University of Dublin

Trinity College

Derivatives of Extended Regular Expressions

Ciarán Regan
B.A. (Mod.) Computer Science, Linguistics and a Language
Final Year Project, April 2015
Supervisor: Dr. Tim Fernando

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_____________________________ April 7, 2015

Ciarán Regan
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

.................................................. April 7, 2015

Ciarán Regan
Acknowledgements

First and foremost I would like to thank Tim Fernando for his much appreciated help throughout the past year.

Thank you to Carl Vogel and Martin Emms for their work as course directors and mentors over the last four years.

I would like to express my gratitude to my family and friends for their combined support that kept me calm during times of stress.

Finally, a special thanks to all my classmates for an unforgettable college experience.
Abstract

The idea of regular expression derivatives is one that has been overlooked in the world of computer science. Their efficiency in finite state machine creation highlights their importance as well as the fact that there remain several areas in which they have not yet been implemented. The purpose of this paper is to first describe two methods for computing derivatives of extended regular expressions and then create two toolkits that operate based on these two methods. The second part of the paper covers some of the possible applications of these toolkits, mainly focusing on areas where derivatives of extended regular expressions have not yet been applied.
Contents

1 Introduction 7
  1.1 Project Overview .......................... 7
  1.2 Preliminaries ............................. 7

2 Regular Expression Derivatives 9
  2.1 Definition ................................ 9
  2.2 Rule-Based Calculation of Derivatives of Extended Regular
      Expressions .................................. 9
  2.3 Computing Derivatives of Extended Regular Expressions Using
      FSAs ......................................... 11

3 Implementing a Derivative Calculator 14
  3.1 DFA Manipulation Method .................. 14
    3.1.1 Tools .................................. 14
    3.1.2 Implementation .......................... 14
    3.1.3 Challenges and Results............... 19
  3.2 Rule-Based Method ......................... 22
    3.2.1 Tools .................................. 22
    3.2.2 Implementation .......................... 22
    3.2.3 Challenges and Results............... 25
  3.3 Comparison ................................ 26

4 Applications of Derivatives 28
  4.1 DFA Creation ................................ 28
    4.1.1 States as Languages ................... 30
  4.2 Derivatives applied to MSO ................. 34
    4.2.1 Background ............................. 34
    4.2.2 Derivatives and MSO ................... 36
  4.3 Derivatives of Extended Regular Expressions and Frames .. 38
    4.3.1 Background ............................. 38
    4.3.2 Derivatives Applied to Frames ....... 43

5 Conclusion 45
  5.1 Achievements .............................. 45
  5.2 Future Work ............................... 46

6 References 47
# Appendix

## 7.1 DFA Manipulation System

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Source File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.1</td>
<td>Calculator.cpp</td>
<td>48</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Generator.cpp</td>
<td>48</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Converter.cpp</td>
<td>48</td>
</tr>
<tr>
<td>7.1.4</td>
<td>TransitionTuple.cpp</td>
<td>49</td>
</tr>
</tbody>
</table>

## 7.2 Simplifier

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Source File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.1</td>
<td>MainSimplifier.cpp</td>
<td>49</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Simplifier.cpp</td>
<td>49</td>
</tr>
</tbody>
</table>

## 7.3 Rule-Based System

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Source File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.1</td>
<td>DerivativeCalculator.cpp</td>
<td>50</td>
</tr>
<tr>
<td>7.3.2</td>
<td>RegularExpression.cpp</td>
<td>50</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Project Overview

The main objective of this project is to develop a toolkit that can compute derivatives of extended regular expressions, first introduced by Brzozowski (1964), and provide the reader with an idea of the applications of such derivatives. A number of topics will be covered such as a background on regular expression derivatives as described by Brzozowski (1964) and Owens et al. (2009). I will then present two different methods for computing regular expression derivatives and how they were implemented, the difficulties associated with the development of each system and how they compare overall. Finally I will discuss the advantages and uses of such a system starting with the possibility of using derivatives of extended regular expressions for deterministic finite state automata (DFA) construction. This will be followed by their application to monadic second order logic and using derivatives to calculate preimages of functions before finishing with a description of frame theory (Barsalou 1992, Fillmore 1982) and how derivatives of extended regular expressions can be applied to frames.

1.2 Preliminaries

This section defines the notation that will be used through this report. The notation differs slightly from standard notation in order to keep the report consistent with symbols used by the Xerox Finite State Toolkit (xfst), which will be discussed later in section 3.1.1. \( \Sigma \) denotes a finite alphabet over a language \( L \), and \( \Sigma \) represents the set of all finite strings over \( \Sigma \). The characters \( a, \ldots, z \) correspond to symbols, and \( \epsilon \) to the empty string. Note that in xfst the symbol 0 denotes the empty string, however I will be using the standard \( \epsilon \) to avoid confusion with the empty set \( \emptyset \).

Given the regulars expressions \( r \) and \( s \), we have the following regular expression operators.
∅  empty set
ε  empty string
a  a ∈ Σ
r · s  concatenation
r*  Kleene-closure
r|s  alternation
r&s  logical and
¬r  complement

Owens et al. (2009) refer to these expressions extended regular expressions, since their expressivity has been enhanced with boolean operations. To illustrate this I have included an example where we compare the two regular expressions shown below. Both strings represent the language in which all strings start and end with the sequences */ and */ respectively, and all strings in between these two sequences do not contain the end sequence */ (much like Java and C++ style comments). Regular expression (1) has been extended with the boolean operator ¬. We can see how the expressivity of (1) is reduced when we try to construct an equivalent regular expression without such an operator. The result is the more cumbersome expression given by (2).

\[
/\ast¬(\Sigma^\ast / /\Sigma^\ast\ast/) \\
/\ast((\Sigma\{\ast\})^\ast(\epsilon | \ast\ast (\Sigma\{/\ast\})\ast)\ast/) 
\]

Expression (2) is much more complicated. The first inner expression \((\Sigma\{\ast\})^\ast\) denotes the string that contains any number of characters from our alphabet, but not the character \*. The second inner expression \((\epsilon | \ast\ast (\Sigma\{/\ast\})\ast)\ast\) denotes the string that contains either the empty string or any number of \* characters (given by \ast\ast) so long as this is followed by any set of characters in our alphabet except for the characters / and \* (given by \Sigma\{/\ast\})
2 Regular Expression Derivatives

2.1 Definition
In this section I will cover the background of regular expression derivatives and how they are computed. The idea of derivatives of regular expressions was first popularised by Brzozowski (1964) when he found that they could be used in a more elegant method for converting regular expression descriptions into finite state diagrams. He defines the derivative of a set of strings $S$ with respect to a symbol $a$ as the set of strings that are generated by removing the leading $a$ from the set of strings in $S$ that start with the symbol $a$. This can be expressed with the formula

$$\partial_a L = \{s | as \in L\}$$

As a simple example take the regular expression $r = ab|ac$, where $r$ defines a language $L$ which recognises the strings $\Sigma^* = \{ab, ac\}$. If we wish to compute the derivative of $L$ with respect to $a$, then according to our definition this would involve removing the leading $a$ from the set of strings in $L$. This is illustrated below.

$L = \{ab, ac\}$

$$\partial_a L = \{ab, ac\} = \{b, c\}$$

In the above example the regular expression $\partial_a r$ that represents the language $\partial_a L$ is easy to find by simply looking at its set of strings $\{b, c\}$. We can have a string consisting of either a single $b$ or a single $c$, so our regular expression must be $\partial_a r = b|c$. With more complex regular expressions it becomes too difficult to compute derivatives by simply observing the set of strings that they produce. In this report I discuss two ways in which this can be achieved. The first is discussed in the next section and relies on a set of calculus rules that can be applied to regular expressions in order to compute their derivatives. The second involves the manipulation of an already constructed finite state automaton (FSA) and is discussed in section 3.1.

2.2 Rule-Based Calculation of Derivatives of Extended Regular Expressions
In the previous section we discussed how the derivative of $L$ with respect to $a$ is obtained by removing the leading $a$ from the set of strings in $L$. To
compute the derivative of a regular expression \( r \) that represents the set of strings accepted by the language \( L \), we can use the following set of rules presented by Owens et al. (2009).

\[
\begin{align*}
\partial_a \epsilon &= \emptyset \\
\partial_a a &= \epsilon \\
\partial_a b &= \emptyset \quad \text{for } b \neq a \\
\partial_a \emptyset &= \emptyset \\
\partial_a (r \cdot s) &= \partial_a (r) \cdot s \cup \nu(r) \cdot \partial_a s \\
\partial_a (r^*) &= \partial_a r \cdot r^* \\
\partial_a (r|s) &= \partial_a r \cup \partial_a s \\
\partial_a (r \& s) &= \partial_a r \& \partial_a s \\
\partial_a (\neg r) &= \neg (\partial_a r)
\end{align*}
\]

Notice the helper function \( \nu(r) \) function in rule (5). The regular expression \( r \) is nullable if the language it defines contains the empty string. If \( r \) is nullable then the function \( \nu(r) \) returns \( \epsilon \). Otherwise, it returns \( \emptyset \).

To demonstrate how these rules work let’s look at an example by computing the derivative of the regular expression \( r = (a^*bc)^* \) with respect to \( a \). An idea of the set of strings generated by \( r \) is given below.

\[
\Sigma^* = \{ \epsilon, \\
bc, bc, bc, bc, bc, \ldots, \\
abc, abc, abc, \ldots, \\
aabc, aabc, aabc, aabc, \ldots \}
\]

Taking our regular expression \( (a^*bc)^* \), we see that we must first employ rule number (6) that deals with Kleene-closure. Doing this gives us the following.

\[
\partial_a (a^*bc)^* = \partial_a (a^*bc) \cdot (a^*bc)^*
\]

According to the result of the above equation, we must now compute the derivative of the regular expression \( a^*bc \) with respect to \( a \) and concatenate this with the expression \( (a^*bc)^* \). Computing \( \partial_a (a^*bc) \) proves to be rather simple. We no longer have a Kleene-star operator (or any other operator)
that ranges over the entire expression so we compute the derivative of the string $a^*bc$, focusing on the starting symbol $a^*$.

$$\partial_a a^* = \partial_a a \cdot a^*$$

$$= \epsilon \cdot a^*$$

$$= a^*$$

So in fact, $\partial_a (a^*bc) = a^*bc$. Concatenating this with $(a^*bc)^*$ gives us the expression $(a^*bc)(a^*bc)^*$, which we can simplify to $(a^*bc)^+$. So, by computing the derivative of $r = (a^*bc)^*$ we get $\partial_a r = (a^*bc)^+$. Note that while it is not specifically employed, the result of rule (1) given above can be seen by comparing the set of strings generated by $r$: \{\epsilon, a \ldots bc, a \ldots bca \ldots, \ldots\}$, with the set of strings generated by $\partial_a r$: \{a \ldots bc, a \ldots bca \ldots, \ldots\}. Notice how the set for $r$ contains $\epsilon$ while the set for $\partial_a r$ does not, as per rule (1).

Now that we have seen these rules and how they work we can go about implementing a system that can compute the derivative of a regular expression using such rules. I describe this system in section 3.2.

### 2.3 Computing Derivatives of Extended Regular Expressions Using FSAs

As well as rule-based calculation of regular expression derivatives discussed in section 2.2, there exists another method that we can use. This method involves manipulating finite state automata and will be covered in this section.

In simple terms, we can obtain the derivative of a regular expression by first representing it as a DFA and then changing the initial state of this DFA depending on the symbol with respect to which we are computing the derivative. Take the earlier example from section 2.2 in which we computed $\partial_a ((a^*bc)^*)$. Converting this regular expression into a DFA we obtain the network shown figure 1.
Let’s say we want to compute $\partial_a((a^*bc)^*)$ using the method of changing initial state. What we must first do is check that there is an arc labelled $a$ leading from the current initial state, which in this case is $fs\emptyset$, to another state in the automaton. For the DFA in figure 1 there exists such a transition from $fs\emptyset$ to $s1$. This state becomes the new initial state in our network resulting in the DFA shown in figure 2. This DFA is equivalent to that which would result from converting the regular expression $(a^*bc)^+$ into a DFA (recall that this is the same expression that we got when computing $\partial_a(a^*bc)^*$ using calculus rules). Testing equivalence between two regular expressions is discussed further in section 3.1.2 but for now let’s assume that $(a^*bc)^+$ and $\partial_a(a^*bc)^*$ are equivalent.

So far we have taken the regular expression $r = (a^*bc)^*$ and converted it into the DFA in figure 1. We then changed the initial state of this DFA to
give us the DFA in figure 2 that represents $\partial_a r$. If we want to then generate a regular expression representing $\partial_a r$ we must then convert from the DFA in figure 2 back into a regular expression. With small DFAs such as that in figure 2 this is rather simple to do if we just examine the DFA. Starting at the initial state we notice that $s_1$ has a transition to itself labelled $a$, giving us the symbol $a^*$. As $s_1$ is non-final we must eventually leave this state if a string is to be accepted. We can only move on to $s_2$ via the transition labelled $b$, then from $s_2$ to $f s \emptyset$ along the arc labelled $c$. These two steps now leave us with the expression $a^*bc$. We can now either end at $f s \emptyset$ as it is final, or we can go back to $s_1$ again. This implies that we must have at least one occurrence of $a^*bc$ followed by many repetitions, giving us the expression $(a^*bc)^+$. Notice how this is the same regular expression derivative that we calculated in section 2.2.

The step-by-step method I just described works well for small and simple DFAs. However, when we are dealing with larger and more complicated regular expressions, or indeed if we want to create a system that can carry out such a task then this method is no longer sufficient. Instead we need a more sophisticated algorithm. This algorithm is explained in section 3.1 where I discuss how I implemented a DFA based derivative calculator.

In summary, given a regular expression $r$ we want to compute its derivative with respect to $a$, $\partial_a r$. First we convert $r$ into a DFA. We then look for a transition labelled $a$ from this DFA’s initial state to another state in the network. This state becomes the new initial state so that the DFA now represents the language generated by $\partial_a r$. If we then want to generate a regular expression for $\partial_a r$ we must convert this DFA back into a regular expression.

At this point we have now seen two methods that we can use for computing the derivative of an extended regular expression. The first method involves the use of a set of calculus rules that we can apply to extended regular expressions. The second is a method that involves representing a regular expression as a DFA, manipulating this DFA to represent $\partial_a r$ by changing its initial state and then converting this DFA back into a regular expression.

The next decision to be made was which method to implement in my system. The rule-based method of derivative calculation seems to be focused on much more in most literature that covers regular expression derivatives and is more defined. The DFA manipulation method is less covered but in theory seems to be much more simple to implement. I decided to start with the DFA manipulation method as there are many advanced finite state toolkits available that I could use for the DFA generation aspect and an algorithm
for DFA to regular expression conversion provided by Savage (1997). The implementation of this system will be covered in section 3.1 and will include a section on the difficulties encountered during its development. After experiencing these problems I decided that it would be worthwhile to develop a second derivative calculator modelled on the rule-based evaluation of regular expression derivatives. The systems could be compared afterwards and evaluated in terms of their advantages and disadvantages over one another.

3 Implementing a Derivative Calculator

3.1 DFA Manipulation Method

3.1.1 Tools

The task required the use of a toolkit for generating finite state automata that would allow freedom to manipulate the automata after they had been generated. After researching several options it was found that the majority of finite state toolkits are specifically designed for modelling morphological aspects of language and as such, they can be limited in what they allow the user manipulate once an FSA has been created. Changing the initial state of a network was a feature that all toolkits seemed to omit. This problem was solved by slightly modifying the DFA to regular expression algorithm and will be discussed further in the next section. I would still need a toolkit that could generate a DFA and so I chose to use the Xerox finite state toolkit (xfst). My reasons for this being that xfst has been used by many researchers in the past and by default it creates deterministic minimal automata. In addition to this it has a built in tool which allows us to test if two FSAs are equivalent. This becomes important later on and is explained in section 3.1.2. It does not allow the user to change the initial state of the networks it creates, however this problem was solved by altering the method discussed in the next section for converting a DFA back into a regular expression.

The programming language I used for this project was C++. Having experience with object-oriented concepts it was narrowed down to a choice between this and Java, with the deciding factor being that C++ provides a simpler way of interacting with external programmes and reading output files from these programmes.

3.1.2 Implementation

In this section I explain how I developed a system for computing the derivative of a regular expression using the method of DFA manipulation. We can
break it down into the following tasks.

1. Read in a regular expression and input this expression to xfst to create a DFA.

2. Change the initial state of this DFA (as stated in section 3.1.1 I had to solve this problem by slightly altering the algorithm used to complete step (3))

3. Convert the DFA back into a regular expression for $\partial r$

Step (1) was straight forward, we simply have to read in a regular expression then write the appropriate xfst commands to a script file which xfst can then use to construct a DFA that represents this expression. I will now explain the algorithm used to complete step (3) and then explain how it was modified to complete step (2).

This is the algorithm for converting a DFA into a regular expression. First we take a DFA $A$ representing the regular expression $r$. $Q$ is a set containing all states $q_1 \ldots q_n$ in $A$ and the set $F$ contains all of its final states. Then, for every pair of states $(q_i, q_j)$ in $A$ we have a regular expression $r_{i,j}^{(k)}$ that takes $A$ from $q_i$ to $q_j$. For our first table $k = 0$, where $k$ stands for the length of the strings that take $A$ from $q_i$ to $q_j$. For $k > 0$ we must then find the regular expression $r_{i,j}^{(k)}$ denoting a language whose strings are of maximum length $k$, and will take $A$ from $q_i$ to $q_j$. For each state of state transition $(q_i, q_j)$, $r_{i,j}^{(k)}$ can be determined with the following formula.

$$r_{i,j}^{(k)} = r_{i,j}^{(k-1)} + r_{i,k}^{(k-1)} (r_{k,k}^{(k-1)})^* r_{k,j}^{(k-1)}$$
From the DFA in figure 3 we construct the first table $T^{(0)}$ shown below

\[
T^{(0)}
\begin{array}{c|cccc}
   i \backslash j & 1 & 2 & 3 & 4 \\
\hline
   1 & \epsilon & a & \emptyset & \emptyset \\
   2 & \emptyset & \epsilon & b & \emptyset \\
   3 & \emptyset & \emptyset & \epsilon & a \\
   4 & \emptyset & a & b & \epsilon \\
\end{array}
\]

From the formula given on the previous page we can see that the formula for filling in the second table $T^{(1)}$ is $r_{i,j}^{(1)} = r_{i,j}^{(0)} + r_{i,1}^{(0)}(r_{1,1}^{(0)})^*r_{1,j}^{(0)}$. It happens to be that $T^{(1)}$ is the same as $T^{(0)}$

\[
T^{(1)}
\begin{array}{c|cccc}
   i \backslash j & 1 & 2 & 3 & 4 \\
\hline
   1 & \epsilon & a & \emptyset & \emptyset \\
   2 & \emptyset & \epsilon & b & \emptyset \\
   3 & \emptyset & \emptyset & \epsilon & a \\
   4 & \emptyset & a & b & \epsilon \\
\end{array}
\]

For $T^{(2)}$ we get the formula $r_{i,j}^{(2)} = r_{i,j}^{(1)} + r_{i,2}^{(1)}(r_{2,2}^{(1)})^*r_{2,j}^{(1)}$

\[
T^{(2)}
\begin{array}{c|cccc}
   i \backslash j & 1 & 2 & 3 & 4 \\
\hline
   1 & \epsilon & a & ab & \emptyset \\
   2 & \emptyset & \epsilon & b & \emptyset \\
   3 & \emptyset & \emptyset & \epsilon & a \\
   4 & \emptyset & a & b + ab & \epsilon \\
\end{array}
\]
The formula we use to fill the fourth table $T^{(3)}$ is $r_{i,j}^{(3)} = r_{i,j}^{(2)} + r_{i,3}^{(2)} (r_{3,3}^{(2)}) r_{3,j}^{(2)}$

<table>
<thead>
<tr>
<th>$T^{(3)}$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i \ j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$\epsilon$</td>
<td>a</td>
<td>ab</td>
<td>aba</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>$\epsilon$</td>
<td>b</td>
<td>ba</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\epsilon$</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>a</td>
<td>b + ab</td>
<td>(b + ab)a</td>
</tr>
</tbody>
</table>

For the fifth and final table $T^{(4)}$ we have the formula $r_{i,j}^{(4)} = r_{i,j}^{(3)} + r_{i,4}^{(3)} (r_{4,4}^{(3)}) r_{4,j}^{(3)}$. Due to the lengths of the regular expressions we now have I have only entered the values for $r_{1,3}^{(4)}$ and $r_{1,4}^{(4)}$. We know that this is the final table because if we were to create another table we would have $k = 5$, which is not possible as we only have four states.

<table>
<thead>
<tr>
<th>$T^{(4)}$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i \ j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
<td>ab+aba((b+ab)a)*(b+ab)</td>
<td>ab+aba((b+ab)a)*</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Now that we have completed the final table we can determine the regular expression for this DFA. We take the regular expression from the all cells that represent string which takes us from the initial state in the network to the final state. In this network the only initial state is 1 and we have two final states, 3 and 4. This leaves us with the regular expression $r_{1,3}^4 \mid r_{1,4}^4$. Taking the values from the final table that correspond to the regular expressions $r_{1,3}^4 \mid r_{1,4}^4$ we get:

$$
\begin{align*}
(rababa((b|ab)a)^*(b|ab)) & \mid (aba|aba((b|ab)a)^+) \\
\end{align*}
$$

This final step is where I changed to algorithm to accommodate for the fact that xfst does not allow the user to change the initial state of a network. When constructing our final regular expression we took 1 as our initial state and 3 and 4 as our final states giving us the formula $r_{1,3}^4 r_{1,4}^4$. If instead
we treat 2 as our initial state we get the formula $r_{2,3}^4 | r_{2,4}^4$. This gives us a regular expression that takes 2 as the initial state of the network, and therefore results in the same regular expression we would get if we changed the initial state of the network from 1 to 2.

When this change in the algorithm was made it was not taken for granted that altering the algorithm would always produce the same result as if the structure of the DFA itself was changed. There is a tool in xfst that allows us to compare the languages defined by two DFA and determine whether or not they are equivalent. I started by passing the regular expression obtained from the altered algorithm to xfst. I then made a finite state diagram, setting the initial state equal to state 2 instead of state 1 and worked out its regular expression by hand. This regular expression was then passed into xfst and then the two DFAs were compared using xfst’s comparison tool. If the DFAs were equivalent then the altered algorithm above could be considered an appropriate alternative to changing the initial state of the network. This proved to be the case as I tested for multiple DFAs with aim being to test for different combinations of each of the regular expression operators.

There is one more observation that I will mention. If our initial state is $q_1$ and we want to change our initial state to $q_2$, this will sometimes cause $q_1$ to become inaccessible as there is no arc leading back to it. See figure 4 for an example of this. As $q_1$ becomes redundant we can simply delete it from the network. This is done automatically by xfst through a process called pruning in which all states along a path that does not lead to an accepting state are removed.

**Figure 4:** DFA representing the regular expression $\partial_a(abc)$, or simply $bc$
In figure 4, \( q_1 \) is no longer accessible so we do not need it in our network. We can simply represent it as the DFA in figure 5.

Figure 5: DFA representing the regular expression \( \partial_a(abc) \), or simply \( bc \)

There are also other situations where changing \( q_2 \) does not cause \( q_1 \) to become inaccessible such as in figure 6. In this case we do not need to prune the network as we may still need to access \( q_1 \).

Figure 6: DFA representing the regular expression \( \partial_a(a^*bc)^* \)

3.1.3 Challenges and Results

The first major challenge was finding a toolkit that would allow the initial state of a network to be changed. After researching several different options I found no such toolkit available. There are some open source finite state toolkits such as hfst that attempt to mimic xfst, however these seem to be more specified towards particular areas of morphology such as Finnish morphology in the case of hfst. As explained in section 3.1.2, I dealt with this problem by altering the algorithm for converting a DFA back into a regular expression.

Another challenge involved the output produced by this algorithm. The output was cumbersome, containing many unnecessary brackets and concatenation of epsilon symbols. There would also be cases in which epsilon
characters would have a Kleene-closure operator, e.g \( \epsilon^* \) or often \((\epsilon|abc)^*\) which can be simplified to \((abc)^*\). There were often cases where there would be a disjunction operator between equivalent expressions such as \((b|b)\) which should be simplified to just \(b\). I will give an example below of putting the regular expression \((abcd)^*\) through the derivative calculator to illustrate some of the problems with the output. Note that 0 is the xfst notation for \(\epsilon\).

\[
\partial_a (abcd^*) = ((b(0a^*b)0^*c)(0)((d0^*a)0^*b)0^*c)^*(d|d0^*0)
\]

It would obviously be worthwhile simplifying the above output because as it stands it is rather illegible. To deal with this problem I created a simplifier, which would take the output of the derivative calculator and attempt to simplify this expression. The tasks were as follows:

1. Remove unnecessary brackets, e.g \((bc)d\) simplifies to \(bcd\).
2. Remove epsilon concatenations, e.g \(\epsilon b\) simplifies to \(b\).
3. Remove equivalent expressions separated by a disjunction, e.g \((b|b)\) to \(b\).

To complete the first step we first find the index of the first closing bracket. We then backtrack to find the index of its opening bracket. If there are any disjunction operators within the scope of these brackets, or if the is a Kleene star over these brackets, e.g \((...)^*\), then we keep the brackets as they are. Otherwise we can remove them.

To complete the second step we search for any epsilon characters. If we find an epsilon and the next character is a kleene star, an alphabet symbol or another epsilon then we delete this character.

The third step was more complex to complete as we need to find the scope of the | operator. For small segments of regular expressions such as \((b|00^*b)\) this is rather simple. To find the starting scope to the left hand side of the disjunction symbol we must find the first opening bracket character. This bracket’s closing bracket will mark the end of the scope on the right hand side. We make the LHS and RHS into substrings and pass them as input to xfst. We then utilise xfst’s equivalence test. If the expressions are equivalent we delete the longer one, if they are not equivalent then we leave the expression as it is. For more complex expressions where we have more than one final state we can end up with many disjunction characters. It becomes increasingly difficult to find their scope, but this was a problem that I was able to deal with when implementing the rule-based calculator and will
be discussed further in section 3.2.2. The three steps I just explained were repeated until the regular expression being passed through the simplifier as input was the same as the output regular expression.

I managed to get the simplifier to a point where it could produce the following output for the expression for $\partial(abcd)^*$ on the previous page.

$$bc(dabc)*d$$

This is the equivalent of the expression $bcd(abcd)^*$ which we would get if we used the calculus rules for computing derivatives and can again be tested using xfst’s equivalence testing tool. This is the most simplified form we can get without just rearranging symbols to possibly make the expression more legible.

One of the major problems I encountered with this algorithm was that for certain regular expressions it will compute a slightly incorrect derivative. The structure will be similar with many of the states and transitions remaining the same, however sometimes the final states would be incorrect. It was difficult to test by hand as this was an issue for more complex regular expressions, such as $(abcd)^*(aef)^*$, as these take a long time to work out. Even after altering the last step of the DFA to regular expression algorithm the final states were still incorrect. If a toolkit were modified to incorporate the changing of the initial of an FSA then this problem could possibly be avoided.
3.2 Rule-Based Method

3.2.1 Tools

This task did not initially require the use of a finite state toolkit, as I would only be using the rule-based method in the system. I did however make use of xfst when comparing both systems. After entering the same regular expression into both programmes I could test whether the regular expression derivatives they produced as output were equivalent using xfst’s equivalence test command.

Again, I used C++ for this system for similar reasons to the other implementation. Experience with object-oriented programming and being able to easily handle output from external programmes were the main factors I took into account.

3.2.2 Implementation

There were two main challenges involved in this developing this system. The first was to find the scope of all of the operators in the regular expression and check if a possible starting point was within the scope of each operator. Consider the following example below. The possible starting points in this expression are underlined.

\[ ab \mid gd(c\mid f)gh \mid ij \]

At first I believed that the easiest way to solve this problem would be to use the shunting yard algorithm for evaluating infix expressions. However, this proved to be problematic. For this algorithm to work we assign each operator an order of precedence. The orders of precedence for regular expression operators from highest to lowest are:

<table>
<thead>
<tr>
<th>\</th>
<th>Escape</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Parentheses</td>
</tr>
<tr>
<td>*,+,?,{n...}</td>
<td>Quantifiers</td>
</tr>
<tr>
<td>^, $ , concatenation</td>
<td>Anchors and Sequences</td>
</tr>
<tr>
<td></td>
<td>Alternation/Disjunction</td>
</tr>
</tbody>
</table>

When implementing this algorithm I found that there was an issue with the concatenation operator. Concatenation is not explicitly stated in regular expressions but is applied when characters occur in sequence. This reduced the efficiency of this algorithm as it made it difficult to determine which characters on the stack should be concatenated and which should not
as there was no operator on the stack to indicate this. As a result the this
problem I was forced to use another algorithm for the findStartPoints func-
tion. I have outlined what this algorithm does below. The regular expression
is passed to the findStartingPoints function as a parameter and undergoes the
following steps:

1. The programme iterates through the string until a | character is found.

2. We look at the character char that occurs directly after the last dis-
junction we came across. (If no disjunction character has already been
found we look at the first character in the string)

3. If char is an opening bracket, we determine whether this bracket en-
closes characters under the Kleene-star operator. If this is the case
then we could have multiple starting points in this substring, e.g
(ab)*ef|gh. We create two substrings, the first of which contains ev-
erything within the brackets, i.e. ab, the second starts at the first
character after these brackets and ends at the | character that we have
just found, i.e. ef. The findStartingPoints function is recursively
called with these substrings being passed as parameters.

If this bracket does not mark the scope of a Kleene-star operator
then we only create a single substring containing everything within
the brackets and recursively call the findStartingPoints function. E.g
Consider the cases [ab|cd]ef and [gh]+ij. For the first example ab|cd
will be passed as a parameter and for the second we pass gh.

If char is an alphabet character, we create a substring starting at
this character and ending at the last disjunction and recursively call
the findStartingPoints function. E.g abc|def. In this case abc will be
passed to findStartingPoints as a parameter.

4. We continue through the string and repeat steps 2 and 3 at each dis-
junction we find. However, recall our example from earlier where we
had gb | gd(e|f)gh | ij. The programme will ignore the | at (e|f) as
it is within the wrong scope. This is done by keeping track of the
amount of opening brackets vs. closing brackets that we have seen.

5. This process is repeated until we reach the end of the string. If there
were no disjunction characters in the regular expression and the first
character in the string is an alphabet character, then this character
will be marked as a starting point. Otherwise we do the same as in
step 2.
This algorithm would be easier to understand with the help of an example. Consider the regular expression \( r = (ab)^* cd|(qr|st)yz \)

Iterate through \( r \) until the underlined \(|\) is found

\((ab)^* cd|(qr|st)yz\) \hspace{1cm} (1)

Starting character is an opening bracket marking Kleene-closure so \( \text{findStartingPoints} \) is called recursively with \( ab \) and \( cd \)

Iterate through \( ab \) until end of string is reached

(3)

Start of string is an alphabet character so \( a \) is marked as a starting point

(4)

Iterate through \( cd \) until end of string is reached

(5)

Start of string is an alphabet character so \( c \) is marked as a starting point

(6)

Continue iterating through original expression

\((ab)^* cd|(qr|st)yz\) \hspace{1cm} (7)

Skip the \(|\) character at \( qr|st \), as it is located within brackets

(8)

When the end of the string is reached, check the first character after the last disjunction we came across.

Character after last disjunction is a bracket but does not mark the scope of Kleene-star, so only \( qr|st \) is passed to \( \text{findStartingPoints} \)

(10)

Iterate through \( qr|st \) until \(|\) is reached

(11)

First character is an alphabet symbol so \( q \) is marked as a starting point

(12)
Continue to iterate through \( qr|st \) until end of string is reached \( (13) \)

First character after last disjunction is the alphabet symbol \( s \), so \( s \) is marked as a starting point \( (14) \)

All starting points have now been marked. \( (15) \)

\[(ab)^* cd | (qr|st)yz\] \( (16) \)

Now that we have a way of finding all possible starting points in our regular expression we can begin the next task which involves computing the derivatives of these expressions. This is implemented in a similar way to the algorithm for finding the starting points of the regular expression. We iterate through the expression until we reach a disjunction character. We then recurse left through the expression until we reach a \(*\), \(+\), or \(^\)\(\) operator. If this operator has scope over a starting point then we apply its respective calculus rule. Recall that for operators such as \(*\) we have the rule \( \partial_a r = \partial_a r \cdot r \). This means that we must create a substring containing the characters over which the \(*\) operator scope recursively call the derive method with this substring as a parameter. E.g. for the expression \((abc)^*\) we create the substring \( abc \) and pass it through the derive method. We then concatenate the result with \((abc)^*\) to get \( bc(abc)^*\).

If a starting point is outside the scope of any of these operators we simply remove it from the string. So a regular expression such as \( abc \) becomes \( bc \). See section 7.3.2 in the appendix for more information on this algorithm.

### 3.2.3 Challenges and Results

The main challenge in implementing this system was finding which points in the regular expression mark the starting point in a string. The fact that concatenation does not have an explicit operator made it difficult to implement the shunting yard algorithm which I believe would have been the most practical way of completing such a task. As this system is more useful as a toolkit for computing derivatives it would be cumbersome for the user to enter a regular expression including a special character for concatenation, as this is not usually done. For future implementations it could be worthwhile employing a formatting method that would read in a regular expression in
standard notation and add concatenation characters where necessary. However, when dealing with a greater number of larger regular expressions the process of formatting each individual expression could reduce the efficiency of such a system.

Another problem that was encountered during testing involved the ^ operator, as there is no explicit rule given in Owens et al. (2009) for this. We can, however use the method of DFA manipulation to investigate how the derivative of such an expression should look. Take the expressions (a|b)^3 and ((ab)*cd)^3.

Computing ∂d(a|b)^3 using the method of changing the initial state of a DFA gives us the regular expression (a|b)^2. This implies that the rule for the ^n operator should be to simply subtract 1 from n. However, computing ∂a((ab)*cd)^3 using the DFA manipulation method returns the expression (b(ab)*cd)^3. Here, the value of n is unchanged but instead must compute the derivative of expression inside the brackets. So it seems that there is an undefined rule that is more complex to deal with. It is possible that with more testing we would find that the rule for ^ could differ depending on the nested operators over which it has scope. The rule I cover in my system is the first possibility, i.e. where we substract 1 from n, as it is relevant to a concept that will be discussed in section 4.1.1.

3.3 Comparison

We now have two systems that differ vastly in their implementation, but both work towards the same goal of computing regular expression derivatives. Both seem to have their obvious advantages and disadvantages which are outlined below.
<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less complicated in theory. Only changing initial state</td>
<td>However, converting from DFA back into regular expressions is more complicated than regular expression to DFA conversion</td>
</tr>
<tr>
<td>Useful when focusing more on DFA as opposed to the regular expression on which it is based</td>
<td>Produces messy output that must be simplified</td>
</tr>
<tr>
<td>Seems to be more accurate for expressions containing $\wedge n$ operator</td>
<td>Simplification step can be expensive for larger expressions and output can still be difficult to read as alphabet symbols have been rearranged</td>
</tr>
<tr>
<td>Less computationally expensive if changing initial state is trivial</td>
<td>DFA to regular expression algorithm produced some incorrect results for expressions such as $(abc)^<em>(aef)^</em>$</td>
</tr>
</tbody>
</table>
### Rule-Based Method

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not have to convert from DFA back into regular expression</td>
<td>More complicated in theory as we are dealing with multiple rules at once</td>
</tr>
<tr>
<td>Produces simpler output than DFA manipulation method</td>
<td>More computationally expensive if changing initial state is trivial</td>
</tr>
<tr>
<td>Useful if focusing more on regular expressions as opposed to the DFAs they produce as output is more legible</td>
<td>Some rules, e.g. for $^n$ are difficult to define. In some cases we just subtract 1 from $n$, in other cases rule follows the form $\partial_a r^n = \partial_a r \cdot r$.</td>
</tr>
</tbody>
</table>

One of the main questions at this point should be the practicality of a toolkit that can compute derivatives of extended regular expressions. Some applications of such a system will be covered in section 4. For some of these applications the DFA manipulation method might be better suited to completing the task, while for others it would be more logical to used a rule-based derivative calculator. When deciding which method to use for these applications the advantages and disadvantages of both implementations should be taken into account. I discuss this further in each application’s respective section.

## 4 Applications of Derivatives

### 4.1 DFA Creation

Brzozowski (1964) was the first to employ this method of using derivatives for DFA construction and is also discussed by Owens et al. (2009) when they compare derivative based construction with other DFA construction methods. The results were positive and showed that the regular expression derivative method produced a smaller finite state machine.

In this DFA construction method, states are labelled as regular expressions and the derivative functions are transitions from these states. Consider the initial state $q_0$ and $q_0$ is labelled with the language $L$. Computing the derivative of $L$ with respect to $\partial_a$ gives us the derivative of $L$, which is $\partial_a L$. If $\partial_a L$ is not equal to $L$, then we create a new state $q_2$ labelled with the language $\partial_a L$. The arc between $q_1$ and $q_2$ is then labelled with the derivative character $a$. A worked through example from Owens et al. (2009) can be found below.
\( r = ab|ac \)
\( \Sigma = \{a, b, c\} \)

\[ \partial_a q_0 = \partial_a(ab|ac) = b|c \quad \text{which is new, so call it } q_1 \quad (1) \]
\[ \partial_a q_1 = \partial_a(b|c) = \emptyset \quad \text{which is new, so call it } q_2 \quad (2) \]
\[ \partial_a q_2 = \partial_a\emptyset = \emptyset = q_2 \quad (3) \]
likewise for \( \partial_b q_2 \) and \( \partial_c q_2 \) (4)
\[ \partial_b q_1 = \partial_b(b|c) = (\epsilon + \emptyset) = \epsilon \quad \text{which is new, so call it } q_3 \quad (5) \]
\[ \partial_a q_3 = \partial_a \epsilon = \emptyset = q_2 \quad (6) \]
likewise for \( \partial_b q_3 \) and \( \partial_c q_3 \) (7)
\[ \partial_c q_1 = \partial_c(b|c) = (\emptyset|\epsilon) = \epsilon = q_3 \quad (8) \]
\[ \partial_b q_0 = \partial_b(ab|ac) = \emptyset = q_2 \quad (9) \]
\[ \partial_c q_0 = \partial_c(ab|ac) = \emptyset = q_2 \quad (10) \]

Figure 7: DFA for \( r = ab|ac \)

This simple algorithm has some issues in terms of its efficiency. These issues are addressed by Owens et al. (2009) when building on Brzozowski’s work. They are:

1. The problem of determining when two regular expressions are equivalent is expensive. Owens et al. (2009) use weaker notions of regular
expression equivalence to resolve this issue. Xfst compares regular expressions by the DFAs that they produce, so the rulebased toolkit produced during this project could be extended for DFA construction.

2. Iteration over symbols in the alphabet \( \Sigma \) is okay for small alphabets but becomes inefficient when dealing with large alphabets such as the Unicode character set of over 1.1 million code points. Owens et al. (2009) use character sets as a way of dealing with this problem.

In this report we have seen two possible ways of computing regular expression derivatives. One involves creating a DFA and changing the DFA’s initial state while the other uses a set of calculus rules. The rule-based toolkit seems to be better suited to the task of DFA creation for the following reasons. If we were to complete this task using the DFA manipulation method we would have to read in a regular expression and convert it into a DFA, which is what we are trying to do in the first place. So it seems rather pointless to use this method if it already involves creating a DFA. For the rule-based implementation we simply compute \( \partial_a L \) using the rules we have been given, use \( \partial_a L \) to label the next state in our DFA and use the derivative character \( a \) to label the arc between the states labelled \( L \) and \( \partial_a L \).

Before moving on to the next application of derivatives of extended regular expressions, derivatives applied to MSO, I will address an issue involving labelling states as languages. This is covered in the next section.

4.1.1 States as Languages

This section addresses another issue with the derivative method for DFA construction discussed in the previous section. We have already said that studies showed how the derivative method for DFA creation tends to produce smaller finite state machines. From this we can see the appeal of using this method and labelling states as languages. However, there are some cases where this is not ideal.

Take the example of the language \( L_n \) defined by the regular expression below.

\[
L_n := (0|1)^*1(0|1)^{n-1}
\]

Focusing on the second half of this expression we get \( 1(0|1)^{n-1} \). The important thing to notice here is that strings accepted by this language must contain a 1 followed by \( n - 1 \) 0s or 1s. In other words, the \( n \)th last character in the string must be a 1. We do not need to worry so much about
the first part of the expression (0|1)*, so long as this first constraint is met. See below and example where \( n = 6 \). The 6th last bit has been underlined.

\[(0|1)^*1(0|1)^{n-1}\quad \text{where } n = 6\]

\[
\begin{array}{c}
1001011 \\
\overbrace{00101}^{n-1}
\end{array}
\]

The languages that are accessible by the derivatives of \( L_n \) follow the form:

\[L_n + \sum_{i \in q} (0 + 1)^{i-1} \quad \text{for some } q \subseteq \{1, \ldots, n\}\]

where \( q \) stores the last \( n \) bits seen.

If we were to represent \( L \) as a DFA in which the states are languages our initial state \( q_0 \) would be labelled \( L_n \). Our alphabet for this language is \( \Sigma = \{0, 1\} \). So to find our next transition and states we first compute the derivative with respect to 0.

\[
\partial_0 L_n = \partial_0((0|1)^*1(0|1)^{n-1}) \\
= (0|1)^*1(0|1)^{n-1}
\]

So computing the derivative of \( L_n \) with respect to 0 just gives us \( L_n \) back. This means that if we are at \( q_0 \) and we see a 0, we stay at \( q_0 \). Now lets compute the derivative of \( L_n \) with the next character in our alphabet, 1.

\[
\partial_1 L_n = \partial_1((0|1)^*1(0|1)^{n-1}) \\
= (0|1)^*1(0|1)^{n-1} \mid (0|1)^{n-1} \\
= L_n \mid (0|1)^{n-1}
\]

We use this derivative to create a transition labelled 1 between \( q_0 \) and the new state \( q_2 \) labelled \( L_n \mid (0|1)^{n-1} \).

If we now compute the derivative of the language at \( q1 \) with respect to both 0 and 1, we get the following.
\[ \partial_0(L_n \mid (0|1)^{n-1}) = L_n \mid (0|1)^{n-2} \]

\[ \partial_1(L_n \mid (0|1)^{n-1}) = L_n \mid (0|1)^{n-1} \mid (0|1)^{n-2} \]

From the above equations we can see that once we see a 1, we assume that this could be the \( n \)th last bit. Hence, the value being subtracted from \( n \) grows with each character we read. This is useful as it tells us what we can expect to see with the next character we read in. If this 1 is the \( n \)th last bit in the string, we should expect to see \( n - 1 \) occurrences of \((0|1)\), then with the next character we read we expect \( n - 2 \) more occurrences of \((0|1)\). To illustrate this, let’s look at the sample string from before

\[ 100101100101 \]

The first three occurrences of 1 could have the FSA believing it has reached the \( n \)th last bit. However, this turns out not to be the case. Regular expressions such as these when represented as DFAs can be expensive to store in terms of their size. A DFA accepting \( L_n \) will have at least \( 2^n \) states, whereas an NFA will only have \( n + 1 \) states.

**Claim**: An NFA accepting \( L_n \) has \( n + 1 \) states.

**Proof**: Initial state \( q_0 \) for \((0|1)^n \) and \( n \) states for \( 1(0|1)^{n-1} \)

**Claim 2**: A DFA accepting \( L_n \) has at least \( 2^n \) states.

**Proof**: Let \( M \) be a DFA with \(< 2^n \) states. On 2 strings \( s, s' \in (0|1)^n \), \( M \) ends up at the same state. Let \( k \) be an index where \( s \) and \( s' \) disagree. Exactly one of \( s0^{k-1} \) and \( s'0^{k-1} \) is in \( L_n \), so \( M \) can’t accept \( L_n \)
Recall before when we said that languages accessible by derivatives from $L_n$ have the form:

$$L_n + \sum_{i \in q} (0 + 1)^{i-1} \quad \text{for some } q \subseteq \{1, \ldots, n\}$$

where $q$ stores the last $n$ bits seen.

Instead of using languages to label states, we can use the subsets $q$ as states in a DFA, where the initial state is $\emptyset$. The formulae below show how the subset $q$ is altered each time we see a 0 and a 1.

$$\delta(q, 0) = \{i - 1 \mid i \in q \text{ and } i > 1\} \quad (1)$$

$$\delta(q, 1) = \delta(q, 0) \cup \{n\} \quad (2)$$

$q$ is final iff $1 \in q \quad (3)$

To demonstrate this in operation, let’s look at an example where $n = 4$ and the string we are reading is 001011. Note that when we read in the first two 0 characters the current state of the subset $q$ is equal to $\emptyset$, so the transition function $\delta(q, 0)$ returns $\emptyset$. In this walkthrough, the character that we are currently reading in is underlined.

<table>
<thead>
<tr>
<th>String</th>
<th>State</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>001011</td>
<td>$q = \emptyset$</td>
<td>by rule (1)</td>
</tr>
<tr>
<td>001011</td>
<td>$q = \emptyset$</td>
<td>by rule (1)</td>
</tr>
<tr>
<td>001011</td>
<td>$q = {4}$</td>
<td>by rule (2)</td>
</tr>
<tr>
<td>001011</td>
<td>$q = {3}$</td>
<td>by rule (1)</td>
</tr>
<tr>
<td>001011</td>
<td>$q = {4, 2}$</td>
<td>by rule (2)</td>
</tr>
<tr>
<td>001011</td>
<td>$q = {4, 3, 1}$</td>
<td>by rule (2)</td>
</tr>
</tbody>
</table>

$q$ is final as $1 \in q$ by rule (3).

In the above figure $1 \in q$ at the state we end at, meaning that the string 001011 is accepted by this FSA. We can see the advantage that a system like this may have over one that has languages as states. While the method of using languages as states shows us what we should expect to see, this method in which the the subsets $q$ are used as states allows us to keep track of what we have already seen.
4.2 Derivatives applied to MSO

4.2.1 Background

Monadic second-order logic (MSO) is an extension of first order logic that allows quantification over unary predicates. Binary and ternary predicates and functions can be used but only unary predicates may be quantified over (Gurevich 1985). We can use MSO-sentences to logically represent regular language strings. See the example below for the regular expression $ab^*b$, $\Sigma^* = \{ab, abb, \ldots\}$

$$(\exists x)(\exists y) \ P_a(x) \land P_b(y) \land (\forall z)(\neg S(z, x) \land (z = x \land P_b(z)))$$

We can build on this idea and represent regular language strings as MSO-models. Some preliminaries we need are an understanding of the binary relation (successor) $S$, e.g. $S(x_1, x_2)$ meaning $x_2$ follows $x_1$, and the unary relation symbol $P_\sigma$ for each symbol $\sigma \in \Sigma$. Below is an example of an MSO-model for the regular expression $abbc$, where $D$ is the set of string positions and $S$ is the set of successor relations.
\[ \langle D_4, S_4, [P_a], [P_b], [P_c] \rangle \]

\[ D_4 = \{1, 2, 3, 4\} \]

\[ S_4 = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle\} \]

\[ [P_a] = \{1\} \]

\[ [P_b] = \{2, 3\} \]

\[ [P_c] = \{4\} \]

\[ (\exists x_1)(\exists x_2)(\exists x_3)(\exists x_4)P_a(x_1) \land P_b(x_2) \land P_b(x_3) \land P_c(x_4) \land S(x_1, x_2) \land S(x_2, x_3) \land S(x_3, x_4) \land \neg(\exists x)(S(x, x_1) \land S(x_4, x)) \]

We can now look at a more in-depth example with the string \( abacd \) and the assignment functions \( f(X) = \{1, 2, 3\} \) and \( f(x) = 2 \). If we box the set of positions as opposed to enclosing it in curly braces we get the representation given below.

\[ \begin{array}{cccc}
  a & X & b & x \\
  a & X & c & X & b
\end{array} \]

In the above string representation, each box corresponds to a position in our string. So, in the first position have \( a \) and \( X \), in the second position we have \( b \) and \( x \) and so on. The corresponding MSO model for this string would be

\[ n = 5 \]
\[ P_a = \{1, 3\}, P_b = \{2, 5\}, P_c = \{4\}, P_x = \{2\}, P_X = \{1, 3, 4\} \]

We can place restrictions, or \( A \)-reducts, over these. These reducts can be defined by the formula

\[ \rho_A(\alpha_1 \alpha_2 \ldots \alpha_n) = (\alpha_1 \cap A)(\alpha_2 \cap A) \ldots (\alpha_n \cap A) \]

where

\[ A \subseteq \Sigma \text{ and } a_i = \{a \in \Sigma | i \in P_a\} \]

In simple terms we can say that reducts allow us to filter certain predicates out of our string by only keeping the set of \( \rho \) values. So from our last example:
\[
\rho_{\{a,b,c\}}(a, X b, x a, X c, X b) = a \ b \ a \ c \ b
\]
\[
\rho_{\{b,x\}}(a, X b, x a, X c, X b) = b, x
\]

### 4.2.2 Derivatives and MSO

Given a language \( L \) and a relation \( R \) between strings, let \( \langle R \rangle L \) be the preimage of \( L \) with respect to \( R \).

\[
\langle R \rangle L = \{ s \mid (\exists s' \in L)(s s') \}
\]

If we want to calculate derivative of a preimage with respect to a string function \( f \) we have the following formula

\[
\partial_a((f) L) = \langle f \rangle \partial_{f(a)}(L)
\]

As an example:

\[
L = (\emptyset | b, x) (\emptyset | b)^*
\]

\[
f = \rho_{\{b,x\}}
\]

\[
a = y, b, x \quad f(a) = b, x
\]

\[
\partial_a((f) L) = \langle f \rangle (\emptyset | b)^* b, x (\emptyset | b)^*
\]

The relevance of applying derivatives to MSO ties in with the concepts discussed in section 4.1. Computing the derivative with respect to \( b, x \) gives us a transition labelled \( b, x \) to another state labelled with the resulting derivative expression. From the example above, our starting state is labelled with the language \( (\emptyset | b)^* b, x (\emptyset | b)^* \). Computing the derivative with respect to \( b, x \) results in an arc labelled \( b, x \), leading to a state labelled with the language \( (\emptyset | b) \). From this we can see how using monadic second order logic

36
with derivatives can be useful in terms of natural language processing. If we were to extend the algorithm for DFA creation and apply it to MSO, we could use the rule-based derivative calculator to assist in creating DFAs based on MSO models. For this to work the toolkit would have to be extended to compute derivatives of extended regular expressions with respect to sets of symbols such as \([b, x]\), as opposed to only computing derivatives with respect to alphabet symbols and strings. This could be done by adding a separate class to represent these boxed sets instead of just using characters. We can then compute the derivative the same way as before, only instead of comparing the derivative character with the character at each starting point, we would be comparing boxed sets of symbols.

The last application of regular expression derivatives to be covered is the idea of applying derivatives to Frames.
4.3 Derivatives of Extended Regular Expressions and Frames

4.3.1 Background

Frames are used to model knowledge of abstract concepts and events in human cognition. They extend on and provide a solution to the problems associated with the idea of feature structures (Barsalou 1992, Fillmore 1982). Where feature lists are comprised of specific characteristics that a category member may possess, frames include co-occurring sets of attributes that can have different values. Thus, the frame model implies that people make distinctions between different entities based on more abstract attributes as opposed to specific values. Barsalou (1992) uses the example of a person distinguishing a car from a dog, with the car having attributes such as engine and transmission but a dog having the unrelated attributes of fur and temperament. Each of these attributes can be made up of further attributes, e.g. fur: colour, or may point to a specific value; transmission: manual. It is the distinction between specific attributes that an entity may possess, such as engine in the case of a car, that enables us to distinguish between vastly different entities, i.e. the difference between car and dog. It is the values that these attributes possess that allow us to distinguish between entities that are more closely related, for example; the difference between a car having the attribute-value pair transmission: manual and another having transmission: manual.

Barsalou (1992) also gives examples of studies which give some insight into how animals use attribute-value sets in discrimination learning. One of the stimuli-reward studies showed how animals were able to adapt to intradimensional transfer, i.e. changing the value of the same attribute, faster than they would adapt to extradimensional transfer, where the change is between attributes themselves. The results of this study imply that animals store information about the stimulus as attribute values, not as independent features which are a characteristic of feature structures.

The idea of frames can be extended to represent events, in which attributes are labelled by semantic roles. Take the example in figure 9 from Muskens (2013), a frame describing the sentence In Paris, John smashed the window with a hammer for Susan.
From the above figure we can see that the event takes its name from the verb, in this case smash, and this verb becomes the root of the frame. This is typical of what we can observe with simple examples; the frame-evoking unit, which will be the root, is the main verb in the sentence and all of the attributes linked to it are its syntactic dependents (Ellsworth et al. 2010).

The next property of frames that should be mentioned is their recursive nature. Barsalou (1992) uses this as a key factor for distinguishing frames from feature lists. The idea here is that the value assigned to an attribute of the root of a frame, can itself be made up of other attributes. An example that Muskens (2013) uses adds more information to the frame in figure 9. A window can in itself be assigned further attribute sets (shape: round, material: glass) to help distinguish it from other entities within its category. See figure 9 on the next page for a frame modelling the event In Paris, John smashed the round glass window with a hammer for Susan.
Figure 10: Frame modelling the event *In Paris, John smashed the round glass window with a hammer for Susan*

By understanding this idea of recursion we can form tables to represent these frames such that the two shown below.

<table>
<thead>
<tr>
<th>smash =</th>
<th>ATTRIBUTE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AGENT</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td>THEME</td>
<td>window</td>
</tr>
<tr>
<td></td>
<td>INSTRUMENT</td>
<td>hammer</td>
</tr>
<tr>
<td></td>
<td>LOCATION</td>
<td>Paris</td>
</tr>
<tr>
<td></td>
<td>BENEFICIARY</td>
<td>Susan</td>
</tr>
</tbody>
</table>
window = | ATTRIBUTE | VALUE |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAPE</td>
<td>round</td>
</tr>
<tr>
<td>MATERIAL</td>
<td>glass</td>
</tr>
</tbody>
</table>

Take the first frame for the smash event. One of the values in this frame is window, and has been assigned to the attribute THEME. This value points to another frame, the frame for window, which provides us with more information about this entity, i.e. its SHAPE and MATERIAL.

Also notice that this representation of a frame is similar in structure to an FSA, in which the state labelled smash is our initial state. All other states are final and provide additional information about the event at our initial state, i.e. they add more information about the smash event. The state labelled window also has arcs which lead to states providing more information about this particular entity. The fact that the window state is final tells us that this additional information is optional.

Muskens (2013) provides a way for us to translate a structure such as the one in figure 8 into a series of logical facts but before doing this we must first understand that events of this nature are made up of facts. These facts are constrained by axioms that Muskens (2013) borrows from Veltman (1985) and are listed below. In these axioms the constant 0 is used to denote the improper fact and the symbol $\circ$ represents the combination of facts. From this with get this axiom $f \circ g = 0$, which signifies that the facts $f$ and $g$ are incompatible.

\[
\exists f \ f \neq 0 \\
f \circ f = f \\
f \circ g = g \circ f \\
(f \circ g) \circ h = f \circ (g \circ h) \\
0 \circ f = 0
\]

There are two further properties worth noting. The first is that if $f \circ g = f$, then $f$ incorporates $g$. This is written as $f \leq g$. It follows that if $f \leq g$ and $h \leq f$, then $h \leq g$. This can be obtained from the axiom below

\[
f \not\leq g \rightarrow \exists h (Ah \leq f \land h \not\leq g)
\]
With these axioms, and remembering that events are made up of a combination of facts, we can begin to translate the event represented in figure 9 as a logical sequence. To do this we treat each value as a single place relation, i.e. we assign it a unique variable. In our example the variables we have are $y_1, \ldots, y_5$, each corresponding to John, window, hammer, Paris and Susan respectively. We must then treat each attribute as a two place relation between the root and attribute value. The root smash is assigned the variable $x$, and is then related to each individual variable $y_1, \ldots, y_5$. Combining each of the resulting relations with the $\circ$ operator gives us the model below.

$$\text{smash } x \circ \text{AG } xy_1 \circ \text{John } y_1 \circ \text{TH } xy_2 \circ \text{window } y_2 \circ \text{INSTR } xy_3 \circ \text{hammer } y_3 \circ \text{LOC } xy_4 \circ \text{Paris } y_4 \circ \text{BEN } xy_5 \circ \text{Susan } y_5$$

The example above, given by Muskens, can be extended to include information relating to the window frame by adding the relations: SHAPE $y_2y_6 \circ$ round $y_6 \circ$ MATERIAL $y_2y_7 \circ$ glass $y_7$.

Now that we have a sense of how frames can be represented logically, we can talk about how derivatives can be applied to them.
4.3.2 Derivatives Applied to Frames

In the last section we said that frame structures can be modelled using FSAs in which we use the event name to label our initial state. Recall our example:

Figure 11: Frame modelling the event In Paris, John smashed the round glass window with a hammer for Susan

Here we have two frames; the frame modelling the smash event and the frame containing additional information about the theme of this event, window. If we were to apply the notion of derivatives to frames it would give us a new way to traverse such a structure. If we wanted more information about a particular fact within the smash frame, we could compute the derivative of smash with respect to one of its attributes, e.g. computing $\partial_{TH} smash$ gives us access to the state labelled window. This could be implemented using the DFA manipulation method, where computing the $\partial_{TH} smash$ returns a DFA containing all information about the THEME, i.e. the window.

The programme would have to be able to handle multi-character symbols, which is already implemented by xfst, making the extension of the toolkit described in section 3.1 much easier. The main issue would be the fact that xfst does not allow us to change the names of states. It uses the notation $s_0, s_1, \ldots, s_n$ to label states and $f s_0, f s_1, \ldots, f s_n$ to label final states. This could be handled in the toolkit by using a pair object to map each state value $s0, \ldots, s_n$ to the language we want to use to label our state. So for the smash event frame we could map $s_0$ (the initial state) to smash, $f s 1$ to John and so on. The arcs in xfst can be labelled with multi-char
values so xfst would be capable of creating a transition $s_0 \to AGENT \to s_1$.
So if we wish to apply derivatives to frames, the toolkit using DFA manipulation would a feasible option with some improvements.

We can also say that since frames can be represented as DFA, we should also be able to define them using regular expressions. This is an idea that has not quite been implemented yet but such regular expressions should be compatible with the calculus rules for computing regular expression derivatives. If this were the case then we could use these expressions to create DFAs modelling frames using the DFA creation method discussed in section 4.1. The derivatives of these expressions would again be languages used to label states while the derivative function would label the transitions between states. The toolkit could be put to use if it were extended to handle multi-character symbols. As an example, in figure 11 we have transition arcs labelled as words relating to semantic roles, as opposed to single characters.
5 Conclusion

5.1 Achievements

The goal at the start of this project was to implement two toolkits for computing derivatives of extended regular expressions and discussing some possible applications of such a toolkit. One system involves calculated derivative using a method of DFA modification in which we change the initial state of a DFA depending on the derivative character, then converting this DFA back into a regular expression. The second system is based on a set of calculus rules for computing derivatives of extended regular expressions.

The two systems differed vastly in terms of their advantages and disadvantages; the DFA modification method being useful in terms of its efficiency when focusing mainly on the DFA that the derivative of a regular expression produces as opposed to the regular expression itself, while the rule-based implementation produces a more simplified legible expression as output making it more useful for tasks which focus more on the derivative expression as opposed to its DFA.

We then covered some topics where toolkits such as these could be put to use. The first has been done by Brzozowski when he used regular expression derivatives for DFA creation. We then went over the advantages and disadvantages of this method such as the efficiency of using languages produced by computing derivatives to label states in an FSA.

The next topic involved the idea of applying derivatives to regular expressions extended by MSO, with MSO being a way of modelling regular languages through MSO-models that can be used to create FSAs. Using the idea of derivatives for DFA creation we can apply the rule-based toolkit to regular expressions based on MSO models to produce DFAs that represent them.

The final topic covered the application of derivatives to frames. We saw how frames can be represented as FSAs and how these frames are also recursive in that the value for an attribute of a frame can in fact be another frame. Derivatives allow us to access these other frames by changing the initial state of the DFA the represents them. We then went over how the toolkit that implements the DFA modification method could be extended to handle computing the derivatives of these frames.

Other than the extensions listed above there are other improvements that could be made for both toolkits. These were covered in sections 3.1.3 and 3.2.3. If a finite state toolkit were developed which would allow the
initial state of a DFA to be changed then the efficiency the the DFA manipulation based toolkit would greatly improved. An example of one of the improvements that could be made to the rule based toolkit would be the ability for better handle regular expressions that contain the $^{\text{n}}$ operator.

5.2 Future Work

First and foremost, if an open source finite state toolkit could be extended to allow the user to change the initial state of an FSA then the DFA manipulation based system could be improved.

A system that implements some of the applications covered in section 4 could also be developed using the toolkits created during the course of this project. The toolkit would have to be modified to handle the input of boxed sets for MSO, or multi-character symbols for frames.

Frames have not yet been modelled as regular expressions. This could be useful if one wishes to create DFAs representing frames from regular expressions. The DFA creation method in section 4.1 could be used to carry out such a task.
6 References


7 Appendix

The input to this programme should be in xfst format. For full documentation consult the xfst manual by Beesley & Karttunen (2003). Spaces should be left between characters in the regular expression and the operators the same as those used in standard regular expression notation, with the exception of the alternation operator which is |. Square brackets should be used instead of rounded brackets. Example: \([abc]\ast def\)[ghi] + [jkl]\^n

7.1 DFA Manipulation System

7.1.1 Calculator.cpp

Main class for this system. Handles reading in a regular expression and printing the output to the screen.

1. Reads in the regular expression, then uses a generator object to create a DFA for this expression.

2. Stores the information for each transition (start state, transition label and next state) as a TransitionTuple object.

3. Creates a converter object to convert from DFA to regular expression then print the output.

7.1.2 Generator.cpp

Class used to send regular expressions to xfst and create their DFAs.

1. void createFSA - create a script file which is send as input to xfst. Save xfst output to FSA_specs.txt

2. vector<TransitionTuple> getStateTransitions - reads in the FSA_specs.txt file and searches for information about the transitions in this FSA. Stores this information as a vector of TransitionTuple objects and returns this vector. Also counts the number of states and stores the final states in an vector of integer type values.

7.1.3 Converter.cpp

Used to convert from a DFA into a regular expression. The only functions in this class are the constructor and convert function. The algorithm used in the convert function was described in section 3.1.2.
7.1.4 TransitionTuple.cpp

Class used to store information about transitions. Has three instance variables:

1. **int first** - stores the number of the starting state
2. **string label** - stores the character for the transition
3. **int next** - stores the number of the state that you end up at if you follow the transition labelled with label from the state first

7.2 Simplifier

Programme that simplifies the previous system’s output.

7.2.1 MainSimplifier.cpp

1. Reads in a regular expression.
2. Creates a Simplifier object and passes the regular expression to this object so that it will be simplified.
3. Print the simplified output to the screen

7.2.2 Simplifier.cpp

1. **string Simplify** - function to simplify the string regular expression. Uses a vector of MyPair objects to store the index of each [, | and ] character we come across. As each opening square bracket or alternation character is read, we push their index and the character symbol itself to the MyPair vector. When we come across a closing square bracket and the last operator to be pushed to the stack was an alternation character, then we must compare its LHS and RHS. The LHS is the substring between the last opening bracket’s index and the index of the alternation symbol. The RHS is the substring between the alternation symbol and the closing bracket. We pass each of these substrings to the compare function. If they are equivalent then we remove the longer substring from our original string. Otherwise we leave things as they are.

   If we find a closing bracket but the last symbol to be pushed to the MyPair vector was an opening bracket, then we check the symbol to
the right of the closing bracket. If it is an alphabet symbol, then we can remove both the opening and closing brackets. If it is an operator symbol then we leave things as they are.

If we see a 0 character (recall that this is xfst notation for $\epsilon$) and it is concatenated with another 0 or an alphabet symbol, then we can delete the 0 symbol. Also, if it has a Kleene-star we delete the Kleene-star as it serves no purpose.

By the end, the function should have produced a string that containing less epsilon symbols, no disjunction between equivalent symbols and no epsilons under Kleene-closure.

2. **bool compare** - compares two regular expression strings. First converts the string to xfst format by inserting spaces between characters where necessary. Then creates a script file to pass to xfst as input. Reads the output and if the result line contains a 1 then the expression are equivalent. If the result line contains a 0 then the expressions are not equivalent. Returns the appropriate boolean value.

7.3 Rule-Based System

Notation is the similar to the DFA manipulation based system. However, we do not leave spaces between characters.

7.3.1 DerivativeCalculator.cpp

1. Read in regular expression.

2. Compute derivative of first character in string by creating a Regular-Expression object and calling the derive method. When the derivative of the regular expression has been computed we compute the derivative of this expression with respect to the next character in the string until we have reached the end of the string.

3. Prints the output to the screen.

7.3.2 RegularExpression.cpp

Class that represents a regular expression object. Instance variable _re stores the regular expression as a string.

1. **string derive** - returns the derivative of the regular expression string that is passed as a parameter (reg_expression). First finds all possible
starting point indexes by calling the findStartingPoints method. The programme iterates through the reg_expression until a | character is found. It then backtracks through reg_expression until the index of the last disjunction is reached. Everytime we see an operator such as *, ^n, + we check if it has scope over a starting point. If it does, then we apply its derivative rule. For certain operators such as the Kleene-star we have the rules that take the form $\partial_a r = \partial_a r \cdot r$. So, we call the derive function recursively by passing a substring that contains all of the characters within the brackets marking the scope of the Kleene-star operator and concatenate this string with a substring that starts from the end of the Kleene-star and ends at the | character that we are currently at.

If we reach the previous disjunction index and the first character turns out to be an alphabet symbol we recursively call the derive function by creating a substring that contains all characters between both | operators. This will eventually break the expression down into a string that no longer contains | symbols.

We continue through the original reg_expression string until we reach the end of the string. If the string contains no | symbols then we look at the first character in the string. If it is an alphabet character and this character is equal to the character with respect to which we are computing the derivative, we remove this character from the string and return the resulting string. However, if the first character is an opening bracket we must recursively call the derive function, passing the substring located between these brackets as a parameter. By the end of this method we will have a string that represents the derivative of reg_expression. This string is then returned.

2. string trim - Used to remove unwanted brackets and | symbols that could cause the programme to break or the output to look messy. For example if we compute $\partial_a (bc|a)$, from the derive algorithm we would end up with $(bc|)$ so the | should be removed. This is done by check if alternation characters are located directly next to brackets.

3. int findOpeningBrackets - In the derive and findStartingPoints functions we often have to search for a closing bracket’s corresponding opening bracket. This function accomplishes this using a bracket counter. This counter is initially set equal to 0 and the programme iterates back through the string, starting at the closing bracket index. Each time we see an opening bracket we increment its value by 1, and
each time we see a closing bracket we decrement its value by 1. If the character at the current index is an opening bracket and the bracket counter is equal to 0, we return the value of this index.

4. **int findClosingBrackets** - Same as the **findOpeningBrackets** function, only it finds a closing bracket’s corresponding opening bracket.

5. **vector<int> findStartingPoints** - Function that finds all possible starting string points in the regular expression. Algorithm used in this function is given in section 3.2.2.