Program synthesis for integer sequence generation

Natasha Butt
University of Amsterdam

Auke Wiggers
Qualcomm AI Research*

Taco Cohen
Qualcomm AI Research*

Max Welling
University of Amsterdam

Abstract

Recent advances in program synthesis have shown success with methods that employ supervised learning on synthetic data generated from domain specific languages (DSLs). In this work, we propose an algorithm for program synthesis that extends these methods. It uses transfer learning from pre-trained language models, and uses a policy improvement operator based on policy-guided search. This hybrid approach combats the challenges of searching a large language space with sparse rewards. We show its effectiveness on the task of integer sequence generation, a special case of programming-by-examples with fixed inputs. Our results demonstrate that the inclusion of our policy improvement operator leads to a 32% increase in performance compared to a supervised baseline method.

1 Introduction

Program synthesis refers to a class of techniques that generate programs given a specification of semantic and syntactic requirements. In programming-by-examples, this specification is a set of input-output examples [14, 15]. In this work, we consider a specific setting: program synthesis for integer sequence generation. The goal is to find a program that outputs a given integer sequence, e.g. $\{4, 9, 16, \ldots, 100\}$. This task is a form of symbolic regression, as our programs contain a representation of the general formula for computing the nth term of a sequence. This task may also be seen as list processing on a fixed input.

Programs can be constructed as a sequence of operations in a programming language. Searching for programs that produce a desired output then requires solving the combinatorial optimization problem of finding the right order of operations, which poses multiple challenges. First, the space of possible programs is large, and we encounter combinatorial explosion as program length and language size increase. Second, it is difficult to obtain a signal that indicates whether the search is moving in the right direction as it is rare to find a program that produces the correct output on execution.

Our method employs three techniques to overcome these challenges, the combination of which results in a powerful program synthesis algorithm: 1) transfer learning from pre-trained language models, 2) synthetic data generation from a DSL, and 3) use of a policy improvement operator that iterates between policy-guided search and training. This method achieves a 32% increase in performance on the task of integer sequence generation compared to the supervised baseline method. We expect that this method will generalize to other program synthesis tasks and combinatorial optimization problems in future work.

*Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on Math-AI.

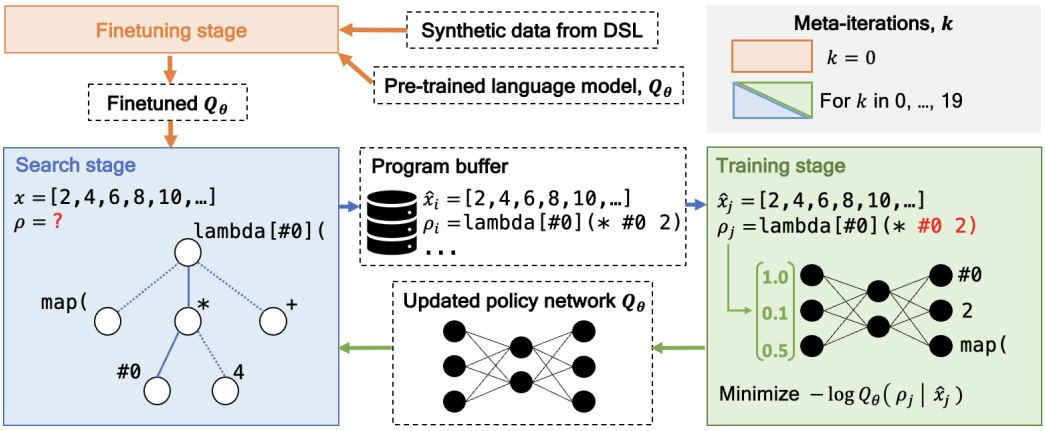

Figure 1: Overview of our method. In the finetuning stage, the pre-trained language model Q_θ is trained to predict programs in an autoregressive manner using synthetic programs and sequences sampled from the DSL. The finetuned model represents a task-conditional policy network. In the search stage, new programs ρ are found for the task x, and added to the program buffer along with the sequences generated on program execution, $\hat{x}$. In the training stage, Q_θ is trained further on programs and sequences sampled from the program buffer.

2 Method

We describe our method in Algorithm 1, and visualize it schematically in Figure 1. The inputs are 1) a language model acting as a policy Q_θ, 2) our DSL, 3) the target integer sequences, and 4) a prioritized program buffer. The policy network is initialized based on a pre-trained language model. We first finetune the pretrained policy network on synthetic programs and sequences, randomly generated from the DSL, using transfer learning. We then iterate through a two-stage process: the search stage finds new programs ρ (guided by the policy Q_θ), and policy training improves the policy (based on programs found in search). By alternating these two stages over $n_k = 20$ meta-iterations, we expect that over time, the search stage will find more useful programs, and the policy will become better at guiding the search. We describe these steps in more detail below.

Algorithm 1 program_synthesis

Require: Pre-trained model: Q_θ, DSL: $\mathcal{L}$, target integer sequences: X, program buffer initialised with an empty set of program and program output pairs: $b = \{\}$
Ensure: Finetuned policy network: $Q_{\hat{\theta}}$, updated program buffer: $b = \{\rho_i, \hat{x}_i\}_{i=1}^{n}$

```
k ← 0
{ρ̃_i, x̃_i}_{i=1}^m ← sample_and_execute(L)      ▷ Sample programs and execute for sequences
Q_θ̂ ← train(Q_θ, {x̃_i, ρ̃_i}_{i=1}^m)             ▷ Finetune policy network on synthetic data
while k < 20 do
    b ← search(X, Q_θ̂, b)                        ▷ Search for programs that generate sequences
    Q_θ̂ ← train(Q_θ̂, b)                          ▷ Finetune policy network on programs found in search
    k ← k + 1
end while
```

Transfer learning with pre-trained language models The pre-trained language model that we use as input for our algorithm is the CodeTrans model for program synthesis [10]. This is a transformer based on T5-small [31], trained on unsupervised datasets in various programming languages (code only) and fine-tuned on list processing program synthesis tasks from the AlgoLISP dataset [27] (code and text prompt). The encoder takes a textual prompt as input, representing a partial specification of a program, and outputs an embedded representation. The decoder takes this as input and outputs code in a LISP-inspired DSL.

Program:	lambda[#0](map(lambda[#1](*(/(*(#1 8) 9) #1)) #0)
Python:	lambda x: map(lambda y: round(y*8 / 9) * y, x)
Output:	1,4,9,16,20,30,42,56,72,90,110,132,156,168,...

Table 1: Example synthetic program, its Python equivalent, and program output.

Model transfer is possible as the CodeTrans model and our policy network take inputs that represent a partial program specification; In the CodeTrans model, this is a text prompt and, in our case, this is the integer sequence that the program should generate. There are also similarities between our DSL and the languages that the CodeTrans decoder is trained on. Thus, we expect the decoder to have already learnt about 'how to code' given a partial specification represented by the encoder embedding.

Synthetic data generation from Domain Specific Language We create a Domain Specific Language (DSL). A program in our DSL is defined in the same way as the LISP-inspired DSL of Polosukhin et al. [27] and their AlgoLISP dataset. Each program is a set of arguments (with arguments defined by name and type) and a program tree. Each node in the tree has a symbol type: constant, argument, function call, function, or lambda. Our DSL differs from that of Polosukhin et al. [27] in some primitive constants and functions. Our primitive constants are the integers 0-9. Our primitive functions are: +, -, *, /, %, **, `sqrt`, `min`, `max`, `reduce`, `filter`, `map`, ==, !, ||, &, $<$, $>$.

We construct synthetic programs in an iterative manner. Starting from a partial program containing the `lambda` term, we sample a valid next term from the DSL uniformly at random. Validity is described by the type system of the DSL. We append the new term to the partial program, and repeat this procedure until we reach the end of the program, or we reach a pre-defined maximum program length. Once the maximum program length is reached, we only allow primitives to be added.

We add three constraints to ensure that the generated programs are realistic and varied. First, we reject programs if their output sequence contains numbers larger than 10^{50}. Second, if the output sequence contains the same number for more than 90% of the total length, we reject it with 50% probability. Third, we only accept a program if the integer sequence it produces is unique. An example program in our DSL is given in Table 1.

Policy improvement: search Each meta-iteration of policy improvement consists of a search stage and a training stage. First, for every target sequence in our test set, we search for programs that output the sequence. This procedure can be best thought of as a tree-search, where each node corresponds to a partial state of a program. The policy network guides this tree-search by outputting a distribution over tokens, given the embedding of the target integer sequence and the partial program. For consistency with related work [6], we only give the policy network access to the first 25 numbers of the sequence. We use beam search, as this results in more diverse programs than greedy search. We maintain 10 beams and sequentially add the most likely next token to our beams, keeping only the 10 most likely sequences.

We add programs found during search to the buffer only if they output a sequence of integers with a length longer than 2. Additionally, when searching for a program that outputs a target integer sequence x_i, we may instead find a program that outputs another sequence x_j that does belong to our set of target integer sequences X. Following related work [13], we therefore implement a version of hindsight replay [1]. For any found program ρ_i, the output $\hat{x}_i$ is compared to all the target integer sequences. If numbers 26 to 35 are equal, the sequences are considered equivalent, and the program is added to the program buffer with an indicator that it solves task x_j.

Policy improvement: training In the training stage, we draw training examples from the program buffer. To encourage sampling of useful programs, we assign priorities based on heuristics. The priority is the percentage of the next ten sequence terms correctly generated by executing the program, multiplied by a time decay of 0.5^t, where t is the number of meta-iterations since the entry was added. This decay incentives more recent programs to be sampled more often.

For each program in our buffer, we assign the priority as the maximum priority of corresponding entries (as the same program may be added as a solution for multiple sequences during search). We draw 10,000 programs from the normalised probabilities defined by our maximum priorities. Our policy network is trained for 200 epochs on these programs and then it is passed back to search.

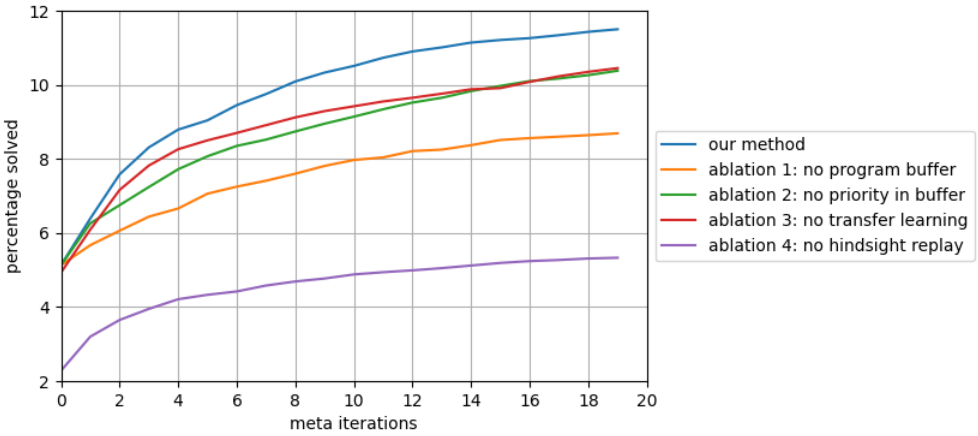

Figure 2: Percentage of correct integer sequences for each method and meta-iteration.

3 Experiments

We sample 10,000 synthetic program and integer sequence pairs from the DSL, 9,000 for training and 1,000 for validation. The On-line Encyclopedia of Integer Sequences (OEIS) [16] contains $\approx 350,000$ integer sequences. We filter the dataset by keyword 'easy' ($\approx 80,000$) and take the first 10,000 sequence with length greater than 35 as our test set. Integer sequences are tokenized before being passed as input to the language model. We tokenize each integer seperately, and add a symbol in between integers that acts as delimiter and represents the sign of an integer. Integers larger than 10^9 are tokenized with 'NaN'.

We first finetuned the CodeTrans model on our synthetic data by training for 300 epochs with batch size 10, Adam optimizer [18] and learning rate 10^{-4}, taking 20 hours on one NVIDIA GeForce GTX 1080 Ti GPU. For model selection, we greedily decode the embeddings of the validation sequences, and our validation metric is the percentage of correct terms 26 to 35 of the sequences generated on execution of the decoded programs.

As baseline, in ablation 1, we omit the program buffer and instead train on 10,000 new unconstrained program samples from our DSL at each meta-iteration. This is a small-scale proxy for the work of D'Ascoli et al. [6]. We perform three further ablations: 2) omitting the priority from the program buffer and instead sampling under a uniform distribution 3) randomly initialising the CodeTrans model before finetuning on synthetic data 4) omitting hindsight replay. Sequences are 'solved' if a found program outputs terms 26 to 35 correctly given the first 25 terms, following D'Ascoli et al. [6].

Figure 2 shows that our method outperforms the baseline (ablation 1) by 32%, from 8.7% solved to 11.5% solved after 20 meta-iterations, demonstrating the power of policy improvement. The inclusion of prioritized sampling from the buffer results in an 11% increase in performance, from 10.4% solved to 11.5% solved after 20 meta-iterations, over uniform sampling from the program buffer. Further, transfer learning using the pre-trained CodeTrans model offers some improvement compared to ablation 3 with random initialization. Omitting hindsight replay in ablation 4 results in a 46% decrease in performance, from 11.5% solved to 5.3% solved after 20 meta-iterations.

These preliminary results highlight the benefit of using our method with transfer learning and policy improvement over supervised learning with synthetic data alone. Further work will look at implementing this method on a larger scale to make direct comparisons to the findings of D'Ascoli et al. [6] and Gauthier et al. [13]. D'Ascoli et al. train on 5 million synthetic examples whereas we have finetuned on on 200,000 training examples in total after 20 meta-iterations. Further, Gauthier et al. implement their method on the entire OEIS set, whereas we use the 'easy' subset of around 3% of the size consistent with D'Ascoli et al. As a result, future work will look at implementing our method with a larger test set, finetuning on more sequences and training for longer.

4 Related Work

The combination of search and learning is a powerful tool that has been used in many contexts, ranging from theorem proving [20, 28, 29, 30] to game-playing [34] to general combinatorial optimization [19, 21] and maths problems [35]. In these settings, we generally observe large search spaces and sparse rewards. The feedback loop between search and learning provides a solution to overcome these challenges.

In the context of program synthesis and program optimization, supervised learning of large (policy) models has often been employed as well [2, 5, 8, 17, 22, 24, 25, 33]. A common approach is to define a DSL, then use the resulting search-space to build programs [4, 7, 26, 27, 23, 36, 11]. We follow a similar approach in this work.

In program synthesis for list processing tasks, DeepCoder [3] trains a network to predict properties of the desired program in order to augment search. Further work includes DreamCoder [9], which introduces policy improvement via iteration of search, training and abstraction phases. The addition of the abstraction phase aggregates similar parts of programs into repeatable modules, allowing reduced program length, although it requires domain knowledge of the used language. Moreover, in program synthesis for matrix multiplication, AlphaTensor [11] applies policy improvement by translating the problem to a single player game and iteratively training the policy on played games and synthetic demonstrations.

For the specific problem of integer sequence program synthesis, D'Ascoli et al. [6] demonstrate that training a policy on synthetic data alone followed by beam search is effective. Specifically, they find programs that, given the first 25 terms in a sequence, generate terms 26 to 35 correctly for 21% of sequences. We extend this work with our policy improvement operator, which prevents domain mismatch resulting from training on synthetic data [32]. Direct comparisons are challenging because D'Ascoli et al. have access to significantly more resources (training for 250 epochs on 16 GPUs), but our small scale baseline acts as a proxy for their results.

Recent work that also considers policy improvement for integer sequences is the work of Gauthier et al. [12, 13]. This work uses tree-based neural networks to parse programs, and updates the network in meta-iterations. In their first meta-iteration, Gauthier et al. solve less than 0.1% of the tasks with random search, highlighting the bootstrapping challenge associated with finding correct programs. Where they start search and training from scratch, which arguably makes the method more flexible, we show that a language model trained on different programming languages can be used to initialise search and training in the first meta-iteration. Additionally, the choice to represent programs as trees is intuitive, but may be limiting as it requires more domain knowledge than parsing programs as text directly.

5 Conclusion

In this work, we propose an algorithm for program synthesis based on three core techniques: transfer learning with pre-trained language models, supervised learning on synthetic data and policy improvement via iterating policy-guided search and training. These three techniques are known to scale well with data and compute. Further, they are applicable in many settings. Here, we apply them to the integer sequence generation problem, where the goal is to synthesize a program that generates a target integer sequence. By using a language model pre-trained for program synthesis tasks, and finetuning on synthetic programs, we overcome the challenging bootstrapping stage as our search does not start from scratch. Our approach uses a policy improvement operator by alternating search and training, and we show that this outperforms a supervised learning baseline by 32%. Future work will look at implementing this method on a larger scale while targeting more program synthesis and combinatorial optimization tasks.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*. OpenReview. net, 2017.

[4] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[6] Stéphane d'Ascoli, Pierre-Alexandre Kamienny, Guillaume Lample, and François Charton. Deep symbolic regression for recurrent sequences. *CoRR*, abs/2201.04600, 2022.

[7] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[8] Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, et al. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.

[9] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. *CoRR*, abs/2006.08381, 2020.

[10] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing, 2021.

[11] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Barekatain Mohammadamin, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610, 2022.

[12] Thibault Gauthier. Program synthesis for the oeis, 2022.

[13] Thibault Gauthier and Josef Urban. Learning program synthesis for integer sequences from scratch, 2022.

[14] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[15] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[16] OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2022.

[17] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[19] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems!, 2018.

[20] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022.

[21] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Hui Chen, Torbjørn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: enabling self-play reinforcement learning for bin packing. 2019.

[22] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *SIGPLAN Not.*, 53(4):436–449, jun 2018.

[23] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

[24] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, and Charles Sutton. BUSTLE: Bottom-up program-Synthesis Through Learning-guided Exploration. *ArXiv*, abs/2007.14381, 2021.

[25] Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. *CoRR*, abs/2002.09030, 2020.

[26] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

[27] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *CoRR*, abs/1802.04335, 2018.

[28] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *CoRR*, abs/2202.01344, 2022.

[29] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[30] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. In *International Conference on Learning Representations*, 2020.

[31] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[32] Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. *CoRR*, abs/1912.12345, 2019.

[33] Alexander G Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. Learning to superoptimize real-world programs. In *Deep Learning for Code Workshop*, 2022.

[34] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[35] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022.

[36] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems*, 31, 2018.