic Programming Theory and Practice XIV. Genetic and Evolutionary Computation. Springer, Ann Arbor, USA (2016)

[16] Helmuth, T., McPhee, N.F., Pantridge, E., Spector, L.: Improving generalization of evolved programs through automatic simplification. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO '17, pp. 937–944. ACM, Berlin, Germany (2017). https://doi.org/10.1145/3071178.3071330. http://doi.acm.org/10.1145/3071178.3071330

[17] Helmuth, T., McPhee, N.F., Spector, L.: Program synthesis using uniform mutation by addition and deletion. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO '18, pp. 1127–1134. ACM, Kyoto, Japan (2018). https://doi.org/10.1145/3205455.3205603. http://doi.acm.org/10.1145/3205455.3205603

[18] Helmuth, T., Pantridge, E., Woolson, G., Spector, L.: Genetic source sensitivity and transfer learning in genetic programming.

In: Artificial Life Conference Proceedings, pp. 303–311. MIT Press, Online (2020). https://doi.org/10.1162/isal_a_00326. https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_00326

[19] Saini, A.K., Spector, L.: Using modularity metrics as design features to guide evolution in genetic programming. In: Banzhaf, W., Goodman, E., Sheneman, L., Trujillo, L., Worzel, B. (eds.) Genetic Programming Theory and Practice XVII, pp. 165–180. Springer, East Lansing, MI, USA (2019). https://doi.org/10.1007/978-3-030-39958-0_9

[20] Saini, A.K., Spector, L.: Why and when are loops useful in genetic programming? In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion. GECCO '20, pp. 247–248. Association for Computing Machinery, internet (2020). https://doi.org/10.1145/3377929.3389919. https://doi.org/10.1145/3377929.3389919

[21] Pantridge, E., Spector, L.: Code building genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. GECCO '20, pp. 994–1002. Association for Computing Machinery, internet (2020). https://doi.org/10.1145/3377930.3390239. https://arxiv.org/abs/2008.03649

[22] Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In: Vellasco, M. (ed.) 2018 IEEE Congress on Evolutionary Computation (CEC). IEEE, Rio de Janeiro, Brazil (2018). https://doi.org/10.1109/CEC.2018.8477953

[23] Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Extending program synthesis grammars for grammar-guided genetic programming. In: Auger, A., Fonseca, C.M., Lourenco, N., Machado, P., Paquete, L., Whitley, D. (eds.) 15th International Conference on Parallel Problem Solving from Nature. LNCS, vol. 11101, pp. 197–208. Springer, Coimbra, Portugal (2018). https://doi.org/10.1007/978-3-319-99253-2_16. https://www.springer.com/gp/book/9783319992587

[24] Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Towards effective semantic operators for program synthesis in genetic programming. In: GECCO '18: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1119–1126. ACM, Kyoto, Japan (2018). https://doi.org/10.1145/3205455.3205592

[25] Kelly, J., Hemberg, E., O'Reilly, U.-M.: Improving genetic programming with novel exploration - exploitation control. In: Sekanina, L., Hu, T., Lourenço, N., Richter, H., García-Sánchez, P. (eds.) EuroGP 2019: Proceedings of the 22nd European Conference on Genetic Programming, pp. 64–80. Springer, Leipzig, Germany (2019)

[26] O'Neill, M., Brabazon, A.: Mutational robustness and structural complexity in grammatical evolution. In: Coello, C.A.C. (ed.) 2019 IEEE Congress on Evolutionary Computation, CEC 2019, pp. 1338–1344. IEEE Press, Wellington, New Zealand (2019). https://doi.org/10.1109/CEC.2019.8790010. IEEE Computational Intelligence Society

[27] Sobania, D., Rothlauf, F.: Challenges of program synthesis with grammatical evolution. In: Hu, T., Lourenco, N., Medvet, E. (eds.) EuroGP 2020: Proceedings of the 23rd European Conference on Genetic Programming. LNCS, vol. 12101, pp. 211–227. Springer, Seville, Spain (2020). https://doi.org/10.1007/978-3-030-44094-7_14

[28] Lynch, D., McDermott, J., O'Neill, M.: Program synthesis in a continuous space using grammars and variational autoencoders. In: Baeck, T., Preuss, M., Deutz, A., Wang2, H., Doerr, C., Emmerich, M., Trautmann, H. (eds.) 16th International Conference on Parallel Problem Solving from Nature, Part II. LNCS, vol. 12270, pp. 33–47. Springer, Leiden, Holland (2020). https://doi.org/10.1007/978-3-030-58115-2_3

[29] Hernandez, J.G., Lalejini, A., Dolson, E., Ofria, C.: Random subsampling improves performance in lexicase selection. In: GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 2028–2031. ACM, Prague, Czech Republic (2019). https://doi.org/10.1145/3319619.3326900

[30] Ferguson, A.J., Hernandez, J.G., Junghans, D., Lalejini, A., Dolson, E., Ofria, C.: Characterizing the effects of random subsampling and dilution on lexicase selection. In: Banzhaf, W., Goodman, E., Sheneman, L., Trujillo, L., Worzel, B. (eds.) Genetic Programming Theory and Practice XVII, East Lansing, MI, USA (2019)

[31] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. SIGPLAN Not. **46**(1), 317–330 (2011). https://doi.org/10.1145/1925844.1926423

[32] Katayama, S.: Recent improvements of MagicHaskeller. In: Approaches and Applications of Inductive Programming. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_9. http://dx.doi.org/10.1007/978-3-642-11931-6_9

[33] Pantridge, E., Helmuth, T., McPhee, N.F., Spector, L.: On the difficulty of benchmarking inductive program synthesis methods. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. GECCO '17, pp. 1589–1596. ACM, Berlin, Germany (2017). https://doi.org/10.1145/3067695.3082533. http://doi.acm.org/10.1145/3067695.3082533

[34] Helmuth, T., Kelly, P.: General Program Synthesis Benchmark Suite Datasets. https://github.com/thelmuth/program-synthesis-benchmark-datasets

[35] Wastl, E.: Advent of Code: Not Quite Lisp. Accessed: 2020-01-20. https://adventofcode.com/2015/day/1

[36] g964: Code Wars: Bouncing Balls. Accessed: 2020-01-20. https://www.codewars.com/kata/5544c7a5cb454edb3c000047

[37] dnolan: Code Wars: Ten-Pin Bowling. Accessed: 2020-01-20. https://www.codewars.com/kata/5531abe4855bcc8d1f00004c/javascript

[38] jhoffner: Code Wars: Convert String to Camel Case. Accessed: 2020-01-20. https://www.codewars.com/kata/517abf86da9663f1d2000003

[39] Euler, P.: Project Euler: Coin Sums. Accessed: 2020-01-20. https://projecteuler.net/problem=31

[40] myjinxin2015: Code Wars: Fastest Code: Half It IV. Accessed: 2020-01-20. https://www.codewars.com/kata/5719b28964a584476500057d

[41] Euler, P.: Project Euler: Dice Game. Accessed: 2020-01-20. https://projecteuler.net/problem=205

[42] Wastl, E.: Advent of Code: Report Repair. Accessed: 2020-01-20. https://adventofcode.com/2020/day/1

[43] stephenyu: Code Wars: Fizz Buzz. Accessed: 2020-01-20. https://www.codewars.com/kata/5300901726d12b80e8000498

[44] Wastl, E.: Advent of Code: The Tyranny of the Rocket Empire. Accessed: 2020-01-20. https://adventofcode.com/2019/day/1

[45] RVdeKoning: Code Wars: Greatest Common Divisor. Accessed: 2020-01-20. https://www.codewars.com/kata/5500d54c2ebe0a8e8a0003fd/python

[46] smile67: Code Wars: Text Search. Accessed: 2020-01-20. https://www.codewars.com/kata/56b78faebd06e61870001191

[47] MrZizoScream: Code Wars: Array Leaders. Accessed: 2020-01-20. https://www.codewars.com/kata/5a651865fd56cb55760000e0

[48] mcclaskc: Code Wars: Validate Credit Card Number. Accessed: 2020-01-20. https://www.codewars.com/kata/5418a1dd6d8216e18a0012b2

[49] Shivo: Code Wars: Get the Middle Character. Accessed: 2020-01-20. https:

//www.codewars.com/kata/56747fd5cb988479af000028

[50] Wastl, E.: Advent of Code: Inverse Captcha. Accessed: 2020-01-20. https://adventofcode.com/2017/day/1

[51] rb50: Code Wars: Shopping List. Accessed: 2020-01-20. https://www.codewars.com/kata/596266482f9add20f70001fc

[52] KenKamau: Code Wars: The Boolean Order. Accessed: 2020-01-20. https://www.codewars.com/kata/59eb1e4a0863c7ff7e000008

[53] xDranik: Code Wars: Stop gninnipS My sdroW! Accessed: 2020-01-20. https://www.codewars.com/kata/5264d2b162488dc400000001

[54] MysteriousMagenta: Code Wars: Square Every Digit. Accessed: 2020-01-20. https://www.codewars.com/kata/546e2562b03326a88e000020

[55] jacobb: Code Wars: Simple Substitution Cipher Helper. Accessed: 2020-01-20. https://www.codewars.com/kata/52eb114b2d55f0e69800078d

[56] StephenLastname2: Code Wars: Distance Between Two Points. Accessed: 2020-01-20. https://www.codewars.com/kata/5a0b72484bebaefe60001867

[57] Helmuth, T., Kelly, P.: PSB2: The Second Program Synthesis Benchmark Suite. Zenodo (2021). https://doi.org/10.5281/zenodo.4678739

[58] Olson, R.S., La Cava, W., Orzechowski, P., Urbanowicz, R.J., Moore, J.H.: Pmlb: a large benchmark suite for machine learning evaluation and comparison. BioData Mining **10**(1), 36 (2017). https://doi.org/10.1186/s13040-017-0154-4

[59] Le, T.T., La Cava, W., Romano, J.D., Gregg, J.T., Goldberg, D.J., Chakraborty, P., Ray, N.L., Himmelstein, D., Fu, W., Moore, J.H.: Pmlb v1.0: an open source dataset collection for benchmarking machine learning methods. arXiv preprint arXiv:2012.00058 (2020)

[60] Dua, D., Graff, C.: UCI Machine Learning Repository (2017). http://archive.ics.uci.edu/ml

[61] Bartz-Beielstein, T., Doerr, C., van den Berg, D., Bossek, J., Chandrasekaran, S., Eftimov, T., Fischbach, A., Kerschke, P., Cava, W.L., Lopez-Ibanez, M., Malan, K.M., Moore, J.H., Naujoks, B., Orzechowski, P., Volz, V., Wagner, M., Weise, T.: Benchmarking in optimization: Best practice and open issues. arXiv (2020) https://arxiv.org/abs/2007.03488 [cs.NE]

[62] Spector, L., Robinson, A.: Genetic programming and autoconstructive

evolution with the push programming language. Genetic Programming and Evolvable Machines **3**(1), 7–40 (2002). https://doi.org/10.1023/A:1014538503543

[63] Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: GECCO 2005: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, vol. 2, pp. 1689–1696. ACM Press, Washington DC, USA (2005). https://doi.org/10.1145/1068009.1068292

[64] Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation **19**(5), 630–643 (2015). https://doi.org/10.1109/TEVC.2014.2362729

[65] Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report. In: McClymont, K., Keedwell, E. (eds.) 1st Workshop on Understanding Problems (GECCO-UP), pp. 401–408. ACM, Philadelphia, Pennsylvania, USA (2012). https://doi.org/10.1145/2330784.2330846. http://hampshire.edu/lspector/pubs/wk09p4-spector.pdf

[66] Robinson, A.: Genetic programming: Theory, implementation, and the evolution of unconstrained solutions. Division III thesis, Hampshire College (May 2001). http://hampshire.edu/lspector/robinson-div3.pdf

[67] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, pp. 317–330. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926423. http://doi.acm.org/10.1145/1926385.1926423

[68] Menon, A.K., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.T.: A Machine Learning Framework for Programming by Example. ICML, 9 (2013)

[69] Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: Learning to write programs. In: ICLR (2017)

[70] Bladek, I., Krawiec, K.: Evolutionary program sketching. In: Castelli, M., McDermott, J., Sekanina, L. (eds.) EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming. LNCS, vol. 10196, pp. 3–18. Springer, Amsterdam (2017). https://doi.org/10.1007/978-3-319-55696-3_1. http://repozytorium.put.poznan.pl/publication/495662

[71] Zohar, A., Wolf, L.: Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. NIPS (2018). arXiv: 1809.04682. Accessed

2021-10-10

[72] Gulwani, S., Pathak, K., Radhakrishna, A., Tiwari, A., Udupa, A.: Quantitative programming by examples. arXiv (2019) https://arxiv.org/abs/1909.05964 [cs.PL]

[73] Cropper, A., Morel, R.: Learning programs by learning from failures. Machine Learning **110**(4), 801–856 (2021). https://doi.org/10.1007/s10994-020-05934-z. Accessed 2021-10-09

[74] Polosukhin, I., Skidanov, A.: Neural program search: Solving programming tasks from description and examples. arXiv (2018) https://arxiv.org/abs/1802.04335 [cs.AI]

[75] Bednarek, J., Piaskowski, K., Krawiec, K.: Ain't Nobody Got Time for Coding: Structure-Aware Program Synthesis from Natural Language. arXiv, 12 (2019)

[76] Rahmani, K., Raza, M., Gulwani, S., Le, V., Morris, D., Radhakrishna, A., Soares, G., Tiwari, A.: Multi-modal Program Inference: a Marriage of Pre-trained Language Models and Component-based Synthesis. arXiv:2109.02445 [cs] (2021). arXiv: 2109.02445. Accessed 2021-09-15

[77] Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W., Nichol, A., Babuschkin, I., Balaji, S., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating Large Language Models Trained on Code. arXiv (2021)

[78] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program Synthesis with Large Language Models. arXiv (2021). arXiv: 2108.07732. Accessed 2021-08-21

[79] Solar-Lezama, A.: Program sketching. International Journal on Software Tools for Technology Transfer **15**(5), 475–495 (2013)

[80] Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design, pp. 1–8 (2013). https://doi.org/10.1109/FMCAD.2013.6679385

[81] Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Commun. ACM **61**(12), 84–93 (2018). https://doi.org/10.1145/3208071

[82] Lee, W., Heo, K., Alur, R., Naik, M.: Accelerating search-based program synthesis using learned probabilistic models, 436–449 (2018). https://doi.org/10.1145/3192366.3192410

[83] Welsch, T., Kurlin, V.: Synthesis through unification genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. GECCO '20, pp. 1029–1036. Association for Computing Machinery, internet (2020). https://doi.org/10.1145/3377930.3390208. https://doi.org/10.1145/3377930.3390208