

拡張ポジションオートマトンを生成するモナド結合子

Monadic combinators constructing position automata with augmented transitions *

奥居 哲

中部大学工学部情報工学科

Satoshi Okui

Department of Computer Science, Chubu University

概要

拡張されたポジションオートマトン (Glushkov オートマトン) の結合子を用いた構築と、それを利用した正規表現のパターン照合について述べる。Packrat パーサに代表される従来のパーサ結合子が、ローカルな (浅い) バックトラックとメモ化 (memoization) を用いているのに対し、提案する結合子はバックトラックを一切用いない。そのため、ローカルなバックトラックに基づく手法に固有の「言語の隠蔽」(language hiding) の問題等が回避される。関数型言語 Haskell を用いた結合子のモナドに基づく実装を与え、この結合子による記述例を検討することで、正規表現による記述と比較して、より可読性の高い記述が可能であることを例証する。

1 はじめに

正規表現や文脈自由文法を用いたデータの解析技術は、コンパイラにおける構文解析処理にとどまらず、情報処理の様々な分野で用いられている基本的な要素技術である。現代のプログラミング言語の多くが、言語レベルで、あるいは標準ライブラリとして、正規表現をサポートしている。また、様々な言語向けの構文解析器生成系が存在する。

その一方で、これらのツールがプログラマー一般に十分に活用されてはいないという指摘もある。使用するプログラミング言語 (ホスト言語) とは別個に、これらのツールに固有の規約 (正規表現の書き方等) に従わなければならないことや、処理の結果をホスト言語から利用する際のインターフェイスに制約が大きいこと等が、その理由である。

このような問題を解決する方法として、関数型言語やオブジェクト指向言語で広く用いられているのがパーサ結合子 (parser combinator) [3] である。これは、単純な処理を行うパーサ (構文解析器) を結合子によって組み合わせることで、複雑な処理を行うパーサを組み立てる方法である。基本となるパーサも、結合子も、純粋にホスト言語の関数ないしオブジェクトであるので、ホスト

* 鈴木大郎氏 (会津大学) との共同研究に基づく。本研究は科研費基盤研究 (C)No.22500019 の助成を受けて行われている。

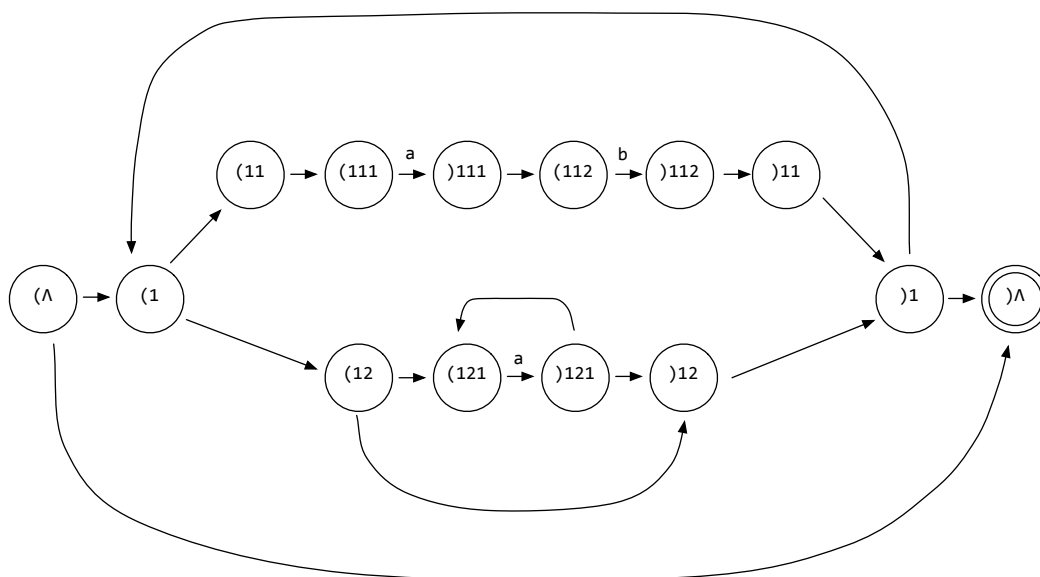


図1 正規表現 $(ab|a^*)^*$ から生成される Thompson NFA

言語に習熟したプログラマにとって扱いが容易になる。さらに、**モナド**に基づくパーサ結合子では、必要に応じて新たな結合子を定義することも容易であり、柔軟性、拡張性が高い。

本論文では、筆者らが既に提案した**拡張ポジションオートマトン** (position automata with augmented transitions; PAT) [7] を生成する結合子について述べる。PAT に基づく正規表現のパターン照合は、部分集合構成と優先度によるパスの枝刈りに基づいており、バックトラックを一切行わない。よって、本論文で述べる結合子も、Packrat パーサ [1] に代表されるローカルな（浅い）バックトラックとメモ化に基づく従来のパーサ結合子とは大きく異なる。その結果、従来の Packrat パーサのような**言語の隠蔽** (language hiding) の問題 [8] は生じない。

以下では、第2節で、Thompson NFA と比較しつつ PAT を直観的に導入した後、第3節で、正規表現から PAT を構築するアルゴリズムを与える。第4節で、PAT を構築するモナドに基づく結合子を導入し、結合子を用いた記述例を検討する。最後に第5節で、まとめを述べる。

2 拡張ポジションオートマトン

正規表現から有限オートマトンを構築する際に広く用いられているのが **Thompson の構成法** [9] である。これは、正規表現の各部分式を状態と見なして ϵ -NFA を構築する方法である。例えば正規表現 $(ab|a^*)^*$ に括弧を補って部分式を明示すると $((a|b))^*$ となる、この7対14個の括弧を状態とみなすことで、図2のような ϵ -NFA が構築できる。括弧には、正規表現の抽象構文木のポジションで添字を付けた。ラベルの無い遷移は、すべて ϵ -遷移である。

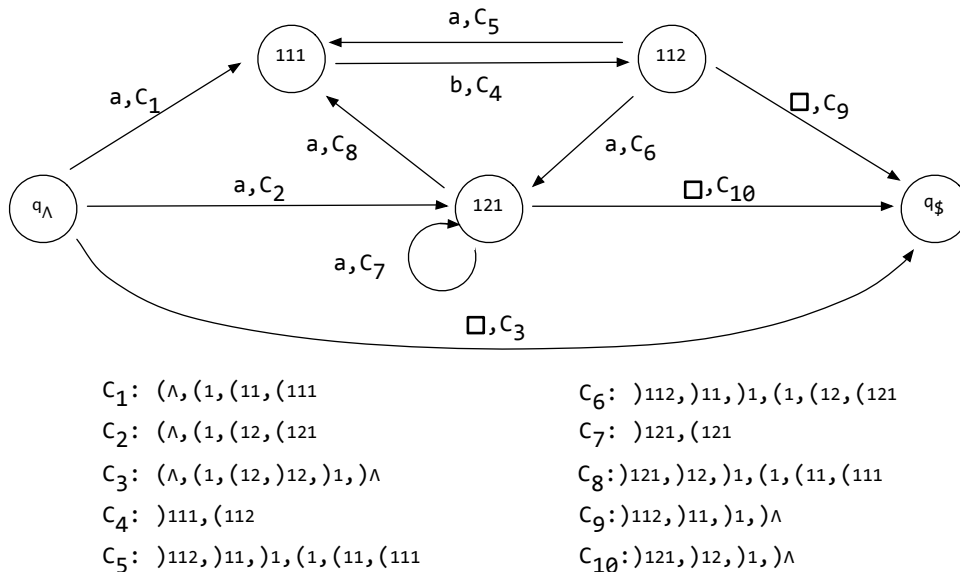


図2 正規表現 $(ab|a)^*$ から生成される PAT

一方、正規表現における文字の出現位置を状態とみなし NFA を構築するのが、**ポジションオートマトン**[6]（あるいは **Glushkov オートマトン**[4]）の考え方である。例 $(ab|a)^*$ の場合だと、文字の出現は3カ所あるので、図2のような NFA が構築できる。^{*1} ここで□は、任意の文字による遷移を表す。 q_Λ は初期状態、 $q_\$$ は終状態である。

ポジション NFA は Thompson NFA における ε -遷移を次の非 ε -遷移が現れるまで一度に辿ったものであると考えることが可能である。よって、ポジションオートマトンのほうには、 ε -遷移は無い。遷移毎に、必ず文字を一つ読み進める。古典的なポジションオートマトンとは異なり、[7]では終状態が唯一になるように遷移を補っている。このため、先読みを想定していて、入力文字列の最後には、文字列終端を表す文字（\$と書く）が置かれていると仮定する。

Thompson NFA では、 ε -遷移の閉路が存在する場合がある（図2）が、これは、パターン照合の戦略（最左最長戦略や貪欲（greedy）戦略）を考える際に問題を生じる場合がある。例えば貪欲戦略は、反復 R^* を再帰的に $R^* = RR^*|\varepsilon$ と見なした上で左の選択肢を右の選択肢より優先する深さ優先探索であると、おおよそ考えてよい。ただ、これでは探索が無限ループに陥る場合（例： a^* と文字列“b”の照合）があるので、厳密には、単純な深さ優先探索と考える訳にはいかない。^{*2} ([2]では、これを problematic case と呼び、この問題への対処に多くの議論を割いている)。ポジションオートマトンでは、そもそも ε -遷移の閉路は存在しないので、無限ループの問題は発生しない。

一方、Thompson NFA の利点は、もとの正規表現の構造（部分式のネスト構造）に関する情報

^{*1} 各状態間の遷移が高々一つになるように最適化した後のものを示している。

^{*2} 本質的には再帰下降型構文解析における**左再帰**の問題の特殊な場合である。

を完全に保持している点にある。これに対して、ポジションオートマトンでは、もとの正規表現の構造の情報は失われており、このことは、正規表現の最左最長マッチングや貪欲マッチングの実現や、部分式とマッチする文字列の捕獲を困難にする。

そこで、ポジションオートマトンの遷移に元の正規表現の部分式に関する情報を補おうというのが、**拡張ポジションオートマトン** (position automata with augmented transitions; PAT) である。図 2 の下部に、各遷移に付加される情報を示す。

遷移に拡張情報を付加するというアイデアは様々な文献にみられるが、PAT は特に Laurikari の **タグ付きオートマトン** (TNFA) [5] から影響を受けている。ただし、TNFA が ε -遷移に単一の括弧をタグとして付加するのに対し、PAT では一般に複数の括弧の並びを非 ε -遷移に付加する点異なる。また、PAT では、括弧の並び以外の任意の情報を付加してよい。例えば、[7] では、最左最長マッチングを効率的に実現するために、各遷移に整数を付加している。PAT はまた、**逐次変換器** (sequential transducer) の一種と考えることもできる。

3 拡張ポジションオートマトンの構築

PAT を形式的に定義し、正規表現から PAT を構成する方法を、[7] にしたがって与える。

3.1 PAT の形式的な定義

拡張ポジションオートマトン (PAT) は 6 組

$$\langle \Sigma, Q, T, \Delta, q_{\wedge}, q_{\$} \rangle$$

で定義される。ここで、 Σ は、文字の集合、 Q は、**状態**の有限集合、 T は、拡張情報を表す**タグ**の有限集合、 Δ は、 $Q \times \Sigma \times Q \times T^*$ の部分集合、 $q_{\wedge}$ は、 Q における**初期状態**、 $q_{\$}$ は、 Q における**終了状態**である。 Δ の要素を**遷移**と呼ぶ。

$M = \langle \Sigma, Q, T, \Delta, q_{\wedge}, q_{\$} \rangle$ を PAT とし、 w を**入力文字列**とすると、3 組 $\langle u, p, \alpha \rangle$ を**配置** (あるいは**様態**) と呼ぶ。ここで、 u は w のサフィックス、 p は Q における状態、 α はタグの列を表す。 Δ における遷移 $\delta = \langle p_1, a, p_2, \tau \rangle$ にたいして、 $u_1 = au_2$ かつ $\alpha_2 = \alpha_1 \tau a$ であるとき、

$$\langle u_1, p_1, \alpha_1 \rangle \vdash_{\delta} \langle u_2, p_2, \alpha_2 \rangle$$

と表す。 $\vdash$ の添字は、必要なければ省略する。

入力文字列 w にたいし、 $w = uv$ となる u, v を考える。このとき、 n ステップからなる列

$$\langle uv, q_{\wedge}, \varepsilon \rangle \vdash \cdots \vdash \langle v, q, \alpha \rangle$$

が存在するならば、PAT M は状態 q に n ステップで**到達**し、(w のプレフィックス u にたいして) α を**産出**するという。ただし、 $uv = w$ とする。さらに、 $v = \varepsilon$ かつ $q = q_{\$}$ であるとき、PAT M は u を**受理**するという。

3.2 正規表現から PAT への変換

正規表現 r とポジション p から構築される PAT $M(r, p)$ を, r の構造に関する帰納法により, 以下のように定義する.

1. $M(1, p)$ は, $\langle \Sigma, \{q_\wedge, q_\S\}, \{(p,)_p\}, \{\langle q_\wedge, a, q_\S, (p)_p \rangle \mid a \in \Sigma, q_\wedge, q_\S \rangle$ である.
2. $a \in \Sigma$ にたいし, $M(a, p)$ は, $\langle \Sigma, \{q_\wedge, p, q_\S\}, \{(p,)_p\}, \Delta, q_\wedge, q_\S \rangle$ である. ここで, Δ は, 遷移 $\langle q_\wedge, a, p, (p) \rangle$ と任意の $b \in \Sigma$ に対する遷移 $\langle p, b, q_\S, (p) \rangle$ からなる.
3. $i = 1, 2$ にたいし $M(r_i, p.i)$ が $\langle \Sigma, Q_i, T_i, \Delta_i, q_\wedge, q_\S \rangle$ であるならば, $M(r_1 + r_2, p)$ は,

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{(p,)_p\}, [\Delta_1 \cup \Delta_2]_p, q_\wedge, q_\S \rangle$$

である.

4. $i = 1, 2$ にたいし $M(r_i, p.i)$ が $\langle \Sigma, Q_i, T_i, \Delta_i, q_\wedge, q_\S \rangle$ であるならば, $M(r_1 \cdot r_2, p)$ は,

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{(p,)_p\}, [\Delta_1 \cdot \Delta_2]_p, q_\wedge, q_\S \rangle.$$

である.

5. $M(r, p.1)$ が $\langle \Sigma, Q, T, \Delta, q_\wedge, q_\S \rangle$ であるならば, $M(r^*, p)$ は,

$$\langle \Sigma, Q, T \cup \{(p,)_p\}, [\Delta^*]_p, q_\wedge, q_\S \rangle.$$

である.

ただし, $[\Delta]_p$, $\Delta_1 \cdot \Delta_2$ および Δ^* は, それぞれ, 以下のように定義される.

$$\begin{aligned} [\Delta]_p = & \{ \langle q_1, a, q_2, \tau \rangle \in \Delta \mid q_1 \neq q_\wedge, q_2 \neq q_\S \} \cup \{ \langle q_\wedge, a, q_\S, (p\tau)_p \rangle \mid \langle q_\wedge, a, q_\S, \tau \rangle \in \Delta \} \\ & \cup \{ \langle q_\wedge, a, q, (p\tau)_p \rangle \mid \langle q_\wedge, a, q, \tau \rangle \in \Delta, q \neq q_\S \} \\ & \cup \{ \langle q, a, q_\S, \tau \rangle_p \mid \langle q, a, q_\S, \tau \rangle \in \Delta, q \neq q_\wedge \} \end{aligned}$$

$$\begin{aligned} \Delta_1 \cdot \Delta_2 = & \{ \langle q_1, a, q_2, \tau \rangle \in \Delta_1 \mid q_2 \neq q_\S \} \cup \{ \langle q_1, a, q_2, \tau \rangle \in \Delta_2 \mid q_1 \neq q_\wedge \} \\ & \cup \{ \langle q_1, a, q_2, \tau_1 \tau_2 \rangle \mid \langle q_1, -, q_\S, \tau_1 \rangle \in \Delta_1, \langle q_\wedge, a, q_2, \tau_2 \rangle \in \Delta_2 \} \end{aligned}$$

$$\begin{aligned} \Delta^* = & \Delta \cup \{ \langle q_\wedge, a, q_\S, \varepsilon \rangle \mid a \in \Sigma \} \\ & \cup \{ \langle q_1, a, q_2, \tau_1 \tau_2 \rangle \mid \langle q_1, -, q_\S, \tau_1 \rangle \in \Delta, \langle q_\wedge, a, q_2, \tau_2 \rangle \in \Delta, q_1 \neq q_\wedge, q_2 \neq q_\S \} \end{aligned}$$

最後に, $M(r, \Lambda) = \langle \Sigma, Q, T, \Delta, q_\wedge, q_\S \rangle$ にたいし, PAT $M(r)$ を $\langle \Sigma, Q, T, \Delta \cup \Delta_0, q_\wedge, q_\S \rangle$ と定義する. ここで, Δ_0 は任意の $a \in \Sigma$ にたいする遷移 $\langle q_\wedge, a, q_\wedge, \varepsilon \rangle$ と, 任意の $a \in \Sigma$ にたいする遷移 $\langle q_\S, a, q_\S, \varepsilon \rangle$ からなる.

正規表現 r による文字列 w の**正形** (canonical) な導出木 (の列による表現) の集合は, $M(r)$ が入力文字列 w を受理した際に産出する列の集合と等しい [7]. すなわち, 上記の方法で正規表現 r から構築される PAT $M(r)$ は, r による「正しいと認められる」パターン照合の可能な解を正確に列挙する.

4 モナド結合子による表現

前節の PAT の構成法に従い、PAT を構築する構成子を Haskell でモナドを用いて与える。

4.1 PAT の計算のモナドによる表現

前節の定義に従えば、PAT の計算は遷移の列である。各遷移では、その遷移に付与されている拡張情報を値として産出すると同時に、現在の配置が次の配置に更新される。さらに遷移は非決定的である。つまり、現在の配置に対して、値と次の配置の組は、複数存在する（あるいは、ひとつも存在しない場合もある）。よって、PAT における遷移は、Haskell のデータ型として以下のように表現できる。

```
type Pos = [Int]
data MState = I | F | St Pos deriving (Show, Eq, Ord)
data Trans a = Trans { runTrans :: MState -> [(a, MState)] }
```

PAT の状態 MState は、文字の出現位置（正規表現の抽象構文木は高々二つの部分木しか持たないので、これを整数 1 または 2 の並びで表す）Pos に、初期状態 I と終了状態 F を加えたものである。定義の最後の行における a は任意の拡張情報を表す型変数であり、右辺の Trans と runTrans は、それぞれ、型のコンストラクタとデコンストラクタである。

この後 PAT における遷移をより単純な遷移の組み合わせとして構成していくが、この作業を簡単に行えるようにするために、Trans をモナドと見なす。型 Trans を **Monad** クラスのインスタンスに加える。

```
instance Monad Trans where
    return a = Trans tr where tr s = [(a, s)]
    m >>= f = Trans tr where
        tr s = do { (a, s) <- runTrans m s; runTrans (f a) s; }
```

ここで、**return a** は、PAT の状態を全く変更せずに拡張情報 a を生産する最も基本的な遷移を表す。>>= は、遷移を連結する演算である。**return** と >>= は、以下のモナド則を満足する。

```
return x >>= f           == f x
m >>= return             == m
m >>= (\x -> f x >>= g) == (m >>= f) >>= g
```

これは **return** を単位元とした、2 項演算 >>= の一種の結合則である。

演算子 >> を $m1 \gg m2 = m1 \gg= (\lambda - \> m2)$ と定義する。また、

```
do a1 <- m1
  a2 <- m2
  ...
  mn
```

は、>>= の連鎖

```
m1 >>= (\a1 -> m2 >>= (\a2 -> ... mn) ... ))
```

に対する構文糖衣である。

PAT における遷移をモナドとみなすことで、単純な遷移を組み合わせて複雑な遷移を構成する作業が極めて容易になる。例えば、二つの遷移 t_1 、 t_2 をこの順に連結し、それぞれが産出する拡張情報 a_1 、 a_2 に 2 引数関数 f を適用した結果を拡張情報として産出するような遷移は、

```
do a1 <- t1
   a2 <- t2
   return (f a1 a2)
```

あるいは、

```
lift2M f t1 t2
```

のように表現できる。

遷移 $\text{Trans } a$ は、**MonadPlus** クラスのインスタンスでもある。

```
instance MonadPlus Trans where
    mzero      = Trans tr where tr _ = []
    m1 'mplus' m2 = Trans tr where tr s = (runTrans m1 s) ++ (runTrans m2 s)
```

mzero は、遷移の結果が存在しないことを表している。二項演算 **mplus** は、遷移の結果が、もとの二つの非決定的な遷移の結果の併合になるような新たな遷移を表す。演算 **mplus** は **mzero** を零元として結合的である。

```
mzero 'mplus' m == m 'mplus' mzero == m
m1 'mplus' (m2 'mplus' m2) == (m1 'mplus' m2) 'mplus' m3
```

さらに、遷移 $\text{Trans } a$ は **MonadState** クラスのインスタンスであり、それぞれ状態の取得と更新をおこなう演算 **get** と **put** が定義される。

```
instance MonadState MState Trans where
    get  = Trans tr where tr s = [(s,s)]
    put s = Trans tr where tr _ = [((),s)]
```

以上の定義の下、PAT を、文字を与えると遷移を返す関数として定義する。

```
type Pat a = Char -> Trans a
```

まとめると、PAT の計算は状態遷移を表すモナドと非決定計算を表すモナドの組み合わせである。Haskell の **mtl** ライブラリには、任意のモナドから状態遷移のモナドを作り出すモナド変換子 (monad transformer) **StateT** が用意されているので、拡張情報の型を a とする PAT の遷移を表すモナドは実際には、単に

```
type Trans a = StateT MState [] a
```

と表される。(付録 A の全コードのリストでは、このモナド変換子を用いている。)

4.2 遷移の組み立て

2.2 節の定義によれば、二つの PAT を連結してできる PAT における遷移では、最初の遷移 (t_1 と呼ぶ) において終状態に到達する場合には、さらに次の遷移 (t_2 と呼ぶ) の始状態から到達可能な状態にまで引き続き遷移させる必要がある。これは、そのまま以下のように表現できる。

```

joinP :: (a -> a -> a) -> (Trans a -> Trans a -> Trans a)
joinP op t1 t2 = do
  a <- t1
  q <- get
  if q == F then do { put I; b <- t2; return (a 'op' b); }
  else return a

```

ここで、op は、二つの拡張情報を一つに結合する任意の演算である。

この関数joinPを用いれば、二つの PAT を一つに連結して得られる PAT における遷移を作り出す結合子 catP は、以下のようになる。

```

catP :: (a -> a -> a) -> (Trans a -> Trans a -> Trans a)
catP op t1 t2 = (joinP op t1 t2) 'mplus' (t2 'notComeFrom' I)

```

ただし、t2 'notComeFrom' I は、遷移t2の定義域から初期状態Iを除いたものを表す（定義は付録を参照）。

PAT を 1 回以上の反復する場合も同様である。

```

rep1P :: (a -> a -> a) -> (Trans a -> Trans a)
rep1P op t = (joinP op (t 'notComeFrom' I) (t 'notGoTo' F)) 'mplus' t

```

ただし、t 'notGoTo' F は、遷移tの値域から終状態 F を除いたものである（定義は付録を参照）。

上記の反復の定義中で、始状態から直接終状態に至る遷移が除外されているのは、正形（canonical）でない照合を除外するためである。

二つの PAT の和（並列結合）を表す結合子は、単に mplus の別名である。

```

altP :: Trans a -> Trans a -> Trans a
altP t1 t2 = t1 'mplus' t2

```

4.3 タグの並びを拡張情報として持つ PAT の構築

以上で、正規表現の演算子に対応して遷移を組み立てることが可能になったので、これに基づいて、正規表現の部分式の構造を表す括弧の並び（以下、**タグ**の並び）を拡張情報として持つような PAT を生成する結合子を定義する。タグを

```

data Tag = O Pos | C Pos | A Char deriving (Show, Eq, Ord)

```

のように表すことにすると、タグの並びを拡張情報として持つ PAT の型 PatT は、正規表現の部分式の位置を定義域とする関数で表される。

```

type PatT = Pos -> Pat [Tag]

```

まずは、構成の出発点として、空文字列を受理する正規表現 ε と、1 文字を受理する**文字クラス**に対応する PAT（それぞれ、emp と letter と呼ぶ）を定義する。これらは、任意の状態 q1 から q2 への遷移 proceed q1 q2 を

```

proceed :: MState -> MState -> Trans ()
proceed q1 q2 = confirm (== q1) >> put q2

```

とすれば、以下のようになる。

```

emp :: PatT
emp p = proceed I F >> return [O p, C p]

letter :: [Char] -> PatT
letter cs p c = (guard (c `elem` cs) >> proceed I (St p) >> return [O p, A c])
               'mplus' (proceed (St p) F >> return [C p])

```

次に、より複雑な正規表現に対応する PAT を作る。この際、3.2 節の演算 $[]_p$ に対応して、各部分式の出現位置に応じてタグを補うための関数が、`enclose` である。

```

enclose :: Pos -> Trans [Tag] -> Trans [Tag]
enclose p t = do
  a <- gets addO
  b <- t
  c <- gets addC
  return (a ++ b ++ c) where
    addO q = if q == I then [O p] else []
    addC q = if q == F then [C p] else []

```

`enclose p t` は、遷移 `t` 中の初期状態から始まるものと終状態に至るものに、部分式の出現位置 `p` に対応するタグの情報を付加してできる遷移を表している。

4.2 で定義した結合子と `enclose` を組み合わせると、連結 `cat`、選択 (和) `alt`、1 回以上の反復 `rep1` を意味する結合子は、それぞれ以下のようになる。

```

cat :: PatT -> PatT -> PatT
(m1 'cat' m2) p c = enclose p $ catP (++) t1 t2 where
  t1 = m1 (p ++ [1]) c
  t2 = m2 (p ++ [2]) c

alt :: PatT -> PatT -> PatT
(m1 'alt' m2) p c = enclose p $ t1 'altP' t2 where
  t1 = m1 (p ++ [1]) c
  t2 = m2 (p ++ [2]) c

rep1 :: PatT -> PatT
rep1 m p c = enclose p $ rep1P (++) t where t = m (p ++ [1]) c

```

これらを組み合わせて、さらに有用な結合子を作り出すこともできる。例えば、0 回以上の反復 `rep0`、省略可能 `opt`、区切り記号をはさんだ並び `sepBy` は、以下のようになる。

```

rep0 :: PatT -> PatT
rep0 m = (rep1 m) 'alt' emp

opt :: PatT -> PatT
opt m = m 'alt' emp

sepBy :: PatT -> PatT -> PatT
m 'sepBy' sp = m 'cat' rep0 (sp 'cat' m)

```

4.4 結合子による表現例

前節で定義した結合子を用いることで、任意の正則言語を Haskell のプログラムとして表現できる。いくつかの例を示す。最初の例は、数字 num、英字 alpha、さらに英数字 alnum を定義し、それらを用いてプログラミング言語の識別子を表現したものである。

```
num    = letter ['0'..'9']
alpha  = letter (['a'..'z'] ++ ['A'..'Z'])
alnum  = alpha 'alt' num
ident  = alpha 'cat' (rep0 alnum)
```

ここで、`['0'..'9']` は、Haskell に用意されている '0' から '9' までの 10 個の文字からなるリストの表記法である。Haskell の正規の表記法であるにもかかわらず、正規表現における文字クラスの表記として自然に読むことができる。

次の例は、簡単な **コンマ区切り形式**(CSV) のデータを表したものである。

```
csv = (rep1 alnum) 'sepBy' (letter ",")
```

最後は、メールアドレスを表す例である。ただし、RFC 5321,5322 の仕様になるべく合わせているが、準拠している訳ではない（ローカルアドレスの部分にピリオドを許していない等）。

```
mailAddr = local 'cat' (letter "@") 'cat' (domain 'alt' ipaddr) where
  special1 = letter "!#$%&'*+,-/=^_`{|}~"
  special2 = letter "(<>[:;@,)"
  space    = rep1 (letter " ")
  quoted p = (letter "\"") 'cat' p 'cat' (letter "\"")
  dotStr    = (rep1 (alnum 'alt' special1))
  quotedStr = quoted (rep1 (foldr1 alt [alnum, special1, special2, space]))
  local     = dotStr 'alt' quotedStr
  domain    = ((rep1 alnum) 'sepBy' letter ".")
  ipaddr    = (letter "[") 'cat'
              ((rep1 num) 'sepBy' letter ".") 'cat' (letter "]")
```

結合子は Haskell の通常関数であるため、他の通常関数との柔軟な混在が可能である。この例では、両端に引用符を付加する補助関数 `quoted` や、標準ライブラリにある高階関数 `foldr1` を混在して用いている。

4.5 結合子を用いた照合処理

結合子による PAT の表現は、それ自体、Haskell の関数なので、照合したい文字列を与えるだけで実行可能である。正確には、結合子による PAT の表現にルートポジション `[]` を与え、さらに照合したい文字列の各文字を与えて得られる遷移をモナドとして連結し、初期状態 `I` を与えればよい。例えば PAT `ident` と文字列 `"foo123"` を照合するには、

```
runTrans (mapM (ident []) "foo123$") I
```

とする。そこで、

```
fullMatch :: PatT -> [Char] -> [([Tag], MState)]
fullMatch m cs = runTrans (mapM (m []) (cs ++ "$")) 1
```

と定義しておく。

GHC による実行例を示す（改行を補って整形した）。

```
*Main> fullMatch ident "foo123"
([([O []],O [1],A 'f'),
 [C [1],O [2],O [2,1],O [2,1,1],O [2,1,1,1],A 'o'],
 [C [2,1,1,1],C [2,1,1],O [2,1,1],O [2,1,1,1],A 'o'],
 [C [2,1,1,1],C [2,1,1],O [2,1,1],O [2,1,1,2],A '1'],
 [C [2,1,1,2],C [2,1,1],O [2,1,1],O [2,1,1,2],A '2'],
 [C [2,1,1,2],C [2,1,1],O [2,1,1],O [2,1,1,2],A '3'],
 [C [2,1,1,2],C [2,1,1],C [2,1],C [2],C []]],F)]
*Main>
```

実行結果から、終了状態 F に至るまでにどのような経路を経由したかが読み取れる。

遷移は非決定的な計算であるので、このままでは計算コストが指数的に増大する場合がある。同一の状態に到達する経路の間で優先度を用いた枝刈りを行うことで、経路の数を最大でも、おおよそ状態数（すなわち、再頻出文字の出現数）の自乗に抑えられる [7]。この点について本論文では議論しないが、結合子がモナドを用いて抽象的に定義されているため、枝刈りの導入による既存のコードの変更は、僅かで済む。

ローカルなバックトラックに基づく通常の Packrat Parser [1] では、その受理する言語が、しばしば宣言的に期待されるものと異なるという「言語の隠蔽」の問題 [8] が指摘されている。例えば、

```
(rep1 (letter "a")) 'cat' (letter "a")
```

と

```
(letter "a") 'cat' (rep1 (letter "a"))
```

は宣言的には同じ意味であり、2 つ以上任意個の a からなる文字列を受理することが期待されるが、例えば前者に文字列 "aaa" を与えると Packrat Parser は照合に失敗する。(rep1 (letter "a")) が a を 3 個すべて受理した後、(letter "a") の照合に失敗するが、バックトラックが行われないためである。メモ化を行う Packrat Parser ではグローバルなバックトラックの導入は困難である。これに対し、そもそもバックトラックやメモ化に依存しない本手法では、上の 2 つは全く同じ意味に扱われる。

5 おわりに

拡張ポジションオートマトンを生成する結合子を提案した。純粋な Haskell の関数として正則言語を記述できるため正規表現を用いる方法と比べて可読性が高く、一般の関数との混在が容易になる等、柔軟性が高いことを示した。また、結合子の構築はモナドを用いて抽象的に定義されているため、極めて簡潔で、拡張性が高いことを示した。

参考文献

- [1] B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.
- [2] A. Frisch and L. Cardelli. Greedy regular expression matching. In *ICALP04 (LNCS 3142)*, pp. 618–629, 2004.
- [3] R. Frost and L. John. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, Vol. 23, No. 2, pp. 108–121, 1989.
- [4] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, Vol. 16, No. 5, pp. 1–53, 1961.
- [5] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2000.
- [6] R. McNaughton and H Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, Vol. 9, pp. 39–47, 1960.
- [7] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In *CIAA 2010 (LNCS 6482)*, pp. 231–240, 2010.
- [8] R. Redziejowski. BITES instead of FIRST for parsing expression grammar. *Fundamenta Informaticae*, Vol. 109, No. 3, pp. 323–337, 2011.
- [9] K. Thompson. Regular expression search algorithm. *Communication of the ACM*, Vol. 11, No. 6, pp. 419–422, 1969.

付録 A 結合子の全コード

```
1 import Control.Monad
2 import Control.Monad.State
3
4 type Pos = [Int]
5
6 data MState = I | F | St Pos deriving (Show, Eq, Ord)
7
8 type Trans a = StateT MState [] a
9
10 type Pat a = Char -> Trans a
11
12 data Tag = O Pos | C Pos | A Char deriving (Show, Eq, Ord)
13
14 confirm :: (MState -> Bool) -> Trans ()
15 confirm cond = do { q <- get; guard (cond q); }
16
17 comeFrom, goTo, notComeFrom, notGoTo :: Trans a -> MState -> Trans a
18 t 'comeFrom'    q = confirm (== q) >> t
```

```

19 t 'goTo'          q = do { a <- t; confirm (== q); return a; }
20 t 'notComeFrom'  q = confirm (/= q) >> t
21 t 'notGoTo'      q = do { a <- t; confirm (/= q); return a; }
22
23 proceed :: MState -> MState -> Trans ()
24 proceed q1 q2 = confirm (== q1) >> put q2
25
26 joinP :: (a -> a -> a) -> (Trans a -> Trans a -> Trans a)
27 joinP op t1 t2 = do
28     a <- t1
29     q <- get
30     if q == F then do { put I; b <- t2; return (a 'op' b); }
31     else return a
32
33 catP :: (a -> a -> a) -> (Trans a -> Trans a -> Trans a)
34 catP op t1 t2 = (joinP op t1 t2) 'mplus' (t2 'notComeFrom' I)
35
36 rep1P :: (a -> a -> a) -> (Trans a -> Trans a)
37 rep1P op t = (joinP op (t 'notComeFrom' I) (t 'notGoTo' F)) 'mplus' t
38
39 altP :: Trans a -> Trans a -> Trans a
40 altP t1 t2 = t1 'mplus' t2
41
42 enclose :: Pos -> Trans [Tag] -> Trans [Tag]
43 enclose p t = do
44     a <- gets addO
45     b <- t
46     c <- gets addC
47     return (a ++ b ++ c) where
48         addO q = if q == I then [O p] else []
49         addC q = if q == F then [C p] else []
50
51 type PatT = Pos -> Pat [Tag]
52
53 cat :: PatT -> PatT -> PatT
54 (m1 'cat' m2) p c = enclose p $ catP (++) t1 t2 where
55     t1 = m1 (p ++ [1]) c
56     t2 = m2 (p ++ [2]) c
57
58 alt :: PatT -> PatT -> PatT
59 (m1 'alt' m2) p c = enclose p $ t1 'altP' t2 where
60     t1 = m1 (p ++ [1]) c
61     t2 = m2 (p ++ [2]) c
62
63 rep1 :: PatT -> PatT
64 rep1 m p c = enclose p $ rep1P (++) t where t = m (p ++ [1]) c
65
66 emp :: PatT
67 emp p _ = proceed I F >> return [O p, C p]
68

```

```

69 letter :: [Char] -> PatT
70 letter cs p c = (guard (c `elem` cs) >> proceed I (St p) >> return [O p, A c])
71                 `mplus` (proceed (St p) F >> return [C p])
72
73 rep0 :: PatT -> PatT
74 rep0 m = (rep1 m) `alt` emp
75
76 opt :: PatT -> PatT
77 opt m = m `alt` emp
78
79 sepBy :: PatT -> PatT -> PatT
80 m `sepBy` sp = m `cat` rep0 (sp `cat` m)
81
82 fullMatch :: PatT -> [Char] -> [[[Tag]], MState]
83 fullMatch m cs = runStateT (mapM (m []) (cs ++ "$")) I

```