An Edit Calculus for Probabilistic Program Repair

Breandan Considine

School of Computer Science
McGill University
Montreal, Quebec, Canada

November 4, 2024

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.
- **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 the exchange of information between programmers 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 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 probabilistic model of language 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.

2

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 lets us represent 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 the 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 of interest, but provide a fast procedure for rejecting invalid programs and parsing valid ones.

Like all representations, grammars are a trade-off between expressiveness and efficiency. It is often 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 deterministic finite automaton containing millions of states, or a simple conjunctive grammar containing a few productions.

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

Like sets, it is possible to abstractly combine languages by manipulating their grammars, mirroring the setwise operations of union, intersection, 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 both necessary for a program to be considered valid. We can treat valid programs P as a subset of the language intersection $P \subseteq \mathcal{L}(G_a) \cap \mathcal{L}(G_b)$.

3

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 number of edits 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. As it turns out, there is an automaton, called the Levenshtein automaton [**?**], that recognizes all strings within a certain Levenshtein distance of a given string. We can use this automaton to find the nearest most likely repair consistent with the observed program and the grammar.

Given the source code for a computer program $\hat{\sigma}$ and a grammar G, our goal is to find the most likely valid string σ consistent with the grammar G and the observed program $\hat{\sigma}$. We can formalize all possible repairs as a language intersection problem.

4

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 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 the previous tokens. This allows us to compute the likelihood of a string 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.

5

Discussion

Official McGill Guidelines: The discussion of findings must be in line with disciplinary expectations. A comprehensive discussion is expected to be a minimum of 10 pages, double-spaced for doctoral students and a minimum of 5 pages, double-spaced for Master's students (including figures, images, and tables). It pertains to the entirety of a thesis. The discussion of findings should provide an final, overarching summary of study themes, limitations, and future directions.

In the case of a manuscript-based thesis, the comprehensive discussion should encompass all of the chapters of the thesis and should not be a repetition of the individual chapters. This section can be used to address issues not sufficiently covered in the preceding chapters or papers (e.g., critiques raised by reviewers that could not be incorporated into published works, or reintroducing discussion arguments removed from published papers upon reviewer request). This section can also be used to elaborate on the practical/applied aspects of published findings in a manner that is more accessible to less expert readers.

6

Conclusions and Future Work

Official McGill Guidelines: Clearly state how the objectives of the research were met and discuss implications of findings.

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]

Acronyms

This part is likewise optional. But it does not hurt to provide a list of all acronyms, e.g.:

- **REST**: **Re**presentational **S**tate **T**ransfer

Bibliography

[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.