Initiation to Research
Compiler correctness

Neven Villani

2022-01-11

Based on "TRX: A Formally Verified Parser Interpreter"
by Amam Koprowski and Henri Binsztok at MLstate

Original work available at
https://arxiv.org/pdf/1105.2576.pdf
https://link.springer.com/content/pdf/10.1007/978-3-642-11957-6_19.pdf
(some minor differences between the two, the review should be considered to refer to the first one)

This report can also be read at
https://perso.crans.org/vanille/share/compverif/report.pdf
https://github.com/Vanille-N/compverif
and the related beamer:
https://perso.crans.org/vanille/share/compverif/beamer.pdf

1 Summary

1.1 Context and motivation

Despite parsing being the first step in compilation, more interest has been placed on verifying the correctness of optimisations concerning manipulations of ASTs rather than the initial generation of an AST from the source code.

Given that the most widespread parser generator (Yacc) has known bugs in many of its implementations, any verified compiler should include a verified parser alongside. CakeML does so, by formally verifying the implementation of their parser. However CakeML's approach is not generic enough to be easily applicable to parsers of other grammars.

Much like Yacc is a parser generator intended to prevent needless efforts in producing a new parser from the ground up, there is a need for a verified parser generator that could reliably produce verified parsers from a description of their grammars, in a way that requires less work and maintenance than implementing a complete parser.

1.2 Contribution

TRX is one such verified parser generator. It accepts a generic description of a grammar and produces a parser which reads text to construct an AST. Said parser is proven correct and complete with respect to the input grammar.

TRX not only rules out incorrect implementations of parsing, it also detects inappropriate grammars which contain incorrectly typed productions and rejects any grammar that is susceptible not to terminate.

1.3 Choice of tools

The parser interpreter is implemented using Coq's program extraction facilities. It uses parsing expression grammars (PEG) as grammar descriptors, which are little more than a declarative specification of recursive descent parsers.
Compared to context-free grammars (CFG), PEG are closer to regular expressions and allow to forego the lexing step. This makes them a better candidate for verification because the structure of the parser is more closely

related to that of the grammar, there is only a single tool to verify instead of two, and there is no ambiguity. As an added convenience, some grammars that are not context-free can nonetheless be expressed by PEG.

1.4 Results

Little effort needed to be devoted towards proof of correctness, instead more focus was placed on proof of termination. This is mostly attributed to the choice of tools and the nature of PEG: being close in structure to the recursive descent parser it is easy to prove equivalence between the two and since the grammar is specified within Coq the translation can be done in few steps. On the other hand PEG are inherently prone to nontermination and separate properties of the grammar have to be established and proved sufficient.

TRX's performance is compared to other non-verified implementations with comparable functionality, and excluding some suboptimal encodings of standard datatypes, it can be said that TRX's runtime speed is in the same order of magnitude as parsers that lack formal verification.

1.5 Limitations

PEG themselves – although expressive – carry a high performance cost for certain grammars which require a lot of backtracking. Memoization can solve the exponential time cost at the price of an increased memory footprint, but is not yet implemented in TRX.

Other limitations more closely related to TRX itself are on usability: the grammar being specified within Coq, it requires a certain degree of familiarity with Coq in order to produce a parser. In this aspect the basic design of TRX is the root of the problem: it is impossible to make TRX a standalone and eliminate the interaction with Coq without switching to code generation instead of code extraction. This would require substantially more efforts and would likely make the proof of correctness much harder since the semantics of the target language would need to be considered.

2 Review

2.1 Introduction

The interest of parsing as a tool for the Web is somewhat of a surprising and restrictive first motivation, but it is understandable given the prior work of the authors on a Web framework.

The context is well laid out, and includes references to prior work both theoretical and practical. In addition to the theoretical interest in verified parsing, the concern for the practicality of the resulting tool is apparent all throughout the article which is expected given that the resulting tool is to be used as-is in production.

The introduction does a good job justifying in a single paragraph the interest in choosing PEG rather than CFG. An unspoken but likely motivation, inferred from the performance benchmarks which reference a production-quality code-generating parser interpreter already used in the organization, is that the tool that TRX was supposed to replace was itself a PEG-based parser generator. Having a readily-available parser to compare TRX with without a doubt made the implementation of TRX easier.

2.2 Interest

The theoretical contribution is lacking: known earlier work includes parsers that feature more in-depth static analysis of the grammar and several aspects are suggested to have been chosen mostly to not pose too much of a challenge. I personally interprete the article more as a report of what has been figured out as a byproduct of the development of a tool rather than purposeful research to establish new techniques.

The main difference between this parser and previous work is claimed to be usability: this approach of emphasizing pragmatic design choices rather than unexplored techniques does have an advantage that the resulting tool is more likely to be usable in practice, which is after all one of the assumed primary goals of this paper. Nevertheless later developments show that "more usable in practice" does not extend to user unfamiliar with Coq.

2.3 Core

There are few – three to be exact – theorem statements, but they are accompanied by well-written proof sketches. The only of the three that is not a straightforward induction is well-explained. It is not very clear whether the criteria for termination are new or were already known. Clarifying to what extent the study of completeness of grammars is a new contribution would have been welcome. One can to an extent infer from the section on related works that in fact these analysis techniques were already known.

Concerning the more practical study of the software, the benchmarks are well-analyzed and seem appropriate. The technical and theoretical limitations with regards to implementing a language to specify grammars are well-explained.

The benchmarks feature plenty of comparisons to already existing tools that implement similar functionality without the added formal verification, but there are also comparisons to parsing libraries developed in Agda or HOL or Coq with particular focus on the improved usability for the first, augmented expressivity as well as safety for the second, and termination for the third.

2.4 Accessibility

The level of detail is appropriate and most of the article is easy to understand. The principles and advantages of PEG are well explained and the excerpt from the body of the main recursive parsing routine is clear, though it could use a few comments in the code.

One point that is not developed in great detail is on page 11 concerning the use of an over-approximation for $\mathbb{P}_0$ by assuming by default that for all e, $e \in \mathbb{P}_{>0}$ and $e \in \mathbb{P}_{\perp}$. It is specified that said over-approximation is known to reject even simple valid grammars, but none of which are known not to be useless (e.g. grammars that always reject). The approximation is thus considered acceptable since it is not known to wrongly reject any grammar that would be used in practice. It is unclear if and how this fact has been actually used in the development : does TRX use the over-approximation ? would doing so simplify the proofs ? If not then this may just be used to simplify the intuitive explanation of the reasoning in the paper, which is also fine but could have been made more explicit.

3 Later progress

3.1 TRX as a standalone

I am a bit disappointed by the difficulty of finding TRX online. At a first glance the only resource available other than the article seems to be a patent claim. The difficulty of finding the source code may be made more difficult by the existence of an unrelated `.trx` file format which collides with both the name and the file extension of this project.

Nevertheless traces of TRX can be found in `github:MLstate/opalang` (the GitHub repository of the parent project Opa, which is a concurrency- and safety-focused programming language for secure web applications) under the name "teerex": in `tools/teerex/certfied/trx2cert.ml` one will find a tool to convert from grammars in the `.trx` format to their specification in Coq. An example of a such TRX grammar is found at `libtrx/trxparse.trx` and proves to be very similar to what one would expect from more standard PEG parser generator grammar specifications. Other files found in neighboring directories include runtime libraries for extracted code, and high-level wrappers.

Thus although I was unfortunately unable to find the complete proofs of TRX (and thus have to take the authors' words for granted concerning the lengths and difficulties of the proofs), it seems that at least the issue of making TRX a standalone has been mostly solved: TRX still needs Coq to be installed to be able to function, but since Coq is invoqued transparently and `.v` files are generated on the fly the user does not need to directly interact with Coq by writing actual code.

3.2 Memoization

The main performance limitation of PEG evoqued was the time/memory tradeoff in implementing memoization to avoid exponential-time parsing in some worst cases. A paper by C. Blaudeau and N. Shankar in this direction seems to be at the stage of a preprint: A Verified Packrat Parser Interpreter for PEG ("packrat" being the name of the memoization technique for PEG parser generators)

Another example is Certified Web Services in Ynot by R. Wisnesky, G. Malecha and G. Morrisett in which (although not the main focus of the article) a verified packrat PEG parser is implemented.

TRX itself seems to have never reached that stage, since the aforementioned GitHub repository contains no references to "packrat", "memoization", "exponential", and no relevant reference to "performance". Its parent project Opa seems to be at a standstill: after a peak of activity between 2012 and 2013 there have been few commits after 2014 and none after 2016. Opa being to the best of my knowledge the only language to use TRX in production, it is unsurprising that the development of TRX would have stopped. Nevertheless it shows that TRX was good enough for use in an actual language and that the exponential-time worst-case behavior was never deemed a notable issue.

3.3 Other parser architectures

Following TRX, verified parsers generators have been implemented for other grammar classes.

In 2012, J.-H. Jourdan, X. Leroy and F. Pottier published Validating LR(1) parsers. In contrast to TRX, they present not a translator from the grammar specification to a parser, but a verifier that checks *a posteriori* that the parser and grammar agree. This approach is in a sense more generic because it is applicable to parsers and parser generators that have already been written beforehand independently of how they were compiled. At the same time it requires more work from the user since the parser generator must be chosen separately and integrated with the verifier.
They mention having thus validated a parser generated by Menhir and replaced CompCert's unverified parser with this new one. The performance cost seems to be comparable to that of TRX, and similar reasons are invoked: redundant type conversions and runtime checks, and the input being parsed twice (once by the parser, once by the validator).

In 2019, S. Lasser, C. Casinghino, K. Fisher and C. Roux looked at $LL(1)$ grammars in A Verified LL(1) Parser Generator. This time the source code was easy to find since the paper itself includes a reference to the GitHub repository.
This one as well follows the approach of extracting a parser from proofs, and has the same limitation of requiring that the grammar be specified within Coq since a specialized language to specify grammars is not available.
It is hard to relate the length of proofs with that of TRX since the architecture is completely different, but it can be said that this time the correctness proof was absolutely not easy, contrary to what was suggested in TRX: there are a few thousand lines of soundness proofs including lemmas.