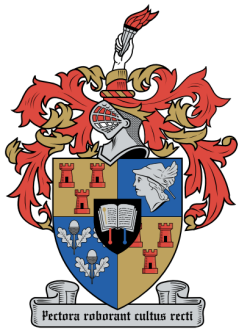

UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

Functional Programming Assignment 3

Marie-Louise Steenkamp 20722796
Hendrik van Heerden 20916892
Christoff Van Zyl 20072015

November 25, 2020

Contents

1 Introduction

The purpose of this paper is to work on a non-trivial project in Haskell, and to use all relevant available projects to build a new program. Here the focus was on deriving Definite Finite Automata (DFAs) from Regular Expressions (REs) using the notion of RE derivatives as is described in [1]. The efficiency of this approach is measured in terms of the ratio of dissimilar REs up to a given length that produces minimal DFAs using the aforementioned derivative approach.

2 Background

A regex or a regular expression, is a string of text that allows you to create patterns that help match, locate, and manage text. DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.

2.1 Regular Expressions

The syntax for REs is defined recursively for some alphabet $\Sigma = \{c_1, c_2, ...\}$, where c_i is a unique character for the alphabet. The formal definition [1] for this with regards to character sets is shown in Table 1. Here however the definition was slightly modified to distinguish the empty character set.

r, s	::=	$\emptyset$	The empty set
	|	$\mathcal{S} \subseteq \Sigma$	A non-empty subset of the alphabet
	|	ϵ	The empty string
	|	$\neg r$	Compliment
	|	r^*	Kleene Star
	|	$r + s$	Logical Or (Union)
	|	$r \& s$	Logical And (Intersection)
	|	$r \cdot s$	Concatenation

Table 1: Recursive Regular Expression Definition

The size of a RE constructed using this recursive definition, $|r|$ is defined simply as the amount of operators and elements from Table 1 used to express to the RE.

2.2 Nullability and Derivatives

Nullability as the function $\nu(r)$ specifies whether a RE r accepts the empty string. Specifically $\nu(r) = \epsilon$ implies that r accepts the empty string and $\nu(r) = \emptyset$ implies that r does not. This is used in computing the derivative $\delta_{c_i} r$ of r later on. It is governed by the rules listed in Table 2

$$
\begin{array}{rcl}
\nu(\emptyset) & = & \emptyset \\
\nu(\epsilon) & = & \epsilon \\
\nu(\alpha) & = & \emptyset \\
\nu(r \cdot s) & = & \nu(r) \& \nu(s) \\
\nu(r + s) & = & \nu(r) + \nu(s) \\
\nu(r^*) & = & \epsilon \\
\nu(r \& s) & = & \nu(r) \& \nu(s) \\
\nu(\neg r) & = & \begin{cases} \epsilon, \text{if } \nu(r) = \emptyset \\ \emptyset, \text{if } \nu(r) = \epsilon \end{cases}
\end{array}
$$

Table 2: Recursive Regular Expression Definition

Using this we can compute the derivative of a RE with respect to some character a. The rules for this is listed in Table 3. In order to see whether a RE accepts a string, the RE is derived with respect to each character of the string sequentially and we find that it only accepts the string if the RE produced by this process is nullable itself.

$$
\begin{array}{rcl}
\partial_a \epsilon & = & \emptyset \\
\partial_a a & = & \epsilon \\
\partial_a b & = & \emptyset \text{ for } b \neq a \\
\partial_a \emptyset & = & \emptyset \\
\partial_a (r \cdot s) & = & \partial_a r \cdot s + v(r) \cdot \partial_a s \\
\partial_a (r^*) & = & \partial_a r \cdot r^* \\
\partial_a (r + s) & = & \partial_a r + \partial_a s \\
\partial_a (r \& s) & = & \partial_a r \& \partial_a s \\
\partial_a (\neg r) & = & \neg (\partial_a r)
\end{array}
$$

Table 3: Recursive Regular Expression Definition

3 Implementation

3.1 Libraries Used

- **HaLeX** : Although this library allows for the representation of both REs and DFAs, it does not incorporate all of the desired simplification methods for REs, specifically with regards to the Kleen Star and a standard structure for nested binary operators. Therefore only its DFA functionality was used for computing minimal DFAs.
- **unordered-containers** : Used for its implementations of HashMaps and HashSets.
- **sort** : Used specifically for its `uniqueSort` function which quarantees the removal of duplicates during the sort.

- **hashable** : Required to create hashable data structures for HashMaps and HashSets.
- **QuickCheck** : Used to create generators for the implemented RE data structure for automated testing.
- **hspec** : Used for unit testing.
- **hspec-core** : Required to use alter the amount of random tests when using hspec in conjunction with **QuickCheck**.

3.2 RE Representation

The REs for this implementation are represented as a tree-like structure, where

- Operands ($\{\mathcal{S}, \emptyset, \epsilon\}$) are represented by leaf nodes.
- Unary Operators ($\{*, \neg\}$) are represented by internal nodes with exactly one child.
- Binary Operators ($\{+, \&, \cdot\}$ are represented by internal nodes with exactly two children.

For visualization of the REs, the operators are used in prefix form, without parenthesis to represent the RE in a concise manner. Table 4 shows some examples for this. Character sets are enclosed with [...]. The infix form is however used in the paper for easier interpretation.

Prefix Pattern	Infix Pattern	Validity
$\cdot[a][b]$	$[a] \cdot [b]$	Valid.
$\cdot + \epsilon\epsilon\epsilon$	$(\epsilon + \epsilon) \cdot \epsilon$	Valid.
$\cdot\epsilon + \epsilon\epsilon$	$\epsilon \cdot (\epsilon + \epsilon)$	Valid.
$\&[a]$	$[a]\&(???)$	Invalid. There are not enough operands to fill all the operators.
$+[a][b]e$	$(???)$	Invalid. There are more operands than all of the operators require.

Table 4: Regular Expression Visualization

Two pitfalls with this representation is that it is not always intuitive to interpret the prefix representation by hand, and it requires explicit representation of the concatenation operator ($\cdot$) in order as an alternative for parenthesis to prevent ambiguity. However, this format is easily transformed into the aforementioned tree-like structure.

3.3 RE Similarity Classes and Simplified Formats

Given the statements of similarity described in [1], we are able to create the definition for a simplified RE. To do this however we also require a standard format for simplified REs. To accomplish this, we construct a structural order for REs, such that for any two REs, r and s, the following holds:

- $r = r$
- $r = s \iff s = r$
- $r < s \iff s > r$

Given the above structural comparison operators and REs, r, s, t, we can derive the of a RE's similarity class, $\mathcal{C}(r)$, recursively, using these statements of similarity. We include $(r*) \cdot (r*) \approx (r*)$ in this list, as it is easily shown to be true, in an attempt to keep the derivative of the similar REs from affecting their similarity, however it would appear that more research is required in order to accomplish this. Furthermore, given r that which is a list of $r_1, r_2, ..., r_n$ of unique class labels ($r_i \neq r_j \forall i \neq j$), linked with a single type of transitive binary operator (such as $+$) and some operand precedence governed by nested parenthesis, the class label for the entire list is given as $\mathcal{C}(r) = r_{k_1} + (r_{k_2} + (...))$, where $k_1, k_2, ..., k_n$ is simply a reordering of $1, 2, ..., n$, such that $r_{k_1} < r_{k_1} < ... < r_{k_n}$. For the intransitive $\cdot$ operator, $\mathcal{C}(r) = r_1 \cdot (r_2 \cdot (...))$. This definition of $\mathcal{C}(r)$ reduces any RE r to its class label. A RE is said to be simplified if and only if $\mathcal{C}(r) = r$. This increases the performance when attempting to produce a dissimilar list for REs up to some d, by creating the list of all REs up to d, and simply pruning the elements for which $\mathcal{C}(r) \neq r$. This overcomes the need to compare the individual elements of such a list with each other.

3.4 DFA Production and Minimization

The production of DFAs construction of four element tuple required to construct a DFA [1] was managed without the use of external libraries, except for container data types and sorting methods which acted as helper functions. The production and minimization of DFAs however was outsourced to **HaLex**.

4 Computation Results

For $1 \leq i \leq 8$, the following results were collected:

- The set of dissimilar REs (r) for the alphabet $\{a, b\}$, such that $|r| \leq i$. Call this set $\mathbf{D}$.
- The subset $\mathbf{M} \subseteq \mathbf{D}$ of REs which produce minimal DFAs using the derivative approach.
- The maximum size of all DFAs produced by the REs of $\mathbf{M}$.

The results are compiled in Table 5

RE size	Dissimilar REs ($\|\mathbf{D}\|$)	Minimal DFAs ($\|\mathbf{D}\|$)	Maximum Minimal DFA size
1	5	5	3
2	13	13	3
3	39	30	4
4	187	139	5
5	847	474	5
6	4327	2289	7
7	22910	10408	7
8	123735	49971	11

Table 5: Dissimilar RE and minimal DFA results for alphabet $\{a, b\}$.

5 Testing Strategy

This section will describe the testing strategy used in detail. Testing was implemented with Hspec and Quickheck.

5.1 Hspec

Hspec was used to perform unit testing for non automated tests, for trivial or known cases.

5.2 Quickcheck

QuickCheck was used to confirm the robustness of the program, by making use of the generator which was written to generate REs. The algorithm below represents this process of constructing a RE tree-like data structure for the alphabet $\{a, b\}$.

function GENERATERE(d, p)
  if $d = 0$ **then return** $\emptyset$
  else if $d = 1$ **then return** GEN($\{\emptyset, \epsilon, [a], [b], [ab]\}$)
  else if $d = 2$ **then return** GEN($\{\neg, *\}$) of GENERATERE($d - 1$, p)
  else
    $q \Leftarrow$ GEN($[0.0, 1.0]$)
    if $q \leq p$ **then**
      $dl \Leftarrow$ GEN($\{1, 2, ..., d - 2\}$)
      $r \Leftarrow$ GENERATERE(dl, p)
      $s \Leftarrow$ GENERATERE($d - dl$, p) **return** GEN($\{+, \&, \cdot\}$) of r and s
    else return GEN($\{\neg, *\}$) of GENERATERE($d - 1$, p)
    end if
  end if
end function

Here GEN is a generating function which uniformly selects an element from a list or interval, d is the size of the RE which we require and q is the probability of selecting a binary operator instead of a unary operator for the current node of the tree. For this implementation the initial function is called with $\text{GEN}(\{1, 2, ..., 25\})$ and $q = \frac{5}{6}$.

The description of the underlying process is that at each node in the tree, knowing how much we should still grow the tree (d), we either:

- Turn the node into a leaf if $d = 1$, labelling it with a RE of size 1.
- Label the node with unary operator, and create a single child RE of size $d - 1$.
- Label the node with a binary operator, and create two children of sizes d_1 and d_2, such that $d_1 + d_2 + 1 = d$.

5.3 Base Case Sanity Checks

The following base cases were tested to ensure program correctness. Note that $=$ refers to exact structural equivalence.

- The correctness with respect to nullability for Table 2 where $r = [a]$ and $s = [b]$.
- The correctness of the simplification process on some trivial cases.

5.4 Simplification and Derivative Sanity

For simplification indempotency was tested, as well as language preservation. Both of these tests are conducted using randomly generated REs as described in 5.2, with 10000 random cases for indempotency and 1000 cases for language preservation, where the output of each randomly generated RE (r) and its simplification $\mathcal{C}(r)$ is compared across 400 randomly generated words of length $1 \leq l \leq 20$ from the alphabet $\{a, b\}$.

5.5 Dissimilar REs List Generation

The process of creating a list of dissimilar REs up to size d starts with creating the list of all valid REs up to that size and then filtering out all elements for which $\mathcal{C}(r) \neq r$. In order to ensure correctness of the program in this regard, these lists were computed by hand for sizes $d = 1, 2, 3$ and compared with the results given by the program. Furthermore, for $d = 1, 2, ..., 8$, it was ensured that for the dissimilar list $\mathbf{D}$:

$$\mathcal{C}(r_i) \neq \mathcal{C}(r_j) \; 1 \leq i < j \leq |\mathbf{D}|$$

5.6 Equivalence Testing

With regards to testing the language equivalence between a RE and its generated DFA, 1000 random REs are generated according to the algorithm in Section 5.2. For each such random RE, 400 random words of of length $1 \leq l \leq 20$ from the alphabet $\{a, b\}$ is generated. A DFA is constructed for the RE and for each word the outputs of the RE and DFA is are compared. The test will only return true if for all 1000 random REs with 400 random testing words each, the RE and DFA returns the same result.

6 Result validation

The is not much that can be said in this regard, except maybe that all the tests passed for the final version of the program, such that the program can be said to be adequately robust for general use cases.

7 Conclusion

This project gave us as students a hands-on experience with reusing projects and libraries to build a new Haskell program. The resulting program performs RE conversions to DFAs. Comprehensive testing was also implemented using various techniques. Finally, this report validates results obtained by the program.

References

[1] OWENS, S., REPPY, J., & TURON, A. (2009). Regular-expression derivatives re-examined. Journal of Functional Programming, 19(2), 173-190. doi:10.1017/S0956796808007090