Demo Paper: Coqlex, an approach to generate verified lexers

Wendlasida Ouedraogo, Danko Ilik, Lutz Strassburger

Demo Paper: Coqlex, an approach to generate verified lexers

Wendlasida OUEDRAOGO
Siemens Mobility, Chatillon, France & *INRIA Saclay*, Palaiseau, France
lix.polytechnique.fr/Labo/Wendlasida_Tertius.OUEDRAOGO/

Danko ILIK
Siemens Mobility, Chatillon, France

Lutz STRASSBURGER
INRIA Saclay, Palaiseau, France, lix.polytechnique.fr/Labo/Lutz.Strassburger

***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[1] of a *verified lexer-generator* based on the analysis of regular expressions in Coq. Our lexer-generator implements the main features of *OCamllex* [Leroy 1997] and provides the proofs of certain correctness properties on our program and the generated lexer.

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 $\emptyset$, ϵ (empty string), booleans (or (alternation), and, not[2]), 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 wining 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.

[1] http://www.lix.polytechnique.fr/Labo/Wendlasida_Tertius.OUEDRAOGO/ML-2021/coqlex-v0.2.zip

[2] not and and booleans are not necessary for common use-cases of regular expression. As we are doing an experimental tool, we will keep them in our software

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 2016], 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 change 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 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.

```
(*Implementation of position data type*)
(*Inspired from OCamllex*)
Record Position : Set := mkPosition
{l_number : nat; (*line number*)
 c_number : nat; (*column number*)
 abs_pos : nat (*number of character already read*)}

Definition positionLE sp ep :=
( (abs_pos sp) <= (abs_pos ep)) /\
  ((l_number sp) < (l_number ep) \/
    ((l_number sp) = (l_number ep) /\
    (c_number sp) <= (c_number ep))).

Definition ignore_nfirst_char n str :=
substring n ((length str)- n) str.

Record RecLexer {Token Hist : Set} : Set := mkRecLexer
{
(*The lexical analyzer*)
tokenizer : (string -> Position ->
LexingResult (Token := Token));

(*Position related properties*)
lexer_start_cur_position : forall str_to_lex
param_pos r,
tokenizer str_to_lex param_pos = r
-> positionLE param_pos (extractCurrentPosition r)

;lexer_tok_position : forall str_to_lex
param_pos ts_pos remaining_str,
tokenizer str_to_lex param_pos = r ->
extractStartPosition r = Some ts_pos
-> positionLE ts_pos (extractCurrentPosition r)

;lexer_start_tok_position : forall str_to_lex
param_pos r, tokenizer str_to_lex param_pos = r ->
extractStartPosition r = Some ts_pos
-> positionLE param_pos ts_pos

;lexer_start_cur_position_abs : forall str_to_lex
param_pos r, lexer str_to_lex param_pos = r ->
abs_pos (extractCurrentPosition r) =
abs_pos param_pos + ((String.length str_to_lex) -
(String.length (extractRemainingString r)))

(*Characters consumption correctness*)
;lexer_correct_consum : forall str_to_lex param_pos r
, tokenizer str_to_lex param_pos history = r ->
extractRemainingString r = ignore_nfirst_char
((abs_pos (extractCurrentPosition r)) -
(abs_pos param_pos)) str_to_lex
}.
```

Listing 1. Definition of RecLexer

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

```
{
    open Lexing
}

rule my_lexer =
parse
  'b' 'a'* 'b'          { 0 }
  | 'a'*                { my_lexer lexbuf }
  | EOF                 { 1 }

{
    let lexbuf = Lexing.from_string "c" in
    my_lexer lexbuf
}
```

Listing 2. Example of a lexer whose execution loops

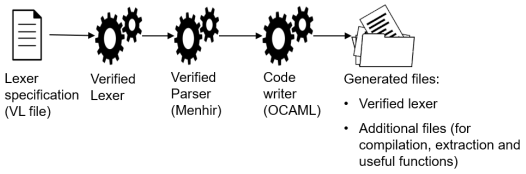

Fig. 1. General structure of the *Coqlex*

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 that we successively optimized in two versions and proved the correctness of those two optimizations. Figure 2 shows the execution time of OCamllex and the three versions of Coqlex including the successive optimizations, when run on different file sizes of JSON and MiniML source code.

We can observe that the performance of Coqlex is not comparable to OCamllex. We noticed that the most time consuming part were election and score computation that are naively implemented. To improve the situation we implented two optimizations (denoted as v0.1 and v0.2 in the Figure 2), and proved their correctness.

File size	Language/ #of tokens	OCamllex	Coqlex v0.0	Coqlex v0.1	Coqlex v0.2
1ko	JSON/98	6×10^{-5}	15	0.2	0.07
	MiniML/19	6×10^{-5}	10	0.16	0.06
4ko	JSON/386	5×10^{-4}	246.49	1.12	0.3
	MiniML/799	2×10^{-3}	224.27	0.83	0.54
16ko	JSON/1590	0.0013	≥ 600	9.200	6.4
	MiniML/3198	0.013	≥ 600	16.96	14.53
Mean Time / token		9.03×10^{-6}	–	0.049	0.0339

Fig. 2. Comparison of execution time in seconds for different input language/size

The proof provided by Coqlex can be summarized as following:

- The election system follows the priority and longest match rules.
- The start position of the found lexeme (the match string during election) is greater than the position at the beginning of the analysis.
- The position after lexical analysis is greater than the start position of the found lexeme.
- The distance between the physical start positions of a token and its end position is equal to the length of the first lexeme
- Coqlex cannot generate a lexer that loops

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

[ASW13] Alasdair Armstrong, Georg Struth, and Tjark Weber. "Kleene algebra". In: *Archive of Formal Proofs* 324 (2013).

[Bar+97] Bruno Barras et al. *The Coq Proof Assistant Reference Manual : Version 6.1*. Research Report RT-0203. Projet COQ. INRIA, May 1997, p. 214. URL: https://hal.inria.fr/inria-00069968.

[Ber08] Yves Bertot. "A short presentation of Coq". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, pp. 12–16.

[Brz64] Janusz A Brzozowski. "Derivatives of regular expressions". In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[Kum+14] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 179–191. ISSN: 0362-1340. DOI: 10.1145/2578855.2535841. URL: https://doi.org/10.1145/2578855.2535841.

[Ler+16] Xavier Leroy et al. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[Ler97] Xavier Leroy. *The OCaml system release 1.07*. 1997.

[Miy11] Takashi Miyamoto. *Coq Regular Expression Git Page*. https://github.com/coq-contribs/regexp. Accessed: 2020-10-14. 2011.

[Pez+16] Felipe Pezoa et al. "Foundations of JSON schema". In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pp. 263–273.

[PR16] François Pottier and Yann Régis-Gianas. "The Menhir parser generator". In: *See: http://gallium.inria.fr/fpottier/menhir* (2016).

[Rin18] Christian Rinderknecht. *A Mini-ML programming language*. https://github.com/rinderknecht/Mini-ML/blob/master/Lang0/Lexer.mll. 2018.

[Sik17] Aleksandra Sikora. *Tutorial: parsing JSON with OCaml*. Dec. 2017. URL: https://medium.com/@aleksandrasays/tutorial-parsing-json-with-ocaml-579cc054924f.

Appendix

```
{
    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