Demo Paper: Coqlex, an approach to generate verified lexers

Wendlasida Ouedraogo, Danko Ilik, Lutz Strassburger

To cite this version:

Wendlasida Ouedraogo, Danko Ilik, Lutz Strassburger. Demo Paper: Coqlex, an approach to generate verified lexers. ML 2021-ACM SIGPLAN Workshop on ML, Aug 2021, Online event, United States. hal-03470713

HAL Id: hal-03470713
https://inria.hal.science/hal-03470713
Submitted on 8 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract—A compiler consists of a sequence of phases going from lexical analysis to code generation. Ideally, the formal verification of a compiler should include the formal verification of every component of the tool-chain. In order to contribute to the end-to-end verification of compilers, we implemented a verified lexer generator with usage similar to OCamllex. This software – Coqlex – reads a lexer specification and generates a lexer equipped with Coq proofs of its correctness.

Although the performance of the generated lexers does not measure up to the performance of a standard lexer generator such as OCamllex, the safety guarantees it comes with make it an interesting alternative to use when implementing totally verified compilers or other language processing tools.

Index Terms—Lexer, Tokenizer, RegExp, Coq, Formal verification, OCamllex, Compilers, lexical analyzers

I. INTRODUCTION

Lexing is one of the first components in compilers and other language processing tools. Combined with parsing, lexers whose goal is to perform lexical analysis, help to transform input texts into structured data, usually abstract syntax trees. The theoretical foundations of lexing are quite mature today and there exist performing and sophisticated tools, libraries and generators to help in implementing optimized lexers.

The issues this paper focuses on is the correctness of lexical analysis: how can we prove that lexer behaviour meets its specifications? How can we have a formal guarantee that a lexer generator generates a lexer that respects the input specification?

In this document, we describe an implementation of a lexical analyzer that satisfies that specification. As in OCamllex, a lexer specification is equivalent to a set of regular expression (regexp) with associated semantic actions. In the following points, we detail how we define and verify the components and concepts used to build Coqlex.

II. OUR APPROACH AND METHOD

Coqlex is formally verified software that reads a lexer specification (.vl file written with a particular syntax close to OCamllex .mll files) and generates a Coq file containing the implementation of a lexical analyzer that satisfies that specification. As in OCamllex, a lexer specification is equivalent to a set of regular expression (regexp) with associated semantic actions. In the following points, we detail how we define and verify the components and concepts used to build Coqlex.

A. Basic concepts and their verification

1) Verified regular expressions: Our work uses the Coq [Barras et al. 1997; Bertot 2008] implementation of Janusz Brzozowski’s algorithm [Miyamoto 2011; Brzozowski 1964]. This work provides:

- a definition of regex type: containing $\emptyset$, $\epsilon$ (empty string), booleans ($\lor$ (alternation), $\land$, $\lnot$), concatenation and literal char
- derivation and string matching functions
- necessary proofs, e.g. showing that the implementation satisfies the axioms of Kleene Algebra [Armstrong, Struth, and Weber 2013]

2) Election verification: The interpretation of the specification of a lexer in Coqlex is the same as in most lexer generators. When the generated lexer analyzes a string, the lexer has to select an action to execute. For this election, the interpretation follows these 2 rules:

- for every pair of a regexp with associated action, the lexer computes the longest prefix of the input string that the regexp (of the pair) can match. This score is compared to those of other pairs and the pair with the highest score wins: it is the longest-match rule.
- in case of equality, the winning pair is the one that comes first in the specification list: it is the priority rule.

We implemented and verified a function that computes correctly the score and the also an election function that follows those rules.

3) RecLexer type: In a nutshell, a lexer can be defined as a function that takes an input string with a start position and returns a token, its start position, the position after lexical analysis and the remaining string, when the process succeeds.

\[ \text{RecLexer type: } \text{In a nutshell, a lexer can be defined as a function that takes an input string with a start position and returns a token, its start position, the position after lexical analysis and the remaining string, when the process succeeds.} \]
According to that point of view, we defined a record type called RecLexer (see Listing 1) that is our general definition of what a correct lexer is. Its field called tokenizer is the functional definition of lexer and its other fields are lemmas related to the input and output positions and the input and output strings. Implementing a lexer-generator is equivalent to writing a RecLexer constructor.

From a set of regular expressions with associated semantic actions we can generate lexers that satisfy assumptions of Menhir [Pottier and Régis-Gianas 2015], a popular parser generator for OCaml, on lexers but also provides proofs related to the correctness of physical positions of tokens and of the text processing.

4) The action API: Lexers generated by OCamllex take a lexer buffer (a mutable record) and returns a token. This mutable record contains the last matched token start position, the current position and the remaining string to analyze. They update necessary fields of the input buffer during the election process and then execute the selected action (are arbitrary OCaml expressions) that can also update those fields.

Being a purely functional language, Coq does not allow side-effect. So, we had to define a data structure and interpretation rules for actions. In our system, an action is a sequence of the following authorized and atomic operation:

- modify of “logical” position: line number and column number.
- perform a recursive call.
- call of another RecLexer.
- return of a token.

This API is extensible: the user can implement the missing but correct (regarding our properties stated in RecLexer) features for semantic actions by implementing a particular RecLexer constructor that would perform the wanted action.

Notes: Regarding OCamllex action definition, we can define actions whose processing can lead to infinite loop during their interpretation (see Listing 2). Coq has guards that prevent implementing functions that can loop infinitely and requires termination proofs for non-trivial functions. For this reason, we had to define looping conditions in order to provide the termination proof of our interpreter. Listing 2 shows a simple example of a a of a lexer whose interpretation can lead to an infinite loop. The problem with this specification is that when analyzing the input string (“c”), the second action which is a recursive call is selected with a score equal to 0 (no character consumption). So performing the selected action is equivalent to make a recursive call on the lexer with the same input regarding the input string (due to the absence of consumption) leading to an infinite loop. In Coqlex, a similar code would return an error, noticing the detection of a looping case.

**B. Coqlex software**

Using our implementation of all the concepts mentioned in the previous subsection, the user can generate lexers by providing a Coq code of their specifications (regex-action list). This can be syntactically heavy and alter the user experience. To avoid that, we implemented a text-processor, Coqlex, that will convert a markup language close to OCamllex specification language into its equivalent Coq code.

It’s software architecture is detailed in Figure 1. According to this Figure, two thirds of Coqlex components are verified:

- its lexer has been implemented using RecLexer constructor (specifications are written in coq)
- its parser has been implemented using menhir with –coq
let lexbuf = Lexing.from_string "c" in
my_lexer lexbuf

Listing 2. Example of a lexer whose execution loops

Fig. 1. General structure of the Coqlex

switch to generate verified parsers
• its code writer or code generator is written in OCaml (not verified)

In practice, Coqlex generates a Coq file that implements the lexer and additional lines for extraction to OCaml. It also provides necessary OCaml files that help users to use Coqlex generated lexer like OCamllex generated ones.

Main differences: The structure of a Coqlex specification file is quite similar to the one of OCamllex. There exists some light differences related to our file generation and to Coq restrictions. For example, the user may have to declare or redefine the token type when Coq cannot guess it. In addition, due to our code generation and our data structure, mutual recursion is not allowed.

III. RESULTS

We evaluated the performance of our generator by implementing the lexer of 2 languages: JSON [Pezoa et al. 2016, Sikora 2017] and the first version of MiniML [Rinderknecht 2018], a toy subset of OCaml. It is easy to recognize the similarities between OCamllex syntax and Coqlex one (Listing 3 in Appendices shows the equivalent of Listing 2 in Coqlex). We compared the time performance and noticed a huge difference between OCamllex generated and Coqlex generated lexers.

The bottleneck of Coqlex comes from the election function. Most lexers are more complicated and it can be hard to find a simple and deterministic function that is equal to the election system. Even if the performance of Coqlex is not comparable to OCamllex, it lays the foundation for verified lexer generation, allowing to make one more step in proving end-to-end correctness of compilers.

IV. CONCLUSION

The formal correctness of lexing does not seem to be extensively studied in the literature. For instance, even for the formally proven compiler CompCert [Leroy et al. 2016], lexing is one of the phases which is not formally verified. In existing approaches to verify lexers, like in CakeML [Kumar et al. 2014], the lexer is implemented by hand (without using a generator) and proven equal to a simple and deterministic function. Most lexers are more complicated and it can be hard to find a simple and deterministic function that is equal to the lexer.

In future work, the performance of our generator could be enhanced in several ways, for instance by refining the election process by using finite automata, improving the action API to make it more intuitive and more complete, defining its parser to simplify the specification file, implementing a static analyzer to detect potential looping cases (it can be done by using a graph representation of finite automata), prove more properties, etc.

Even if the performance of Coqlex is not comparable to that of OCamllex, it lays the foundation for verified lexer generation, allowing to make one more step in proving end-to-end correctness of compilers.

ACKNOWLEDGEMENTS

We thank Gabriel Scherer for helpful comments improving the readability of this paper.
REFERENCES


APPENDIX

```coq
{ Require Import String. }

%set_history_type string.
%set_default_history_value EmptyString.

let one = 'b' @ 'a'*@ 'b'
let two = 'a'@

rule tok_lexer = parse
 | one { simplyReturnToken 1 } 
 | two { [SelfRec] ++ (simplyReturnToken 2) } 
 $ { simplyReturnToken 0 }
```

Listing 3. Equivalent of the looping lexer example in Coqlex