Regex matchers in HOL4 (Project)

Anoud Alshnakat

1 Regex matching

Regular expressions (regex) plays a substantial role in computer science. For example, in big data science it is troublesome to filter the data stored in a server, however it is relatively easy to store and retrieve it. Fischer et al. [2] presented multiple matching algorithms implementations in Haskell that are rather efficient compared to intuitive methods.

This project is based on the paper [2], where three of the matching algorithms were adopted to study and proved to be correct in HOL4 [1]. Moreover, EmitML-based SML code was later generated and tested with various words and regex combinations. The following sections include a brief explanation of each algorithm and the key theorems that were taken into account to prove their correctness.

1.1 Intuitive Regex Matching

In ACT I of the paper [2], the authors discussed an intuitive method to examine whether a regex conforms to a given pattern of arbitrary typed list. In this project [1] the intuitive matcher was indicated by the definition of `accept`.

Matching `Eps`, `Sym` and `Alt` are rather naive and straightforward compared to `Seq` and `Rep`. Thus, the algorithm required a further implementation of two other auxiliary functions, namely `split` and `parts`.

The `split` function takes a list as an argument and then produces all possible decompositions of a string into two factors. Each decomposition is stored as a tuple. It assists `accept` to match the regex `Seq p q` by trying to find at least one case of the decomposed tuples, such that it satisfies the sequential regex match. The first field of that tuple must match the `p` while the second must match `q`.

The `parts` function takes a list as an argument and then produces all possible decompositions of a string. It assists `accept` to match the regex `Rep r` by matching all the decomposed strings lists and check whether they satisfy the regex `Rep r`.

As suggested in the project instructions, a definition `language_of` had been created in order to check the correctness of the `accept` definition. It returns a set of the language accepted by the regex.

Proving correctness of `accept`. The goal of this part is clearly to prove the correctness of the intuitive matching definition `accept`. The primary backbone theorems and lemmas that helped in order to prove such a thing are the following:

Theorem (Thm_x)**.** *Flattening any member list p of the* `parts` *output results an identical list to the input. Also, p does not contain any empty list.*

$$\forall\, l\, p\,.\, MEM\, p\ (parts\ l) \Leftrightarrow (\neg(MEM\ [\,]\ p) \wedge (FLAT\, p = l))$$

Theorem (EQ_ACCEPT_LANG_IMP1). *If the word matches a regex, then then word is a member of the set of the language accepted by that regex.*

$$\forall r\, w\, .\, accept\, r\, w\ \Rightarrow\ w\ \in\ (language\_of\, r)$$

Theorem (EQ_ACCEPT_LANG_IMP2). *If the word is a member of the set of the language accepted by a regex, then the word matches that regex.*

$$\forall r\, w\, .\ w\ \in\ (language\_of\, r) \Rightarrow accept\, r\, w$$

Theorem (ACCEPT_LANG_EQ). *Given both previous theorems* $EQ\_ACCEPT\_LANG\_IMP1$ *and* $EQ\_ACCEPT\_LANG\_IMP2$ *then the logical equivalence between the accepted matching, and the membership of the word in the language of the regex is valid.*

$$\forall r\, w\, .\, accept\, r\, w\ \Leftrightarrow w\ \in\ (language\_of\, r)$$

1.2 Marked Regex Matching

The intuitive method is not efficient enough to match longer strings, due to the decompositions of `parts` can grow exponentially, i.e. a string of length n produces 2^{n-1} decompositions. Alternatively, a non-deterministic finite constructions (NDFA), Glushkov automaton, can be used to enhance the performance of the regex matching algorithms as suggested in ACT II. In order to check whether a string matches a given regex, then the letters of the string must be tracked by going through the regex from left to right. The tracking should determine the symbols that are possibly responsible for matching the character in the string. In this implementation, `MReg` datatype will be slightly changed. The symbol type will have an extra Boolean field indicates that the state of being marked or not, i.e. might be responsible for matching or not.

To implement such an algorithm, it is necessary to add multiple auxiliary definitions that aid the marked regex matching `acceptM`. The additional definitions are: `empty`, `final`, `shift` and `language_of_marked`.

The `empty` definition determines whether the regex can be empty. Thus, it helps `acceptM` to reason about the empty string case with a given regex.

The `final` definition determines whether the regex contains a final marked character. Thus, it helps the `acceptM` to reason about the marks at the end of the word in order to accept the string.

The `shift` definition pushes the marks from left to right through the regex. It takes a preceding mark `m`, a regex, and the current character being read from the string. Then it returns a marked expression.

A definition of the language after marking it had been created indicated by `language_of_marked`. This definition is important to reason about many aspects, for example, reasoning about the type `Reg` that got a mark field using `MARK_REG` and turned into a `MReg` type. Initially, the mark field of the `Sym F c` is false, which indicates that it is not marked yet, thus it does not have a language at all i.e. $\emptyset$.

Proving correctness of `acceptM`. The goal of this part is clearly to prove the correctness of the marked matching definition `acceptM`. The primary backbone theorems and lemmas that helped in order to prove such a thing are the following:

Theorem (acceptM_Empty_EQ). *The result of matching an empty word with a regex in* `acceptM` *is logically equivalent to the result of checking whether the empty word is a member of the language before marking it.*

$$\forall r\, .\, acceptM\ (MARK\_REG\, r)\, [\,]\ \Leftrightarrow\ [\,]\ \in\ (language\_of\, r)$$

Theorem (final_in_lang)**.** *The result of determining whether the regex has a final mark symbol* `final` *- in order to accept it - is logically equivalent to the result of checking whether the empty word is a member of the set of language accepted by the regex. Thus, both formulas can determine that the end of the reg being marked or not.*

$$\forall r\,.\,final\,r \Leftrightarrow [\,] \in (language\_of\_marked\ r)$$

Theorem (lang_in_shift)**.** *Whenever the mark being initialised (i.e. m = T) in a regex with respect to the character h, then the character h initially could have been in either* `language_of` *or* `language_of_marked` *before attempting to push the initial mark of it. Also, when the mark is being shifted (i.e. m = F) through the regex with respect to character h, then character h is in the original language before it has been shifted.*

$$\forall r\ m\ h\ t.\ t \in (language\_of\_marked\ (shift\ m\,r\,h)) \Leftrightarrow$$

$$h::t \in (language\_of\_marked\ (r)) \vee h :: t \in (language\_of\ (UNMARK\_REG(r))) \wedge m$$

Theorem (lang_in_fold)**.** *checking the resulted set of language after shifting a mark from left to right continuously for a subsequent h, is equivalent to checking if the whole word is initially accepted by the regex before shifting the marks through the regex.*

$$\forall r\ h\ t.\ t \in (language\_of\_marked\ (FOLDL\,(\,shift\,F)\,r\,h)) \Leftrightarrow h :: t \in (language\_of\_marked\ (r))$$

1.3 Cached Marked Regex Matching

To improve the performance of the marked expression, the result of the `empty` and `final` can be cached in an inner nodes of the regex. The datatype and the definitions of the previous part were altered accordingly. Additionally, a definition that denoted by the "smart constructor" in the paper was implemented. The primary backbone theorems and lemmas that helped in order to prove the correctness of the cached marked matching algorithm are the following:

Theorem (empty_CMempty)**.** *The result of marked empty definition is equivalent to the cached marked empty definition.*

$$\forall r\,.\,empty\,(r) \Leftrightarrow CMempty\,(CACHE\_REG\ r\,)$$

Theorem (final_CMfinal)**.** *The result of marked final definition is equivalent to the cached marked final definition.*

$$\forall r\,.\,final\,(r) \Leftrightarrow CMfinal\,(CACHE\_REG\ r\,)$$

Theorem (shift_CMshift)**.** *Caching the result of the marked shift definition is equivalent to the cached marked shift definition.*

$$\forall m\,r\,c\,.\,CACHE\_REG\,(shift\ m\,r\,c) \Leftrightarrow CMshift\ m\,(CACHE\_REG\ r\,)\,c$$

Theorem (acceptCM_acceptM)**.** *The definitions of the acceptCM and acceptM are equivalent.*

$$\forall r\,w.\ acceptCM\,(CACHE\_REG\,r)\,w \Leftrightarrow acceptM\,r\,w$$

2 EmitML and testing results

SML code was extracted using EmitML. The generated code was used to write an interface in order to test the three matching functions along with a fourth one, which is a built-in matcher found in `regexpMatch.sml` in EmitML library.

Table 1 shows the result of testing all combinations of each string with the regex

$$((a|b)*c\,(a|b)*c)*(a|b)*$$

proposed in the paper.

	accept			**acceptM**			**acceptCM**			**regexpMatch**		
Words	sys[s]	user[s]	real[s]	sys	user	real	sys	user	real	sys	user	real
"abcc"	0.000	0.001	0.003	0.001	0.001	0.001	0.000	0.001	0.001	0.000	0.005	0.005
"abcccc"	0.008	0.012	0.019	0.000	0.002	0.002	0.000	0.002	0.002	0.008	0.109	0.114
"aabcc"	0.000	0.001	0.002	0.000	0.001	0.001	0.000	0.001	0.001	0.000	0.020	0.019
"aabbcc"	0.000	0.016	0.013	0.000	0.002	0.002	0.001	0.001	0.002	0.004	0.136	0.125
"aabbbcc"	0.049	0.284	0.267	0.009	0.003	0.012	0.008	0.007	0.014	0.008	0.704	0.722
"aabbbbcc"	0.196	2.609	2.756	0.008	0.056	0.067	0.004	0.070	0.074	0.115	5.571	5.704
"aaabbbcc"	1.793	56.492	58.131	0.001	0.657	0.671	0.012	0.696	0.781	0.390	57.174	57.612
"aaaabbbb"	0.000	0.179	0.176	0.004	0.056	0.068	0.004	0.073	0.083	0.035	5.946	5.987

Table 1: system, user and real run-times -for all possible combinations of the words- versus four matching algorithms

References

[1] Anoud Alshnakat. Anoudalshnakat/regex-in-hol4. https://github.com/AnoudAlshnakat/Regex-in-HOL4, May 2020. (Accessed on 06/04/2020).

[2] Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions: functional pearl. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 357–368, 2010.