Programming in the Age of Intelligent Machines

Breandan Considine

August 13, 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 [8]. Due to fundamental limitations in data and processing power, many of these ambitions had not come to pass until the last few years, thanks to the availability of *big code* [2], the development of differentiable programming libraries for gradient-based learning [6], and attention-based language models [15], 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 [9]. 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 [7], 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 sketching rudimentary programs and boilerplate code templates, 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 well-formed, constraints on the class of valid programs are ill-posed. As a consequence, a large fragment of languages generated may be syntactically valid but semantically unsound, i.e. may throw undesired runtime errors, 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 are prone to be inherited and reproduced by an IPT.

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, the art of programming is not 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

For many decades, computer scientists have pondered the nature of search. Search is an indispensable tool in the working programmer's repertoire and goes to the heart of many fundamental problems in artificial intelligence, from classical to statistical optimization, and information retrieval to computational linguistics. Programming itself can be seen as a kind of search-based optimization problem [4], consistent with its original mathematical interpretation, e.g. linear or stochastic programming. Coupled with a grammatical template, one could imagine searching through the space of valid programs to produce a higher-order function satisfying some criteria. Indeed, this exact setup is routinely-studied in the annual syntax-guided synthesis (SyGuS) competition [3].

Returning to our earlier question of, "What else could human 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 evaluate as many programs from a dataset of candidate solutions as possible within the allotted budget. Many programmers do in fact practice this style of copy-paste programming as evidenced by duplicate code studies [14], a problem known to adversely bias machine learning models, and which must be corrected for during data curation [1].

For nearly all practical programming languages, the space of valid programs can hardly be enumerated, never mind evaluated in a reasonable amount of time. A more refined strategy is needed: for example, we could select a small set of reusable building blocks, then compose and evaluate partial programs using an execution-guided scheme [10, 16]. By interacting with an interpreter, we may be able to arrive at a solution via incremental improvement. As with most dynamic programming algorithms, the problem comes down to a question of substructure: if modification invalidates prior work, search can become exponential or worse.

For example, many useful programs belong to the class of context free languages. Sampling is possible using a probabilistic grammar, but at what cost? The number of distinct parse trees grows super-exponentially with height. Various strategies have been designed to inhibit this growth, but even with judicious pruning, the topology of many languages does not admit an efficient beam search, or the cost required may be prohibitive. Yet humans are able to solve many computationally hard search problems with deceptive ease. How?

One possibility is that humans possess more computational resources than might we give them credit for, and a similarly-enriched neural-guided search would be equally capable. Another hypothesis is that we are not *searching* for programs as such, or at least thinking about this process in the wrong terms. When someone is painting a portrait or writing a novel, we do not call this search. Likewise, programming is not always about searching for an answer, but finding the right question to ask, of understanding, exploring and visualizing the design space, for which there is no specification. The program acquires a meaning of its own as the medium for a dialog between human and machine. In the following sections, we explore two contrasting models for this dialog, one where the human is the teacher (§4), and one where the IPT is the teacher (§5).

4 Automatic and Declarative Programming

In The Art of Computer Programming [13], Donald Knuth memorably writes, "Programs are meant to be read by humans and only incidentally for computers to execute." Taking this perspective, one may be tempted to ask, "Why must programming languages be so difficult that we need IPTs to write down our ideas in the first place?" If we consider programming a matter of communicating human intent to machines, language designers should take great pains to simplify the language so that users may convey their intent in an effortless manner, then harness machine intelligence in the service of fulfilling their intent, rather than force them to specify how to accomplish it in painstaking detail. Known as declarative programming, this approach has been well-studied in languages like SQL, Prolog and miniKanren, among others.

The idea of what is today known as declarative programming can be traced back to the 1940s when researchers started applying the tools of constraint programming to what is today known as operations research. In this early work, programmers would first state their intent as a minimization problem on a system of inequalities, such as a linear or quadratic program. The Soviets used this technique to study many problems related to optimal transport and economic planning. It also turns out to be useful as a means of defining metrics on graphs and probability distributions.

Our primary interest in declarative programming is twofold: (1) it is the basis of the first successful attempt to popularize gradient-based learning [6], and (2) is a practical framework for realizing the once-scorned [11] but now increasingly plausible [9] idea of natural language programming.

Not only can constraint programming produce numerical solutions, but the same tools can be applied to programming in a more general manner. By considering different algebras, we can compute different properties objects. These things are called algebraic path problems and have many useful applications in combinatorial optimization.

It is possible to implement many fundamental computer science algorithms in a much simpler way as iterated matrix multiplication on a semiring algebra. A commutative monoid $(S, \bullet, ①)$ is a set S with a binary operator $\bullet : S \times S \to S$ which has the following properties:

$$\frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c} \qquad \frac{a \bullet ①}{a} \qquad \frac{a \bullet b}{b \bullet a}$$

Associativity *Neutrality* *Commutativity*

A semiring algebra, denoted $(S, \oplus, \otimes, ⓪, ①)$, is a set together with two binary operators $\oplus, \otimes : S \times S \to S$ such that $(S, \oplus, ⓪)$ is a commutative monoid and $(S, \otimes, ①)$ is a monoid. It has the following additional properties:

$$\frac{a \otimes (b \oplus c)}{(a \otimes b) \oplus (a \otimes c)} \qquad \frac{(a \oplus b) \otimes c}{(a \otimes c) \oplus (b \otimes c)} \qquad \frac{a \otimes ⓪}{⓪}$$

Distributivity *Annihilation*

It is possible solve a rich family of search problems using iterated matrix multiplication on semirings. Known as *propagation* or *message passing*, this procedure consists of two steps: *aggregate* and *update*. Let δ_{st} denote some distance metric on a path between two vertices s and t in a graph. To obtain δ_{st}, one may run the following procedure on a desired path algebra until convergence (a few examples are provided for reference):

$$\delta_{st} = \overbrace{\bigoplus_{P \in P^*_{st}} \underbrace{\bigotimes_{e \in P} W_e}_{\text{Aggregate}}}^{\text{Update}}$$

S	$\oplus$	$\otimes$	⓪	①	Path
$\mathbb{R} \cup \{\infty\}$	min	$+$	∞	0	Shortest
$\mathbb{R} \cup \{\infty\}$	max	$+$	$-\infty$	0	Longest
$\mathbb{R} \cup \{\infty\}$	max	min	0	∞	Widest

Many dynamic programming algorithms, including Bellman-Ford, Floyd-Warshall, Dijkstra, belief, constraint, error and expectation propagation, Markov chains, and others can all be neatly expressed as message passing on a semiring algebra. We refer the curious reader to [12, 5] for a more complete summary of the algebraic path problem and its many wonderful applications.

For example, one could use ADTs to represent higher-order functions, i.e. for program synthesis if they so desired.

The problem with automatic programming is that the only means of feedback to the user is by outputting a solution, or failing that, producing lots of heat. It would be nice to have a more interactive mechanism.

5 Computer-Aided Reasoning Tools

The programmer is not the teacher, the programmer is actually the student.

When programmers ask for an intelligent programming tool, what they really want is not a 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: not only does it provide highly precise feedback, but also allows us to sample a space of semantically constrained programs. Thus the interaction is bidirectional - the user provides a partial program sketch, and asks the type system to infer the missing thing.

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 In-*

ternational 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] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[4] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93, 2018.

[5] John S Baras and George Theodorakopoulos. Path problems in networks. *Synthesis Lectures on Communication Networks*, 3(1):1–77, 2010.

[6] 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.

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

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

[9] 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.

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

[11] Edsger W Dijkstra. On the foolishness of natural language programming. In *Program construction*, pages 51–53. Springer, 1979.

[12] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer Science & Business Media, 2008.

[13] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[14] 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.

[15] 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.

[16] 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.