An Edit Calculus for Probabilistic Program Repair

Breandan Considine

School of Computer Science
McGill University
Montreal, Quebec, Canada

January 7, 2025

A thesis proposal submitted to McGill University in partial
fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

We introduce an edit calculus for correcting syntax errors in arbitrary context-free languages, and by extension, any programming language with a context-free grammar. Syntax errors with a small repair seldom have many unique small repairs, which can usually be enumerated up to a small edit distance then quickly reranked. Our work places a heavy emphasis on precision: the enumerated set must contain every possible repair within a given radius and no invalid repairs. To do so, we construct a grammar representing the language intersection between a Levenshtein automaton and a context-free grammar, then decode it in order of probability. This produces an ordered set of repairs that contains with high probability the intended revision.

Abrégé

Official McGill Guidelines: La même chose en français.

Contribution

The work presented in this thesis is the sole result of the author's original research, except where otherwise indicated. The author has made the following contributions to the work presented in this thesis:

- The conception of syntax repair as a language intersection task.
- The adaptation and specialization of the Bar-Hillel construction to probabilistic program repair.
- The formalization of the program repair objective as a pragmatic language game between a human and a machine.
- The design and implementation of the probabilistic program repair system called Tidyparse.

Acknowledgements

Official McGill Guidelines: Among other acknowledgements, the student is required to declare the extent to which assistance (paid or unpaid) has been given by members of staff, fellow students, research assistants, technicians, or others in the collection of materials and data, the design and construction of apparatus, the performance of experiments, the analysis of data, and the preparation of the thesis (including editorial help).

- In addition, it is appropriate to recognize the supervision and advice given by the thesis supervisor(s) and advisors.

Contents

List of Figures

List of Tables

Terminology

Technical and vernacular collisions induce a strange semantic synesthesia, e.g., complete, consistent, kernel, reflexive, regression, regular, sound. The intension may be distantly related to standard English, but if one tries to interpret such jargon colloquially, there is no telling how far astray they will go. For this reason, we provide a glossary of terms to help the non-technical reader navigate the landscape of this thesis.

- **Automaton**: A mathematical model of computation that can occupy one of a finite number of states at any given time, and makes transitions between states according to a set of rules.
- **Deterministic**: A property of a system that, given the same input, will always produce the same output.
- **Grammar**: A set of rules that define the syntax of a language.
- **Language**: A set of words generated by a grammar. For the purposes of this thesis, the language can be finite or infinite.
- **Word**: A member of a language, consisting of a sequence of terminals. For the purposes of this thesis, a word is always finite.
- **Terminal**: A single token from an alphabet. For the purposes of this thesis, the alphabet is always finite.
- **Intersection**: The set of elements common to two or more sets.
- **Probabilistic**: A property of a system that, given the same input, may produce different outputs.
- **Theory**: A set of sentences in a formal language.

1

Introduction

> Pray, Mr. Babbage, if you put
> into the machine wrong figures,
> will the right answers come out?
>
> –Charles Babbage (1791–1871)

Computer programs are instructions for performing a chore that humans would rather avoid doing ourselves. In order to persuade the computer to do them for us, we must communicate our intention in a way that is plain and unambiguous. Programming languages are protocols for this dialogue, designed to enable programmers to conveniently express their intent and facilitate information exchange between humans and computers.

Programs are seldom written from left-to-right in one fell swoop. During the course of writing a program, the programmer often revisits and revises code as they write, sharing additional information and receiving feedback. Often, during this process, the programmer makes a mistake, causing the program to behave in an unexpected manner. These mistakes can manifest as a simple typographic or syntactic error, or a more subtle logical error.

To intercept these errors, programming language designers have adopted a convention for specifying valid programs, called a grammar, which serves two essential purposes. The first is to reject obviously ill-formed programs, and the second is to parse the source code into an intermediate representation that can be handled by a compiler. We will focus on the first case.

When a parser enters an invalid state, a chain of unfortunate events occurs. The compiler halts, raising an error message. To rectify this situation,

the programmer must pause their work, inspect the message, plan a fix, apply it, then try to remember what they were doing beforehand. The cognitive overhead of this simple but repetitive chore can be tiresome. To make matters worse, the error message may be unhelpful or challenging to diagnose.

Program repair attempts to address such errors by inferring the author's intent from an approximate solution. We can think of this as playing a kind of language game. Given an invalid piece of source code for some programming language, the objective of this game is to modify the code to satisfy the language specification. The game is won when the proposed solution is both valid and the author is satisfied with the result. We want to play this game as efficiently as possible, with as little human feedback as possible.

Prior work on program repair focuses on approximate or semidecision procedures. These methods are heuristic and often brittle, relying on statistical guarantees to locate probable repairs. Furthermore, they rely on a handcrafted set of often language-specific rules, which may not generalize to other programming languages. To our knowledge, no existing approach can repair programs in a language-agnostic way, or guarantee (1) soundness (2) naturalness and (3) completeness in a unified framework. Most are based on software engineering compromises, rather than formal language theory.

Our goal in this thesis is to introduce an edit calculus of program repair. Broadly, our approach is to repair faulty programs by combining probabilistic language models with exact combinatorial methods. We do so by reformulating the problem of program repair in the parlance of formal language theory. In addition to being a natural fit for syntax repair, this also allows us to encode and compose static analyses as grammatical specifications.

Program repair is a highly underdetermined problem, meaning that the validity constraints do not uniquely determine a solution. A proper theory of program repair must be able to resolve this ambiguity to infer the user's intent from an incomplete specification, and incrementally refine its guess as more information becomes available from the user.

This calculus we propose has a number of desirable properties. It is highly compositional, meaning that users can manipulate constraints on programs while retaining the algebraic closure properties, such as union, intersection, and differentiation. It is well-suited for probabilistic reasoning, meaning we can use any autoregressive language model to guide the repair process. It is also amenable to incremental repair, meaning that we can repair programs in a streaming fashion, while the user is typing.

To explain the virtues of our approach, we need some background. For-

mal languages are not always closed under set-theoretic operations, e.g., CFL $\cap$ CFL is not CFL in general. Let $\cdot$ denote concatenation, $*$ be Kleene star, and $\complement$ be complementation:

	$\cup$	$\cap$	$\cdot$	$*$	$\complement$
Finite[1]	✓	✓	✓	✓	✓
Regular[1]	✓	✓	✓	✓	✓
Context-free[1,2]	✓	✗[†]	✓	✓	✗
Context-sensitive[2]	✓	✓	✓	+	✓
Recursively Enumerable[2]	✓	✓	✓	✓	✗

For a background theory of program repair, we would like a language family that is (1) tractable, i.e., has polynomial recognition and search complexity and (2) reasonably expressive, i.e., can represent syntactic and some semantic properties of real-world programming languages.

[†] However, CFLs are closed under intersection with regular languages.

1.1 Examples

Syntax errors are a familiar nuisance for programmers, arising due to a variety of factors, from inexperience, typographic error, to cognitive load. Often the mistake itself is simple to fix, but manual correction can disrupt concentration, a developer's most precious and fickle resource. Syntax repair attempts to automate the correction process by trying to anticipate which program, out of the many possible alternatives, the developer actually intended to write.

Taking inspiration from formal and statistical language modeling alike, we adapt a construction from Bar-Hillel [BHPS61] for formal language intersection, to the problem of syntactic program repair. Our work shows this approach, while seemingly intractable, can be scaled up to handle real-world program repair tasks. We will then demonstrate how, by decoding the Bar-Hillel construction with a simple Markov model, it is possible to predict human syntax repairs with the accuracy of large language models, while retaining the correctness and interpretability of classical repair algorithms.

In particular, we consider the problem of ranked syntax repair under a finite edit distance. We experimentally show it is possible to attain a signifi-

cant advantage over state-of-the-art neural repair techniques by exhaustively retrieving every valid Levenshtein edit in a certain distance and scoring it. Not only does this approach guarantee both soundness and completeness, we find it also improves precision when ranking by naturalness.

Our primary technical contributions are (1) the adaptation of the Levenshtein automaton and Bar-Hillel construction to syntax repair and (2) a method for enumerating or sampling valid sentences in finite context-free languages in order of naturalness. The efficacy of our technique owes to the fact it does not synthesize likely edits, but unique, fully-formed repairs within a given edit distance. This enables us to suggest correct and natural repairs with far less compute and data than would otherwise be required by a large language model to attain the same precision.

1.2 Example

Syntax errors can usually be fixed with a small number of edits. If we assume the intended repair is small, this imposes strong locality constraints on the space of possible edits. For example, let us consider the following Python snippet: `v = df.iloc(5:, 2:)`. Assuming an alphabet of just a hundred lexical tokens, this tiny statement has millions of possible two-token edits, yet only six of those possibilities are accepted by the Python parser:

(1) `v = df.iloc(5:, 2,)` (3) `v = df.iloc(5[:, 2:])` (5) `v = df.iloc[5:, 2:]`

(2) `v = df.iloc(5), 2()` (4) `v = df.iloc(5:, 2:)` (6) `v = df.iloc(5[:, 2])`

With some semantic constraints, we could easily narrow the results, but even in their absence, one can probably rule out (2, 3, 6) given that `5[` and `2(` are rare bigrams in Python, and knowing `df.iloc` is often followed by `[`, determine (5) is the most likely repair. This is the key insight behind our approach: we can usually locate the intended fix by exhaustively searching small repairs. As the set of small repairs is itself often small, if only we had some procedure to distinguish valid from invalid patches, the resulting solutions could be simply ranked by likelihood.

The trouble is that any such procedure must be highly efficient to be practically useful for developers. We cannot afford to sample the universe of possible d-token edits, then reject invalid samples – assuming it takes just 10ms to generate and check each sample, (1-6) could take 24+ hours to find. The hardness of brute-force search grows exponentially with edit distance,

sentence length and alphabet size. We will need a more efficient procedure for sampling all and only small valid repairs.

We will first proceed to give an informal intuition behind our method, then formalize it in the following sections. At a high level, our approach is to construct a language that represents all syntactically valid patches within a certain edit distance of the invalid code fragment. To do so, we first lexicalize the invalid source code, which simply abstracts over numbers and named identifiers, but retains all other keywords.

From the lexical string, we build an automaton that represents all possible strings within a certain edit distance. Then, we proceed to construct a synthetic grammar, recognizing all strings in the intersection of the programming language and the edit ball. Finally, this grammar is reduced to a normal form and decoded with the help of a statistical model to produce a list of suggested repairs.

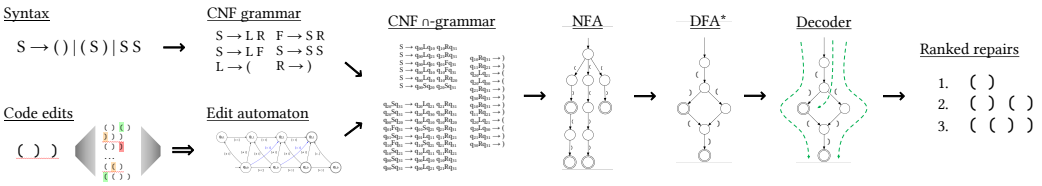


Figure 1.1: Simplified dataflow. Given a grammar and broken code fragment, we create an automaton generating the language of small edits, then construct a grammar representing the intersection of the two languages. This grammar can be converted to a finite automaton, determinized, then decoded to produce an explicit list of repairs.

The advantage of using this approach for syntax repair is that it performs the intersection not in the set domain, but in the domain of grammars and automata, thereby avoiding a great deal of useless work that would be necessary were we to compute the intersection by sampling and rejecting edits. Furthermore, it is a complete decision procedure, meaning that every possible repair within a given edit distance will eventually be found by the decoder.

To explain each component, we will require a more detailed background on the formal language theory, then we can describe the full Bar-Hillel construction and our specialization to Levenshtein automata intersections. But first, we will cover some related literature on program repair.

2

Related Literature

Translating ideas into computer programs demands a high degree of precision, as computers have strict criteria for admitting valid programs. These constraints act as a failsafe against faulty programs and runtime errors, but can be tedious to debug. During the editing process, constraints are invariably violated by the hasty or inexperienced programmer, requiring manual repair. To assist with this task, automated program repair (APR) attempts to generate possible revisions from which the author may choose. This subject has been closely investigated by programming language research and treated in a number of existing literature reviews [Mon18, LGPRC21]. We direct our attention primarily towards syntax repair, which attempts to fix parsing errors, the earliest stage in program analysis.

2.1 Syntactic program repair

Spellchecking is an early precursor to syntax repair that was originally developed for word processing and seeks to find, among a finite dictionary, the most likely intended revision of a misspelled word [KCG90]. Similarly, syntax repair considers the case where this dictionary is not necessarily finite, but rather generated by a grammar representing a potentially infinite collection of words called a *language*. This has applications in natural language processing [BYQ+23], although we are primarily interested in programming languages. In the case of programming language syntax, the language and corresponding grammar is typically context-free [CS59].

Various methods have been proposed to handle syntactic program errors, which have been a longstanding open problem since the advent of context-free languages. In 1972, Aho and Peterson [AP72] first introduce an algorithm

that returns a syntactically valid sequence whose distance from the original sequence is minimal. Their method guarantees that a valid repair will be found, but only generates a single repair and does attempt to optimize the naturalness of the generated solution, only the proximity and validity.

While algorithmically elegant, deterministic repair methods lack the flexibility to model the natural features of source code. It does not suffice to merely suggest parseable repairs, but a pragmatic solution must also generate suggestions a human is likely to write in practice. To model code conventions, stylistic patterns and other programming idioms that are not captured in the formal grammar, researchers have adopted techniques from natural language processing, in particular advances in neural language modeling.

Recent work attempts to use neural language models to generate probable fixes. For example, Yasunaga et al. [YL21] use an unsupervised method that learns to synthetically corrupt natural source code (simulating a typographic noise process), then learn a second model to repair the broken code, using the uncorrupted source as the ground truth. This method does not require a parallel corpus of source code errors and fixes, but can produce a misaligned noise model and fail to generalize to out-of-distribution samples. It also does not guarantee the generated fix is valid according to the grammar.

Sakkas et al. [SEG+22] introduce a neurosymbolic model, Seq2Parse, which adapts the Early parser [Ear70] with a learned PCFG and a transformer-classifier to predict error production rules. This approach aims to generate only sound repairs, but lacks the ability to generate every valid repair within a given edit distance. While this has the benefit of better interpretability than end-to-end neural repair models, it is not clear how to scale up this technique to accommodate additional test-time compute.

Neural language models are adept at learning statistical patterns, but often sacrifice validity, precision or latency. Existing neural repair models are prone to misgeneralize and hallucinate syntactically invalid repairs and do not attempt to sample from the space of all and only valid repairs. As a consequence, they have difficulty with inference scaling, where additional test-time compute does not translate to a significant improvement on the target domain. Furthermore, even if theoretically sound, the generated samples may not even be syntactically valid, as we observe in practice.

Our work aims to address all of these concerns. We try to generate every nearby valid program and prioritize the solutions by naturalness, while ensuring response time is tolerable. In other words, we attempt to satisfy soundness, completeness, naturalness and latency simultaneously.

2.2 Semantic program repair

Automated program repair is either implicitly or explicitly defined over a *search space*, which is the space of all possible solutions. Previously, we looked at a very coarse-grained approximation, based on syntactic validity. In practice, one might wish to layer additional refinements on top of these syntactic constraints, corresponding to so-called *semantic* properties such as type-soundness or well-formedness. This additional criteria lets us *prune* invalid solutions or *quotient* the search space by an equivalence relation, often vastly reducing the set of candidate repairs.

Semantically valid program representations are typically framed as a subset of the syntactically valid ones. In some cases, the syntax of a programming language is not even context-free, in which case syntax repair may be viewed as a kind of semantic repair. The C/C++ [MN04] language, for example, implements a so-called lexer-hack, introducing type names into the symbol table used for parsing. Though generally considered in poor taste from a language design perspective, handling these kinds of scenarios is important for building practical developer tools.

One approach to handling more complex synthesis tasks uses *angelic execution* to generate and optimistically evaluate execution paths. Shi et al.'s FrAngel [SSL19] is a particular example of this approach, which uses component-based program synthesis in conjunction with angelic nondeterminism to repair a broken program. The idea of angelic execution can be retraced to Bodík et al. [BCG+10] who attribute the original idea to Floyd's nondeterministic `choice` operator [Flo67]. In the context of semantic program repair, angelic execution has been successfully developed for program synthesis by Singh et al. [SGSL13] for auto-grading and providing feedback on programming assignments.

The idea of angelic execution has also been employed to great effect to assist with automated program repair. In particular, Long & Rinard [LR16] introduce a tool called Prophet and use a very similar evaluation to ours to generate and rank candidate patches from a search space, then rank the generated patches according to a learned probabilistic model. Chandra et al. [CTBB11] also use angelic execution to guide the search for semantic repairs. Both systems bypass the syntax checking stage and search directly for semantic repairs, using a set of test cases as guard conditions to reject invalid candidates. Their approach is closely related to fault localization and mutation testing in the software engineering literature.

3

Formal Language Theory

In computer science, it is common to conflate two distinct notions for a set. The first is a collection sitting on some storage device, e.g., a dataset. The second is a lazy construction: not an explicit collection of objects, but a representation that allows us to efficiently determine membership on demand. This representation lets us describe infinite sets without requiring an infinite amount of storage. Inclusion then, instead of being simply a lookup query, becomes a decision procedure. This is the basis of formal language theory.

The representation we are chiefly interested in is called a *grammar*, a common metanotation for specifying the syntactic constraints on programs, shared by nearly every programming language. Programming language grammars are overapproximations to the true language, but provide a reasonably detailed specification for rejecting invalid programs and parsing valid ones.

Formal languages are arranged in a hierarchy of containment, where each language family strictly contains its predecessors. On the lowest level of the hierarchy are finite languages. Type 3 contains finite and infinite languages generated by a regular grammar. Type 2 contains context-free languages, which admit parenthetical nesting. Supersets, such as the recursively enumerable sets, are Type 0. There are other kinds of formal languages, such as logics and circuits, which are incomparable with the Chomsky hierarchy.

Most programming languages leave level 2 after the parsing stage, and enter the realm of type theory. At this point, compiler authors layer additional semantic refinements on top of syntax, but must deal with phase ordering problems related to the sequencing of such analyzers, breaking commutativity and posing challenges for parallelization. This lack of compositionality is a major obstacle to the development of modular static analyses.

The advantage of dealing with formal language representations is that we can reason about them algebraically. Consider the context-free grammar: the arrow $\to$ becomes an $=$ sign, $|$ becomes $+$ and AB becomes $A \times B$. The ambiguous Dyck grammar, then, can be seen as a system of equations.

$$S \to () \mid (S) \mid SS \iff f(x) = x^2 + x^2 f(x) + f(x)^2 \tag{3.1}$$

We will now solve for $f(x)$, giving us the generating function for the language:

$$0 = f(x)^2 + x^2 f(x) - f(x) + x^2 \tag{3.2}$$

Now, using the quadratic equation, where $a = 1, b = x^2 - 1, c = x^2$, we have:

$$f(x) = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-x^2 + 1 \pm \sqrt{x^4 - 6x^2 + 1}}{2} \tag{3.3}$$

Note there are two solutions, but only one where $\lim_{x \to 0} = 1$. From the ordinary generating function (OGF), we also have that $f(x) = \sum_{n=0}^{\infty} f_n x^n$. Expanding $\sqrt{x^4 - 6x^2 + 1}$ via the generalized binomial theorem, we have:

$$f(x) = (1+u)^\alpha = \sum_{k=0}^{\infty} \binom{\alpha}{k} u^k \tag{3.4}$$

$$= \sum_{k=0}^{\infty} \binom{\frac{1}{2}}{k} (x^4 - 6x^2)^k \text{ where } u = x^4 - 6x^2 \tag{3.5}$$

Now, to obtain the number of ambiguous Dyck trees of size n, we can extract the x^n-th coefficient using the binomial series:

$$[x^n] f(x) = [x^n] \frac{-x^2 + 1}{2} + \frac{1}{2} [x^n] \sum_{k=0}^{\infty} \binom{\frac{1}{2}}{k} (x^4 - 6x^2)^k \tag{3.6}$$

$$[x^n] f(x) = \frac{1}{2} \binom{\frac{1}{2}}{n} [x^n](x^4 - 6x^2)^n = \frac{1}{2} \binom{\frac{1}{2}}{n} [x^n](x^2 - 6x)^n \tag{3.7}$$

We can use this technique, first described by Flajolet & Sedgewick [Fla09], to count the number of trees of a given size or distinct words in an unambiguous CFG. This lets us understand grammars as a kind of algebra, which is useful for enumerative combinatorics on words and syntax-guided synthesis.

Naturally, like algebra, there is also a kind of calculus to formal languages. Janusz Brzozowski [Brz64] introduced the derivative operator for regular languages, which can be used to determine membership, and extract subwords from the language. This operator has been extended to CFLs by Might et al. [MDS11], and is the basis for a family of elegant parsing algorithms.

The Brzozowski derivative has an extensional and intensional form. Extensionally, we have $\partial_a L = \{b \in \Sigma^* \mid ab \in L\}$. Intensionally, we have an induction over generalized regular expressions (GREs), which are a superset of regular expressions that supports intersection and negation.

$$
\begin{array}{lll}
\partial_a(\ \varnothing\ ) = \varnothing & & \delta(\ \varnothing\ ) = \varnothing \\
\partial_a(\ \varepsilon\ ) = \varnothing & & \delta(\ \varepsilon\ ) = \varepsilon \\
\partial_a(\ a\ ) = \varepsilon & & \delta(\ a\ ) = \varnothing \\
\partial_a(\ b\ ) = \varnothing \text{ for each } a \neq b & & \delta(\ R^*\ ) = \varepsilon \\
\partial_a(\ R^*\ ) = (\partial_x R) \cdot R^* & & \delta(\ \neg R\ ) = \varepsilon \text{ if } \delta(R) = \varnothing \\
\partial_a(\ \neg R\ ) = \neg \partial_a R & & \delta(\ \neg R\ ) = \varnothing \text{ if } \delta(R) = \varepsilon \\
\partial_a(\ R \cdot S\ ) = (\partial_a R) \cdot S \vee \delta(R) \cdot \partial_a S & & \delta(\ R \cdot S\ ) = \delta(R) \wedge \delta(S) \\
\partial_a(\ R \vee S\ ) = \partial_a R \vee \partial_a S & & \delta(\ R \vee S\ ) = \delta(R) \vee \delta(S) \\
\partial_a(\ R \wedge S\ ) = \partial_a R \wedge \partial_a S & & \delta(\ R \wedge S\ ) = \delta(R) \wedge \delta(S)
\end{array}
$$

Similar to sets, it is possible to combine languages by manipulating their grammars, mirroring the setwise notions of union, intersection, complementation and difference over languages. These operations are convenient for combining, for example, syntactic and semantic constraints on programs. For example, we might have two grammars, G_a, G_b representing two properties that are desirable or necessary for a program to be considered valid.

Like all representations, grammars are themselves a trade-off between expressiveness and efficiency. It is possible to represent the same finite set with multiple representations of varying complexity. For example, the set of strings containing ten or fewer balanced parentheses can be expressed as a finite automaton containing millions of states, or a simple language conjunction containing a few productions, e.g., $\mathcal{L}\Big(S \to () \mid (S) \mid SS\Big) \cap \Sigma^{[0,10]}$.

The choice of representation is heavily usage-dependent. For example, if we are interested in recognition, we might favor a disjoint representation, allowing properties to be checked independently without merging, whereas if we are interested in generation or deciding non-emptiness, we might prefer a

unified representation which can be efficiently sampled without rejection.

Union, concatenation and repetition are all mundane in the theory of formal languages. Intersection and negation are more challenging concepts to borrow from set theory, and do not translate naturally into the Chomsky hierarchy. For example, the intersection of two CFLs is Turing Complete, but the intersection of a CFL and a regular language is a CFL.

Deciding intersection non-emptiness (INE) of a finite collection of finite automata is known to be PSPACE-complete [Koz77]. It is still unknown whether a faster algorithm than the product construction exists for deciding INE of just two finite automata.

The textbook algorithm proceeds as follows: create an automaton containing the cross-product of states, and simulate both automata in lockstep, creating arcs between states that are co-reachable. If a final state is reachable in the product automaton, then the intersection is non-empty. This requires space proportional to the Cartesian product of the two states.

Figure 3.1: TODO: depict product construction for finite automata here.

The goal of this thesis is to speed up the product construction by leveraging (1) parameterized complexity (2) pruning and (3) parallelization to speed up the wallclock runtime of the product construction and generalize it to CFG-REG intersections. We show it is possible to decide INE in realtime for intersections with Levenshtein automata and build a tool to demonstrate it on real-world programming languages and grammars.

Finally, we show a probabilistic extension of the REG-CFL product construction, which can be used to decode the top-K most probable words in the intersection of two formal languages. This is useful for languages with both formal and natural characteristics, where we might want to find the most natural word that satisfies multiple constraints, such as being a valid repair with fewer than k edits whose probability is maximized.

4

Deterministic Program Repair

Parsimony is a guiding principle in program repair that comes from the 14th century Fransiscan friar named William of Ockham. In keeping with the Fransiscan minimalist lifestyle, Ockham's principle basically says that when you have multiple hypotheses, the simplest one is the best. It is not precisely clear what "simple" ought to mean in the context of program repair, but a first-order approximation is to strive for the smallest number of changes required to transform an invalid program into a valid one.

Levenshtein distance is one such metric for measuring the minimum number of changes between two strings. First proposed by the Soviet scientist Vladimir Levenshtein, it quantifies how many insertions, deletions, and substitutions are required to transform one string into another. Conveniently, there is an automaton, called the Levenshtein automaton [SM02], that recognizes all strings within a given edit distance of a given string. We can use this automaton to locate the positions and contents of the most likely repair consistent with the observed program and the grammar.

The closure of CFLs under intersection with regular languages was first established in 1961 by Bar-Hillel, implying the existence of a context-free grammar representing the conjunction of any finite automaton and context-free grammar. Such a construction was given by Salomaa in 1973, who provides a direct, but inefficient, construction. In our work, we refine this construction to intersections with Levenshtein automata, which recognize all and only strings within a given edit distance of a reference string. Using this refinement, we demonstrate it is feasible to repair multiline syntax errors in practical programming languages.

Given the source code for a computer program $\underset{\sim}{\sigma}$ and a grammar G, our goal is to find every valid string σ consistent with the grammar G and within a certain edit distance, d. Consider the language of valid strings within a given Levenshtein distance from a reference string $\underset{\sim}{\sigma}$. We can intersect the language given by the Levenshtein automaton with the language of all valid programs given by the grammar G. The resulting language, $\mathcal{L}(G_\cap)$ will contain every repair within the designated edit distance.

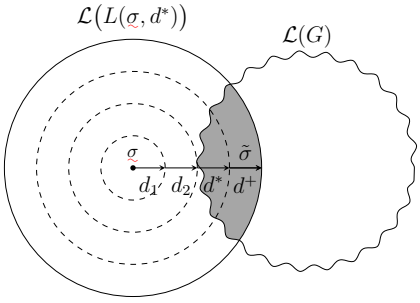


Figure 4.1: CFL intersection with the local edit region of a given broken code snippet.

4.1 Levenshtein Automata

Levenshtein automata are finite automata that recognize all and only strings within a given edit distance of another string by permitting insertions, deletions, and substitutions. For instance, suppose we have the input, `( ) )`, and wish to find nearby or parsimonious edits. To represent the language of parsimonious edits, we can construct the Levenshtein-1 automaton accepting every string that can be formed by inserting, substituting or deleting a single parenthesis. We depict this automaton in Figure 4.2.

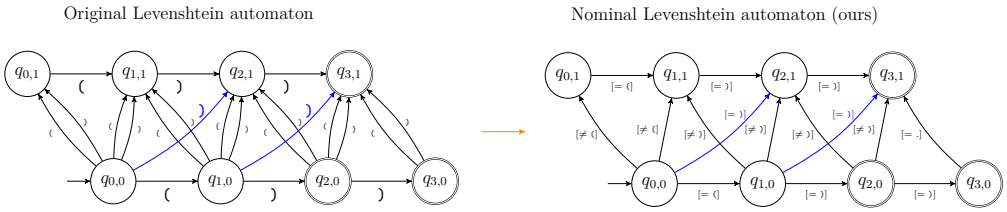


Figure 4.2: Automaton recognizing every 1-edit patch. We nominalize the original automaton, ensuring upward arcs denote a mutation, and use a symbolic predicate, which deduplicates parallel arcs in large alphabets.

The original automaton is nondeterministic, containing an upward arc for each token. This can be avoided with a simple modification that matches an inequality predicate. The machine enters at $q_{0,0}$ and at each step, accepts the labeled token. Final states are encircled twice, denoting that any trajectory ending at such a state is considered valid. When the edit distance grows larger, we introduce some additional arcs to handle multi-token deletions,

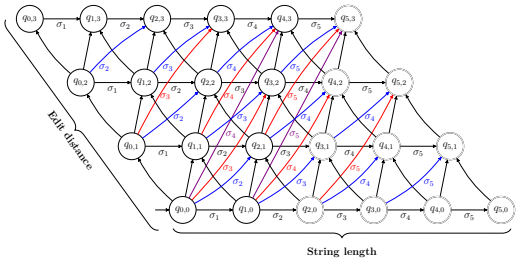


Figure 4.3: NFA recognizing Levenshtein $L(\sigma : \Sigma^5, 3)$.

but the overall picture remains unchanged. We depict a 3x5 automaton recognizing 3-edit patches of a length-5 string in Figure 4.3.

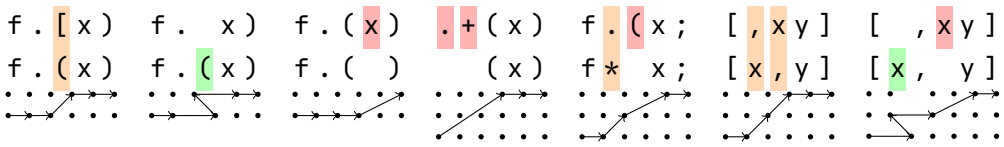
Here, a pattern begins to emerge: the automaton is a grid of states, with each horizontal arc consuming a token in the original string, and upwards arcs recognizing mutations. Traversing a vertical arc corresponds to an insertion or substitution, and a diagonal arc corresponds to a deletion. Levenshtein automata can also be defined as a set of inference rules, which generalize this picture to arbitrary length strings and edit distances. The indices are a bit finicky, but the rules are otherwise straightforward.

$$\frac{s \in \Sigma \quad i \in [0, n] \quad j \in [1, d_{\max}]}{(q_{i,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nwarrow \qquad \frac{s \in \Sigma \quad i \in [1, n] \quad j \in [1, d_{\max}]}{(q_{i-1,j-1} \xrightarrow{s} q_{i,j}) \in \delta} \nwarrow\!\!\!\!\nearrow$$

$$\frac{i \in [1, n] \quad j \in [0, d_{\max}]}{(q_{i-1,j} \xrightarrow{\sigma_i} q_{i,j}) \in \delta} \rightarrow \qquad \frac{d \in [1, d_{\max}] \quad i \in [d+1, n] \quad j \in [d, d_{\max}]}{(q_{i-d-1,j-d} \xrightarrow{\sigma_i} q_{i,j}) \in \delta} \cdots$$

$$\frac{}{q_{0,0} \in I} \textsc{Init} \qquad \frac{q_{i,j} \in Q \qquad |n - i + j| \leq d_{\max}}{q_{i,j} \in F} \textsc{Done}$$

Each rule recognizes a specific type of edit. $\nwarrow$ handles insertions, $\nwarrow\!\!\!\!\nearrow$ handles substitutions and $\cdots$ handles deletions of one or more terminals. Let us consider some illustrative cases depicting the edit trajectory with specific Levenshtein alignments. Note that the trajectory may not be unique.



4.2 The Bar-Hillel Construction

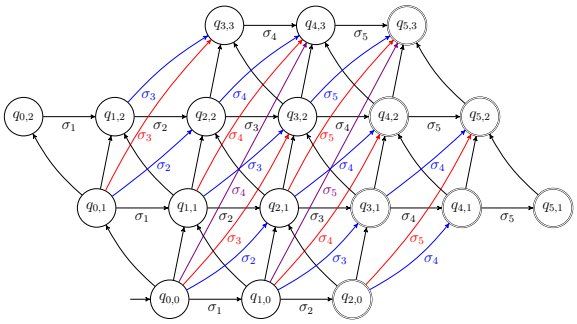
The Bar-Hillel construction is a method for conjoining a context-free grammar with a finite automaton. First proposed by Bar-Hillel in 1961, and later realized by Salomaa in 1973, this construction is based on the idea of a product automaton, generalized to a grammar. It consists of three rules:

$$\frac{q \in I \quad r \in F}{\big(S \to qSr\big) \in P_\cap}\,\surd \qquad \frac{(A \to a) \in P \qquad (q \xrightarrow{a} r) \in \delta}{\big(qAr \to a\big) \in P_\cap}\,\uparrow$$

$$\frac{(w \to xz) \in P \qquad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap}\,\bowtie$$

4.2.1 State elimination

The $\bowtie$ rule has a strong dependency on the number of states. So, the primary target is to first reduce the number of states in the Levenshtein automaton. We can reduce the number of states without compromising the integrity of the Bar-Hillel construction by pruning states which are obviously inaccessible. For example, let us consider the following scenario, where $G = S \to \texttt{(}\ S\ \texttt{)} \mid \texttt{[}\ S\ \texttt{]} \mid S\ \texttt{+}\ S \mid \texttt{1}$ and $\sigma = \texttt{[}\ \texttt{(}\ \texttt{+}\ \texttt{)}\ \texttt{]}$. If we can establish $\mathcal{L}(\_\ \_\ \texttt{+}\ \texttt{)}\ \texttt{]}) = \varnothing \wedge \mathcal{L}(\_\ \_\ \_\ \texttt{)}\ \texttt{]}) \neq \varnothing$ and $\mathcal{L}(\texttt{[}\ \texttt{(}\ \texttt{+}\ \_\ \_) = \varnothing \wedge \mathcal{L}(\texttt{[}\ \texttt{(}\ \_\ \_\ \_) \neq \varnothing$, then:



We can determine the monoedit bounds by conducting a binary search for the rightmost and leftmost states with an empty porous completion problem, and remove all states from the automaton which absorb trajectories that are incompatible. Similar bounds can be established for multi-edit locations.

Now, let us consider the Parikh constraints.

4.2.2 Parikh Refinements

To identify superfluous q, v, q' triples, we define an interval domain that soundly overapproximates the Parikh image, encoding the minimum and maximum number of terminals each nonterminal must and can generate, respectively. Since some intervals may be right-unbounded, we write $\mathbb{N}^* = \mathbb{N} \cup \{\infty\}$ to denote the upper bound, and $\Pi = \{[a, b] \in \mathbb{N} \times \mathbb{N}^* \mid a \leq b\}^{|\Sigma|}$ to denote the Parikh image of all terminals.

Definition 1: Parikh mapping of a nonterminal

Let $p : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ be the Parikh operator [Par66], which counts the frequency of terminals in a string. We define the Parikh map as a function, $\pi : V \to \Pi$, returning the smallest interval such that $\forall \sigma : \Sigma^*, \forall v : V,\ v \Rightarrow^* \sigma \vdash p(\sigma) \in \pi(v)$.

The Parikh mapping computes the greatest lower and least upper bound of the Parikh image over all strings in the language of a nonterminal. The infimum of a nonterminal's Parikh interval tells us how many of each terminal a nonterminal *must* generate, and the supremum tells us how many it *can* generate. Likewise, we define a similar relation over NFA state pairs:

Definition 2: Parikh mapping of NFA states

We define $\pi : Q \times Q \to \Pi$ as returning the smallest interval such that $\forall \sigma : \Sigma^*, \forall q, q' : Q,\ q \overset{\sigma}{\Longrightarrow} q' \vdash p(\sigma) \in \pi(q, q')$.

Next, we will define a measure on Parikh intervals representing the minimum total edits required to transform a string in one Parikh interval to a string in another, across all such pairings.

Definition 3: Parikh divergence

Given two Parikh intervals $\pi, \pi' : \Pi$, we define the divergence between them as $\pi \parallel \pi' = \sum_{n=1}^{|\Sigma|} \min_{(i,i') \in \pi[n] \times \pi'[n]} |i - i'|$.

We know that if the Parikh divergence between two intervals is nonzero, those intervals must be incompatible as no two strings, one from each Parikh interval, can be transformed into the other with fewer than $\pi \parallel \pi'$ edits.

Definition 4: Parikh compatibility

Let q, q' be NFA states and v be a CFG nonterminal. We call $\langle q, v, q' \rangle : Q \times V \times Q$ *compatible* iff their divergence is zero, i.e., $v \lhd qq' \iff \big(\pi(v) \parallel \pi(q, q')\big) = 0$.

For efficiency, Parikh compatibility can be precomputed for each $Q \times V \times Q$ triple and reused for each synthetic production. Finally, we are ready to define the modified Bar-Hillel construction:

Definition 5: Modified Bar-Hillel construction

Let w, x, z be nonterminals in a CNF CFG and p, q, r be states in an FSA. We modify the $\bowtie$ rule from the BH construction as follows:

$$\frac{w \lhd pr \quad x \lhd pq \quad z \lhd qr \qquad (w \to xz) \in P \qquad p, q, r \in Q}{\big(pwr \to (pxq)(qzr)\big) \in P_\cap} \;\hat{\bowtie}$$

Once constructed, we normalize $G_\cap$ by removing unreachable and non-generating productions [FU15] to obtain $G'_\cap$, which is a recognizer for the admissible set, i.e., $\mathcal{L}(G'_\cap) = \ell_\cap$. Note, the original BH construction and our adapted version both reduce to the same CNF, $G'_\cap$, but normalization becomes significantly more tractable for large intersections, as far fewer useless productions are instantiated to only later be removed during normalization. This modified rule is not specific to Levenshtein automata and can be used to accelerate any FSA-CFG intersection.

5

Probabilistic Program Repair

As we have seen, the problem of program repair is highly underdetermined. To resolve this ambiguity, we will use a probabilistic model to induce a distribution over the language of valid programs. This distribution will guide the repair process by assigning a likelihood to each possible repair. Then, taking the maximum over all possible repairs, we can find the most likely repair consistent with the constraints and the observed program.

Specifically, we will define an ordering over strings by their likelihood under the probabilistic model. We then define a suggested repair as the most likely string consistent with the observed program and the grammar. We factorize the probability of a string as the product of the probability of each token in the string, conditioned on its prefix. This allows us to compute the joint probability in a left-to-right fashion.

This probabilistic model will generally admit programs that are locally probable, but globally inconsistent with the grammar. To enforce syntactic validity, we will use the probabilistic language model to "steer" a generative sampler through the automaton representing the repair language. This has two advantages: first, it allows us to sample from the repair language incrementally, and second, it ensures that subsequences with high probability are retrieved first, and all trajectories are syntactically valid.

We will consider two kinds of probabilistic models: a constrained Markov model and an unconstrained transformer-based neural network trained on program repair, then evaluate the performance of these models on a syntax repair benchmark consisting of pairwise program transformations. As we will show, the constrained Markov model is able to achieve state-of-the-art precision on blind prediction of the lexical sequence.

Here we give each model 5k+ syntax repairs of varying lengths and Levenshtein distances and measure the precision at varying cutoffs. For example, if the ground truth syntax repair was contained in the top 10 results for half of the repair instances, the model's P@10 would be 50%.

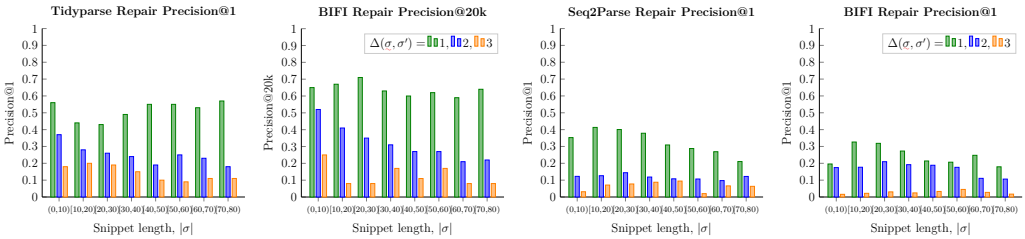


Figure 5.1: Total repair precision across the entire test set.

If we give it an equivalent number of samples, the constrained Markov model attains an even wider margin.

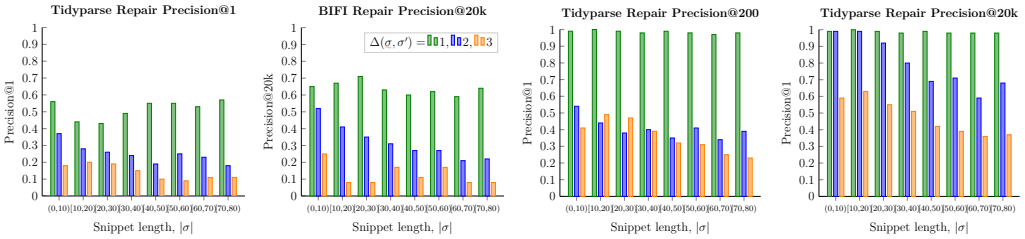


Figure 5.2: Sample efficiency increases sharply at larger precision intervals.

Next, we measure latency, which attains state-of-the-art precision at about 10 seconds, and additional time results in higher precision.

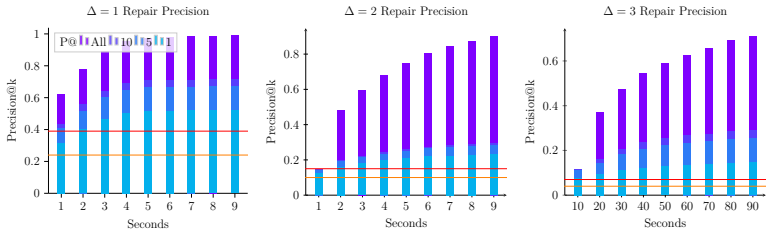


Figure 5.3: Latency benchmarks. Note the varying axis ranges. The red line marks Seq2Parse and the orange line marks BIFI's Precision@1.

For Precision@k, we measure the precision of our model at top-k prediction out of all instances presented, regardless of outcome. Four outcomes are possible in each repair instance, each a strict superset of the successor.

1. $|G_\cap| < \textsc{MaxHeap}$: the intersection grammar fits in memory
2. $\sigma' \in \mathcal{L}(G_\cap)$: the true repair is recognized by the intersection grammar
3. $\textsc{Dec}(G_\cap) \rightsquigarrow \sigma'$: the true repair is sampled by the decoder
4. $\textsc{Rank}(\sigma') < K$: the top-K sorted results contain the true repair

Repair cases that pass all four are the ideal, meaning the true repair was sampled and ranked highly, but (4) often fails. This indicates the decoder drew the true repair but was not discerning enough to realize its importance. Cases that fail (2) mean the decoder had the opportunity to, but did not actually draw the true repair, which occurs when the intersection language is too large to fully explore. In rare cases, the decoder was incapable of sampling the true repair, as the JVM ran out of memory. Below, we give a summary of distribution over outcomes across the test set.

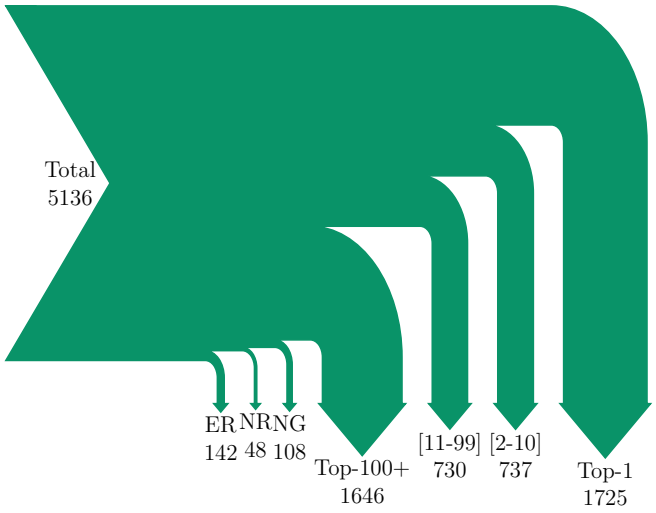


Figure 5.4: Summarized repair outcomes from the SO Python dataset. (ER=Error, NR=Not recognized, NG=Not generated). Time: ∼10h on M1.

Here, we plot the ground truth next token's log mean rank in the constrained and unconstrained decoding settings, across normalized snippet positions. That is, we take a trained LLM from the GPT-2 model family, and measure the rank of true token at each position when sorted by the model's predicted logit score, then average the ranks over normalized position across the StackOverflow dataset.

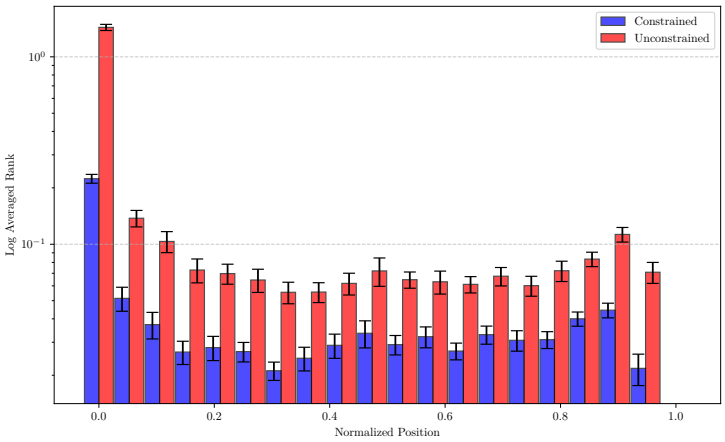


Figure 5.5: Mean rank of constrained versus unconstrained next-token prediction across normalized snippet positions (lower is better).

What this tells us is that the model is most likely to make an error at the beginning and end of the snippet, and the rate of decoding errors is significantly higher in the unconstrained case. Constrained decoding removes a large fraction of "junk" tokens that the model is likely to generate, but are syntactically invalid or inconsistent with the predicted edit distance, thereby improving the true token's rank and rate of matching the true repair.

6

Discussion

Our work shows a surprising connection between advanced structured prediction and formal language learning. Large language models are very sample efficient, but are expensive to train. Verifying the correctness of a large language model is a hard problem and open research question. However, we show that if the specification can be expressed as a context-free grammar or finite intersection thereof, one can easily force the model to produce only valid text. Furthermore, if one is careful in their modeling assumptions, they can ensure every valid sentence has a nonzero probability of being generated.

Not only from a safety perspective, pairing constraints with a weak autoregressive model such as a low-order Markov chain can be competitive with SoTA neural language models on certain kinds of sequence modeling tasks. Most of the LLM is dedicated to [poorly] relearning syntax, and if we can remove the burden of modeling the syntax, we can use constrained decoding and a weak model to obtain higher precision at a fraction of the cost.

This manifests as a practical tool for repairing syntax in an IDE, as well as an interesting case study on language modeling. We can treat the grammar as an incremental verifier. If you have access to such a verifier (which allows you to preemptively reject continuations of partial trajectories before evaluating a full rollout) and massively parallelize the sampler, then you can often saturate the entire sample space or finite slices thereof. Together with a cheap ranking function, this method is highly competitive with large, expensive LLMs.

7

Conclusions

Our work, while a case study on syntax repair, is part of a broader line of inquiry in program synthesis that investigates how to weave formal language theory and machine learning into helpful programming tools for everyday developers. In some ways, syntax repair serves as a test bench for integrating learning and language theory, as it lacks the intricacies of type-checking and semantic analysis, but is still rich enough to be an interesting challenge. By starting with syntax repair, we hope to lay the foundation for more organic hybrid approaches to program synthesis.

Two high-level codesign patterns have emerged to combine the naturalness of neural language models with the precision of formal methods. One seeks to filter the outputs of a generative language model to satisfy a formal specification, typically by some form of rejection sampling. Alternatively, some attempt to use language models to steer an incremental search for valid programs via a reinforcement learning or hybrid neurosymbolic approach. However, implementing these strategies is often painstaking and their generalization behavior can be difficult to analyze.

In our work, we take a more pragmatic tack - by incorporating the distance metric into a formal language, we attempt to exhaustively enumerate repairs by increasing distance, then use the stochastic language model to sort the resulting solutions by naturalness. The more constraints we can incorporate into formal language, the more efficient sampling becomes, and the more precise control we have over the output. This reduces the need for training a large, expensive language model to relearn syntax, and allows us to leverage compute for more efficient search and ranking.

There is a delicate balance in formal methods between soundness and

completeness. Often these two seem at odds because the target language is too expressive to achieve them both simultaneously. In syntax repair, we also care about *naturalness*. Fortunately, syntax repair is tractable enough to achieve all three by modeling the problem using language intersection. Completeness helps us to avoid missing simple repairs that might be easily overlooked, soundness guarantees all repairs will be valid, and naturalness ensures the most likely repairs receive the highest priority.

From a usability standpoint, syntax repair tools should be as user-friendly and widely accessible as autocorrection tools in word processors. We argue it is possible to reduce disruption from manual syntax repair and improve the efficiency of working programmers by driving down the latency needed to synthesize an acceptable repair. In contrast with program synthesizers that require intermediate editor states to be well-formed, our synthesizer does not impose any constraints on the code itself being written and is possible to use in an interactive programming setting.

The design of the tool itself is relatively simple. Tidyparse accepts a context-free language and a string. If the string is valid, it returns the parse forest, otherwise, it returns a set of repairs, ordered by likelihood. This approach has many advantages, enabling us to repair broken syntax, correct typos and recover from small errors, while being provably sound and complete with respect to the grammatical specification and a Levenshtein bound. It is also compatible with neural program synthesis and repair techniques, which can be used to score and rank the generated repairs.

We have implemented our approach and demonstrated its viability as a tool for syntax assistance in real-world programming languages. Tidyparse is capable of generating repairs for invalid source code in a range of practical languages with little to no data required. We plan to continue expanding the prototype's autocorrection functionality to cover a broader range of languages and hope to conduct a more thorough user study to validate its effectiveness in practical programming scenarios.

7.1 Future work

We identify three broad categories of limitations in evaluating Tidyparse and suggest directions for future work: naturalness, complexity, and semantics.

7.1.1 Naturalness

Firstly, Tidyparse does not currently support intersections between weighted CFGs and weighted finite automata, a la Pasti et al. [POP+23]. This feature would allow us to put transition probabilities on the Levenshtein automaton corresponding to edit likelihood then construct a weighted intersection grammar. With this, one could preemptively discard unlikely productions from $G_\cap$ to reduce the complexity in exchange for relaxed completeness. We also hope to explore more incremental sampling strategies such as SMC [LZXGM23].

The scoring function is currently computed over lexical tokens. We expect that a more precise scoring function could be constructed by splicing candidate repairs back into the original source code and then scoring plaintext, however this would require special handling for insertions and substitutions of names, numbers and identifiers that were absent from the original source code. For this reason, we currently perform the scoring in lexical space, which discards a useful signal, but even this coarse approximation is sufficient to achieve state-of-the-art precision.

Furthermore, the scoring function only considers each candidate repair $P_\theta(\sigma')$ in isolation, returning the most plausible candidate independent of the original error. One way to improve this would be to incorporate the broken sequence ($\underset{\sim}{\sigma}$), parser error message (m), original source (s), and possibly other contextual priors to inform the scoring function. This would require a more expressive probabilistic language model to faithfully model the joint distribution $P_\theta(\sigma' \mid \underset{\sim}{\sigma}, m, s, \ldots)$, but would significantly improve the precision of the generated repairs.

7.1.2 Complexity

Latency can vary depending on several factors including string length, grammar size, and critically the Levenshtein edit distance. This can be an advantage because, without any contextual or statistical information, syntax and minimal Levenshtein edits are often sufficiently constrained to identify a small number of valid repairs. It is also a limitation because the admissible set expands rapidly with edit distance and the Levenshtein metric diminishes in usefulness without a very precise metric to discriminate natural solutions

in the cosmos of equidistant repairs.

Space complexity increases sharply with edit distance and to a lesser extent with length. This can be partly alleviated with more precise criteria to avoid creating superfluous productions, but the memory overhead is still considerable. Memory pressure can be attributed to engineering factors such as the grammar encoding, but is also an inherent challenge of language intersection. Therefore, managing the size of the intersection grammar by preprocessing the syntax and automaton, then eliminating unnecessary synthetic productions is a critical factor in scaling up our technique.

7.1.3 Toolchain integration

Lastly and perhaps most significantly, Tidyparse does not incorporate semantic constraints, so its repairs whilst syntactically admissible, are not guaranteed to be type safe. It may be possible to add a type-based semantic refinement to our language intersection, however this would require a more expressive grammatical formalism than CFGs naturally provide.

Program slicing is an important preprocessing consideration that has so far gone unmentioned. The current implementation expects pre-sliced code fragments, however in a more practical scenario, it would be necessary to leverage editor information to identify the boundaries of the repairable fragment. This could be achieved by analyzing historical editor states or via ad hoc slicing techniques.

Additionally, the generated repairs must be spliced back into the surrounding context, which requires careful editor integration. One approach would be to filter all repairs through an incremental compiler or linter, however, the latency necessary to check every repair may be non-negligible.

We envision a few primary use cases for Tidyparse: (1) helping novice programmers become more quickly familiar with a new programming language, (2) autocorrecting common typos among proficient but forgetful programmers, (3) as a prototyping tool for PL designers and educators, and (4) as a pluggable library or service for parser-generators and language servers.

Publications

This part is optional, but it gives a nice touch to list all the publications (official venues down to poster sessions) throughout your PhD.

Keep this in the same style as publications in your academic CV: Conference / Year - Title - Authors. And here comes a sample ref for the bibliography: [Con23]

- Under Review (2024) – Syntax Repair as Language Intersection
- Midwest PL Summit (2024) – Let's wrap this up! Incremental structured decoding with resource constraints
- POPL, LAFI (né PPS) (2024) – A Tree Sampler for Bounded Context-Free Languages
- Doctoral Symposium at SPLASH (2023) – A Pragmatic Approach to Syntax Repair
- TEACH Workshop at ICML (2023) – Idiolect: A Reconfigurable Voice Coding Assistant
- BotSE Workshop at ICSE (2023) – Idiolect: A Reconfigurable Voice Coding Assistant
- LIVE Workshop at SPLASH (2022) – Tidyparse: Real-Time Context Free Error Correction
- ARRAY Workshop at PLDI (2022) – Probabilistic Array Programming on Galois Fields

Acronyms

Below are a list of acronyms used in the construction of this thesis:

- **CFG**: **C**ontext **F**ree **G**rammar
- **CFL**: **C**ontext **F**ree **L**anguage
- **CNF**: **C**homsky **N**ormal **F**orm
- **DFA**: **D**eterministic **F**inite **A**utomaton
- **NFA**: **N**ondeterministic **F**inite **A**utomaton
- **GRE**: **G**eneralized **R**egular **E**xpression
- **OGF**: **O**rdinary **G**enerating **F**unction
- **LBH**: **L**evenshtein **B**ar-**H**illel [construction]
- **IID**: **I**ndependent and **I**dentically **D**istributed
- **INE**: **I**ntersection **N**on-**E**mptiness
- **PPM**: **P**arameterized **P**arikh **M**ap

Bibliography

[AP72] Alfred V Aho and Thomas G Peterson. A minimum distance error-correcting parser for context-free languages. SIAM Journal on Computing, 1(4):305–312, 1972.

[BCG+10] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 339–352, 2010.

[BHPS61] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. Sprachtypologie und Universalienforschung, 14:143–172, 1961.

[Brz64] Janusz A Brzozowski. Derivatives of regular expressions. Journal of the ACM (JACM), 11(4):481–494, 1964.

[BYQ+23] Christopher Bryant, Zheng Yuan, Muhammad Reza Qorib, Hannan Cao, Hwee Tou Ng, and Ted Briscoe. Grammatical error correction: A survey of the state of the art. Computational Linguistics, 49(3):643–701, 2023.

[Con23] Breandan Considine. A pragmatic approach to syntax repair. In Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, pages 19–21, 2023.

[CS59] Noam Chomsky and Marcel P Schützenberger. The algebraic theory of context-free languages. In Studies in Logic and the Foundations of Mathematics, volume 26, pages 118–161. Elsevier, 1959.

[CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In Proceedings of the 33rd International Conference on Software Engineering, pages 121–130, 2011.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. Communications of the ACM, 13(2):94–102, 1970.

[Fla09] P Flajolet. Analytic Combinatorics. Cambridge University Press, 2009.

[Flo67] Robert W Floyd. Nondeterministic algorithms. Journal of the ACM (JACM), 14(4):636–644, 1967.

[FU15] Denis Firsov and Tarmo Uustalu. Certified normalization of context-free grammars. In Proceedings of the 2015 Conference on Certified Programs and Proofs, pages 167–174, 2015.

[KCG90] Mark D Kernighan, Kenneth Church, and William A Gale. A spelling correction program based on a noisy channel model. In COLING 1990 Volume 2: Papers presented to the 13th International Conference on Computational Linguistics, 1990.

[Koz77] Dexter Kozen. Lower bounds for natural proof systems. In 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pages 254–266. IEEE, 1977.

[LGPRC21] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. IEEE Software, 38(4):22–27, 2021.

[LR16] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, pages 298–312, 2016.

[LZXGM23] Alexander K Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K Mansinghka. Sequential monte carlo steering of large language models using probabilistic programs. arXiv preprint arXiv:2306.03081, 2023.

[MDS11] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: a functional pearl. ACM sigplan notices, 46(9):189–195, 2011.

[MN04] Scott McPeak and George C Necula. Elkhound: A fast, practical GLR parser generator. In International Conference on Compiler Construction, pages 73–88. Springer, 2004.

[Mon18] Martin Monperrus. The living review on automated program repair. PhD thesis, HAL Archives Ouvertes, 2018.

[Par66] Rohit J. Parikh. On context-free languages. J. ACM, 13(4):570–581, oct 1966.

[POP+23] Clemente Pasti, Andreas Opedal, Tiago Pimentel, Tim Vieira, Jason Eisner, and Ryan Cotterell. On the intersection of context-free and regular languages. In Andreas Vlachos and Isabelle Augenstein, editors, Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pages 737–749, Dubrovnik, Croatia, May 2023. Association for Computational Linguistics.

[SEG+22] Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, and Ranjit Jhala. Seq2parse: neurosymbolic parse error repair. Proceedings of the ACM on Programming Languages, 6(OOPSLA2):1180–1206, 2022.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, pages 15–26, 2013.

[SM02] Klaus U Schulz and Stoyan Mihov. Fast string correction with levenshtein automata. International Journal on Document Analysis and Recognition, 5:67–85, 2002.

[SSL19] Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: component-based synthesis with control structures. Proceedings of the ACM on Programming Languages, 3(POPL):1–29, 2019.

[YL21] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In International Conference on Machine Learning, pages 11941–11952. PMLR, 2021.