Programming in the Age of Intelligent Machines

Breandan Considine

August 11, 2021

1 Introduction

Since the invention of modern computers in the mid 20th century, computer programming has undergone a number of paradigm shifts. From the rise of functional programming, to dynamic, object-oriented, and type-level programming, to the availability of myriad tools and frameworks – its practitioners have witnessed a veritable Renaissance in the art of computer programming. With each of these paradigm shifts, programmers have realized new conceptual frameworks for reasoning and expressing their ideas more clearly and concisely.

Over the last few years, another paradigm shift has been set in motion, with significant implications for how we think about and write programs in the coming century. By most measures, computers have grown steadily more intelligent and capable of assisting programmers with mentally taxing chores. For example, intelligent programming tools (IPTs) powered by neural language models have this year helped over 10 million unique human beings program computers. As IPTs help digitally illiterate communities to discover their innate aptitude for computer programming, this population will continue to rise.

Computer programming is a uniquely creative exercise among the range of human activities. It channels our innate linguistic, logical, imaginative, and social abilities to bring abstract ideas into reality, and ultimately, gives humans the freedom to create new realities of their own design. In collaboration with other humans and the increasing participation of IPTs, vast and elaborate virtual worlds have been manufactured, where the majority of humankind now chooses to spend their daily lives. With the expanding opportunities these new digital frontiers promise, their population too will continue to grow.

Today IPTs share an equal role in shaping many aspects of computer programming, from knowledge discovery to API design, and program synthesis to validation and verification. However, this balance is shifting beneath our feet. Once its creators, programmers are now primarily consumers of information provided by an IPT, and increasingly rely on them to perform their daily work. With the unique opportunities and risks this partnership presents, what division of labor should exist between humans and our new coding collaborators? This is the question we have set out to understand in the following literature review.

2 Neural Language Models for Source Code

Programming researchers have long held an interest in using intelligent tools to help them write programs [5]. Due to fundamental limitations in data and processing power, many of these ambitions did not come to fruition until the last few years, thanks the availability of *big code* [2], the development of differentiable programming libraries for gradient-based learning [3], and attention-based language models [9], among other technical achievements. Armed with this new repertoire, programming researchers have revisited their interest in IPTs. Naturally, one of the first applications considered was code completion.

Following their initial success in natural language, rapid progress continues to be made in the application and specialization of transformers to source code, as well as industrial transfer where this technology is now trained and deployed on millions of programmers worldwide [6]. Given a natural language description of an incomplete method, these models are capable of inferring programmer intent and completing multiline code snippets. Progress is expected to improve.

The problem comes down to a question of grammar induction. Based on empirical results, fixed-precision transformers (e.g. GPT-2, BERT) are thought capable of recognizing the class of counter languages [4], i.e. somewhere between context-free and context-sensitive, although this characterization requires a more careful theoretical analysis. For source code typically stored on GitHub, this class would appear to suffice – models trained on such datasets are currently capable of rudimentary program sketching and boilerplate code completion, however more complex fragments require additional oversight.

An important shortcoming of imitation learning is the question of data provenance and validation. Even if the training data is syntactically and semantically valid, constraints on the class of valid programs are ill-posed. As a consequence, a large fragment of programs generated may be syntactically valid but semantically unsound, i.e. may throw runtime errors at best, or appear to work at first, but are in fact broken in a subtle manner. Like most language models of its kind, performance is highly sensitive to the dataset quality, as common errors in the training data can be inherited and reproduced.

The vast majority of modern programming consists of writing ceremonial boilerplate, tasks for which neural language models are well-suited. A tremendous amount of human labor is spent on such chores, and reallocating those resources towards more intellectually stimulating tasks may encourage a larger demographic to become programmers who would otherwise lack the patience or interest. By removing these barriers to entry, programmers can more quickly arrive at the rewarding parts of program design and implementation.

Nevertheless, mere imitation is a somewhat dissatisfying approximation to programming from a computer-scientific perspective – lacking in some essential aspect the qualities its practitioners aspire to fulfill. Helpful though it may be for tedious chores, programming is not about reading gigabytes of code and minimizing a cross-entropy loss. Programming requires imagination, creativity, problem-solving – qualities which cannot be conferred by simply scaling existing language models with more data and parameters. What could be missing?

3 Knowledge Discovery and Neural Code Search

Search is an indispensable tool in computer programming. Many problems in computer science can be distilled to search, and the subject is widely studied in machine learning, from dynamic programming to statistical optimization, and information retrieval to computational linguistics.

Regarding the earlier question of, "What else could programmers be doing besides imitation learning?", one plausible answer could be trial and error. Given some program specification and a computational budget, a naïve strategy could be to simply fetch existing programs from a dataset, and evaluate as many as possible in the computational resources allotted. Many programmers do in fact practice this style of copy-paste programming as evidenced by duplicate code studies [8], a problem which can introduce bias to machine learning models and must be corrected for during data curation [1].

A slightly more refined strategy would be to choose a more granular building block for programs, and build up the program incrementally by searching for reusable components. In what order should candidates be evaluated? Many heuristics have been proposed, from execution-guided search [7, 10], to weighted search via reinforcement learning, to other dynamic programming strategies. These all rely on judicious pruning of the search space, however the problem reduces to a question of optimal substructure. If this property cannot be established, search becomes computationally intractable

Many problems we want to solve require searching a space whose complexity is simply inaccessible. Various mitigations such as backtracking and random restarts have been proposed to alleviate this issue, however the topology of many search problems does not admit an efficient beam search.

4 Automatic and Synthetic Programming

5 Computer-Aided Reasoning Tools

When programmers ask for an intelligent programming tool, what they really want is not a subservient genie who grants wishes without question, but a teacher who rapidly gives them feedback about the implications of their design choices within the confines of a well-defined language. This is one advantage of a type system: while not itself particularly intelligent by any measure, it is at least reliable.

6 Future Directions of Computer Programming

7 Conclusion

References

[1] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[3] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[4] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

[5] M Bras and Y Toussaint. Artificial intelligence tools for software engineering: Processing natural language requirements. *WIT Transactions on Information and Communication Technologies*, 2, 1993.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[8] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[10] Chenglong Wang, Po-Sen Huang, Alex Polozov, Marc Brockschmidt, and Rishabh Singh. Execution-guided neural program decoding. In *ICML workshop on Neural Abstract Machines and Program Induction v2 (NAMPI)*, 2018.