知識工学と人工知能 60 － 8
（１９８８．９．２１）

高階の差分を用いた入出力例からの関数の合成

犬塚 信博　　石井 直宏
名古屋工業大学　電気情報工学科

有限個の入出力例から、そのような振舞いをする関数を導く方法について論ずる。関数の領域はＬＩＳＰのＳ式である。本報告の内容は、主に二つある。一つは、この種の方法の一つである、Summersによるアルゴリズムを、高階の差分を導入することで拡張したことである。これは、彼の方法の繰り返し的な適用に基づくということ、また、一般性や探索をほとんど含まないといった特徴を継承している点で、自然な拡張である。もう一つは、この種のアルゴリズムに与えるべき、ふさわしい例の集合について考察し、任意に与えられた集合を、ふさわしい集合に変換する手続きを提案したことである。このふさわしい集合を、典型的集合という。この変換を付加することで、アルゴリズムの適用範囲が広がり、柔軟性を持たせることができる。

Construction of Functions from Examples by Using High Order Differences

Nobuhiro Inuzuka and Naohiro Ishii
Department of Electrical and Computer Engineering
Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya, 466 JAPAN

We developed a method for constructing the functions from a finite number of examples of what they do. The domains of the functions treated are the set of S-expressions of LISP. This paper contains two matters. First, we extend the Summers' algorithm that is a kind of the constructing algorithms by using the high order differences. This extension can be considered to be natural, because it is based on the repeats of Summers' algorithm, and inherits the generality and the unnecessity of search, which are merits of his algorithm. Second, we study how set of examples fits to this method. We call the fitted set typical. And, we present a procedure that transforms an arbitrary set into a typical set. The method with this procedure becomes of wide application and flexible.

1．はじめに

有限個の事実から一般的な法則を見出だす推論の方式を、帰納的推論と言う。LISPで言うS式上の関数の自動生成を、この帰納的推論のプロトタイプとして検討するのが、本稿の目的である。ここで考慮に入れる関数のクラスは、S式を定義域および値域とする関数の中で、出力が、そこに含まれるアトムの性質に依存しないようなものである。

LISPの関数を、その入出力例から導くというテーマは、Summersがある関数生成の方法を提案[1]して以来、多くの研究に引き継がれている[2]。彼の方法は、特殊なヒューリスティックを与えないという意味で、一般性のあるものである。

本報告では、この方法の主眼である、優れた一般性から逸脱せずに、この方法の自然な拡張を試みる。また、この拡張された方法により生成された関数が、与えられた例、およびそれに自然と思われる汎化(帰納的推論)を施した結果と一致するという意味で、正当であることを示す。また、この種の方法では、関数のクラスを表現するために、その関数の振舞いを表現する、適当な例の集合が必要である。そこで、そのような集合がどのようなものであるかについて考察し、一般的な例の集合、即ち、任意に与えられた適当とは言えない例の集合から、そのような、適当な集合を導くための方法について言及する。このことにより、関数生成の手続きに柔軟性を持たせることができる。

2．入出力例からの関数の生成

2．1　対象と関数フラグメント

ここでの検討の対象は、1節で述べたように、LISPでのS式を定義域、値域とする関数である。ただし、その出力は、入力されたS式に含まれるアトムそのものの性質には依存しない。つまり、リスト(2 7 1)を入力とし、そこから要素を加算して、10を得るような関数は、考慮に入れない。例えば、ここで考える関数は、次の例の集合で示されるようなものである。

{( )　→ ( )；　　(A)　→ ((A))；
(A B) → ((A) (B))；(A B C) → ((A) (B) (C))}

この集合で示される関数は、「リストの第一要素がAである」といった、含まれるアトムの性格ではなく、リストに含まれる要素の個数など、リストの構造のみに出力の形式が左右される。このような範囲で考えるなら、基本関数としてcar,cdr,cons,atomだけを用いればよい。(アトムを判別する必要はないので、eq等は要らない。)

基本関数の単なる組み合わせ(合成)からなる関数として、関数フラグメントを定義する。便宜上、ラムダ記法を用いる。関数フラグメントは、S式からS式への単純な変換を記述するものとも考えられる。

[定義2－1]　n引数の関数フラグメントの集合FFnは、次の(1)～(4)のみで導かれる。

(1) aがアトムならば $\lambda x_1 \cdots x_n . a \in FF_n$

(2) $\lambda x_1 \cdots x_n . x_i \in FF_n,\ 1 \leq i \leq n$

(3) $f \in FF_n$ならば

$\lambda x_1 \cdots x_n . car[f[x_1;\cdots;x_n]] \in FF_n,$

$\lambda x_1 \cdots x_n . cdr[f[x_1;\cdots;x_n]] \in FF_n$

(4) $f_1, f_2 \in FF_n$ならば

$\lambda x_1 \cdots x_n . cons[f_1[x_1;\cdots;x_n];$
$f_2[x_1;\cdots;x_n]] \in FF_n$　□

また、述語のクラスとして、次のPを定義する。この述語は、生成される関数の定義において用いるもので、入力されたS式のタイプを区別して、それぞれで適した操作をするためのものである。

[定義2－2]　述語の集合Pは、1引数の関数フラグメントFF_1から、次のように導かれる。

$f \in FF_1$ならば $\lambda x . atom[f[x]] \in P$　□

2．2　Summersのアルゴリズム

有限個の入出力例の集合から、それが示すような振舞いをする関数を導くためのアルゴリズムが、Summersによって与えられている[1]。ここで、k個の入出力例の集合 $\{x_i \rightarrow y_i \mid i \leq k\}$ が与えられているとすると、そのアルゴリズムは、次のような手順からなる。('$x \rightarrow y$'は、入出力関係であることを強調してこのように書くが、単に対を意味している。)

(1) 各入出力例 $x_i \rightarrow y_i$ から、x_i を y_i に変換する関数フラグメント(function fragment) $f_i \in FF_1$ を求める。つまり、$y_i = f_i[x_i]$ の関係を持つ。

(2) 関数フラグメント間の規則性を、繰り返し的関係として見出だし、次のように記述する。

$\forall x, f_{i+1}[x] = a[f_i[b[x]];x],\ i \leq k-2$

(3) $\{x_1, x_2, \cdots, x_k\}$ から、次のような述語の集合 $\{p_1, p_2, \cdots, p_k\}$ を導く。

$p_i[x_j] = true\ (i=j)\ or\ false\ (i \leq j)$
(その他のi,jでは任意)

(4) 述語間の規則性を繰り返し的関係として、次のように記述する。

$\forall x, p_{i+1}[x] = p_i[b[x]],\ i \leq k-2$

(5) 関数フラグメントの規則性、述語の規則性を、与えた例についてだけでなく、一般的に成り立つものとして汎化する。

(6) 汎化された規則性から、関数を再帰的プログラムとして生成する。

最後の、規則性からのプログラムの生成は、次の定理により、生成されたプログラムの正当性が保証される。

[定理2－3] (基本構成定理)　繰り返し関係、

$f_1,\ f_{i+1} = \lambda x.a[f_i[b[x]];x]$

$p_1,\ p_{i+1} = \lambda x.p_i[b[x]]\quad i \geq 1$

として記述される関数は、次の再帰的な関数Fと等価である。

$F \leftarrow \lambda x.[p_1[x] \rightarrow f_1[x];\ T \rightarrow a[F[b[x]];x]$　□

この定理は、Summersのアルゴリズムを支えるものであり、その証明は4節で行う。

2．3　例題

次に、先程説明に用いた例の集合に適用して、上の手続きを眺める。

{ ( ) → ( );　(A) → ((A));
(A B) → ((A) (B)); (A B C) → ((A) (B) (C)) }

これらから、それぞれ関数フラグメントを求める。順に対応して、次のようになる。

$f_1 = \lambda x.nil$

$f_2 = \lambda x.cons[cons[car[x];nil];nil]$

$f_3 = \lambda x.cons[cons[car[x];nil];$
$\quad cons[cadr[x];nil];nil]]$

$f_4 = \lambda x.cons[cons[car[x];nil];$
$\quad cons[cadr[x];nil];$
$\quad\quad cons[caddr[x];nil];nil]]]$

例えば、関数フラグメントf_3が、$f_3[(A\ B)] = cons[cons[A;nil];cons[B;nil];nil]] = ((A)\ (B))$ であるように、各関数フラグメントは入出力間の関係を示している。これらの関数フラグメントの間には、次のような規則性を見ることができる。

$f_{i+1} = \lambda x.cons[cons[car[x];nil];f_i[cdr[x]]]$
$\quad (i = 2,3,4)$　(1)

また、各例の入力を識別するための述語は、次のように求められる。

$p_1 = \lambda x.atom[x]$

$p_2 = \lambda x.atom[cdr[x]]$

$p_3 = \lambda x.atom[cddr[x]]$

$p_4 = \lambda x.t$　(tは真であることを表す定数)

これらの間にも、次のような規則性がある。

$p_{i+1} = \lambda x.p_i[cdr[x]]\quad (i = 2,3)$　(2)

(1),(2)式について、iに関する条件を外し、汎化する。

$f_{i+1} = \lambda x.cons[cons[car[x];nil];f_i[cdr[x]]]$

$p_{i+1} = \lambda x.p_i[cdr[x]]\quad (i \geq 2)$　(3)

この繰り返し的な規則性に、定理2－3を適用することで次の関数unpackを得る。

```
unpack = λx.
    [atom[x] → nil;
     T → cons[cons[car[x];nil];
              unpack[cdr[x]]]]
```

この関数は、与えた例の振舞いを記述するだけでなく、その一般化になっている。

しかし、このようにして関数を一般化するには、与えられた例から導かれる関数フラグメントに、明白な規則性がなくてはならない。つまり、関数フラグメントの列において、ある関数フラグメントが、一定の形式で、一つ前の関数フラグメントにより、記述されなければならない。また、同様の順序で、述語も一つ前の述語を使って、一定の形式で表されなければならない。この規則性の発見は、条件としてかなり厳しいものであり、広い範囲で発見できるとは限らない。Summersは、そのため変数付加(variable addition)の方法[1]を用いている。しかし、ここでは変数付加を用いず、多少なりともこの方法の能力を高めることを考える。

3．高階の差分を用いた関数の生成

3．1　規則性発見の拡張へ

2節で述べた方法では、関数フラグメントが各々、

$f_{i+1}[x] = a[f_i[b[x]];x]$

といった一定の形式を用いて、関数フラグメントの列中、一つ前のもの(一つ若い添字を持つもの)で表されなければならなかった。つまり、関数フラグメントf_{i+1}が、一つ前の関数フラグメントf_iと、2引数関数a、1引数関数bを使って、表現されなければならない。ところが実際には、aは一定にならず、規則性を持ちながらもそれぞれで異なったものになることが多い。(bも同様なことが言える。しかし、その可能性は低く、ここではaのみを考えることにする。)そこで、aを添字に依存する形で表してみる。

$f_2 = \lambda x.a_1[f_1[b[x]];x]$

$f_3 = \lambda x.a_2[f_2[b[x]];x]$

・・・

$f_{i+1} = \lambda x.a_i[f_i[b[x]];x]$

・・・

そして、このa_iについての規則性を、f_iの規則性を求めるときに用いたものと同様の方法により、発見することを考える。つまり、Summersの方法の繰り返し的適用を考える。

3．2　差分と繰り返し関係

2節で述べたような、関数フラグメントを別の関数フラグメントで表すという操作を、'差分'という。

[定義3－1]　2つの1変数関数フラグメント $f_1, f_2 \in FF_1$に対し、任意の入力xで、

$f_2[x] = a[f_1[b[x]];x]$

となる$a \in FF_2$, $b \in FF_1$を求めることを、差分する'という。また、aを'f_2のf_1に対する差分関数フラグメント'、bを'引き渡し関数'という。　□

[定義3－2]　関数フラグメントのn組$(f_1, \cdots, f_n)$に対し、$f_i^{(1)}$がf_{i+1}のf_iに対する差分関数で、そのときの引き渡し関数が全て等しいとき、$(f_1^{(1)}, \cdots, f_{n-1}^{(1)})$を、'$(f_1, \cdots, f_n)$の差分'という。　□

Summersのアルゴリズムは、与えられた例の集合から作られた関数フラグメントの組の差分が、全て同じ差分関数フラグメントになることで、規則性を見付けるものであった。そうした差分関数フラグメントを見付けることにより、関数フラグメントの組を繰り返し関係として記述できる。ここでは、差分の概念を次のように拡張する。このことにより、一定にならなかった関数フラグメントの中に規則性を見出だす手段を与える。

[定義3－3] (3－1の拡張)　2つのn変数関数フラグメント$f_1, f_2 \in FFn$に対し、任意の入力$x_1, \cdots, x_n$で、

$f_2[x_1;\cdots;x_n] = a[f_1[b_1[x_1];\cdots;b_n[x_n]];x_1;\cdots;x_n]$

となる$a \in FF$　および$b_1, \cdots, b_n \in FF_1$を求めることを、'差分する'という。また、aを'$f_2$の$f_1$に対する差分関数フラグメント'、$b_1, \cdots, b_n$を'引き渡し関数'という。　□

定義3－2で定義した'差分'は、定義3－3での拡張された定義でも、そのまま言うことにする。

[定義3－4]　関数フラグメントのn組$(f_1, \cdots, f_n)$に対し、その差分$(f_1^{(1)}, \cdots, f_{n-1}^{(1)})$を、'1階差分'という。また、1階差分中の関数フラグメント$f_i^{(1)}$を、'1階差分関数フラグメント'という。

m階差分$(f_1^{(m)}, \cdots, f_{n-m}^{(m)})$の差分、$(f_1^{(m+1)}, \cdots, f_{n-m-1}^{(m+1)})$を'm＋1階差分'という。　□

与えられた例の集合から作られた関数フラグメントに対し、1階差分、2階差分、3階差分、… と取り、ある所で一定になれば規則性を見出だすことができる。

例えば、次の例を取り上げる。

```
{( )          → ( );
 (A)          → (A);
 (A B)        → (B B A);
 (A B C)      → (C C C B B A);
 (A B C D)    → (D D D D C C C B B A);
 (A B C D E)  → (E E E E E D D D D C C C C B B A)}
```

ここから構成される関数フラグメントは、それぞれ次のようになる。

$f_1 = \lambda x_1.nil$

$f_2 = \lambda x_1.cons[car[x_1];nil]$

$f_3 = \lambda x_1.cons[cadr[x_1];cons[cadr[x_1];$
$cons[car[x_1];nil]]]$

$f_4 = \lambda x_1.cons[caddr[x_1];cons[caddr[x_1];$
$cons[caddr[x_1];cons[cadr[x_1];$
$cons[cadr[x_1];cons[car[x_1];nil]]]]]]$

$f_5 = \lambda x_1.cons[cadddr[x_1];cons[cadddr[x_1];$
$cons[cadddr[x_1];cons[cadddr[x_1];$
$cons[caddr[x_1];cons[caddr[x_1];$
$cons[caddr[x_1];cons[cadr[x_1];$
$cons[cadr[x_1];cons[car[x_1];$
$nil]]]]]]]]]$

$f_6 = \lambda x_1.cons[caddddr[x_1];cons[caddddr[x_1];$
$cons[caddddr[x_1];cons[cadddr[x_1];$
$cons[cadddr[x_1];cons[cadddr[x_1];$
$cons[cadddr[x_1];cons[cadddr[x_1];$
$cons[cadddr[x_1];cons[caddr[x_1];$
$cons[caddr[x_1];cons[caddr[x_1];$
$cons[cadr[x_1];cons[cadr[x_1];$
$cons[car[x_1];nil]]]]]]]]]]]]$

したがって、$(f_1, f_2, f_3 f_4, f_5, f_6)$の1階差分$(f_1^{(1)}, f_2^{(1)}, f_3^{(1)}, f_4^{(1)}, f_5^{(1)})$は次のようになる。

$f_1^{(1)} = \lambda x_1 x_2.cons[car[x_2];x_1]$

$f_2^{(1)} = \lambda x_1 x_2.cons[cadr[x_2];cons[cadr[x_2];x_1]]$

$f_3^{(1)} = \lambda x_1 x_2.cons[caddr[x_2];cons[caddr[x_2];$
$cons[caddr[x_2];x_1]]]$

$f_4^{(1)} = \lambda x_1 x_2.cons[cadddr[x_2];cons[cadddr[x_2];$

cons[cadddr[x2];cons[cadddr[x2];x1]]]]

$f_5^{(1)}$ = λx1x2.cons[caddddr[x2];cons[caddddr[x2];
cons[caddddr[x2];cons[caddddr[x2];
cons[caddddr[x2];x1]]]]]

このときの引き渡し関数はλx.xである。

もう一度差分することで2階差分($f_1^{(2)}$, $f_2^{(2)}$, $f_3^{(2)}$, $f_4^{(2)}$)を得る。

$f_1^{(2)}$ = λx1x2x3.cons[cadr[x3];x1]

$f_2^{(2)}$ = λx1x2x3.cons[caddr[x3];x1]

$f_3^{(2)}$ = λx1x2x3.cons[cadddr[x3];x1]

$f_4^{(2)}$ = λx1x2x3.cons[caddddr[x3];x1]

このときの引き渡し関数は、それぞれλx.x, λx.cdr[x]である。

更にもう一度差分することで3階差分($f_1^{(3)}$, $f_2^{(3)}$, $f_3^{(3)}$)を得る。

$f_1^{(3)}$ = λx1x2x3x4.x4

$f_2^{(3)}$ = λx1x2x3x4.x4

$f_3^{(3)}$ = λx1x2x3x4.x4

このときの引き渡し関数は、それぞれ、λx.x,任意,λx.cdr[x]である。

この3階差分に含まれる関数フラグメントが全て一致しており、これで規則性が見出だされた。規則性は次のような繰り返し関係である。

f_1 = λx1.nil, f_{i+1} = λx1.$f_i^{(1)}$[f_i[x1]];x1]

ただし、

$f_1^{(1)}$ = λx1x2.cons[car[x2];x1],

$f_{i+1}^{(1)}$ = λx1x2.$f_i^{(2)}$[$f_i^{(1)}$[x1;cdr[x2]];x1;x2]

$f_1^{(2)}$ = λx1x2x3.cons[cadr[x3];x1],

$f_{i+1}^{(2)}$ = λx1x2x3.$f_i^{(2)}$[x1;x2;cdr[x3]]

述語についても同様である。例の集合から生成される述語は次の通りである。

p_1 = λy.atom[y]

p_2 = λy.atom[cdr[y]]

p_3 = λy.atom[cddr[y]]

p_4 = λy.atom[cdddr[y]]

p_5 = λy.atom[cddddr[y]]

これは、1階差分によって次のような規則性にまとめられる。

p_1 = λy.atom[y], p_{i+1} = λy.p_i[cdr[y]]

これら関数フラグメントおよび述語の繰り返し関係から関数を生成した結果について、次に述べる。

3．3　関数の生成

3．2で述べたように、差分よって求められた規則性から、関数が求められる。これは、定理2−3を拡張した次の定理に従う。この定理は、4節で証明される。

[定理3−5]　繰り返し関係、

f_1, f_{i+1} = λx.$f_i^{(1)}$[f_i[b_1[x]];x]

p_1, p_{i+1} = λx.p_i[bp[x]]　i≧1

ただし、

$f_1^{(1)}$, $f_{i+1}^{(1)}$ = λx1x2.
$f_i^{(2)}$[$f_i^{(1)}$[$b_1^{(1)}$[x1];$b_2^{(1)}$[x1]];x1;x2]

. . .

$f_1^{(n-2)}$, $f_{i+1}^{(n-2)}$ = λx1…xn-1.
$f_i^{(n-1)}$[$f_i^{(n-2)}$[$b_1^{(n-2)}$[x1];…;$b_{n-1}^{(n-2)}$[xn-1]];
x1;…;xn-1]

$f_1^{(n-1)}$, $f_{i+1}^{(n-1)}$ = λx1…xn.
a[$f_i^{(n-1)}$[$b_1^{(n-1)}$[x1];…;$b_n^{(n-1)}$[xn]];
x1;…;xn]

として記述される関数は、次の関数Gと等値である。

G = λx.F[x;x]

ただし、

F = λx1y.
[p_1[y] → f_1[x1];
T → $F^{(1)}$[F[b_1[x1];bp[y]];x1;y]]

$F^{(1)}$ = λx1x2y.
[p_2[y] → $f_1^{(1)}$[x1;x2];
T → $F^{(2)}$[$F^{(1)}$[$b_1^{(1)}$[x1];$b_2^{(1)}$[x2];bp[y]];
x1;x2;y]]

. . .

$F^{(n-2)}$ = λx1…xn-1y.
[p_{n-1}[y] → $f_1^{(n-2)}$[x1;…;xn-1];
T → $F^{(n-1)}$[$F^{(n-2)}$[$b_1^{(n-2)}$[x1];…;$b_{n-1}^{(n-2)}$[xn-1];bp[y]]
x1;…;xn-1;y]]

$F^{(n-1)}$ = λx1…xny.
[p_n[y] → $f_1^{(n-1)}$[x1;…;xn];
T → a[$F^{(n-1)}$[$b_1^{(n-1)}$[x1];…;$b_n^{(n-1)}$[xn];bp[y]];
x1;…;xn]]　□

これを3．2の例題で得た規則性に適用することで、次のような関数Gが得られる。

G = λx.F[x;x]

ただし、

F = λx1y.
[atom[y] → nil;
T → $F^{(1)}$[F[x1;y];x1;y]]

$F^{(1)}$ = λx1x2y.

$$[atom[cdr[y]] \rightarrow cons[car[x_2];x_1];$$
$$T \rightarrow F^{(2)}[F^{(1)}[x_1;cdr[x_2];y];x_1;x_2;y]]$$

$$F^{(2)} = \lambda x_1 x_2 x_3 y.$$
$$[atom[cddr[y]] \rightarrow cons[cadr[x_3];x_1];$$
$$T \rightarrow F^{(2)}[x_1;x_2;cdr[x_3];y]]$$

である。

この手法は、関数生成の一つの方法であり、Summersの基本的な手法を包含するものである。また、場合によっては、彼の変数付加の方法と同等か、それ以上の振舞いをする。しかし、Summersの変数付加の技法に取って代わるという意味ではない。これを他の手法と組み合わせて用いることで、より効果があると考える。

4．構成定理の証明

2．2で述べた定理2−3は、[1]にその証明が与えられている。その概略としては、汎関数、

$$\tau = \lambda F.\lambda x.[p_1[x] \rightarrow f_1[x];$$
$$T \rightarrow a[F[b[x]];x]]$$

を考え、これが不動点を持つこと、そして、その最小不動点が繰り返し関係として与えられる関数と等価であること、を証明することである。

この節では、定理2−3の拡張としての定理3−5を、2階の場合に限って証明する。

ここでの証明も、不動点意味論[3]に基づいて行う。そこで、再帰的な関数を汎関数による記述に直し、その汎関数の不動点として論ずる。この定理を汎関数を使った記述に直すと、次のようになる。

[2階差分を使った構成定理]

次の汎関数θ: $D_3 \rightarrow D_3$は不動点を持つ。

$$\theta = \lambda F.\lambda x_1 x_2 y.$$
$$[p_1[b_2[y]] \rightarrow f_1^{(1)}[x_1;y];$$
$$T \rightarrow a[F[b_1^{(1)}[x_1];b_2^{(1)}[x_2];bp[y]];x_1;x_2]]$$

その最小不動点をF^1とする。このとき、次の汎関数$\sigma: D_2 \rightarrow D_2$は不動点を持つ。

$$\sigma = \lambda F.\lambda x_1 y.$$
$$[p_1[y] \rightarrow f_1[x_1];$$
$$T \rightarrow F^{(1)}[F[b_1[x_1];bp[y]];x_1;y]]$$

その最小不動点をFとする。このとき、次のように定義される関数Gは、

$$G = \lambda x.F[x;x]$$

次の繰り返し関係によって定義される関数Fと等価である。

$$f_1[x], f_{i+1}[x] = f_i^{(1)}[f_i[b_1[x]];x]$$
$$p_1[x], p_{i+1}[x] = p_i[bp[x]]$$

ただし、

$$f_1^{(1)}[x;y],$$
$$f_{i+1}^{(1)}[x;y] = a[f_i^{(1)}[b_1^{(1)}[x];b_2^{(1)}[y]];x;y]$$

である。 □

ここで、S式の集合をS、未定義要素をωで表すと、D_3は、$D_3 = ((S \cup \{\omega\}) \times (S \cup \{\omega\}) \times (S \cup \{\omega\}) \rightarrow (S \cup \{\omega\}))$である。同様に、$D_2 = ((S \cup \{\omega\}) \times (S \cup \{\omega\}) \rightarrow (S \cup \{\omega\}))$である。

前に述べたような再帰的関数Fや$F^{(1)}$は、上の定理のθやσの最小不動点である。

D_3の元は、$(S \times S \times S \rightarrow S)$の元として表される関数の近似であると見ることができる。そこで、近似の度合による半順序$\leq_F$を、次のように考える。

[定義4−1] $F, G \in D_3$に対し、$F \leq_F G$
$\Leftrightarrow \forall x \in S, F[x] \neq \omega$ならば$F[x] = G[x]$ □

すると次の諸性質を示すことができる。

[性質4−2] $\leq_F$はD_3において半順序を成す。 □

[性質4−3] D_3上の$\leq_F$に関する任意の鎖(chain)、つまり、$F_0, F_1, \cdots (F_i \in D_3, F_i \leq_F F_{i+1})$は一意な極限F(即ち、$\{F_0, F_1, \cdots\}$の一意な最小上界、$\lim_{n\to\infty} F_n$と書く。)を持つ。 □

鎖$F_0, F_1, \cdots$の一意な極限は、次のように定まるFであることが証明できる。

$x, y, z \in D_3$に対し、

$$F[x;y;z] = \begin{cases} a & \text{あるnで} Fn[x;y;z] = a \\ & (\neq \omega) \text{となるとき。} \\ \omega & \text{その他のとき。} \end{cases}$$

[性質4−4] D_3は、$\leq_F$に関して$\Omega_3 = \lambda x y z.\omega$なる最小元を持つ。 □

汎関数$\theta: D_3 \rightarrow D_3$の連続性についても次の性質がある。

[性質4−5] 汎関数$\theta: D_3 \rightarrow D_3$は連続である。即ち、次の2つの条件を満たす。

(1) θは単調である。つまり、任意の$x, y \in D_3$に対し、$x \leq_F y$ならば$\theta[x] \leq_F \theta[y]$。

(2) D_3上の任意の鎖$F_0, F_1, \cdots$に対し、次の式が成り立つ。 $\theta[\lim_{n\to\infty} Fn] = \lim_{n\to\infty} \theta[Fn]$ □

こうした性質(性質4−2〜5)、および次の不動点定理[3][4]により、θが最小不動点として、$\lim_{n\to\infty} \theta^n[\Omega_3]$を持つことが結論される。$\sigma$についても同様に、最小不動点として、$\lim_{n\to\infty} \sigma^n[\Omega_2]$を取る。ただし$\Omega_2 = \lambda x y.\omega$である。

[定理4−6] (不動点定理)[3][4] D_1が次の(1)〜(3)を満たすとき、連続関数$f: D_1 \rightarrow D_2$は、$x = \lim_{n\to\infty} f^n[\omega]$で

与えられる最小不動点を持つ。

(1) D_1はある関係$\leq$において半順序集合を成す。

(2) D_1は$\leq$に関して最小元ωを持つ。

(3) D_1の鎖はすべてD_1に一意な極限を持つ。 □

次に、このように最小不動点として与えられる関数が、繰り返し関係で定義される関数と等値なものになることを示す。

まず、繰り返し関係によって与えられる$p_1, p_2, \cdots$および$f_1^{(1)}, f_2^{(1)}, \cdots$を用いた、次の関数$F^{m(1)}$を考える。

$$\begin{aligned}F^{m(1)} = \lambda x\, y\, z.[&p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &p_2[bp[z]] \rightarrow f_2^{(1)}[x;y];\\ &\quad\cdots\\ &p_m[bp[z]] \rightarrow f_m^{(1)}[x;y];\\ &T \rightarrow \omega]\end{aligned}$$

$F^{m(1)}$は、$\{x \mid p_1[x] \vee p_2[x] \vee \cdots \vee p_m[x]\}$においてのみ、(未定義要素$\omega$ではない)値を取る関数である。これを$F^{(1)}$の第m近似という。

同様にして、Fの第m近似F_mも次のように考える。

$$\begin{aligned}F_m = \lambda x\, y.[&p_1[y] \rightarrow f_1[x];\\ &p_2[y] \rightarrow f_2[x];\\ &\quad\cdots\\ &p_m[y] \rightarrow f_m[x];\\ &T \rightarrow \omega]\end{aligned}$$

すると、このF_mにおけるmを無限大にしたものは、繰り返し関係によって与えられる関数の言い替えになっている。よって、$\lim_{m\to\infty} \theta^m[\Omega_3] = \lim_{m\to\infty} F_m^{(1)}$、および、その結果を用いて$\lim_{m\to\infty} \sigma^m[\Omega_2] = \lim_{m\to\infty} F_m$を示すことにより、最小不動点が、繰り返し関係で定義される関数と等価なものになることを言うことができる。

特に、ここでは前者のみを示すことにする。そのためには、$\theta^m[\Omega_3] = F^{m(1)}$を数学的帰納法を用いて証明すればよい。

まず、m＝1のときは次のように成り立つ。

$$\begin{aligned}&\theta^1[\Omega_3][x;y;z]\\ &= [p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &\quad T \rightarrow a\,[\Omega_3[b_1^{(1)}[x];b_2^{(1)}[y];bp[z]];x;y]]\\ &= [p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &\quad T \rightarrow \omega]\\ &= F_1^{(1)}[x;y;z]\end{aligned}$$

m＝kのとき、次のように仮定する。(帰納法の仮定)

$$\theta^k[\Omega_3] = F_k^{(1)}$$

するとm＝k＋1のとき、

$$\begin{aligned}&\theta^{k+1}[\Omega_3][x;y;z]\\ &= \theta\,[\theta^k[\Omega_3]][x;y;z]\\ &= \theta\,[p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &\qquad\cdots\\ &\quad p_k[bp[z]] \rightarrow f_k^{(1)}[x;y];\\ &\quad T \rightarrow \omega]\\ &= [p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &\quad T \rightarrow\\ &\quad a\,[p_1[bp[bp[z]]] \rightarrow f_1^{(1)}[b_1^{(1)}[x];b_2^{(1)}[y]];\\ &\qquad\cdots\\ &\quad p_k[bp[bp[z]]] \rightarrow f_k^{(1)}[b_1^{(1)}[x];b_2^{(1)}[y]];\\ &\quad T \rightarrow \omega];x;y;z]]\\ &= [p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &\quad p_1[bp[bp[z]]] \rightarrow a\,[f_1^{(1)}[b_1^{(1)}[x];b_2^{(1)}[y]];\\ &\qquad x;y];\\ &\qquad\cdots\\ &\quad p_k[bp[bp[z]]] \rightarrow a\,[f_k^{(1)}[b_1^{(1)}[x];b_2^{(1)}[y]];\\ &\qquad x;y];\\ &\quad T \rightarrow \omega]\\ &= [p_1[bp[z]] \rightarrow f_1^{(1)}[x;y];\\ &\quad p_2[bp[z]] \rightarrow f_2^{(1)}[x;y];\\ &\qquad\cdots\\ &\quad p_{k+1}[bp[z]] \rightarrow f_{k+1}^{(1)}[x;y];\\ &\quad T \rightarrow \omega]\\ &= F_{k+1}^{(1)}[x;y;z]\end{aligned}$$

となり、k＋1でも成り立つ。ゆえに、常に$\theta^m[\Omega_3] = F^{m(1)}$が言え、極限でも$\lim_{m\to\infty} \theta^m[\Omega_3] = \lim_{m\to\infty} F^{m(1)}$が言える。したがって、$\lim_{m\to\infty} F^{m(1)}$が$\theta$の最小不動点であることが分かった。

同様に、$\lim_{m\to\infty} F_m$がσの最小不動点であることも、常に$F_m = \sigma^m[\Omega_2]$となることを示すことで証明できる。

したがって、θやσで与えられる再帰的な関数が、繰り返し的な関係によって与えられる関数と、一致することが分かる。

5．非典型的な例からの関数の構成

5．1　典型的な例の集合

ここまで、例の集合から関数を導くということについて考察してきたが、そこで用いてきた例の集合は次のような形のものであった。

$$\begin{aligned}\{&(\,) \rightarrow (\,);\\ &(a) \rightarrow ((a));\\ &(a\ b) \rightarrow ((a)\ (b));\\ &(a\ b\ c) \rightarrow ((a)\ (b)\ (c))\}\end{aligned}$$

これは、入力リストの要素の数が一つずつ増え、そしてそれに伴って出力も順に変化している。また、増えてゆく要素は、単純なアトムである。このような、いわば整然とした集合を与えることで、初めてアルゴリズムは動作することになる。アルゴリズムを動作させるのに適した、このような例の集合、つまり、求めたい関数の性質を最もよく記述していると考えられる例の集合を、'典型的である'ということにする。また、典型的な例の集合に含まれる一つ一つの例を典型例という。

典型的な例の集合は、下の条件(1)～(5)を満たす必要があると考えられる。ここで、関係'≦'は、次のようなもので、S式全体の集合において半順序となることが示せる。ただし、ωは任意のアトムを、またa,b,c,dは任意のS式を表す。

ω≦a, a≦bかつc≦d ⇒ (a.c)≦(b.d)

そして、S式はこの関係において束を作ることも示せる。また、二つの入力例$x_1 \rightarrow y_1$と$x_2 \rightarrow y_2$ ($x_1 \leq x_2$)が、'同じ型の入出力例である'とは、$x_1 \rightarrow y_1$からできる関数フグメントfが$y_2 = f[x_2]$となる、つまり、$x_2 \rightarrow y_2$にも当てはまることである。

(1) 関数の特殊な振舞いは網羅している。
(2) 各入出力例は、それと同じ型の入出力関係を示すものの中で、最も簡単な(入力部分を半順序≦で並べたとき最も小さな)ものである。
(3) 同じ型の入出力例は、高々一回与えられている。
(4) 各例を、その入力部分に半順序≦を用いて並べたとき、抜けがない。つまり、$x_1 \rightarrow y_1$および$x_2 \rightarrow y_2$なる二例が与えられた例の集合に含まれており、もし、$x_1 \leq x \leq x_2$なるx ($\neq x_1, x_2$)が存在するならば、そのxに関する例も与えられている。
(5) 十分多く与えられている。

5．2 非典型的な例から典型的な例への変換

前に挙げた(1)～(5)を満たすような場合は、2、3で述べたようなアルゴリズムが、そのまま適用可能である。しかし、任意に与えられた例の集合、つまり、(1)～(5)を満たさないような集合から、それを説明する関数を導くという問題も、興味深い問題である。また、これを解決することは、アルゴリズムの適用範囲を広げるという意味で、柔軟性を高めることにもなる。

そこで、任意に与えられた例の集合を、典型的な例の集合に変換する方法を考察し、それを与える。この変換の目的は、不必要な例を削除し、必要な例を与える入力を求めることにある。したがって、どのような例が必要で、どのような例が不必要かについてを処理し、実際に入出力関係がどうなるかについては、基本的に外部に問い合わせることにする。しかし、他の例から推測可能な部分はアドホックな方法を用いて類推することにし、その後外部に確認する。また、分からない部分は、直接外部に問い合わせることにする。

任意の例の集合$EX = \{x_i \rightarrow y_i \mid 1 \leq i \leq n\}$が与えられたとして、その手順を次に示す。また、入力部分の集合を$IN = \{x_i \mid 1 \leq i \leq n\}$とする。

(1) 与えられた例の集合EX内の各例を節とし、$x_i, x_j \in IN$が、$x_i \leq x_j$で、$x_i \leq x_k \leq x_j$ ($x_k \in IN, x_k \neq x_i, x_j$)なる$x_k$が存在しないとき、$x_i \rightarrow y_i$から$x_j \rightarrow y_j$への有向辺を持つ有向グラフTを考える。
(2) nilを入力とした例が与えられていないとき、問い合わせなどしてTに加える。
(3) Tの分岐点(子が2つ以上ある節)について、そこから出ている辺の内、同じ型の入出力関係を示す子へ入る辺を開放除去する。同時に、分岐点とその子の間に抜けがある場合は、その入力に対する入出力関係を、他からの類推などによって埋め、Tに加える(その入出力例を節として加え、それに対応する辺も加える)。結果のグラフに対し、nilの入出力例の属する連結成分を、新たにTとする。
(4) (3)で抜けを埋めたために生ずる分岐点において、同じ型の入出力関係を示す子への辺を開放除去する。そして、nilの入出力例の属する連結成分を、新たにTとする。
(5) 各要素間に抜けがあれば埋め、Tに加える。
(6) nilの入出力例から辿り、同じ入出力関係を持つ子が現れたら、それへの辺を開放除去し、nilの入出力例の属する連結成分を、新たにTとする。

5．3 変換の例

次の例が与えられた場合を考える。

```
{( )          → ( );
 (a)          → ((a));
 ((a))        → (((a)));
 ((a) b)      → (((a)) (b));
 (a b c)      → ((a) (b) (c));
 (a (b) c d)  → ((a) ((b)) (c) (d))}
```

(1) この例の入力部分の集合は、

{( ), (a), ((a)), ((a) b), (a b c), (a (b) c d)}

であり、これらの間の順序関係により得られるグラフTは図１．aのようになる。ここで、本来は存在しない節で、S式の中にはある筈の節を'○'で示すと、図１．bのようになる。

(2) (2)の操作の結果は図１．cのようになる。分岐点は(a)のみである。((a))の入出力は(a)と同じ型なので、それへの辺が除去される。(a b)については、類推によりその出力((a) (b))が求められ、埋められる。

(3) ((a) b)の入出力関係は(a b)と同じ型なので除去される。(図１．d参照)

(4) (a b c d)が類推により埋められる。(図１．e参照)

(5) (a b (c) d)の入出力関係は(a b c d)の入出力関係と同じ型なので、除去される。(図１．f参照)

これらの操作により、次の典型的な例の集合が得られる。こうして得られた例の集合へは、アルゴリズムを適用することができる。

{( )　　→ ( )；
(a)　　→ ((a))；
(a b)　　→ ((a) (b))；
(a b c)　→ ((a) (b) (c))；
(a b c d) → ((a) (b) (c) (d))}

このように、ここで提案した手順を施すことで、条件に囚われずに例を与えることができるようになる。

6．おわりに

本報告は、主に二つの内容を含んでいる。一つは、Summersのアルゴリズムを自然に発展させたこと。もう一つは、この種のアルゴリズムに与えるべき典型的な例の集合について考察し、任意の例の集合をそれに変換する手続きを与えたことである。

高階の差分を用いたアルゴリズムは、何等探索を必要とせず、直線的に最終的な結果まで進むことができる。この特徴は、Summersのアルゴリズムが本来持っていた優れた特徴である。しかし、与えた入出力例から関数フラグメントへの変換、また、二つの関数フラグメントから、差分関数フラグメントを求める過程には、多少の選択の余地があり、現在はその中の一つに固定してある。したがって、これらの選択の違いが、結果に影響するかどうかの考察は、今後に残された課題である。

また、同じくSummersのアルゴリズムの特徴として、アドホックな方法を用いないという意味で一般性に優れているということがある。本手法はこの特徴も継承

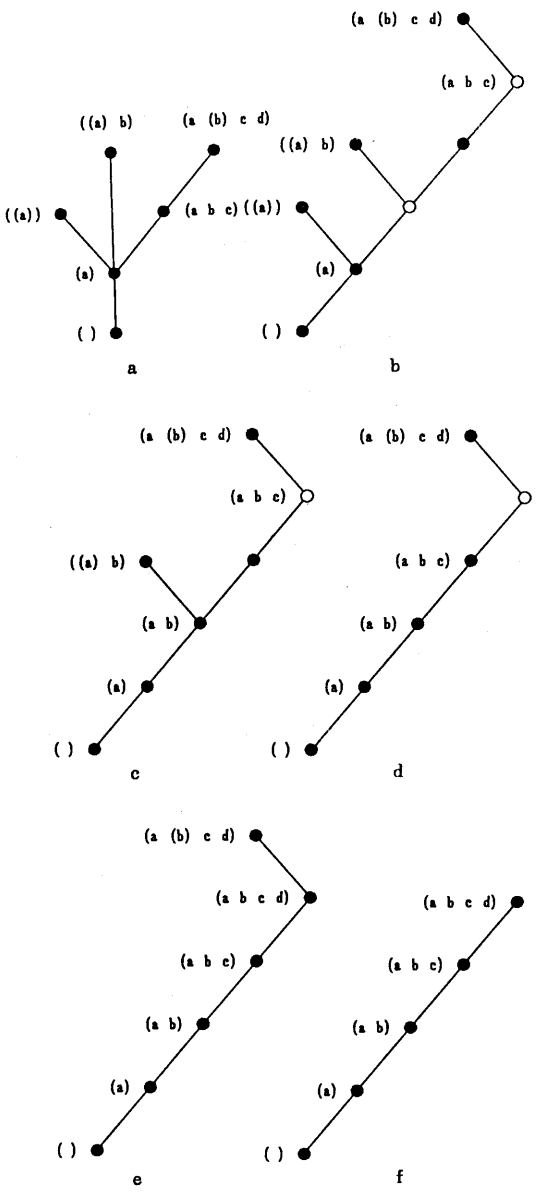

図１　非典型例から典型例への変換

している。これら二つの特徴を受け継いでいるという意味で、本手法は、Summersのアルゴリズムの自然な発展であると考えることができる。

第二点の典型例の変換について言えば、これは、アルゴリズムの能力を十分発揮するための前処理であると考えられる。これを付加することにより、全体として、柔軟性を持たせることができた。５．１で述べた典型例の条件と、アルゴリズムの動作との厳密な関係

については、現在検討中である。この前処理を含めたアルゴリズム全体の流れを図２に示す。

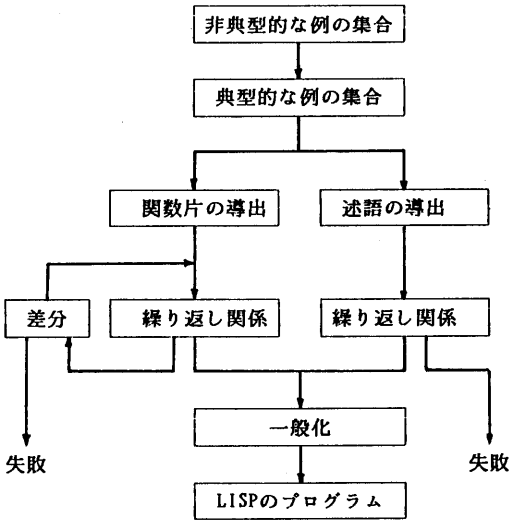

図２　アルゴリズムの流れ

このアルゴリズムは、LISPのＳ式上の関数に対するものであるが、これがこの領域で上手くいっているのは、次の二つの理由によると考えられる。一つは、Ｓ式を操作する有限で完備なオペレータ(car,cdr,cons)が存在するため、入力から出力を得るための式(関数フラグメント)をこれらを使って得ることができること。もう一つは、入力であるＳ式が束を構成しており、順序付けることができるということである。このような性質を満たせば、他の領域にも応用が可能であると考えられる。

参考文献

[1] P.D.Summers,A Methodology for LISP Program Construction from Examples, JACM 24:161－165,Jan.,1977

[2] D.Angluin,C.H.Smith,Inductive Inference: Theory and Methods,ACM Computing Surveys, Vol.15,No.3,237－269,1983

[3] R.Bird,PROGRAMS AND MACHINES An Introduction to the Theory of Computation,1976 (邦訳 土居範久訳,プログラム理論入門)

[4] S.C.Kleene,Introduction to Mathematics, Van Nostrand,Princeton,N.J.,1952