Derivatives of Regular Expressions with Cuts

Niklas Zechner

Umeå University

Abstract
Derivatives of regular expressions are an operation which for a given expression produces an expression for what remains after a specific symbol has been read. This can be used as a step in the process of transforming an expression into a finite string automaton. Cuts are an extension of the ordinary regular expressions; the cut operator is essentially a concatenation without backtracking, formalising a behaviour found in many programming languages. Just as for concatenation, we can also define an iterated cut operator. We show and derive expressions for the derivatives of regular expressions with cuts and iterated cuts.

1. Introduction

Derivatives of regular expressions were introduced in 1964 by Janusz A. Brzozowski [1]. The idea is to compute a resulting regular expression after a given symbol has been read by a given expression. This fundamental operation can help avoid repeating costly computations.

The cut operator is an additional operator that can be used in regular expressions. It can be seen as a variant of concatenation, with the difference being that the left operand always matches the maximum number of symbols. With the ordinary concatenation operator $E \cdot F$, typical implementations try to match as much as possible of the given string to E, and then backtrack if that does not work. For example, consider the expression $\mathrm{a}^* \cdot \mathrm{b}^* \cdot \mathrm{a} \cdot \mathrm{b}$, and the string "aab". The computer will typically start by matching "aa" to the a^* part of the expression, but when that does not give an accepting result, it backtracks to matching only the first "a" to the a^* part, nothing to the b^* part, and lastly "ab" to $\mathrm{a} \cdot \mathrm{b}$. The cut operator is similar, but without backtracking. We write the cut of expressions E and F as $E \,!\, F$. If we replace one concatenation in the above example, and get $\mathrm{a}^* \cdot \mathrm{b}^* \,!\, \mathrm{a} \cdot \mathrm{b}$, this no longer matches the string "aab"; "aa" is matched to a^*, "b" is matched to b^*, and then when there is nothing to match with $\mathrm{a} \cdot \mathrm{b}$, no backtracking is done, so there is no match. The lack of backtracking makes cuts faster to evaluate than regular concatenation, and the determinism is useful in programming languages that allow capturing parts of the expression [2].

The idea of this kind of operators was first suggested by Friedl [3], and the cut operation as shown here was formalised by Berglund et al. [2]. Similar concepts are available in many programming languages that use regular expressions, such as Perl [4] and Java [5]. They have several purposes; sometimes the specific behaviour is what you want, and sometimes these operations are used instead of concatenation because it removes the nondeterminism and speeds up the process. Expressions with cuts still denote regular languages; for example, $\mathrm{a}^* \cdot \mathrm{b}^* \,!\, \mathrm{a} \cdot \mathrm{b}$ can be rewritten as $\mathrm{a}^* \cdot \mathrm{b} \cdot \mathrm{b}^* \cdot \mathrm{a} \cdot \mathrm{b}$. But they can be much more succinct; there are classes of cut expressions with a single cut operator such that the corresponding cut-free regular expression (or automaton) is exponentially larger [2].

We can also define a closely related operator, the iterated cut. Just as the iteration operator is repeated concatenation, the iterated cut is repeated cut operations. We will write the iterated cut of E as $E!^*$. Thus, a string matches $E!^*$ if and only if it matches $E \,!\, (E \,!\, ...(E \,!\, (E \,!\, \{\epsilon\}))...)$ for any number of cuts.

We will show how to compute the derivatives of expressions with cuts and iterated cuts. Since similar operators are available in several practical implementations of regular expressions, this is an important step for implementing algorithms based on derivatives. It has practical uses in many real-world applications, such as handling server requests [6].

2. Preliminaries

We assume that the reader is familiar with regular expressions, as well as (deterministic finite string) automata. We take $\mathrm{L}(E)$ to be the language recognised by a regular expression E, and consider regular expressions formed by the following:

$\emptyset$ is the empty set
ϵ is the empty string
$\cdot$ is the concatenation operator
$|$ is the alternation (union) operator
$*$ is the iteration operator

Derivatives can also be applied to extended regular expressions, including:

$\&$ is the combination (intersection) operator
$\sim$ is the complement operator

That is, $E \,\&\, F$ matches any string that matches both E and F, and $\sim E$ matches any string that does not match E. When there is no ambiguity, we use regular expressions directly as shorthand for the languages thereof, and we use EF as shorthand for $E \cdot F$.

2.1. Derivatives

Formally, derivatives are applied to languages, that is, sets of strings. For a language M, the derivative of M with respect to a symbol s is the set of remainder strings after having read s from any string in M. There are several ways to write derivatives; here, we denote this derivative by $\mathrm{D}_s\, M$.

$$w \in \mathrm{D}_s\, M \leftrightarrow s \cdot w \in M$$

Less formally, we apply derivatives to regular expressions. We use

$$\mathrm{D}_s\, E = F$$

as a shorthand for

$$\mathrm{D}_s\, \mathrm{L}(E) = \mathrm{L}(F)$$

For example, $\mathrm{D}_a\,(\mathrm{abc}) = \mathrm{bc}$, and $\mathrm{D}_c\,((\mathrm{ab} \mid \mathrm{cd})\ \mathrm{e}) = \mathrm{de}$. This can easily be extended to derivatives with respect to strings of letters; we will not get into that here. If $\epsilon \in \mathrm{L}(E)$, we say that E is *nullable*, which we denote $\text{Њ}E$. The derivatives of the usual regular expression operators are

```
$\mathrm{D}_s\, \emptyset = \emptyset$
$\mathrm{D}_s\, \epsilon = \emptyset$
$\mathrm{D}_s\, s = \epsilon$
$\mathrm{D}_s\, t = \emptyset$, if $t \neq s$
$\mathrm{D}_s\,(E^*) = (\mathrm{D}_s\, E) \cdot E^*$
$\mathrm{D}_s\,(E \mid F) = (\mathrm{D}_s\, E) \mid (\mathrm{D}_s\, F)$
$\mathrm{D}_s\,(E \,\&\, F) = (\mathrm{D}_s\, E) \,\&\, (\mathrm{D}_s\, F)$
$\mathrm{D}_s\,(\sim E) = \sim (\mathrm{D}_s\, E)$
$\mathrm{D}_s\,(E \cdot F) =$
    if $\text{Њ}E$, then $((\mathrm{D}_s\, E) \cdot F) \mid (\mathrm{D}_s\, F)$
    else $(\mathrm{D}_s\, E) \cdot F$
```

2.2. Cut expressions

As mentioned in Section 1, the cut operator is a variant of concatenation, without backtracking, so the left operand must match the longest possible substring. Formally,

$$\mathrm{L}(E \,!\, F) = \{uv \mid u \in E, v \in F, \forall w \in \mathrm{pref}(v) \setminus \{\epsilon\} : uw \notin E\}$$

where $\mathrm{pref}(v)$ is the set of prefixes of v. We also introduce the iterated cut, which we denote by $E!^*$, for an expression E. It stands for any number of iterations of cuts, so

$$E!^* = \epsilon \mid E \mid (E \,!\, E) \mid (E \,!\, E \,!\, E) \mid ...$$

More formally, we can define it through an auxiliary function:

```
$\mathrm{ncuts}(n, E) =$
    if $n = 0$, then $\epsilon$
    else $E \,!\, \mathrm{ncuts}(n-1, E)$
$L(E!^*) = \{x \mid \exists n : x \in \mathrm{ncuts}(n, E)\}$
```

3. Derivatives of cut expressions

In the following, we will present and derive expressions for derivatives of the simple cut and iterated cut. For the former, we define an auxiliary operator:

$$L(E \,{}^{\eta}\mathrm{b}\, F) = L(F) \setminus L(E\ \Sigma^*)$$

That is, $E \,{}^{\eta}\mathrm{b}\, F$ matches strings that match F, but have no prefix that matches E. Using the extended regular expression operators, this could also be written as

$$E \,{}^{\eta}\mathrm{b}\, F = \sim (E\ \Sigma^*)\ \&\ F$$

3.1. Simple cut

We can write the derivatives of both operators as follows.

Theorem 3.1
$\mathrm{D}_s\,(E \,!\, F) =$
  if $\text{Ѣ}\,E$, *then* $((\mathrm{D}_s\,E) \,!\, F) \mid ((\mathrm{D}_s\,E) \,{}^{\eta}\mathrm{b}\, (\mathrm{D}_s\,F))$
  else $(\mathrm{D}_s\,E) \,!\, F$

Theorem 3.2
$\mathrm{D}_s\,(E \,{}^{\eta}\mathrm{b}\, F) =$
  if $E = \emptyset$, *then* $\mathrm{D}_s\,F$
  else if $\text{Ѣ}\,E$, *then* $\emptyset$
  else $(\mathrm{D}_s\,E) \,{}^{\eta}\mathrm{b}\, (\mathrm{D}_s\,F)$

We will now prove these results.

Proof of Theorem 3.1
Recall the definition of cuts,

$$\mathrm{L}(E \,!\, F) = \{uv \mid u \in E, v \in F, \forall w \in \mathrm{pref}(v) \setminus \{\epsilon\} : uw \notin E\}$$

For the purpose of derivatives with respect to a symbol s, it suffices to consider strings in $L(E \,!\, F)$ that start with s; that is,

$$\mathrm{D}_s\,L(E \,!\, F) = \mathrm{D}_s\,(L(E \,!\, F) \cap L(s \cdot \Sigma^*))$$

We can rewrite $L(E \,!\, F) \cap L(s \cdot \Sigma^*)$ as a union of two sets – one for when the initial s gets read by E, and one for when it gets read by F. The derivative of $E \,!\, F$ is equal to the derivative of the union of the sets,
$\{sxy \mid sx \in E, y \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : sxw \notin E\}$ and
$\{sy \mid \epsilon \in E, sy \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : sw \notin E\}$.

If E is not nullable, then the first symbol has to match E, so we get only the first case:

$$\begin{aligned}
\mathrm{D}_s\,(E\;!\;F) &= \{xy \mid sx \in E, y \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : sxw \notin E\} = \\
&= \{xy \mid x \in \mathrm{D}_s\,E, y \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : xw \notin \mathrm{D}_s\,E\} = \\
&= (\mathrm{D}_s\,E)\;!\;F
\end{aligned}$$

If E is nullable, we include both sets:

$$\begin{aligned}
\mathrm{D}_s\,(E\;!\;F) &= \{xy \mid sx \in E, y \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : sxw \notin E\} \cup \\
&\quad \cup \{y \mid sy \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : sw \notin E\} = \\
&= \{xy \mid x \in \mathrm{D}_s\,E, y \in F, \forall w \in \mathrm{pref}(y) \setminus \{\epsilon\} : xw \notin \mathrm{D}_s\,E\} \cup \\
&\quad \cup \{y \mid y \in \mathrm{D}_s\,F, \forall w \in \mathrm{pref}(y) : w \notin \mathrm{D}_s\,E\} = \\
&= ((\mathrm{D}_s\,E)\;!\;F) \mid ((\mathrm{D}_s\,E)\;{}^{\text{ı}}\text{b}\,(\mathrm{D}_s\,F))
\end{aligned}$$

Proof of Theorem 3.1
As stated above, $E\;{}^{\text{ı}}\text{b}\,F$ can be written as $\sim(E\;\Sigma^*)\;\&\;F$. From that, we get

$$\begin{aligned}
&\mathrm{D}_s\,(\sim(E\;\Sigma^*)\;\&\;F) = \\
&= \mathrm{D}_s\,(\sim(E\;\Sigma^*))\;\&\;\mathrm{D}_s\,F = \\
&= \sim\mathrm{D}_s\,(E\;\Sigma^*)\;\&\;\mathrm{D}_s\,F =
\end{aligned}$$

$$\begin{aligned}
&= \text{if } \text{Ѣ}\,E : \sim((\mathrm{D}_s\,E)\;\Sigma^* \mid \mathrm{D}_s\,(\Sigma^*))\;\&\;\mathrm{D}_s\,F \\
&\ \text{else} : \qquad \sim((\mathrm{D}_s\,E)\;\Sigma^*)\;\&\;\mathrm{D}_s\,F =
\end{aligned}$$

$$\begin{aligned}
&= \text{if } E = \emptyset : \ \sim(\emptyset\;\Sigma^*)\;\&\;\mathrm{D}_s\,F = \mathrm{D}_s\,F \\
&\ \text{else if } \text{Ѣ}\,E : \sim(\Sigma^*)\;\&\;\mathrm{D}_s\,F = \emptyset \\
&\ \text{else} : \qquad \sim((\mathrm{D}_s\,E)\;\Sigma^*)\;\&\;\mathrm{D}_s\,F = (\mathrm{D}_s\,E)\;{}^{\text{ı}}\text{b}\,(\mathrm{D}_s\,F)
\end{aligned}$$

3.2. Iterated cut

The derivative of the iterated cut is similar to that of the common iteration operator.

Theorem 3.3
$\mathrm{D}_s\,(E!^*) = (\mathrm{D}_s\,E)\;!\;E!^*$

Proof of Theorem 3.3 Like the iteration operator, the iterated cut matches zero or more times. We can rewrite that as

$$E!^* = \epsilon \mid (E\;!\;E!^*)$$

We have

$$\begin{aligned}
\mathrm{D}_s\,E!^* &= \quad \mathrm{D}_s\,(\epsilon \mid (E\;!\;E!^*)) = \\
&= \quad (\mathrm{D}_s\,\epsilon) \mid \mathrm{D}_s\,(E\;!\;E!^*) = \\
&= \text{if } \text{Ѣ}\,E : \emptyset \mid ((\mathrm{D}_s\,E)\;!\;E!^*) \mid ((\mathrm{D}_s\,E)\;{}^{\text{ı}}\text{b}\,(\mathrm{D}_s\,E!^*)) \\
&\ \text{else} : \quad \emptyset \mid ((\mathrm{D}_s\,E)\;!\;E!^*)
\end{aligned}$$

Since $E!^*$ starts with E, we know that $((\mathrm{D}_s\,E)\,{}^{\mathrm{n}}\mathrm{b}\,(\mathrm{D}_s\,E!^*)) = \emptyset$, so

$$\mathrm{D}_s\,(E!^*) = (\mathrm{D}_s\,E)\;!\;E!^*$$

4. Discussion

One application of derivatives is for transforming a regular expression into an automaton. Brzozowski [1] described a method for this: We start with a single state corresponding to the original expression, and then process each state by adding new states for all of its derivatives, until there are no more states to add.

An advantage of this technique is that we can run the algorithm only partway, leaving some quasi-states representing expressions. That way, we get the speed of an automaton for the first part of the string, but without the space requirements of the complete automaton, and without having to compute it. One use is when the expressions are likely to lead to a reject on the first few symbols. Then we can run the algorithm breadth-first until we have states for a number of symbols, and save the regular expressions for the remainder. Another use is when the input strings are likely to be similar. Then we can start expanding quasi-states as required, making a lazy evaluation of regular expressions. Then the first string may take a long time, but subsequent similar strings will be faster.

The apparent downside of the method is that a new expression might be equivalent to an existing one. If we do nothing, the algorithm might not terminate; there is always a finite set of equivalence classes [1], but in some cases we might get endless sequences of equivalent expressions. If we check for equivalence, the method could be very slow, since equivalence testing of regular expressions is PSPACE-complete. However, there are simpler checks that can be done efficiently, and are enough to make sure the algorithm terminates, the only downside being that the automaton will not necessarily be minimal [7]. This means that this can be a viable method for transforming regular expressions to automata.

Although definitions vary, cuts or similar operations are available in programming languages and other software. They are an effective tool, and replacing them with cut-free expressions can be very inefficient. This means that when developing algorithms for transforming regular expressions to automata, it would be valuable to include operators like these. The expressions for their derivatives, which we have shown here, are a step towards that goal.

5. Acknowledgements

The author wishes to thank Bruce Watson for very helpful advice and discussions.

References

[1] J. A. Brzozowski, Derivatives of regular expressions, Journal of the ACM (JACM) 11 (4) (1964) 481–494.

[2] M. Berglund, H. Björklund, F. Drewes, B. van der Merwe, B. Watson, Cuts in regular expressions, in: Developments in Language Theory, Springer, 2013, pp. 70–81.

[3] J. E. Friedl, Mastering regular expressions, O'Reilly Media, Inc., 2002.

[4] Perl documentation, accessed 2016-06-06.
URL `http://perldoc.perl.org/perlre.html`

[5] Java documentation, accessed 2016-06-06.
URL `https://docs.oracle.com/javase/7/docs/api/java/util/\penalty\z@regex/Pattern.html`

[6] B. W. Watson, Hardware implementations of finite automata and regular expressions, in: Implementation and Application of Automata, Springer, 2015, pp. 13–17.

[7] B. W. Watson, et al., Taxonomies and toolkits of regular language algorithms, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1995.