Universiteit Leiden
Opleiding Informatica

Equivalence checking of regular expressions

using non-deterministic finite automata

Name:	Arthur Stuivenberg
Studentnr:	s0870943
Date:	14-01-2014
1st supervisor:	Marcello Bonsangue
2nd supervisor:	Jurriaan Rot

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction

Solving equivalence of regular expressions can be quite interesting for computer scientists. For example, say you found an interesting regular expression but it's complex and long, why not find an easier regular expression depicting the same language. Or something that seems trivial but actually is not like wondering if a language is empty.
In this paper we present two implementations which can check equivalence of regular expressions. The implementations are based on the creation of an non-deterministic automaton using a linearized version of the RE's involved. When created we can use the both automata for the bisimulation up-to techniques presented in [1]. We will explain the basic definitions in Section 2. Section 3 contains an explanation of the programming language Maude in which the implementation is written. The implementation itself is explained in section 4. The results are to be found in section 5 and we will conclude this paper in section 6. Finally the Appendix shows the source code of our implementation in Maude.

2 Theory

The definitions introduced in this section are from [2]. An alphabet Σ is a finite set of symbols. Throughout this paper we will use alphabet $\Sigma = \{a, b\}$ unless stated otherwise. The set of all strings over Σ is defined as Σ^* and to represent such a string we use u, v. A language L is a set of strings, $L \subseteq \Sigma^*$. For a string u, $|u|$ denotes the length (number of symbols) of u. The null-string λ is a string over Σ and by definition $|\lambda| = 0$.

2.1 Regular Expressions and Regular Languages

The syntax of regular expressions (RE) includes the operations: concatenation, Kleene-closure and alternation (also called logical-or). In addition it is extended with the Boolean operation and. Throughout this paper we will use r, s and t to denote regular expressions.

Definition 1 *A regular expression is defined as:*

1. $\emptyset$ *is a regular expression*
2. λ *is a regular expression*
3. a *is a regular expression, for every alphabet symbol* $a \in \Sigma$
4. $r \cdot s$ *is a regular expression if* r *and* s *are regular expressions*
5. $r + s$ *is a regular expression if* r *and* s *are regular expressions*
6. $r\&s$ *is a regular expression if* r *and* s *are regular expressions*
7. r^* *is a regular expression if* r *is a regular expression*
8. *nothing else is a regular expression*

Definition 2 *The language of a regular expression* r *is the smallest set of strings* $L(r) \subseteq \Sigma^*$ *generated by the following rules:*

1. $L(\emptyset) = \emptyset$
2. $L(\lambda) = \{\lambda\}$
3. $L(a) = \{a\}$, $a \in \Sigma$
4. $L(r \cdot s) = \{u \cdot v | u \in L(r)$ *and* $v \in L(s)\}$
5. $L(r + s) = L(r) \cup L(s)$
6. $L(r\&s) = L(r) \cap L(s)$
7. $L(r^*) = \{\lambda\} \cup L(r \cdot r^*)$

All languages that can be described by the rules above are called regular languages.

2.2 Finite automata

A finite automata is a model of a computation device, which acts as a language acceptor. In this paper we will use a non-deterministic finite automaton (NFA) but let's first see what a deterministic finite automata (DFA or FA) looks like:

Definition 3 *A deterministic finite automaton is a 5-tuple* $(Q,\ \Sigma,\ q0,\ A,\ \delta)$ *where:*

- Q *is a finite set of states*
- Σ *is a finite input alphabet*
- $q0 \in Q$ *is the initial state*
- $A \subseteq Q$ *is the set of accepting states*
- $\delta : Q \times \Sigma \to Q$ *is the transition function*

Definition 4 *The extended transition function* $\delta^* : Q \times \Sigma^* \to Q$ *is defined as follows:*

- $\delta^*(q,\ \lambda) = \{q\}$
- $\delta^*(q,\ au) = \delta^*(\delta(q,\ a),\ u)$

The language of a DFA M is $L(M) = \{u \in \Sigma^* | \delta^*(q0,\ u) \in A\}$

Having these definitions we can state Kleene's theorem: For every alphabet Σ, every regular language over Σ can be accepted by a finite automaton. According to Kleene's theorem the reverse is also true: for every finite automaton $M = (Q,\ \Sigma,\ q0,\ A,\ \delta)$, the language $L(M)$ is regular. Proofs for both parts of this theorem can be found in [2].

A non-deterministic finite automaton has the same definition as a DFA but with a different transition function:

Definition 5 *A non-deterministic finite automaton is a 5-tuple* $(Q,\ \Sigma,\ q0,\ A,\ \delta)$ *where:*

- Q *is a finite set of states*
- Σ *is a finite input alphabet*
- $q0 \in Q$ *is the initial state*
- $A \subseteq Q$ *is the set of accepting states*
- $\delta : Q \times \Sigma \to 2^Q$ *is the transition function*

The transition function is altered in such a way that transitions while reading nothing (λ) are possible and it's also possible that for a combination state, input-letter there are multiple transitions possible. The extended transition function for an NFA looks as follows:

Definition 6 *The extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$ is defined as follows:*

1. $\delta^*(q,\ \lambda) = \{q\}$
2. *For every $q \in Q$, every $u \in \Sigma^*$, and every $a \in \Sigma$,*

$$\delta^*(q,\ ua) = \bigcup\{\delta(p,\ a) | p \in \delta^*(q,\ u)\}$$

2.3 Linear Expressions

In order for us to construct the NFA from an regular expression we need to linearize the regular expression. The following definitions are from [3]. Let r be a regular expression over the alphabet Σ_r. The alphabetic width of r, denoted by $||r||$, is the number of occurrences of symbols in r. r is said to linear over Σ_r if and only if every symbol of $_r$ occurs (at most) one time in r. For all $[1,\ ||r||]$, if a is the jth occurrence of symbol in r, then the pair $(a,\ j)$ is called a position of r. By abuse of notation $(a,\ j)$ is written a_j.

Definition 7 *The set of positions of r is denoted by Pos_r:*

- $Pos_r = \{a_j | a \in \Sigma_r,\ j \in [1,\ ||r||]\}$

Let Σ' be an alphabet of size $||r||$, and $\mu_{\Sigma'}$ a one-to-one mapping from Pos_r onto Σ'. The expression r' over Σ' obtained from r by replacing symbol a of rank j by $\mu_{\Sigma'}(a_j)$, for all j in $[1,\ ||r||]$, is called a linearization of r. Any linearization r' is linear over Σ'. The linearized version of r, denoted by $\overline{r}$ is its linearization over Pos_r.

Let us take for example $r = a{\cdot}(a{+}b){+}(a{+}b){\cdot}(1{+}b)$. We have $Pos_r = \{a_1,\ a_2,\ b_3,\ a_4,\ b_5,\ b_6\}$, and $\overline{r} = a_1 \cdot (a_2 + b_3) + (a_4 + b_5) \cdot (1 + b_6)$.

Let h_r be the alphabetic mapping from Pos_r to Σ_r such that $h_r(a_i) = x,\ \forall i \in [1,\ ||r||]$, and $h_r(\overline{r}) \equiv r$. Then for every linearization r' over Σ', $h_{r'} = h_r \circ \mu_{\Sigma'}^{-1}$ is a mapping from Σ' to Σ_r such that $h_r(r') \equiv r$. In our current example, we have: $h_r(a_1) = h_r(a_2) = h_r(a_4) = a$ and $h_r(b_3) = h_r(b_5) = h_r(b_6) = b$.

2.4 From regular expression to NFA

Now that we know how to linearize a regular expression we can construct a non-deterministic finite automaton corresponding to that regular expression. Let $\overline{r}$ be a linearized regular expression and $M = (Q,\ \Sigma,\ q0,\ A,\ \delta)$ an NFA. How can we choose M such that it corresponds with $\overline{r}$? Let's choose Σ to be Σ_r. Then we define Q to be:

$$Q = \{0\} \cup \{a_j | \forall j \in [1,\ ||r||]\}$$

So M has the same number of states as symbols in $\overline{r}+1$. The special state 0 will be the initial state, so $q0 = \{0\}$. Defining A is a little harder, we want A to contain all states corresponding to all $a_j \in \overline{r}$ in which $\overline{r}$ can end, or $Last(r)$:

$$A = \begin{cases} Last(r) \cup \{0\} & \text{if } \lambda \in L(r) \\ Last(r) & \text{otherwise} \end{cases},$$

where $Last(r) = \{a \in Pos_r | ua \in L(\overline{r})\}$, the set of positions that match the last symbol of some word in L($\overline{r}$).

We split the transition function into two definitions, transitions from the special initial state 0, and transitions from all other states.

$$\delta(0,\ a) = \{x \in First(r) \mid h(x) = a\},\ \forall a \in \Sigma,$$

$$\delta(x,\ a) = \{y \mid y \in Follow(r,\ x)\ and\ h(y) = a\},\ \forall x \in Pos_r,\ \forall a \in \Sigma,$$

where $First(r) = \{a \in Pos_r | av \in L(\overline{r})\}$, the set of positions that match the first symbol of some word in $L(\overline{r})$, and $Follow(r,\ x) = \{y \in Pos_r | uxyv \in L(\overline{r})\}$, for all x in Pos_r: the set of positions that follow the position x in some word of $L(\overline{r})$. $First$, $Follow$ and $Last$ are explained in more detail in [3].

2.5 Bisimulation up-to

So now that we can create a non-deterministic automaton corresponding to a regular expression we can start the actual equivalence checking. Naturally we will start simulating both NFA's in the initial state, and from there save all equivalent sets $(S,\ T)$ to an equivalence relation R we find by executing the NFA's step by step ($S \subseteq Q_1$, $T \subseteq Q_2$, where Q_1 is the set of states of the first NFA, and Q_2 of the second). S and T are equivalent if and only if $S \subseteq A_1 \Leftrightarrow T \subseteq A_2$ and $\delta_1(S,\ x) \subseteq A_1 \Leftrightarrow \delta_2(T,\ x) \subseteq A_2, \forall x \in \Sigma$, where A_1 and A_2 are the sets of accepting states and δ_1 and δ_2 the transition functions of the first and second NFA's respectively. With this knowledge we obtain the following algorithm:

```
function BiSimulation(S, T) {
    if R contains (S, T)
        return 1
    if S equiv T
        add (S, T) to R
        return BiSimulation(Follow(S, a), Follow(T, a)) &&
            BiSimulation(Follow(S, b), Follow(S, b))
    else
        return 0
}
```

This recursive function adds all new found equivalence classes $(S,\ T)$ to R until either it finds a set $(S,\ T)$ which isn't equivalent, or it finds a set $(S,\ T)$ which has already been added to R which obviously won't generate new sets. This function is called with $S = \{0\},\ T = \{0\}$ and $R = \{(\emptyset,\ \emptyset)\}$ to begin with, furthermore $BiSimulation$ should be called with $Follow(S,\ x),\ Follow(T,\ x),\ \forall x \in \Sigma$ as arguments.

Definition 8 $Follow(S,\ a)$ *returns:*

$$\bigcup \{follow(x,\ a)\},\ \forall x \in S$$

with $follow(x,\ a)$ *as defined in the previous section.*

To achieve bisimulation up to congruence (as described in section 3.2 of [1]), called $c(R)$, we need to define a few properties to which R should satisfy:

- identity: $\frac{x\ R\ y}{x\ c(R)\ y}$
- reflexive: $\frac{}{x\ c(R)\ x}$
- symmetric: $\frac{x\ c(R)\ y}{y\ c(R)\ x}$
- transitive: $\frac{x\ c(R)\ y\ \ y\ c(R)\ z}{x\ c(R)\ z}$
- context: $\frac{X_1\ c(R)\ Y_1\ \ X_2\ c(R)\ Y_2}{X_1 + X_2\ c(R)\ Y_1 + Y_2}$

3 Maude

The implementation which we will explain about in the next section is written in Maude [4] . "So what is Maude? Maude is a programming language that models systems and the actions whitin those systems." [5]. This section contains some of the basics of Maude and just enough of them to show why we choose Maude instead of any of the established languages (C++, JAVA, ..). Unlike those languages Maude is interpreted and not compiled. The main power of Maude is pattern matching and replacing. Once you have written a Maude program (called a module) you can load it in Maude and perform operations on the definitions within the module.

Example 1 *A Simple Maude module: (taken from [5])*

```
mod Animals is
    sorts Animal Dog .
    subsort Dog < Animal .
    ops bloodhound terrier pitbull schnauzer : -> Dog .
    op penguin : -> Animal .

    op breed : Dog Dog : -> Dog .
endfm
```

In this module we define two sorts, Animal and Dog, after which we say that a Dog is a subsort of a Animal. Bloodhound, terrier, pitbull and schnauzer are a defined to be of type Dog, and penguin is defined an Animal. The operation breed takes two arguments (of type Dog) and returns Dog. So when we ask Maude to reduce breed with two arguments, like reduce breed(terrier, pitbull); it will return a positive result: Dog. Now say we will give it two other arguments like reduce breed(penguin, schnauzer); it will not return a positive result because penguin is not of type Dog and Animal is not a subsort of Dog. The reduce command in Maude is used to reduce an expression (like above) as far as possible. When no more reductions are possible, Maude returns the result.

3.1 Equations

Maude uses equations to reduce the left part of an expression to the right part of an expression. This example will show how a simple equation is defined.

Example 2 *Maude equations*

```
mod SimpleEq is
    sorts RegExp .
    op _+_ : RegExp RegExp -> RegExp [comm] .
    var E : RegExp .
    eq 0 + E = E.
endm
```

This module defines only one sort: $RegExp$, and an operator on two $RegExp$'s, $+$. The underscores on each side of the $+$ denote that this operator can be used as an infix operator. Also, we defined the operator as commutative (with the keyword $comm$ in betwoon brackets) which means the order of the arguments this operator takes doesn't matter. We define E as a variable which we can then use in equations. The equations we have defined says if there's a regular expression E and it's added with zero (or zero is added to it because of the commutativity) Maude is allowed to reduce it to just E. These types of equations are very useful because it allows quick reducing to simpler expressions.

3.2 Why Maude?

So why would we use Maude and not any of the established languages like C++ or JAVA. The main reason is because the implementation in Maude is much easier. Data structures and their relations are defined within one or two lines of code and parsing is done by Maude so you don't need to worry about for example verifying the input. The following example shows how easy something can be implemented in Maude.

Example 3 *Making elements in a list unique*

```
Eq O | O = O .
Eq O | P | O = O | P .
```

In these two lines we have two variables, O which is of type $Element$ and P which is of type $List$. The operator | in this case is the separator of two elements in a list. So what this does is when two Elements are the same ($O = O$) and they are in the list next to each other ($O|O$) the first equation will remove the duplicate. The second equation will remove a duplicate if there are other elements separating the two duplicates ($O|O_1|O_2|...|O_i|O$ will be reduced to $O|O_1|O_2|...|O_i$). So to make a list contain unique elements only, Maude only needs two lines of code whereas for example in C++ you need at least a nested for loop or some data structure to store elements you already encountered.

The reader might ask "So, you never use loops in Maude?". The answer is no, it is not even possible to write a loop as we know it from other programming languages. Of course there are other ways to loop like recursion. Consider these lines of code:

Example 4 *Recursion*

```
op length : RegExp -> Nat .
ceq length(A E) = 1 + length(E) if E =/= 1 .
eq length(A) = 1 .
```

In this example we have variables A and E which are used for an alphabet letter and a regular expression respectively. Instead of looping over the regular expression and counting the alphabet letters we can use recursion the count the length of the regular expression fed to this equation. The first equation (a conditional equation to be explained below) may be confusing because length expects only one argument and it looks like length takes two, A and E, but something

else is going on here. The $space$ between the A and E actually is an operator, namely the concatenation operator, so A and E concatenated results in another regular expression. The function accepts $A\ E$ as a regular expression but in the computation of the equation A and E can be used separately now. So if for example we call $reduce\ length(a\ b\ b\ b)$. A will be a and E will be $b\ b\ b$ on the right side of the equation. We know that A contains one alphabet letter (by definition of the alphabet letter, not shown here) so we can count that as 1 and need to add the length of the rest (E) to that, hence $1 + length(E)$. At some point (in our example when length is called with the last b, of our original input $a\ b\ b\ b$) Maude will try to match a single alphabet letter with $length(A\ E)$ and will succeed. This is because concatenation has the identity property 1 ($E\ 1 = E$ but also $E = E\ 1$) so in our example Maude will concatenate the b we have left with 1. But that's not what we want, so we use a conditional equation saying Maude can only use that equation for reducing if the if-statement is true. In our case $E = / = 1$ is false (because $= / =$ denotes "not equal" in Maude) so Maude cannot use that equation and will instead use the equation on the following line which states that the length of a single alphabet letter is 1. And so our input string $a\ b\ b\ b$ will have the following derivation:

$length(a\,b\,b\,b) \rightarrow 1{+}length(b\,b\,b) \rightarrow 1{+}1{+}length(b\,b) \rightarrow 1{+}1{+}1{+}length(b) \rightarrow 1{+}1{+}1{+}1 \rightarrow 4$

4 Implementation

The goal of the implementation is to write several modules which use the techniques described in section 2 to ultimately check if two regular expressions are equivalent. The following modules are present in the implementation in Maude:

- $RegularExpression$, models a regular expression and the appropriate operations/definitions.
- $Linearize$, linearizes a regular expression.
- $GenAut$, generates an automaton (non-deterministic) from a given regular expression.
- $Compare$, using modules above, compares two regular expressions on equivalence.

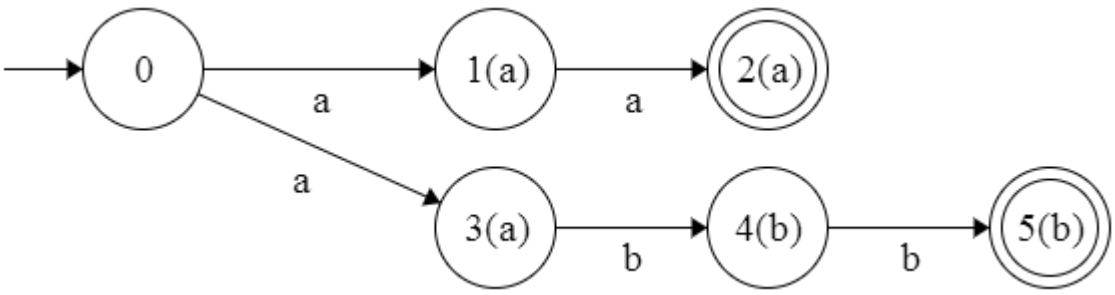
The module $RegularExpression$ will not be discussed as this is the set of basic definitions and some equations as described in section 2.

4.1 Non-deterministic automaton

The NFA will be created by the module $GenAut$. Section 2.4 describes how to generate a NFA corresponding to a regular expression, a procedure that we used in the implementation as well. In the implementation a NFA is represented as a combination of transitions and accepting states only. All states are represented using natural numbers. It is not necessary to store a list of states information as these can be extracted from the transitions. Transitions are stored in a list as follows: $< Transition > | < Transition > |...$, where $< Transition >= [x,\ a,\ y]$. The triple $[x,\ a,\ y]$ corresponds with the transition $\delta(x,\ a) = y$.

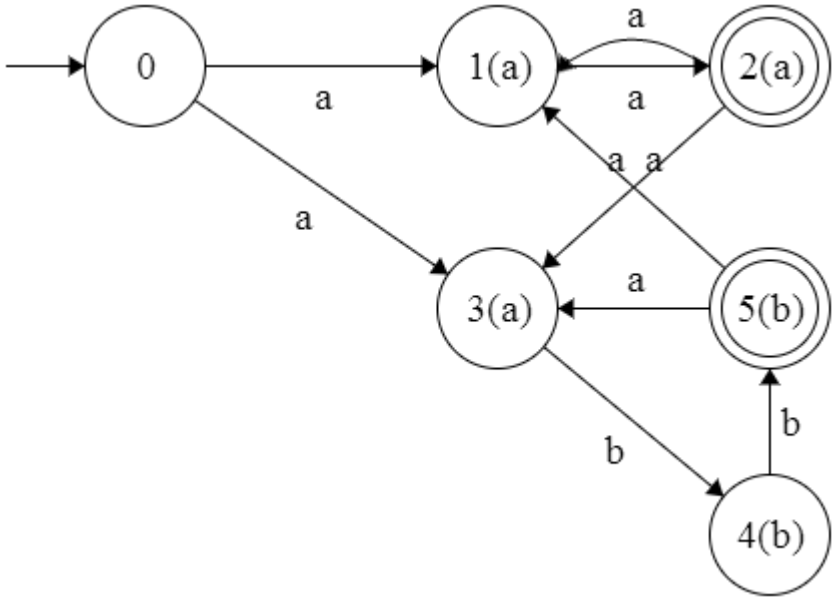
4.1.1 Generating Transitions

The module $GenAut$, and in particular the function $genaut$ returns a list of transitions corresponding to a regular expression. It is a recursive function which keeps track of the number of symbols in linear expression $\overline{r}$ it processed already, I. Every time a new symbol in $\overline{r}$ is processed that symbol will correspond to a new state $I + 1$. There are a few special cases to consider:



This is the NFA corresponding to the regular expression $a\ a + a\ b\ b$. In this case two transitions from the initial state have to be created, after which $genaut$ can be called recursively the remaining parts a $(I = 1)$ and $b\ b$ $(I = 3)$. As described in section 2.4 the function $First$ is used to generate transitions from the initial state, and $Follow$ is used to generate all other transitions.

It will become even more interesting when processing for example $(a\ a\ +\ a\ b\ b)^*$. Not only will transitions of the form $Last(a\ a)$ to $First(a\ a)$ be generated, but also transitions of the form $Last(a\ a)$ to $First(a\ b\ b)$ and $Last(a\ b\ b)$ to $First(a\ a)$. This simple regular expression will generate 9 transitions.



4.1.2 Finding accepting states

The function $Last(\overline{r})$ will find the symbols in which a linear expression $\overline{r}$ can terminate as described in section 2.4. This shows how easy Maude can be in these situations as you can basically just copy the rules from the theory into Maude:

```
ceq last(E1, I) = 1 if I == 0 and nullable(E1) == 1 .
ceq last(E1, I) = 1 if c(E1, I) == 1 .
ceq last(E1, I) = 1 if nullable(c(E1, I)) == 1 .
eq last(E1, I) = 0 [owise] .
```

4.2 Comparing regular expressions

Function $compare$ is the function (in module $Compare$) the user will call which in turn will use the previously explained modules to compare the two regular expressions the user provides as input.

First, the function will call the $Linearize$ module to linearize both regular expressions whose result will then be fed to $GenAut$. This, as we've read in the previous section, will return two sets of transition-lists, followed by initializing the current state lists and the equivalence relation R. $compare$ will then check if both automata accept the empty string (λ), both initial states accepting or both not. If this check fails it's easy to see that both regular expression are not equivalent. When the check succeeds the current states are saved to the equivalence relation. From this moment the algorithm reaches a repetitive pattern: advancing in the NFA's, checking if the current states are in the equivalence class, if so, return 1, if not, checking if the current states are still equivalent, if so, adding them to the equivalence class and repeat, if not

returning 0. Advancing in the NFA means looking for transitions from the current state with all symbols from the alphabet and save the newly reached states as current states. Say C_1, C_2 are the current states of the first and second NFA respectively then C_1 and C_2 are equivalent $S \subseteq A_1 \Leftrightarrow T \subseteq A_2$ and $\delta_1(S,\ x) \subseteq A_1 \Leftrightarrow \delta_2(T,\ x) \subseteq A_2, \forall x \in \Sigma$, where A_1 and A_2 are the sets of accepting states and δ_1 and δ_2 the transition functions of the first and second NFA's respectively. When C_1, C_2 is reached and this is already part of the equivalence relation (R) it means it has been reached before it means no more pairs of current states will be found, and since both are still equivalent we can conclude that so are the regular expressions.

4.3 Bisimulation

As described in section 2.5 the equivalence relation R should satisfy five properties to achieve bisimulation up to congruence. However, in our implementation these properties are not applicable: reflexive, symmetric, transitive. The way we fill R is by adding relations of the form $(X_1,\ Y_1)$ to the relation, where $X_1 \subseteq Q_1$ and $Y_1 \subseteq Q_2$. Q_1 is the set of states of the NFA corresponding to the first regular expression, and Q_2 the set of states of the second. Say we did want to apply reflexivity to our relation it would result in elements of the form $(X_1,\ X_1)$ which is a useless pair because the only types of pairs we will search for in the relation are those of the form $(X_1,\ Y_1)$ as described above. The same goes for the symmetric and transitive properties.
Our implementation keeps track of two lists, the first for all found pairs $(X_1,\ Y_1)$ (as described above) and the other which contain the same pairs $(X_1,\ Y_1)$ and has the context property applied to it. So whenever a new pair $(X_1,\ Y_1)$ is found first we check if it's in the second list and if not we add it to both lists and continue from there. If that newfound pair is in the second list, we can return 1 as the algorithm in section 2.5 states. We only take pairs from the first list to continue our execution of the algorithm. When at some point the first list is found to be empty and the equivalence still holds we can also return 1. If at any point during execution the equivalence does not hold 0 is returned.

5 Experimental results

This table shows some results of comparing the regular expressions in the first two columns with the implementation described in the previous section.

r_1	r_2	output	no. of rewrites	no. of rewrites (congruence)
a	b	0	112	149
a	a	1	182	165
ab	b	0	151	247
$a+b$	$b+a$	1	364	390
$a+ab$	$b+a$	0	345	320
a^*	a	0	4	4
b^*	b^*	1	219	202
$(a+b)^*$	$(b+a)^*$	1	563	589
$(a+b)^*$	$(a+b)^*$	1	455	481
$a\ a\ a$	$b\ b\ b$	0	312	2092
$a\ a\ a$	$a\ a\ a$	1	784	793
$(a\ b+b\ a)^*$	$(b\ a+a\ b)^*$	1	1671	2723
$(a\ b+b\ a)^*$	$(b\ b+a\ a)^*$	1	1147	1302
$(a+(b\ a+a\ b)^*)^*$	$((a\ b+a)^*+b\ a)^*$	1	2728	3355
$(a+(b\ +a\ b)^*)^*$	$((a\ b+a)^*+b\ a)^*$	0	1234	1209
$(a+b)^*\ a\ b\ (a+b)^*$	$a\ b$	0	698	2840
$ab+b\ (a\ b+b)$	$a\ b+b$	0	850	825

This table shows the results of checking the equivalence of regular expressions r_1 and r_2. The output column is either 0 or 1, not equivalent or equivalent. The last two columns are the most interesting because they show how much rewrites Maude had to do to determine equivalence, which we aimed to get as low as possible. The first of the last to columns is the number of rewrites Maude had to do without the congruence property (see section 2.5) applied to the equivalence relation R and the second is with that property applied.

6 Conclusion

The previous section offers an overview of the experiments we tested. The output of all tested regular expression pairs is what we expected it to be, so we succeeded in our implementation of checking for regular expression equivalence. However, it is more interesting to look at the last two columns.
We expected the implementation which applied the congruence property (section 2.5) to the equivalence relation R to require less rewrites than the implementation without that property. As it turns out both implementations require more or less the same amount of rewrites to determine equivalence. This could be due to the fact that Maude needs a lot of rewrites to apply the context property (as described in section 2.5) and thus generates a lot of overhead. The equivalence relation is saved with the context property applied so the property doesn't have to regenerate all pairs every iteration, it still has be applied every time a new element is added to the relation which requires to combine all elements with the freshly added element which could also add new elements which in turn need to combined again. As you can see this could be a time consuming operation.
So even though the amount of rewrites needed to compare two regular expressions might be lower with the congruence property applied, because of the overhead the results appear about the same for both implementations.

For future work a few things can be done. First of all extending the alphabet (instead of just $\Sigma = \{a,\ b\}$ on which the implementation works can improve the testing done on regular expressions. More testing can come in handy to verify the just described reason for the implementation with the congruence property not being quicker than the implementation without. Another thing to research is optimizing the algorithm for applying the context property to the equivalence relation, if this can be achieved the results will certainly show an increase of speed (a decrease of rewrites needed) to determine equivalence. A last optimization can be to hardcode a few often occurring forms of regular expressions and their equivalence which can easily be done in Maude.

Appendix

This appendix shows the Maude source code of the implementation with the congruence property (see section 2.5) applied to the equivalence relation R.

```
mod RegularExpression is
  protecting NAT .
  sorts RegExp Alphabet .
  subsort Alphabet < RegExp .

  ops a
  b : -> Alphabet .
  ops 1
  0 : -> RegExp .
  op _+_ : RegExp RegExp -> RegExp [ ctor assoc idem prec 30 ] .
  op _&_ : RegExp RegExp -> RegExp [ ctor assoc comm prec 25 ] .
  op __ : RegExp RegExp -> RegExp [ ctor assoc prec 20 ] .
  op _* : RegExp -> RegExp [ ctor prec 10 ] .
  op length : RegExp -> Nat [memo] .

  vars E E1 E2 : RegExp .
  vars A B : Alphabet .

  eq (0).RegExp (E).RegExp = (0).RegExp .
  eq (E).RegExp + (E).RegExp = (E).RegExp .
  eq (0).RegExp & (0).RegExp = (0).RegExp .
  eq (0).RegExp & (1).RegExp = (0).RegExp .
  eq (1).RegExp & (1).RegExp = (1).RegExp .
  eq (0).RegExp + (1).RegExp = (1).RegExp .
  eq (1).RegExp + (0).RegExp = (1).RegExp .

  eq length(1) = 0 .
  eq length(0) = 0 .
  eq length(A E) = 1 + length(E) .
  eq length(A) = 1 .
  ceq length(E1 + E2) = length(E1) + length(E2) if E1 =/= 0 and E2 =/= 0 .
  eq length(E1 *) = length(E1) .
  ceq length(E1 * E2) = length(E1) + length(E2) if E1 =/= 0 and E2 =/= 0 .
endm

mod Linearize is
  protecting RegularExpression .
  protecting NAT .
  sorts Lin .
  subsort Lin < Alphabet .

  op [_,_] : Alphabet Nat -> Lin .
  op linearize : RegExp -> RegExp [memo] .
  op linearize : RegExp Nat -> RegExp [memo] .

  vars A : Alphabet .
  vars E1 E2 : RegExp .
  vars I : Nat .

  ceq linearize(E1) = linearize(E1,1) if E1 =/= 0 .
  eq linearize(A,I) = [A,I] .
  ceq linearize(E1 *,I) = linearize(E1,I) * if E1 =/= 0 .
```

```
53    ceq linearize(E1 E2,I) = linearize(E1,I) linearize(E2,I + length(E1))
54      if E1 =/= 0 and E2 =/= 0 .
55    ceq linearize(E1 + E2,I) = linearize(E1,I) + linearize(E2,I + length(E1))
56      if E1 =/= 0 and E2 =/= 0 .
57  endm
58
59  mod GenAut is
60    protecting RegularExpression .
61    protecting Linearize .
62    sorts Trans TransList .
63    subsort Trans < TransList .
64
65    op nill : -> TransList [ctor] .
66    op [_,_,_] : Nat Alphabet Nat -> Trans [prec 40].
67    op _|_ : TransList TransList -> TransList [assoc id: nill prec 45] .
68    op c : RegExp Nat -> RegExp .
69    op d : RegExp -> RegExp .
70    op last : RegExp Nat -> RegExp .
71    op follow : RegExp Nat -> RegExp [memo] .
72    op first : RegExp -> RegExp [memo] .
73    op nullable : RegExp -> RegExp [memo] .
74    op genaut : RegExp -> TransList .
75    op genaut : RegExp Nat -> TransList .
76    op gentrans : RegExp Nat -> TransList .
77
78    vars A : Alphabet .
79    vars E1 E2 : RegExp .
80    vars L1 : Lin .
81    vars I J : Nat .
82
83    eq 1 E1 = E1 .
84
85    eq c(0, I) = 0 .
86    eq c(1, I) = 0 .
87    ceq c([A,J], I) = 1 if J == I .
88    ceq c([A,J], I) = 0 if J =/= I .
89    ceq c(E1 + E2, I) = c(E1, I) if c(E1, I) =/= 0 .
90    eq c(E1 + E2, I) = c(E2, I) [owise] .
91    ceq c(E1 E2, I) = c(E1, I) E2 if c(E1, I) =/= 0 .
92    eq c(E1 E2, I) = c(E2, I) [owise] .
93    eq c(E1 *, I) = c(E1, I) E1 * .
94
95    eq d(0) = 0 .
96    eq d(1) = 0 .
97    eq d([A,J]) = [A,J] .
98    eq d(E1 + E2) = d(E1) + d(E2) .
99    eq d(E1 *) = d(E1) .
100   ceq d(E1 E2) = d(E1) + d(E2) if nullable(E1) == 1 .
101   eq d(E1 E2) = d(E1) [owise] .
102
103   eq follow(E1, I) = d(c(E1, I)) .
104   ceq last(E1, I) = 1 if I == 0 and nullable(E1) == 1 .
105   ceq last(E1, I) = 1 if c(E1, I) == 1 .
106   ceq last(E1, I) = 1 if nullable(c(E1, I)) == 1 .
107   eq last(E1, I) = 0 [owise] .
108   eq first(E1) = d(E1) .
109
```

```
110   eq nullable(1) = 0 .
111   eq nullable(0) = 0 .
112   eq nullable(A) = 0 .
113   eq nullable(E1 E2) = nullable(E1) & nullable(E2) .
114   eq nullable(E1 + E2) = nullable(E1) + nullable(E2) .
115   eq nullable(E1 *) = 1 .
116
117   eq genaut(E1) = genaut(E1, 0) .
118   eq genaut(E1, 0) = gentrans(first(E1), 0) | genaut(E1, 1) .
119   ceq genaut(E1, I) = gentrans(follow(E1, I), I) | genaut(E1, I + 1)
120     if I =/= length(E1) + 1 .
121   eq genaut(E1, I) = nill [owise] .
122
123   eq gentrans([A, J], I) = [I, A, J] .
124   eq gentrans([A, J] + E1, I) = [I, A, J] | gentrans(E1, I) .
125   eq gentrans(0, I) = nill .
126 endm
127
128 mod Compare is
129   protecting RegularExpression .
130   protecting Linearize .
131   protecting GenAut .
132   sorts CSList RelationList RelationEle .
133   subsort RelationEle < RelationList .
134   subsort Nat < CSList .
135
136   op compare : RegExp RegExp -> RegExp .
137   op compare : RegExp TransList RegExp TransList RelationList RelationList
138         -> RegExp .
139   op containsfinal : RegExp CSList -> RegExp .
140   op containsrel : RelationList RelationList -> RegExp .
141   op contcl : RelationList -> RelationList [memo] .
142   op stepa : Nat TransList -> CSList [memo] .
143   op stepb : Nat TransList -> CSList [memo] .
144   op nillr : -> RelationList [ctor] .
145   op (_,_) : CSList CSList -> RelationEle [comm assoc prec 50] .
146   op _/_ : RelationList RelationList -> RelationList [assoc id: nillr prec 60] .
147   op nills : -> CSList [ctor] .
148   op _$_ : CSList CSList -> CSList [comm assoc id: nills prec 49] .
149
150   vars A B : Alphabet .
151   vars E1 E2 : RegExp .
152   vars C1 C2 C3 C4 C5 C6 : CSList .
153   vars I J K : Nat .
154   vars T1 T2 : TransList .
155   vars R1 R2 R3 : RelationList .
156   vars RE1 RE2 : RelationEle .
157
158   eq RE1 / RE1 = RE1 .
159   eq RE1 / R1 / RE1 = RE1 / R1 .
160   eq I $ I = I .
161   eq I $ C1 $ I = I $ C1 .
162
163   ceq compare(E1, E2) = 0 if nullable(E1) =/= nullable(E2) .
164   eq compare(E1, E2) = compare(linearize(E1), genaut(linearize(E1)), linearize(E2),
165                 genaut(linearize(E2)), (0, 0), (nills, nills)) [owise] .
166   eq compare(E1, T1, E2, T2, nillr, R2) = 1 .
```

```
167   ceq compare(E1, T1, E2, T2, (C1, C2), R2) = 1
168     if containsrel((stepa(C1, T1), stepa(C2, T2)), R2) == 1 and
169     containsrel((stepb(C1, T1), stepb(C2, T2)), R2) == 1 .
170   ceq compare(E1, T1, E2, T2, (C1, C2) / R1, R2) = 1
171     if containsrel((stepa(C1, T1), stepa(C2, T2)), R2) == 1 and
172     containsrel((stepb(C1, T1), stepb(C2, T2)), R2) == 1 .
173   ceq compare(E1, T1, E2, T2, (C1, C2) / R1, R2) = 0
174     if (stepa(C1, T1) == nills and stepa(C2, T2) =/= nills) or
175     (stepa(C1, T1) =/= nills and stepa(C2, T2) == nills) or
176     (stepb(C1, T1) =/= nills and stepb(C2, T2) == nills) or
177     (stepb(C1, T1) =/= nills and stepb(C2, T2) == nills) .
178   ceq compare(E1, T1, E2, T2, (C1, C2) / R1, R2) =
179     compare(E1, T1, E2, T2, R1, contcl(R2 / (stepa(C1, T1),
180         stepa(C2, T2)) / (stepb(C1, T1), stepb(C2, T2))))
181       if containsfinal(E1, stepa(C1, T1)) == containsfinal(E2, stepa(C2, T2)) and
182       containsfinal(E1, stepb(C1, T1)) == containsfinal(E2, stepb(C2, T2)) and
183       (C1, C2) == (stepa(C1, T1), stepa(C2, T2)) and
184       (C1, C2) == (stepb(C1, T1), stepb(C2, T2)) .
185   ceq compare(E1, T1, E2, T2, (C1, C2) / R1, R2) =
186     compare(E1, T1, E2, T2, R1 / (stepa(C1, T1), stepa(C2, T2)) / (stepb(C1, T1),
187       stepb(C2, T2)), contcl(R2 / (stepa(C1, T1),
188       stepa(C2, T2)) / (stepb(C1, T1), stepb(C2, T2))))
189       if containsfinal(E1, stepa(C1, T1)) == containsfinal(E2, stepa(C2, T2)) and
190       containsfinal(E1, stepb(C1, T1)) == containsfinal(E2, stepb(C2, T2)) .
191   eq compare(E1, T1, E2, T2, (C1, C2) / R1, R2) = 0 [owise] .
192
193   eq stepa(nills, T1) = nills .
194   ceq stepa(I $ C1, T1) = stepa(I, T1) $ stepa(C1, T1) if C1 =/= nills .
195   eq stepa(I, [J, b, K] | T1) = stepa(I, T1) .
196   ceq stepa(I, [J, a, K] | T1) = K $ stepa(I, T1) if I == J .
197   eq stepa(I, [J, a, K] | T1) = stepa(I, T1) [owise] .
198   ceq stepa(I, [J, a, K]) = K if I == J .
199   eq stepa(I, [J, a, K]) = nills [owise] .
200   eq stepa(I, nill) = nills .
201
202   eq stepb(nills, T1) = nills .
203   ceq stepb(I $ C1, T1) = stepb(I, T1) $ stepb(C1, T1) if C1 =/= nills .
204   eq stepb(I, [J, a, K] | T1) = stepb(I, T1) .
205   ceq stepb(I, [J, b, K] | T1) = K $ stepb(I, T1) if I == J .
206   eq stepb(I, [J, b, K] | T1) = stepb(I, T1) [owise] .
207   ceq stepb(I, [J, b, K]) = K if I == J .
208   eq stepb(I, [J, b, K]) = nills [owise] .
209   eq stepb(I, nill) = nills .
210
211   eq containsfinal(E1, nills) = 0 .
212   ceq containsfinal(E1, I $ C1) = 1 if last(E1, I) == 1 .
213   eq containsfinal(E1, I $ C1) = containsfinal(E1, C1) [owise] .
214   eq containsfinal(E1, I) = last(E1, I) .
215
216   eq containsrel(RE1, nillr) = 0 .
217   ceq containsrel(RE1, RE2 / R1) = 1 if RE1 == RE2 .
218   eq containsrel(RE1, RE2 / R1) = containsrel(RE1, R1) [owise] .
219
220   ceq contcl((C1, C2) / (C3, C4) / R1) = contcl((C1, C2) / (C3, C4) /
221                           (C1 $ C3, C2 $ C4) / R1)
222     if containsrel((C1 $ C3, C2 $ C4), (C1, C2) / (C3, C4) / R1) == 0 .
223   eq contcl((C1, C2) / (C3, C4) / R1) = contcl((C1, C2) / R1) / contcl((C3, C4) / R1) .
```

```
224     eq contcl((C1, C2)) = (C1, C2) .
225
226  endm
```

References

[1] Filippo Bonchi, Damien Pous, Checking NFA equivalence with bisimulations up to congruence, Principle of Programming Language, 2013

[2] John C. Martin, Introduction to Languages and the Theory of Computation, Fourth edition, McGraw-Hill, 2011

[3] J.-M. Champarnaud, D. Ziadi, Canonical derivatives, partial derivatives and finite automaton constructions, Elsevier, 2002

[4] Maude, http://maude.cs.uiuc.edu/

[5] Theodore McCombs, Maude 2.0 Primer, August 2003