Regular-expression derivatives re-examined

SCOTT OWENS
University of Cambridge
(e-mail: Scott.Owens@cl.cam.ac.uk)

JOHN REPPY
University of Chicago
(e-mail: jhr@cs.uchicago.edu)

AARON TURON
University of Chicago, Northeastern University
(e-mail: turon@ccs.neu.edu)

Abstract
Regular-expression derivatives are an old, but elegant, technique for compiling regular expressions to deterministic finite-state machines. It easily supports extending the regular-expression operators with boolean operations, such as intersection and complement. Unfortunately, this technique has been lost in the sands of time and few computer scientists are aware of it. In this paper, we reexamine regular-expression derivatives and report on our experiences in the context of two different functional-language implementations. The basic implementation is simple and we show how to extend it to handle large character sets (e.g., Unicode). We also show that the derivatives approach leads to smaller state machines than the traditional algorithm given by McNaughton and Yamada.

1 Introduction
The derivative of a set of strings \( S \) with respect to a symbol \( a \) is the set of strings generated by stripping the leading \( a \) from the strings in \( S \) that start with \( a \). For regular sets of strings, i.e. sets defined by regular expressions (REs), the derivative is also a regular set. In a 1964 paper, Janusz Brzozowski presented an elegant method for directly constructing a recogniser from an RE based on RE derivatives (Brzozowski, 1964). His approach is elegant and easily supports extended REs, i.e. REs extended with Boolean operations such as complement. Unfortunately, RE derivatives have been lost in the sands of time, and few computer scientists are aware of them.\(^1\)

Recently, we independently developed two scanner generators, one for PLT Scheme and one for Standard ML, using RE derivatives. Our experiences with this approach have been quite positive: the implementation techniques are simple; the generated scanners are usually optimal in size; and the extended RE language allows for

\(^1\) A quick survey of several standard compiler texts does not turn up any description of them (Aho et al., 1986; Fisher & LeBlanc, Jr., 1988; Appel, 1998). The only mention we found, in fact, is an exercise in Aho and Ullman’s Theory of Parsing and Translation (Aho & Ullman, 1972).
more compact scanner specifications. Of special interest is that the implementation
techniques are well suited to functional languages that provide good support for
symbolic term manipulation (e.g. inductive data types and pattern matching).

The purpose of this paper is largely educational. Our positive experience with
RE derivatives leads us to believe that they deserve the attention of the current
generation of functional programmers, especially those implementing RE recognisers.
We begin with a review of background material in Section 2, introducing the notation
and definitions of REs and their recognisers. Section 3 gives a fresh presentation of
Brzozowski’s work, including DFA construction with RE derivatives. In addition to
re-examining Brzozowski’s (1964) work, we also report on the key implementation
challenges we faced in Section 4, including new techniques for handling large
character sets such as Unicode (Unicode Consortium, 2003). Section 5 reports our
experience in general and includes an empirical comparison of the derivative-based
scanner generator for SML/NJ with the more traditional tool it replaces. We
conclude with a review of related work and a summary.

2 Preliminaries

We assume a finite alphabet $\Sigma$ of symbols and use $\Sigma^*$ to denote the set of all finite
strings over $\Sigma$. We use $a$, $b$, $c$, etc. to represent symbols and $u$, $v$, $w$ to represent
strings. The empty string is denoted by $\varepsilon$. A language of $\Sigma$ is a (possibly infinite) set
of finite strings $L \subseteq \Sigma^*$.

2.1 REs

Our syntax for REs includes the usual operations: concatenation, Kleene-closure
and alternation. In addition, we include the empty set ($\emptyset$) and the Boolean operations
‘and’ and ‘complement’.\footnote{Other logical operations, such as exclusive or, can also be added.}

Definition 2.1

The abstract syntax of an RE over an alphabet $\Sigma$ is given by the following grammar:

\[
\begin{align*}
  r, s & ::= \emptyset \quad \text{empty set} \\
       & | \varepsilon \quad \text{empty string} \\
       & | a \quad a \in \Sigma \\
       & | r \cdot s \quad \text{concatenation} \\
       & | r^* \quad \text{Kleene-closure} \\
       & | r + s \quad \text{logical or (alternation)} \\
       & | r \& s \quad \text{logical and} \\
       & | \neg r \quad \text{complement}
\end{align*}
\]

These expressions are often called extended REs, but since the extensions are
conservative – i.e. regular languages are closed under Boolean operations (Rabin &
Scott, 1959) – we refer to them as REs. Adding boolean operations to the syntax
of REs greatly enhances their expressiveness, as we demonstrate in Section 5.1. We use juxtaposition for concatenation and add parentheses, as necessary, to resolve ambiguities.

The regular languages are those languages that can be described by REs according to the following definition:

**Definition 2.2**

The language of an RE $r$ is a set of strings $\mathcal{L}[r] \subseteq \Sigma^*$ generated by the following rules:

$$
\begin{align*}
\mathcal{L}[\emptyset] & = \emptyset \\
\mathcal{L}[\varepsilon] & = \{\varepsilon\} \\
\mathcal{L}[a] & = \{a\} \\
\mathcal{L}[r \cdot s] & = \{u \cdot v \mid u \in \mathcal{L}[r] \text{ and } v \in \mathcal{L}[s]\} \\
\mathcal{L}[r^*] & = \{\varepsilon\} \cup \mathcal{L}[r \cdot r^*] \\
\mathcal{L}[r + s] & = \mathcal{L}[r] \cup \mathcal{L}[s] \\
\mathcal{L}[r \& s] & = \mathcal{L}[r] \cap \mathcal{L}[s] \\
\mathcal{L}[\neg r] & = \Sigma^* \setminus \mathcal{L}[r]
\end{align*}
$$

To avoid notational clutter, we often let an RE $r$ denote its language $\mathcal{L}[r]$ and refer to REs and their languages interchangeably.

### 2.2 Finite state machines

Finite state machines (or finite automata) provide a computational model for implementing recognisers for regular languages. For this paper, we are interested in deterministic automata, which are defined as follows:

**Definition 2.3**

A deterministic finite automaton (DFA) over an alphabet $\Sigma$ is 4-tuple $\langle Q, q_0, \mathcal{F}, \delta \rangle$, where $Q$ is a finite set of states; $q_0 \in Q$ is the distinguished start state; $\mathcal{F} \subseteq Q$ is a set of final (or accepting) states; and $\delta : Q \times \Sigma \rightarrow Q$ is a partial function called the state transition function.

We can extend the transition function $\delta$ to strings of symbols

$$
\hat{\delta}(q, \varepsilon) = q \\
\hat{\delta}(q, au) = \hat{\delta}(q', u) \quad \text{when } q' = \delta(q, a) \text{ is defined}
$$

The language accepted by a DFA is defined to be the set of strings

$$
\{u \mid \hat{\delta}(q_0, u) \in \mathcal{F}\}
$$

### 3 RE derivatives

In this section, we introduce RE derivatives and show how they can be used to construct DFAs directly from REs.
3.1 Derivatives

The notion of a derivative applies to any language. Intuitively, the derivative of a language \( L \subseteq \Sigma^* \) with respect to a symbol \( a \in \Sigma \) is the language that includes only those suffixes of strings with a leading symbol \( a \) in \( L \).

**Definition 3.1**

The derivative of a language \( L \subseteq \Sigma^* \) with respect to a string \( u \in \Sigma^* \) is defined to be

\[
\partial_u L = \{ v \mid u \cdot v \in L \}.
\]

For example consider the language defined by the RE \( r = ab^* \). The derivative of \( r \) with respect to \( a \) is \( b^* \), while the derivative with respect to \( b \) is the empty set.

Derivatives are useful for scanner construction in part because the regular languages are closed under the derivative operation, as stated in the following theorem:

**Theorem 3.1**

If \( L \subseteq \Sigma^* \) is regular, then \( \partial_u L \) is regular for all strings \( u \in \Sigma^* \).

**Proof**

We start by showing that for any \( a \in \Sigma \), the language \( \partial_a L \) is regular. Let \( \langle Q, q_0, \mathcal{F}, \delta \rangle \) be a DFA that accepts the regular language \( L \). Then we can construct a DFA that recognises \( \partial_a L \) as follows: if \( \delta(q_0, a) \) is defined, then \( \langle Q, \delta(q_0, a), \mathcal{F}, \delta \rangle \) is a DFA that recognises \( \partial_a L \), and, thus, \( \partial_a L \) is regular. Otherwise \( \partial_a L = \emptyset \), which is regular. The result for strings follows by induction.

For regular languages that are represented as REs, there is a natural algorithm for computing the derivative as another RE. First we need a helper function \( v \) from REs to REs. We say that an RE \( r \) is nullable if the language it defines contains the empty string, that is if \( \epsilon \in L[r] \). The \( v \) function has the property

\[
v(r) = \begin{cases} 
\epsilon & \text{if } r \text{ is nullable} \\
\emptyset & \text{otherwise}
\end{cases}
\]

and is defined as follows:

\[
\begin{align*}
v(\epsilon) &= \epsilon \\
v(a) &= \emptyset \\
v(\emptyset) &= \emptyset \\
v(r \cdot s) &= v(r) \& v(s) \\
v(r + s) &= v(r) + v(s) \\
v(r^*) &= \epsilon \\
v(r \& s) &= v(r) \& v(s) \\
v(\neg r) &= \begin{cases} 
\epsilon & \text{if } v(r) = \emptyset \\
\emptyset & \text{if } v(r) = \epsilon
\end{cases}
\end{align*}
\]
The following rules, owed to Brzozowski (1964), compute the derivative of an RE with respect to a symbol $a$:

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

The rules are extended to strings as follows:

\[
\begin{align*}
\partial_x r &= r \\
\partial_u a r &= \partial_a (\partial_u r)
\end{align*}
\]

### 3.2 Using derivatives for RE matching

Suppose we are given an RE $r$ and a string $u$, and we want to determine that $u \in L[r]$. We have $u \in L[r]$ if, and only if, $\varepsilon \in L[\partial_u r]$, which is true exactly when $\varepsilon = \nu(\partial_u r)$. Combining this fact with the definition of $\partial_u$ leads to an algorithm for testing if $u \in L[r]$. We express the algorithm in terms of the relation $r \sim u$ ($r$ matches the string $u$), defined as the smallest relation satisfying

\[
\begin{align*}
r \sim \varepsilon &\iff \nu(r) = \varepsilon \\
r \sim a \cdot w &\iff \partial_a r \sim w
\end{align*}
\]

It is straightforward to show that $r \sim u$ if, and only if, $u \in L[r]$.

Notice that when an RE matches a string, we compute a derivative for each of the characters in the string. For example consider the derivation of $a \cdot b^* \sim abb$:

\[
\begin{align*}
a \cdot b^* \sim abb &\iff \partial_a a \cdot b^* \sim bb \\
&\iff b^* \sim bb \\
&\iff \partial_b b^* \sim b \\
&\iff b^* \sim b \\
&\iff b^* \sim \varepsilon \\
&\iff \nu(b^*) = \varepsilon
\end{align*}
\]
When the RE does not match the string, we reach a derivative that is the RE $\emptyset$ and stop. For example

$$a \cdot b^* \sim aba \iff \partial_a a \cdot b^* \sim ba \iff b^* \sim ba \iff \partial_b b^* \sim a \iff b^* \sim a \iff \partial_a b^* \sim \varepsilon \iff \emptyset \sim \varepsilon \iff \nu(\emptyset) = \varepsilon \text{ (false)}$$

3.3 Using derivatives for DFA construction

Before describing DFA construction, we need another definition.

**Definition 3.2**

We say that $r$ and $s$ are equivalent, written $r \equiv s$, if $L[r] = L[s]$. We write $[r]$ for the set \{ $s \mid r \equiv s$ \}, which is the equivalence class of $r$ under $\equiv$.

For example $a + b \equiv b + a$.

The matching relation gives an algorithm for testing a string against an RE by computing successive derivatives of the RE for successive characters in the string. At each step we have a residual RE that must match a residual string. If, instead of computing the derivatives on the fly, we precompute the derivative for each symbol in $\Sigma$, we can construct a DFA recogniser for the language of the RE. The states of the DFA are RE equivalence classes, and the transition function is the derivative function on those classes: $\delta(q, [a]) = [\partial_a(q)]$. This function is well defined because the derivatives of equivalent REs are equivalent. In constructing the DFA, we label each state with an RE representing its equivalence class. Accepting states are those states labelled by nullable REs, and the error state is labelled by $\emptyset$. The key challenge in making this algorithm practical is developing an efficient test for RE equivalence. We will return to this point in the next section.

Figure 1 gives the complete algorithm for constructing a DFA $\langle Q, q_0, F, \delta \rangle$, using derivatives. The goto function constructs the transition from a state $q$ for when the symbol $c$ is encountered, while the explore function collects together all of the possible transitions from the state $q$. Together, these functions perform a depth-first traversal of the DFA's state graph while constructing it. Note that we test RE equivalence when checking to see if $q_c$ is a new state. Brzozowski (1964) proved that an RE can only have finitely many derivatives (up to RE equivalence), which guarantees the termination of the algorithm. Once the state graph, represented by the $(Q, \delta)$ pair, has been constructed, it is simple to compute the accepting states and construct the DFA 4-tuple.
Consider the RE $a \cdot b + a \cdot c$ over the alphabet $\{a, b, c\}$. The DFA construction for this RE starts with $q_0 = \partial_e (a \cdot b + a \cdot c) = a \cdot b + a \cdot c$ and proceeds as follows:

1. compute $\partial_a q_0 = \partial_a (a \cdot b + a \cdot c) = b + c$, which is new and so is called $q_1$;
2. compute $\partial_a q_1 = \partial_a (b + c) = \emptyset$, which is new and so is called $q_2$;
3. compute $\partial_a q_2 = \partial_a \emptyset = \emptyset = q_2$;
4. likewise $\partial_b q_2 = q_2$ and $\partial_c q_2 = q_2$;
5. compute $\partial_b q_1 = \partial_b (b + c) = (\epsilon + \emptyset) \equiv \epsilon$, which is new and so is called $q_3$;
6. compute $\partial_a q_3 = \partial_a \epsilon = \emptyset = q_2$;
7. likewise $\partial_b q_3 = q_2$ and $\partial_c q_3 = q_2$;
8. compute $\partial_c q_1 = \partial_c (b + c) = (\emptyset + \epsilon) \equiv \epsilon = q_3$;
9. compute $\partial_b q_0 = \partial_b (a \cdot b + a \cdot c) = \emptyset = q_2$; and
10. compute $\partial_c q_0 = \partial_c (a \cdot b + a \cdot c) = \emptyset = q_2$.

Note that since $v(q_3) = \epsilon$, $q_3$ is an accepting state. Figure 2 shows the resulting DFA in graphical form.

---

**3.4 An example**

Consider the RE $a \cdot b + a \cdot c$ over the alphabet $\{a, b, c\}$. The DFA construction for this RE starts with $q_0 = \partial_e (a \cdot b + a \cdot c) = a \cdot b + a \cdot c$ and proceeds as follows:

1. compute $\partial_a q_0 = \partial_a (a \cdot b + a \cdot c) = b + c$, which is new and so is called $q_1$;
2. compute $\partial_a q_1 = \partial_a (b + c) = \emptyset$, which is new and so is called $q_2$;
3. compute $\partial_a q_2 = \partial_a \emptyset = \emptyset = q_2$;
4. likewise $\partial_b q_2 = q_2$ and $\partial_c q_2 = q_2$;
5. compute $\partial_b q_1 = \partial_b (b + c) = (\epsilon + \emptyset) \equiv \epsilon$, which is new and so is called $q_3$;
6. compute $\partial_a q_3 = \partial_a \epsilon = \emptyset = q_2$;
7. likewise $\partial_b q_3 = q_2$ and $\partial_c q_3 = q_2$;
8. compute $\partial_c q_1 = \partial_c (b + c) = (\emptyset + \epsilon) \equiv \epsilon = q_3$;
9. compute $\partial_b q_0 = \partial_b (a \cdot b + a \cdot c) = \emptyset = q_2$; and
10. compute $\partial_c q_0 = \partial_c (a \cdot b + a \cdot c) = \emptyset = q_2$.

Note that since $v(q_3) = \epsilon$, $q_3$ is an accepting state. Figure 2 shows the resulting DFA in graphical form.
4 Practical DFA construction

While the algorithm given in Figure 1 is simple, we are faced with three issues that must be addressed to build an efficient implementation:

1. The problem of determining when two REs are equivalent, which is used to test if \( q' \equiv q_c \) in the goto function, is expensive. In fact, deciding language equality for REs with intersection and complement operators is of non-elementary complexity (Aho et al., 1974).

2. The iteration over the symbols in \( \Sigma \) that is used to compute the \( \delta \) function is not practical for large alphabets. (For example, the Unicode character set has over 1.1 million code points.)

3. A scanner generator typically takes a collection of REs as its input specification, whereas the algorithm in Figure 1 builds a DFA for a single RE.

These issues are addressed in the next three subsections.

4.1 Weaker notions of RE equivalence

The DFA construction algorithm in Figure 1 only introduces a new state when no equivalent state is present. Brzozowski (1964) proved that this check for state equivalence guarantees the minimality of the DFA produced by the algorithm, but checking RE equivalence is expensive; so in practice we change the test to

\[ \exists q' \in Q \text{ such that } q' \approx q_c \]

where \( \approx \) is an approximation of RE equivalence that is defined as follows:

**Definition 4.1**

Let \( \approx \) denote the least relation on REs, including the following equations:

\[
\begin{align*}
   r \& r &\approx r & \quad (\ast) & \quad r + r &\approx r \\
   r \& s &\approx s \& r & \quad (\ast) & \quad r + s &\approx s + r \\
   (r \& s) \& t &\approx r \& (s \& t) & \quad (\ast) & \quad (r + s) + t &\approx r + (s + t) \\
   \emptyset \& r &\approx \emptyset & & -\emptyset + r &\approx -\emptyset \\
   -\emptyset \& r &\approx r & & \emptyset + r &\approx r \\
   (r \cdot s) \cdot t &\approx r \cdot (s \cdot t) & & (r^*)^* &\approx r^* \\
   \emptyset \cdot r &\approx \emptyset & & \varepsilon^* &\approx \varepsilon \\
   r \cdot \emptyset &\approx \emptyset & & \emptyset^* &\approx \varepsilon \\
   \varepsilon \cdot r &\approx r & & -(\neg r) &\approx r \\
   r \cdot \varepsilon &\approx r \\
\end{align*}
\]

Two REs \( r \) and \( s \) are **similar** if \( r \approx s \) and **dissimilar** otherwise.

**Theorem 4.1**

If \( r \approx s \) then \( r \equiv s \); that is, similar REs are equivalent.

**Proof**

By induction on the rules defining similarity. The non-inductive cases are simple algebraic consequences of Definition 2.2. \( \square \)
Brzozowski (1964) proved that a notion of RE similarity including only the above rules marked with (\(\ast\)) is enough to ensure that every RE has only a finite number of dissimilar derivatives. Hence, DFA construction is guaranteed to terminate if we use similarity as an approximation for equivalence. In our experience, including only the marked rules results in very large machines, but using the full set yields the minimal machine in most cases (see Section 5).

In our implementations, we maintain the invariant that all REs are in \(\approx\)-canonical form and use structural equality to identify equivalent REs. To ensure this invariant, we represent REs as an abstract type and use smart-constructor functions to build \(\approx\)-canonical forms. Each RE operator has an associated smart-constructor function that checks its arguments for the applicability of the \(\approx\) equations. If an equation applies, the smart constructor simplifies the RE using the equation as a reduction from left to right. For example the constructor for negation inspects its argument, and if it is of the form \((\neg r)\), the constructor simply returns \(r\).

For the commutativity and associativity equations, we use these equivalences to sort the subterms in lexical order. We also use this lexical order to implement a functional finite map with RE keys. This map is used as the representation of the set \(\mathcal{Q}\) of DFA states in Figure 1, where RE labels are mapped to states. The membership test \(q_c \in \mathcal{Q}\) is just a look-up in the finite map.

### 4.2 Character sets

The presentation of traditional DFA construction algorithms (Aho et al., 1986) involves iteration over the alphabet \(\Sigma\), and the derivative-based algorithm in Figure 1 does as well. Iteration over \(\Sigma\) is inefficient but feasible for small alphabets, such as the ASCII character set, but for large alphabets, such as Unicode (Unicode Consortium, 2003), iteration over \(\Sigma\) is impractical. Since the out degree of any given state is usually much smaller than the size of the alphabet, it is advantageous to label state transitions with sets of characters. In this section, we describe an extension to Brzozowski’s (1964) work that uses character sets to greatly reduce the number of derivatives that must be computed when determining the transitions from a given state.

The first step is to reformulate the abstract syntax of REs as follows:

\[
\begin{align*}
r, s & ::= \mathcal{S} \quad \text{where } \mathcal{S} \subseteq \Sigma \\
& | \varepsilon \quad \text{empty string} \\
& | r \cdot s \quad \text{concatenation} \\
& | r^* \quad \text{Kleene-closure} \\
& | r + s \quad \text{logical or (alternation)} \\
& | r \& s \quad \text{logical and} \\
& | \neg r \quad \text{complement}
\end{align*}
\]

Note that \(\mathcal{S}\) covers both the empty set and single character cases from Definition 2.1, as well as character classes. The definitions of Sections 2 and 3 extend naturally to
character sets:

\[
\mathcal{L}\left[\mathcal{F}\right] = \mathcal{F}
\]

\[
v(\mathcal{F}) = \emptyset
\]

\[
\partial_{a} \mathcal{F} = \begin{cases} 
\varepsilon & a \in \mathcal{F} \\
\emptyset & a \notin \mathcal{F}
\end{cases}
\]

As before, our implementation uses simplification to canonicalise REs involving character sets.

\[
R + S \approx T \text{ where } T = R \cup S
\]

\[
\neg S \approx T \text{ where } T = \Sigma \setminus S
\]

where \(R, S\) and \(T\) denote character sets.

As we remarked above, a given state \(q\) in a DFA will usually have many fewer distinct outgoing state transitions than there are symbols in \(\Sigma\). Let \(S_1, \ldots, S_n\) be a partition of \(\Sigma\) such that whenever \(a, b \in S_i\), we have \(\delta(q, a) = \delta(q, b)\) (equivalently \(\partial_a q \approx \partial_b q\)). If we somehow knew the partition \(S_1, \ldots, S_n\) for \(q\) in advance, we would only need to calculate one derivative per \(S_i\) when computing the transitions from \(q\). Note that if the derivatives are distinct, then the partition is minimal. This last situation is described by the following definition:

**Definition 4.2**

Given an RE \(r\) over \(\Sigma\) and symbols \(a, b \in \Sigma\), we say that \(a \simeq_r b\) if and only if \(\partial_a r \equiv \partial_b r\). The **derivative classes** of \(r\) are the equivalence classes \(\Sigma/\simeq_r\). We write \(\left[a\right]_r = \{b \mid a \simeq_r b\}\) for the derivative class of \(r\) represented by \(a\).

For example the derivative classes for \(a + b \cdot a + c\) are \(\{a, c\}, \{b\}\) and \(\Sigma \setminus \{a, b, c\}\).

Whenever two symbols belong to the same derivative class for two REs, those symbols belong to the same derivative class for any combination of the REs. This insight is formalised by the following lemma:

**Lemma 4.1**

Let \(r\) and \(s\) be REs and \(a\) and \(b\) symbols such that \(a \simeq_r b\) and \(a \simeq_s b\). Then the following equations hold:

- \(a \simeq_{(r \cdot s)} b\)
- \(a \simeq_{(r+s)} b\)
- \(a \simeq_{(r \& s)} b\)
- \(a \simeq_{\neg r} b\)
- \(a \simeq_{\neg s} b\)

**Proof**

The proof follows from simple equational reasoning. For example

\[
\partial_{a} (r \cdot s) \equiv \partial_{a} r \cdot s + v(r) \cdot \partial_{a} s
\]

\[
\equiv \partial_{b} r \cdot s + v(r) \cdot \partial_{b} s
\]

\[
\equiv \partial_{b} (r \cdot s)
\]

and thus \(a \simeq_{(r \cdot s)} b\). The other equations follow similarly. \(\square\)
We could determine the derivative classes of each state before finding any derivatives, but in general it is not possible to compute them without doing work $O(|\Sigma|)$. Instead, we define a function $C : \text{RE} \to 2^\Sigma$ by structural recursion that computes an approximation of the derivative classes. For atomic REs, $C$ gives an exact result:

$$C(\epsilon) = \{\Sigma\}$$

$$C(\emptyset) = \{\emptyset, \Sigma \setminus \emptyset\}$$

But compound REs are somewhat trickier. Lemma 4.1 provides guidance: if $a$ and $b$ are related in both $C(r)$ and $C(s)$, then they should also be related in $C(r \cdot s)$, etc. Our algorithm is conservative because it assumes that only those symbols that are related in both $C(r)$ and $C(s)$ are related in $C(r \cdot s)$ as specified by the following notation:

$$C(r) \land C(s) = \{S_r \cap S_s \mid S_r \in C(r), S_s \in C(s)\}$$

We can now define the remaining cases for $C$:

$$C(r \cdot s) = \begin{cases} C(r) & \text{if } r \text{ is not nullable} \\ C(r) \land C(s) & \text{otherwise} \end{cases}$$

$$C(r + s) = C(r) \land C(s)$$

$$C(r \& s) = C(r) \land C(s)$$

$$C(r^*) = C(r)$$

$$C(\neg r) = C(r)$$

Consider once more the example $a + b \cdot a + c$:

$$C((a + b \cdot a) + c) = C(a + b \cdot a) \land C(c)$$

$$= (C(a) \land C(b \cdot a)) \land C(c)$$

$$= (C(a) \land C(b)) \land C(c)$$

$$= (\{a\} \land \{b\} \setminus \{a\}) \land (\{c\} \land \{c\} \setminus \{c\})$$

$$= \{0, \{a\}, \{b\}, \Sigma \setminus \{a, b\}\} \land \{\{c\}, \Sigma \setminus \{c\}\}$$

$$= \{0, \{a\}, \{b\}, \{c\}, \Sigma \setminus \{a, b, c\}\}$$

As stated above, the exact derivative classes for this RE are $\{a, c\}$, $\{b\}$ and $\Sigma \setminus \{a, b, c\}$, and so the approximation has overpartitioned the alphabet. Nevertheless, we have reduced consideration to five symbol sets and need only compute one derivative for each set.

The correctness of the derivative class approximation is easy to prove.

Theorem 4.2
Let $r$ be an RE. Then for all $\mathcal{I} \in C(r)$ and $a \in \mathcal{I}$, we have $\mathcal{I} \subseteq [a]_r$.

Proof
By induction on the structure of $r$, using Lemma 4.1. \(\square\)
With the approximation of derivative classes, we can modify the algorithm for DFA construction to only compute one derivative per approximate class. This version of the algorithm is shown in Figure 3.

4.3 Regular vectors

In order to use this DFA construction algorithm in a scanner generator, we need to extend it to handle multiple REs in parallel. Brzozowski (1964) recognised this problem and introduced regular vectors as an elegant solution.

Definition 4.3
An \( n \)-tuple of REs, \( R = (r_1, \ldots, r_n) \), is called a regular vector.

Rather than labelling DFA states with REs, we now label them with regular vectors. The transition function is still just the derivative function, where the derivative of a regular vector is defined componentwise:

\[
\partial_a (r_1, \ldots, r_n) = (\partial_a r_1, \ldots, \partial_a r_n)
\]

The definitions for accepting and error states must also be revised. A state is accepting if its regular vector contains a nullable RE. The error state is the regular vector with components all equal to the empty language, \( \emptyset \). Finally, we can approximate the derivative classes of a regular vector by intersecting the approximate derivative classes of its components:

\[
C(r_1, \ldots, r_n) = \bigwedge C(r_i)
\]

5 Experience

We have experience with two independent implementations of RE-derivative-based scanner generators: \texttt{ml-u1ex}, which is an SML scanner generator developed at the
University of Chicago, and the PLT Scheme scanner generator. Both of these tools support extended REs and are being used on a regular basis.

5.1 Extended REs

The inclusion of the complementation operator in the RE language increases its ability to express natural and concise specifications. For example the following RE matches C-style comments, where a comment is started by the ‘/⋆’ sequence and ended by the first following ‘⋆/’ sequence (comment-opening sequences are ignored inside of comments; i.e. these comments do not nest):

\[ /⋆(Σ^*/Σ^*)⋆/ \]

The inner RE ‘(Σ^*/Σ^*)’ denotes the strings that contain the comment-ending sequence ‘⋆/’, and so its negation denotes the strings that do not contain the comment-ending sequence. Thus, the entire RE denotes strings that start with the comment-opening sequence and do not contain the comment-ending sequence except as the last two elements. Expressing this pattern without the complement operator is more cumbersome:

\[ /⋆((Σ\{⋆\})^*(ε + *((Σ\{/,⋆\})))^*⋆/ \]

One common use of the boolean operations on REs is to implement RE subtraction, i.e., \( r \& \neg s \) to denote the strings in \( L[r] \setminus L[s] \). For example, the DrScheme programming environment (Findler et al., 2002) highlights erroneous lexemes in red. To detect these lexemes, it uses the following RE:

\[(idchar)^+ \& \neg(identifier + number)\]

where \( idchar \) is the set of characters that can appear in an identifier; \( identifier \) is an RE-matching valid identifier; and \( number \) is an RE that matches numeric literals. The RE on the left of the ‘&’ includes all potential bad identifiers, but it also includes valid strings, such as valid identifiers and numbers. To match just the erroneous identifiers, we subtract the valid identifiers and numbers. In this example, the RE subtraction idiom removes the need to devise a positive definition of just the invalid lexemes. Such a definition would be exceptionally complex because of the nature of PLT Scheme’s lexical syntax. For example an identifier can start with the # character – but only when one of several specific strings immediately follow it.

5.2 DFA size

Our experience has been that using RE derivatives is a straightforward way to generate recognisers from REs. It also turns out that the use of RE derivatives produces smaller state machines than the algorithm used by tools like lex and ml-lex (Appel et al., 1994). We compared the size of the state machines generated by the ml-lex tool with those generated by our new ml-ulex tool. We also ran a DFA minimisation algorithm over the state machines generated by ml-ulex. As test cases, we used 14 pre-existing ml-lex specifications for various languages, a
Table 1. Number of states (best results in bold)

<table>
<thead>
<tr>
<th>Lexer</th>
<th>ml-lex</th>
<th>ml-ulex</th>
<th>Minimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burg</td>
<td>61</td>
<td>58</td>
<td>58</td>
<td>A tree-pattern match generator</td>
</tr>
<tr>
<td>CKit</td>
<td>122</td>
<td>115</td>
<td>115</td>
<td>ANSI C lexer</td>
</tr>
<tr>
<td>Calc</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>Simple calculator</td>
</tr>
<tr>
<td>CM</td>
<td>181</td>
<td>146</td>
<td>146</td>
<td>The SML/NJ compilation manager</td>
</tr>
<tr>
<td>Expression</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>A simple expression language</td>
</tr>
<tr>
<td>FIG</td>
<td>150</td>
<td>144</td>
<td>144</td>
<td>A foreign-interface generator</td>
</tr>
<tr>
<td>FOL</td>
<td>41</td>
<td>41</td>
<td>41</td>
<td>First-order logic</td>
</tr>
<tr>
<td>HTML</td>
<td>52</td>
<td>49</td>
<td>49</td>
<td>HTML 3.2</td>
</tr>
<tr>
<td>MDL</td>
<td>161</td>
<td>158</td>
<td>158</td>
<td>A machine-description language</td>
</tr>
<tr>
<td>ml-lex</td>
<td>121</td>
<td>116</td>
<td>116</td>
<td>The ml-lex lexer</td>
</tr>
<tr>
<td>Scheme</td>
<td>324</td>
<td>194</td>
<td>194</td>
<td>RS Scheme</td>
</tr>
<tr>
<td>SML</td>
<td>251</td>
<td>244</td>
<td>244</td>
<td>Standard ML lexer</td>
</tr>
<tr>
<td>SML/NJ</td>
<td>169</td>
<td>158</td>
<td>158</td>
<td>SML/NJ lexer</td>
</tr>
<tr>
<td>Pascal</td>
<td>60</td>
<td>55</td>
<td>55</td>
<td>Pascal lexer</td>
</tr>
<tr>
<td>ml-yacc</td>
<td>100</td>
<td>94</td>
<td>94</td>
<td>The ml-yacc lexer</td>
</tr>
<tr>
<td>Russo</td>
<td>4803</td>
<td>3017</td>
<td>2892</td>
<td>System-log data mining</td>
</tr>
<tr>
<td>$L_2$</td>
<td>n/a</td>
<td>147</td>
<td>106</td>
<td>Monitoring stress-test</td>
</tr>
</tbody>
</table>

This last example requires use of the boolean operations for concise specification; so we did not test the ml-lex tool on it. The results are presented in Table 1. In most cases, the RE derivative method produced a smaller state machine. Most of the time, the difference is small, but in two cases (Scheme and Russo), the ml-ulex DFAs have a third fewer states. Furthermore, ml-ulex produces the minimal state machine for every example except Russo, where the DFA is 4% larger than optimal, and $L_2$, where the DFA is 39% larger. In both of these cases, ml-lex did significantly worse.

The reason that the derivative approach produces smaller machines can be illustrated using a small example, but first we must give a quick description of the algorithm used by ml-lex. This algorithm was invented by McNaughton and Yamada (1960) and is described in the “Dragon Book” (Aho et al., 1986). It directly translates the abstract syntax tree (AST) representation of an RE to a DFA. The non-$\varepsilon$ leaves in the AST are annotated with unique positions, and sets of positions are used to represent the DFA states. Intuitively, if $a_i$ is a symbol in the RE and $i$ is in a state $q$, then there is a non-error transition from $q$ on $a$ in the DFA. The state

---

3 We adjusted the number of states reported by ml-lex downward by 2, because it includes the error state and a redundant initial state in its count, whereas ml-ulex reports only the non-error states.
transition from a state \( q \) on the symbol \( a \) s computed by

\[
\bigcup \text{Follow}(i) \text{ such that } i \in q \text{ and } a_i \text{ is in the RE}
\]

where \( \text{Follow}(i) \) is the set of positions that can follow \( a_i \) in a string matched by the RE. We demonstrate this algorithm on the following RE, which also illustrates why

the derivative algorithm produces smaller DFAs:

\[
(a_1c_2 + b_3c_4)S_5
\]

Here we have annotated each symbol with its position and denoted the position at the end of the RE by \( S_5 \). The initial state is \( q_0 = \{1,3\} \). The construction of the DFA proceeds as follows:

1. compute \( \delta(q_0,a) = \{2\} \), which is new and so is called \( q_1 \);
2. compute \( \delta(q_0,b) = \{4\} \), which is new and so is called \( q_2 \);
3. compute \( \delta(q_1,c) = \{5\} \), which is new and so is called \( q_3 \); and
4. compute \( \delta(q_2,c) = \{5\} \), which is \( q_3 \).

This construction produces the four-state DFA shown in Figure 4(a).

Now consider building a DFA for this RE, using the derivative algorithm. The first state is \( q_0 = \partial \varepsilon a \cdot c + b \cdot c = a \cdot c + b \cdot c \):

1. compute \( \delta(q_0,a) = \partial_a (a \cdot c + b \cdot c) = c \), which is new and so is called \( q_1 \);
2. compute \( \delta(q_0,b) = \partial_b (a \cdot c + b \cdot c) = c = q_1 \); and
3. compute \( \delta(q_1,c) = \partial_c c = \varepsilon \), which is new and so is called \( q_2 \).

This construction produces the smaller, three-state, DFA shown in Figure 4(b). As can be seen from this example, the use of positions in the Dragon Book algorithm causes equivalent states (i.e. \( q_1 \) and \( q_2 \) in the example) to be distinguished, whereas the use of canonical REs to label the states in the derivative algorithm allows their equivalence to be detected.

### 5.3 Effectiveness of character classes

We also used the above suite of lexer specifications to measure the usefulness of character classes. For a DFA with \( n \) states and \( m \) distinct state transitions, one has to compute at least \( m \) but no more than \( n|\Sigma| \) derivatives. We instrumented \texttt{ml-ulex} to count the number of distinct state transitions and the number of approximate character classes computed by our algorithm. In all but two cases (Scheme and \( L_2 \)), the approximation was perfect. In the two cases in which it was not perfect, our algorithm computed 5.4% and 6.2% more derivatives than necessary. What is more impressive is the number of derivatives that we avoid computing. If we assume the seven-bit ASCII character set as our input alphabet, then our algorithm computes only 2%–4% of the possible derivatives. Thus, we conclude that character classes provide a significant benefit in the construction of DFAs, even when the underlying alphabet is small.

---

\(^4\) For this exercise, we are ignoring the error state.
6 Related work

RE derivatives have been occasionally used to perform on-the-fly RE matching (without building automata) in XML validation tasks (English, 1999; Schmidt, 2002). Other than our systems, we know of at least two uses of derivatives in DFA construction. The first two versions of the Esterel language used derivatives, but the approach was abandoned in 1987 for being too memory intensive (Berry, 1999); furthermore, the REs and DFAs were not used for lexical analysis. More recently, Sen and Roşu (2003) used RE derivatives to construct DFAs for program trace monitoring. Their system generates minimal DFAs by testing full RE equivalence, using a technique called circular coinduction. This approach seems less practical than the approximate equivalence testing of our systems: for example they report that computing the optimal DFA for the $L_2$ RE mentioned in the previous section took 18 minutes, whereas ml-u1ex takes less than a second to compute a DFA that has only 40% more states than the optimal machine. The slowness of their approach may be owing to the fact that their method is based on rewriting, since even if we apply state minimisation to this example, ml-u1ex still takes less than a second to construct the optimal DFA.

Derivatives have largely been ignored by the scanning literature. One exception is a paper by Berry and Sethi (1986) that shows how a derivative-based algorithm for DFA construction can be used to derive the McNaughton and Yamada, a.k.a. the Dragon Book, algorithm (McNaughton & Yamada, 1960). The key difference between their work and Brzozowski’s (1964) derivative algorithm is that they mark each symbol in the RE with a unique subscript. These subscripts mean that states that Brzozowski’s (1964) algorithm would conflate are instead distinguished as illustrated...
in Figure 4. Ken Thompson, in his seminal paper on RE matching (Thompson, 1968), claims:

In the terms of Brzozowski, this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched.

This claim is true if one is computing derivatives for REs in which occurrences of symbols have been marked to distinguish them but not if one is using Brzozowski’s (1964) algorithm. Again, the example from Figure 4 can be used to illustrate this difference.

Berry and Sethi (1986) observed that the unmarking homomorphism does not commute with RE complement and intersection, and so algorithms based on marked symbols (e.g. the Dragon Book algorithm) cannot be easily modified to support these operations. On the other hand, since the complement of a DFA is simple to compute, the standard NFA to DFA construction can be extended to support RE complements. When the algorithm encounters a complemented RE \( \neg r \) it builds an NFA for \( r \) as usual, then converts the NFA to a DFA, which can be simply complemented and converted back to an NFA. The algorithm then proceeds as usual. The lexer generator for the DMS system (Baxter et al., 2004), supports complement in exactly this way.\(^5\) We are unaware of any other lexer generator that supports the complement operator.

7 Concluding remarks

In this paper, we have presented RE derivatives, which are an old, but largely forgotten, technique for constructing DFAs directly from REs. Our experience has been that RE derivatives are a superior technique for generating scanners from REs, and they should be in the toolkit of any programmer. Specifically, RE derivatives have the following advantages:

- They provide a direct RE to DFA translation that is well suited to implementation in functional languages.
- They support extended REs almost for free.
- The generated scanners are often optimal in the number of states and uniformly better than those produced by previous tools.

In addition to presenting the basic RE to DFA algorithm, we have also discussed a number of practical issues related to implementing a scanner generator that is based on RE derivatives, including supporting large character sets.

Acknowledgments

Comments from Russ Cox and the anonymous referees were quite helpful in improving the presentation. Aaron Turon’s work on ml-ulex at the University of Chicago was supported, in part, by NSF Grant CRI: Standard ML Software Infrastructure.

\(^5\) Personal correspondence, Michael Mehlich, 5 May 2004.
References


