Regular Expression Sub-Matching using Partial Derivatives

(Some errata, see accompanying talk (at the end): `http://www.home.hs-karlsruhe.de/~ suma0002/talks/ppdp12-part-deriv-sub-match-talk.pdf`**)**

Martin Sulzmann
Hochschule Karlsruhe - Technik und Wirtschaft
martin.sulzmann@gmail.com

Kenny Zhuo Ming Lu
Nanyang Polytechnic
luzhuomi@gmail.com

Abstract

Regular expression sub-matching is the problem of finding for each sub-part of a regular expression a matching sub-string. Prior work applies Thompson and Glushkov NFA methods for the construction of the matching automata. We propose the novel use of derivatives and partial derivatives for regular expression sub-matching. Our benchmarking results show that the run-time performance is promising and that our approach can be applied in practice.

Categories and Subject Descriptors F.1.1 [**Computation by Abstract Devices**]: Models of Computation—Automata; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—Operations on languages

General Terms Algorithms, Languages, Performance

Keywords Regular expression, Automata, Matching

1. Introduction

Regular expression matching is the problem of checking if a word matches a regular expression. For example, consider the word $ABAAC$ comprising of letters A, B and C and the regular expression $(A + AB)(BAA + A)(AC + C)$. Symbol $+$ denotes the choice operator. Concatenation is implicit. It is straightforward to see that $ABAAC$ matches the regular expression.

Specifically, we are interested in sub-matchings where for each sub-part of a regular expression we seek for a matching sub-word. To refer to sub-parts we are interested in, we annotate our regular expression with distinct variables:

$$(x_1 : (A + AB))(x_2 : (BAA + A))(x_3 : (AC + C))$$

These variables will be bound to the sub-matchings. For example, the first two letters AB of the input word $ABAAC$ will be bound to x_1, the third letter A to x_2 and the remainder AC to x_3.

Many of the real-world implementations of regular expression (sub)matching are very slow for even simple matching problems. For example, consider the pattern $A?^n A^n$ and the input string A^n where A^n stands for repeating the letter A n-times. As reported in [5], Perl shows some exponential behavior for this example because of its back-tracking matching algorithm. However, the running time of the matching algorithm can be linear in the size of the input string if proper automata-based methods are applied.

The works in [5, 12] advocate the use of Thompson NFAs [24]. The NFA non-deterministically searches for possible matchings without having to back-track. Thus, a linear running time can be guaranteed. There are several other NFA constructions which can serve as a basis to build the matching automata. For example, the work in [7] relies on Glushkov NFAs for finding (sub)matchings.

In this work, we propose the novel use of Brzozowski's regular expression derivatives [3] and Antimirov's partial derivative NFAs [2] for sub-matching which in our view leads to a particular elegant formalization of regular expression sub-matching. We obtain the proof of correctness of regular expression sub-matching by construction. The further advantage of partial derivatives is that on the average the partial derivative NFA is smaller than the Glushkov NFA. There are no ϵ-transitions compared to Thompson NFAs. We can thus build a highly efficient implementation in Haskell which is the fastest among all Haskell implementations of regular expression sub-matching we are aware of. Our implementation incorporates many of the extensions found in real world regular expressions and we are competitive compared to state-of-the art C-based implementations such as RE2 and PCRE.

In summary, we make the following contributions:

- We give a rigorous treatment of regular expression sub-matching (Section 3).
- We extend Brzozowski's regular expression derivatives [3] and Antimirov's partial derivative [2] to obtain algorithms which implement POSIX and greedy left-most matching (Sections 4 and 5).
- We give a comparison among Thompson, Glushkov and partial derivatives NFA approaches (Section 5.4).
- We show that our approach can support the many extensions typically found in real world regular expressions (Section 6).
- We have built an optimized implementation of regular expression sub-matching with partial derivatives and provide empirical evidence that our implementation yields competitive performance results (Section 7).

All our implementations, including reference implementations of greedy left-most matching using Thompson and Glushkov NFAs, are available via

`http://hackage.haskell.org/package/regex-pderiv`

Related work is discussed in Section 8. Section 9 concludes.

2. The Key Ideas

We motivate the key ideas of our regular expression sub-matching approach via some examples. Our starting point are Brzozowski's derivatives [3] which have recently been rediscovered for matching a word against a regular expression [16].

A word w matches a regular expression r if w is an element of the language described by r, written $w \in L(r)$. This problem can be elegantly solved via derivatives as follows. The idea is that

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

where $r\backslash l$ is the derivative of r with respect to l. In language terms, $L(r\backslash l) \;=\; \{w \mid lw \in L(r)\}$. Constructively, we obtain $r\backslash l$ from r by taking away the letter l while traversing the structure of r. We will shortly see examples explaining the workings of the derivative operator $(\cdot\backslash\cdot)$. To check that word $l_1 ... l_n$ matches regular expression r, we simply build $r\backslash l_1 \backslash ... \backslash l_n$ and test if this regular expression accepts the empty string.

Our idea is to transfer derivatives to the pattern sub-matching problem. Patterns p are regular expressions annotated with pattern variables as shown in the introduction. Variable environments Γ hold the bindings of these pattern variables. The derivative operation in this setting is as follows:

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

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 $p\backslash l$ is similar to regular expressions. The crucial difference is that we also take care of sub-matchings.

As an example we consider pattern $(x : A + y : AB + z : B)^*$ and the to be matched input AB. To be clear, the pattern's meaning is $((x : A) + (y : AB) + (z : B))^*$ but we generally avoid parentheses around pattern variable bindings. Next, we show the individual derivative steps where notation $p_1 \xrightarrow{l} p_2$ denotes that p_2 is the derivative of p_1 with respect to l. For the first step, we also show the intermediate steps indicated by subscript notation. We write $p_1 \xrightarrow{l}_i p_2$ to denote the ith intermediate step.

$$\begin{array}{ll} & (x : A + y : AB + z : B)^* \\ \xrightarrow{\epsilon}_1 & (x : A + y : AB + z : B)(x : A + y : AB + z : B)^* \\ \xrightarrow{\epsilon}_2 & (x_1 : A + y_1 : AB + z_1 : B)(x : A + y : AB + z : B)^* \end{array}$$

$$\begin{array}{ll} A \xrightarrow{A} \epsilon & x_1 : A \xrightarrow{A} x_1|A : \epsilon \\ AB \xrightarrow{A} B & y_1 : AB \xrightarrow{A} y_1|A : B \\ B \xrightarrow{A} \phi & z_1 : B \xrightarrow{A} z_1|A : \phi \end{array}$$

$$\begin{array}{ll} \xrightarrow{A}_3 & (x_1|A : \epsilon + y_1|A : B + z_1|A : \phi) \\ & (x : A + y : AB + z : B)^* \end{array}$$

For space reasons, we put the concatenated expressions $(x_1|A : \epsilon + y_1|A : B + z_1|A : \phi)$ and $(x : A + y : AB + z : B)^*$ below each other.

The purpose of the intermediate steps are: (1) We unfold the Kleene star, (2) generate for clarity fresh variables for each iteration and (3) we apply the derivative operation to each subcomponent. The sub-matchings are stored within in the sub-pattern. This saves us from keeping track of an additional variable environment which describes the current match. For example, $z_1|A : \phi$ denotes that z_1 is so far bound to A and ϕ (the empty regular expression) is the residue of the derivative operation.

We continue with step $\xrightarrow{B}$ starting with

$$\underbrace{(x_1|A : \epsilon + y_1|A : B + z_1|A : \phi)}_{p_1}\underbrace{(x : A + y : AB + z : B)^*}_{p_2}$$

In case of concatenated expressions $p_1 p_2$ the derivative operation is applied to the leading expression p_1. Hence, we find

$$\begin{array}{ll} & (x_1|A : \epsilon + y_1|A : B + z_1|A : \phi) \\ & (x : A + y : AB + z : B)^* \\ \xrightarrow{B} & (x_1|AB : \phi + y_1|AB : \epsilon + z_1|AB : \phi) \\ & (x : A + y : AB + z : B)^* \end{array} \tag{1}$$

For our example, the leading expression p_1 matches the empty word and therefore the derivative operation is also applicable to p_2. The individual steps are similar to the steps above, unrolling Kleene star etc. In p_1, we must replace sub-parts which match the empty word by ϵ, otherwise, ϕ. This is to ensure that any matches involving letter take place 'behind' p_1.

$$\begin{array}{ll} & (x_1|A : \epsilon + y_1|A : B + z_1|A : \phi) \\ & (x : A + y : AB + z : B)^* \\ \xrightarrow{B} & (x_1|A : \epsilon + y_1|A : B + z_1|A : \phi) \\ & (x_2|B : \phi + y_2|B : \phi + z_2|B : \epsilon) \\ & (x : A + y : AB + z : B)^* \end{array} \tag{2}$$

Both cases (1) and (2) are combined via choice.

$$\begin{array}{ll} & (x_1|A : \epsilon + y_1|A : B + z_1|A : \phi) \\ & (x : A + y : AB + z : B)^* \\ \xrightarrow{B} & \left(\begin{array}{l} (x_1|AB : \phi + y_1|AB : \epsilon + z_1|AB : \phi) \\ (x : A + y : AB + z : B)^* \end{array}\right) \\ & + \\ & \left(\begin{array}{l} (x_1|A : \epsilon + y_1|A : B + z_1|A : \phi) \\ (x_2|B : \phi + y_2|B : \phi + z_2|B : \epsilon) \\ (x : A + y : AB + z : B)^* \end{array}\right) \end{array}$$

We simplify the final pattern by removing parts which are connected to ϕ. The thus simplified pattern is

$$\begin{array}{c} (y_1|AB : \epsilon)(x : A + y : AB + z : B)^* \\ + \\ (x_1|A : \epsilon + y_1|A : B)(z_2|B : \epsilon)(x : A + y : AB + z : B)^* \end{array}$$

We can directly read out the sub-matchings and collect them in some variable environments. Of course, we only consider sub-matchings whose residue matches the empty word which applies here to all cases. Hence, the first match is $\Gamma_1 = \{y_1 : AB\}$ and the second match is $\Gamma_2 = \{x_1 : A, z_2 : B\}$.

To guarantee that there is a unique, unambiguous match, we impose a specific matching policy such as POSIX or greedy left-most as found in Perl. In the above example, the first match is the POSIX match whereas the second match is the greedy left-most match.

As we will see, the first match reported by our derivative matching algorithm is always the POSIX match. Obtaining the greedy left-most match via the derivative matching algorithm is non-trivial because the derivative operation maintains the structure of the pattern whereas the greedy left-most match policy effectively ignores the structure.

To obtain the greedy left-most match, we make use of Antimirov's partial derivatives [2]. Partial derivatives are related to derivatives like non-deterministic automata (NFAs) are related to deterministic automata (DFAs). Derivatives represent the states of a deterministic automata whereas partial derivatives represent the states of a non-deterministic automata. The partial derivative operation $\cdot\backslash_p\cdot$ yields a set of regular expressions partial derivatives. The connection to the derivative operation $\cdot\backslash\cdot$ in terms of languages is as follows:

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

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

One of Antimirov's important results is that the set of partial derivatives of a regular expression and its descendants is finite. For our running example $(x : A + y : AB + z : B)^*$ we obtain the partial derivatives

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

For example,

$$\begin{array}{lll} p_1\backslash_p A = \{ & (x : \epsilon)(x : A + y : AB + z : B)^*, \\ & (y : B)(x : A + y : AB + z : B)^* \} \end{array}$$

$$
\begin{array}{lll}
\text{(T1)} & p_1 \xrightarrow{x \mapsto A} & p_1 \\
\text{(T2)} & p_1 \xrightarrow{y \mapsto A} & p_2 \\
\text{(T3)} & p_1 \xrightarrow{z \mapsto B} & p_1 \\
\text{(T4)} & p_2 \xrightarrow{y \mapsto B} & p_1
\end{array}
$$

Figure 1. Transitions for $(x : A + y : AB + z : B)^*$

That is, choice '+' is broken up into a set. We can further simplify $(x : \epsilon)(x : A+y : AB+z : B)^*$ to p_1. Hence, after simplifications $p_1 \backslash_p A = \{p_1, (y : B)p_1\}$.

Partial derivatives p_1 and p_2 are states of an NFA where p_1 is the starting as well as final state. The NFA transitions are given in Figure 1. Each transition carries a specific match. Matches associated to transitions are incremental. For example, in case of (T3) the current input B will be added to the current match binding of z. The above represents a non-deterministic, finite match automata in style of Laurikari's NFAs with tagged transitions [12].

Via the partial derivative NFA match automata it is straightforward to compute the greedy left-most match. We follow NFA states, i.e. apply transitions, from left to right in the order as computed by the pattern partial derivative function $\cdot \backslash_p \cdot$. Each resulting state is associated with an environment. We label each environment with the corresponding state. The initial state has the empty environment $\{\}^{p_1}$. In each transition step, the match function is applied to the environment yielding an updated environment.

For input AB, we find the following derivation steps

$$
\begin{array}{ll}
\{p_1\} & \{\}^{p_1} \\
\xrightarrow{A} \{p_1, p_2\} & \{x : A\}^{p_1} \;\; \{y : A\}^{p_2} \\
\xrightarrow{B} \{p_1, p_1'\} & \{x : A, z : B\}^{p_1} \\
 & \{y : AB\}^{p_1'} \\
\rightarrow & \text{keep left-most match} \\
\{p_1\} & \{x : A, z : B\}^{p_1}
\end{array}
$$

In the second derivation step, we could reach p_1 from p_1 and p_2 but we only keep the first p_1 resulting from transition (T1) due to the left-to-right traversal order. The second p_1' is discarded. State p_1 is final. Hence, we obtain the greedy left-most match $\{x : A, z : B\}$.

To summarize, the derivative operation maintains the pattern structure whereas the partial derivative operation ignores the pattern structure by for example breaking apart choice. This becomes clear when consider derivative and partial derivative of p_1 w.r.t. A:

$$p_1 \backslash_p A = \{p_1, (y : B)p_1\}$$

$$p_1 \backslash A = (x|A : \epsilon + y|A : B)p_1$$

Under a greedy matching policy, we try to maximize the left-most match. As can be seen from our example, matching via derivatives still respects the original pattern structure and therefore we obtain the POSIX match. Partial derivatives break apart the original pattern structure. Therefore, we obtain the greedy left-most match as found in Perl.

Next, we formalize this idea.

3. Regular Expression Pattern Sub-Matching

Figure 2 defines the syntax of words, regular expressions, patterns and environments. 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. The treatment of extensions such as character classes, back-references

Words:

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

Regular expressions:

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

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:

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

Language:

$$
\begin{array}{lcl}
L(r_1 + r_2) & = & L(r_1) \cup L(r_2) \\
L(r_1, r_2) & = & \{w_1 w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2)\} \\
L(r^*) & = & \{\epsilon\} \cup \{w_1 ... w_n | i \in \{1, .., n\} \;\; w_i \in L(r)\} \\
L(\epsilon) & = & \{\epsilon\} \\
L(\phi) & = & \{\} \\
L(l) & = & \{l\}
\end{array}
$$

Figure 2. Regular Expressions Syntax and Language

is postponed until the later Section 6. Environments are ordered multi-sets, i.e. lists. We write $\uplus$ to denote multi-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 upcoming match rule for Kleene star patterns. The reason for using an ordered multi-set is that we will compare the matchings in the order they appear, e.g. from left-to-right.

Concatenation among regular expressions and patterns is often left implicit. That is, we may write the shorter form $r_1 r_2$ instead of (r_1, r_2). 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 $(x : A) + (y : AB)$.

Figure 3 defines regular expression matching in terms of $w \vdash p \rightsquigarrow \Gamma$ where word w and pattern p are input arguments and matching environment Γ is the output argument, 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.

$$\text{(VarBase)}\quad \frac{w \in L(r)}{w \vdash x : r \rightsquigarrow \{x : w\}}$$

$$\text{(VarGroup)}\quad \frac{w \vdash p \rightsquigarrow \Gamma}{w \vdash x : p \rightsquigarrow \{x : w\} \uplus \Gamma}$$

$$\text{(Pair)}\quad \frac{\begin{array}{c} w = w_1 w_2 \\ w_1 \vdash p_1 \rightsquigarrow \Gamma_1 \\ w_2 \vdash p_2 \rightsquigarrow \Gamma_2 \end{array}}{w \vdash (p_1, p_2) \rightsquigarrow \Gamma_1 \uplus \Gamma_2}$$

$$\text{(ChoiceL)}\quad \frac{w \vdash p_1 \rightsquigarrow \Gamma_1}{w \vdash p_1 + p_2 \rightsquigarrow \Gamma_1}$$

$$\text{(ChoiceR)}\quad \frac{w \vdash p_2 \rightsquigarrow \Gamma_2}{w \vdash p_1 + p_2 \rightsquigarrow \Gamma_2}$$

$$\text{(Star)}\quad \frac{\begin{array}{c} w = w_1 ... w_n \\ w_i \vdash p \rightsquigarrow \Gamma_i \quad \text{for } i = 1..n \end{array}}{w \vdash p^* \rightsquigarrow \Gamma_1 \uplus \ ... \uplus \ \Gamma_n}$$

Figure 3. Pattern matching relation: $w \vdash p \rightsquigarrow \Gamma$

- $\{xyz : ABA, y : AB, x : A\}$.
  We first match AB (bound by y) and in the final last iteration then A (bound by x).

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

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

Next, we formalize greedy left-most matching (like in Perl) followed by POSIX matching to obtain a deterministic matching relation.

3.1 Greedy Left-Most Matching

The greedy left-most matching strategy is implemented by Perl and by the PCRE library. We present here a formalization of greedy left-most matching in terms of our notation of patterns and regular expressions.

We start off by establishing some basic definitions. The first definition establishes an ordering relation $\geq$ among structured words w_1 and w_2 which are represented as tuples. For example, we wish that $ABB \geq (AB, B)$ but $AB \not\geq (AB, B)$. That is, the ordering relation $\geq$ favors the longest (matching) word, starting from left to right.

DEFINITION 1 (Ordering among Word Tuples). *Let $|w|$ denote the length, i.e. number of symbols, of word w. Then, we inductively define the (partial) ordering among tuples $(w_1, ..., w_n)$ of words as follows:*

- $w_1 \geq w_2$ *iff* $|w_1| \geq |w_2|$
- $(w_1, ..., w_n) \geq (w'_1, ..., w'_m)$ *iff*
  1. $|w_1| > |w'_1|$, *or*
  2. $|w_1| = |w'_1| \wedge n > 1 \wedge m > 1 \wedge$
     $(w_2, ..., w_n) \geq (w'_2, ..., w'_m))$

The ordering relation $\geq$ will only be applied on tuples $(w_1, ..., w_n)$ and $(w'_1, ..., w'_m)$ where flattening the tuples will either result in the same sequence of letters or one is a prefix of the other.

The next definition extends the ordering relation among structured words to an ordering among environments. We will write $(x_{i_j} : w_{i_j}) \in \Gamma_i$ to refer to each binding in $\Gamma_i = \{x_{i_1} : w_{i_1}, ..., x_{i_n} : x_{i_n}\}$.

DEFINITION 2 (Ordering among Environments). *Let $(x_{i_j} : w_{i_j}) \in \Gamma_i$ for $i \in \{1, ..., n\}$ and $(x'_{i_j} : w'_{i_j}) \in \Gamma'_i$ for $i \in \{1, ..., m\}$. The sequence of variables in $\Gamma'_1 \uplus ... \uplus \Gamma'_m$ is a prefix of the sequence of variables in $\Gamma_1 \uplus ... \uplus \Gamma_n$. Recall that bindings are ordered multi-sets. Then, $(\Gamma_1, ..., \Gamma_n) \geq (\Gamma'_1, ..., \Gamma'_m)$ iff $(w_{1_1}, ..., w_{n_{l_n}}) \geq (w'_{1_1}, ..., w'_{m_{l_m}})$.*

Environments are ordered multi-sets (lists) as well. Hence, the order of w_{i_j} and w'_{i_j} is fixed by the order of their corresponding environment variables x_{i_j}.

For comparison, we may only consider selected environment variables. We write $\Gamma_{|V}$ to restrict the environment to those variables mentioned in V. That is, $\Gamma_{|V} = \{x : w | x : w \in \Gamma \quad x \in V\}$. We write $(\Gamma_1, ..., \Gamma_n)_{|V}$ as a short-hand notation for $(\Gamma_{1_{|V}}, ..., \Gamma_{n_{|V}})$.

For all pattern variables we record the match in some environment. We compute those variables via function fv. See the upcoming definition. To decide which environment is the greedy left-most match, we only consider variables of base patterns $x : r$. We compute these variables via function $baseFv$.

DEFINITION 3 (Free Pattern Variables). *The set of free pattern variables is defined as follows:*

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

The set of free variables belonging to base patterns $x : r$ is defined as follows:

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

We have everything at hand to formalize greedy left-most matching in Figure 4. Judgment $\cdot \vdash_{lm} \cdot \rightsquigarrow \cdot$ performs the matching for all intermediate nodes. The rules strictly favor the left-most match as shown by rules (LM-ChoiceL) and (LM-ChoiceR). In case of (LM-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-most comparison in rules (LM-Star) and (LM).

In case of the Kleene star, we greedily follow the left-most matching policy. See rule (LM-Star). Recall that the bindings in Γ are ordered in the left-to-right matching order.

The top judgment $\cdot \vdash_{lm_{top}} \cdot \rightsquigarrow \cdot$ and rule (LM) finally select the greedy left-most match. For selection, we must only consider the base bindings resulting from sub-patterns $x : r$ because of the depth-first left-to-right nature of greedy left-most matching.

For example, $\{xyz : ABAAC, xy : ABA, x : AB, y : A, z : AC\}$ is the greedy left-most match for pattern $(xyz : (xy : (x : A + AB, y : BAA + A), z : AC + C))$ and input $ABAAC$.

A subtle point is that this is not strictly enough to ensure that we compute the 'Perl-style' greedy left-most match. The reason is that we do not require to fully annotate a pattern with variables.

For example, consider pattern $(x : (A + AB), y : (B + \epsilon))$ and input AB where we find that

$$AB \vdash_{lm_{top}} (x : (A + AB), y : (B + \epsilon)) \rightsquigarrow \{x : AB, y : \epsilon\}$$

$$\boxed{w \vdash_{lm} p \rightsquigarrow \Gamma}$$

(LM-VarBase)
$$\frac{w \in L(r)}{w \vdash_{lm} x : r \rightsquigarrow \{x : w\}}$$

(LM-VarGroup)
$$\frac{w \vdash_{lm} p \rightsquigarrow \Gamma}{w \vdash_{lm} x : p \rightsquigarrow \{x : w\} \uplus \Gamma}$$

(LM-Pair)
$$\frac{\begin{array}{c} w = w_1 w_2 \\ w_1 \vdash_{lm} p_1 \rightsquigarrow \Gamma_1 \\ w_2 \vdash_{lm} p_2 \rightsquigarrow \Gamma_2 \end{array}}{w \vdash_{lm} (p_1, p_2) \rightsquigarrow \Gamma_1 \uplus \Gamma_2}$$

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

(LM-ChoiceR)
$$\frac{\begin{array}{c} \text{there is no } \Gamma_1 \text{ s.t. } w \vdash_{lm} p_1 \rightsquigarrow \Gamma_1 \\ w \vdash_{lm} p_2 \rightsquigarrow \Gamma_2 \\ fv(p_1) = \{x_1, .., x_n\} \\ \Gamma_1' = \{x_1 : \epsilon, ..., x_n : \epsilon\} \end{array}}{w \vdash_{lm} p_1 + p_2 \rightsquigarrow \Gamma_1' \uplus \Gamma_2}$$

(LM-Star)
$$\frac{\begin{array}{c} w = w_1 ... w_n \\ w_i \vdash_{lm} p \rightsquigarrow \Gamma_i \quad \text{for } i = 1..n \\ \text{for all } (w_1', \Gamma_1'), ..., (w_m', \Gamma_m') \text{ such that} \\ \bullet w_1 ... w_n = w_1' ... w_m', \text{ and} \\ \bullet w_i' \vdash p \rightsquigarrow \Gamma_i' \text{ for } i = 1, ..., m \\ \text{we have that} \\ (\Gamma_1, ..., \Gamma_n)_{|baseFv(p)} \geq (\Gamma_1', ..., \Gamma_m')_{|baseFv(p)} \end{array}}{w \vdash_{lm} p^* \rightsquigarrow \Gamma_1 \uplus ... \uplus \Gamma_n}$$

(LM)
$$\frac{\begin{array}{c} w \vdash_{lm} p \rightsquigarrow \Gamma \\ \text{for all } \Gamma' \text{ such that } w \vdash_{lm} p \rightsquigarrow \Gamma' \\ \text{we have that } \Gamma_{|baseFv(p)} \geq \Gamma'_{|baseFv(p)} \end{array}}{w \vdash_{lm_{top}} p \rightsquigarrow \Gamma}$$

Figure 4. Greedy Left-Most Matching

This match arises because we do not look further inside the base pattern $x : (A + AB)$. However, this is not quite the Perl-style match which is $\{x : A, y : B\}$.

To obtain the Perl-style match, we must simply fully annotate the pattern with variables:

$$(x : (x_1 : A + x_2 : AB), y : (y_1 : B + y_2 : \epsilon))$$

By fully annotating the pattern we guarantee that the input letter A is matched against the left-most occurrence of A in $A + AB$. Thus, we obtain the desired match $\{x : A, x_1 : A, y : B, y_1 : B\}$.

We conclude this section by stating some elementary properties and also consider a few further examples.

PROPOSITION 3.1 (Greedy Left-Most Completeness). *Let w be a word, p be a pattern and Γ be a binding such that $w \vdash p \rightsquigarrow \Gamma$. Then, $w \vdash_{lm_{top}} p \rightsquigarrow \Gamma'$ for some Γ' such that $\Gamma(x) = \Gamma'(x)$ for all $x \in dom(\Gamma)$.*

PROPOSITION 3.2 (Greedy Left-Most Determinism). *Let w be a word, p be a pattern and Γ_1, Γ_2 be two bindings such that $w \vdash_{lm_{top}} p \rightsquigarrow \Gamma_1$ and $w \vdash_{lm_{top}} p \rightsquigarrow \Gamma_2$. Then, $\Gamma_1 = \Gamma_2$.*

$$\boxed{w \vdash_{POSIX} p \rightsquigarrow \Gamma}$$

(POSIX-VarBase)
$$\frac{w \in L(r)}{w \vdash_{POSIX} x : r \rightsquigarrow \{x : w\}}$$

(POSIX-VarGroup)
$$\frac{w \vdash_{POSIX} p \rightsquigarrow \Gamma}{w \vdash_{POSIX} x : p \rightsquigarrow \{x : w\} \uplus \Gamma}$$

(POSIX-ChoiceL)
$$\frac{w \vdash_{POSIX} p_1 \rightsquigarrow \Gamma}{w \vdash_{POSIX} p_1 + p_2 \rightsquigarrow \Gamma}$$

(POSIX-ChoiceR)
$$\frac{\begin{array}{c} \text{there is no } \Gamma_1 \text{ s.t. } w \vdash_{POSIX} p_1 \rightsquigarrow \Gamma_1 \\ w \vdash_{POSIX} p_2 \rightsquigarrow \Gamma \end{array}}{w \vdash_{POSIX} p_1 + p_2 \rightsquigarrow \Gamma}$$

(POSIX-Pair)
$$\frac{\begin{array}{c} w = w_1 w_2 \\ w_1 \vdash_{POSIX} p_1 \rightsquigarrow \Gamma_1 \\ w_2 \vdash_{POSIX} p_2 \rightsquigarrow \Gamma_2 \\ w_1, w_2 \text{ is the maximal word match} \end{array}}{w \vdash_{POSIX} (p_1, p_2) \rightsquigarrow \Gamma_1 \uplus \Gamma_2}$$

(POSIX-Star)
$$\frac{\begin{array}{c} w = w_1 ... w_n \\ w_i \vdash_{POSIX} p \rightsquigarrow \Gamma_i \quad \text{for } i = 1..n \\ w_1, ..., w_n \text{ is the maximal word match} \end{array}}{w \vdash_{POSIX} p^* \rightsquigarrow \Gamma_1 \uplus \ ... \uplus \ \Gamma_n}$$

Figure 5. POSIX Matching

PROPOSITION 3.3 (Greedy Left-Most Correctness). *Let w be a word, p be a pattern and Γ be a binding such that $w \vdash_{lm_{top}} p \rightsquigarrow \Gamma$. 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 (LM-ChoiceL) and (LM-ChoiceR), the greedy left-most binding Γ represents a superset of the binding Γ' computed via Figure 3. 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 co-domain 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 $\Gamma''(x) = \{A, B\}$.

It is easy to see that the greedy left-most match is stable under associativity of concatenation. Consider rule (LM-Pair) rule and the case $((p_1, p_2), p_3)$ versus $(p_1, (p_2, p_3))$. We have that $\uplus$ is associative and therefore for each case we obtain the same binding.

PROPOSITION 3.4 (Greedy Left-Most Associative Stability). *Let w be a word, p_1, p_2, p_3 be patterns and Γ and Γ' be bindings such that $w \vdash_{lm_{top}} ((p_1, p_2), p_3) \rightsquigarrow \Gamma$ and $w \vdash_{lm_{top}} (p_1, (p_2, p_3)) \rightsquigarrow \Gamma'$. Then, we have that $\Gamma = \Gamma'$.*

3.2 POSIX Matching

POSIX matching favors the longest word match w_1,...,w_n relative to some pattern structure p_1,...,p_n where each sub-word w_i matches sub-pattern p_i. We say that w_1,...,w_n is the maximal (longest) word match if for any other matching sequence w_1',...,w_n' we have that w_1',...,w_n' is smaller than w_1,...,w_n w.r.t. to the ordering relation $\geq$ among word tuples. The precise definition follows.

DEFINITION 4 (Maximal Word Match). *We say that w_1,...,w_n is the maximal (word) match w.r.t. patterns p_1,...,p_n and environment Γ_1,...,Γ_n iff*

$$
\begin{array}{lcl}
\phi\backslash l & = & \phi \\
\epsilon\backslash l & = & \phi \\
l_1\backslash l_2 & = & \begin{cases} \epsilon & \text{if } l_1 == l_2 \\ \phi & \text{otherwise} \end{cases} \\
(r_1 + r_2)\backslash l & = & r_1\backslash l + r_2\backslash l \\
(r_1, r_2)\backslash l & = & \begin{cases} (r_1\backslash l, r_2) + r_2\backslash l & \text{if } \epsilon \in L(r_1) \\ (r_1\backslash l, r_2) & \text{otherwise} \end{cases} \\
r^*\backslash l & = & (r\backslash l, r^*)
\end{array}
$$

Figure 6. Regular Expression Derivatives

1. $w_i \vdash p_i \rightsquigarrow \Gamma_i$ *for* $i = 1, ..., n$, *and*
2. *for all* $(w'_1, \Gamma'_1), ..., (w'_m, \Gamma'_m)$ *such that*
   - $w_1 ... w_n = w'_1 ... w'_m$, *and*
   - $w'_i \vdash p_i \rightsquigarrow \Gamma'_i$ *for* $i = 1, ..., m$

   we have that $(w_1, ..., w_n) \geq (w'_1, ..., w'_m)$.

PROPOSITION 3.5 (Maximal Word Match Existence and Uniqueness). *The maximal word match exists and is unique because the ordering relation among word matches and environment matches is well-founded.*

POSIX Matching favors the left-most match which respects the pattern structure. Figure 5 formalizes this requirement. The maximal word match condition in rule (POSIX-Pair) ensures 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.

We state some elementary properties about POSIX matching. The first property states that if there is a match there is also a POSIX match

PROPOSITION 3.6 (POSIX Completeness). *Let* w *be a word,* p *be a pattern and* Γ *a binding such that* $w \vdash p \rightsquigarrow \Gamma$. *Then, there exists* Γ' *such that* $w \vdash_{POSIX} p \rightsquigarrow \Gamma'$

Determinism of POSIX matching follows from the fact that the maximal match is unique and we favor the left-match in case of choice patterns.

PROPOSITION 3.7 (POSIX Determinism). *Let* w *be a word,* p *be a pattern and* Γ_1 *and* Γ_2 *be two bindings such that* $w \vdash_{POSIX} p \rightsquigarrow \Gamma_1$ *and* $w \vdash_{POSIX} p \rightsquigarrow \Gamma_2$. *Then,* $\Gamma_1 = \Gamma_2$.

A straightforward induction shows that every POSIX match is still a valid match w.r.t the earlier indeterministic matching relation in Figure 3.

PROPOSITION 3.8 (POSIX Correctness). *Let* w *be a word,* p *be a pattern and* Γ *a binding such that* $w \vdash_{POSIX} p \rightsquigarrow \Gamma$. *Then,* $w \vdash p \rightsquigarrow \Gamma$

Unlike greedy left-most, POSIX matching is not stable under associativity of concatenation. For example, $\{x : A, y : BAA, z : C\}$ is the POSIX match for pattern $((x : A + AB, y : BAA + A), z : AC + C)$ and input $ABAAC$. For pattern $(x : A + AB, (y : BAA + A, z : AC + C))$, we find the different POSIX match $\{x : AB, y : A, z : AC\}$.

4. Derivatives for Sub-Matching

We formalize the derivatives matching algorithm motivated in Section 2. Figure 6 summarizes all cases for building regular expression derivatives. For example, $l\backslash l = \epsilon$ and $(r_1 + r_2)\backslash\ l = r_1\backslash l + r_2\backslash l$. The pair case 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.

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

Pattern derivative of letter: $\cdot\backslash\cdot :: p \to l \to p$

$$
\begin{array}{lcl}
(x|w : r)\backslash l & = & (x|w{+\!+}\,[l] : r\backslash l) \\
(x|w : p)\backslash l & = & (x|w{+\!+}\,[l] : p\backslash l) \\
(p_1 + p_2)\backslash l & = & p_1\backslash l + p_2\backslash l \\
(p_1, p_2)\backslash l & = & \begin{cases} (p_1\backslash l, p_2) + (p_{1_\epsilon}, p_2\backslash l) & \text{if } \epsilon \in L(p_1\downarrow) \\ (p_1\backslash l, p_2) & \text{otherwise} \end{cases} \\
p^*\backslash l & = & (p\backslash l, p^*)
\end{array}
$$

Pattern derivative of word: $\cdot\backslash\cdot :: p \to w \to p$

$$
\begin{array}{lcl}
p\backslash\epsilon & = & p \\
p\backslash lw & = & (p\backslash l)\backslash w
\end{array}
$$

Empty pattern of shape p: $\cdot_\epsilon :: p \to p$

$$
\begin{array}{lcl}
(x|w : r)_\epsilon & = & \begin{cases} (x|w : \epsilon) & \text{if } \epsilon \in L(r) \\ (x|w : \phi) & \text{otherwise} \end{cases} \\
(x|w : p)_\epsilon & = & (x|w : p_\epsilon) \\
p_1 + p_{2_\epsilon} & = & p_{1_\epsilon} + p_{2_\epsilon} \\
(p_1, p_2)_\epsilon & = & (p_{1_\epsilon}, p_{2_\epsilon}) \\
p^*{}_\epsilon & = & (p_\epsilon)^*
\end{array}
$$

Extract regular expression from p: $\cdot\downarrow :: p \to r$

$$
\begin{array}{lcl}
(x|w : r)\downarrow & = & r \\
(x|w : p)\downarrow & = & p\downarrow \\
p_1 + p_2\downarrow & = & p_1\downarrow + p_2\downarrow \\
(p_1, p_2)\downarrow & = & (p_1\downarrow, p_2\downarrow) \\
p^*\downarrow & = & (p\downarrow)^*
\end{array}
$$

Figure 7. Pattern Derivatives

$env(\cdot) :: p \to \{\Gamma\}$

$$
\begin{array}{lcl}
env((x|w : r)) & = & \begin{cases} \{\{(x, w)\}\} & \text{if } \epsilon \in L(r) \\ \{\} & \text{otherwise} \end{cases} \\
env((x|w : p)) & = & \{\{(x, w)\} \uplus es | es \in env(p)\} \\
env((p_1, p_2)) & = & \{e_1 \uplus e_2 | e_1 \in env(p_1), e_2 \in env(p_2)\} \\
env((p_1 + p_2)) & = & env(p_1) \uplus env(p_2) \\
env(p^*) & = & env(p)
\end{array}
$$

$match(\cdot, \cdot) :: p \to w \to \{\Gamma\}$

$$match(p, w) \quad = \quad env(p\backslash w)$$

Figure 8. Derivative Matching

Figure 7 formalizes the construction of pattern derivatives $p\backslash l$. In case of a pattern variable, we build the derivative of the regular expression (base variable) or inner pattern (group variable). The match is recorded in the pattern itself by appending l to the already matched word w. The pattern syntax in case of variables is therefore slightly extended. The cases for choice and star are similar to the regular expression case. The pattern match for star records the binding for each iteration.

The pair case differs compared to the regular expression case. The $\cdot\downarrow$ helper function 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 $\cdot_\epsilon$.

Pattern equality: $\cdot = \cdot :: p \to p \to Bool$

$$
\begin{array}{lll}
(x|\_ : r_1) = (x|\_ : r_1) & \text{iff} & L(r_1) = L(r_2) \\
(x|\_ : p_1) = (x|\_ : p_2) & \text{iff} & p_1 = p_2 \\
(p_1, p_2) = (p_3, p_4) & \text{iff} & p_1 = p_3 \wedge p_2 = p_4 \\
p_1 + p_2 = p_3 + p_4 & \text{iff} & p_1 = p_3 \wedge p_2 = p_4 \\
{p_1}^* = {p_2}^* & \text{iff} & p_1 = p_2
\end{array}
$$

Simplifications:

$$
\begin{array}{llll}
\text{(S1)} & p_1 + p_2 & \longrightarrow & p_2 \quad \text{where } L(p_1 \downarrow) = \emptyset \\
\text{(S2)} & p_1 + p_2 & \longrightarrow & p_1 \quad \text{where } L(p_2 \downarrow) = \emptyset \\
\text{(S3)} & p_1 + p_2 & \longrightarrow & p_1 \quad \text{where } p_1 = p_2
\end{array}
$$

Figure 9. Pattern Simplifications

Figure 8 puts the pieces together. The pattern derivative function $\cdot\backslash\cdot$ builds the derivative of pattern p w.r.t the input word w. Function $env(\cdot)$ computes the list of all bindings of the resulting pattern (we treat multi-sets like lists). We assume that initially the matched words in patterns are empty (ϵ).

Soundness and completeness of matching with derivatives follow immediately.

PROPOSITION 4.1 (Pattern Derivative Soundness). *Let w be a word, p be a pattern and Γ a binding such that $w \vdash p \rightsquigarrow \Gamma$. Then, $\Gamma \in env(p\backslash w)$.*

PROPOSITION 4.2 (Pattern Derivative Completeness). *Let w be a word and p be a pattern. For all $\Gamma \in env(p\backslash w)$ we have that $w \vdash p \rightsquigarrow \Gamma$.*

As motivated earlier, the first match obtained via the derivative matcher must also be the POSIX match. This is the case because derivatives don't break apart the pattern structure. Via derivatives we greedily match the left-most parts of the pattern. Hence, this must be the POSIX match.

PROPOSITION 4.3 (Pattern Derivative POSIX Match). *Let w be a word, p be a pattern and Γ be an environment such that Γ is the first environment in $env(p\backslash w)$. Then, Γ is the POSIX match.*

A well-known problem with the derivative approach is that derivatives may grow exponentially. For example, consider the following example where we again use the earlier notation $\cdot \xrightarrow{\cdot} \cdot$ to denote the derivative step.

$$
\begin{array}{ll}
 & (x|\epsilon : A^*, y|\epsilon : A^*) \\
\xrightarrow{A} & (x|A : A^*, y|\epsilon : A^*) + (x|\epsilon : A^*, y|A : A^*) \\
\xrightarrow{A} & ((x|AA : A^*, y|\epsilon : A^*) + (x|A : A^*, y|A : A^*)) + \\
 & ((x|A : A^*, y|A : A^*) + (x|\epsilon : A^*, y|AA : A^*)) \\
\xrightarrow{A} & \ldots
\end{array}
$$

This exponential blow-up is not surprising given that via the derivative approach, we can compute all possible matchings. Given that our main interest is in the (first) POSIX match, we can apply some simplifications which have been identified in [20] in the context of testing regular language membership.

If we ignore the accumulated matchings, we can see that the underlying regular expressions of each pattern in

$$
\begin{array}{l}
((x|AA : A^*, y|\epsilon : A^*) + (x|A : A^*, y|A : A^*)) + \\
((x|A : A^*, y|A : A^*) + (x|\epsilon : A^*, y|AA : A^*))
\end{array}
$$

are identical and equivalent to (A^*, A^*). Hence, it suffices to keep only the left-most pattern which is

$$(x|AA : A^*, y|\epsilon : A^*)$$

$\cdot\backslash_p\cdot :: r \to l \to \{r\}$

$$
\begin{array}{lcl}
\phi\backslash_p l & = & \{\} \\
\epsilon\backslash_p l & = & \{\} \\
l_1\backslash_p l_2 & = & \begin{cases} \{\epsilon\} & \text{if } l_1 == l_2 \\ \{\} & \text{otherwise} \end{cases} \\
(r_1 + r_2)\backslash_p l & = & r_1\backslash_p l \cup r_2\backslash_p l \\
(r_1, r_2)\backslash_p l & = & \begin{cases} \{(r, r_2) | r \in r_1\backslash_p l\} \cup r_2\backslash_p l & \text{if } \epsilon \in L(r_1) \\ \{(r, r_2) | r \in r_1\backslash_p l\} & \text{otherwise} \end{cases} \\
r^*\backslash_p l & = & \{(r', r^*) \mid r' \in r\backslash_p l\}
\end{array}
$$

Figure 10. Regular Expression Partial Derivatives

Figure 9 formalizes the simplifications for the pattern case. (S1) and (S2) remove failed matches. (S3) favors the left-most match. These simplifications should be applied after each derivative step. For our running example, we obtain then the following derivation.

$$
\begin{array}{ll}
 & (x|\epsilon : A^*, y|\epsilon : A^*) \\
\xrightarrow{A} & (x|A : A^*, y|\epsilon : A^*) \\
\xrightarrow{A} & (x|AA : A^*, y|\epsilon : A^*) \\
\xrightarrow{A} & \ldots
\end{array}
$$

Thus, we achieve a reasonable performance. However, in our experience the partial derivative matching approach appears to be superior in terms of performance. In general, it is more effective to build a (partial derivative) automata whose size is per construction at most linear in the size of the regular expression pattern, instead of constructing a potentially exponentially large (derivative) automata which needs to be simplified. Hence, we will take a closer look at the partial derivative approach next.

5. Partial Derivatives for Sub-Matching

Our goal is to construct a NFA match automata as outlined in Section 2. The states of the NFA are pattern partial derivatives.

5.1 Pattern Partial Derivatives

First, we repeat the definition of regular expression partial derivatives in Figure 10. Operator $\cdot\backslash_p\cdot$ computes partial derivatives and is similar to the derivative $\cdot\backslash\cdot$ operator. The crucial difference is that we now put sub-results into a set instead of combining them via the choice operator $+$.

For expression A^* we find

$$A^*\backslash_p A = \{(\epsilon, A^*)\} =_{simplification} \{A^*\}$$

For brevity, we omit some obvious simplifications, e.g. $(\epsilon, A^*) \longrightarrow A^*$, to reduce the number of partial derivatives. We can restate the following result already reported in [2].

PROPOSITION 5.1 (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 construction of pattern partial derivatives follows closely the construction of regular expression partial derivatives. Instead of recording the pattern match in the pattern itself as in the derivative case, we associate a pattern matching function f to each partial derivative. Figure 11 contains the details.

In case of $x : r$, we compute the partial derivatives r' of the base regular expression r. The resulting set consists of elements $(x : r', x \mapsto l)$ where $x \mapsto l$ records that letter l is consumed by some pattern variable x. In case of a variable group pattern, we compose the matchings f of the partial derivatives of the underlying pattern p with the outer group match $x \mapsto l$.

$\cdot\backslash_p \cdot :: p \to l \to \{(p, x \to l)\}$

$$
\begin{array}{lcl}
(x:r)\backslash_p l & = & \{(x:r', x \mapsto l) | r' \in r\backslash_p l\} \\
(x:p)\backslash_p l & = & \{(p', (x \mapsto l) \circ f) | (p', f) \in p\backslash_p l\} \\
(p_1 + p_2)\backslash_p l & = & p_1\backslash_p l \cup p_2\backslash l \\
(p_1, p_2)\backslash_p l & = & \begin{cases} \{((p', p_2), f) | (p', f) \in p_1\backslash_p l\} \cup p_2\backslash_p l & \text{if } \epsilon \in L(p_1\downarrow) \\ \{(q, p_2) | q \in p_1\backslash_p l\} & \text{otherwise} \end{cases} \\
p^*\backslash_p l & = & \{((p', p^*), f \circ iterate_{fv(p)}) | (p', f) \in p\backslash_p l\}
\end{array}
$$

Figure 11. Pattern Partial Derivatives with Matching Function

The cases for choice and concatenation are straightforward. In case of the Kleene star, the purpose of $iterate_{fv(p)}$ is to keep track of the number of iterations in case of a star pattern. Thus, we can customize the matcher to keep all matchings concerning $fv(p)$ or only keep the last match (which is the typical case). For example, consider pattern $(x : AB + C)^*$. and input $ABCAB$. If $iterate_{fv(p)}$ is customized to keep the last match, then we obtain $\{x : AB\}$. If $iterate_{fv(p)}$ accumulates the individual matchings then the result will be $\{x : AB, x : C, x : AB\}$.

DEFINITION 5 (Star Pattern All Matchings). *We follow the* star pattern all matchings *policy if $iterate_{fv(p)}$ accumulates all matchings of each the individual iteration step.*

DEFINITION 6 (Star Pattern Last Match). *We follow the* star pattern last match *policy if $iterate_{fv(p)}$ removes the bindings for all variables in $fv(p)$ besides the last, i.e. current, match.*

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

PROPOSITION 5.2 (Finiteness of Pattern Partial Derivatives). *For a finite alphabet $\Sigma = \{l_1, ..., l_n\}$ and pattern p, the set P of pattern partial derivatives of p and its descendants computed via $\cdot\backslash_p \cdot$ is finite. The set P can be described as the least fix point of the following equation*

$$\mathcal{P}(p) \;=\; \{q | (q, f) \in p\backslash_p l_1 ... l_n\} \cup \{q' | q \in \mathcal{P}(p), q' \in q\backslash_p l_1 ... l_n\}$$

where $q\backslash_p l_1 ... l_n = q\backslash_p l_1 \cup ... \cup q\backslash_p l_n$. The size of the set $\mathcal{P}(p)$ is linear in the size of the pattern.

We consider construction of partial derivatives for our earlier example $(x : A + y : AB + z : B)^*$. We start with $p_1 = (x : A + y : AB + z : B)^*$. Next,

$$
\begin{array}{rcl}
p_1\backslash_p A & = & \{((x:\epsilon)p_1, x \mapsto A), ((y:B)p_1, y \mapsto A)\} \\
 & & \text{simplification} \\
 & = & \{(p_1, x \mapsto A), (\underbrace{(y:B)p_1}_{p_2}, y \mapsto A)\}
\end{array}
$$

Like in the regular expression case, we apply some simplifications. The remaining calculations are as follows.

$$
\begin{array}{rcl}
p_1\backslash_p B & = & \{((z:\epsilon)p_1, z \mapsto B)\} \\
 & & \text{simplification} \\
 & = & \{(p_1, z \mapsto B)\} \\
 & & \\
p_2\backslash_p A & = & \{\} \\
 & & \\
p_2\backslash_p B & = & \{((y:\epsilon)p_1, y \mapsto B)\} \\
 & & \text{simplification} \\
 & = & \{(p_1, y \mapsto B)\}
\end{array}
$$

We have reached a fix point.

5.2 NFA Match Automata

The above result allows us to build a non-deterministic, finite match automata in style of Laurikari's NFAs with tagged transitions [12].

DEFINITION 7 (NFA Match Automata). *We define the NFA match automata for pattern p as follows. Pattern p is the initial state. The set of final states equals*

$$\{q | q \in \mathcal{P}(p), \epsilon \in L(q\downarrow)\}$$

That is, a pattern is final if its underlying regular expression contains the empty word.

The NFA transitions result from the pattern partial derivative operation as follows. For each $(p', f) \in (p\backslash_p l)$, we introduce a transition

$$p \xrightarrow{(l,f)} p'$$

We use a Mealy automata where the letter l is the triggering condition and the match function f is the output function applied to the match environment.

For our running example $(x : A + y : AB + z : B)^*$, the NFAs transitions are shown in Figure 1.

5.3 Greedy Left-Most Matching Algorithm

DEFINITION 8 (Greedy Left-Most Matching Algorithm). *The greedy left-most matching algorithm for pattern p and input word w is defined as the left-to-right traversal of the NFA match automata resulting from p for input word w. Sink states of transitions are kept in the order as they appear in the set of partial derivatives. Duplicate states are removed, only the left-most state is kept.*

Precisely, transitions operate on configurations

$$\{p_1, ..., p_n\} \;\; \Gamma_1^{p_1} ... \Gamma_n^{p_n}$$

For transitions

$$
\begin{array}{c}
p \xrightarrow{(l,f_1)} p'_1 \\
... \\
p \xrightarrow{(l,f_m)} p'_m
\end{array}
$$

where $\{(p'_1, f_1), ..., (p'_m, f_m)\} = p\backslash_p l$ and configuration

$$\{p_1, ..., p, ..., p_n\} \;\; \Gamma_1^{p_1} ... \Gamma^p ... \Gamma_n^{p_n}$$

we obtain the (intermediate) derivation step

$$
\begin{array}{rl}
 & \{p_1, ..., p, ..., p_n\} \;\; \Gamma_1^{p_1} ... \Gamma^p ... \Gamma_n^{p_n} \\
\xrightarrow{l} & \{p_1, ..., p'_1, ..., p'_m, ..., p_n\} \;\; \Gamma_1^{p_1} ... f_1(\Gamma)^{p'_1} ... f_m(\Gamma)^{p'_m} ... \Gamma_n^{p_n}
\end{array}
$$

Of course, we need to reduce the remaining p_i's w.r.t. letter l to obtain a complete derivation step.

In a configuration, the resulting set of states is simplified by removing duplicate states where we keep the left-most state. That is, $P_1 \cup \{p\} \cup P_2 \cup \{p\} \cup P_3$ is simplified to $P_1 \cup \{p\} \cup P_2 \cup P_3$ until there are no duplicates.

We elaborate on a few aspects of the algorithm.

To guarantee the greedy left-most matching order, it is important that transitions are executed as in the (left-to-right) order of partial derivatives $\{(p'_1, f_1), ..., (p'_m, f_m)\}$ as computed by $p\backslash_p l$.

Our algorithm does not require to fully annotate the pattern. The reason is that the construction of pattern partial derivatives strictly breaks apart the pattern structure. Consider the base case $(x : r)$ from Figure 11

$$(x:r)\backslash_p l \;=\; \{(x:r', x \mapsto l) | r' \in r\backslash_p l\}$$

For $p = (x : (A + AB), y : (B + \epsilon))$, we obtain (after simplification)

$$p\backslash_p A \;=\; \{(y : (B + \epsilon), x \mapsto A), ((x:B)(y : (B + \epsilon)), y \mapsto B)\}$$

This guarantees that we compute the greedy left-most match $\{x : A, y : B\}$ for input AB.

The set of NFA states is a constant, bound by the size of p. Hence, the running time of the algorithm is linear in the size of the input. In summary, we can state the following results. By construction, the algorithm follows the greedy left-to-right matching policy.

PROPOSITION 5.3 (Greedy Algorithm Correctness and Complexity). *The greedy left-most matching algorithm implements the greedy left-most matching policy and it's running time is linear in the size of the input. The space required to store the final and all intermediate match environments is a constant, bound by the size of the pattern.*

5.4 NFA Comparison

In Figure 12 we show the size of the resulting match automata for Thompson, Glushkov and partial derivative NFAs. Our focus is on the specific automata construction method without any post-simplification step. As can be seen, the partial derivative NFA is 'smaller' compared to the other NFAs. This is confirmed through [1], [2] and [21]. Thompson NFAs often have the largest sets of states and transitions due to the ϵ transitions. According to [21], for large alphabet sets, partial derivative NFAs are about half the sizes of Glushkov NFAs in terms of states and transitions.

In the last example, we use Σ to denote the union of all ASCII characters. In this particular case, the Glushkov NFA construction scales the worst. This is due to the fact that each character creates a state in the NFA [15]. Due to the Kleene star, there are at least 256*256 transitions. The Thompson NFA does not scale well in terms of states, either. Our implementation de-sugars $(A+B+C)$ to $(A+(B+C))$ and therefore more states will be generated. Of course, the size of the Thompson NFA could be significantly reduced if we avoid this de-sugaring step.

We have built reference implementations of greedy left-most matching for all three NFA approaches. Basic measurements show that the matcher based on partial derivatives is generally much faster. These are 'non-optimized' reference implementations. Hence, the result is not necessarily conclusive but is an indication that matching with partial derivatives is promising. We provide conclusive evidence in the later Section 7.

6. Extensions for Real world Applications

So far, we have presented the core of a system to support regular expression pattern matching. Regular expression patterns used in the real world applications require some extensions. 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

6.1 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.

6.2 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

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

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

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

6.3 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 $\cdot\backslash_p\cdot$ in Figure 11 as follows,

$$(p_1, p_2)\backslash_p l \;=\; \begin{cases} p_2\backslash_p l \;\cup & \\ \{((p', p_2), f) | (p', f) \in p_1\backslash_p l\} & \text{if } \epsilon \in L(p_1 \downarrow) \\ & \wedge \neg greedy(p_1) \\ \{((p', p_2), f) | (p', f) \in p_1\backslash_p l\} & \\ \cup\; p_2\backslash_p l & \text{if } \epsilon \in L(p_1 \downarrow) \\ & \wedge greedy(p_1) \\ \{(y, p_2) | y \in p_1\backslash_p l\} & \text{otherwise} \end{cases}$$

Extending our pattern language with the greediness symbol is straight-forward and the definition of $greedy(\cdot) :: p \rightarrow bool$ is omitted for brevity.

6.4 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 implementation clearly supports anchored matches. To support unanchored match, we could rewrite the unanchored pattern `p` into an equivalent anchored form, `^.*?p.*$`, and proceed with anchored match.

6.5 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.

7. Experimental Results

We measure the performance of our regular expression sub-matching approach based on partial derivatives. We have built an optimized implementation of greedy left-to-right matching. Our

Pattern	Thompson NFA		Glushkov NFA		Partial Derivative NFA	
	# states	# transitions	# states	# transitions	# states	# transitions
$(x : A + AB, (y : BAA + A, z : AC + C))$	23	24	11	14	8	11
$(x : A^*, y : A^*)$	6	7	3	5	3	5
$(x : (A + B)^*, y : (A, (A + \epsilon), B))$	14	16	6	12	5	9
$(x : \Sigma^*)$	768	1023	257	65792	2	512

Figure 12. NFA Match Automata Comparison

implementation is entirely written in Haskell and we employ several (fairly standard) optimizations such as hashing of states etc.

The benchmarkings were conducted on a Mac OSX 10.6.8 with 2.4GHz Core 2 Duo and 8GB RAM. The benchmark programs were compiled using GHC 7.0.4. We divide the benchmarkings into two groups. In the first group, we compare our implementation with other native Haskell implementation. In the second group, we challenge the C implementations.

The complete set of the benchmark results can be located via

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

in which the broader scope of comparison is considered.

7.1 Contenders and Benchmark Examples

The contenders are:

- PD-GLR our greedy left-to-right matching implementation.
- Weighted [7], the native Haskell implementation of regular expression matching. Weighted's sub matching is fairly limited because it only supports one variable pattern, i.e. `_ :: .* ( x :: r ) _ :: .*`. Nevertheless, we included Weighted in the comparison. [1] The implementation is accessible via the Haskell package Text.Regex.Weighted;
- TDFA, the native Haskell implementation of [11]. It is accessible via the library Text.Regex.TDFA [19].
- RE2, the google library re2;
- PCRE, the pcre library, accessible via the Haskell wrapper Text.Regex.PCRE [17];
- PCREL, the light-weight wrapper to the pcre library, accessible via Text.Regex.PCRE.Light [18]

The benchmark sets consist of examples adopted from [6], and some others being extracted from the real world applications that we encountered. For the up-coming performance measurements, we select the following representative examples:

1. A simple pattern which involves no choice, in which our implementation does not take any advantage;
2. A contrived pattern which builds up a lot of choices, in which our algorithm out-performs PCRE's and is on par with RE2's;
3. Two real world application examples in which we assess the practicality of our implementation.

7.2 Competing with Native Haskell Implementations

In Figure 13, Figure 14 and Figure 15, we compare the run-time performance of PD-GLR, TDFA and Weighted. As a convention in all figures, the x-axis measures the size of the input; the y-axis measures the time taken to complete the match, measured in seconds.

Figure 13 shows the results of matching the pattern `^.*$` against some randomly generated text. PD-GLR's performance is on par

[1] Other implementations, e.g. Text.Regex.Parsec, Text.Regex.DFA and Text.Regex.TRE couldn't be compiled at the time when this paper was written. Text.RegexPR didn't scale at all for any of our examples. Therefore, we exclude these packages from our benchmark comparison.

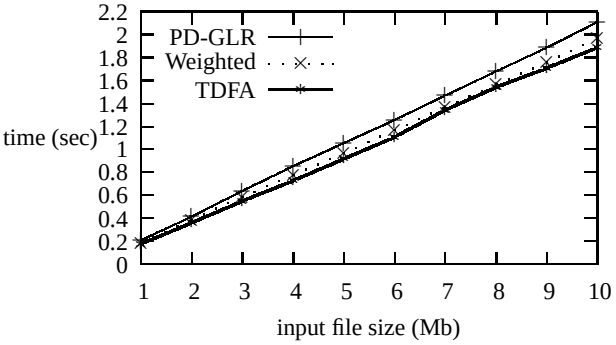

Figure 13. Native Haskell benchmark `^.*$`

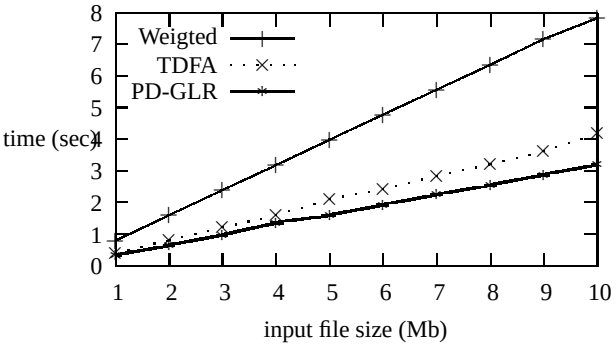

Figure 14. Native Haskell benchmark `^.*foo=([0-9]+).*bar=([0-9]+).*$`

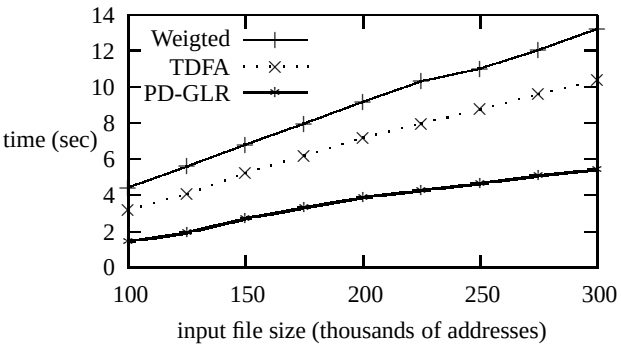

Figure 15. Native Haskell benchmark `^(.*) ([A-Za-z]{2}) ([0-9]{5})(-[0-9]{4})?$`

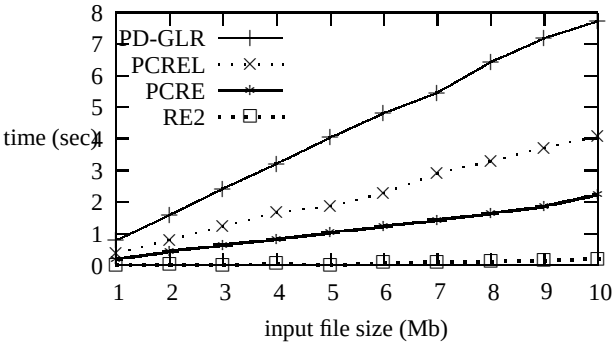

Figure 16. C benchmark `(.*)$`

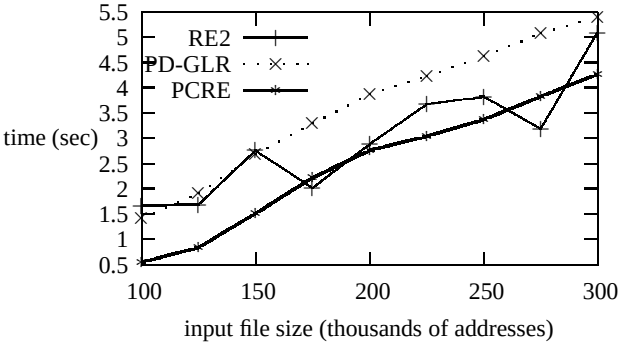

Figure 17. C benchmark `^(.*) ([A-Za-z]{2}) ([0-9]{5})(-[0-9]{4})?$`

with others because the pattern is simple.[2] In Figure 14 and Figure 15, we use two examples extracted from some real world applications. The pattern in Figure 14,

```
^.*foo=([0-9]+).*bar=([0-9]+).*$
```

extracts the values of some HTTP URL Get parameters, `foo` and `bar`, from the random input strings. For example, the string "http://www.mysite.com/?foo=123&bar=567" matches with the above pattern.

The pattern in Figure 15

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

extracts the US addresses from the input. For instance, the address "Mountain View, CA 90410" matches with the pattern. Note that the x-axis in Figure 15 measures the number of addresses in the input. The charts show that in case of complex patterns, PD-GLR out-performs Weighted thanks to the smaller automata. PD-GLR performs marginally better than TDFA in these examples.

7.3 Competing with C Implementation and Wrappers

In this section, we challenge ourselves by benchmarking against some implementation directly using C or wrapping around the C PCRE library.

In Figure 16, we use a wild card pattern `(.*)$` to match with some randomly generated text. PD-GLR's performance is slightly worse than PCRE and PCREL. RE2 is taking the lead by a factor

[2] Weighted does not support the anchor extension. In the actual benchmark code, we mimic it by using the `fullMatch` function. Further more, Weighted does not support group matching via (), which is automatically ignored by the regular expression compiler.

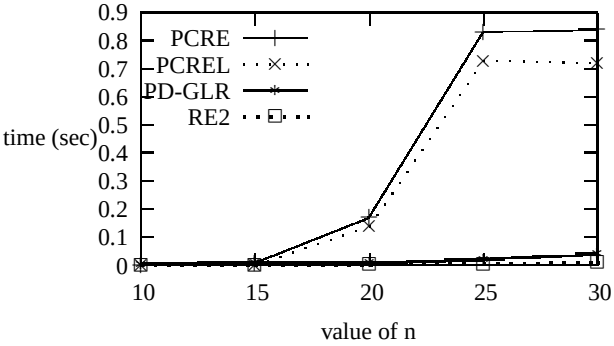

Figure 18. C benchmark `^(a?){n}(a){n}$`

of ten. Profiling PD-GLR for this example shows that most time is spent in de-composing the input ByteString [4], which is of course less efficient than the C counterparts which have direct access to the character array.

In Figure 17, we re-apply the example from Figure 15 to PCRE and RE2. It shows that PD-GLR and RE2 perform slightly worse than PCRE. The difference is about 1 second, due to the ByteString decomposition. Note that the run-time statistics of PCREL is omitted in Figure 17. In this particular example, PCREL performs the worst. The difference is by hundreds of seconds compared to others.

In the last example in Figure 18, we match a string with `n` "a"s against the pattern `^(a?){n}(a){n}$`. The x-axis measures the value of input `n`. For instance, let `n` is 2, we match "aa" with `^(a?){2}(a){2}$`. The sub-pattern `(a?){2}` will *not* match anything. PCRE does not scale well because of back-tracking. When the input `n` reaches 30, the program exits with a segmentation fault. PCREL shares the same behavior as PCRE since they share the same C backend library. The non-backtracking algorithms PD-GLR and RE2 show the similar performance. What is omitted in Figure 18 is that, when `n > 30`, PD-GLR is performing non-linearly. Profiling reveals that the overhead is arising from the compilation phase. We plan to address this issue in the future work.

7.4 Performance Measurement Summary

Our measurements show that ours is the fastest native Haskell implementation of regular expression sub-matching. Compared to state-of-the art C-based implementations RE2 and PCRE, our implementation has a fairly competitive performance.

8. Related Work and Discussion

Our use of derivatives and partial derivatives for regular expression matching is inspired by our own prior work [14, 22] where we derive up-cast and down-cast coercions out of a proof system describing regular expression containment based on partial derivatives. The goal of this line of work is to integrate a XDuce-style language [9] into a variant of Haskell [23]. The focus of the present work is on regular expression sub-matching and some specific matching policies such as greedy and POSIX.

Regular expression derivatives have recently attracted again some attention. The work in [16] employs derivatives for scanning, i.e. matching a word against a regular expression. To the best of our knowledge, we are the first to transfer the concept of derivatives and partial derivatives to the regular expression sub-matching setting.

Prior work relies on mostly Thompson NFAs [24] for the construction of the matching automata. For example, Frisch and Cardelli [8] 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 [10].

Laurikari [12, 13] 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 partial derivative operation $\cdot\backslash_p\cdot$.

Kuklewicz has implemented Laurikari style tagged NFAs in Haskell. He [11] discusses various optimizations techniques to bound the space for matching histories which are necessary in case of (forward) left-to-right POSIX matching.

Cox [6] 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 [6] and the references therein for further details. He introduces the idea of right-to-left scanning of the input for POSIX matching.

As said, all prior work on efficient regular expression matching relies mostly on Thompson NFAs or variants of it. Partial derivatives 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 [8] considers ϵ-transitions as problematic for the construction of the matching automata. Laurikari [12, 13] therefore first removes ϵ-transitions whereas Cox [6] 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 [2] 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.

Fischer, Huch and Wilke [7] discuss a rewriting-based approach to support regular expressions based on Glushkov NFAs. They build the matching automata incrementally during the actual matching whereas we build a classic matching automata based on partial derivative NFAs. There exists a close connection between Glushkov and partial derivative NFAs, e.g. see [1]. However, as our benchmark results show it appears that matching via partial derivative NFAs is superior. As mentioned earlier, the Fischer et. al. approach is fairly limited when it comes to sub-matching. Three matching policies are discussed (leftmost, leftlong and longest). Longest seems closest to POSIX and greedy left-most but not formal investigations on this topic are conducted in [7].

9. Conclusion

Our work shows that regular expression sub-matching can be elegant using derivatives and partial derivatives. The Haskell implementation of our partial derivative matching algorithm is the fastest among all native in Haskell implementations we are aware of. Our performance results show that we are competitive compared to state-of-the-art C-based implementations such as PCRE and RE2. Our extensive set of benchmarks show that our approach yields competitive performance results.

Acknowledgments

We thank Christopher Kuklewicz for useful discussions about his TDFA system, Christian Urban for his comments and Russ Cox for pointing us to some related work. We are grateful to some ICFP'10, ICFP'11 and PPDP'12 reviewers for their helpful comments on previous versions of this paper.

References

[1] C. Allauzen and M. Mohri. A unified construction of the Glushkov, follow, and Antimirov automata. In *Proc. of MFCS'06*, volume 4162 of *LNCS*, pages 110–121. Springer, 2006.

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

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

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

[5] R. Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...), 2007. `http://swtch.com/~rsc/regexp/regexp1.html`.

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

[7] S. Fischer, F. Huch, and T. Wilke. A play on regular expressions: functional pearl. In *Proc. of ICFP'10*, pages 357–368. ACM Press, 2010.

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

[9] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL '01*, pages 67–80. ACM Press, 2001.

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

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

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

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

[14] K. Z. M. Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer-Verlag, 2004.

[15] G. Navarro and M. Raffinot. Compact dfa representation for fast regular expression search. In *Proc. of Algorithm Engineering'01*, volume 2141 of *LNCS*, pages 1–12. Springer, 2001.

[16] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.

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

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

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

[20] G. Rosu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Proc. of RTA'03*, volume 2706 of *LNCS*, pages 499–514. Springer, 2003.

[21] Nelma Moreira Sabine Broda, Antonio Machiavelo and Rogerio Reis. Study of the average size of glushkov and partial derivative automata, Octorber 2011.

[22] M. Sulzmann and K. Z. M. Lu. A type-safe embedding of XDuce into ML. In *Proc. of ACM SIGPLAN Workshop on ML*, Electronic Notes in Computer Science, pages 229–253, 2005.

[23] M. Sulzmann and K. Z. M. Lu. Xhaskell - adding regular expression types to haskell. In *Proc. of IFL'07*, volume 5083 of *LNCS*, pages 75–92. Springer-Verlag, 2007.

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