Programming in the Age of Intelligent Machines

Breandan Considine

September 6, 2021

Contents

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 and object-oriented 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 expressing their ideas more clearly and concisely.

Over the last few years, a new 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 people program computers. As IPTs help digitally illiterate communities to discover their innate aptitude for programming, this population will continue to increase.

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 are being manufactured, where the majority of humankind now chooses to spend their lives. With the expanding opportunities these new digital frontiers offer, 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 Code Completion and Program Synthesis

Programming researchers have long held an interest in using intelligent tools to help them write programs [11]. Due to fundamental limitations in data and processing power, many of these ambitions have come to pass only recently, thanks to the availability of *big code* [2], the development of differentiable programming libraries for gradient-based learning [9], and attention-based language models [35], 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 [12]. Given a natural language description of an incomplete method, these models are capable of inferring programmer intent and completing multiline code snippets.

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 [10], 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, however more complex fragments require additional oversight.

An important shortcoming of imitation learning is the question of data provenance: even if the training data is syntactically well-formed, constraints on the class of valid programs are ill-posed. As a result, a large fragment of languages generated may be syntactically valid but semantically unsound, i.e., may throw runtime errors, or appear to work at first, but are in fact broken in a subtle manner. Like most language models of its kind, 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 redirecting attention 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, programmers can more quickly arrive at the rewarding parts of program design and implementation.

Nevertheless, 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 in reading gigabytes of code and minimizing a cross-entropy loss. Programming requires imagination, problem-solving – qualities which cannot be conferred by scaling existing models with more data and parameters. What could be missing?

Computer scientists have long pondered the nature of search. Search is an indispensable tool in the 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 [5], 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 studied in the annual syntax-guided synthesis (SyGuS) competition [4].

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 a 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 [28], a problem known to adversely bias machine learning models, and which must be corrected for during data curation [1].

For most 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 [13, 37]. 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: unless the problem can be decomposed into overlapping subproblems, program search can be 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 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 per se: when painting a portrait or writing a novel, we do not call this search. Likewise, **programming is not necessarily about searching for an answer, but finding the right question, of exploring a design space whose specification is a consequence, not a catalyst of the design process.** The program's source code could merely be an artifact of an ephemeral dialog between a human and a machine, not the final result. In the following section, we explore two contrasting models for this dialog, one where the human is the teacher and one where the IPT is the teacher.

3 Declarative and Automatic Programming

In The Art of Computer Programming [27], 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 to be simply 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 that intent, rather than force the user to give precise instructions. Known as *declarative programming*, this approach can be found in languages like SQL, Prolog, and miniKanren.

The essence of declarative programming can be traced back to the 1940s when researchers started applying tools from mathematical optimization to what is today known as operations research [24]. In this early work, programmers would state their intent as a solution to a system of inequalities, e.g., for optimal transport or economic planning. More recently, it has found important applications for defining metrics on probability distributions, the construction of which are an essential aspect of machine learning. Metrics provide a way of biasing learning algorithms towards solutions of a certain form, and thus can be seen as an indirect form of programming.

Not only can declarative programming be used to produce numerical solutions to systems of equations, but the same ideas can be applied to synthesize other kinds of data, such as programs. Given some existing program $\mathcal{P}$ and a set of declarative constraints in the form of program transformations, we could construct an *adjoint* program $\mathcal{P}'$ by applying the transformations recursively. For example, this might allow us to approximate a probability density function, or compute its derivative with respect to one or more inputs. These techniques are broadly known as *automatic programming*.

Our interest in automatic programming is twofold: (1) as the basis for the first successful open-source implementation of gradient-based learning [9], and (2) as a practical framework for realizing the once-scorned [16] but now increasingly plausible [12] idea of natural language programming. For example, consider the following natural language specification:

```
fun transformStringIntoListOfUniqueWords(s: String) = TODO()
```

This is the completion provided by Codex [12], a recent IPT from OpenAI:

```
fun transformStringIntoListOfUniqueWords(s: String) =
    s.toLowerCase().replace(Regex("[^a-zA-Z]"), " ").trim()
    .split("\\s+".toRegex()).filter { it.isNotBlank() }.toSet()
```

The problem with automatic programming is that it assumes the specification is (1) infallible and (2) faithfully represents its author's intent. If either assumption is invalid, the resulting output could be nonsensical, or appear to work but actually contain a subtle error. How could we detect such an error and provide earlier feedback if the specification were ill-posed?

As adept as human programmers are at certain tasks, they can be short-sighted. The larger a program grows, the more likely its author is to make a mistake due to combinatorial explosion – whenever a new feature is added, it has the potential to interact with every other feature in unpredictable ways. This is why language designers are often hesitant to add new features, a challenge which has come to be known as the feature-interaction problem [7]. Taking this perspective, the programmer is not an all-knowing oracle, but actually a student who proposes ideas to the IPT, and in return, receives feedback about their consequences in the context of a *type system*.

When programmers ask for an intelligent programming tool, what they really want is not a subordinate who blindly follows orders, but a tutor who quickly gives feedback about the implications of their design choices. This is the advantage of having a type system: not only does it give relevant feedback, but allows us to constrain the space

of semantically valid programs. The interaction model is bidirectional: the user provides a typed program sketch. If the program is valid, the type is witnessed by an inhabitant. And if the program is inconsistent, the type system provides feedback to the user as to where and how.

For example, all procedures in a statically-typed programming language have a *type signature*, e.g., `getHomePhone: Person → PhoneNumber`. This represents a *contract* between the procedure and its caller: if the procedure `getHomePhone` is called with a `Person`, it will return a `PhoneNumber`. All other input and output types are forbidden by the type system.

What if we could encode the type system constraints into our neural language model – how would we go about doing that? One way is to use neural guided search, where we do beam search over a generative language model and discard the programs which are not well-typed. The second way is more difficult, but morally proper: we learn a probability distribution on the space of semantically correct programs from the outset, so the resulting samples are always valid programs and rejection becomes unnecessary. Broadly speaking, this approach is known as type-safe probabilistic programming, and has been considered for various applications, for the analysis of cryptographic algorithms, to physical simulation, to Bayesian program learning [29].

4 Knowledge-based Programming

Historically, most knowledge was stored as natural language. A growing portion is now *code* [2]. The majority of code is procedural knowledge, written by a human and intended to operate a machine. Though it shares many statistical properties in common with natural language [19], code has an unambiguous grammar and well-defined semantics [31]. We can use these properties to construct graphical models and reason about their structure.

Early work in program learning realized the importance of graph-based representations [3], however explicit graph construction requires extensive feature-engineering. More recent work in program synthesis has explored incorporating a terminal [17], graphical [36] or other user interface to explore the space of valid programs, however they do not consider the scope or variety of artifacts in a software project. Others have shown the feasibility of learning a local graph [23] from source code, but still require an explicit parser to form the initial graph. Moreover, adaption to settings where style, terminology and document structure vary remains a challenge.

Application programming interfaces (APIs) are interfaces which describe available ways of structuring computation to achieve related programming tasks. We call the graph of all possible ways to compose an API the *API surface.* One traverses the API surface by composing accessible procedures in a *call graph.* Thus, we can view the API as a kind of *procedural knowledge base* representing common data transformations and how to compose them. In practice, how to achieve some desired goal is often far from obvious, requiring a large amount of documentation to explain.

Many consumers of popular APIs publish code and documentation in open source repositories, a largely untapped source of knowledge for programming tools. New work seeks to find ways of linking knowledge contained in open source repositories to help users locate examples and compose programs. In the following section, we review three applications for document alignment (§ 4.1), code search (§ 4.2), and knowledge extraction (§ 4.3).

4.1   Documentation alignment

Documentation is an indispensable resource for software developers learning to use a new API or programming language. Maintainers of popular software projects often publish web-based developer documents, typically in markup languages such as HTML or Markdown. These documents contain a mixture of natural language sentences, code snippets, and hyperlinks to related documents and source code files. All of these artifacts hold rich semantic information: the markup graph describes the text in relation to other entities in the document hierarchy, while the link graph describes relationships between relevant documents or artifacts in a software repository.

Consider the typical workflow of a software developer who is seeking information about an unfamiliar API. To effectively locate relevant documentation, a specific fragment of text (e.g., a function name, error message, or identifier) must first be copied from a development environment into a search engine, providing relevant contextual information. The query must be descriptive enough to retrieve relevant documents with high probability, while omitting extraneous information (e.g., user-defined tokens) unlikely to occur outside the scope of the developer's personal environment or project.

Prior work in information retrieval for software development investigated recommending API documentation [33] and Q&A content [34] to developers. Similar work in natural language processing has studied the relationship between comments and source code entities [21, 30] strictly within source code.

Examples of cross-domain entity linking in the source-to-doc (S2D) and doc-to-source (D2S) setting are scarce, however these results indicate alignment between natural language and software artifacts may be feasible.

More recent work seeks to facilitate procedural knowledge discovery by enriching lexical queries with semantic information extracted from a programming environment, and prioritizing semantically relevant software artifacts among a set of matching search results. Broadly, the tools in this literature review can be used to study both source code and documentation. Reasoning about relations between these artifacts will require developing new approaches to feature engineering, unsupervised learning and entity alignment in the low-data regime. Such an application would allow developers to more quickly and easily locate semantically or contextually relevant code samples in API documentation and open source repositories. These tools could also help authors maintain a consistent set of API documents and usage examples across a large codebase – a persistent obstacle for software maintenance.

4.2   Human-mediated code search

Given a learned similarity metric between procedures, one straightforward application is code search. Prior work has explored type-directed [22], learning-based [18] and semantic [32] code search. These techniques all use a fixed, or synthetic ordering over search results. For a given context, there are often many valid completions within an API or codebase. Given a corpus of procedures in their surrounding typing environment, is it possible to estimate a probability distribution on a shared embedding between contexts and results, and measure the likelihood that a given search result occurs in an empty location? This requires:

1. Efficiently searching a corpus for a well-typed pattern

2. Ranking the matching search results by semantic alignment

3. Incorporating information into user's context (e.g., variable renaming)

Given a cursor and the surrounding context (e.g., in an IDE or editor), such a tool would need to search a database for the most similar contexts, extract common snippets to estimate their *concordance* or *agreement* with the surrounding code context, then adapt the foreign code snippet into the user's context. Public code samples, API-documentation or version control

history, could help the user to fix some erroneous or outdated piece of code. Some open research questions include:

1. **Semantic segmentation**: How do we slice or truncate context?

2. **Graph search**: What kernels enable fast subgraph detection?

3. **Context ranking**: What features best measure contextual similarity?

4. **Refactoring**: How to integrate a selected result into the user's code?

Other work in the code search literature explores the text-to-code [20] setting, where queries are typically considered to be a short sequence composed by the user, or code-to-code [25] setting where the query takes the form of a code snippet. Model performance is typically evaluated using mean reciprocal rank (MRR), mean average precision (MAP), normalized discounted cumulative gain (NDCG), precision and recall at top N-results (P/R@N), or similar metrics. Although some [8] do incorporate other features from the local document, few consider the query in the context of a broader project. The ability to align contextual features from the surrounding project, we argue, is essential to delivering semantically relevant search results.

4.3   Software knowledge extraction

What is a language model? Disregarding meaning, a language model is no more than a statistical description of symbolic information. Natural languages differ from mechanical languages in a few key ways. Natural languages are linear: almost all human languages have a fixed maximum recursion depth governed by biological and cognitive factors. Programming languages, by contrast, are more strongly nonlinear due to their metalinguistic properties. Thus, current natural language models cannot represent many programming languages without significantly constraining their expressiveness. Prior studies of source code have been undertaken [39, 14, 12] to characterize the families of computational languages that neural language models can recognize. These studies rely on statistical analysis of code corpora, but do not provide tight bounds on the expressiveness of neural language models.

One might argue – since natural language models are effectively linear – a language model with a large enough working memory, given enough data, should be able to model language fragments whose description lengths fit

inside their working memory. Indeed, studies show that natural language models are surprisingly adept at modeling idioms in source code. However, we would expect such models to struggle with fragments whose complexity exceeds its working memory (e.g., whose average MDL stretch this limit).

What is the difference between working memory, i.e., reasoning, and long-term memory, i.e., knowledge? Long term memory is information stored in the learned parameters, i.e., the topological structure of the network – as a model *learns*, this information is passively encoded in the topological structure. Long-term memories must be conserved by the data distribution: distributional shift tends to erase past memories, a phenomena known as catastrophic forgetting. In contrast, short-term or working memories are dynamical patterns of activity which allow a network to reason about incoming signals and adapt to previously unseen scenarios without memorization. To be propagated forward in time, working memories must be actively conserved by the network topology. **Both learning and reasoning are a forms of message passing over a graphical structure: learning shapes the graph topology, and reasoning propagates signals through it.**

Models which cannot fit a computational language into working memory must learn maximal-length fragments and use long-term memory to fill-in the gaps – i.e., by memorizing transitions between fragments in long-term memory. If this is the case, we would expect to find transitions that are locally consistent, but globally incoherent. For instance, can source code models learn balanced parentheses? Balanced intermingled parentheses? {([])} Other algebraic datastructures (e.g., bushes, imbalanced trees, etc.)? What are the limit languages that can be learned by natural language models of source code (e.g., transformers, RNNs) and their failure modes in practice?

Finally, can we extract any useful knowledge from a trained language model? Two high-level strategies have been proposed in recent literature: we could either decode a useful representation, such as a finite [38] or weighted finite automaton [40] from a pretrained neural language model using queries and counterexamples a la Angluin [6], or we could train a model to directly synthesize a graph using a reinforcement learning based approach [23]. While early results appear promising, further investigation is required to understand the tradeoffs between these two approaches.

5 Conclusion

The last few years have shown a variety of different interaction models between humans and IPTs. One potential future is that human beings will be eventually left behind in the race for higher forms of intelligence (computer scientists are largely divided over this topic). Regardless whether such a thing will come to pass, we will need new tools for humans to communicate their values and preferences effectively. How can we convince machines to use their intelligence in service of our mutual good, whilst giving them agency to explore solutions that we may not understand? This question is broadly known as the AI alignment problem [26, 15] and is attracting an interest from both the computing sciences and humanities.

Our position is that alignment will require a closer collaboration between computer science and humanities than historically practiced, and this partnership will be invigorating for both disciplines.

One way of ensuring that our values are faithfully encoded by machines is through *programming*. In this literature review, we give various examples of programming models that have been envisioned, e.g., from autocompletion and program synthesis (§2), to autoprogramming and type inference (§3). Another way to propagate human values is by teaching the humanities: arts, language, culture, history and philosophy. By keeping these traditions alive, we transmit our cultural heritage to future generations so that they, by studying the lessons of our past, may avoid repeating the same mistakes. Those generations may soon be enlisted to help share our values with computers.

Given the clear and present risks associated with misuse, great care must be taken to ensure that intelligent systems do not bring harm into the world. For this reason, we encourage the careful study of programming language theory, which helps ensure the safety, security and energy efficiency of computer programs and statistical learning theory, which has developed tools like risk minimization and probabilistic programming to reason about uncertainty.

6 Acknowledgements

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] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

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

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

[6] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[7] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 1–8, 2013.

[8] M. H. Asyrofi, F. Thung, D. Lo, and L. Jiang. Ausearch: Accurate api usage search in github repositories with type resolution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 637–641. IEEE, 2020.

[9] 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.

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

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

[12] 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.

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

[14] Nadezhda Chirkova and Sergey Troshin. Empirical study of transformers for source code. *CoRR*, abs/2010.07987, 2020.

[15] Brian Christian. *The Alignment Problem: Machine Learning and Human Values*. WW Norton & Company, 2020.

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

[17] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. *arXiv preprint arXiv:1906.04604*, 2019.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[19] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[20] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

14

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

[22] Michael B James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for Haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[23] Daniel D Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. *arXiv preprint arXiv:2007.04929*, 2020.

[24] Leonid V Kantorovich. Mathematical methods of organizing and planning production. *Management science*, 6(4):366–422, 1960.

[25] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: A code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 946–957, New York, NY, USA, 2018. Association for Computing Machinery.

[26] Tae Wan Kim, Thomas Donaldson, and John Hooker. Mimetic vs anchored value alignment in artificial intelligence. *arXiv preprint arXiv:1810.11116*, 2018.

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

[28] 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.

[29] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017.

[30] Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. Associating natural language comment and source code entities. In *AAAI*, 2020.

[31] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software Foundations. `https://softwarefoundations.cis.upenn.edu`, 2020.

[32] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *PLDI*, pages 1066–1082, 2020.

[33] Martin P Robillard and Yam B Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[34] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 392–403, New York, NY, USA, 2016. Association for Computing Machinery.

[35] 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.

[36] Homer Walke, R Kenny Jones, and Daniel Ritchie. Learning to infer shape programs using latent execution self training. *arXiv preprint arXiv:2011.13045*, 2020.

[37] 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.

[38] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pages 5247–5256. PMLR, 2018.

[39] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. *CoRR*, abs/1805.04908, 2018.

[40] Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. *arXiv preprint arXiv:1910.13895*, 2019.