#12 Tidyparse: Real-Time Context Free Error Correction

Email notification
Select to receive email on updates to reviews and comments.

PC conflicts
None

Accepted

Abstract

Tidyparse is a program synthesizer that performs real-time error correction for context free languages. Given both an arbitrary context free grammar (CFG) and an invalid string, the tool lazily generates admissible repairs while the author is typing, ranked by Levenshtein edit distance. Repairs are guaranteed to be complete, grammatically consistent and minimal. Tidyparse is the first system of its kind offering these guarantees in a real-time editor. To accelerate code completion, we design and implement a novel incremental parser-synthesizer that transforms CFGs onto a dynamical system over finite field arithmetic, enabling us to suggest syntax repairs in-between keystrokes. We have released an IDE plugin demonstrating the system described.

Authors

Breandan Considine (McGill University)
<BREANDAN.CONSIDINE@GMAIL.COM>
Jin L.C. Guo (McGill University) <jguo@cs.mcgill.ca>
Xujie Si (McGill University) <xsi@cs.mcgill.ca>

The Submission

acmart.pdf (461kB)

	OveMer	RevExp
Review #12A	2	2
Review #12B	4	3

You are an **author** of this submission.

Edit submission Add comment

Reviews in plain text

Review #12A

Overall merit

2. Weak reject

Reviewer expertise

2. Some familiarity

Paper summary

The paper presents a program synthesizer that given a string with parsing errors or holes provides suggestions for possible correct strings with short Levenstein distance to the provided string. The presented approach is motivated by challenges in manual repair of syntax errors, specifically for novice programmers, and has been realized in an IDE plugin. The utility of the approach is demonstrated with examples and evaluated with regard to performance.

Comments for authors

After reading the paper I have concerns regarding the positioning of the work and the evaluation of the approach. I also think the presentation needs a bit more work. Please see my comments below.

Section 1:

- The positioning of the work can be improved. The main problem being mentioned in the paper is that of novice programmers having problems with manual repair of syntax errors. If there is a user study available to back up this claim then that would be good to include here, or mention that it is your observation.
- With no related work section I would expect more references to related work on error recovery in the introduction. For instance, work like the following:
  - de Jonge & Visser (2012). “Automated evaluation of syntax error recovery”, ASE’12.
  - Kats et al. (2009). “Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing” ACM Sigplan Notices 44(10):445-464.

Section 2:

- What is the purpose of having support for holes in the string? Is this part of the intended interaction with the system or is it a feature convenient for evaluation? Does the number of underscores in the hole have any meaning?

Section 3:

- “... nonterminals, V, ...” -> “... noterminals (V), ...” ? and the same for the other
- “§6” -> “Section 6” here and elsewhere in the paper. At least this is what I could expect for section references.

Section 4:

- “Unlike classical parsers which need special care to recover from errors …” Please include more details and references.

Section 6:

- What does ‘hollowing’ mean here?

Section 7:

- This section comes across as a bit odd to me with first a larger example compared to earlier, something about grammar assistance and then support for expansion. It may be that this section just needs a bit more text to explain that the section is about features in the IDE plugin. The grammar example doesn’t fully fit into that picture. I assume it’s inclusion is about showing support for complexity but it’s included without much commentary in the text in a way that comes across as a bit odd to me.

- With one string rendering 7 suggestions for the expansion feature, it seems like it would soon become difficult to manage for larger programs. I would expect some discussion of this.

Section 8:

- The evaluation is a little baffling to me in that the main motivation for the work appears to be better support for novice programmers, but the evaluation is focusing on performance. Performance is definitely a factor in the usability of the system, but it’s not lifted as a main focus of the work in the introduction.

- The evaluation is focusing on latency and using strings of length 50 with a selection of grammars. These strings come across as fairly small. Why are they appropriate for this evaluation?

- The difference between the first and the second experiment is not entirely clear to me. Why is it interesting to consider known locations and then unknown locations?

Section 9:

- “We observe that errors are typically concentrated nearby historical edit locations …” Is this from your own experience or from observing students? What was the context and language used for these observations?

- Why is the lack of a neural language model a shortcoming? Isn’t this rather a new feature to add to the approach, something for future work, rather than something that should’ve already happened.

“Tidyparse is typically competitive with classical parsing methods.” This claim requires more details and supporting evidence.

- “… in our experience it is much more interpretable than classical parsers …” This line of reasoning becomes much stronger if the experience is not that of the tool developers, as

you may have built a tool tuned to your needs. It's unclear to what extent you are representative. It would also help to be more specific here.

Review #12B

Overall merit

4. Accept

Reviewer expertise

3. Knowledgeable

Paper summary

This paper presents a technique and implemented tool, Tidyparse, for enumerating repairs of invalid strings with respect to arbitrary context-free grammars. The repairs are guaranteed to be complete and are generated lazily in real-time as the programmer types.

The technique is a novel error-correcting extension of Valiant's encoding of CFG parsing as matrix multiplication to a fixpoint. The idea is to enumerate different possible replacements of tokens in the input string as holes; in the corresponding matrix construction, these holes become variables whose solutions can be generated by a SAT solver and which correspond to different modifications at these holes.

The authors give a basic example of usage and present some performance benchmarks of Tidyparse's repair generation time on various Dyck languages (langs consisting of various matching braces/delimiters).

Comments for authors

Interesting work! It seems like there are a number of distinct workflows enabled by this technique: manual insertion of holes to fill, generating error repairs, stepwise non-terminal expansion, substring parsing, etc.

In the presentation (and future work), I'd be interested to see the authors elaborate on these workflows. Could the authors illustrate a scene for me of a concrete user and task and how these workflows might fit into completing that task? Are there any interesting workflows that might arise by fine-grained interleaving of these workflows?

I'm glad there is a performance evaluation, and I think the use of Dyck languages is a reasonable synthetic benchmark as a first step. I'd be interested to see how this approach fares on larger grammars. Have you seen Don't Panic! Better, Fewer, Syntax Errors for LR Parsers? They provide an error-correcting parsing approach as well, though restricted LR grammars, and evaluate their approach on a large corpus of real-world Java programs with syntax errors. If viable, I'd be interested to see how your approach fares on the same benchmarks.

Questions and comments:

- How does the given matrix-based approach compare to Aho and Peterson's original proposal?
- I'm confused by the hole-filling example in Section 2. There are three holes in the first stretch holes, which seem to be filled by either 2 or 1 tokens. Is this because the holes can be filled by ϵ? On the one hand, I find that confusing as a user interface. On the other hand... I kind of like how that enables some flexibility in your hole configurations -- no need to come up with the exact number of holes to be filled, just an upper bound. Then again, I'm not sure when I or a novice would engage in requesting token-level hole fillings that require specifying the token locations.
- In the commutative diagram in Section 3, I think the Vs should be $P(V)$s?
- I would consider moving Section 3.1 to Section 6. Seems independent of the surrounding details and feels out of place in current spot.
- I didn't understand Section 3.2. I'm not sure what the example matrix is representing. I'm not sure what "established nonterminal forests" vs "seeded nonterminal forests to be grown" are.
- What is Γ in Section 3.3?
- I was confused by the "Krummholz" reference in Section 5. I think you're probably referring to the way the normalized tree leans to the right. I would say so explicitly. The italics also make it seem like you're introducing a known or new technical term, which added to my confusion.
- Section 6, Step 2: what is "hollowing"?

HotCRP