プログラミング言語 20－5
（1989. 2. 10）

パターン照合プログラムの変換

赤間 陽二　　武市 正人

東京大学 工学部 計数工学科

本稿では関数プログラムの変換によって、パターン照合の問題に対する素朴なアルゴリズムから効率のよいものを導出する手法を提示する。ここで扱うプログラム変換は、高階関数の導入、および関数の部分評価といった関数プログラミングに特徴的な概念を用いたものであり、変換の各段階は数学的に証明することのできる事実に基づいている。また、部分評価で得られる情報は、変換の最終段階で、基本的なデータ構造を用いたメモ化(memo-ization)によって効率よく参照される。

Transformational Programming Applied to Pattern Matching Algorithms

Yoji Akama and Masato Takeichi
Department of Mathematical Engineering and Information Physics
Faculty of Engineering, University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan

We show a formal transformation technique for obtaining efficient pattern matching programs from naive algorithms. Our method is based on the best idea of functional programming such as higher order function and partial parametrization, and each transformation step is subject to a simple mathematical law. The final step of transformation consists of memo-ization techniques to keep the information obtained by partial parametrization.

1. はじめに

われわれは[7]において、パターン照合の問題の仕様記述 -- 素朴なアルゴリズムの記述 -- 関数プログラムの実現 -- プログラムの変換という過程を経て、Knuth-Morris-Pratt アルゴリズム [5]を導出した。本稿では、この方法における効率向上のための手法を再検討し、より一般的なパターン照合アルゴリズムへの適用性を調べる。この問題におけるプログラム変換の特徴は、高階関数の導入とその部分評価(部分適用)である。関数プログラムに特徴的なこれらの考え方に基づくプログラム変換法は[6]にも示されているが、ここではこれをパターン照合の問題に適用しようというものである。高階関数の部分評価は、すでに知られている効率のよいパターン照合アルゴリズムが、パターンのみに依存する情報によって前処理を行なっていることに対応する。われわれの方法は、従来のアルゴリズムの導出法と違って、アルゴリズムの記述に関数プログラムを用い、数学的に自然に証明できるいくつかの定理に基づき、直感に頼らずに効率のよいアルゴリズムを得ている。

本稿では、第2節で考え方の基本となっている変換法の概要を述べ、第3節で効率向上のためのメモ化に対して2種類の実現法を紹介する。第4節では、複数個のパターン照合の問題の定式化と効率のよいプログラムの導出に向けて検討を加える。

2. Knuth-Morris-Prattアルゴリズムの導出

Knuth-Morris-Pratt のアルゴリズム[5](以下 KMPアルゴリズムと呼ぶ)は、1個の文字列パターンPと文字列のテキストTが与えられたとき、PがTの部分列であるかどうかを判定する高速アルゴリズムである。KMPアルゴリズムにはテキストとは独立にパターンのみに依存する前処理の段階があり、そこで得られた情報を用いてパターンの文字とテキストの文字が一致しなかった場合にもテキストを逆戻りして調べることはない。このような性質を持つKMPアルゴリズムを素朴なアルゴリズムから導出する方法は、Knuthらの原論文[5]のほかにもいくつか提示されている。プログラムの変換という方向から述べているものとして[3]がある。そこでは、素朴なアルゴリズムをもとに、再帰法を導入して変換を行ない、さらに配列を用いた表を利用して計算の効率改善を図ることによりKMPアルゴリズムを導出している。これに対して、[7]では関数プログラムを用いて、以下のようにして同じアルゴリズムを導出した。

ここでは、PとTとはリスト

$ps_1 = [p_1, p_2, \ldots, p_m]$

$ts_1 = [t_1, t_2, \ldots, t_n]$

で表わされるものとする。また、これらのリストに対して、一般のkについて

$ys_k = [y_k, y_{k+1}, \ldots, y_s]$

とする。リストysの先頭要素を(hd ys)、先頭要素を取り除いた残りのリストを(tl ys)

で表わす。

テキストts$_j$の中にパターンps$_1$があるかどうかを判定する素朴なアルゴリズムは図１のように表わすことができよう。これを用いて、match(ps$_1$,ts$_1$)によって問題を解くことができる。図２はこのアルゴリズムを実行可能なプログラムとして書いたものである。ここで、$\overline{qs}$はリストqsを逆転したリストを表わす。++はリストの連接の演算子、:はリストの手前に要素を付加する演算子である。この変換の根拠は定理１としてあげてある。

次の段階は高階関数の導入である。図３に示すような関数Matchを用いると、

match(qs,ps,ts)=Match(qs,ps,ts)[ ]

であり、関数Matchに関して定理２が成立する。この関係を用いて（単純な書き換えを行なって）、qsを除去すると、図４の定義を得ることができる。図４に現われるMatch(ps$_1$, tl $\tilde{ps}$)はパターンのみに依存するものである。高階関数Matchをこうして部分的に評価することによって、テキストが与えられたときに効率よく照合することのできる関数を得ることができる。これが、いわゆる「失敗処理関数(failure function)」であり、パターンとテキストの照合が失敗したときには、テキスト中を逆戻りする代わりに、この関数をテキストに適用する。

失敗処理関数は漸化式にしたがって求めることもできるので、メモ化(memo-ization)によって効率を改善することができる。

以上がわれわれのプログラム変換法の概略であるが、最終段階のメモ化については、いくらか発見的なところがある。関数プログラミングにおいては、手続き的に処理を記述する際に表を配列で実現する手法(tabulation)は適用できない。表の要素の部分的更新、あるいは破壊的更新によらない方法を用いる必要があるからである。

A naive algorithm for finding a match ps_1 in ts_j:

図 1

$$
\begin{aligned}
& match_j(ps_1, ts_j) \text{ where} \\
& \quad match_j(ps, ts) \\
& \qquad = \text{MATCH}, \; ps=[\,] \\
& \qquad = \text{NO MATCH}, \; ts=[\,] \\
& \qquad = match_j(tl\ ps, tl\ ts), \; hd\ ps = hd\ ts \\
& \qquad = match_{j+1}(ps_1, ts_{j+1})
\end{aligned}
$$

Program 1

図 2

$$
\begin{aligned}
& match([\,], ps_1, ts_1) \text{ where} \\
& \quad match(qs, ps, ts) \\
& \qquad = (qs, [\,], ts), \; ps=[\,] \\
& \qquad = (qs, ps, [\,]), \; ts=[\,] \\
& \qquad = match(hd\ ps{:}qs, tl\ ps, tl\ ts), \; hd\ ps = hd\ ts \\
& \qquad = match([\,], \overline{qs}{+\!+}ps, tl(\overline{qs}{+\!+}ts))
\end{aligned}
$$

where $\overline{qs} = reverse qs$.

Theorem 1

In the definition of *match* in Program 1, $\overline{qs}{+\!+}ps=ps_1$ holds. □

Function *Match*

図 3

$$
\begin{aligned}
& Match(qs, ps, ts)\, xs \\
& \quad = (qs, [\,], ts{+\!+}xs), \; ps=[\,] \\
& \quad = (qs, ps, xs), \; ts=[\,] \\
& \quad = Match(hd\ ps{:}qs, tl\ ps, tl\ ts)\, xs, \; hd\ ps = hd\ ts \\
& \quad = Match([\,], \overline{qs}{+\!+}ps, tl(\overline{qs}{+\!+}ts))\, xs
\end{aligned}
$$

Theorem 2

$$Match(qs, ps, ts'{+\!+}ts'') = Match(Match(qs, ps, ts')ts''). \; □$$

Function *Match*

図 4

$$
\begin{aligned}
& Match(ps, ts)\, xs \\
& \quad = ([\,], ts{+\!+}xs), \; ps=[\,] \\
& \quad = (ps, xs), \; ts=[\,] \\
& \quad = Match(tl\ ps, tl\ ts)\, xs, \; hd\ ps = hd\ ts \\
& \quad = Match(ps_1, tl\ ts)\, xs, \; ps=ps_1 \\
& \quad = Match(Match(ps_1, tl\ \bar{ps})\, ts)\, xs
\end{aligned}
$$

where $\bar{ps}{+\!+}ps=ps_1$.

3. メモ化

経費のかかる計算に対して効率を上げる一つの方法として、一度計算した結果をデータ構造に貯え、再びその計算結果が必要になったときにはデータ構造中の計算結果を使うことによって、同一の計算は再び行なわないで済ませるメモ化の手法が用いられることが多い。とくに、再帰的アルゴリズムと表作成に関する変換法[4]はよく知られている。また、計算過程を考慮しない一般の状況下でもこのようなメモ化を実現する方法も考えられている。計算システムで自動的にメモ化を行なうメモ関数の考え方はその一つといえる。

われわれの必要とするメモ化の機能は、パターンのみによって決まる情報を事前に計算して、いわば表の形に貯えることである。計算の内容については既知であるので、一般的な状況下に対応するメモ関数ではなく、プログラムでデータ構造に貯えるのが適当であろう。

図4において、パターンのみに依存する失敗処理関数Match(ps_1, tl $\widetilde{ps}$)をps_iごとに(i>2に対して)

$$f_i = \mathrm{Match}(ps_1,\ \mathrm{tl}\ [p_1, p_2, \ldots, p_{i-1}])$$

で定めるものとすると、定理2より次のような関係が成立する。

$$\begin{aligned} f_{i+1} &= \mathrm{Match}(ps_1,\ [p_2, \ldots, p_{i-1}, p_i]) \\ &= \mathrm{Match}(\mathrm{Match}(ps_1,\ [p_2, \ldots, p_{i-1}])[p_i]) \\ &= \mathrm{Match}(f_i\ [p_i]) \end{aligned}$$

$$f_2\ s = (ps_1,\ s)$$

そこで、それぞれのps_i、あるいは各文字p_iに対応して、f_iをこの漸化式にしたがって計算する際に再計算を防ぐためにメモ化を用いることが考えられる。これは、パターン照合の際にも利用することができる。

手続き型プログラムにおいては配列を用いるのが一般的である。添字による表の検索や要素の(破壊的)更新が効果的にできるからである。しかし、以下では、このような性質を利用しないでメモ化を実現する方法を示すこととする。これらは、いくらか発見的なものではあるが、関数で表現されている「値」の一部を表によって実現していると考えることもできるからである。

(1) 双方向リストによるメモ化

f_iの計算にはf_{i-1}とp_{i-1}を参照することができればよいので、図5のような双方向リストを用いてメモ化を実現することが考えられる[7]。このような双方向リストは図6のプログラムで得られる。ここでは局所的な再帰的定義が本質的である。

このデータ構造を用いたKMPアルゴリズムのプログラムは図7のようになる。

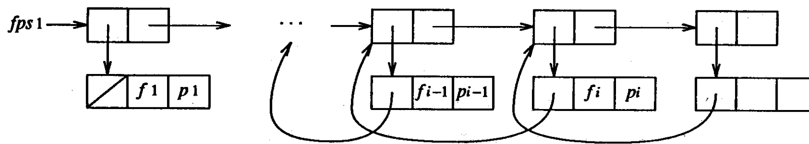

図 5　Pattern list with failure functions

Pattern list with failure functions fps_1

$fps_1 = ([\,], \lambda s.\,(fps_1, tl\ s), hd\ ps_1) : twlist\ fps_1\ (tl\ ps_1)$

where

$twlist\ rs\ ps$

$= [\,], ps = [\,]$

$= rs'$ **whererec** $rs' = (rs, Match(f\ [p]), hd\ ps) : twlist\ rs'\ (tl\ ps)$

where $(rs'', f, p) = hd\ rs$

図 6

Program 2

$match\,(ps_1, ts_1) = Match\,(fps_1, ts_1)\,[\,]$

where

$Match\,(fps, ts)\,xs$

$= ([\,], ts{+\!\!+}xs)$, $fps=[\,]$

$= (fps, xs)$, $ts=[\,]$

$= Match\,(tl\ fps, tl\ ts)\,xs$, $p = hd\ ts$

$= Match\,(f\ ts)\,xs$

where $(rs, f, p) = hd\ fps$ for $fps \neq [\,]$

and fps_1 defined above.

図 7

Construction of data structure for memo-ization ps_1^*

$ps_1^* = (ps_1, S\ ps_1^*, \lambda s.\,(ps_1^*, tl\ s))$

where

$S\ (ps, w, f)$

$= [\,], ps = [\,]$

$= (tl\ ps, S\ w, Match\ (f\ [hd\ ps]))$

図 8

(2) データ構造の構成子によるメモ化

図4のプログラムにおいて、照合に失敗した場合だけでなく、成功したときにも「成功処理関数」Sが用意されているものとすると、

$$ps_i^* = (ps_i,\ S\ ps_i^*,\ f_i)$$

のようなデータ構造を用いることができよう。この関数Sはデータの構成子と見ることができる。この考え方に基づいたメモ化の実現法を図8に与えてある。

この場合には、Sは通常のリストの構成子、すなわち演算子 : と本質的に同じものであり、ps*は、ストリームと対応するものである。

4. おわりに

本稿では、単純なパターン照合プログラムの変換法を扱ったが、これを一般化する際にはいくつか検討すべきことがある。

たとえば、複数個のパターンが与えられたときに、それらのうちのどれかと一致するテキストの部分を求める問題や、複数個のパターンのうちのどれかと一致するような部分列のすべてを求める問題などが考えられる[1]。これらに対する効率のよいアルゴリズムはすでに提示されていて（たとえば、Aho-Corasick法[2]）、われわれの扱ったKMPアルゴリズムの拡張と見られるものも多い。複数個のパターンを考える際にも、パターンのみに依存する部分を抜き出す変換については本稿の第2節とまったく同様に議論することができる。しかし、メモ化についてはまだ十分な成果が得られていない。この段階は、いわば有限オートマトンの状態遷移表を効率よく実現することであるともいえる。

第3節の(2)で述べた方法は一般化することができ、リスト（ストリーム）だけでなく、たとえば、木構造の上にメモ化の情報を貯えることも可能であろう。これらについては、今後の研究に待ちたい。

ここで扱ったプログラムは、それ自体が最適なパターン照合プログラムというわけではなく、また、導出したアルゴリズムもすでに知られているものである。われわれは、新しいアルゴリズムの提案をするものでも、最適性を主張するものでもない。数学的に証明を与えることのできる事実のみによって、正当性を保存するプログラムの変換法を追究しようとするものである。ここでは、証明は省略したが、詳細については[7]を参照されたい。

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Aho, A.V., Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM 18*, 6(1975), 333-340.

[3] Bird, R.S.: Improving Programs by the Introduction of Recursion, *Comm. ACM 20*, 11(1977), 856-863.

[4] Bird, R.S.: Tabulation Techniques for Recursive Programs, *Computing Surveys 12*, 4(1980), 403-417.

[5] Knuth, D.E., Morris, J. H., Jr., and Pratt, V. R.: Fast Pattern Matching in Strings, *SIAM J. on Computing 6*, 2(1977), 323-350.

[6] Takeichi, M.: Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica 24*, 1(1987), 57-77.

[7] Takeichi, M. and Akama, Y.: Deriving Functional Knuth-Morris-Pratt Algorithm by Transformational Programming, *Tech. Rep. Dept. Math. Eng. University of Tokyo*, METR88-08(1988).