Xinyun Chen - Research Statement

My long-term research goal is to democratize programming, so that everyone can benefit from the advancement of computing technologies. Nowadays, programming is ubiquitous not only among professional software developers, but also for general computer users. However, gaining programming expertise is time-consuming and challenging. Therefore, program synthesis has many applications, where the computer automatically synthesizes the program from the specification, i.e., the description of the required program functionalities. Program synthesis transforms the way we interact with computers, which makes programming more friendly and accessible to general users, facilitates data scientists to process data, and improves the programming efficiency of software developers.

Classic program synthesis techniques are largely based on heuristic-guided search and rule-based generation. These hand-engineered systems require a lot of manual effort to tune the search heuristics and synthesis rules for different applications, and they are not capable of handling program specifications that are noisy and less well-defined, such as natural language descriptions. On the other hand, recent advancements in deep learning have shown impressive performance in a variety of areas. With abundant open-source projects available online, deep neural networks have become a great fit for representing different specification formats and efficiently learning the synthesis rules from data. Eventually, learning-based techniques are necessary for broadening the impact of program synthesis.

Despite the recent progress of program synthesis, including learning-based approaches, prior works still suffer from limited **complexity** and **generalizability**, i.e., the predicted programs tend to become inconsistent with the specification when the specification and its corresponding program are long and complicated. Meanwhile, understanding **heterogeneous specification formats** remains a challenge for the real-world deployment of program synthesizers.

I am among the first researchers focusing on **learning-based program synthesis**, and I have developed deep learning-based techniques towards addressing the aforementioned challenges and demonstrating the real-world impact of this new programming paradigm. Meanwhile, my research also aims to address the core challenges of artificial intelligence and machine learning regarding generalization, compositionality, interpretability, and reasoning. Specifically, I have designed **neural-symbolic frameworks** that interleave neural and symbolic modules, which learn to produce problem solutions represented as programs.

In terms of program synthesis applications, I have proposed learning-based program synthesis approaches to synthesize programs from various types of specifications, including input-output examples [8, 10, 16], natural language descriptions [1, 5, 9], and reference programs [7, 11, 14, 20]. Our **SpreadsheetCoder** model [9], which predicts spreadsheet formulas from the tabular context, was integrated into Google Sheets [1]. The formula suggestion feature could potentially benefit hundreds of millions of users, and makes data analysis easier and more efficient. Meanwhile, our **execution-guided synthesis** technique [8, 10, 19] brings significant performance gains for synthesizing more complex programs.

On the other hand, my research on learning-based program synthesis also introduces a new methodology to reason over data. Existing deep neural networks have been primarily designed to learn what to predict, instead of the rationale behind the predictions. As a result, despite the remarkable success of deep neural networks in various applications, they are insufficient for more complex reasoning beyond superficial pattern matching, such as numerical calculation and logical reasoning. Furthermore, deep neural networks have exposed limitations in generalization, even if the test input only slightly deviates from the training distribution. Facing these challenges of **reasoning** and **generalization**, I have developed neural-symbolic techniques that empower neural networks with the ability to synthesize programs that represent the reasoning process. By integrating the symbolic component into deep neural networks, our **neural-symbolic reader** [3] demonstrates decent performance on challenging numerical reasoning over text, which is not naturally achievable even with massive pre-training. Meanwhile, our **neural-symbolic stack machines** [2, 6] learn execution traces that reveal the compositional rules for language understanding, which achieve full generalization to unseen test cases.

Going forward, I am excited about continuing my research agenda to enable more program synthesis applications with deep learning, towards the ultimate goal of end-user programming that allows everyone to be involved in the software development process. I am also passionate about pushing forward the reasoning capability and generalizability of neural networks with neural-symbolic techniques. In addition, I plan to incorporate my expertise in **adversarial machine learning** [4, 18] to further investigate the vulnerabilities and biases of neural networks.

1 Learning-based Program Synthesis for Real-world Applications

For program synthesis applications, I have developed neural network architectures that learn structured representations of the input specifications and output programs, which better capture the syntactic and semantic characteristics of the programming languages in consideration. My works demonstrate the importance of structured representation learning for a wide range of applications, including spreadsheet formula synthesis [9], visualization code synthesis [1], program translation [7], code optimization [11, 20], and program decompilation [14].

Spreadsheet formula prediction. Spreadsheets are ubiquitous for data storage, with hundreds of millions of users. Helping users write formulas in spreadsheets is a powerful feature for data analysis. Systems such as FlashFill [15] and RobustFill [13] have been developed to synthesize programs for string transformation tasks in spreadsheets, where the users provide a few input-output examples as the specification. However, the domain-specific language designed in these works only supports a subset of spreadsheet functions for string processing, whereas spreadsheet languages also support a wide range of numerical calculation functions. In addition, the input-output specification format does not take the spreadsheet table structure into account, where cells in different rows could be correlated.

[1] `https://ai.googleblog.com/2021/10/predicting-spreadsheet-formulas-from.html`.

To address the above limitations, we developed SpreadsheetCoder [9], a neural network architecture that encodes the spreadsheet context in its table format for formula prediction. Specifically, SpreadsheetCoder includes a row-based encoder and a column-based encoder to model both row-oriented and column-oriented tabular structures. The decoder generates the spreadsheet formula in a two-stage process, where it first predicts a formula sketch (consisting of formula operators without cell ranges), and then generates the corresponding ranges using cell addresses relative to the target cell.

On our large-scale benchmark of real-world Google Sheets, SpreadsheetCoder achieves over 42% top-1 full-formula accuracy and 57% top-1 formula-sketch accuracy, both of which we find high enough to be practically useful. With various ablation experiments, we demonstrate that modeling the structure of the spreadsheet context is crucial for obtaining good performance. **SpreadsheetCoder has been integrated into Google Sheets** to support the formula suggestion feature, showing the power of learning-based program synthesis in real products.

Structured program representations. In SpreadsheetCoder work, we demonstrate the importance of modeling the structure of the input specification. In my work on other program synthesis applications, I also propose neural network architectures to represent the program structures, which improves the complexity of programs that can be correctly generated. Compared to natural languages, a desirable property of programming languages is that each program has an unambiguous parse tree, which could be leveraged to better understand the program structure. In our work on program translation [7], we propose a tree-to-tree neural network that learns the alignment of parse trees in source and target programming languages. We empirically demonstrate that our tree-to-tree model not only outperforms other deep neural networks that do not fully utilize the parse trees with structured information of programs, but also outperforms existing rule-based program translation systems and statistical translation techniques on real-world projects. Besides the translation among high-level programming languages, we also show that modeling the code structure yields a large performance gain for program decompilation [14], especially when the input assembly code is long and corresponds to a high-level program with sophisticated control flows.

2 Execution-Guided Program Synthesis

Most existing learning-based program synthesizers directly utilize the autoregressive decoder architecture designed for natural language modeling, which generates the program as a token sequence. Although such approaches achieve a reasonable performance on synthesizing short and straight-line programs in domain-specific languages, the performance degrades dramatically for programming languages that support more complicated control flow constructs, such as loops and conditionals. I hypothesize that one drawback of the standard decoder design is that it does not effectively model the program semantic information. In particular, the program execution states are not utilized. To learn better program representations, I have developed execution-guided synthesis techniques to incorporate both partial program execution states and full program execution results.

Modeling partial program execution for program synthesis. The principle of execution-guided synthesis is to view the program execution as a sequence of manipulations to transform a program input into the corresponding output. From this perspective, when a partial program is synthesized, we can obtain the intermediate execution state, which explicitly reveals the synthesis progress, and guides the followup program generation process to move on to reach the target program output. When an interpreter is available to obtain the execution states of partial programs, we demonstrate that feeding the execution states as the synthesizer input significantly improves the synthesis performance [8]. For programming languages that do not support partial program execution, such as C, we further show that we can learn a neural executor to approximate the partial program execution states, which still provides remarkable performance gain [10].

Utilizing full program execution for program improvement. Incorporating partial program execution states improves the synthesis performance, but it does not fully resolve all prediction errors. From another point of view, even expert programmers might not be able to always write the correct program in one shot. Instead, human programmers would go through the code and fix some program fragments after observing wrong program outputs. Inspired by the trial-and-error human coding procedure, we proposed to incorporate a neural program repair component into the program synthesis framework, which learns to iteratively debug the predicted code according to its execution results [16]. The neural program repair component notably improves the synthesis performance, while requiring a smaller sample size for program decoding. We have also adapted this high-level idea of iterative program improvement to other applications, including code optimization [11, 20] and program decompilation [14].

3 Neural-Symbolic Reasoning for Language Understanding

Besides developing learning techniques for program synthesis applications, another line of my research introduces program synthesis as a new learning formulation. I have proposed neural-symbolic frameworks that integrate symbolic modules into neural networks, which allows the neural network to compose and execute symbolic operators to represent its knowledge of the data. In particular, our neural-symbolic models learn to interpret and reason over complex text, comprehend grammar rules that reveal the compositionality in languages, and generalize the knowledge to new inputs.

Neural-symbolic reader for discrete reasoning. Reading comprehension is a popular task that measures the ability to answer questions grounded on a passage, where recent neural language models have surpassed human performance on some benchmarks. However, progress has so far been mostly limited to extractive question answering, in which the answer is a single span

from the text. When solving the questions requires discrete reasoning, such as counting, sorting, and numerical calculation, the state-of-the-art language models achieve very low accuracy.

Instead of directly generating the final answer, we designed the neural-symbolic reader [3] that synthesizes the program in a domain-specific language, which produces the answer after execution. To train the model from weak supervision where no manual program annotation is available, we further designed a training algorithm to iteratively search for candidate programs and filter out spurious ones (i.e., programs producing the correct answers for wrong rationales). The neural-symbolic reader achieves the state-of-the-art on several benchmarks that measure the model capability of numerical computation and multi-hop inference, and outperforms pure pre-trained language models (e.g., BERT and GPT-3) by over 40%.

Neural-symbolic stack machine for compositional generalization. Humans have an exceptional capability of compositional reasoning. Given the basic components and a few demonstrations of their combinations, a person could effectively capture the underlying compositional rules, and generalize the knowledge to novel combinations. In contrast, deep neural networks typically lack such generalization abilities; e.g., their accuracy could drop to 0% when tested on longer inputs than training samples [6, 17]. We observed that the main obstacle is because neural networks have great expressiveness to fit the training data in arbitrary ways, which makes it hard for the model to exactly locate the correct solution when the training set does not sufficiently cover the whole data distribution.

To tackle this challenge, we proposed neural-symbolic stack machines [2, 6], which learn a neural network controller to operate a symbolic machine. Instead of directly generating the target sequence, the neural controller predicts an execution trace performing the translation process. The stack machine regularizes the search space and encourages the controller to extract translation rules that are applicable to a wide range of input sequences rather than individual samples. The neural-symbolic stack machine achieves 100% test accuracies on several benchmarks assessing the compositional generalization, where the transformation from input to output sequences satisfies a rigorous rule set, but the test input sequences are unseen combinations of words in the training vocabulary, and could be up to several hundred times longer than the training samples.

4 Ongoing and Future Research

I envision that low-code development is the future of the programming paradigm, where coding skills are no longer required for programming computers, hence everyone is able to create new software. I see several grand challenges of developing learning-based program synthesis techniques towards the goal: (1) limited scalability and efficiency of program search; (2) a lack of understanding of the mechanism driving the predicted outputs; and (3) the weaknesses and vulnerabilities of learning models. In the following, I highlight some concrete future directions to realize my research vision.

Learning-based program synthesis for scalable software tool development. My past work has demonstrated the feasibility of learning-based program synthesis approaches for various synthetic benchmarks and real-world applications. However, we still observe significant challenges in scaling up the techniques to handle more sophisticated code in large-scale projects. Moreover, existing learning-based program synthesis models generally suffer from sample inefficiency for synthesizing general-purpose code; e.g., tens or even hundreds of samples might be required to correctly predict a Python utility function implemented in 10 lines of code. In my future research, I plan to continue improving neural network architectures to better capture the underlying semantics of programs. Meanwhile, I aim to develop new program search algorithms to further improve the sample efficiency. For example, I will extend our execution-guided synthesis framework to more specification formats, and draw inspiration from classic divide-and-conquer algorithms to leverage the compositionality in programming languages and explore the program search space more efficiently.

Human-friendly interactive programming from multi-modal specifications. My work on program improvement shows the importance of iteratively updating the predicted program according to the execution results. Besides program execution, we can also directly seek user feedback to gradually refine the program. In my work on visualization code synthesis, I demonstrated that we can learn a model to synthesize code in real-world Python Jupyter notebooks crawled from GitHub, where the input specification contains interleaved code blocks and natural language markdown [1]. This interactive programming paradigm allows users to break down the full program and provide step-by-step natural language descriptions of each building block, so that the program synthesizer does not have to absorb the full program specification all at once. One important future research direction is developing interactive program synthesis systems that learn to adapt the predictions according to the user feedback, and ask for clarifications when necessary. To give users more flexibility in specifying the program intents, I also plan to develop neural network architectures to effectively aggregate the information from input specifications of different types, e.g., simultaneously supporting natural language descriptions, input-output examples, and other external resources as the reference.

Symbolic reasoning towards better robustness and generalization. In my past work, I have revealed several types of weaknesses and limitations of existing deep neural networks, including their generalizability and reasoning capability. Besides that, my work on adversarial machine learning also highlighted the security risks of deep neural networks, such as test-time attacks [18, 21] and training-time data poisoning [4, 12]. By learning to generate a symbolic representation as the model output, neural-symbolic models could potentially be more robust to distribution shift, while the predictions are also more interpretable and easier to verify. In the future, I plan to work on more systematic investigation of the success and failure modes of deep neural networks, including large-scale pre-trained models for program synthesis and other domains. Meanwhile, I am passionate about extending our neural-symbolic framework to support more diverse and noisy inputs, such as open-domain images and natural language text, and designing new pre-training and data augmentation schemes to strengthen the reasoning capability.

References

[1] Xinyun Chen, Linyuan Gong, Alvin Cheung, and Dawn Song. Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, 2021.

[2] Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. In *Advances in Neural Information Processing Systems*, 2020.

[3] Xinyun Chen, Chen Liang, Adams Wei Yu, Denny Zhou, Dawn Song, and Quoc V Le. Neural symbolic reader: Scalable integration of distributed and symbolic representations for reading comprehension. In *International Conference on Learning Representations*, 2020.

[4] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[5] Xinyun Chen, Chang Liu, Eui Chul Shin, Dawn Song, and Mingcheng Chen. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems*, pages 4574–4582, 2016.

[6] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. In *International Conference on Learning Representations*, 2018.

[7] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, 2018.

[8] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[9] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. Spreadsheetcoder: Formula prediction from semi-structured context. In *International Conference on Machine Learning*, 2021.

[10] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. In *Advances in Neural Information Processing Systems*, 2021.

[11] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6281–6292, 2019.

[12] Xinyun Chen, Wenxiao Wang, Chris Bender, Yiming Ding, Ruoxi Jia, Bo Li, and Dawn Song. Refit: a unified watermark removal framework for deep learning systems with limited data. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 321–335, 2021.

[13] 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, 2017.

[14] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3708–3719, 2019.

[15] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[16] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *arXiv preprint arXiv:2007.08095*, 2020.

[17] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning*, pages 2873–2882, 2018.

[18] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *International Conference on Learning Representations*, 2016.

[19] Hongyu Ren, Hanjun Dai, Bo Dai, Xinyun Chen, Michihiro Yasunaga, Haitian Sun, Dale Schuurmans, Jure Leskovec, and Denny Zhou. Lego: Latent execution-guided reasoning for multi-hop question answering on knowledge graphs. In *International Conference on Machine Learning*, 2021.

[20] Hui Shi, Yang Zhang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Deep symbolic superoptimization without human knowledge. In *International Conference on Learning Representations*, 2020.

[21] Xiaojun Xu, Xinyun Chen, Chang Liu, Anna Rohrbach, Trevor Darrell, and Dawn Song. Fooling vision and language models despite localization and attention mechanism. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4951–4961, 2018.