Correct and Efficient POSIX Submatch Extraction with Regular Expression Derivatives

Martin Sulzmann
Hochschule Karlsruhe - Technik und Wirtschaft
martin.sulzmann@hs-karlsruhe.de

Kenny Zhuo Ming Lu
Nanyang Polytechnic
luzhuomi@gmail.com

Abstract

The POSIX submatching policy favors left-most longest submatches. Compared to other policies such as greedy left-most found in Perl, the POSIX policy is easier to explain but much harder to implement. Almost all POSIX implementations are buggy as observed by Kuklewicz. We show how to obtain an elegant and efficient POSIX submatching engine based on Brzozowski's regular expression derivatives. Correctness is fairly straightforward to establish and our benchmark results show that our approach is promising.

1. Introduction

Regular expression matching is the problem of deciding if a string is an element of the language denoted by a regular expression. In practice, plain matching is not enough. We wish to know which subexpressions match which substrings. In general, the extraction of submatches is ambiguous because there may be several possible ways to split a string into substrings.

There are two popular strategies to disambiguate submatching: POSIX [9] and greedy left-most [20] as found in Perl. Our focus here is on POSIX submatching. The works in [8, 25] provide for a formal basis by clarifying some points which have been left open in the original specification [9]. In a nutshell, POSIX submatching can be summarized as follows:

> "Subpatterns should match the longest possible substrings, where sub- patterns that start earlier (to the left) in the regular expression take priority over ones starting later. Hence, higher-level subpatterns take priority over their lower-level component subpatterns. Matching an empty string is considered longer than no match at all."

Despite the fact that POSIX enjoys a clear and concise specification, it appears that most POSIX submatching implementations fail to compute the proper POSIX submatch [11]. The challenge of POSIX submatching is that the left-most longest match must be selected. This effectively means that all alternatives must be tried out and among the successful ones the earliest (left-most) longest match must be selected. A naive method is to rely on backtracking to exhaustively search for the proper POSIX submatches. Such a method is obviously correct but potentially has an exponential run time and space usage due to backtracking.

Laurikari [15] introduces the idea of NFAs with tagged transition to keep track of submatches. Tagged NFAs yield an efficient submatching algorithm which computes submatches linear (in the size of the input string) and in constant space. However, Laurikari-style tagged NFAs won't yield the proper POSIX match unless some adjustments are made, see [6, 13, 18].

The adjustments are subtle and establishing their correctness, i.e. the resulting algorithm yields the POSIX match, is not that straightforward. In addition, these adjustments carry some additional cost at run-time. An additional comparison step is required to select the proper POSIX submatch among the accumulated set of submatchings.

In this work, we propose a novel DFA-based method for computing POSIX submatches which like the above works runs in linear time and uses constant space. Advantages of our approach are that correctness is fairly straightforward to establish, almost by construction, and at run-time there is no need to apply a comparison step to select the longest submatch among several submatches. Experiments confirm that our approach yields improved running times for ambiguous regular expression patterns.

Our method for computing POSIX submatches is based on Brzozowski's regular expression derivatives [1]. A sketch of how derivatives could be applied to compute POSIX submatches is given in our own prior work [24]. The present work includes some significant improvements such as formal correctness statements, numerous optimizations and a competitive implementation.

In summary, we make the following contributions:

- We give an executable specification of POSIX submatching (Section 3).
- We present a succession of improved methods to compute POSIX submatches based on Brzozowski's regular expression derivatives:
  - The integration of submatching with the derivatives operation (Sections 5 and 6)
  - The addition of normalization (Section 7)
  - The construction of an explicit POSIX submatch DFA (Section 8).
- We have built an optimized implementation where for many cases we achieve excellent benchmark results. For suboptimal cases, we discuss improvements (Section 9).

To the best of our knowledge, there exist only a few related works [6, 13, 18] which apply automata-based methods for POSIX submatching. The upcoming Section 2 provides for an overview. Section 9 gives a performance comparison.

Throughout the paper, we will use Haskell as our specification and implementation language. The exception is the up-coming section where we use standard math syntax for regular expressions to

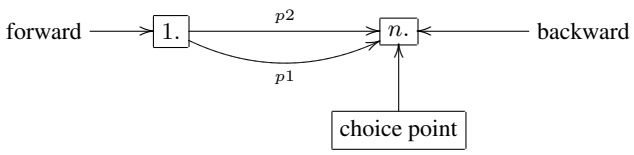


Figure 1. Forward versus Backward POSIX Selection

explain the challenge of POSIX submatch extraction and highlight the key ideas of our proposed solution.

Informal proof sketches for propositions will be provided. All propositions are stated as QuickCheck [3] properties and have been tested extensively.

2. The Challenge of POSIX Submatch Extraction

For input string AB and regular expression $(A + AB + B)^*$, there are two possible ways to break apart input AB:

$$(1)\ A, B \quad \text{and} \quad (2)\ AB$$

Either in the first iteration subpattern A matches substring A, and in the second iteration subpattern B matches substring B, or subpattern AB immediately matches the input string.

Case (1) is the greedy left-most match whereas case (2) is the POSIX match. POSIX favors left-most longest submatches. Clearly, AB is the longer submatch compared to A.

Let's see how an automata-based method finds the proper POSIX match. We assume an NFA approach in style of [15]. For convenience, we use regular expressions to denote NFA states. We write

$$\{r_1, ..., r_m\} \xrightarrow{c} \{r'_1, ..., r'_n\}$$

to denote the NFA transition relation where from states $\{r_1, ..., r_m\}$ we reach states $\{r'_1, ..., r'_n\}$ after consuming character c.

For our example, the starting state is $(A + AB + B)^*$ from which we can either reach the starting state again or state $B(A + AB + B)^*$ after consuming A. Subsequently, from $(A + AB + B)^*$ and $B(A + AB + B)^*$ we reach each time $(A + AB + B)^*$ after consuming B. Hence, we find the following derivation:

$$\begin{array}{lll} \{(A + AB + B)^*\} & \xrightarrow{A} & \{(A + AB + B)^*, B(A + AB + B)^*\} \\ & \xrightarrow{B} & \{\underline{(A + AB + B)^*}, \underline{\underline{(A + AB + B)^*}}\} \end{array}$$

A choice point is reached. We shall only keep one of the underlined states.

Both states represent different submatches. The single underlined state corresponds to case (1) and the double underlined state corresponds to case (2). To select the proper POSIX match, we must compare the set of submatches accumulated in each iteration and select the left-most longest match. It suffices to compare the range of the substring matched by a subpattern. For convenience, we will use here the actual substrings.

For the single underlined state we have A in the first iteration and B in the second iteration of the Kleene star whereas for the double underlined state we have AB in the first iteration of the Kleene star. Clearly, the double underlined state is the POSIX match.

As it seems, we must keep track of the set of accumulated submatches for *each* iteration of a Kleene star for *all* potential POSIX submatches. In the worst case, we require space linear in the size of the input string.

Pruning of Search Space We are aware of three automata-based methods [6, 13, 18] which show how to compute the proper POSIX match in linear time and constant space. [1] The insight is that the maximum path between choice points is bound by the regular expression. In essence, the works [6, 13, 18] apply some pruning to limit the size of the search space of possible submatches. Hence, via some appropriate bookkeeping of submatches, the proper POSIX match can be selected when reaching a choice point. Hence, the storage space can be bound by the number of captured substrings which is bound by the number of subpatterns which is bound by the size of the regular expression.

Pruning via Forward NFA Submatching The works in [13, 18] apply a forward approach for tracking ambiguous submatches. Roughly, the method in [18] achieves time complexity $O(m * n^2)$ where m is the size of the input and n is the size of the NFA (transitions and states) which equals the size of the regular expression. A stack is used to record submatches where the maximal stack height is bound by n. A similar approach is pursued by [13] based on the idea of 'orbit tags'. We are not aware of any precise complexity results but the time complexity should be in a similar range as [18].

Pruning via Backwards NFA Submatching Cox [6] observes that by performing the matching from right-to-left, i.e. running the NFA backwards, there is no need to track submatches along alternative paths. This reduces the amount of bookkeeping and leads to a reduced storage space. Figure 1 illustrates the workings of forward and backwards submatching.

In case of forward submatching we must keep track of submatches along alternative paths until we reach a choice point. In the illustration, we assume that there are two alternative (forward) paths, p_1 and p_2, which after n unfoldings of a Kleene star lead to a common NFA state (i.e. choice point). At this point, the proper POSIX match is selected based on the accumulated submatches in p_1 and p_2.

The key insight by Cox is that by running the NFA backwards we reach the choice point "earlier" and can immediately make a decision based on the current submatches. Thus, we avoid some extra bookkeeping. In addition, to improvement in space usage, the approach by Cox should also yield an improved time complexity of $O(m * n)$. However, we are not aware of any formal results.

Constant Overhead due to Submatch Selection Common to all three approaches is that a comparison step is required to select the proper POSIX submatch. In case of an ambiguous pattern when reaching a choice point, we must select the proper POSIX submatch among the set of accumulated submatches. The number of comparisons is bound by the regular expression and therefore the comparison cost can be assumed to be a constant.

Our Idea In our approach such a comparison step is *not* required. Our insight is that by making use of Brzozowski's regular expression derivatives [1] we can build a DFA which yields the proper POSIX match virtually by construction.

Let $r\backslash c$ denote the derivative of a regular expression r w.r.t a character c. The derivative is again a regular expression where where c has been consumed. In language terms, we have that $L(r\backslash c) = \{w \mid cw \in L(r)\}$.

For example, $(A + AB + B)^*\backslash A = (\epsilon + B)(A + AB + B)^*$ where we simply unroll the Kleene star once and build the derivative of the pattern under the Kleene star. For our running example, we find the following derivative derivation steps:

$$\begin{array}{lll} (A + AB + B)^* & \xrightarrow{A} & (\epsilon + B)(A + AB + B)^* \\ & \xrightarrow{B} & \phi(A + AB + B)^* + \epsilon(A + AB + B)^* \end{array}$$

The important observation is that the derivative operation strictly consumes characters from the left. Thus, we straightforwardly obtain *longest* submatches. The *left-most* property is guaranteed by

[1] We treat the size of the regular expression pattern as a constant.

```
data Re l where
  Choice :: l → [Re l] → Re l
  Pair :: l → Re l → Re l → Re l
  Star :: l → Re l → Re l
  Ch :: l → Char → Re l
  Eps :: l → Re l
  Phi :: Re l

class Var l where
   v :: Int → l

instance Var Int where
  v = λx → x
```

Figure 2. Regular Expression Patterns

extracting submatches from the left in case of a regular expression choice. The advantage is that a comparison step as required in [6, 13, 18] is not necessary for our approach.

For efficiency reasons, we simplify expressions by removing non-accepting and non-POSIX parts. The above example will be simplified to $(A + AB + B)^*$. Details will be explained later.

Our experiments confirm that our approach works well in practice and has in particular advantages in case of ambiguous patterns.

3. Executable POSIX Specification

In first step, we develop an executable POSIX Specification which we will use for QuickCheck-style property testing of our approach.

3.1 Regular Expression Patterns

The data type in Figure 2 describes the possible patterns of regular expressions. For brevity, we neglect anchored patterns which are supported by our optimized implementation. Constructor `Choice` takes a list of alternatives. `Pair` represents concatenation and `Star` represents the Kleene star. Individual characters are wrapped with `Ch`. The empty string is represented by `Eps` whereas `Phi` denotes the empty language.

All constructors carry an annotation `l` to connect matched subpatterns to substrings. The exception is `Phi` because this pattern can't match any input. For the purpose of formalizing the POSIX disambiguation policy, we instantiate `l` with `Int`.

In later parts of the paper, we consider different instantiations of `l`. Type class `Var` provides for a uniform interface, mapping distinct identifiers, `Int` values, to annotations `l`.

In general, regular expressions are assumed to be *well-formed*, i.e. each subpattern is identified by a distinct, positive `Int` value. We refer to the `Int` value as the *label* of the subpattern.

Here is the earlier example written in our regular expression pattern syntax

```
-- (A + AB + B)*
r1 :: Var l ⇒ Re l
r1 = Star (v 1) $ Choice (v 2)
          [ Ch (v 3) 'A',
            Pair (v 4) (Ch (v 5) 'A')
                 (Ch (v 6) 'B'),
            Ch (v 7) 'B' ]
```

3.2 POSIX Submatch Policy Specification

Function `posix` in Figure 4 formalizes the POSIX disambiguation policy in Haskell by following the formal description in [25]. Figure 3 contains some type definitions and helper functions.

Function `posix` computes the POSIX match which is either the empty list if no match exists, or the list of POSIX submatches. The environment of the individual submatching is represented as a list of pairs where the first component refers to the subpattern and the second component holds the actual submatch.

```
type SubMatch = (Int, Maybe (Int,Int))
type Word = [(Int,Char)]

range :: Word → Maybe (Int,Int)
range [] = Nothing
range w = let (left,_) = head w
              (right,_) = last w
          in Just (left,right)

split2 :: Word → [(Word,Word)]
split2 [] = [ ([],[]) ]
split2 (w@(c:ws)) =
   nub $ (w,[]) : ([],w) :
         (map (λ(w1,w2) → (c:w1,w2)) $ split2 ws)

split :: Word → [[Word]]
split [] = [ [] ]
split [c] = [ [[c]] ]
split (w@(c:ws)) =
 [w]:[ take i w : xs | i ← [1..length ws],
                       xs ← split (drop i w) ]
```

Figure 3. Submatch Type Definitions and Helper Functions

For each submatch, we record the range, i.e. left and right positions within the original string. Via the `Maybe` data type, we signal if there's a submatch at all. This representation is more efficient than copying entire substrings. Therefore, function `posix` takes as input a `Word` which is a list of characters attached with position information. For convenience, we provide an interface using common strings as input:

```
posix' :: String → Re Int → [SubMatch]
posix' s r = posix (zip [1..] s) r
```

Function `posix` is defined structurally over the various pattern cases. Cases `Eps` and `Ch` are straightforward. In case of `Choice`, we try all alternatives and select the first (left-most) successful match.

In case of `Pair`, we consider all possible combinations of splitting input `w` into two parts `w1` and `w2` via the helper function `split2`. Among the successful combinations `((w1,w2),m1,m2)`, we then select the maximal combination `(w1,w2)` w.r.t. the canonical lexicographic order among pairs of words. In Haskell, these definitions are already predefined. For example, we have that

```
([(1,'A'),(2,'B')],[(3,'A')])
                  >
([(1,'A')],[(2,'B'),(3,'A')])}
```

By construction the left-most longest match equals the maximal combination. The result is simply `(l, range w) : m1 ++ m2` where `(l, range w)` describes the pair submatch and `m1` and `m2` are the submatch results of the left and right components.

In case of case `Star`, we split `w` into all combinations `[w1,...,wn]` and select the maximal, successful combination. As in case of pairs, the lexicographic order among lists of words is already predefined in Haskell. For example, we have that

```
[[(1,'A')],[(2,'B'),(3,'A')]]
                  >
[[(1,'A')],[(2,'B')],[(3,'A')]]
```

We follow the standard convention to record the last match within a Kleene star pattern only. Therefore, we apply `last` on maximal list of submatches in case of `Star`.

For example, we find

```
posix "AB" r1
⟹
[(1,Just (1,2)),(2,Just (1,2)),
 (4,Just (1,2)),(5,Just (1,1)),(6,Just (2,2))]
```

```
posix :: Word → Re Int → [SubMatch]
posix [] (Eps l) = [(l,Nothing)]
posix [(i,c)] (Ch l c')
  | c == c'   = [(l,Just (i,i))]
  | otherwise = []
posix w (Choice l rs) =
  case (filter ([] /=) $ map (posix w) rs) of
    (m:_) → (l,range w) : m
    []    → []
posix w (Pair l r1 r2) =
  case (filter (λ(_,m1,m2) → m1 /= [] && m2 /= []) $
        map (λ(w1,w2) → ((w1,w2),
                         posix w1 r1, posix w2 r2)) $
        split2 w) of
    [] → []
    ms → let (_,m1,m2) = maximumBy
                         (λ(p1,_,_)→ λ(p2,_,_)→
                            if p1 < p2 then LT
                            else if p1 > p2 then GT
                                 else EQ)
                         ms
         in (l,range w) : m1 ++ m2
posix [] (Star l r) = [(l,Nothing)]
posix w (Star l r) =
  case (filter (λxs → all (λ(_,m) → m /= []) xs) $
        map (λws→ map (λw → (w,posix w r)) ws) $
        split w) of
    [] → []
    ms → let (_,m) =
               last $ maximumBy
                      (λxs→ λys→
                      let ws1 = map snd xs
                          ws2 = map snd ys
                      in if ws1 < ws2 then LT
                          else if ws1 > ws2 then GT
                               else EQ)
                      ms
         in (l,range w) : m
posix _ _ = []
```

Figure 4. POSIX Submatching Specification

Next, we review the standard notion of a derivative of a regular expression and then show how to adapt derivatives to compute POSIX submatches.

4. Regular Expression Derivatives

Figure 5 implements the derivative operation in terms of the function `deriv`. The call `deriv c r` yields a regular expression where the character `c` has been consumed. For example, in case of `Ch` we check if the character equals the to be consumed character. If yes, we obtain the empty string `Eps`. Otherwise, we obtain the empty language `Phi` because consumption of `c` is impossible. In case of `Eps` and `Phi`, we always obtain `Phi` because no character can be consumed.

For `Choice`, we consume `c` from all alternatives `rs`. For `Pair`, only the leading expression `r1` consumes `c`, unless `r1` contains the empty string. Then the subsequent expression `r2` may consume `c` as well. For `Star`, we unroll the Kleene star once and consume `c` from the expression under the Kleene star.

Thus, we can elegantly define a function `member` to check if a string is an element of the language denoted by a regular expression `r`. We repeatedly apply the derivative function and check if the empty string is part of the final regular expression. In essence, function `member` builds a DFA on the fly where derivatives represent the states of the DFA.

```
containsEps :: Re l → Bool
containsEps (Choice _ rs) =
          or $ map containsEps rs
containsEps (Pair _ r1 r2) =
          (containsEps r1) && (containsEps r2)
containsEps Star{} = True
containsEps Ch{} = False
containsEps Eps{} = True
containsEps Phi{} = False

deriv :: Char → Re l → Re l
deriv c (Choice l rs) =
    Choice l $ map (deriv c) rs
deriv c (Pair l r1 r2)
   | containsEps r1 =
         Choice l [Pair l (deriv c r1) r2,
                   deriv c r2]
   | otherwise = Pair l (deriv c r1) r2
deriv c (t@(Star l r)) = Pair l (deriv c r)
                                (Star l r)
deriv c (Ch l c')
   | c == c'   = Eps l
   | otherwise = Phi
deriv _ Eps{} = Phi
deriv _ Phi = Phi


member :: String → Re l → Bool
member s r = containsEps $
           foldl (λr→ λc→ deriv c r) r s
```

Figure 5. Regular Expression Derivatives

5. Derivatives and Left-most Longest Submatches

For submatching, the natural choice is to record submatches as part of the regular expression. Specifically, annotations `l` are used to record the individual submatches. For the moment, we leave the concrete representation of `l` for submatches abstract.

Via type class `MatchCl` in Figure 6, we specify the functionality required to connect submatches to `l`. Method `extM` extracts the accumulated submatch from an annotation `l`. Method `mCh` extends an existing submatch with a given character. Method `dontCare` yields an annotation where the submatches don't matter. Methods `collapse` and `combine` provide variants of merging annotations. The purpose of these methods will become clear shortly.

Function `mDeriv` in Figure 6 is an extension of function `deriv` which additionally keeps track of submatches. A technicality is that we additionally supply the position information of the to be consumed character. Therefore, function `mDeriv` takes a value of type `(Int,Char)` where `deriv` only takes a `Char` value.

The main difference to `deriv` is that we apply `mCh` to extend the existing submatch accumulated in `l`. For example, in case of `Choice l rs`, we apply `mDeriv` to each element in `rs` and apply `mCh` to `l`.

A further difference arises in case of `Pair l r1 r2` where `r1` contains the empty string. Recall the definition of `deriv`:

```
deriv c (Pair l r1 r2)
   | containsEps r1 =
         Choice l [Pair l (deriv c r1) r2,
                   deriv c r2]
```

where we either consume `c` from `r1` or `r2`. In the second alternative, we simply drop the leading `r1` because `r1` contains the empty string.

In the submatching setting, we can't simply drop `r1` because annotations in regular expressions carry now submatch information. Simply keeping `r1` is not an option either because `r1` shall no further participate in the computation of submatches. The solution is to keep `r1` but "make the pattern empty". Function `mkE` traverses `r1`

```
class MatchCl l where
  extM     :: l → [SubMatch]
  mCh      :: (Int,Char) → l → l
  dontCare :: l
  collapse :: l → l → l
  combine  :: l → l → l

mDeriv :: MatchCl l ⇒ (Int,Char) → Re l → Re l
mDeriv c (Choice l rs) =
    Choice (mCh c l) $ map (mDeriv c) rs
mDeriv c (Pair l r1 r2)
   | containsEps r1  =
       Choice dontCare
              [Pair (mCh c l) (mDeriv c r1) r2,
               Pair (mCh c l) (mkE r1) (mDeriv c r2)]
   | otherwise = Pair (mCh c l) (mDeriv c r1) r2
mDeriv c (t@(Star l r)) =
      Pair (mCh c l) (mDeriv c r) (Star dontCare r)
mDeriv c@(_,ch) (Ch l ch')
   | ch == ch' = Eps (mCh c l)
   | otherwise = Phi
mDeriv _ Eps{} = Phi
mDeriv _ Phi = Phi

mkE :: Re l → Re l
mkE (Choice l rs) = Choice l $ map mkE rs
mkE (Pair l r1 r2) = Pair l (mkE r1) (mkE r2)
mkE (Star l r) = Star l $ mkE r
mkE Ch{} = Phi
mkE (e@Eps{}) = e
mkE Phi = Phi
```

Figure 6. Derivatives and Submatching

```
extract :: MatchCl l ⇒ Re l → [SubMatch]
extract (Choice l rs) =
       case (filter containsEps rs) of
           (r:_) → extM l ++ extract r
           _ → []
extract (r@(Pair l r1 r2))
  | containsEps r = extM l ++ extract r1 ++ extract r2
  | otherwise     = []
extract (Star l r) = extM l
extract (Eps l) = extM l
extract _ = []
```

Figure 7. Left-most Submatch Extraction

and replaces each pattern `Ch` by `Phi`. This ensures that from `r1` we only extract submatches which are connected to the empty string.

Another adjustment is the use of `dontCare`. In case of condition `containsEps r1`, we create two alternatives. We are only interested in the submatches of one of the alternatives and "don't care" about the submatches accumulated by `Choice` which combines both alternatives. We express this fact by attaching the `dontCare` annotation to `Choice`.

Like for `deriv`, we unroll (Star l r) once. Annotation `l` is moved to the resulting `Pair` where we additionally apply `mCh`. Similarly, as in case of `Pair`, we attach a `dontCare` annotation to the following `Star` pattern.

By construction, function `mDeriv` maximizes earlier submatches. To obtain the left-most longest match, we simply need to favor successful submatches which appear earlier, i.e. are left-most. That's what function `extract` in Figure 7 achieves.

In case of `Choice`, we select the left-most successful match among all alternatives. In case of `Pair`, we first check if there's a successful match, i.e. the pattern contains the empty string. Then,

```
dontCare_ = -1

instance MatchCl SubMatch where
   extM (x,m) = [(x,m)]
   mCh (i,c) (x,Nothing) = (x,Just (i,i))
   mCh (i,c) (x,Just (l,r)) = (x,Just (l,r+1))
   dontCare = (dontCare_,Nothing)
   collapse = error "not in use"
   combine = error "not in use"

instance Var SubMatch where
  v i = (i,Nothing)

subMatches :: Word → Re SubMatch → [SubMatch]
subMatches w r = select $ extract $
                 foldl (λr→ λc→ mDeriv c r) r w

select :: [SubMatch] → [SubMatch]
select xs = map last $
            groupBy (λ(i,_)→ λ(j,_) → i == j) $
            sort $
            filter (λ(l,_)→ l /= dontCare_) xs
```

Figure 8. A Simple POSIX Submatcher

we extract via `extM` the submatch associated with `Pair` and the submatches of the left and right component.

For `Star` we extract the submatch associated to the Kleene star but ignore the underlying pattern `r`. At first sight, it may seem sufficient to use the simpler definition

```
extract (Star m r) = []
```

After all, function `mDeriv` operates by unrolling the Kleene star and only consumes the character from the pattern underlying the Kleene star.

Indeed, the above simple version is sufficient for the simple POSIX method discussed in the up-coming section. Once we incorporate normalizations into our approach, discussed in the upcoming Section 7, the version in Figure 7 becomes necessary. The reason is that Kleene star annotations may carry submatches which result from patterns which have been removed due to some normalizations.

6. A Simple POSIX Submatch Method

To obtain an actual method for computation of POSIX submatches we must provide a concrete representation for annotations `l`. The natural choice is to instantiate `l` with the `SubMatch` type. Figure 8 provides the necessary instances for type classes `MatchCl` and `Var`. Methods `collapse` and `combine` are not in use. Both methods become important when integrating normalizations into our approach which will be discussed later.

Function `subMatches` repeatedly applies function `mDeriv`. On the final regular expression pattern, we apply `extract` which yields the list of left-most longest submatches. Function `select` ensures that in case of multiple submatches in case of Kleene star, we only keep the last one.

Algorithmically, the last submatch is selected as follows. We first throw away any submatch we don't care about. Then, the sorted list of submatches is grouped according to the label associated to subpatterns. In case a Kleene star has been unrolled only once and for all other subpatterns the associated groups are singleton. A group contains multiple elements if the Kleene star has been unrolled several time. We uniformly obtain the 'last' submatch by applying `last` on each group.

We summarize the above observations in the following proposition which states that `subMatches` computes POSIX submatches.

PROPOSITION 6.1 (POSIX Correctness). *For any word* `w` *and regular expression* `r` *we have that the following property holds:*

```
λ(w,r) → all (λxm → elem xm $ subMatches w $ conv r) $
         rmNoth $ posix w r
```

where

```
rmNoth = filter (λ(_,m) → case m of
                 Nothing → False
                 _       → True)
```

Function `conv` converts between the representation of regular expressions expected by `posix` and `subMatches`.

```
conv :: Var l ⇒ Re Int → Re l
conv (Choice l rs) = Choice (v l) $ map conv rs
conv (Pair l r1 r2) = Pair (v l) (conv r1) (conv r2)
conv (Star l r) = Star (v l) $ conv r
conv (Ch l c) = Ch (v l) c
conv (Eps l) = Eps $ v l
conv Phi = Phi
```

There are two technical points, we would like to highlight. The proposition ignores POSIX submatches connected to the empty string, i.e. `Nothing`. See function `rmNoth`. We only state inclusion and not equality among the set of non-`Nothing` submatches computed by `posix` and `subMatches`. The reason for both points are as follows.

Submatches connected to `Nothing` are ignored because the derivative formulation simply does not record such submatches.

We only state inclusion because `subMatches` is slightly more liberal in the interpretation of 'last' submatches. For example, for input word `[(1,'A'),(2,'B'),(3,'B')]` and the earlier regular expression `r1`, function `posix` yields

```
[(1,Just (1,3)),(2,Just (3,3)),(7,Just (3,3))]
```

whereas `subMatches` yields

```
[(1,Just (1,3)),(2,Just (3,3)),(4,Just (1,2)),
 (5,Just (1,1)),(6,Just (2,2)),(7,Just (3,3))]
```

Function `posix` strictly records the last iteration of a Kleene star pattern only whereas `subMatches` keeps a submatch unless there is a later submatch. Therefore, for `subMatches` we find the additional submatches `(4,Just (1,2))`, `(5,Just (1,1))` and `(6,Just (2,2)` which arise from the first iteration.

7. Normalization for Efficiency

Our current method for computing POSIX submatches has a serious problem. The size of derivatives may explode exponentially as the following example shows. For example, consider

```
-- (A∗, A∗)
r2 :: Var l ⇒ Re l
r2 = Pair (v 1) (Star (v 2) (Ch (v 3) 'A'))
                (Star (v 4) (Ch (v 5) 'A'))
```

We find [2]

```
pretty $ mDeriv (1,'A') (conv r2 :: Re SubMatch)
⟹
(((eps , ('A')∗) , ('A')∗) + ((phi)∗ , (eps , ('A')∗)))
```

and in a subsequent call

```
pretty $ mDeriv (2,'A') $
           mDeriv (1,'A') (conv r2 :: Re SubMatch)
⟹
(((((phi , ('A')∗) + (eps , (eps , ('A')∗))) , ('A')∗) +
  ((eps , (phi)∗) , (eps , ('A')∗))) + (((phi , (phi)∗) ,
   (eps , ('A')∗)) + ((phi)∗ ,
   ((phi , ('A')∗) + (eps , (eps , ('A')∗))))))
```

[2] For convenience, we make use of a `pretty` function whose straightforward definition we omit.

```
type Match = ([SubMatch], [SubMatch])

(+++) xs ys = map last $
              groupBy (λ(i,_)→ λ(j,_) → i == j) $
              sort $ xs ++ ys

instance MatchCl Match where
  extM (m1,m2) = m1 +++ m2
  mCh (i,c) (actives, inactives) =
      (map (λ(x,m) → case m of
                     Nothing → (x, Just (i,i))
                     Just (l,r) → (x, Just (l,r+1)))
           actives,
      inactives)
  dontCare = ([],[])
  collapse (a1,i1) (a2,i2) = (a1,i1+++i2+++a2)
  combine (a1,i1) (a2,i2) = (a1++a2,i1+++i2)

instance Var Match where
  v i = ([(i,Nothing)],[])
```

Figure 9. Active and Inactive Submatches

and so on. In each step, we obtain a new derivative where the size of derivatives grows exponentially.

The original paper on regular expression derivatives [1] identifies three rewrite rules to normalize derivatives:

$$(1)\ r + r \Rightarrow r \quad (2)\ r_2 + r_1 \Rightarrow r_1 + r_2 \text{ where } r_1 < r_2$$

$$(3)\ (r_1 + r_2) + r_3 \Rightarrow r_1 + (r_2 + r_3)$$

As shown in [1], the size of the normalized derivatives w.r.t. rewrite rules (1-3) is finite.

The work in [19] argues that further normalizations are required such the size of derivatives remains manageable:

$$(4)\ (\epsilon, r) \Rightarrow r \quad (5)\ ((r_1, r_2), r_3) \Rightarrow (r_1, (r_2, r_3)) \quad \text{etc}$$

There are two challenges we face to adapt these normalizations to the POSIX submatching setting.

The first challenge is correctness. For example, normalizations (2) and (5) can't be applied because under the POSIX submatching policy commutativity of regular expression choice and associativity of regular expression pair doesn't hold necessarily.

The second challenge is that essential normalizations such as (4) $(\epsilon, r) \Rightarrow r$ require us to store somewhere ϵ's submatches, e.g. by moving them to r. Hence, we must refine the structure of annotations `l` which is currently instantiated with `SubMatch`.

7.1 Active and Inactive Submatches

Our idea to address the second challenge is to connect annotations `l` to a pair of values of type `SubMatch`. We refer to the pair as `Match`. The first component carries the set of 'active' submatches and the second component the set of 'inactive' submatches. Initially, we have `([i,Nothing],[])` for each subpattern at position `i`. Inactive submatches are submatches which result from 'empty' patterns such as ϵ which have been removed as part of normalizations.

Figure 9 provides an instance on `Match` for type class `MatchCl`. Method `mCh` only extends the list of active submatches. Method `dontCare` is straightforward.

When extracting the accumulated submatches, we simply build the union of active and inactive submatches. Helper `(+++)` guarantees that only the last submatch is kept in case a subpattern has been matched multiple times.

Methods `combine` simply builds the union of the respective list of active and inactive submatches. For unioning inactive submatches, we make use of `+++` to maintain the invariant that only the last submatch is kept. Method `collapse` is similar but "deactivates" the active submatches of the second argument. The exact use

```
simp :: MatchCl l ⇒ Re l → Re l
simp (Pair l1 (Eps l2) r)
  | isPhi r   = Phi
  | otherwise = shift (collapse l1 l2) r
simp (Pair l r1 r2)
  | isPhi r1 || isPhi r2 = Phi
  | otherwise            = Pair l (simp r1) (simp r2)
simp (Choice l []) = Eps l
simp (Choice l [r]) = shift l r
simp (Choice l rs)
 | any isChoice rs =
    Choice l $
       foldl (λrs→ λr→ case r of
                Choice l rs2 → rs ++ (map (shift l) rs2)
                _            → rs ++ [r])
             [] rs
 | otherwise = Choice l $ nub $ filter (not.isPhi) $
                                map simp rs
simp (Star l1 (Eps l2)) = Eps $ collapse l1 l2
simp (Star l1 (Star l2 r)) = Star (combine l1 l2) r
simp (Star l r)
 | isPhi r   = Eps l
 | otherwise = Star l $ simp r
simp x = x

shift :: MatchCl l ⇒ l → Re l → Re l
shift l1 (Choice l2 rs) = Choice (combine l1 l2) rs
shift l1 (Pair l2 r1 r2) = Pair (combine l1 l2) r1 r2
shift l1 (Star l2 r) = Star (combine l1 l2) r
shift l1 (Ch l2 c) = Ch (combine l1 l2) c
shift l1 (Eps l2) = Eps $ combine l1 l2
shift l1 Phi = Phi
```

Figure 10. Integrating Normalization

of both methods will become clear when discussing normalization which we will do next.

7.2 Faithful Normalization

Figure 10 describes some normalization steps in terms of functions `simp`. The normalizations steps are carefully chosen such that POSIX submatches are retained.

Let's consider the first case

```
simp (Pair l1 (Eps l2) r)
  | isPhi r   = Phi
  | otherwise = shift (collapse l1 l2) r
```

If `r` equals the empty language, there's nothing to be done. We simply return `Phi` because a match doesn't exist for this case. Helper `isPhi` is defined in Figure 11. Otherwise, the capturing `Pair` and `Eps` can be dropped and we return `r`. Of course, we must retain `Pair`'s and `Eps`'s submatches.

Removal of `Eps` means that its active submatches recorded in `l1` become inactive because the pattern has been completely consumed. The active submatches connected to `Pair` shall of course remain active. Therefore, we apply `collapse l1 l2`. Then, the resulting pair of active and inactive submatches is then moved to `r` via function `shift`. Function `shift` always moves submatches to the nearest, i.e. top-most, position/annotation, by unioning the submatches via `combine`.

Before we proceed to discuss the remaining cases of `simp`, we consider an example to illustrate the workings of the various functions.

```
-- (A,A*)
r3 = Pair (v 1) (Ch (v 2) 'A')
                (Star (v 3) (Ch (v 4) 'A'))
```

Suppose, we build the derivative w.r.t. `(1,'A')`

```
mDeriv (1,'A') (conv r3 :: Re Match)
⟹
Pair ([(1,Just (1,1))],[])
       (Eps ([(2,Just (1,1))],[]))
       (Star ([(3,Nothing)],[]) (Ch ([(4,Nothing)],[]) 'A'))
```

Subpattern labeled 2 has been reduced to `Eps`. Normalization via `simp` removes the capturing `Pair` labeled 1 and `Eps`. We obtain

```
simp $ mDeriv (1,'A') (conv r3 :: Re Match)
⟹
Star ([(1,Just (1,1)),(3,Nothing)],[(2,Just (1,1))])
         (Ch ([(4,Nothing)],[]) 'A')
```

As we can see, label 1 is still active whereas 2 has been moved to the inactive set of submatches.

The above example also shows that in the presence of normalization we must extract the submatches connected to the Kleene star. See Figure 7.

We return to the definition of function `simp`. The second case for `Pair` deals with the case that the pattern denotes the empty language. Then, we simply return `Phi`. Otherwise, we apply normalizations on each subpattern.

The first two cases for `Choice` are straightforward. If the list of alternatives is empty, we replace the pattern by `Eps`. If there's only one alternative `r`, we return `r` but make sure to shift any submatches of `Choice` to `r`.

The third case either (a) flattens any nested `Choice` or (b) normalizes each alternative. In essence, (a) performs the following normalization:

$$(r_1 + r_2) + (r_3 + r_4) \Rightarrow r_1 + r_2 + r_3 + r_4$$

In the submatching setting, we additionally need to take care of the submatches connected to the flattened `Choice`s. We simply shift them to the alternatives. See

```
Choice l rs2 → rs ++ (map (shift l) rs2)
```

Besides normalization in case of (b), we also remove duplicates and patterns which are equivalent to the empty language. The Haskell Prelude function `nub` maintains the order of elements, i.e. keeps earlier patterns. This guarantees that we favor left-most submatches as demanded by POSIX.

The three cases for `Star` are straightforward. In the first case, an empty string under a Kleene star is normalized to the empty string. Submatches are shifted appropriately. In the second case, repeated Kleene stars are removed. The active submatches of both Kleene stars remain active. In the third case, we normalize the pattern underlying the Kleene star, unless the pattern is equivalent to the empty language. Then, we return `Eps l`. That is, we only keep the submatches of the Kleene star but drop any submatch results of `r`.

For all other cases, we simply return the regular expression. Importantly, normalization steps performed by `simp` preserve left-most longest submatches. We summarize this observation in the following proposition.

PROPOSITION 7.1 (Normalization Correctness). *For any expression* `r` *of type* `Re Match` *resulting from an application of* `mDeriv` *we have that the following property holds:*

```
λr → (rmNoth $ extract r) == (rmNoth $ extract $ simp r)
```

Similar to Proposition 6.1, we ignore 'Nothing' submatches.

Normalizations will be applied exhaustively until a fixpoint is reached. For this process to terminate, we must guarantee that the set of normalized derivatives obtained by exhaustive application of `simp` is finite.

Finiteness of normalized derivatives has already been shown for 'plain' regular expressions. See Theorem 5.2 in [1]. In fact, the set of normalized derivatives is at most exponential in the size of the initial regular expression. This result is implicit in the proof of Theorem 4.3(a) in [1].

```
isPhi :: Re l → Bool
isPhi (Choice _ rs) = and $ map isPhi rs
isPhi (Pair _ r1 r2) = (isPhi r1) || (isPhi r2)
isPhi Star{} = False
isPhi Ch{} = False
isPhi Eps{} = False
isPhi Phi{} = True

isChoice :: Re l → Bool
isChoice Choice{} = True
isChoice _ = False

instance Eq (Re l) where
  (==) (Choice _ a) (Choice _ b) = a == b
  (==) (Pair _ a b) (Pair _ c d) = a == c &&
                                   b == d
  (==) (Star _ a) (Star _ a) = a == b
  (==) (Ch _ a) (Ch _ b) = a == b
  (==) Eps{} Eps{} = True
  (==) Phi Phi = True
  (==) _ _ = False
```

Figure 11. Normalization Helper Functions

The essential condition is application of $(1)\ r+r \Rightarrow r$ which we also provide in `simp`. In our formulation, we omitted $(2)\ r_2+r_1 \Rightarrow r_1+r_2$ where $r_1 < r_2$ because it we may lose POSIX submatches. But this rule is not essential to achieve the desired results.

In our formulation, there's one case where we slightly depart from the original derivative formulation. Recall that in case of `Pair` where the first component contains the empty string, our formulation yields as one of the alternatives `Pair (mCh c l) (mkE r1) (mDeriv c r2)`. The 'plain' derivative formulation yields `deriv c r2`. [3] However, `simp` guarantees that `mkE r1` will be normalized to either `Phi` or `Eps`. Hence, the alternative `Pair (mCh c l) (mkE r1) (mDeriv c r2)` will eventually reduce to `mDeriv c r2` (we ignore any necessary updates to the set of active/inactive submatches). But then we have reached a similar normalized form as in the 'plain' regular expression case. Hence, finiteness follows also for our case.

We summarize the above observations in the following proposition.

PROPOSITION 7.2 (Normalized Derivatives are Finite). *For any expression* `r` *of type* `Re Int` *we have that the following property holds:*

```
λr →
  (toInteger $ length $ allDerivs r) ≤ 2 ^ (size r) + 1
```

where

```
allDerivs :: Re Int → [Re Match]
allDerivs r =
  let go rs =
       if (nub $ rs ++ next) == rs
       then rs else nub $ rs ++ next
       where
       next = nub $ rs ++ [ simpFix $ mDeriv (1,c) r |
                            c ← sigma r, r ← rs ]
  in go [conv r :: Re Match]
```

Function `sigma` extracts all non-duplicate characters from `r`. Function `size` computes the size of `r` by counting `1` for each constructor. For brevity, we omit the obvious definitions of `sigma` and `size`.

[3] Brzozowski's formulation would actually yield `Pair l (Eps l) (deriv c r2)` which can be reduced to our form via normalization rule $(4)\ (\epsilon, r) \Rightarrow r$.

```
data MatchM = MatchM
                (IM.IntMap (Maybe (Int,Int)))
                (IM.IntMap (Maybe (Int,Int)))


instance MatchCl MatchM where
   mCh (pos,_) (MatchM m1 m2) =
    MatchM (IM.map (λy → case y of
                           Nothing → Just (pos,pos)
                           (Just (l,r)) → Just (l,r+1))
                   m1)
           m2
   extM (MatchM m1 m2) =
        (map (λx→ (x,fromJust $ IM.lookup x m1)) $
             map fst $ IM.toList m1)
    +++ (map (λx→ (x,fromJust $ IM.lookup x m2)) $
             map fst $ IM.toList m2)

   collapse (MatchM a1 i1)
            (MatchM a2 i2) =
             MatchM a1
                    (IM.unionWith unionF i1 $
                     IM.unionWith unionF i2  a2)
             where
             unionF x y = if x < y then y else x
   combine (MatchM a1 i1)
           (MatchM a2 i2) =
            MatchM (IM.union a1 a2)
                   (IM.unionWith unionF i1 i2)
            where
            unionF x y = if x < y then y else x
   dontCare = MatchM IM.empty IM.empty

instance Var MatchM where
   v i = MatchM (IM.insert i Nothing $ IM.empty)
                IM.empty
```

Figure 12. IntMap for Efficient Submatch Storage and Access

7.3 POSIX Submatching with Normalization

Having established finiteness of our notion of derivatives, we can give an improved version of our POSIX submatcher where normalization is applied exhaustively after each derivative step.

```
simpFix :: MatchCl l ⇒ Re l → Re l
simpFix p = let q = simp p
            in if q == p
               then q
               else simpFix q

subMatchesS :: Word → Re Match → [SubMatch]
subMatchesS w r =
      (λe → e +++ []) $ extract $
      foldl (λr→ λc→ simpFix $
                     mDeriv c (simpFix r)) r w
```

Function `subMatchesS` yields POSIX submatches.

PROPOSITION 7.3 (POSIX Correctness with Normalization). *For any word* `w` *and regular expression* `r` *we have that the following property holds:*

```
λ(w,r) → all (λxm → elem xm $ subMatchesS w $ conv r) $
         rmNoth $ posix' w r
```

8. Explicit DFA Construction

Instead of building the POSIX submatch DFA on the fly, we build an explicit DFA on which we then apply the input string.

First, we introduce a more efficient data structure for storing submatches. We make use of the Haskell library IntMap (`IM`) which

```
mDeriV :: MatchCl l ⇒
          Char → Re l
          → (Re l, Int → Re MatchM → Re MatchM)
mDeriV c r = (mDeriv (1,c) r, λp→ λr→ mDeriv (p,c) r)

simP :: MatchCl l ⇒ Re l →
       (Re l, Re MatchM → Re MatchM)
simP r = (simp r, λr → simp r)

simPFix :: MatchCl l ⇒
           Re l → (Re l, Re MatchM → Re MatchM)
simPFix r =
  let go (rf@(r,f)) = let (r',g) = simP r
                      in if r == r' then rf
                         else go (r', g ∘ f)
  in go (r,id)

next :: MatchCl l ⇒ Char → Re l →
        (Re l, Char, Re l,
         Int → Re MatchM → Re MatchM)
next c r = let (r',f) = mDeriV c r
               (r'',g) = simPFix r'
           in (r,c,r'',λp → λm→ g $ f p m)
```

Figure 13. Derivative and Normalization Transformer

provides for an efficient implementation of maps from integer keys to values. Details are in Figure 12.

For each kind of submatch (active or inactive), we maintain a map from each label (i.e. integer key) to its current submatch. See data type definition `MatchM`. As before, we simply provide appropriate method definitions for instance `MatchCl MatchM` and can thus reuse the earlier defined functions `mDeriv`, `simp` etc.

Method definitions `mCh` applies `IM.map` to adjust all submatches within a map. For `extM`'s definition, we use `IM.lookup` for collection of submatches where `IM.toList` yields the domain of a map. In `collapse` and `combine`, we apply `IM.unionWith` to selectively union two maps where the combining function `unionF` guarantees that we keep the last submatch in case a subpattern has been matched multiple times.

For the explicit DFA construction, we use `Int` values to represent derivatives, i.e. DFA states. We assume that the initial regular expression is assigned to the state `1`. DFA transitions are mappings from `(Int,Char)` to `Int`. The first component of the pair represents the starting state and the second component the character on which the transition shall fire.

In addition, each transition carries a transformation function `Int -> Re MatchM -> Re MatchM`. The first argument is the position of the character in the input word. When supplied we obtain a function to transform the current submatches by extending the appropriate subpatterns with the given character at the given position. We use a map with key `(Int,Char)` to represent DFA transitions.

```
M.Map (Int,Char)
      (Int, Int → Re MatchM → Re MatchM)
```

Building of the DFA requires us to to derive the transformation function on submatches from the given set of functions `mDeriv` and `simp`. Figure 13 shows the necessary adjustments to obtain the required transformation functions.

Function `mDeriV` returns a pair consisting of the derivative and the transformation function. To obtain the derivative, we apply `mDeriv (1,c) r` where the supplied position `1` doesn't matter. Derivatives are used to build the states of the DFA whereas the transformation function `\p-> \r-> mDeriv (p,c) r` executes the actual submatching steps. Functions `simP` and `simpPFix` are derived similarly.

```
type DFA = M.Map (Int,Char)
                 (Int, Int → Re MatchM → Re MatchM)
buildDFA :: MatchCl l ⇒ Re l → DFA
buildDFA r =
  let go (rs,tble) curr_rs =
        let new_ts =
              map (λ(c,r) → next c r)
                  [(c,r) | c ← sigma r,
                           r ← curr_rs]
            new = nub [ r | (_,_,r,_) ← new_ts,
                            not (elem r rs)]
            as = rs ++ new
            map_as = zip as [1..]
            new_tble =
               foldl (λtble→ λ(r,c,r'',f) →
                       let s1 = fromJust $
                                lookup r map_as
                           s2 = fromJust $
                                lookup r'' map_as
                       in M.insert (s1,c) (s2,f) tble)
                     tble new_ts
        in if length new == 0
           then new_tble
           else go (as, new_tble) new
  in go ([r], M.empty) [r]

subMatchesD :: String → Re MatchM → [SubMatch]
subMatchesD w r =
   let dfa = buildDFA r
       patM s m pos [] = m
       patM s m pos (c:w') =
          let (s',f) = fromJust $ M.lookup (s,c) dfa
              m' = f  pos m
              pos' = pos + 1
          in patM s' m' pos' w'
   in select $ extract $ patM 1 r 1 w
```

Figure 14. POSIX SubMatch DFA Construction

Thus, we can easily build the required transitions via function `next`. The returned tuple yields the starting state in symbolic form as a regular expression, the triggering character, the resulting state and the transformation function which we obtain by composition of the transformations obtained from `mDeriV` and `simP`.

Construction of the DFA is now straightforward. We accumulate the set of states and transitions until we have reached a fixpoint. Recall that we assume that the initial expression is assigned to state `1`. See function `buildDFA` in Figure 14.

We run the DFA by the looking up the appropriate transition. The transformation function is supplied with the current position and then applied on the current submatch binding. Like before, we use `extract` and `select` to obtain the POSIX submatch from the final submatch binding. See `subMatchesD` in Figure 14.

Correctness follows directly from our earlier methods.

PROPOSITION 8.1 (POSIX DFA Correctness). *For any word* `w` *and regular expression* `r` *we have that the following property holds:*

```
λ(w,r) → all (λxm → elem xm $ subMatchesD w $ conv r) $
         rmNoth $ posix' w r
```

8.1 Time and Space Complexity

We investigate the time and space complexity of the POSIX DFA approach. Submatchings are stored as part of the regular expression. Hence, the space complexity is determined by the maximal size of normalized derivatives and and the maximal size of active/inactive submatches.

PROPOSITION 8.2 (Exponential Size of Normalized Derivatives). *For any expression* `r` *of type* `Re Int` *we have that the following property holds:*

```
λr → (maximum $ map size $ allDerivs r) ≤ 2 ^ (size r)
```

The above result follows from [23]. The work in [23] states a tighter bound of `size r * size r` if the following rewrite rule is added.

$$(r1, r) + (r2, r) \Rightarrow (r1 + r2, r)$$

The above rule avoids the exponential blow-up in case of the Kleene star. We don't support this rule because in our practical experience such cases don't seem to appear in practice.

The maximal size of active/inactive submatches per subpattern is bound by the size of the regular expression `r`. The list of actives carries at most the submatches connected to any subpattern `r`. Hence, the size of each list of active submatches is bound by the size of `r`. A similar argument applies to the list of inactive submatches. Recall that we keep at most one submatch in case there are multiple submatches connected to the same subpattern.

PROPOSITION 8.3 (Linear Space for Active/Inactive Submatches). *For any word* `w` *and regular expression* `r` *we have that the following property holds:*

```
λ(w,r)→ maxLen (size r) $
        foldl (λr→ λc→ simpFix $
                       mDeriv c (simpFix r)) r w
```

Function `maxLen` asserts that the size of the list of submatches is bound by `size r`. We omit the straightforward definitions.

From above, we can conclude the following.

PROPOSITION 8.4 (Space Usage of POSIX DFA). *The space usage of POSIX DFA only depends on the regular expression* `r` *and is independent of the input word* `w`. *Let* n *be* `size r`. *Then, the worst case storage requirements are as follows. Storage of the DFA is* $O(2^n)$ *and storage for submatchings stored in* `r` *is* $O(n * 2^n)$.

The worst-case time complexity is as follows.

PROPOSITION 8.5 (Time Complexity of POSIX DFA). *Let* `r` *be the regular expression and* `w` *be the input word where* $m =$`size w` *and* $n =$`size r`. *Then, the worst-case time complexity of POSIX DFA is* $O(m * log\ (2^n) * 2^n * n * n)$.

Operations on Maps are $O(\log\ k)$ where k is the size of the map whereas operations on IntMaps are $O(k)$ in the worst case. For each input character in `w` we lookup the appropriate transition in $O(log\ (2^n))$. In each step, we traverse the structure of the regular expression whose max size is (2^n). Each of the $O(n)$ subpatterns requires in the worst-case a shift/collapse/combine where each of these operations has complexity $O(n)$ due to the underlying IntMap.

The time complexity can be further improved by replacing Map/IntMap with a more efficient data structure. We will discuss some possible optimizations in the upcoming section.

9. Experimental Results

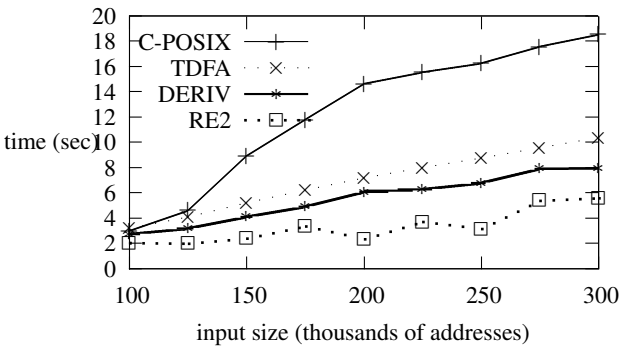
We benchmark our implementation against three contenders.

9.1 Contenders

- DERIV, an optimized Haskell-based implementation of derivative-based POSIX matching [16]. based on the approach described in Section 8.

  We apply a number of standard Haskell optimizations: (1) Enforcing strictness whenever possible, (2) using unboxed integers for the range of a submatch, (3) monomorphizing and inlining internal functions etc. (4) Instead of `String` we use `ByteString` [2] for representation of the input.

- TDFA, a Haskell-based implementation [22] of an adapted Laurikari-style tagged NFA.
- RE2, the google C++ re2 library [4] where for benchmarking the option `RE2::POSIX` is turned on.
- C-POSIX, the Haskell wrapper of the default C POSIX regular expression implementation [21].



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

Figure 15. General Test Suite Benchmark

The automata-based POSIX methods [6, 13] mentioned earlier are represented by RE2 and TDFA. For the approach described in [18] we couldn't obtain a concrete implementation.

For TDFA there exists a ML variant [10] which possibly improves over the Haskell version. The ML variant is still lacking some essential features [14] and therefore wasn't considered in our experiments.

9.2 Regular Expression Notation

For benchmarking we adopt the "external" syntax of regular expression found in languages such as Perl, Python and Java. We write `^r$` to indicate that the input has to be fully matched by the regular expression `r`. `Choice l r1 r2` is written in external syntax `r1|r2` and `Pair l r1 r2` is written `r1r2`. We write `(r)` to indicate that the submatching results of `r` shall be returned as part of the final match result. We write `r{N,M}` as a short hand for repeating `r` for at least `N` times and maximum `M` times. We write `[A-Z]` to denote a pattern accepting a single upper case letter and `.` to denote a pattern accepting an arbitrary symbol. For example,

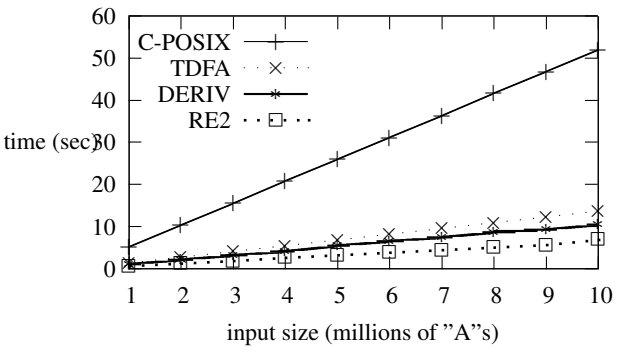
```
Star l1 (Choice l2 [Ch l3 'A',
                    Pair l4 (Ch l5 'A') (Ch l6 'B'),
                    Ch l7 'B'])
```

is written in external syntax `^((A)|(AB)|(B))*$`.

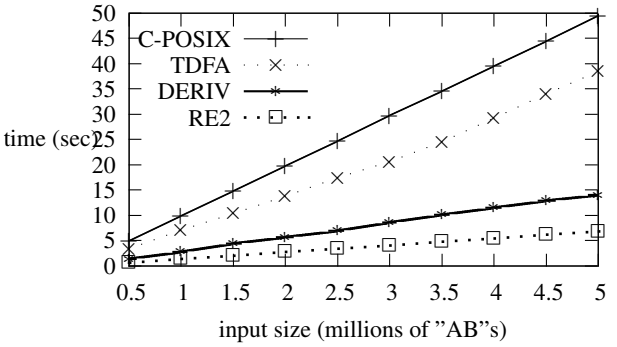
9.3 Benchmark Examples

Benchmarks are executed under Mac OS X 10.7.2 with 2.4GHz Core 2 Duo and 8GB RAM where results were collected based on the median over several test runs. First, we provide an overview of our benchmark examples and performance results. A discussion of results follows later. Due to space limitations, we only consider some characteristic examples. The complete set of results can be retrieved via [16].

General Test Suite consists of examples collected from various sources[7, 12], including some real world applications. A typi-

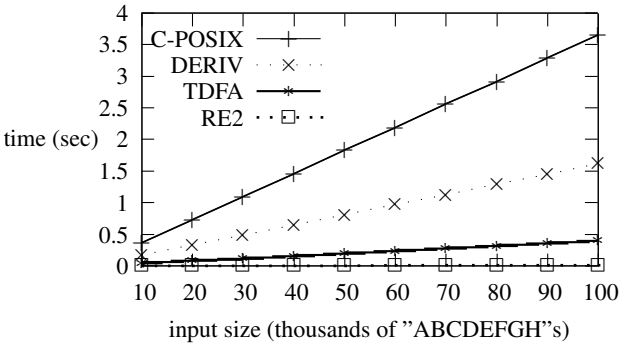


(a) Matching `^((A)|(AB)|(B))*$` with sequences of `A`s



(b) Matching `^((A)|(AB)|(B))*$` with sequences of `AB`s

Figure 16. Ambiguous Pattern Benchmark



(a) Matching pattern `^(((A)((B)((C)((D)((E)((F)(G)))))))(H))*$`

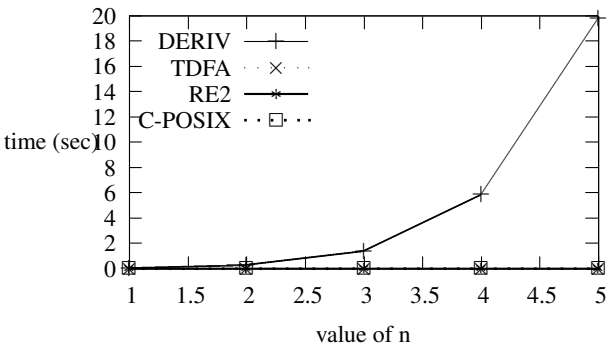
Figure 17. Deeply Nested Pattern Benchmark

cal example is given in Figure 15 where pattern

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

matches with a US address such as "Mountain View, CA 90410". The experiment was carried out by sending thousands of lines of unique addresses to the pattern. As a convention, the x-axis measures the size of the input (or a factor of the size of input), the y-axis measures the running time in seconds.

Ambiguous patterns consists of examples where depending on the input several matches may be possible. The purpose is to measure the extra cost required to compute the proper POSIX match. Figure 16(a) shows our running example where for the given input, a sequence of `A`s, the match is unambiguous. Figure 16(b) considers the ambiguous case as we have seen earlier.



(a) Compiling pattern `^((.?){1,n}Y)*X.*$`

Figure 18. DFA Size Benchmark

Deeply Nested Patterns consists of examples where blocks of concatenated patterns are grouped together such that we obtain a deeply nested pattern structure. Thus, we measure the performance of our data structures for storing submatches. An example is given in Figure 17.

DFA Size consists of examples where the size of the DFA exponentially increases compared to the NFA. The purpose is to measure the time our approach spends on construction of the DFA compared to the other NFA-based approaches.

9.4 Discussion

Overall our DERIV performs well and for most cases we beat TDFA and C-POSIX. RE2 is generally faster but then we are comparing a Haskell-based implementation against a highly-tuned C-based implementation.

For two classes (Deeply Nested Patterns and DFA Size) our current implementation is suboptimal. As we discuss below, there are a number of optimizations we yet need to integrate and which likely will further improve the performance of our approach.

Ambiguous Patterns The ambiguous pattern benchmark shows that our implementation performs in particular well for cases where computation of the POSIX match is non-trivial. As argued earlier, an advantage of our approach is that we don't require a comparison step to compute the POSIX match. To our surprise, RE2 and C-POSIX report incorrect results, i.e. non-POSIX matches, for some examples.

RE2 does not compute the correct POSIX matching when matching a sequence of `ABCDEFG`s with pattern

```
^((A)|(BCDEF)|(G)|(AB)|(C)|(D)|(E)|(EFG)|(FG))*$
```

A similar observation applies to C-POSIX for input `ABAAC` and pattern

```
^(((A|AB)(BAA|A))(AC|C))$
```

Regardless, we included RE2 and C-POSIX in our experiments. [4]

Deeply Nested Patterns The benchmark shows that our data structure for storing submatches is inefficient for deeply nested patterns.

[4] For RE2 there exists a prototype version [5] which appears to compute the correct POSIX match. We have checked the behavior for a few selected cases.

The main cause of inefficiency is due to our use of the Haskell library IntMap for combining and collapsing active/inactive submatches. See Figure 12. Unioning of two maps with either `union` or `unionWith` has time complexity $O(n + m)$ where n, m are the size of the to be unioned maps. This results in a quadratic cost (in the size of the expression) for the deeply nested benchmark example because the size of submatches grows. This is confirmed when considering deeper nested blocks of concatenated patterns.

We expect that the quadratic cost can be reduced to a linear cost by employing some more efficient, mutable data structures. For example, when selecting the 'last' match, instead of applying `unionWith`, we should simply override any existing match with the 'last' match.

Another cause of inefficiency is due to our structural representation of submatches based on the regular expression pattern. For our example, the `simp` function must traverse a 'deep' pattern structure to simplify terms and to shift submatches which results in a higher run-time cost.

One direction to improve the performance is to use a "flat" array-like mutable structure for storing submatches instead of our current tree-like structure. Thus, we avoid some trivial copy operations. In addition, shifting of submatches can be performed without having to traverse a possible complex structure.

Examples in this benchmark class are fairly contrived, hence, we don't consider it to be of a practical issue.

DFA Size It is well-known that the size of a DFA may be exponentially larger compared to the equivalent NFA. Such a worst-case example is given in Figure 18. Most of the time is spent on building the DFA. The actual time spent on building the match is negligible. Clearly, for such examples our DFA approach will be easily beaten by NFA approaches.

A surprisingly simple and efficient method to improve the performance of our approach is to apply some form of abstraction. Instead of trying to find matches for all subpattern locations, we may only be interested in certain locations. For instance, for pattern `^((.?){1,n}Y)*X.*$` we may only be interested in the subpattern locations `((.?){1,n}Y)*`, `X` and `.*` but don't care about any of the locations which are within these subpatterns, e.g. `(.?){1,n}`.

The key to improvement is to combine our DFA method with some NFA method. For all subpattern locations we don't care about, we rely on an NFA. For the locations we care about, we use a DFA. In essence, DFA states are now macro-states where within these states we are running an NFA.

We have already integrated this hybrid approach in some prototype where we use the partial derivative NFA method described in [24]. Via some syntactic extension, the user can specify which are the interesting locations for which a submatch shall be computed. The prototype is not geared for efficiency but we can already provide some numbers regarding the size of the automata. For the above pattern where n equals 4, our standard DFA method yields a DFA with 123 states whereas the hybrid approach has only 14 states. The number of states in the hybrid approach remain linear whereas the standard DFA states grows exponentially.

10. Conclusion

Regular expression derivatives [1] are an old idea and recently attracted again some interest in the context of lexing/parsing [17, 19]. POSIX submatching poses some additional challenges and as we have seen transferring the idea of derivatives to the POSIX submatching is non-trivial.

An important property of our approach is that correctness is fairly straightforward to establish. Our DFA-based approach offers a novel alternative to related NFA-based works [6, 13, 18]. Our experimental results confirm that for most cases our approach scales well and we can beat the TDFA Haskell implementation and remain close to RE2.

As discussed, there is quite a bit of scope for further improvements and we are currently working out the details.

References

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

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

[3] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc of ICFP'00*, pages 268–279. ACM, 2000.

[4] Russ Cox. re2 – an efficient, principled regular expression library. `http://code.google.com/p/re2/`.

[5] Russ Cox. NFA POSIX, 2007. `http://swtch.com/~rsc/regexp/nfa-posix.y.txt`.

[6] Russ Cox. Regular expression matching: the virtual machine approach - digression: Posix submatching, 2009. `http://swtch.com/~rsc/regexp/regexp2.html`.

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

[8] Glenn Fowler. An interpretation of the posix regex standard. `http://www2.research.att.com/~gsf/testregex/re-interpretation.html`.

[9] Institute of Electrical and Electronics Engineers (IEEE): Standard for information technology – Portable Operating System Interface (POSIX) – Part 2 (Shell and utilities), Section 2.8 (Regular expression notation), New York, IEEE Standard 1003.2 (1992).

[10] Chris Kuklewicz. ocaml-regex-tfa. `https://github.com/ChrisKuklewicz/ocaml-regex-tfa`.

[11] Chris Kuklewicz. Regex POSIX. `http://www.haskell.org/haskellwiki/Regex_Posix`.

[12] Chris Kuklewicz. The regex-posix-unittest package. `http://hackage.haskell.org/package/regex-posix-unittest`.

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

[14] Chris Kuklewicz, 2013. personal communication.

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

[16] Kenny Z. M. Lu and Martin Sulzmann. POSIX Submatching with Regular Expression Derivatives. `http://code.google.com/p/xhaskell-regex-deriv`.

[17] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: a functional pearl. In *Proc. of ICFP'11*, pages 189–195. ACM, 2011.

[18] Satoshi Okui and Taro Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In *Proc. of CIAA'10*, pages 231–240. Springer-Verlag, 2011.

[19] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[20] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

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

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

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

[24] Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression submatching using partial derivatives. In *Proc. of PPDP'12*, pages 79–90. ACM, 2012.

[25] Stijn Vansummeren. Type inference for unique pattern matching. *ACM TOPLAS*, 28(3):389–428, May 2006.