拡張正規表現マッチングの計算量解析

高橋 和也　南出 靖彦

文字列の検索等に広く用いられる正規表現マッチングはその多くがバックトラックに基づくアルゴリズムで実装されており，対象文字列の長さに対して線形時間でマッチングを完了できない場合がある．これを事前に検知するために，マッチングに要する計算量を静的解析する手法が複数の先行研究によって提案されている．本研究では先行研究を拡張し，先読みや後方参照など，現実のソフトウェアで使用されている拡張された正規表現に対しても解析が行える手法を提案する．さらに，既存の解析アルゴリズムを高速化し，より実用的な速度で解析を行える手法を提案する．そして，これらの改良を施した計算量解析のツールを Scala で実装し，Weideman らによる既存のツールよりも多くの正規表現が解析できることを実験により確認した．

Regular expression matching often used for searching and replacing strings are mostly implemented with backtracking. This causes that matching may not be completed in linear time in the length of a string. Previous works developed analyses to detect such regular expressions based on ambiguity analysis of nondeterrministic finite automata and growth rate analysis of string-to-tree transducers. We incorporate the techniques developed in the previous works and extend the analysis for extended regular expressions including lookahead and back-reference. We have also improved the core part of the analysis and implemented our analysis tool in Scala. It is shown that our tool can analyze more regular expressions than the existing tool of Weideman et al.

1　はじめに

正規表現マッチングは文字列の検索やフォーマットの検査などの用途で広く用いられている．マッチング処理を行うための正規表現ライブラリには様々な実装が存在するが，その多くはバックトラックに基づくアルゴリズムでマッチングを行っている．これは，正規表現の各部分にマッチング対象文字列のどの部分がマッチするかを深さ優先で探索し，一度マッチに成功する計算の分岐を発見したらそこで探索を打ち切る方法である．バックトラックによるマッチングの利点として，実装が比較的単純であることや拡張正規表現に対してもマッチングが行えることが挙げられる．ここでいう拡張正規表現とは，現実のソフトウェアで使用される，形式言語理論における通常の正規表現には無い先読みやキャプチャなどを含む正規表現のことである．一方で，バックトラックマッチングでは入力文字列の長さに対して線形時間でマッチングを完了できない場合があり，最悪時には指数時間かかってしまうという欠点がある．これにより，悪意を持ったユーザがマッチングに時間のかかる文字列を入力することで ReDoS と呼ばれる DoS 攻撃が成立してしまうことが指摘されている [11]．

このような脆弱性を事前に検知するために，マッチングに要する計算時間を静的解析する手法が提案されてきた．Kirrage らは与えられた正規表現に対する全探索マッチングが最悪時に指数時間かかるかどうかを判定する手法を提案し，後に Rathnayake と Thielecke によってバックトラックマッチングへ拡張された [6] [9]．杉山と南出は，バックトラックマッチ

Complexity Analysis of Extended Regular Expression Matching.
Kazuya Takahashi, Yasuhiko Minamide, 東京工業大学情報理工学院, School of Computing, Tokyo Institute of Technology.
コンピュータソフトウェア, Vol.0, No.0 (0), pp.0–0.
[研究論文] 2020 年 7 月 3 日受付.

ングが線形時間で行えるかどうかを判定する手法を提案し，これは後に中川らによって多項式時間のオーダまで決定できるように拡張された [14][17]．杉山らの手法はトランスデューサの出力サイズ増加率判定への帰着を用いており，等価な問題とその判定手法として Aho と Ullman による DT0L の増加率判定 [1] や，Weber と Seidl による NFA の曖昧性増加率判定 [15] などが知られている．これらの理論を用いたマッチングの計算量解析ツールとしては，Weideman らによる NFA の曖昧性増加率判定に基づいた実装がある [16][10]．

本研究ではこれらの先行研究をベースとして，より実用的な解析ツールとして利用できるようにすることを目標とし，主に二つの点で改良を行った．

一つ目は，拡張正規表現に対しても計算量の解析が行えるように既存の解析手法を拡張することであり，具体的には先読み，後読み，後方参照，先頭/末尾のマッチを扱えるように拡張を行う．先読みとは，マッチするかどうかだけを検査し入力文字列を消費しないパターンであり，現実の正規表現でよく利用されている拡張である．先読み付き正規表現の形式言語的な意味は森畑によって導入され，宮嵜らによってより自然な形で定式化された [18][8]．本研究では，先読み付き正規表現の形式言語的な意味と整合性が取れるように先行研究のマッチングの操作的意味論を拡張し，先読みを含む場合でもマッチングの計算量が判定できる手法を提案する．また，後読みと後方参照については，これらが含まれる正規表現のマッチングの計算量を正確に判定するのは本研究のアプローチでは困難であったため，計算量を上から評価する保守的な解析手法を与える．

二つ目は，解析の実行速度の改善である．既存の解析手法は，本質的に Weber と Seidl による NFA の曖昧性増加率判定のアルゴリズムに基づいている．このアルゴリズムではある特定の構造を見つけることで増加率を判定するが，構造を探索するために構成しなくてはならないグラフの大きさが NFA の状態数を n として $O(n^6)$ と非常に大きいことが実用的な解析ツールとして利用するにあたっての問題点であった．実際，Weideman らによる実装では正規表現が大きくなるとタイムアウトになる場合が多くなってしまっていた．この問題を解決するために，既存の探索アルゴリズムを改良した手法を提案する．提案手法では，増加率の判定には構造を局所的に探索するだけで十分であることに着目し，その局所的な構造の探索に必要な部分グラフのみをその都度構成することで高速化を図る．

これらの改良を組み込んだマッチングの計算量解析ツールを，Scala によって実装した．二つのテストデータを用いた実験の結果，本研究の実装は既存の解析ツールよりも多くの正規表現を解析できることが示された．本論文の構成としては，まず次節で比較実験の結果を示し，3 節以降で理論的な部分の詳細を主定理と共に説明する．証明の概略や必要な補題については付録 B を参照されたい．

2 実装と比較実験

本研究の拡張と高速化を組み込んだマッチングの計算量解析ツールを Scala で実装した †1．このツールでは，通常の正規表現の範囲で省略形として導入できる 1 回以上の繰り返し `r+`や `[a-z]` のような文字クラスなどに加え，以下の拡張正規表現を扱うことができる．

- 先頭/末尾のマッチ `^`，`$`
- 肯定/否定先読み `(?=r)`，`(?!r)`
- 肯定/否定後読み `(?<=r)`，`(?<!r)`
- 後方参照 `\1`，`\2`，...

ただし，後読みはマッチする文字列の長さに上限がある正規表現での後読みに制限する．また，後読みと後方参照を含む正規表現については計算量を上から評価する保守的な解析を行う．

ツールを実行して解析したい正規表現を入力することで，その正規表現に対するバックトラックマッチングの計算量のオーダが出力される．なお，計算量としてあり得るのは $\overset{\infty}{\Theta}(1), \overset{\infty}{\Theta}(n), \overset{\infty}{\Theta}(n^2), \ldots$ または $2^{\overset{\infty}{\Theta}(n)}$ のいずれかであることが知られており [1]†2，

†1 GitHub: https://github.com/minamide-group/regex-matching-analyzer

†2 $\overset{\infty}{\Theta}(f(n))$ は，下から評価する非標準的なオーダ記法 $\overset{\infty}{\Omega}(f(n)) = \{\, g(n) \mid \exists c.\forall n_0.\exists n \geq n_0.\, g(n) \geq cf(n) \,\}$

解析結果としてはこのどれかが出力される．さらに線形オーダではないと判定された場合には，マッチングの計算時間が実際にそのオーダで増加するような文字列のパターンも同時に出力する．例として，正規表現`^.*?<title>(.*?)</title>$`を入力すると次のような出力が得られる．

```
polynomial, degree 2
witness: < (<title>)^n i
```

これは，解析の結果 $\overset{\infty}{\Theta}(n^2)$ の計算量であると判定され，証拠となる文字列のパターンが `<(<title>)`^{n}`i` であることを示している．一般に，$\overset{\infty}{\Theta}(n^k)$ の計算量であると判定された場合に証拠となる文字列のパターンは $u_0 w_1^n u_1 w_2^n u_2 \cdots w_{k-1}^n u_{k-1}$ の形で出力され，繰り返し部分である w_i^n の n を大きくしていくとマッチングの計算時間が $\overset{\infty}{\Theta}(n^k)$ のオーダで増加する．

本研究の実装と，Weideman らによる既存の実装 [10] との比較実験を行った †3．テスト用の正規表現リストは，Kirrage らの実装におけるテストで用いられた RegExLib と Snort のテストデータ [12] を使用した．これらのリストには，重複を削除した上でそれぞれ 2,988 個，5,445 個の正規表現が含まれている．表 1，2 はそれぞれの解析結果について，その結果となった正規表現がいくつあったのかを示したものである．タイムアウトについては今回の実験では制限時間を 10 秒として行った．両方の実装で解析できた正規表現については，RegExLib の 6 個，Snort の 5 個を除き，解析結果が一致することを確認した．一致しない正規表現については，正規表現の構文解析等に関する既存実装のバグであった．

本研究の実装では先読み等の拡張が扱えるようになり，解析可能な正規表現が増えたことが見て取れる．本研究の実装でも未サポートである拡張としては，正規表現の内部で PCRE のオプションを指定する記法が多く見られた．

また既存の実装よりも多くの正規表現を扱えるようになったにもかかわらず，本研究で提案する高速化を施したことでタイムアウトの数も減らすことができた．解析できたものの詳細を見ると，本研究の実装では特に多項式オーダの計算量であると判定できたものが増加した．これは，脆弱性を生むような正規表現を多く検出できるようになったことを意味し，より実用的な解析ツールとなったことを示している．実験結果を詳細に調べると，既存実装でタイムアウトにならないが，本研究の実装でタイムアウトになる正規表現が 129 個あった．これらの原因としては以下が考えられる．

- どちらの実装も解析を NFA の曖昧性増加率判定に帰着しているが，帰着の方法が異なり効率に関しては単純に比較できない．
- 既存実装では，まず，全探索の場合の計算量を判定し，線形時間となった場合は線形時間と判定する効率化を行なっている．

表 3 は，保守的解析ができたものについてその内訳を示したものである．すべて指数オーダと判定されてしまうことはないものの，次数の高い多項式オーダと判定されたものが多く，保守的解析に関しては実用化に向けてさらなる改善が必要であることが示唆された．

3 準備

3.1 MonadPlus

MonadPlus は非決定的な選択と失敗を表現できるようにしたモナドの拡張である．本研究では，失敗を表す $\bot$ の対として成功を表す $\top$ も導入した非標準的な MonadPlus を使用し，正規表現マッチングの操作的意味論を定義する．

定義 3.1. 型構成子 $M : * \to *$ について，以下の演算が定義されているとする．

$$unit_M : \alpha \to M\,\alpha$$
$$\_ \gg\!=_M \_ : M\,\alpha \to (\alpha \to M\,\beta) \to M\,\beta$$
$$\top_M : M\,\alpha \qquad \bot_M : M\,\alpha$$
$$\_ +\!+_M \_ : M\,\alpha \to M\,\alpha \to M\,\alpha$$

これらが以下の (1)-(6) を満たす時，M は *MonadPlus*

を用いて $\overset{\infty}{\Theta}(f(n)) = O(f(n)) \cap \overset{\infty}{\Omega}(f(n))$ と定義される．

†3 Weideman らによる実装には誤った結果を出力してしまうバグが存在する．今回の実験は，発見したバグについては修正してから行った．

表 1　RegExLib の実験結果

解析結果	本研究	既存実装
線形オーダ	1,902	1,272
多項式オーダ	413	127
指数オーダ	140	96
保守的に解析可能	178	-
タイムアウト	129	527
未サポート	226	966

表 2　Snort の実験結果

解析結果	本研究	既存実装
線形オーダ	1,680	1,380
多項式オーダ	939	110
指数オーダ	3	2
保守的に解析可能	1,701	-
タイムアウト	1,105	1,480
未サポート	17	2,473

表 3　保守的解析ができたものの解析結果

解析結果	RegExLib	Snort
線形オーダ	29	4
$O(n^2)$	77	98
$O(n^3)$	36	818
$\geq O(n^4)$	15	781
指数オーダ	21	0

であるという．

$$(unit_M\ a) \gg\!=_M f = f\ a \tag{1}$$

$$m \gg\!=_M unit_M = m \tag{2}$$

$$m \gg\!=_M f \gg\!=_M g = m \gg\!=_M \lambda a.(f\ a \gg\!=_M g) \tag{3}$$

$$\top_M \gg\!=_M f = \top_M \tag{4}$$

$$\bot_M \gg\!=_M f = \bot_M \tag{5}$$

$$(m_1 +\!+_M m_2) \gg\!=_M f = (m_1 \gg\!=_M f) +\!+_M (m_2 \gg\!=_M f) \tag{6}$$

以降，考えている MonadPlus M が明らかな場合には，各演算の添字 M を省略する．

本研究では，木モナドを使用して意味論を定義する．α 型の要素からなる木 $tree\,\alpha$ を以下で定義する．

$$tree\,\alpha ::= \mathrm{Leaf}\,\alpha \mid \top \mid \bot \mid \mathrm{Or}\,(tree\,\alpha) \times (tree\,\alpha)$$

$tree$ は以下で定義される演算群によって MonadPlus となる．

$$unit_{tree}\ a = \mathrm{Leaf}(a) \qquad \mathrm{Leaf}(a) \gg\!=_{tree} f = f\ a$$

$$\top_{tree} = \top \qquad \top \gg\!=_{tree} f = \top$$

$$\bot_{tree} = \bot \qquad \bot \gg\!=_{tree} f = \bot$$

$$t_1 +\!+_{tree} t_2 = \mathrm{Or}(t_1, t_2)$$

$$\mathrm{Or}(t_1, t_2) \gg\!=_{tree} f = \mathrm{Or}(t_1 \gg\!=_{tree} f, t_2 \gg\!=_{tree} f)$$

α 型の値に対して真偽値を割り当てることで，α 型の要素からなる木全体の真偽値を自然に定めることができる．$t \in tree\,\alpha$ の割り当て $v : \alpha \to bool$ に対する真偽値 $t[v]$ を以下で定義する．

$$\mathrm{Leaf}(a)[v] = v(a)$$

$$\top[v] = True$$

$$\bot[v] = False$$

$$\mathrm{Or}(t_1, t_2)[v] = t_1[v] \vee t_2[v]$$

割り当て $v : \alpha \to bool$ は，$True$ を割り当てる要素を集めた集合 $V \subseteq \alpha$ によっても表すことができる．以降，集合による割り当ての表記も適宜使用する．

3.2　string-to-tree トランスデューサ

string-to-tree トランスデューサは文字列を受け取って木を出力する変換器であり，木を別の木に変換するトップダウン木トランスデューサの特殊な場合である．以降，単にトランスデューサと書いた場合には string-to-tree トランスデューサのことを指す．

木モナドの説明では代数的データ型として木を定義したが，これはトランスデューサを定義する上では扱いにくい．そこで，最初に数学的な形で木を再定義する．**ランク付きアルファベット** Γ とは，各 $\gamma \in \Gamma$ に対してランクと呼ばれる非負整数が定められたアルファベットのことである．ランク k を持つ記号 γ を $\gamma^{(k)}$ と書き，Γ の要素のうちランク k を持つ記号を集めた集合を $\Gamma^{(k)}$ と書く．ランク付きアルファベット Γ について，Γ の要素からなる木 T_Γ を以下を満たす最小の集合として定義する．

- $f \in \Gamma^{(k)}, t_i \in T_\Gamma\ (i = 1, 2, \ldots, k) \implies f(t_1, t_2, \ldots, t_k) \in T_\Gamma$

また，S を木の集合として $T_\Gamma(S)$ を以下を満たす最小の集合として定義する．

- $S \subseteq T_\Gamma(S)$
- $f \in \Gamma^{(k)}, t_i \in T_\Gamma(S)\ (i = 1,2,\ldots,k) \Longrightarrow f(t_1,t_2,\ldots,t_k) \in T_\Gamma(S)$

$\Omega \coloneqq \{\top^{(0)}, \bot^{(0)}, \mathrm{Or}^{(2)}\}$ と定義すると，*tree* α は α 型の値をランク 0 の記号とみなすことで $T_\Omega(\alpha)$ と同一視することができる．文字列 $w = a_1a_2\cdots a_n$ について，各入力文字をランク 1 の記号として扱い，終端を表すランク 0 の記号 \$ を付加して構成される木 $a_1(a_2(\cdots(a_n(\$))\cdots))$ を $w\$$ と書く．

定義 3.2. *string-to-tree* トランスデューサは 5 つ組 $\mathcal{T} = (Q,\Sigma,\Gamma,q_0,\Delta)$ である．ただし，Q は状態の有限集合，Σ は入力アルファベット，Γ はランク付き出力アルファベット，$q_0 \in Q$ は初期状態であり，Δ は以下の形式を持つ遷移規則の有限集合である．$Q_x \coloneqq \{q(x) \mid q \in Q\}$ とする．

$$q \xrightarrow{a} t \quad (q \in Q, a \in \Sigma, t \in T_\Gamma(Q_x))$$
$$q \xrightarrow{\$} t \quad (q \in Q, t \in T_\Gamma)$$

木 t の変数 x を木 t' で置き換えたものを $t[t'/x]$ と書く．トランスデューサ $\mathcal{T}$ の動作関係 $\to_\mathcal{T}$ を以下で定義する．

$$\frac{q \xrightarrow{a} t \in \Delta}{q(aw\$) \to_\mathcal{T} t[w\$/x]} \qquad \frac{q \xrightarrow{\$} t \in \Delta}{q(\$) \to_\mathcal{T} t}$$

$$\frac{f \in \Gamma^{(k)} \qquad t_i \to_\mathcal{T} t_i'\ (i = 1,2,\ldots,k)}{f(t_1,t_2,\ldots,t_k) \to_\mathcal{T} f(t_1',t_2',\ldots,t_k')}$$

$\to_\mathcal{T}^*$ を $\to_\mathcal{T}$ の反射推移閉包とする．本研究では，トランスデューサは決定的かつ全域的なもののみを使用する．トランスデューサ $\mathcal{T} = (Q,\Sigma,\Gamma,q_0,\Delta)$ の入力文字列 w による出力 $\mathcal{T}(w)$ を $q_0(w\$) \to_\mathcal{T}^* t$ となる唯一の $t \in T_\Gamma$ として定義する．

4 マッチングの計算量解析の定式化

この節では，正規表現マッチングの計算量解析を定式化するための理論について述べる．これらは複数の先行研究によって少しずつ改良されてきたものである [13] [14] [17] [19].

4.1 マッチングの操作的意味論

アルファベット Σ 上の正規表現 r を以下で定義する．

$$r ::= \emptyset \mid \varepsilon \mid a \mid rr \mid r|r \mid r^*$$

ただし，$a \in \Sigma$ とする．以降，アルファベットは Σ で固定し，Σ 上の正規表現全体の集合を *reg* で表す．正規表現の形式言語的な意味はそれが表す言語 $L(r)$ として定義されるが，マッチングを定式化する際には優先度を考慮した操作的意味論を与える必要がある．特にバックトラックマッチングでは，一度マッチに成功する計算の分岐を見つけたらそこで探索を打ち切るため，探索の順序はマッチングの計算量に影響を与える．本研究では，貪欲なマッチングの動作を模倣した操作的意味論を用いる．貪欲なマッチングでは，以下のように優先度の規則を定める．

- $r_1|r_2$ は，r_1 を優先する．
- r^* は，$rr^*|\varepsilon$ と展開される．すなわち，なるべく多く r にマッチさせることを試みる．

佐久間らは，正規表現マッチングの操作的意味論をリストモナドを用いた非決定性パーサとして定義した [13]．杉山と南出は，木モナドを用いて出力される木がマッチングの計算過程を表すようにこれを拡張した [14]．本研究でもこの木モナドによる定義を用いるが，ここでは山口によるより一般的な MonadPlus を用いた定義 [19] を述べておく．文字列の左商 $w_1^{-1}w$ と右商 ww_2^{-1} を，$w = w_1w_2$ の時にそれぞれ $w_1^{-1}w = w_2, ww_2^{-1} = w_1$ と定義し，それ以外の時は未定義であるとする．

定義 4.1. M を MonadPlus とする．正規表現 r に対して，マッチングの操作的意味論 $[\![r]\!] : \Sigma^* \to M\,\Sigma^*$ を以下で定義する．

$$[\![\emptyset]\!]w = \bot$$
$$[\![\varepsilon]\!]w = \mathit{unit}\, w$$
$$[\![a]\!]w = \begin{cases} \mathit{unit}\, a^{-1}w & (\text{if } a^{-1}w \text{ is defined}) \\ \bot & (\text{otherwise}) \end{cases}$$
$$[\![r_1r_2]\!]w = [\![r_1]\!]w \gg\!= [\![r_2]\!]$$
$$[\![r_1|r_2]\!]w = [\![r_1]\!]w +\!\!+ [\![r_2]\!]w$$
$$[\![r^*]\!]w = \left([\![r]\!]w \gg\!= \lambda w'. \begin{cases} \mathit{unit}\, w' & (\text{if } w' = w) \\ [\![r^*]\!]w' & (\text{otherwise}) \end{cases} \right) +\!\!+ \mathit{unit}\, w$$

$[\![r]\!]$ は文字列を受け取り，それを r にマッチさせたときの残りの文字列の候補を $M\,\Sigma^*$ の形で返す．例

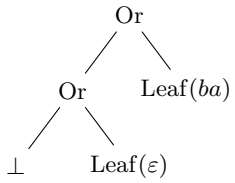


図 1　木モナドの場合の $[\![(ab)^*a]\!]aba$

として，$[\![(ab)^*a]\!]aba$ は以下のように計算される．

$$\begin{aligned}[\![(ab)^*a]\!]aba &= [\![(ab)^*]\!]aba \gg\!= [\![a]\!] \\ &= ([\![(ab)^*]\!]a +\!\!+ \mathit{unit}\, aba) \gg\!= [\![a]\!] \\ &= ((\bot +\!\!+ \mathit{unit}\, a) +\!\!+ \mathit{unit}\, aba) \gg\!= [\![a]\!] \\ &= (\bot +\!\!+ [\![a]\!]a) +\!\!+ [\![a]\!]aba \\ &= (\bot +\!\!+ \mathit{unit}\, \varepsilon) +\!\!+ \mathit{unit}\, ba\end{aligned}$$

$M = \mathit{tree}$ ならば，これは図 1 に示す木となり，このマッチングを計算する過程では失敗する分岐，空文字列が残る（つまり aba がマッチする）分岐，ba が残る（つまり a がマッチする）分岐がそれぞれ一つずつ存在することを表している．$[\![r^*]\!]$ の定義中で条件分岐を用いているのは，$\varepsilon \in L(r)$ である場合に無限ループを防ぐためであり，Perl の正規表現の動作を模倣している [13]．

残った文字列からマッチングの成否を判定する関数 $\chi : \Sigma^* \to M\,\emptyset$ を以下で定義する．

$$\chi = \lambda w. \begin{cases} \top & (\text{if } w = \varepsilon) \\ \bot & (\text{otherwise}) \end{cases}$$

木モナドを用いた場合，$[\![r]\!]w \gg\!= \chi$ は r に対する w の全探索マッチングの過程を表す木となり，この木の大きさはマッチングに要する計算時間に比例する．次の定理は，上記の操作的意味論が正規表現の形式言語的な意味と整合性が取れていることを示している．$[\![r]\!]w \gg\!= \chi$ の葉は $\top, \bot$ のいずれかであるため，空の割り当て $\emptyset$ を形式的に与えることで真偽値が定まることに注意．

定理 4.2 ([13])．$([\![r]\!]w \gg\!= \chi)[\emptyset] \iff w \in L(r)$

4.2　微分によるトランスデューサの構成

正規表現 r を一つ固定すると，文字列 w を受け取って木 $[\![r]\!]w \gg\!= \chi$ を出力するトランスデューサ $\mathcal{T}_r$ を構成することができる．中川らは，正規表現の微分を用いたこのトランスデューサの構成法を与えた [17]．正規表現の微分は Brzozowski によって導入された概念で，言語の左商を正規表現の上で計算する演算である [3]．中川らによるトランスデューサの構成法で使用される微分は，Antimirov によって提案された微分の拡張である偏微分を MonadPlus を返すように変更したものである [2] [19]．

定義 4.3. M を MonadPlus とする．$a \in \Sigma$ について，正規表現の a による微分 $\delta_a : \mathit{reg} \to M\,(\mathit{option}\ \mathit{reg})$ を図 2 で定義する．また，\$ による微分 $\delta_\$: \mathit{reg} \to M\,()$ を以下で定義する．

$$\begin{aligned} &\delta_\$(\emptyset) = \bot && \delta_\$(r_1 r_2) = \delta_\$(r_1) \gg\!= \lambda\_.\delta_\$(r_2) \\ &\delta_\$(\varepsilon) = \mathit{unit}\,() && \delta_\$(r_1|r_2) = \delta_\$(r_1) +\!\!+ \delta_\$(r_2) \\ &\delta_\$(a) = \bot && \delta_\$(r^*) = \delta_\$(r) +\!\!+ \mathit{unit}\,() \end{aligned}$$

$\delta_a(r)$ は，正規表現 r に文字 a をマッチさせた時の残りの正規表現の候補を返す．*None* は，その時点で a のマッチが完了しておらず，まだ a が消費されていないことを表す．$\delta_\$(r)$ は Brzozowski の微分における補助関数 ν に相当するもので，r が空文字列にマッチするかどうかを判定する．

正規表現 r が与えられた時に，微分を用いてトランスデューサ $\mathcal{T}_r = (Q, \Sigma, \Omega, r, \Delta)$ を構成する方法を述べる．準備として，関数 $\mathit{unpack} : \mathit{option}\ \mathit{reg} \to M\ \mathit{reg}$ を以下で定義する．

$$\mathit{unpack} = \lambda x.\mathit{case}\, x\ \mathit{of} \begin{cases} \mathit{Some}\, r \to \mathit{unit}\, r \\ \mathit{None} \to \bot \end{cases}$$

unpack は *option* を外す．*None* はまだ入力文字列が残っているにもかかわらず正規表現が尽きてしまったことを表すので，マッチングの失敗を表す $\bot$ に変換している．$\mathcal{T}_r$ は正規表現を状態として持ち，初期状態は r である．遷移規則は以下のように定義される．

- 任意の $r \in Q$ と $a \in \Sigma$ について，$r \xrightarrow{a} t \in \Delta$．ただし $t = \delta_a(r) \gg\!= \mathit{unpack}$．
- 任意の $r \in Q$ について，$r \xrightarrow{\$} t \in \Delta$．ただし $t = \delta_\$(r) \gg\!= \lambda\_.\top$．

状態 Q は最初に $Q = \{\, r \,\}$ としておき，遷移規則の右辺の木に葉として出現した正規表現を順次追加していく．Q は r から始めて微分によって現れる正規表現全体の集合となり，これは有限でその大きさは r の大きさに対して線形で抑えられることが知られて

$$
\begin{aligned}
\delta_a(\emptyset) &= \bot \\
\delta_a(\varepsilon) &= \mathit{unit}\ \mathit{None} \\
\delta_a(b) &= \begin{cases} \mathit{unit}\,(\mathit{Some}\,\varepsilon) & (\text{if } b = a) \\ \bot & (\text{otherwise}) \end{cases} \\
\delta_a(r_1 r_2) &= \delta_a(r_1) \gg\!= \lambda x.\mathit{case}\,x\ \mathit{of} \begin{cases} \mathit{Some}\,r' \to \mathit{unit}\,(\mathit{Some}\,r' r_2) \\ \mathit{None} \to \delta_a(r_2) \end{cases} \\
\delta_a(r_1|r_2) &= \delta_a(r_1) +\!\!+ \delta_a(r_2) \\
\delta_a(r^*) &= \left(\delta_a(r) \gg\!= \lambda x.\mathit{case}\,x\ \mathit{of} \begin{cases} \mathit{Some}\,r' \to \mathit{unit}\,(\mathit{Some}\,r' r^*) \\ \mathit{None} \to \mathit{unit}\ \mathit{None} \end{cases} \right) +\!\!+ \mathit{unit}\ \mathit{None}
\end{aligned}
$$

図 2 微分の定義

いる [2]．上記で構成されたトランスデューサ $\mathcal{T}_r$ について，以下の定理が成り立つ．

定理 4.4. 任意の $w \in \Sigma^*$ について，$\mathcal{T}_r(w) = [\![r]\!]w \gg\!= \chi$.

トランスデューサ $\mathcal{T}$ について，関数 $\mathcal{G}_\mathcal{T}$ を $\mathcal{G}_\mathcal{T}(n) \coloneqq \max_{|w|=n} |\mathcal{T}(w)|$ で定義する．$\mathcal{T}$ の出力サイズ増加率とは，$\mathcal{G}_\mathcal{T}$ のオーダのことをいう．定理 4.4 によって，全探索マッチングの計算量解析はトランスデューサ $\mathcal{T}_r$ の出力サイズ増加率を求める問題に帰着される．

全探索マッチングではなく，最初にマッチに成功したところで探索を打ち切るバックトラックマッチングを模倣するためには，出力された木を適切に変換する必要がある．この変換はボトムアップ木トランスデューサ $\mathcal{B}^{\mathrm{prune}}$ によって表現でき，$\mathcal{T}_r$ と $\mathcal{B}^{\mathrm{prune}}$ の合成は先読み付きトランスデューサによって実現できる．これらの詳細については付録 A を参照されたい．

5 拡張正規表現の計算量解析

拡張正規表現を扱えるように，先行研究の操作的意味論と微分を拡張する．本研究では，拡張正規表現として先読み，後読み，後方参照，先頭/末尾のマッチを扱う．先読みとは，マッチするかどうかだけを検査し入力文字列を消費しないパターンである．後読みは，今読んでいる位置のすぐ手前の文字列が指定された正規表現とマッチするかを検査するパターンであり，先読みと同じく文字列は消費しない．後方参照は，対応するキャプチャグループの中身にマッチした文字列と同じ文字列にのみマッチするパターンである．

正規表現に以下のように肯定/否定先読み $(\overrightarrow{\&}r, \overrightarrow{!}r)$，肯定/否定後読み $(\overleftarrow{\&}r, \overleftarrow{!}r)$，キャプチャグループ $((r)_x)$，後方参照 $(\backslash x)$，先頭/末尾のマッチ $(\hat{}, \$)$ を追加したものを拡張正規表現と呼ぶ．

$$r ::= \cdots \mid \overrightarrow{\&}r \mid \overrightarrow{!}r \mid \overleftarrow{\&}r \mid \overleftarrow{!}r \mid (r)_x \mid \backslash x \mid \hat{} \mid \$$$

キャプチャグループの名前に用いることができる変数の集合を Var で表す．各 $x \in \mathit{Var}$ について，正規表現全体で名前 x がついたキャプチャグループは高々1つであることを仮定し，また，キャプチャグループは先読み，後読みの中の現れないものとする [†4]．

拡張正規表現の形式言語的な意味 $B(r)$ は $\Sigma^* \times \Sigma^* \times \Sigma^*$ の部分集合として定義される．これは，宮嵜らによる先読み付き正規表現の意味 [8] を後読みに拡張したものであり，それぞれの成分は第一成分がマッチする文字列の前に許される文字列，第二成分がマッチする文字列，第三成分がマッチする文字列の後ろに許される文字列を表している．$R, S \in 2^{\Sigma^* \times \Sigma^* \times \Sigma^*}$ に対して連接演算を $R \cdot S = \{\, (x, yz, w) \mid (x, y, zw) \in R, (xy, z, w) \in S \,\}$ で定義すると，$2^{\Sigma^* \times \Sigma^* \times \Sigma^*}$ は $I \coloneqq \Sigma^* \times \{\, \varepsilon \,\} \times \Sigma^*$

†4 2 節の実験で用いた正規表現はすべてこの条件を満たしていた．先読みの中でキャプチャグループが使われると，表明付き木と State モナドの組み合わせでは正しく正規表現マッチングの動作を模倣できない．また，後読みの操作的意味論では状態の更新を行なっていない．

を単位元とするモノイドをなし，べき乗とスター演算が $R^0 = I, R^{n+1} = R^n R, R^* = \bigcup_{n\geq 0} R^n$ と定義される．これを踏まえて，$B(r)$ を以下で定義する．なお，後方参照については正規言語の範囲を超えるため，形式言語的な意味は考えないことにする．

$$B(\emptyset) = \emptyset \qquad B(a) = \Sigma^* \times \{a\} \times \Sigma^*$$
$$B(\varepsilon) = I \qquad B(r_1|r_2) = B(r_1) \cup B(r_2)$$
$$B(r^*) = B(r)^* \qquad B(r_1 r_2) = B(r_1)B(r_2)$$
$$B(\overrightarrow{\&}r) = \{(x,\varepsilon,yz) \mid (x,y,z) \in B(r)\}$$
$$B(\overrightarrow{!}r) = I \setminus \{(x,\varepsilon,yz) \mid (x,y,z) \in B(r)\}$$
$$B(\overleftarrow{\&}r) = \{(xy,\varepsilon,z) \mid (x,y,z) \in B(r)\}$$
$$B(\overleftarrow{!}r) = I \setminus \{(xy,\varepsilon,z) \mid (x,y,z) \in B(r)\}$$
$$B(\hat{}) = \{\varepsilon\} \times \{\varepsilon\} \times \Sigma^* \qquad B(\$) = \Sigma^* \times \{\varepsilon\} \times \{\varepsilon\}$$

拡張正規表現 r が表す言語 $L(r)$ を $L(r) := \{y \mid (\varepsilon, y, \varepsilon) \in B(r)\}$ で定義する．

先頭/末尾のマッチは，任意の一文字にマッチする正規表現 . を用いてそれぞれ $\hat{} = \overleftarrow{!}., \$ = \overrightarrow{!}.$ と表すことができるので，これらは先読み，後読みの特殊な場合と考えることができる．ただし先頭のマッチについては，実装では後読みとしては扱わずに特別扱いしている．これについての詳細はこの節の最後で述べる．

5.1 部分マッチの模倣

拡張正規表現を扱う前に，現実の正規表現ライブラリで採用されている部分マッチの模倣について考える．正規表現と文字列が与えられた時，文字列全体が正規表現にマッチした時のみマッチ成功とみなすマッチングの方法を完全マッチと呼び，ある部分文字列が正規表現にマッチすればマッチ成功とみなす方法を部分マッチと呼ぶ．これまでの操作的意味論は完全マッチを模倣するものであったが，本研究で行うマッチングの計算量解析を現実の挙動と合わせるために部分マッチを模倣する操作的意味論を与える．部分マッチを模倣するには，先頭の文字をいくつか無視してマッチングを開始できるようにし，さらに末尾に文字列が残っていてもマッチ成功とみなすようにすればよい．

前者は正規表現の先頭に $.^{*?}$ を付け加えることで実現できる．ただし，$r^{*?}$ は非貪欲なスターと呼ばれ，$r^{*?} = \varepsilon|rr^{*?}$ と展開される．非貪欲にすることで，最左マッチを模倣することができる．

一方で後者は，残った文字列からマッチングの成否を判定する関数 χ の代わりに $\chi^{\mathrm{prefix}} = \lambda\_.\top$ と定義される χ^{prefix} を用いればよい．これに伴い，トランスデューサを構成する際に使用する *unpack* を以下で定義される $unpack^{\mathrm{prefix}}$ に変更する必要がある．

$$unpack^{\mathrm{prefix}} = \lambda x.case\, x\, of \begin{cases} Some\, r \to unit\, r \\ None \to \top \end{cases}$$

unpack からの変更点は，*None* の場合に $\bot$ ではなく $\top$ を返すようになったことであり，これにより正規表現が尽きて文字列が余ったときにもマッチ成功として扱われる．*unpack* の代わりに $unpack^{\mathrm{prefix}}$ を用いて正規表現 r から構成されたトランスデューサを $\mathcal{T}_r^{\mathrm{prefix}}$ と書くと，定理 4.2，定理 4.4 はそれぞれ以下のように修正される．

定理 5.1. $([\![r]\!]w \gg\!= \chi^{\mathrm{prefix}})[\emptyset] \iff w \in L(r)\Sigma^*$

定理 5.2. 任意の $w \in \Sigma^*$ について，$\mathcal{T}_r^{\mathrm{prefix}}(w) = [\![r]\!]w \gg\!= \chi^{\mathrm{prefix}}$.

5.2 先読みへの拡張

MonadPlus では非決定的な選択を表現することができ，最初の選択肢を試して，成功したらそこで終了，失敗したら次の選択肢を試すというバックトラックの動作を模倣することができた．これは「OR」の短絡評価に相当する．一方で，正規表現の先読みの動作としては．まず先読み部分にマッチするかを検査し，成功したら後続の正規表現へのマッチを試み，失敗したらその時点でその計算の分岐は失敗とする．この動作は「AND」の短絡評価に相当する．以上の考察から，先読みに対応するためには「AND」を表現できる木を導入する必要がある．そこで，肯定先読み，否定先読みに対応する「AND」として Assert, AssertNot を持つ表明付き木を定義する．モナドとして考える都合上，表明付き木は表明部分とそれ以外の部分で葉の型を別々に指定する．

定義 5.3. 2 つの型 α と β からなる**表明付き木**

$atree\,\alpha\,\beta$ を以下で定義する．

$$\begin{aligned} atree\,\alpha\,\beta \quad ::= \quad & \mathrm{Leaf}\,\beta \mid \top \mid \bot \\ \mid \quad & \mathrm{Or}\,(atree\,\alpha\,\beta) \times (atree\,\alpha\,\beta) \\ \mid \quad & \mathrm{Assert}\,(atree\,\alpha\,\alpha) \times (atree\,\alpha\,\beta) \\ \mid \quad & \mathrm{AssertNot}\,(atree\,\alpha\,\alpha) \times (atree\,\alpha\,\beta) \end{aligned}$$

α は先読み部分で使う型を，β はそれ以外の通常部分で使う型を表す．MonadPlus による記法と揃えるために，Assert, AssertNot を構成する以下の演算子を定義しておく．

$$assert\,t_1\,then\,t_2 = \mathrm{Assert}(t_1, t_2)$$
$$assertNot\,t_1\,then\,t_2 = \mathrm{AssertNot}(t_1, t_2)$$

$atree$ に対する bind 演算は 2 種類考えることができる．一つは木のすべての葉を変換するもの，もう一つは表明部分にある葉を除いて変換するものである．後者は，連接演算の操作的意味論を定義する際に必要となる．2 つの bind 演算

$$\_ \gg\!= \_ : atree\,\alpha\,\alpha \to (\alpha \to atree\,\alpha'\,\alpha') \to atree\,\alpha'\,\alpha'$$
$$\_ \overset{\mathrm{rhs}}{\gg\!=} \_ : atree\,\alpha\,\beta \to (\beta \to atree\,\alpha\,\beta') \to atree\,\alpha\,\beta'$$

を定義する．これらは $\mathrm{Leaf}, \top, \bot, \mathrm{Or}$ に対しては $tree$ の bind 演算と同様に定義され，Assert に対しては以下のように定義される．AssertNot についても同様である．

$$\mathrm{Assert}(t_1, t_2) \gg\!= f = \mathrm{Assert}(t_1 \gg\!= f, t_2 \gg\!= f)$$
$$\mathrm{Assert}(t_1, t_2) \overset{\mathrm{rhs}}{\gg\!=} f = \mathrm{Assert}(t_1, t_2 \overset{\mathrm{rhs}}{\gg\!=} f)$$

$tree\,\alpha \coloneqq atree\,\alpha\,\alpha$ と再定義する．また，α を一つ固定して，$tree_\alpha\,\beta \coloneqq atree\,\alpha\,\beta$ と定義する．この時，$tree$ は $\gg\!=$ によって MonadPlus となり，$tree_\alpha$ は $\overset{\mathrm{rhs}}{\gg\!=}$ によって MonadPlus となる．再定義された $tree\,\alpha$ について，木の真偽値の定義は以下のように拡張される．

$$\mathrm{Assert}(t_1, t_2)[v] = t_1[v] \wedge t_2[v]$$
$$\mathrm{AssertNot}(t_1, t_2)[v] = \neg t_1[v] \wedge t_2[v]$$

5.2.1 操作的意味論

操作的意味論は $tree_{\Sigma^*}\,\Sigma^*$ 上で定義する．先読みについては以下で定義される．

$$[\![\overrightarrow{\&}r]\!]w = assert\,([\![r]\!]w)\,then\,(unit\,w)$$
$$[\![\overrightarrow{!}r]\!]w = assertNot\,([\![r]\!]w)\,then\,(unit\,w)$$

これは，先読み $\overrightarrow{\&}r$ に遭遇したら r にマッチするかどうかの検査 $[\![r]\!]w$ を行い，成功したら文字列を消費せずにマッチングを続行するという動作を模倣している．$tree_{\Sigma^*}\,\Sigma^*$ 上で定義するため，連接とスターの場合に $\overset{\mathrm{rhs}}{\gg\!=}$ を用いることに注意．これは，連接を処理する際に先読み部分と干渉しないようにするためである．$\gg\!=$ を使ってはいけないことを示す例として，$[\![(\overrightarrow{\&}a)ab]\!]ab$ を考える．連接で $\gg\!=$ を使うとすると，$[\![(\overrightarrow{\&}a)ab]\!]ab = [\![\overrightarrow{\&}a]\!]ab \gg\!= [\![ab]\!]$ となる．$[\![\overrightarrow{\&}a]\!]ab$ を計算すると図 3 のようになり，これは先読み部分の検査（Assert の左側）では正規表現 a に文字列 ab をマッチさせて文字列 b が残っているが，マッチした a は消費されないので後続の計算（Assert の右側）としては文字列 ab が残っていることを表している．したがって，$[\![\overrightarrow{\&}a]\!]ab \gg\!= [\![ab]\!]$ を計算すると，左側は $[\![ab]\!]b = \bot$，右側は $[\![ab]\!]ab = \mathrm{Leaf}(\varepsilon)$ となるので結果は図 4 のとおりであり，先読みが失敗したことになってしまう．$\overset{\mathrm{rhs}}{\gg\!=}$ を使用すれば，左側を $\mathrm{Leaf}(b)$ のままにしておくことができ，先読みは成功して残りの文字は ε であるという正しい結果が得られる．

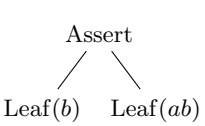


図 3 木 $[\![\overrightarrow{\&}a]\!]ab$

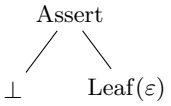


図 4 $\gg\!=$ を使用した場合の木 $[\![(\overrightarrow{\&}a)ab]\!]ab$

先読みを含む場合でも操作的意味論と形式言語的な意味の整合性を示すことができるが，注意することとして，先読み部分の検査では正規表現に残りの文字列全体がマッチする必要はなく，ある prefix がマッチすればマッチ成功とみなされる．したがって，残った文字列からマッチングの成否を判定する関数として部分マッチを模倣するために導入した χ^{prefix} を用いる必要があり，主張としては定理 5.1 が先読みを含む正規表現 r についても新しい bind 演算と $L(r)$ の定義で成り立つ．なお，すべての葉をマッチングの成否に変換する必要があるので，定理中の bind 演算は $\gg\!=$ を用いる．

5.2.2 微分とトランスデューサ

文字による微分は $tree_{option\,reg}\,(option\,reg)$ 上で，\$ による微分は $tree_{()}\,()$ 上で定義する．先読みについ

ては以下で定義される．

$$\delta_a(\overrightarrow{\&}r) = \mathit{assert}\,\delta_a(r)\,\mathit{then}\,(\mathit{unit}\,\mathit{None})$$
$$\delta_\$(\overrightarrow{\&}r) = \mathit{assert}\,\delta_\$(r)\,\mathit{then}\,(\mathit{unit}\,())$$
$$\delta_a(\overrightarrow{!}r) = \mathit{assertNot}\,\delta_a(r)\,\mathit{then}\,(\mathit{unit}\,\mathit{None})$$
$$\delta_\$(\overrightarrow{!}r) = \mathit{assertNot}\,\delta_\$(r)\,\mathit{then}\,(\mathit{unit}\,())$$

微分についても連接を処理する際には $\overset{\mathrm{rhs}}{\gg\!=}$ を用いることに注意．一方で，トランスデューサの遷移規則を定義する際に，微分の後に適用する bind 演算には $\gg\!=$ を用いる．これにより，定理 5.2 が先読みを含む正規表現 r についても成り立つ．

5.3 後読みと後方参照への拡張

マッチングの操作的意味論の定義は，文字列を与えると正規表現に対してそれをマッチさせた後の残りの文字列の候補を返すものであった．先読みまでの拡張ではこれで情報が十分であったが，後読みや後方参照へ拡張する場合には既に読み終えた文字列やキャプチャグループでマッチした文字列の情報を参照できるように状態を持つ必要がある．そこで，任意のモナドに状態を追加できる State モナド変換子を利用する [7]．$\mathit{StateT} : * \to (* \to *) \to * \to *$ を以下で定義する．

$$\mathit{StateT}\,\rho\,M\,\alpha = \rho \to M\,(\alpha \times \rho)$$

ρ は状態の型を表す．ρ を一つ固定した時，任意の MonadPlus M について $\mathit{StateT}\,\rho\,M$ は以下で定義される演算によって MonadPlus となる．

$$\top_{\mathit{StateT}\,\rho\,M} = \lambda\_.\top_M \qquad \bot_{\mathit{StateT}\,\rho\,M} = \lambda\_.\bot_M$$
$$\mathit{unit}_{\mathit{StateT}\,\rho\,M}\,a = \lambda s.\mathit{unit}_M\,(a, s)$$
$$m \gg\!=_{\mathit{StateT}\,\rho\,M} f = \lambda s.m\,s \gg\!=_M \lambda(a, s').f\,a\,s'$$
$$m_1 +\!+_{\mathit{StateT}\,\rho\,M} m_2 = \lambda s.(m_1\,s +\!+_M m_2\,s)$$

状態の更新を扱う関数 $\mathit{update} : (\rho \to \rho) \to \mathit{StateT}\,\rho\,M\,\rho$ を以下で定義する．

$$\mathit{update}\,f = \lambda s.\mathit{unit}_M(s, f\,s)$$

5.3.1 操作的意味論

後読み，後方参照を含む正規表現に対する操作的意味論は，状態として変数から文字列への部分関数 $\mathit{Var} \hookrightarrow \Sigma^*$ を用いて $\mathit{StateT}\,(\mathit{Var} \hookrightarrow \Sigma^*)\,\mathit{tree}\,\Sigma^*$ 上で定義する．なお簡単のため，ここでは先読みを含まない場合を考え，先読みの拡張で必要になる atree を用いたモナドではなく，拡張前の木モナドを用いて操作的意味論を与える．部分関数 s について，x に対応する値を v に変更したものを $s[x \mapsto v]$ と書く．また，behind をこれまでに読んだ文字列を格納するための特別な変数とする．操作的意味論を図 5 に示す．この意味論では，文字を消費した時とキャプチャグループのマッチ後に状態を更新するようにしており，後読みと後方参照を処理する際にそれを参照する．後読みの場合には言語の所属判定を暗黙に行っており，生成される木にはこれに要する計算ステップ数は含まれていない．したがって，所属判定が定数時間で行えることを仮定しないとマッチング時間と比例しない木が生成されてしまう．このため，後読みはマッチする文字列の長さに上限がある正規表現での後読みに制限する．後読み，後方参照を加えた拡張正規表現に対する操作的意味論では，全探索に対応する木は以下の式で得られる．

$$\pi_1([\![r]\!]w\,s) \gg\!= \chi$$

ここで，π_1 は $\mathrm{Leaf}(a, s)$ の第一成分を取り出す $\mathit{tree}\,(\alpha \times \rho)$ から $\mathit{tree}\,\alpha$ への関数とする．

5.3.2 置換による保守的解析

後読み，キャプチャグループ，後方参照に対する操作的意味論では無限の状態空間を用いているため，有限状態のトランスデューサに変換することはできない．そこで，これらの正規表現を別のものに置き換え，マッチングの計算量を上から評価する形の保守的な解析を行うことを考える．

まず，後読みについて考える．後読みを必ず成功するとみなす方法や，必ず失敗するとみなす方法では計算量を上から評価することができない．例として，$(\overleftarrow{\&}r)r_1|r_2$ を考える．$\overleftarrow{\&}r$ が必ず成功するとみなすと，その後 r_1 でマッチに成功する場合には r_2 へのマッチングが行われないので，r_2 のマッチングに時間がかかる場合には計算量が少なくなってしまう．反対に，$\overleftarrow{\&}r$ が必ず失敗するとみなすと r_1 へのマッチングは行われなくなるため，r_1 のマッチングに時間がかかる場合に同様に計算量が少なくなってしまう．

また，後方参照については対応するキャプチャグループの中身で置き換えることが考えられるが，単純に置き換えてしまうとマッチし得る文字列が増え，

$$[\![a]\!]w = \begin{cases} update\ \lambda s.s[behind \mapsto s(behind)\,a] \gg\!= \lambda\_.unit\,a^{-1}w & (\text{if } a^{-1}w \text{ is defined}) \\ \bot & (\text{otherwise}) \end{cases}$$

$$[\![\overleftarrow{\&}r]\!]w = update\ id \gg\!= \lambda s. \begin{cases} unit\,w & (\text{if } (s(behind), w) \in \{\,(xy, z) \mid (x, y, z) \in B(r)\,\}) \\ \bot & (\text{otherwise}) \end{cases}$$

$$[\![\overleftarrow{!}\,r]\!]w = update\ id \gg\!= \lambda s. \begin{cases} unit\,w & (\text{if } (s(behind), w) \notin \{\,(xy, z) \mid (x, y, z) \in B(r)\,\}) \\ \bot & (\text{otherwise}) \end{cases}$$

$$[\![(r)_x]\!]w = [\![r]\!]w \gg\!= \lambda w'.(update\ \lambda s.s[x \mapsto ww'^{-1}] \gg\!= \lambda\_.unit\,w')$$

$$[\![\backslash x]\!]w = update\ id \gg\!= \lambda s. \begin{cases} update\ \lambda s.s[behind \mapsto s(behind)\,s(x)] \gg\!= \lambda\_.unit\,s(x)^{-1}w & (\text{if } s(x)^{-1}w \text{ is defined}) \\ \bot & (\text{otherwise}) \end{cases}$$

図 5 後読み，後方参照を含む正規表現に対する操作的意味論

やはり計算量が変化してしまう場合がある．

計算量を上から評価するためには，変換前の正規表現の成功と失敗のパターンを包含したような動作をする正規表現に置換しなくてはならない．このために，形式的に新しい正規表現 $\diamond$ を導入する．$\diamond$ は ε のように振る舞うが，その先でマッチングに成功する計算の分岐があったとしても $\diamond$ まで戻って改めて失敗とみなしてマッチングを続行することを要請する．

$\diamond$ の意味論を定義するために，木の要素として $\bot^{(1)}$ を追加し，bind 演算を $\bot(t) \gg\!= f = \bot(t \gg\!= f)$ と定義する．これを用いて，$\diamond$ の操作的意味論を以下で定義する．

$$[\![\diamond]\!]w = \bot(unit\,w)$$

また，微分は $\delta_a(\diamond) = \bot(unit\,None), \delta_{\$}(\diamond) = \bot(unit\,())$ と定義される．この $\diamond$ を用いて，正規表現を変換する関数 *approx* を以下のように定義する．

- 後読みを $\diamond$ に置換する．
- 後方参照 $\backslash x$ を対応するキャプチャグループが $(r)_x$ のとき，$r\diamond$ に置換する．

ただし，キャプチャグループの中に後読みと後方参照が現れないことを仮定する．この条件は実際には少し緩めることができ，後読みを含む場合には置き換える際に削除すればよく，後方参照を含む場合については依存関係がサイクルを持たなければ上記の変換を必要なだけ繰り返せばよい．なお，先読みも拡張に含める場合には先読みの中に後読みと後方参照が現れないことが条件に加わる．

approx(r) には後読み，後方参照が含まれないので，State モナドを用いない従来の操作的意味論によって意味が定義され，従来の微分によってトランスデューサを構成することができる．なお，後方参照がすべて消去されているためキャプチャグループは意味をなさず，単純に無視してよい．この置換によって計算量を上から評価できることは，以下の定理によって保証される．ここで，$\mathcal{B}^{\mathrm{prune}}$ は全探索マッチングの過程を表す木をバックトラックマッチングの過程を表す木に変換するボトムアップ木トランスデューサである．$\mathcal{B}^{\mathrm{prune}}$ の定義と定理の証明については付録を参照されたい．

定理 5.4. 後読みと後方参照を加えた拡張正規表現 r と $w \in \Sigma^*$ に対して，以下が成り立つ．

$$|\mathcal{B}^{\mathrm{prune}}(\pi_1([\![r]\!]_{ST}w\,s_0) \gg\!= \chi)| \leq |\mathcal{B}^{\mathrm{prune}}([\![approx(r)]\!]w \gg\!= \chi)|$$

ただし，s_0 は *behind* に対してのみ $s_0(behind) = \varepsilon$ と定義される部分関数とし，$[\![\_]\!]_{ST}$ は State モナドを用いた操作的意味論，$[\![\_]\!]$ は従来の State モナドを用いない操作的意味論を表すものとする．

5.4 先頭のマッチ

先頭のマッチ ^ は，後読みの特殊な場合とみなして置換し保守的な解析を行うこともできる．しかし，^ にマッチするかどうかは，現在読んでいる部分が入力

文字列の先頭か否かだけを情報として保持しておくことで判定することができる．すなわち，状態として *bool* を持つ State モナド上で微分を定義し，トランスデューサを構成することで置換することなく先頭のマッチを扱うことができる．この場合，操作的意味論では以下のように文字を消費した際に先頭か否かを表す状態を *False* に更新し，^の場合に状態を読んで分岐する．

$$[\![a]\!]w = \begin{cases} \mathit{update}\,\lambda\_.\mathit{False} \gg\!= \lambda\_.\mathit{unit}\,a^{-1}w \\ \qquad\qquad (\text{if } a^{-1}w \text{ is defined}) \\ \bot \qquad\qquad (\text{otherwise}) \end{cases}$$

$$[\![\hat{}]\!]w = \mathit{update}\,\mathit{id} \gg\!= \lambda s. \begin{cases} \mathit{unit}\,w & (\text{if } s = \mathit{True}) \\ \bot & (\text{otherwise}) \end{cases}$$

また，微分は以下のように定義される．

$$\delta_a(b) = \begin{cases} \mathit{update}\,\lambda\_.\mathit{False} \gg\!= \lambda\_.\mathit{unit}\,\varepsilon & (\text{if } a = b) \\ \bot & (\text{otherwise}) \end{cases}$$

$$\delta_a(\hat{}) = \mathit{update}\,\mathit{id} \gg\!= \lambda s. \begin{cases} \mathit{unit}\,\mathit{None} & (\text{if } s = \mathit{True}) \\ \bot & (\text{otherwise}) \end{cases}$$

$$\delta_\$(\hat{}) = \mathit{update}\,\mathit{id} \gg\!= \lambda s. \begin{cases} \mathit{unit}\,() & (\text{if } s = \mathit{True}) \\ \bot & (\text{otherwise}) \end{cases}$$

その他の場合については，$\mathit{StateT}\ \mathit{bool}\ M$ を MonadPlus と見ることにより図 2 の定義を用いる．

トランスデューサはこの State モナド上での微分を用いて構成される．状態は正規表現 r と State モナドの状態 s（すなわち真偽値）の組である．遷移規則 Δ は以下のように定義される．

- 任意の $(r,s) \in Q$ と $a \in \Sigma$ について，$(r,s) \xrightarrow{a} t \in \Delta$．ただし $t = (\delta_a(r) \gg\!= \mathit{unpack})\,s$．
- 任意の $(r,s) \in Q$ について，$(r,s) \xrightarrow{\$} t \in \Delta$．ただし $t = (\delta_\$(r) \gg\!= \lambda\_.\top)\,s$．

State モナドの状態 s の取り得る値が有限種類なので，遷移規則も有限になることが保証される．上記の方法で初期状態 (r,s) から構成されたトランスデューサを $\mathcal{T}_{(r,s)} = (Q, \Sigma, \Omega, (r,s), \Delta)$ とすると，$\mathcal{T}_{(r,s)}$ は以下の性質を満たす．

定理 5.5. $\mathcal{T}_{(r,s)}(w) = ([\![r]\!]w \gg\!= \chi)\,s$

マッチングの際には文字列の先頭からマッチを開始するので，正規表現 r が与えられた時にトランスデューサ $\mathcal{T}_{(r,\mathit{True})}$ を構成してその出力サイズ増加率を判定することで，先頭のマッチを置換せずに計算量の解析を行うことができる．

6 トランスデューサの出力サイズ増加率判定とその高速化

マッチングの計算量解析を行うためには，正規表現から構成したトランスデューサの出力サイズ増加率を決定すればよい．この問題は基本的かつ重要な問題であり，いくつかの等価な問題が知られている．本研究では，Weber と Seidl による NFA の曖昧性増加率判定のアルゴリズム [15] を改良して高速化する手法を提案する．

6.1 先行研究

トランスデューサ $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, \Delta)$ からラベル付き有向グラフ $G = (Q, E)$ を構成する．ただし，E は次のように定義する．

- 各 $q \xrightarrow{a} t \in \Delta$ に対して，t の葉を重複を含めて並べたものが $q_1 q_2 \cdots q_n$ の時，$i = 1, 2, \ldots, n$ について辺 (q, a, q_i) を E に追加する．

E は多重集合とする．したがって $q_i = q_j$ となる相異なる i, j が存在した場合には同じ辺が重複して E に現れることになる．この構成法は，Aho と Ullman による GSDT から DT0L への変換 [1] を言い換えたものである．彼らは，この変換によってトランスデューサの出力サイズ増加率判定が DT0L の増加率判定に帰着できることを示した †5．DT0L の増加率判定と等価な問題として，NFA が与えられた時に入力の長さに対する実行パスの総数の増加率を決定する問題である NFA の曖昧性増加率判定問題がある．Weber と Seidl は，NFA，すなわちラベル付き有向グラフ上で特定の構造を探索することで NFA の曖昧性増加率が決定できることを示した [15]．探索すべき構造は図 6，7 のように表される二種類であり，それぞれ *EDA* 構造，*IDA* 構造と呼ぶ．図の波打った矢印はパスを表し，そのラベル w はパスに含まれる辺のラベ

†5 Aho と Ullman は DT0L の増加率が判定可能であることも示したが，その判定法は実用的なものではなかった．

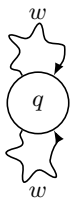


図 6 q 上の **EDA** 構造

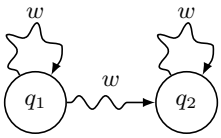


図 7 q_1 から q_2 への **IDA** 構造

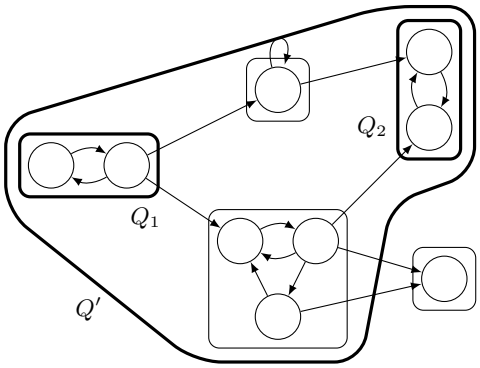


図 8 局所的な **IDA** 構造の探索に十分な集合

ルを順に並べた文字列である．q 上の EDA 構造は q から出発して自身へ戻る，同じラベルを持った 2 つの異なるパスのことであり，q_1 から q_2 への IDA 構造は q_1, q_2 から出発してそれぞれ自身へ戻るパス，q_1 から q_2 へのパスからなる，同じラベルを持った 3 つのパスのことである．

彼らは，直積構成の要領で作られた新しいグラフ上で強連結成分分解を行うことでこの二種類の構造が探索できることを示した．$G = (Q, E)$ が与えられているとする．まず，EDA 構造の探索法を述べる．$G_2 = (Q^2, E_2)$ を以下で構成する．

$$E_2 = \{((q_1, q_2), (q_1', q_2')) \in Q^2 \times Q^2 \mid \exists a \in \Sigma.\, (q_i, a, q_i') \in E\ (i = 1, 2)\}$$

G_2 は，G の同じラベルを持つ 2 つのパスが G_2 の 1 つのパスに対応するように構成している．この時，G における q 上の EDA 構造は，G_2 において (q, q) から出発して $q_1 \neq q_2$ となるノード (q_1, q_2) を訪れて (q, q) に戻ってくるパスに対応する．このようなパスは，G_2 を強連結成分分解することによって容易に発見することができる．

ここで，NFA の曖昧性増加率判定の場合には多重辺が存在しなかったので，これを特殊なケースとして処理する必要がある．q_1 から q_2 への辺 (q_1, a, q_2) が 2 つ以上存在したとする．もし，q_2 から q_1 へ戻るパスが存在すれば，明らかに q_1 から q_1 に戻ってくる同じラベルがついた 2 通りのパスが存在する．したがって，この場合には EDA 構造を持つ．逆に，q_2 から q_1 へ戻るパスが存在しなければこの多重辺 (q_1, a, q_2) が EDA 構造に関与することはないので重複を無視してよい．

次に，IDA 構造の探索法を述べる．$G_3 = (Q^3, E_3)$ を以下で構成する．

$$E_3 = \{((q_1, q_2, q_3), (q_1', q_2', q_3')) \in Q^3 \times Q^3 \mid \exists a \in \Sigma.\, (q_i, a, q_i') \in E\ (i = 1, 2, 3)\}$$

G_3 の構成法は，G_2 の構成法からノードを 3 つ組に変更しただけである．この時，G における q_1 から q_2 への IDA 構造は，G_3 において (q_1, q_1, q_2) から (q_1, q_2, q_2) へのパスに対応する．このパスを発見するためには，少し工夫が必要である．E_3' を以下で定義し，$G_3' = (Q^3, E_3')$ とおく．

$$E_3' = E_3 \cup \{\, ((q_1, q_2, q_2), (q_1, q_1, q_2)) \mid q_1, q_2 \in Q, q_1 \neq q_2 \,\}$$

E_3' は，E_3 に (q_1, q_2, q_2) から (q_1, q_1, q_2) へ戻る辺を追加したものである．このようにすると，G_3' を強連結成分分解することで所望のパスを発見できる．

6.2 高速化

Weber と Seidl は，与えられたグラフ G を強連結成分分解し，同じ強連結成分に属するノードを併合して得られる有向非巡回グラフ上での動的計画法によって増加率を決定する手法を提案した [15]．しかし，彼らの手法では構造の探索に必要となる G_2, G_3 のサイ

ズが大きくなってしまうことが実装上の問題点であった．特に，G_3 は G のノード数を n とすると $O(n^6)$ のサイズを持ち，既存の実装ではこれがボトルネックとなって大きい正規表現に対してはタイムアウトとなってしまっていた．

これを解決するために，アルゴリズムを改良してより効率的に EDA 構造，IDA 構造が探索できる手法を提案する．アイデアとしては，増加率の判定に必要となるのは局所的な EDA 構造，IDA 構造の探索のみであることに着目し，G_2 や G_3 の全体を構成せず，局所的な構造の探索に必要な部分グラフのみをその都度構成する．これにより，巨大なグラフを構成せずに済むようになり，解析速度を高速化することができる．

まず EDA 構造については，動的計画法では G のある強連結成分 Q' を一つ固定してその中に EDA 構造が存在するかを判定できればよかった．強連結成分の定義から，Q' に属さない状態に一度到達したパスは Q' 内の状態には戻って来られないため，パスの探索は Q' の中でのみ行えばよい．したがって，Q' の中で EDA 構造を探索するだけならば G_2 のノードは Q'^2 で十分となる．

次に，IDA 構造の探索について考える．動的計画法では，G の 2 つの強連結成分 Q_1, Q_2 が与えられた時に，ある $q_1 \in Q_1$ と $q_2 \in Q_2$ について q_1 から q_2 への IDA 構造が存在するかを判定できればよかった．G の 2 つの強連結成分 Q_1, Q_2 を固定する．この時，G_3 のノードのうち IDA 構造の探索に必要な部分を考える．まず，自身に戻る 2 つのパスの探索は，EDA 構造の場合と同様にそれぞれ強連結成分 Q_1, Q_2 の中でのみ探索すればよい．もう一つのパスについては，Q_1 から到達可能かつ Q_2 へ到達可能な強連結成分全体の和集合 Q' の中で探索すればよい．これは図 8 のように図示される．以上の考察から，G_3 のノードは $Q_1 \times Q' \times Q_2$ で十分となる．

また，実装上の最適化として，2 つの強連結成分 Q_1, Q_2 の間に明らかに IDA 構造が存在しない場合には G_3 の構成自体をスキップすることができる．明らかに IDA 構造が存在しない場合というのは，Q_1, Q_2 のどちらかがシングルトンで，かつその状態が自己辺を持たない時である．

7 おわりに

本研究では，正規表現マッチングの計算量解析ツールをより実用的なものにするために，先読み等を含む拡張正規表現を扱えるように既存手法を拡張し，さらに解析速度の高速化手法を提案した．本研究の手法を組み込んで実装したツールは，既存のナイーブな実装よりも多くの正規表現を解析可能になったことが実験により示され，実用化に近づいたと考えられる．

また理論的な面では，モナドを用いたマッチングの操作的意味論を先読み，後読み，後方参照へと拡張したが，拡張後も形式言語的な意味との整合性が保たれることを示した．後読み，後方参照については特殊な正規表現への置換によって保守的な解析を行う手法を提案し，マッチングの計算量が上から評価できていることを示した．

保守的解析については，現状では次数の高い多項式オーダと判定されるものが多く，実用化に向けてより精度の高い手法や，実用的には大きな問題にならない程度の制限を課した上で正確な解析が行える手法を与えることが今後の課題である．

謝辞 本研究の一部は JSPS 科研費 19K11899, 20H04162 の助成を受けたものである．

参考文献

[1] Alfred V Aho and Jeffrey D Ullman. Translations on a context free grammar. *Information and Control*, Vol. 19, No. 5, pp. 439–475, 1971.

[2] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, Vol. 155, No. 2, pp. 291–319, 1996.

[3] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, Vol. 11, No. 4, pp. 481–494, 1964.

[4] Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical systems theory*, Vol. 10, No. 1, pp. 289–303, 1976.

[5] Joost Engelfriet and Sebastian Maneth. Macro tree translations of linear size increase are MSO definable. *SIAM Journal on Computing*, Vol. 32, No. 4, pp. 950–1006, 2003.

[6] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *International Confer-*

ence on Network and System Security, pp. 135–148. Springer, 2013.
[7] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 333–343. ACM, 1995.
[8] Takayuki Miyazaki and Yasuhiko Minamide. Derivatives of regular expressions with lookahead. *Journal of Information Processing*, Vol. 27, pp. 422–430, 2019.
[9] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics (extended). *arXiv preprint arXiv:1405.7058*, 2014.
[10] RegexStaticAnalysis. `https://github.com/NicolaasWeideman/RegexStaticAnalysis`.
[11] Regular expression denial of service - ReDoS. `https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS`.
[12] RXXR: Regular expression denial of service (REDoS) static analysis. `https://www.cs.bham.ac.uk/~hxt/research/rxxr.shtml`.
[13] Yuto Sakuma, Yasuhiko Minamide, and Andrei Voronkov. Translating regular expression matching into transducers. *Journal of Applied Logic*, Vol. 10, No. 1, pp. 32–51, 2012.
[14] Sugiyama Satoshi and Minamide Yasuhiko. Checking time linearity of regular expression matching based on backtracking. 情報処理学会論文誌プログラミング (PRO), Vol. 7, No. 3, pp. 1–11, 2014.
[15] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theoretical Computer Science*, Vol. 88, No. 2, pp. 325–349, 1991.
[16] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *International Conference on Implementation and Application of Automata*, pp. 322–334. Springer, 2016.
[17] 中川みなみ, 南出靖彦. バックトラックによる正規表現マッチングの時間計算量解析. 情報処理学会プログラミング研究会第 107 回プログラミング研究発表会, 2016.
[18] 森畑明昌. 先読み付き正規表現の有限状態オートマトンへの変換. コンピュータ ソフトウェア, Vol. 29, No. 1, pp. 1_147–1_158, 2012.
[19] 山口隆成. 正規表現の微分を用いた線形時間マッチングアルゴリズム. 筑波大学情報学群情報科学類卒業研究論文, 2015.

A バックトラックの模倣

マッチングの操作的意味論は全探索マッチングの過程を表す木を構築する．この木をバックトラックマッチングの過程を表す木へ変換する操作はボトムアップ木トランスデューサによって表現できる [14]．

定義 A.1. ボトムアップ木トランスデューサは 4 つ組 $\mathcal{B} = (Q, \Sigma, \Gamma, \Delta)$ である．ただし，Q は状態の有限集合，Σ, Γ はそれぞれ入力，出力のランク付きアルファベット，Δ は以下の形式の遷移規則の有限集合である．$X_k \coloneqq \{ x_1, x_2, \ldots, x_k \}$ とする．

$$f(q_1(x_1), q_2(x_2), \ldots, q_k(x_k)) \to q(t)$$
$$(f \in \Sigma^{(k)}, q_1, q_2, \ldots, q_k, q \in Q, t \in T_\Gamma(X_k))$$

木 t の変数 $x_1, x_2, \ldots, x_k$ を $t'_1, t'_2, \ldots, t'_k$ で置き換えたものを $t[t'_1/x_1, t'_2/x_2, \ldots, t'_k/x_k]$ と書く．ボトムアップ木トランスデューサ $\mathcal{B}$ の動作関係 $\to_\mathcal{B}$ を以下で定義する．

$$\frac{t_i \to_\mathcal{B} q_i(t'_i)\ (i = 1, 2, \ldots, k) \qquad f(q_1(x_1), q_2(x_2), \ldots, q_k(x_k)) \to q(t) \in \Delta}{f(t_1, t_2, \ldots, t_k) \to_\mathcal{B} q(t[t'_1/x_1, t'_2/x_2, \ldots, t'_k/x_k])}$$

string-to-tree トランスデューサと同様に，ボトムアップ木トランスデューサも決定的かつ全域的なもののみを考える．ボトムアップ木トランスデューサ $\mathcal{B}$ の入力 t に対する出力 $\mathcal{B}(t)$ を，$t \to_\mathcal{B} q(t')$ となる唯一の t' として定義する．

$\Omega' \coloneqq \Omega \cup \{ \mathrm{Lft}^{(1)} \}$ と定義する．全探索マッチングの過程を表す木をバックトラックマッチングの過程を表す木に変換するボトムアップ木トランスデューサ $\mathcal{B}^{\mathrm{prune}} = (Q, \Omega, \Omega', \Delta)$ を以下で定義する．

- $Q = \{ q_\top, q_\bot \}$
- Δ は以下の遷移規則からなる．$(z \in \{ \top, \bot \})$

$$\top \to q_\top(\top) \quad \mathrm{Or}(q_\top(x_1), q_z(x_2)) \to q_\top(\mathrm{Lft}(x_1))$$
$$\bot \to q_\bot(\bot) \quad \mathrm{Or}(q_\bot(x_1), q_z(x_2)) \to q_z(\mathrm{Or}(x_1, x_2))$$

$\mathcal{B}^{\mathrm{prune}}$ は，現在走査している場所の下部に成功する枝が存在するかどうかを状態 $q_\top, q_\bot$ で記憶しておく．Or に出会ったときに左の子が $q_\top$ ならば，右の子はバックトラックでは探索されないので Or を Lft に置き換えることでそれを削除する．例えば，$[\![a^* a^*]\!]a$ は図 9 のように表され，その葉をマッチングの成否に変換してからバックトラックマッチングの場合の木に変換した $\mathcal{B}^{\mathrm{prune}}([\![a^* a^*]\!]a \gg\!= \chi)$ は図 10 のように表される． なお，拡張正規表現を扱う際には Ω に $\mathrm{Assert}^{(2)}, \mathrm{AssertNot}^{(2)}, \bot^{(1)}$ が追加されるが，これらに対する規則は以下のように定義される．

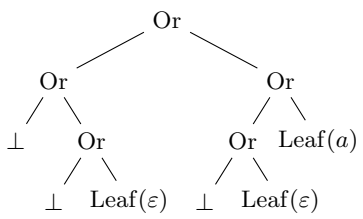


図 9 $[\![a^*a^*]\!]a$

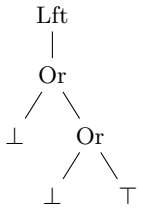


図 10
$\mathcal{B}^{\mathrm{prune}}([\![a^*a^*]\!]a \gg\!= \chi)$

$(z \in \{\top, \bot\})$

$$\mathrm{Assert}(q_\top(x_1), q_z(x_2)) \to q_z(\mathrm{Assert}(x_1, x_2))$$
$$\mathrm{AssertNot}(q_\top(x_1), q_z(x_2)) \to q_\bot(\mathrm{Lft}(x_1))$$
$$\mathrm{Assert}(q_\bot(x_1), q_z(x_2)) \to q_\bot(\mathrm{Lft}(x_1))$$
$$\mathrm{AssertNot}(q_\bot(x_1), q_z(x_2)) \to q_z(\mathrm{AssertNot}(x_1, x_2))$$
$$\bot(q_z(x_1)) \to q_\bot(x_1)$$

全探索マッチングの過程を表す木を出力するトランスデューサ $\mathcal{T}_r$ と，上で定義した $\mathcal{B}^{\mathrm{prune}}$ の合成は先読み付きトランスデューサで表現できる [4]．先読み付きトランスデューサは，遷移する時に入力文字列の残りの部分を見て遷移先を選択できるトランスデューサである．なお，先読み付きトランスデューサの出力サイズ増加率判定も，NFA の曖昧性増加率判定に帰着させることができる [5] [17]．したがって，本研究の高速化手法はバックトラックを模倣する場合でも適用可能である．

定義 A.2. 先読み付き *string-to-tree* トランスデューサは 6 つ組 $\mathcal{T} = (Q, \Sigma, \Gamma, q_0, \Delta, \mathcal{A})$ である．ただし，Q, Σ, Γ, q_0 は先読みのない通常の string-to-tree トランスデューサと同様のものであり，$\mathcal{A} = (P, \Sigma, p_0, F, \delta)$ は DFA である．Δ は以下の形式の遷移規則の有限集合である．

$$q \xrightarrow{a|p} t \quad (q \in Q, a \in \Sigma, p \in P, t \in T_\Gamma(Q_x))$$
$$q \xrightarrow{\$} t \quad (q \in Q, t \in T_\Gamma)$$

先読み付きトランスデューサ $\mathcal{T}$ の動作関係 $\to_{\mathcal{T}}$ を以下で定義する．ただし，$\hat{\delta}$ は DFA の遷移関数を自然に文字列に拡張したものとし，w^{R} は文字列 w の逆順を表す．

$$\frac{q \xrightarrow{a|p} t \in \Delta \qquad \hat{\delta}(p_0, w^{\mathrm{R}}) = p}{q(aw\$) \to_{\mathcal{T}} t[w\$/x]} \qquad \frac{q \xrightarrow{\$} t \in \Delta}{q(\$) \to_{\mathcal{T}} t}$$

$$\frac{t_i \to_{\mathcal{T}} t'_i \ (i = 1, 2, \ldots, k)}{f(t_1, t_2, \ldots, t_k) \to_{\mathcal{T}} f(t'_1, t'_2, \ldots, t'_k)}$$

先読み付きトランスデューサは DFA $\mathcal{A}$ を内部に持っており，入力文字列の残りを逆順にしたもので $\mathcal{A}$ を動作させてたどり着いた状態によって次の遷移を選択する．

$\mathcal{T}_{r_0} = (Q, \Sigma, \Omega, r_0, \Delta)$ が与えられているとする．この時，合成 $\mathcal{B}^{\mathrm{prune}} \circ \mathcal{T}_{r_0}$ を実現する先読み付きトランスデューサ $\mathcal{T}^{\mathrm{prune}}_{r_0}$ の構成法を述べる．DFA $\mathcal{A} = (\mathcal{P}, \Sigma, P_0, \mathcal{F}, \delta)$ を以下で構成する．

- $\mathcal{P} = 2^Q$
- $P_0 = \{\, r \mid r \xrightarrow{\$} t \in \Delta, t[\emptyset] = \mathit{True} \,\}$
- $\mathcal{F} = \{\, P \mid r_0 \in P \,\}$
- $\delta(P, a) = \{\, r \mid r \xrightarrow{a} t \in \Delta, t[P] = \mathit{True} \,\}$

この DFA $\mathcal{A}$ を用いて，先読み付きトランスデューサ $\mathcal{T}^{\mathrm{prune}}_{r_0} = (Q, \Sigma, \Omega', r_0, \Delta', \mathcal{A})$ を構成する．$P \subseteq Q$ について，$\mathcal{B}^{\mathrm{prune}}$ に以下の規則を追加することで入力/出力アルファベットをそれぞれ $\Omega \cup Q, \Omega' \cup Q$ へ拡張したものを $\mathcal{B}^{\mathrm{prune}}_P$ と書く．

- 任意の $r \in P$ について，$r \to q_\top(r) \in \Delta$.
- 任意の $r \notin P$ について，$r \to q_\bot(r) \in \Delta$.

これを用いて，遷移規則 Δ' を以下で定義する．

- $r \xrightarrow{a} t \in \Delta$ $(a \in \Sigma)$ の時，各 $P \in \mathcal{P}$ に対して $r \xrightarrow{a|P} \mathcal{B}^{\mathrm{prune}}_P(t) \in \Delta'$.
- $r \xrightarrow{\$} t \in \Delta$ の時，$r \xrightarrow{\$} \mathcal{B}^{\mathrm{prune}}(t) \in \Delta'$.

定理 A.3. 任意の $w \in \Sigma^*$ に対して，$\mathcal{T}^{\mathrm{prune}}_{r_0}(w) = (\mathcal{B}^{\mathrm{prune}} \circ \mathcal{T}_{r_0})(w)$.

B 証明の概略

B.1 拡張正規表現の近似の一般化

拡張正規表現 r を近似する $\mathit{approx}(r)$ を型体型の考え方で一般化して，一般化した $\mathit{approx}_\Gamma(r)$ が，計算量について過大近似を与えることを B.2 で証明する．

まず，型環境に対応する変数環境 Γ を Var から Σ 上の言語への関数とする．このとき，変数環境 Γ, 拡張正規表現 r, 言語 L に関する関係 $\Gamma \vdash e : L$ を以下

のように帰納的に定義する．

$$\Gamma \vdash \emptyset : \emptyset \quad \Gamma \vdash \varepsilon : \{\varepsilon\} \quad \Gamma \vdash a : \{a\}$$

$$\frac{\Gamma \vdash r_1 : L_1 \quad \Gamma \vdash r_2 : L_2}{\Gamma \vdash r_1 | r_2 : L_1 \cup L_2} \qquad \frac{\Gamma \vdash r_1 : L_1 \quad \Gamma \vdash r_2 : L_2}{\Gamma \vdash r_1 r_2 : L_1 L_2}$$

$$\frac{\Gamma \vdash r : L}{\Gamma \vdash r^* : L^*} \qquad \Gamma \vdash \overleftarrow{\&} r : \{\varepsilon\}$$

$$\frac{\Gamma \vdash r : L \quad L \subseteq \Gamma(x)}{\Gamma \vdash (r)_x : L} \qquad \Gamma \vdash \backslash x : \Gamma(x)$$

また，ある L に対して，$\Gamma \vdash r : L$ が成り立つとき，$\Gamma \vdash r$ と書くことにする．

任意の変数 x に対して $\Gamma(x)$ が正規言語ならば，$approx_\Gamma(r)$ を以下のように定義できる．

$$approx_\Gamma(\overleftarrow{\&} r) = \diamond$$
$$approx_\Gamma(\overleftarrow{!} r) = \diamond$$
$$approx_\Gamma((r)_x) = approx_\Gamma(r)$$
$$approx_\Gamma(\backslash x) = \Gamma(x)\diamond$$

ただし，最後の規則における $\Gamma(x)$ は，$\Gamma(x)$ を正規表現としたものとする．他の構文については，自然に帰納的に定義される．

また，正規表現 r において，後方参照 $\backslash x$ に対応するキャプチャグループが $(r_x)_x$ のときに $\Gamma(x) = L(r_x)$ と定義すれば，$\Gamma \vdash r$ であり，また，5.3.2 節の $approx$ と $approx_\Gamma$ が一致する．

次に，状態 s が変数環境 Γ を満たすことを表す $s \models \Gamma$ を定義する．

$$s \models \Gamma \Leftrightarrow \forall x \in \mathrm{dom}(s) \setminus \{ behind \}.s(x) \in \Gamma(x)$$

また，$\rho_\Gamma = \{s \mid s \models \Gamma\}$ と定義する．このとき，上記の体系は以下の意味で健全である．

補題 B.1. $\Gamma \vdash r : L$ かつ $s \models \Gamma$ とする．このとき，以下が成り立つ．

$$[\![r]\!]_{ST} w\, s \in tree\,(L^{-1} w \times \rho_\Gamma)$$

この補題は，L が r の言語の過大近似になっていることを示している．証明は，定理 4.2 の証明と同様である．

B.2 定理 5.4

$approx_\Gamma(r)$ が，計算量について過大近似を与えることの証明は，木の関係を導入して以下を示すことによって与える．

1. 置換の前後で関係がつくこと．(補題 B.4)
2. 関係がつく場合にそれが木のサイズの大小と対応すること．(補題 B.5)

まず，関係の定義に必要となる文脈 $failContext\,\alpha$ を定義する．

$$\begin{aligned} failTree\,\alpha &::= \bot \mid \bot(tree\,\alpha) \\ &\mid \mathrm{Or}\,(failTree\,\alpha) \times (failTree\,\alpha) \\ failContext\,\alpha &::= \bot(\bullet) \\ &\mid \mathrm{Or}\,(failContext\,\alpha) \times (failTree\,\alpha) \\ &\mid \mathrm{Or}\,(failTree\,\alpha) \times (failContext\,\alpha) \end{aligned}$$

$C \in failContext\,\alpha$ の穴 $\bullet$ に木 t を入れた木を $C[t]$ で表す．

定義 B.2. $tree\,\alpha$ に対する関係 $\preceq$ を，以下を満たす最小の関係として定義する．

$$\top \preceq \top$$
$$\mathrm{Leaf}(a) \preceq \mathrm{Leaf}(a)$$
$$t \in failTree\,\alpha \Longrightarrow \bot \preceq t$$
$$t \preceq t' \Longrightarrow \bot(t) \preceq \bot(t')$$
$$t_1 \preceq t_1', t_2 \preceq t_2' \Longrightarrow \mathrm{Or}(t_1, t_2) \preceq \mathrm{Or}(t_1', t_2')$$
$$t \preceq t', C \in failContext\,\alpha \Longrightarrow t \preceq C[t']$$

この関係に対して，以下の性質が成り立つ．

補題 B.3. $t \preceq t'$, $f : \alpha \to tree\,\alpha'$ とする．このとき，$t \gg\!= f \preceq t' \gg\!= f$ が成り立つ．

以下が定理 5.4 を示すための主要な補題である．補題 B.4 は r の構造と w の長さに関する帰納法で，補題 B.5 は $t \preceq t'$ の導出に関する帰納法で容易に示せる．補題 B.4 の証明には，補題 B.1 を用いる．

補題 B.4. r を後読みと後方参照を加えた拡張正規表現とし，$\Gamma \vdash r$ かつ $s \models \Gamma$ とする．このとき，任意の $w \in \Sigma^*$ について以下が成り立つ．

$$\pi_1([\![r]\!]_{ST} w\, s) \preceq [\![approx(r)]\!] w$$

補題 B.5. $t_1, t_2 \in tree\,\emptyset$ とする．$t_1 \preceq t_2$ ならば，$|\mathcal{B}^{\mathrm{prune}}(t_1)| \leq |\mathcal{B}^{\mathrm{prune}}(t_2)|$.

定理 5.4 は，これらの補題を用いると以下のように示すことができる．補題 B.4 から，

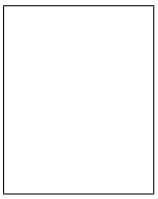
$$\pi_1([\![r]\!]_{ST} w\, s_0) \preceq [\![approx(r)]\!] w$$

補題 B.3 より，

$$\pi_1([\![r]\!]_{ST} w\, s_0) \gg\!= \chi \preceq [\![approx(r)]\!] w \gg\!= \chi$$

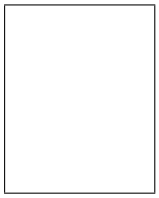
補題 B.5 より，

$$|\mathcal{B}^{\mathrm{prune}}(\pi_1([\![r]\!]_{ST} w\, s_0) \gg\!= \chi)| \leq$$
$$|\mathcal{B}^{\mathrm{prune}}([\![approx(r)]\!] w \gg\!= \chi)|$$



高 橋 和 也

2018 年東京工業大学情報科学科卒業．2020 年同大学大学院数理・計算科学系修士課程修了．現在，ヤフー株式会社にエンジニアとして勤める．プログラミング言語，形式言語理論に興味を持つ．



南 出 靖 彦

1993 年京都大学大学院理学研究科数理解析専攻修士課程修了．同年同大学数理解析研究所助手．1999 年筑波大学電子・情報工学系講師．2007 年同大学大学院システム情報工学研究科准教授．2015 年東京工業大学大学院システム情報工学研究科教授．現在，同大学情報理工学院教授．博士（理学）．ソフトウェア検証，プログラミング言語，形式言語理論に興味を持つ．