A Pragmatic Approach to Syntax Repair

Breandan Considine
bre@ndan.co
McGill University

1 Abstract

Programming languages share a social and formal heritage. These families were historically divided, but share deep roots, and we argue their destined matrimony heralds important consequences for language design and generative language modeling. In our work, we develop a sociotechnical framework for understanding the dynamics of programming and argue it captures many of the social and formal properties of language acquisition and evolution.

2 Motivation

Are languages acquired or constructed? In empirical software engineering and sociolinguistics, many would argue languages are statistical phenomena that arise in the context of repeated social interactions. Tracing linguistic artifacts can shed light into language acquisition and adaptation, and offers insight into the social dynamics of communicating agents. In contrast, programming languages are deliberately constructed and most designers would argue a language's social dynamics, though perhaps informative, are surely less fundamental than its runtime dynamics. This is a curious distinction, since programming languages appear to share many statistical similarities with natural language [4], and the act of programming can itself be understood as a form of communication [2] between minds and machines.

Our research seeks to understand the common principles that shape the creation and evolution of natural and programming language artifacts, informed by cooperative game theory. In particular, we draw inspiration from both natural and computational linguistics to develop a constructive theory of code completion and program repair. Our proposed framework can be translated into practical development tools for software engineering, as well as a benchmarking suite for probing the reasoning capabilities of neural language models on prosocial programming problems (PPPs).

Language games are a pragmatic framework for modeling the social dynamics of code. In one such game, two players, the driver and a navigator, collaborate across a shared workspace to construct a domain-specific language that solves a chosen problem. Initially, both parties have a limited understanding of their counterpart and to reach a common understanding, must exchange messages conveying, e.g., their intent, abilities and knowledge. This game comes to an end when the chosen problem is successfully solved, the clock eventually runs out, or the problem is deemed unsolvable and one player forfeits due to exhaustion.

We call this framework *pragmatic alignment* and argue it not only encompasses many behavioral aspects of social computing such as hackathons and pair programming, but also underpins the design and implementation of integrated development environments (IDEs), and offers a benchmark for evaluating the systematic reasoning and compositional generalization of intelligent programming tools (IPTs). In light of our growing reliance upon IPTs and their evident limitations, new techniques are needed to increase the reliability of programming agents deployed in prosocial environments.

No more evident is the rudimentary state of programmatic reasoning than the currently popular alignment mechanism, which uses imitation learning with human feedback to fine-tune a transformer-based language model, then employs a crude prompt like, "Let's think step-by-step…". While it may convincingly imitate humans on certain reasoning tasks, this strategy is hopelessly flawed, as (1) human feedback is sparse and often incorrect, (2) transformers often learn reasoning [3] shortcuts [5] that fail to generalize across domains, and (3) transformers have known limitations [6] to their expressive [1] capacity preventing generalization, even in the infinite limit of data and compute. These shortcomings raise serious doubts that preference alignment alone can produce programming agents capable of solving decision problems whose expected complexity is significantly harder-on-average than presented in the training distribution.

Instead, the pragmatic approach advocates for training and evaluating programmatic reasoners on a curriculum of language games with increasing average-case complexity, such as proof search or program repair. As we argue, logical or semantic validity is a more trustworthy alignment criterion than preferability, and a much more reliable indicator of systematic and compositional generalization – the hallmarks of programmatic reasoning in humans. Pragmatism can also complement preference alignment to improve the reliability and preferability of generated source code, and offers a more principled approach to designing and implementing IDEs.

To anchor our discussion hereinafter, we shall restrict ourselves to a specific game exemplifying the pragmatic framework that falls at the intersection of programming languages and developer tools: syntactic error correction. Towards that end, we develop a tool for correcting invalid syntax and evaluate it on a variety of toy and practical programming languages, demonstrating SoTA precision and throughput at a fixed timeout relative to transformers. Tidyparse, as we call it, is useful in its own right for program repair, and provides a benchmarking suite for evaluating programming agents.

3 Problem

Broadly put, program repair is the problem of refining an existing program that approximates some specification but falls short. In this regard, repair is a bit like a limit-process: the more we refine our code, the closer it should become to its specification. The pragmatic approach to program repair takes a charitable view of the author: not as a clumsy fool to be scolded into conformity, but an earnest programmer trying to convey their intent, who may have strayed slightly off-course. The pragmatic ideal tries to imagine what the author could have intended, as opposed to prescribing what should have been written. We will direct our attention to one particular form of program repair, namely *syntax correction*.

When writing code, nearly all the intermediate editing states are syntactically invalid. Manual repair, though fairly routine, introduces friction to the development process by occasionally diverting the author's attention. *Syntax correction* is the process of automatically repairing a syntactically invalid program so that it is no longer invalid. This problem may seem trivial but can be quite challenging, as well-formed programs have many semantic constraints, but also because the solutions are highly under-determined: although repairs must be valid, even assuming a tiny number of edits, the search space for valid edits can be vast indeed.

For example, let us consider a cooperative game between two players, Author and Editor: both players see the same grammar and invalid string and both players simultaneously move after a short period of time without seeing each other's move. Author produces a single string, and Editor produces a list of valid strings, with the following outcome:

- If Editor anticipates Author's move, they both win.
- If Author's move is valid and unique, Author wins.

We call this the *syntax repair game* and present a nearly optimal strategy for Editor by observing a few simple rules: Editor should seek out plausible repairs first, but eventually discover every syntactically valid repair within a given number of edits from the original invalid string. Furthermore, Editor must sample repairs without replacement in parallel across all available cores and whensoever interrupted, return the best candidates hitherto discovered. Finally, Editor must be able to repair programs in a variety of languages without any prior statistical knowledge or domain-specific heuristics. These constraints introduce a number of interesting design challenges that will be addressed in the following section.

4 Approach

During the conception of Tidyparse, a number of design choices were made to increase its utility as a real-time programming assistant, which aims to provide precise and continuous feedback while the user is typing across a variety of programming languages. To support this use case, the following criteria were taken into account when designing and evaluating various components of the repair procedure.

- First and foremost, the framework must be **sound** and **complete**. That is, the parser-generator must (1) accept arbitrary context-free grammars, and (2) generate a parser which accepts all and only syntactically valid strings and (3) when a string is invalid, our synthesizer must eventually generate every syntactically valid string within a fixed edit distance from it.
- Second, we require that the resulting repairs be **plausible** and **diverse**. In other words, the framework should generate repairs that are likely to be written by a human being, consistent with the surrounding context, and reasonably diverse in their structure.
- Third, the framework must be **efficient** and **responsive**. It must be able to recognize well-formed strings in subcubic time, and generate admissible repairs in subpolynomial time. These conditions are necessary to provide real-time feedback whilst the user is typing.
- Fourth, the framework must also be **robust** and **scalable**. In practice, this means that the framework should be robust to multiple errors, handle grammars with a large number of productions and be able to scale linearly as the number of processor cores are increased.
- Fifth, the framework must be **flexible** and **extensible**. As a general-purpose tool for generating syntax repairs in a wide variety of programming languages, end-users should be able to extend the framework with their own custom grammars and type constraints.

As we demonstrate both formally and experimentally, our framework is able to satisfy most of these criteria simultaneously, while attaining state-of-the-art performance in terms of precision, throughput and wall-clock latency when compared with a variety of neural and symbolic baselines.

We can reframe syntax error correction as a language edit distance problem, but unlike typical error-recovery tools that only return the nearest repair(s), we try to find every valid repair within a fixed edit distance. This problem can be modeled as the conjunction between a context-free language (CFL) representing the syntax and an automaton that recognizes the Levenshtein ball around an invalid string. Adapting Valiant's [8] algorithm, which reduces CFL recognition to Boolean matrix multiplication, we develop a novel reduction from conjunctive language reachability onto finite field arithmetic. At a high level, we lift Valiant's algorithm into the space of conjunctive languages, conjoin the syntax and Levenshtein automaton, compile this language into a linear system of equations, abstractly interpret it using a SAT solver, decode the fixedpoints as syntax repairs, rerank the repairs using a suitable distance metric, then present them to the user for manual inspection. Our technique is both asymptotically efficient and wall-clock competitive with LLMs.

We have implemented our approach as an IDE plugin and demonstrated its viability as a practical tool for realtime programming. A considerable amount of effort was devoted to supporting fast error correction functionality. Tidyparse is capable of generating rapid and natural repairs for invalid code in a range of practical programming languages using only the grammar and a cheap ranking metric over strings.

5 Evaluation Methodology

There are essentially two ways to evaluate a syntax error corrector: (1) measure the rate at which repairs are accepted by the true parser, or (2) measure the rate at which human repairs are matched. The former approach is technically correct, but does not distinguish between correct and natural repairs. Since most parsers are relatively permissive, (1) tends to admit a variety of repairs that are syntactically valid but otherwise unnatural or practically useless in the absence of further semantic constraints or a strong ranking metric.

The latter evaluation considers both human preferences and syntactic validity. While human repairs are generally more challenging to discover, next to genuine user telemetry, they are arguably the best surrogate for human preference. Although pragmatism generally favors (2), pairwise error-fix datasets can be unavailable in minority languages and can be misleading when considering out-of-distribution errors. We therefor consider (1) for evaluating throughput, soundness and completeness of the solver, and (2) to measure its precision and latency under realistic usage conditions.

We evaluate Tidyparse on an extensive suite of syntax errors in Python code, comparing it with Seq2Parse [7] and Break-It-Fix-It (BIFI) [10], two transformer-based syntax correction models. We compare our Precision@{1, 5, 10, All} with repairs of varying difficulty, from single- to multi-token insertions, deletions and substitutions, and measure throughput and wall-clock latency across a wide variety of error categories and timeout values. We also evaluate Tidyparse on synthetic errors in synthetic languages and synthetic errors in natural programming languages to evaluate our robustness on rare grammars and error categories.

5.1 Hypothesis

By carefully leveraging parallelism, fast parsing and a cheap ranking function, we hypothesize it is (1) possible to achieve a higher overall Precision@{5+} at a lower latency than the existing transformer-based repair models for repairs up to three tokens, and furthermore can do so without using any neural network whatsoever. Additionally, we hypothesize it is (2) possible to retrain a smaller, faster transformer-based model to score and rerank the repairs discovered by Tidyparse, and this hybrid model can be used to attain SotA precision, throughput and wall-clock latency across all relevant categories, thereby validating the merits of pragmatism.

5.2 Evaluation Setup

Using a pairwise dataset of human syntax errors and repairs from Python snippets on StackOverflow [9], we preprocess the data to include only syntax pairs whose broken form is rejected by the true parser, and whose repair is accepted by the true parser. This dataset is then abstracted using a lexical tokenization, deduplicated, and all pairs whose Levenshtein distance is greater than three lexical edits are removed.

For our synthetic evaluation, we synthetically corrupt parseable single-line statements in Java and Kotlin source code by sampling edits uniformly at random from the Levenshtein ball and rejecting any edits accepted by the true parser. Treating the original source code as the ground-truth repair, we then measure the average latency to recover the original statement, and the precision at varying latency cutoffs.

External validity is maximized by evaluating on a large dataset of human syntax errors and repairs across a variety of programming languages and grammars. We qualitatively control for bias by using an identical test set for all models and balancing across edit distances and error categories to model a diversity of real-world error-correction scenarios. Although pairwise repair data is limited due to the paucity of erroneous source code committed to public GitHub repositories and reported on StackOverflow, we can ensure the test set is largely representative of real world syntax errors. In the future, we hope to conduct a proper user study.

References

[1] David Chiang, Peter Cholak, and Anand Pillay. 2023. Tighter Bounds on the Expressivity of Transformer Encoders. *arXiv preprint arXiv:2301.10743* (2023).

[2] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 1979. Social Processes and Proofs of Theorems and Programs. *Commun. ACM* 22, 5 (may 1979), 271–280. https://doi.org/10.1145/359104.359106

[3] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jian, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D Hwang, et al. 2023. Faith and Fate: Limits of Transformers on Compositionality. *arXiv preprint arXiv:2305.18654* (2023).

[4] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[5] Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. 2022. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749* (2022).

[6] William Merrill, Ashish Sabharwal, and Noah A Smith. 2022. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics* 10 (2022), 843–856.

[7] Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, and Ranjit Jhala. 2022. Seq2Parse: neurosymbolic parse error repair. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1180–1206.

[8] Leslie G Valiant. 1975. General context-free recognition in less than cubic time. *Journal of computer and system sciences* 10, 2 (1975), 308–315. http://people.csail.mit.edu/virgi/6.s078/papers/valiant.pdf

[9] Alexander William Wong, Amir Salimi, Shaiful Chowdhury, and Abram Hindle. 2019. Syntax and Stack Overflow: A methodology for extracting a corpus of syntax errors and fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 318–322.

[10] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*. PMLR, 11941–11952.