Regular Expression Matching using Partial Derivatives

Martin Sulzmann
Informatik Consulting Systems AG
martin.sulzmann@gmail.com

Kenny Zhuo Ming Lu
Circos.com, Inc.
luzhuomi@gmail.com

Abstract

Regular expression matching is a classical and well-studied problem. Prior work applies DFA and Thompson NFA methods for the construction of the matching automata. We propose the novel use of derivatives and partial derivatives for regular expression matching. We show how to obtain algorithms for various matching policies such as POSIX and greedy left-to-right. Our benchmarking results show that the run-time performance is promising and that our approach can be applied in practice.

1. Introduction

Regular expression pattern matching is a natural generalization of pattern matching known from ML and Haskell. Here is an example written in a Haskell style language.

```
f :: (Space | Text)* -> Text*
f "" = ""
f (x::Space*, y::(Space | Text)*) = f y
f (x::Text+, y::(Space | Text)*) = x ++ f y
```

The data type `Text` refers to some alpha-numeric character and `Space` refers to white space. The above function removes all white space from the input string as specified by its type signature `f :: (Space | Text)* -> Text*`. For example, `f " Hello Bye"` yields `"HelloBye"`.

Removal of white space is achieved via the three pattern clauses which are applied from top to bottom. The first clause applies in case the input is empty. We use strings to represent sequences of white space and text. Therefore, the empty word is represented by the empty string `""`. In the second clause, the regular expression pattern `(x::Space*, y::(Space | Text)*)` matches any non-empty string (sequence) of white space and text. The point to note is that via the sub-patterns we can refer to sub-parts of the input string.

Variable `x` in `x::Space*` matches any string of white space and variable `y` in `y::(Space | Text)*` matches any remaining input symbol. For our example input, we find the matching `[x::" ", y::"Hello Bye"]`. But this is not the only possible matching because `Space*` matches also the empty string. Hence, the matching `[x::"", y::" Hello Bye"]` is possible as well.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

To make pattern matching unambiguous, we usually impose a specific pattern matching policy such as greedy or POSIX matching. This guarantees that only one matching results from a successful pattern match. For our running example, we assume a greedy matching policy. Therefore, we obtain the matching `[x::" ", y::"Hello Bye"]` which leads to the subsequent function call `f "Hello Bye"`. The last pattern clause is the only one applicable. Under greedy matching, we obtain `[x::"Hello", y::" Bye"]` which leads to yet another function call `f " Bye"` resulting in "Bye". We write `++` to denote string concatenation. Hence, we obtain the final result "HelloBye".

There are numerous prior works, e.g. consider [7, 4, 9, 8, 6, 5], which study regular expression pattern matching and their efficient implementation. Our contributions to this area is the novel use of regular expression derivatives [2] and partial derivatives [1] for regular expression pattern matching. Derivatives and partial derivatives are related to each other like DFAs and NFAs. The derivative-based algorithm has exponential run-time complexity due to backtracking, while the the partial derivative-based algorithm enjoys the optimal linear time complexity.

In summary, we make the following contributions:

- We extend the notion of derivatives of regular expressions to patterns and thus derive an elegant algorithm for regular expression pattern matching (Section 3).
- We show how to obtain a linear time complexity regular expression matching algorithm by employing partial derivatives (Section 4).
- We discuss how to obtain an optimized implementation with competitive performance results (Section 5).

All of our results are stated as propositions. We provide informal explanations but omit formal proofs which we plan to provide at some later stage. We will use Haskell as our executable specification language and for the implementation of all algorithms. Haskell is a natural choice because of the functional nature of the derivative-based approach towards pattern matching. The implementations are available via

`http://code.google.com/p/xhaskell-library/`

Related work is discussed in Section 6. Section 7 concludes.

2. Regular Expression Pattern Matching

We first formally introduce regular expression matching in Figure 1. The definition of words, regular expressions and languages is standard. Σ refers to a finite set of alphabet symbols A, B, etc. To avoid confusion with the EBNF symbol "|", we write "+" to denote the regular expression choice operator. The pattern language consists of variables, pair, choice and star patterns. Pattern variables x are always distinct. The treatment of extensions such as character classes, back-references is postponed until a later section. Environments are ordered multi-sets, i.e. lists. We write $\uplus$ to denote multi-

Words:

$$\begin{array}{rcll} w & ::= & \epsilon & \text{Empty word} \\ & | & l \in \Sigma & \text{Letters} \\ & | & ww & \text{Concatenation} \end{array}$$

Regular expressions:

$$\begin{array}{rcll} r & ::= & r + r & \text{Choice} \\ & | & (r, r) & \text{Concatenation} \\ & | & r^* & \text{Kleene star} \\ & | & \epsilon & \text{Empty word} \\ & | & \phi & \text{Empty language} \\ & | & l \in \Sigma & \text{Letters} \end{array}$$

Languages:

$$\begin{array}{lcl} L(r_1 + r_2) & = & L(r_1) \cup L(r_2) \\ L(r_1, r_2) & = & \{w_1 w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2)\} \\ L(r^*) & = & \{\epsilon\} \cup \{w_1 ... w_n | w_i \in L(r)\} \\ L(\epsilon) & = & \{\epsilon\} \\ L(\phi) & = & \{\} \\ L(l) & = & \{l\} \end{array}$$

Patterns:

$$\begin{array}{rcll} p & ::= & (x : r) & \text{Variables Base} \\ & | & (x : p) & \text{Variables Group} \\ & | & (p, p) & \text{Pairs} \\ & | & (p + p) & \text{Choice} \\ & | & p^* & \text{Kleene Star} \end{array}$$

Environments:

$$\begin{array}{rcll} \Gamma & ::= & \{x : w\} & \text{Variable binding} \\ & | & \Gamma \uplus \Gamma & \text{Ordered multi-set of variable bindings} \end{array}$$

Pattern matching relation: $w \vdash p \rightsquigarrow \Gamma$

$$\text{(VarBase)} \quad \frac{w \in L(r)}{w \vdash x : r \rightsquigarrow \{x : w\}} \qquad \text{(VarGroup)} \quad \frac{w \vdash p \rightsquigarrow \Gamma}{w \vdash x : p \rightsquigarrow \{x : w\} \uplus \Gamma}$$

$$\text{(Pair)} \quad \frac{\begin{array}{c} w = w_1 w_2 \\ w_1 \vdash p_1 \rightsquigarrow \Gamma_1 \\ w_2 \vdash p_2 \rightsquigarrow \Gamma_2 \end{array}}{w \vdash (p_1, p_2) \rightsquigarrow \Gamma_1 \uplus \Gamma_2} \qquad \begin{array}{c} \text{(ChoiceL)} \quad \dfrac{w \vdash p_1 \rightsquigarrow \Gamma_1}{w \vdash p_1 + p_2 \rightsquigarrow \Gamma_1} \\ \\ \text{(ChoiceR)} \quad \dfrac{w \vdash p_2 \rightsquigarrow \Gamma_2}{w \vdash p_1 + p_2 \rightsquigarrow \Gamma_2} \end{array} \qquad \text{(Star)} \quad \frac{\begin{array}{c} w = w_1 ... w_n \\ w_i \vdash p \rightsquigarrow \Gamma_i \quad \text{for } i = 1..n \end{array}}{w \vdash p^* \rightsquigarrow \Gamma_1 \uplus \ ... \uplus \ \Gamma_n}$$

Figure 1. Regular Expressions and Matching Relation

set union, i.e. list concatenation. The reason for using multi-sets rather than sets is that we record multiple bindings for a variable x. See the up-coming match rule for Kleene star patterns.

Concatenation among regular expressions and patterns is often left implicit and to omit parentheses we assume that $+$ has a lower precedence than concatenation. Hence, $A + AB$ is a short-hand for $A + (A, B)$ and $x : A + y : AB$ is a short-hand for $(x : A) + (y : AB)$.

Matching a pattern p against a word w is defined by the matching relation $w \vdash p \rightsquigarrow \Gamma$ which results in a binding Γ, mapping variables to matched sub-parts of the word. The matching relation as defined is indeterministic, i.e. ambiguous, for the following reasons.

In case of choice, we can arbitrarily match a word either against the left or right pattern. See rules (ChoiceL) and (ChoiceR). Indeterminism also arises in case of (Pair) and (Star) where the input word w can be broken up arbitrarily. Next, we consider some examples to discuss these points in more detail.

For pattern $(xyz : (x : A + y : AB + z : B)^*)$ and input ABA the following matchings are possible:

- $\{xyz : ABA, x : A, z : B, x : A\}$.
  In the first iteration, we match A (bound by x), then B (bound by z) , and then again A (bound by x). For each iteration step we record a binding and therefore treat bindings as lists. We write the bindings in the order as they appear in the pattern, starting with the left-most binding.
- $\{xyz : ABA, y : AB, z : B\}$.
  We first match AB (bound by y) and in the final last iteration then B (bound by z).

For pattern $(xyz : (xy : (x : A + AB, y : BAA + A), z : AC + C))$ and input $ABAAC$ we find the following matchings:

- $\{xyz : ABAAC, xy : ABAA, x : A, y : BAA, z : C\}$.
- $\{xyz : ABAAC, xy : ABA, x : AB, y : A, z : AC\}$.

To make the matching relation deterministic, we impose a pattern matching policy such as POSIX. POSIX demands that we must always match the longest word relative to the pattern structure. See Figure 2 where we introduce the POSIX rules (POSIX-Pair) and (POSIX-Star). All other rules from Figure 1 remain unchanged.

Word ordering:

$$
\begin{array}{lcl}
|w| & = & \text{length of word } w \\
w_1 \geq w_2 & = & |w_1| \geq |w_2| \\
(w_1, ..., w_n) \geq (w'_1, ..., w'_m) & = & (w_1 > w'_1) \vee (|w_1| == |w'_1| \wedge n > 1 \wedge (w_2, ..., w_n) \geq (w'_2, ..., w'_m))
\end{array}
$$

Free pattern variables:

$$
\begin{array}{lcl@{\qquad}lcl}
fv(x:r) & = & \{x\} & baseFv(x:r) & = & \{x\} \\
fv(x:p) & = & \{x\} \cup fv(p) & baseFv(x:p) & = & baseFv(p) \\
fv(p_1, p_2) & = & fv(p_1) \cup fv(p_2) & baseFv(p_1, p_2) & = & baseFv(p_1) \cup baseFv(p_2) \\
fv(p^*) & = & fv(p) & baseFv(p^*) & = & baseFv(p) \\
fv(p_1 + p_2) & = & fv(p_1) \cup fv(p_2) & baseFv(p_1 + p_2) & = & baseFv(p_1) \cup baseFv(p_2)
\end{array}
$$

POSIX matching:

$$
\text{(POSIX-Pair)}\quad
\frac{\begin{array}{c}
w = w_1 w_2 \\
w_1 \vdash_{POSIX} p_1 \rightsquigarrow \Gamma_1 \\
w_2 \vdash_{POSIX} p_2 \rightsquigarrow \Gamma_2 \\
\text{forall } w'_1, w'_2, \Gamma'_1, \Gamma'_2 \text{ such that} \\
w = w'_1 w'_2 \\
w'_1 \vdash_{POSIX} p_1 \rightsquigarrow \Gamma'_1 \\
w'_2 \vdash_{POSIX} p_2 \rightsquigarrow \Gamma'_2 \\
\text{we have that} \\
(w_1, w_2) \geq (w'_1, w'_2)
\end{array}}{w \vdash_{POSIX} (p_1, p_2) \rightsquigarrow \Gamma_1 \uplus \Gamma_2}
\qquad
\text{(POSIX-Star)}\quad
\frac{\begin{array}{c}
w = w_1 ... w_n \\
w_i \vdash_{POSIX} p \rightsquigarrow \Gamma_i \quad \text{for } i = 1..n \\
\text{forall } w'_1, ..., w'_m, \Gamma'_1, ..., \Gamma'_m \text{ such that} \\
w = w'_1 ... w'_m \\
w'_i \vdash_{POSIX} p \rightsquigarrow \Gamma'_i \quad \text{for } i = 1..m \\
\text{we have that} \\
(w_1, ..., w_n) \geq (w'_1, ..., w'_m)
\end{array}}{w \vdash_{POSIX} p^* \rightsquigarrow \Gamma_1 \uplus \ ... \uplus \ \Gamma_n}
$$

Greedy left-to-right matching:

$$
\text{(GLR-ChoiceL)}\quad
\frac{\begin{array}{c}
w \vdash_{glr2} p_1 \rightsquigarrow \Gamma_1 \\
fv(p_2) = \{x_1, .., x_n\} \\
\Gamma_2 = \{x_1 : \epsilon, ..., x_n : \epsilon\}
\end{array}}{w \vdash_{glr2} p_1 + p_2 \rightsquigarrow \Gamma_1 \uplus \Gamma_2}
$$

$$
\text{(GLR-ChoiceR)}\quad
\frac{\begin{array}{c}
w \vdash_{glr2} p_2 \rightsquigarrow \Gamma_2 \\
fv(p_1) = \{x_1, .., x_n\} \\
\Gamma_1 = \{x_1 : \epsilon, ..., x_n : \epsilon\}
\end{array}}{tw \vdash_{glr2} p_1 + p_2 \rightsquigarrow \Gamma_1 \uplus \Gamma_2}
$$

$$
\text{(GLR-Star)}\quad
\frac{\begin{array}{c}
w = w_1 ... w_n \\
w_i \vdash_{glr2} p \rightsquigarrow \Gamma_i \quad \text{for } i = 1..n \\
\text{forall } w'_1, ..., w'_m, \Gamma'_1, ..., \Gamma'_m \text{ such that} \\
w = w'_1 ... w'_m \\
w'_i \vdash_{glr2} p \rightsquigarrow \Gamma'_i \quad \text{for } i = 1..m \\
\text{we have that} \\
(\Gamma_1, ..., \Gamma_n) \geq (\Gamma'_1, ..., \Gamma'_m)
\end{array}}{w \vdash_{glr2} p^* \rightsquigarrow \Gamma_1 \uplus ... \uplus \Gamma_n}
$$

$$
\text{(GLR)}\quad
\frac{\begin{array}{c}
w \vdash_{glr2} p \rightsquigarrow \Gamma \\
\text{forall } \Gamma' \text{ such that } w \vdash_{glr2} p \rightsquigarrow \Gamma' \\
\text{we have that } \Gamma_o \geq \Gamma'_o \text{ where} \\
\Gamma_o = \{x : w | x : w \in \Gamma \quad x \in baseFv(p)\} \\
\Gamma'_o = \{x : w | x : w \in \Gamma' \quad x \in baseFv(p)\}
\end{array}}{w \vdash_{glr} p \rightsquigarrow \Gamma}
$$

Figure 2. Matching Policies

The premise in rule (POSIX-Pair) implies that there exists words w_1, w_2 and bindings Γ_1, Γ_2 such that the following conditions hold:

- $w = w_1 w_2$, and
- $w_1 \vdash_{POSIX} p_1 \rightsquigarrow \Gamma_1$, and
- $w_2 \vdash_{POSIX} p_2 \rightsquigarrow \Gamma_2$

That is, w_1 matches p_1 and w_2 matches p_2. The additional for all qualified conditions ensure that the first pattern p_1 is matched by the longest sub-part of w. Similarly, rule (POSIX-Star) demands that in each iteration we match the longest sub-word. For each iteration we record the binding and therefore use multi-sets, i.e. lists.

PROPOSITION 2.1 (POSIX Correctness). *Let w be a word, p be a pattern and Γ a binding such that $w \vdash_{POSIX} p \rightsquigarrow \Gamma$. Then, $w \vdash p \rightsquigarrow \Gamma$*

A different matching policy is greedy left-to-right matching. Judgment $\cdot \vdash_{glr2} \cdot \rightsquigarrow \cdot$ performs greedy left-to-right matching for all intermediate nodes. The rules for choice and star are replaced by rules (GLR-ChoiceL), (GLR-ChoiceR) and (GLR-Star). All other rules remain unchanged. In case of (GLR-ChoiceL), we

append some empty binding Γ_2 behind the left match Γ_1. In case of (GLR-ChoiceR), we append the empty binding Γ_1 ahead of the right match Γ_2. This guarantees that we cover all pattern variables even if they only contribute the empty binding and all bindings reflect the order of the variables in the pattern. This is the basis for the greedy left-to-right comparison in rules (GLR-Star) and (GLR).

Rule (GLR-Star) is similar to rule (POSIX-Pair). The difference is that we favor the earliest match and use the bindings Γ_i instead of words w_i for comparison. Thus, we select the greedy left-to-right match instead of only the longest match as in case of POSIX. The bindings reflect the greedy left-to-right matching order. We write $(\Gamma_1, ..., \Gamma_n) \geq (\Gamma'_1, ..., \Gamma'_m)$ as a short-hand for $(x_{1_1} : w_{1_1}, ..., x_{n_{l_n}} : w_{n_{l_n}}) \geq (x'_{1_1} : w'_{1_1}, ..., x'_{m_{l_m}} : w'_{m_{l_m}})$ where $(x_{i_j} : w_{i_j}) \in \Gamma_i$ and $(x'_{i_j} : w'_{i_j}) \in \Gamma'_i$. The sequence of variables x_{i_j} is a suffix of the sequence x'_{i_j}.

Rule (GLR) finally selects the greedy left-to-right match by only considering the base bindings resulting from $x : r$. See the use of $baseFv$ in the definitions. We write $\Gamma_o \geq \Gamma'_o$ as a short-hand for $(w_1, ..., w_n) \geq (v_1, ..., v_m)$ where $\Gamma_o = \{x_1 : w_1, ..., x_n : w_n\}$ and $\Gamma'_o = \{y_1 : v_1, ..., y_m : v_m\}$. The bindings $x_i : w_i$ and $y_j : v_j$ correspond to the leftmost matching order which implies that $x_1...x_n$ is a suffix of $y_1...y_m$.

PROPOSITION 2.2 (Greedy Left-To-Right Correctness). *Let w be a word, p be a pattern and Γ a binding such that $w \vdash_{glr} p \rightsquigarrow \Gamma$. Then, $w \vdash p \rightsquigarrow \Gamma'$ for some Γ' such that $\Gamma(x) = \Gamma'(x)$ for all $x \in dom(\Gamma')$.*

Because we also record empty bindings resulting from choice patterns, see rules (GLR-ChoiceL) and (GLR-ChoiceR), the greedy left-to-right binding Γ represents a superset of the binding Γ' computed via Figure 1. Therefore, we compare Γ and Γ' with respect to the variable bindings in Γ'. For convenience, we treat bindings like functions and write $dom(\Gamma')$ to denote the function domain of Γ'. The codomain is the power set over the language of words because of repeated bindings in case of the pattern star iteration. For instance, for $\Gamma'' = \{x : A, x : B\}$ we have that $\Gamma''(x) = \{A, B\}$.

Let's revisit our earlier examples. For pattern $(xyz : (x : A{+}y : AB + z : B)^*)$ and input ABA we have that

- $\{xyz : ABA, y : AB, z : B\}$ is the POSIX match, and
- $\{x : A, y : \epsilon, z : \epsilon, x : \epsilon, y : \epsilon, z : B, x : A, y : \epsilon, z : \epsilon\}$ is the greedy left-to-right match.

The repeated bindings for x, y and z arise because of the choice and Kleene star pattern. The above shows that POSIX strictly favors the longest match, regardless whether the match is in the left or right component of a choice pattern.

We consider an additional, new example. For pattern $(xy : (x : A + y : AA)^*)$ and input AA we find that

- $\{xy : AA, y : AA\}$ is the POSIX match, and
- $\{x : A, y : \epsilon, x : A, y : \epsilon\}$ is the greedy left-to-right match. In some intermediate step, we encounter the matchings $\{x : A, y : \epsilon, x : A, y : \epsilon\}$ and $\{x : \epsilon, y : AA\}$. Rule (GLR-Star) then selects $\{x : A, y : \epsilon, x : A, y : \epsilon\}$.

For pattern $(xyz : (xy : (x : A + AB, y : BAA + A), z : AC + C))$ and input $ABAAC$ we have that

- $\{xyz : ABAAC, xy : ABAA, x : A, y : BAA, z : C\}$ is the POSIX match, and
- $\{xyz : ABAAC, xy : ABA, x : AB, y : A, z : AC\}$ is the greedy left-to-right match.

The POSIX match respects the structure of the pattern. Hence, the first match is chosen where the binding $x : ABAA$ is longer than the binding $x : ABA$ in the second match. The pattern structure is ignored by the greedy left-to-right match which selects the match based on the base variables only. Hence, greedy chooses the second match.

Our next goal is to implement the POSIX and greedy left-to-right matching.

```
data RE where
  Phi :: RE                    -- empty language
  Empty :: RE                  -- empty word
  L :: Char -> RE              -- letter
  Choice :: RE -> RE -> RE     -- r1 + r2
  Seq :: RE -> RE -> RE        -- (r1,r2)
  Star :: RE -> RE             -- r*

derivRE :: RE -> Char -> RE
derivRE Phi _ = Phi
derivRE Empty _ = Phi
derivRE (L l1) l2
  | l1 == l2 = Empty
  | otherwise = Phi
derivRE (Choice r1 r2) l =
  Choice (derivRE r1 l) (derivRE r2 l)
derivRE (Seq r1 r2) l =
  if isEmpty r1
  then Choice (Seq (derivRE r1 l) r2) (derivRE r2 l)
  else Seq (derivRE r1 l) r2
derivRE (this@(Star r)) l =
  Seq (derivRE r l) this
```

Figure 3. Regular Expression Derivatives

3. Derivatives for Matching

In a first step, we implement the matching relation from Figure 1 by using Brzozowski's regular expression derivatives [2].

3.1 Regular Expression Derivatives

Derivatives provide for an elegant solution to the word problem:

$$lw \in L(r) \text{ iff } w \in L(r\backslash l)$$

where $r\backslash l$ is the derivative of r with respect to l. In language terms, we can specify derivatives as follows:

$$L(r\backslash l) \;=\; \{w \mid lw \in L(r)\}$$

Constructively, we obtain $r\backslash l$ from r by taking away the letter l while traversing the structure of r.. For example, $l\backslash l = \epsilon$ and $(r1 + r2)\backslash\, l = r1\backslash l + r2\backslash l$.

In Figure 3 we implement the $\cdot\backslash\cdot$ operation via the Haskell function `derivRE`. The Haskell data type `RE` is a literate translation of the expression syntax r from Figure 1.

The pair case (`Seq`), checks if the first component r_1 is empty or not. If empty, the letter l can be taken away from either r_1 or r_2. If non-empty, we take away l from r_1. In case of the Kleene star, we unfold r^* to (r, r^*) and take away the leading l from r.

3.2 Pattern Derivatives

Our idea is to transfer derivatives to the pattern matching setting:

$$lw \vdash p \rightsquigarrow \Gamma \;\text{ iff }\; w \vdash p\backslash l \rightsquigarrow \Gamma$$

Word lw matches the pattern p and yields environment Γ iff w matches the pattern derivative of p with respect to l.

The construction of pattern derivatives is similar to regular expressions. See Figure 4 for an implementation in Haskell. In case of a pattern variable, we build the derivative of the regular expression (base variable) or inner pattern (group variable). For convenience, we record the pattern match in the pattern itself by appending l to the already matched word w. The cases for choice

```
data Pat where
  PVar :: Int -> Word -> RE-> Pat
  PPair :: Pat -> Pat -> Pat
  PChoice :: Pat -> Pat -> Pat
  PStar :: Pat -> Pat
  PatVar :: Int -> Word -> Pat -> Pat

derivPat :: Pat -> Char -> Pat
derivPat (PVar x w r) l = PVar x (w ++ [l]) (derivRE r l)
derivPat (PPair p1 p2) l =
  if (isEmpty (strip p1))
  then PChoice (PPair (derivPat p1 l) p2)
               (PPair (mkEmpPat p1) (derivPat p2 l))
  else PPair (derivPat p1 l) p2
derivPat (PChoice p1 p2) l =
  PChoice (derivPat p1 l) (derivPat p2 l)
derivPat (PatVar x w p) l = PatVar x (w++[l]) (derivPat p l)
derivPat (this@(PStar p)) l = PPair (derivPat p l) this
```

Figure 4. Pattern Derivatives

```
strip :: Pat -> RE
strip (PVar _ w r) = r
strip (PPair p1 p2) = Seq (strip p1) (strip p2)
strip (PChoice p1 p2) = Choice (strip p1) (strip p2)
strip (PStar p) = strip p
strip (PatVar _ w p) = strip p

mkEmpPat :: Pat -> Pat
mkEmpPat (PVar x w r)
  | isEmpty r = PVar x w Empty
  | otherwise = PVar x w Phi
mkEmpPat (PPair p1 p2) = PPair (mkEmpPat p1) (mkEmpPat p2)
mkEmpPat (PChoice p1 p2) =
  PChoice (mkEmpPat p1) (mkEmpPat p2)
mkEmpPat (PatVar x w p) = PatVar x w (mkEmpPat p)
mkEmpPat (PStar p) = PStar (mkEmpPat p)

isEmpty :: RE -> Bool
isEmpty Phi = False
isEmpty Empty = True
isEmpty (L _) = False
isEmpty (Choice r1 r2) = (isEmpty r1) || (isEmpty r2)
isEmpty (Seq r1 r2) = (isEmpty r1) && (isEmpty r2)
isEmpty (Star r) = True
```

Figure 5. Helper Functions

and star are similar to the regular expression case. The pattern match for star records the binding for each iteration.

The pair case differs slightly compared to the regular expression case. The `strip` helper function, see Figure 5, extracts the regular expression to test if the first pattern p_1 is empty. If empty, all further matchings will only consider p_2. However, we can't simply drop p_1 because we record the variable binding in the pattern itself. Instead, we make the pattern empty such that the resulting pattern can't match any further input. See helper function `mkEmpPat`.

For example, consider the pattern

$$([AB]x : (A + B)^*, []y : C^*)$$

For convenience, we use the pattern syntax from Figure 1 instead of the more verbose Haskell data type syntax. Each variable starts with the already matched word. The first pattern has already consumed $[AB]$ while the binding of the second pattern is still empty represented by $[]$.

```
type Word = [Char]
type Env = [(Int,Word)]

allMatch :: Pat -> Word -> Pat
allMatch p w = foldl (\ p → \ l → derivPat p l) p w

allBinding :: Pat -> [Env]
allBinding (PVar x w r) =
  if isEmpty r then [[(x,w)]] else []
allBinding (PChoice p1 p2) =
(allBinding p1) ++ (allBinding p2 ) -- indet choice
allBinding (PPair p1 p2) =
  [ xs ++ ys | xs ← allBinding p1, ys ← allBinding p2 ]
allBinding (PatVar x w p) =
  [ (x,w):env | env ← allBinding p ]
allBinding (PStar p) = allBinding p

match :: Pat -> Word -> [Env]
match p w = allBinding (allMatch p w)
```

Figure 6. Derivative Matching

Computation of the pattern derivative with respect to C proceeds as follows.

$$\begin{aligned}
& \texttt{derivPat}\ ([AB]x : (A+B)^*, []y : C^*)\ \mathrm{C} \\
= \ & ([ABC]\ \texttt{derivPat}\ (x : (A+B)^*)\ C, []y : C^*) + \\
& ([AB]\ \texttt{mkEmpPat}\ (x : (A+B)^*), \texttt{derivPat}\ ([]y : C^*)\ C) \\
= \ & ([ABC]x : (\texttt{derivRE}\ (A+B)\ C, (A+B)^*), []y : C^*) + \\
& ([AB]x : \epsilon, [C]y : (\texttt{derivRE}\ C\ C, C^*)) \\
= \ & ([ABC]x : (\phi + \phi, (A+B)^*), []y : C^*) + \\
& ([AB]x : \epsilon, [C]y : (\epsilon, C^*))
\end{aligned}$$

The sub-pattern $(\phi + \phi, (A + B)^*)$ could be further simplified by ϕ and (ϵ, C^*) by C^*. We omit this implementation step for brevity.

3.3 Derivative Matching

Figure 6 puts the pieces together and implements the matching relation from Figure 1. Function `allMatch` iterates over the input word and applies the pattern derivative function. Function `allBinding` computes the bindings which are recorded in the final pattern.

PROPOSITION 3.1 (Pattern Derivative Soundness). *Let w be a word, p be a pattern and Γ a binding such that $w \vdash p \rightsquigarrow \Gamma$. Then,* `match p w = envs` *for some* `envs` *and* `env` *such that*

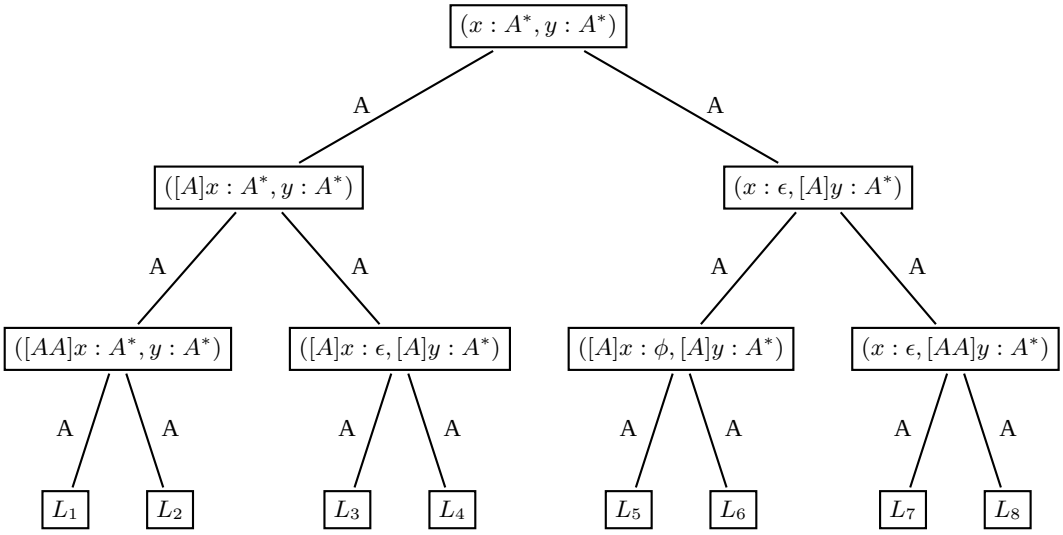
- `env` *is an element of* `envs`*, and*
- *for all $x \in dom(\Gamma)$ we have that* `lookup x env = Just w'` *where $w' = \Gamma(x)$.*

PROPOSITION 3.2 (Pattern Derivative Completeness). *Let w be a word and p be a pattern. If* `match p w = envs` *then for all elements* `env` *in* `envs` *we have that there exists Γ such that*

- *$w \vdash p \rightsquigarrow \Gamma$, and*
- *for all $x \in dom(\Gamma)$ we have that* `lookup x env = Just w'` *where $w' = \Gamma(x)$.*

The `lookup` function retrieves a variable binding. The constructor `Just` indicates that the lookup is successful.

The pattern computed by `allMatch` represents a tree of matchings. For example, consider pattern $(x : A^*, y : A^*)$ for input AAA. In Figure 7, we visualize the resulting pattern as a tree where every branch corresponds to a choice operator in the pattern derivative. Every leaf node corresponds to a potential match result. For convenience, we simplify (ϵ, A^*) by A^*.



$$
\begin{array}{lcl@{\qquad}lcl}
L_1 & = & ([AAA]x : A^*, []y : A^*) & L_2 & = & ([AA]x : \epsilon, [A]y : A^*) \\
L_3 & = & ([AA]x : \phi, [A]y : A^*) & L_4 & = & ([A]x : \epsilon, [AA]y : A^*) \\
L_5 & = & ([AA]x : \phi, [A]y : A^*) & L_6 & = & ([A]x : \phi, [AA]y : A^*) \\
L_7 & = & ([A]x : \phi, [AA]y : A^*) & L_8 & = & ([]x : \epsilon, [AAA]y : A^*)
\end{array}
$$

Figure 7. Match Tree Example

Function `allBinding` traverses the tree to build all valid variable bindings. For example, L_3 is invalid because of the non-empty ϕ pattern. Overall, we obtain the bindings

$$
\begin{array}{l}
\{x : AAA, y : \epsilon\} \\
\{x : AA, y : A\} \\
\{x : A, y : AA\} \\
\{x : \epsilon, y : AAA\}
\end{array}
$$

For most applications, we are not interested in all matchings but only in a specific, for example POSIX, match. We can select a specific match by traversing the tree in a certain order. For greedy left-to-right we choose a depth-first traversal. A problem with this approach is that we might have to back-track in case we hit an invalid leaf node. As the example in Figure 7 shows, back-tracking can take exponential time.

PROPOSITION 3.3 (Pattern Derivative Complexity). *The complexity of* `match` *is exponential in the size of the input in the worst case.*

To avoid back-tracking, and thus the exponential worst-case complexity, we explore several match paths simultaneously by using a non-deterministic automata for matching.

4. Partial Derivatives for Matching

For the non-deterministic match automata construction we make use of partial derivatives.

4.1 Regular Expression Partial Derivatives

Derivatives represent the states of a deterministic automata whereas partial derivatives introduced by Antimirov [1] represent the states of a non-deterministic automata. The partial derivative operation $\cdot\backslash_p\cdot$ yields a set of regular expressions via which we can express derivatives as follows.

$$L(r\backslash l) \;=\; L(r_1 + ... + r_n)$$

where $r\backslash_p l = \{r_1, ..., r_n\}$

```
partDeriv :: RE -> Char -> [RE]
partDeriv Phi l = []
partDeriv Empty l = []
partDeriv (L l') l
  | l == l' = [Empty]
  | otherwise = []
partDeriv (Choice r1 r2) l =
  nub ((partDeriv r1 l) ++ (partDeriv r2 l))
partDeriv (Seq r1 r2) l
  | isEmpty r1 =
      let s1 = [ (Seq r1' r2) | r1' <- partDeriv r1 l ]
          s2 = partDeriv r2 l
      in nub (s1 ++ s2)
  | otherwise = [ (Seq r1' r2) | r1' <- partDeriv r1 l ]
partDeriv (Star r) l =
  [ (Seq r' (Star r)) | r' <- partDeriv r l ]
```

Figure 8. Regular Expression Partial Derivatives

Partial derivatives are computed compositionally by traversing the structure of the regular expression. For example, consider the choice case

$$(r1 + r2)\backslash_p l = (r_1\backslash_p l) \cup (r_2\backslash_p l)$$

Figure 8 implements the operator $\cdot\backslash_p\cdot$ via the Haskell function `partDeriv`. The definition is similar to the `derivRE` function but we now put sub-results into a set instead of combining them via the choice operator $+$. We use lists to represent sets and therefore use the `nub` function to remove duplicate elements.

For expression A^* we find

$$\texttt{partDeriv}\ A^*\ A = [(\epsilon, A^*)]$$

which is equivalent to A^*. Our formulation of the partial derivative operation slightly departs from the formulation given in [1]. Antimirov immediately computes A^* as the partial derivative of A^*

```
data Pat where
  PVar :: Int -> RE -> Pat
  PPair :: Pat -> Pat -> Pat
  PChoice :: Pat -> Pat -> Pat
  PStar :: Pat -> Pat
  PatVar :: Int -> Pat -> Pat
  deriving Eq


pdPat :: Pat -> Char -> [(Pat,Env->Env)]
pdPat (PVar x r) l =
  let pds = partDeriv r l
  in if null pds then []
     else [(PVar x (resToRE pds),
          \ env -> update (x,l) env)]
pdPat (PPair p1 p2) l =
  if (isEmpty (strip p1))
  then nub2 ( [(PPair p1' p2,f) | (p1',f) <- pdPat p1 l] ++
     pdPat p2 l)
  else [ (PPair p1' p2,f) | (p1',f) <- pdPat p1 l ]
pdPat (PChoice p1 p2) l =
  nub2 ( (pdPat p1 l) ++ (pdPat p2 l))
pdPat (this@(PStar p)) l =
  [ (PPair p' this, f) | (p',f) <- pdPat p l]
pdPat (PatVar x p) l =
  [ (PatVar x p', f . (update (x,l))) | (p',f) <- pdPat p l]

update :: (Int,Char) -> Env -> Env
update (x,l) [] = [(x,[l])]
update (x,l) ((y,w):env)
  | (y == x) = (x,w++[l]) : env
  | otherwise = (y,w) : update (x,l) env

nub2 = nubBy (  (p1, _) (p2, _) -> p1 == p2)
```

Figure 9. Pattern Partial Derivatives with Matching Functions

with respect to A. This is a minor detail. Importantly, we can restate the following result already reported in [1].

PROPOSITION 4.1 (Antimirov). *For a finite alphabet Σ and regular expression r, the set of partial derivatives of r and its descendants is finite. The size of the set is linear in the size of the regular expression.*

The above result does not hold for derivatives. For example,

$$\begin{array}{lcl} \texttt{derivRE}\ A^*\ A & = & (\epsilon, A^*) \\ \texttt{derivRE}\ (\epsilon, A^*)\ A & = & (\phi, A^*) + (\epsilon, A^*) \end{array}$$

and so on. On the other hand, for partial derivatives we have that

$$\begin{array}{lcl} \texttt{partDeriv}\ A^*\ A & = & [(\epsilon, A^*)] \\ \texttt{partDeriv}\ (\epsilon, A^*)\ A & = & [(\epsilon, A^*)] \end{array}$$

we reach a fix-point. Note that $\texttt{partDeriv}\ \epsilon\ l = []$.

4.2 Pattern Partial Derivatives

We make use of the finiteness of partial derivatives to build a non-deterministic finite matching automaton. Each NFA transition goes from a regular expression pattern to the set of partial derivatives patterns. To each partial derivative we also associate a pattern matching function f.

$$p \xrightarrow{l} \{(p', f) \mid p' \in (p \backslash_p l)\} \tag{1}$$

The matching function f incrementally records that letter l is consumed by some pattern variable x. As we will see shortly, the final matching will be computed by composition of the incremental matchings.

Figure 9 shows the implementation of the NFA transition relation (1) in terms of the Haskell function `pdPat`. The construction of

```
isEmptyPat :: Pat -> Bool

greedy2:: [(Pat,Env)] -> Word -> [Env]
greedy2 ps [] =
  [ env | (p,env) <- ps, isEmptyPat p ]
greedy2 ps (l:w) =
  let ps2 = [ (p', f env) | (p,env) <- ps,
                            (p',f) <- pdPat p l ]
      ps3 = nub2 ps2
  in greedy2 ps3 w

greedy :: Pat -> Word -> Maybe Env
greedy p w =
  case (greedy2 [(p,[])] w) of
      env:_ -> Just env
      [] -> Nothing
```

Figure 10. Greedy Left-To-Right Matching

pattern partial derivatives follows closely the construction of regular expression derivatives. The cases for pattern variables are the only interesting ones. For `PVar` we build the partial derivative of the base regular expression r. The incremental matching function simply updates the current binding `env` by appending the letter `l`. The construction is similar for `PatVar`. In addition, we apply the incremental match f of the the sub-pattern p.

Antimirov's result straightforwardly transfers to the regular expression pattern setting.

PROPOSITION 4.2 (Finiteness of Pattern Partial Derivatives). *For a finite alphabet Σ and pattern p, the set of pattern partial derivatives of p and its descendants computed via function* `pdPat` *is finite. The size of the set is linear in the size of the pattern.*

The above allows us to build a finite, non-deterministic matching automata.

4.3 Greedy Left-To-Right Matching

The implementation of greedy left-to-right matching is given in Figure 10. For brevity, we omit the straightforward implementation of `isEmptyPat`. The pattern partial derivatives computed by `pdPat` are kept in left-to-right traversal order. We maintain this order while simultaneously exploring the paths of the non-deterministic matching automata. The `nub2` removes duplicate matching states in `ps2`. For removal of duplicates, we only consider the pattern component. An important property is that the `nubBy` (and `nub2`) function is stable. That is, equal elements are not re-ordered. The equality function among patterns is derived automatically, see `deriving Eq` attached to the `Pat` data type definition. Proposition 4.2 guarantees that the size of matching states in `ps` remains finite.

We can summarize the above observations as follows.

PROPOSITION 4.3 (Greedy Correctness and Complexity). *Function* `greedy` *implements the greedy left-to-right matching policy from Figure 2 and it's running time is linear in the size of the input.*

A feature of our implementation is that individual bindings of Kleene star iterations will be concatenated based on the pattern variables.

For example, for $(x : A, y : B)^*$ and input $ABAB$ we compute the final binding $[x : AA, y : BB]$. We could of course also compute the individual bindings for each iteration $[x : A, y : B, x : A, y : B]$. This requires a few modifications for `pdPat`, in particular, the case for `PStar`. The details can be found in the implementation which is available with this paper.

```
-- some adjustment because of
-- right-to-left match, append to front
update (x,l) [] = [(x,[l])]

-- some adjustment, keep only last, most recent match
pdPat (this@(PStar p)) l =
  [ (PPair p' this, f . (reset p)) | (p',f) <- pdPat p l]

-- remove earlier bindings of pat
reset :: Pat -> Env -> Env

geqEnv p e1 e2 =
  let xs = getVar p
  in geq (map (envToWord e1) xs)
         (map (envToWord e2) xs)

getVar :: Pat -> [Int]
envToWord :: Env -> [Int] -> [Word]

geq [] [] = EQ
geq (w1:ws1) (w2:ws2)
  | length w1 > length w2 = GT
  | length w1 == length w2 = geq ws1 ws2
  | otherwise = LT
geq _ _ = LT

getVar :: Pat -> [Int]
getVar (PVar x _) = [x]
getVar (PPair p1 p2) = getVar p1 ++ getVar p2
getVar (PChoice p1 p2) = getVar p1 ++ getVar p2
getVar (PStar p) = getVar p
getVar (PatVar x p) = x : getVar p

nubPosix :: Pat -> [(Pat,Env)] -> [(Pat,Env)]
nubPosix _ [] = []
nubPosix p ps =
  let peqss = groupBy ( \ (p1,_) (p2,_) -> p1 == p2) ps
  in map (maximumBy (geqEnv p)) peqsss

-- set of all pattern partial derivatives
allPD :: Pat -> [Char] -> [Pat]
-- set of letters in pattern
sigmaPat :: Pat -> [Char]

posix :: Pat -> Word -> Maybe Env
posix init v =
  let allPDs = allPD init (sigmaPat init)
      finals = [ (p,[]) | p <- allPDs, isEmptyPat p ]
      pMatch ps [] =
        [maximumBy (geqEnv init)
          [ env | (p,env) <- ps, p == init ]]
      pMatch ps (l:w) =
        let ps2 = [ (p',f env) | (p,env) <- ps,
                                 p' <- allPDs,
                                 (p'',f) <- pdPat p' l,
                                 p'' == p ]
            ps3 = nubPosix init ps2
        in pMatch ps3 w
  in case (pMatch finals (reverse v)) of
     [] -> Nothing
     (env:_) -> Just env
```

Figure 11. POSIX Right-To-Left Matching

4.4 POSIX Right-To-Left Matching

Implementing POSIX matching turns out to be more challenging. We can't rely anymore on a specific traversal strategy, e.g. leftmost, but must follow the POSIX matching order

Recall the earlier example $(xyz : (x : A + y : AB + z : B)^*)$. For input ABA, we find matchings

- $\{xyz : ABA, y : AB, x : A\}$, and
- $\{xyz : ABA, x : A, z : B, x : A\}$

The second one is the greedy left-to-right match and the first one is the POSIX match.

One possible strategy to compute the POSIX match is to keep track of the individual (incremental) matchings and after a subpattern match is complete to perform the POSIX check. See rule (POSIX-Star). But this strategy demands a lot of book-keeping which in turn requires extra space [7].

The key idea is to perform the POSIX match from right-to-left and only perform the POSIX check for the last, most recent match. We owe this insight to [4]. Explanations of why this approach works is missing in [4]. We finally provide some explanations using our running example $(xyz : (x : A + y : AB + z : B)^*)$.

We first build the automata derived from pattern partial derivatives. For convenience, we have slightly simplified the automata by simplifying (ϵ, r^*) to r^*. We also omit the incremental matching functions.

- States:

$$\begin{array}{lcl} p_1 & = & (xyz : (x : A + y : AB + z : B)^*) \\ p_2 & = & (xyz : (y : B, (x : A + y : AB + z : B)^*)) \end{array}$$

- Transitions:

$$\begin{array}{lcl} p_1 & \xrightarrow{A} & p_1 \\ p_1 & \xrightarrow{A} & p_2 \\ p_1 & \xrightarrow{B} & p_1 \\ p_2 & \xrightarrow{B} & p_1 \end{array}$$

Let's consider the greedy left-to-right (forward) match:

$$\begin{array}{lll} & [p_1] & \\ \xrightarrow{A} & [p_1, p_2] & \text{forward choice point} \\ \xrightarrow{B} & [p_1, p_1] & \text{forward conflict point} \end{array}$$

In the second step, we encounter a conflict. The second p_1 corresponds to the POSIX match which in our current scheme will be dropped. If we could foresee the future we could at the choice point already favor p_2 which leads to the POSIX match.

The idea by Cox is to perform the match from right-to-left (backward):

$$\begin{array}{lll} & [p_1] & \\ \xleftarrow{A} & [p_1] & \\ \xleftarrow{B} & [p_1, p_2] & \\ \xleftarrow{A} & [p_1, p_1] & \text{backward conflict point} \end{array}$$

We start off with the final state p_1. We consume the input from right-to-left. That is, we build a (backward) path from final to initial state. We yet again reach a conflict point.

The important insight, and something which hasn't been previously explained, is that the backward conflict point corresponds to the forward choice point (well, we are just slightly ahead by one step). Recall that in case of the forward choice point, we couldn't make any decision yet which duplicate to keep. We haven't seen the future yet and therefore must consume further input. Once we reach a forward conflict point, we can make a decision based on the recorded history of all matchings so far.

In case of the backward conflict point, we have already seen the future! Simply because we are consuming the input from right to left. That is, we can make a decision based on the current match. The current match is sufficient because the backward conflict point represents the latest conflict point. We mean here latest in the sense when viewed from left to right. Hence, there is no need to record a history of all matchings so far. We aggressively resolve conflicts

at the earliest possible point when going backwards (which is latest when going forward).

For our example, we find the following situation

$$[(p_1, [xyz : A]), (p_1, [xyz : AB])] \quad \text{backward conflict point}$$

In addition, we provide information about the incremental match so far. We only record the last, most recent match and focus on the top-most pattern variable. There is now sufficient information to decide that $(p_1, [xyz : AB])$ is the POSIX match which we shall keep.

Figure 11 implements this idea in Haskell. Because of the right-to-left (backward) match, we need to make some adjustments to the pattern partial derivative automata construction. The incremental match appends the letter l to the front. Function `reset` remove earlier bindings. That is, we only keep the last, most recent match. For brevity, we omit the straightforward implementation details.

Function `geqEnv` compares two bindings relative to the order of pattern variables. An important condition for our POSIX match approach to work is that each node in the pattern tree is annotated with a pattern variable. Function `getVar` extracts the pattern variable in top-down left-to-right order. We follow this order to select the POSIX match. See function `nubPosix` which removes all duplicate states with a smaller binding.

Function `posix` builds the POSIX match by traversing the input from right-to-left. For brevity, we omit the helper functions `allPD` and `sigmaPat` whose definitions are straightforward. In each intermediate (backward) step, we apply `nubPosix` to ensure that we keep the POSIX match. In the last step, we select the maximum match among all initial states.

In summary, we can conclude the following.

PROPOSITION 4.4 (POSIX Correctness and Complexity). *Each node in the pattern tree is annotated with a variable. Then, function* `posix` *implements the POSIX matching policy from Figure 2 and it's running time is linear in the size of the input.*

5. Experiments and Extensions

The matching algorithms in the previous sections are implemented in the most straightforward manner in Haskell. They contain space leaks and redundant computations and thus will not provide competitive performance. For example, the `posix` function repeatedly builds the reversed NFA transitions instead of caching them for faster access.

We discuss some optimizations to obtain an implementation which is competitive in terms of performance. We report some experimental results and also discuss extensions to deal with real-world applications of regular expression matching.

5.1 Optimization

The following are some optimization techniques that we adopt.

5.1.1 General Techniques

- Space leak - This is a common problem in many Haskell implementations. Because Haskell is lazy, the program memory can be swarmed by numerous unevaluated thunks. In our optimized version, we carefully apply the `seq` combinator to eliminate unwanted lazy-computation;
- Inefficiency of the `String` data type - In Haskell, `String` is implemented as a linked list of characters. It is a folklore problem that `String` value uses more space than its equivalent forms in other languages such as C and Java. In the optimized version we use `ByteString` [3] instead of `String`.

5.1.2 Specific Techniques

- `pdPat` - In both `greedy` and `posix` algorithms, we compute pattern partial derivatives "on the fly" via the `pdPat` function, (see in Figure 10 and Figure 11). These expensive operations are repeated in the presence of Kleene star. In the optimized version, we avoid this problem by pre-computing and caching pattern partial derivatives in a hash table. At run-time the `pdPat` operation is replaced by the hash operation followed by the look-up operation which then gives us $O(1)$ access.
- Indexed pattern partial derivatives - Another immediate performance gain we got from the above technique is that the pattern partial derivatives are hashed into integer values. Hence routines that require comparison among patterns are optimized in terms of equality test among integers;
- Pattern binding representation - In Figure 9 we store the (intermediate) pattern bindings in string form. In our optimized implementations, we use a `Range` data type, i.e. a pair of integers, to record only the starting and the ending positions of the bindings. Thus, the space required is greatly reduced, and the binding update operation is optimized. To keep track of the bindings in each iteration of Kleene star patterns, we use a list of `Rangess`.

5.2 Extensions for Real world Applications

Regular expression patterns used in the real world applications require some extensions. The regular expression syntax appearing in real world applications is often different from what we presented so far. For instance, patterns with sub-match binding is expressed implicitly in the regular expression pattern via groups, and the concatenation requires no constructor. In the following section, we use `p` (in text mode) to denote a pattern in the real world application syntax, and p (in math mode) to denote a pattern in our internal syntax defined earlier. The syntax of `p` will be explained by examples in the following paragraphs

5.2.1 Group Matching

In many mainstream languages that support regular expression pattern matchings, such as Perl, python, awk and sed, programmers are allowed to use "group operator", $(\cdot)$ to mark a sub-pattern from the input pattern, and the sub strings matched by the sub pattern can be retrieved by making reference to integer index of the group. For instance, `(a*)(b*)` is equivalent to pattern $(x : a^*, y : b^*)$ in our notation. Sending the input "aab" to `(a*)(b*)`. yields $["aa", "b"]$, where the first element in the list refers to the binding of the first group `(a*)` and the second element refers to the binding of the second group `(b*)`. Group matching is supported in our implementation by translating the groups into patterns with pattern variables.

5.2.2 Character Classes

Character class is another extension we consider. For instance, `[0-9]` denotes a single numeric character. `[A-Za-z]` denotes one alphabet character. We translate these two types of character classes into regular expressions via the choice operation $+$. There are some other type of character classes that require more work to support. Character classes can be negated. `[^0-9]` denotes any non-numeric character. Another related extension that is available in real world application is the dot symbol `.`, which can be used to represent any character. There are two different approaches to support the dot symbol and negative character classes. One approach is to translate the dot symbol into a union of all ASCII characters and to translate negative character classes to unions of all ASCII characters excluding those characters mentioned in the negated character classes. The other approach is to introduce these two notations `.`

and [^$l_1 ... l_n$] to our internal regular expression pattern language, such that

$$.\backslash_p l = \{\epsilon\}$$

$$[\hat{}\, l_1 ... l_n]\backslash_p l = \begin{cases} \{\epsilon\} & \text{if } l \in \{l_1, ..., l_n\} \\ \{\} & \text{otherwise} \end{cases}$$

In our implementation, we adopt the latter because the resulting regular expressions are smaller in size hence it is more efficient.

5.2.3 Non-Greedy Match

The symbol `?` in the pattern `(a*?)(a*)` indicates that the first sub pattern `a*` is matched non-greedily, i.e. it matches with the shortest possible prefix, as long as the suffix can be consumed by the sub pattern that follows.

Non-greedy matching can be neatly handled in our implementation. To obtain a non-greedy match for a pair pattern (p_1, p_2) where p_1 is not greedy, we simply reorder the two partial derivatives coming from $(p_1, p_2)\backslash_p l$. We extend the pair pattern case of `pdPat` in Figure 9 as follows,

```
pdPat (PPair p1 p2) l =
  if (isEmpty (strip p1))
  then if isGreedy p1
       then nub2 ([(PPair p1' p2,f) | (p1',f)<-pdPat p1 l]
            ++ pdPat p2 l)
       else nub2 (pdPat p2 l ++
            [(PPair p1' p2,f) | (p1',f) <- pdPat p1 l])
  else [ (PPair p1' p2,f) | (p1',f) <- pdPat p1 l ]
```

Extending our pattern language with the greediness symbol is straight-forward and the definition of `isGreedy` is omitted for brevity.

5.2.4 Anchored and Unanchored Match

Given a pattern `p`, `^p$` denotes an anchored regular expression pattern. The match is successful only if the input string is fully matched by `p`. A pattern which is not starting with `^` and not ending with `$` is considered unanchored. An unanchored pattern can match with any sub-string of the given input, under some matching policy. Our implementations `greedy` and `posix` are clearly the anchored matches. To support unanchored match, we could rewrite the unanchored pattern `p` into an equivalent anchored form, `^.*?p.*$`, and proceed with anchored match.

5.2.5 Repetition Pattern

Repetition patterns can be viewed as the syntactic sugar of sequence patterns with Kleene star. `p{m}` repeats the pattern `p` for `m` times; `p{n,m}` repeats the pattern `p` for at least `n` times and at maximum `m` times. It is obvious that the repetition pattern can be "compiled" away using the composition of sequence and Kleene star operators.

Other extensions such as unicode encoding and back references are not considered in this work.

5.3 Benchmarking

We benchmark ourselves against some existing regular expression libraries available in Haskell. The set of candidates for comparison are as follows,

- GLR - Our greedy left-to-right matching algorithm;
- GRL - A variant of the GLR algorithm, in which the input string is matched from right to left;
- POSIX(PD) - Our POSIX right-to-left matching algorithm;
- PCRE - Text.Regex.PCRE library [11], a wrapper around the C PCRE library;
- PCRELight - Text.Regex.PCRELight library [12];
- Parsec - Text.Regex.Parsec library [10];
- POSIX - Text.Regex.POSIX library [13];
- TDFA - Text.Regex.TDFA library [14].

The tests are conducted on an Intel Core 2 Duo machine running Mac OSX 10.5.8 with 4 giga-byte memory. The test programs are compiled using GHC 6.10.4. The runtime statistics are captured using the GHC run time flag `-sstderr`. The run times are measured in the granularity of seconds, and the memory usage is measured in mega-bytes.

For benchmarking, we use two typical applications of regular expression in real world computer system. The first regular expression pattern is

```
^(.*) ([A-Za-z]{2}) ([0-9]{5})(-[0-9]{4})?$
```

which validates whether an input string is a US address. For instance, sending the input string `"Mountain View, CA 90410"` to the above pattern yields a match `["Mountain View," , "CA", "90410", "" ]` .

The inputs we use are text files, which contain multiple lines of entries. Each entry is an address. The total numbers of entries of the input files are ranging from 100000 to 300000. Our main program (omitted for breivity) is reading and matching the input file line by line.

In Figure 12, the X-axis in the chart denotes the size of the input files. The Y-axis captures the time taken in the matchings, which is in logarithmic scale. As we can see from the chart, our implementations (especially GRL) are performing pretty well compared to other implementations. In Figure 13, we record the memory usage of the same example. In this graph, the Y-axis is measuring the the memory usage. The chart shows that the space efficiency of our implementation is about the same as that of Parsec, slightly less efficient than TDFA and more efficient than the the remaining three candidates.

In the second benchmark program, we consider the regular expression pattern

```
^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
```

which parses a HTTP request from a string which records the web server log. The input files that we used in this benchmark case are sharing the same number of entries. They only differ by the lengths of individual entries. For instance, every entry in the first input file has 200 characters, every entry in the second one has 220 characters, and so on. The run-time performances are shown in Figure 14. The Y-axis is measuring the run-time which is in logarithmic scale. The X-axis is measuring the length of the entries from the input files. In Figure 15, we benchmark the memory usage using the same example. In this example, our implementation is always in the middle-tier compared to the rest. We believe that there is definitely some room to improve the performance of our implementation.

6. Related Work and Discussion

Prior work relies on Thompson NFAs [15] for the construction of the matching automata. For example, Frisch and Cardelli [5] introduce a greedy matching algorithm. They first run the input from right-to-left to prune the search space. A similar approach is pursued in some earlier work by Kearns [6]. We adopt this idea in the GLR version of our greedy algorithm.

Laurikari [9, 8] devises a POSIX matching automata and introduces the idea of tagged transitions. A tag effectively corresponds to our incremental matching functions which are computed as part of `pdPat`.

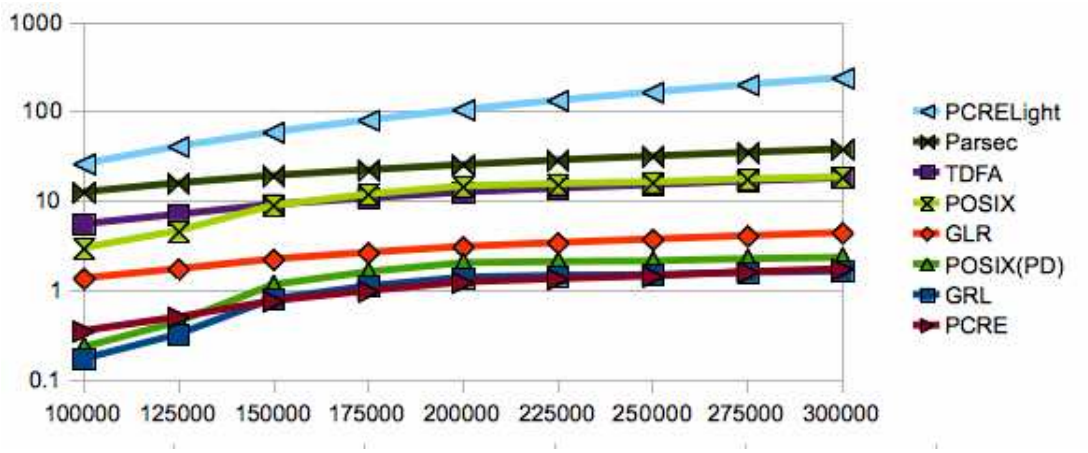


Figure 12. Time comparison using the US address example

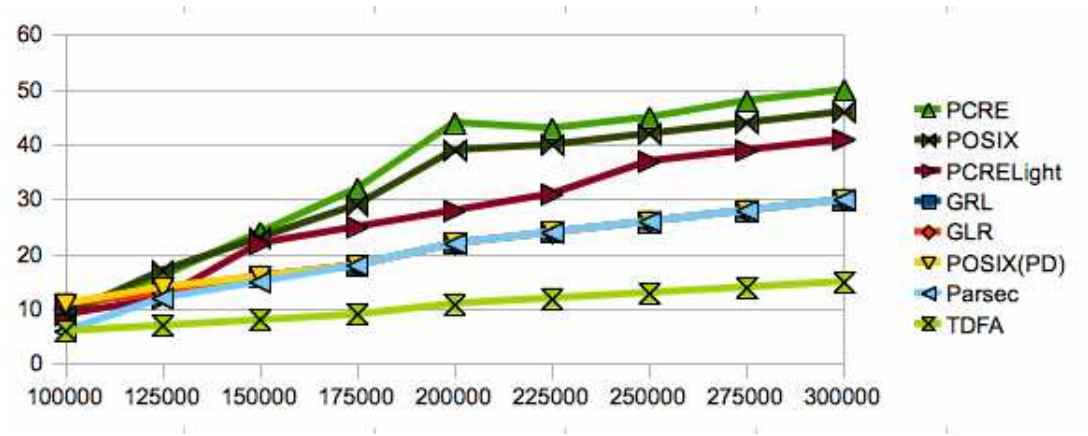


Figure 13. Space comparison using the US address example

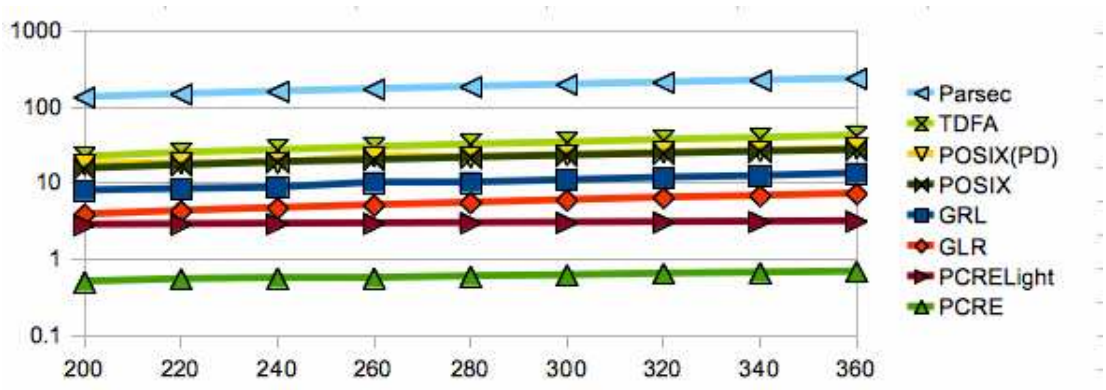


Figure 14. Time comparison using the HTTP request example

Kuklewicz has implemented Laurikari style tagged NFAs in Haskell. He [7] discusses various optimizations techniques to bound the space for matching histories which are necessary in case of (forward) left-to-right POSIX matching.

Cox [4] reports on a high-performance implementation of regular expression matching and also gives a comprehensive account of the history of regular expression match implementations. We refer to [4] and the references therein for further details. He introduces the idea of right-to-left scanning of the input for POSIX matching.

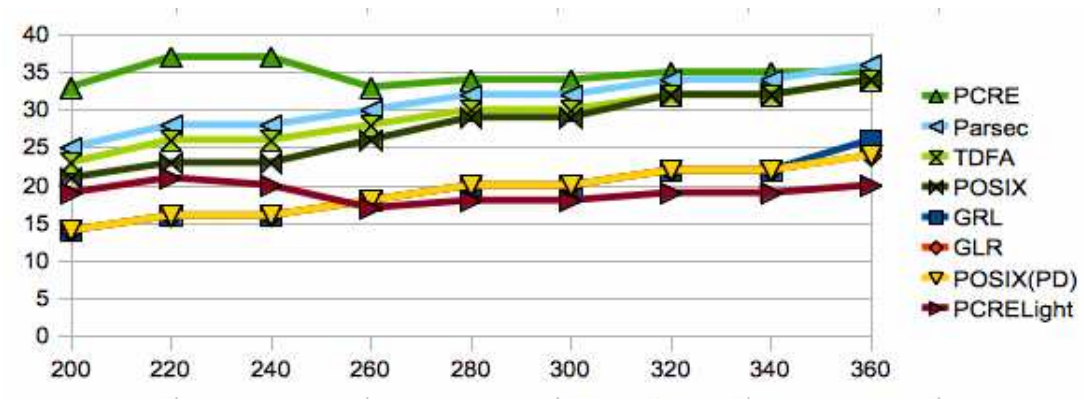


Figure 15. Space comparison using the HTTP request example

We adopt this idea to the setting of partial derivatives and provide informal explanations why this approach works.

As said, all prior work on efficient regular expression matching relies on Thompson NFAs or variants of it. To the best of our knowledge, we are the first to transfer the concept of partial derivatives to the regular expression setting. Partial deritivatives are a form of NFA with no ϵ-transitions. For a pattern of size n, the partial derivative NFA has $O(n)$ states and $O(n^2)$ transitions. Thompson NFAs have $O(n)$ states as well but $O(n)$ transitions because of ϵ-transitions.

The work in [5] considers ϵ-transitions as problematic for the construction of the matching automata. Laurikari [9, 8] therefore first removes ϵ-transitions whereas Cox [4] builds the ϵ-closure. Cox algorithm has a better theoretical complexity in the range of $O(n * m)$ where m is the input language. In each of the m steps, we must consider $O(n)$ transitions. With partial derivatives we cannot do better than $O(n^2 * m)$ because there are $O(n^2)$ transitions to consider. However, as shown in [1] the number of partial derivatives states is often smaller than the number of states obtained via other NFA constructions. Our performance comparisons indicate that partial derivatives are competitive. We leave a more detailed investigation of this topic for future work.

7. Conclusion

Our work tackles the regular expression matching problem from a different angle based on a novel application of regular expression derivatives and partial derivatives. We provide clean and elegant matching algorithms which can be used for educational purposes. Our benchmarks show that our approach yields competitive performance results. In future work, we plan to consider a more detailed study of the performance differences between matching automata build via Thompson NFAs and partial derivative NFAs.

References

[1] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[2] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[3] bytestring: Fast, packed, strict and lazy byte arrays with a list interface. `http://www.cse.unsw.edu.au/~dons/fps.html`.

[4] R. Cox. Regular expression matching in the wild, 2010. `http://swtch.com/~rsc/regexp/regexp3.html`.

[5] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. of ICALP'04*, pages 618– 629. Spinger-Verlag, 2004.

[6] S. M. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991.

[7] C. Kuklewicz. Forward regular expression matching with bounded space, 2007. `http://haskell.org/haskellwiki/RegexpDesign`.

[8] V. Laurikari. Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE*, pages 181–187, 2000.

[9] V. Laurikari. Efficient submatch addressing for regular expressions, 2001. Master thesis.

[10] regex-parsec: A better performance, lazy, powerful replacement of text.regex and jregex. `http://hackage.haskell.org/package/regex-parsec`.

[11] regex-pcre: The pcre backend to accompany regex-base. `http://hackage.haskell.org/package/regex-pcre`.

[12] pcre-light: A small, efficient and portable regex library for perl 5 compatible regular expressions. `http://hackage.haskell.org/package/pcre-light`.

[13] regex-posix: The posix regex backend for regex-base. `http://hackage.haskell.org/package/regex-posix`.

[14] regex-tdfa: A new all haskell tagged dfa regex engine, inspired by libtre. `http://hackage.haskell.org/package/regex-tdfa`.

[15] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.