記　号　処　理 75－1
（1994．9．16）

フラグメントの部分間差分を用いた例からのLISP関数の自動合成

山口 文彦　　中西 正和
慶應義塾大学理工学研究科計算機科学科

入出力例からそのような入力-出力の関係を満たすような関数を推定する問題は，プログラム自動合成の基本的な問題である．Summersはこの問題について一般的な手法を提案したが，彼の手法では複数の例の間の関係を汎化して帰納的な関数を合成するために，複数の例をある順序に従がって与える必要があった．本稿ではフラグメントの構造中に繰り返し的な関係を見出す方法について述べ，例を与える順序への依存が少ないような関数の合成手法について考える．

Synthesis of LISP Functions from Examples Using Differences between Parts of a Fragment

Fumihiko YAMAGUCHI and Masakazu NAKANISHI
Department of Computer Science, Keio University

It is a general problem of program synthesis to infer functions from examples of the arguments and the return value. Summers gave a general method on this problem in which a sequence of examples, whose order has to be defined by the user, are used to find the recurence relations. This paper proposes a method to find such relations from the parts of a fragment and to construct the functions from order-free examples. The experimental synthesiser is written in Lisp.

1 はじめに

SummersはS式をS式に変換するLISPのプログラムを，その入出力例から合成する一般的な手法を提案した[3][4]．そこでは，複数の例の間の構造的な差異を発見することで，特殊なヒューリスティックを含まないという意味で一般的な関数の合成を可能にしている．また彼の手法はその手続き中に探索をほとんど含まないという特徴を持つ．

しかし，Summersの手法では例の間の差異から帰納的関係を仮定するので，適当な順序に従って入出力例を与える必要がある．入出力例に対するこの条件は，目的とする関数に対する知識を仮定したもので，関数の自動合成という観点からは好ましくない．

本稿では構造的な差異を例と例からではなく一つの例の部分と部分から発見することで，与える例が必要とする条件を緩和し，関数の自動合成システムとしてより扱い易いものにすることを考える．

2 対象領域

本稿ではLISPのS式をS式に変換する関数を合成することを目的とし，基本関数として car, cdr, $cons$ を用い，定数として nil を用いる．また関数の返す値は引数であるS式の構造にのみ依存し，引数に含まれるアトムの性質には依存しないものとする．

基本関数のみを合成して得られる関数を**フラグメント**と呼び，与えられた入出力例の入力と出力の間の関係をフラグメントを用いて表したものを**入出力フラグメント**と呼ぶ(図2.1参照)．

入出力例:
$(A\ B\ C) \rightarrow ((A)(B)(C))$

入出力フラグメント:
$\lambda x.(cons\,(cons\,(car\ \ x)nil)$
$\quad (cons\,(cons\,(car\,(cdr\ x))nil)$
$\quad\quad (cons\,(cons\,(car\,(cdr\,(cdr\ x)))nil)$
$\quad\quad\quad nil)))$

図 2.1 入出力例と入出力フラグメント

本稿では単数または複数の入出力フラグメントから帰納関数を推定する問題を扱う．

3 フラグメントの部分間差分

関数が帰納的に定義されているとき，その条件文(および述語)を評価すると(基本)関数が繰り返して適用された形が得られる．例えば関数 f が以下のように定義されているとき，

$$(f\ x) = \begin{cases} (g\ x) & if (p\ x) \\ (h\ (f\ (d\ x))\ x) & otherwise \end{cases}$$

ある入力 a に対する出力 $(f\ a)$ は，例えば以下のようになる．

$$(f\ a) = (h\ \ (h\ \ \ldots (h\ \ (g\ (d\ (d\ \ldots\ (d\ a))))\ (d\ \ldots\ (d\ a)))\ \ldots\ (d\ \ a))\ a)$$

本稿では入出力例 $a \rightarrow (f\ a)$ から作られた入出力フラグメントがこのような形をしているとき，h, g, d であるようなフラグメントおよび適当な述語 p を発見して関数の定義を推測することを考える．

このような繰り返しの形を発見するための主要なアイデアは以下の通りである．$(f\ x) = (h\ (f\ (d\ x))\ x)$ であるとすると，入出力例 $a \rightarrow (f\ a)$ から作られた入出力フラグメントは $(f\ (d\ x))$ を部分として含んでいると考えられる．部分の中から全体とほとんど同じ形を見つけるために単一化のアルゴリズムを用いる．

関数やフラグメント中の変数を別のフラグメントに置き換える操作を代入と呼び，記号列 $\boldsymbol{F}$ に代入 θ を適用した結果得られる記号列を $\boldsymbol{F\theta}$ で表す．また，複数の関数やフラグメントを同じ記号列に変える代入を**単一化代入**と呼ぶ．さらに，代入 θ が $\boldsymbol{F}$ と $\boldsymbol{G}$ の**最汎単一化代入**であるとは，任意の単一化代入 σ に対し

$$\boldsymbol{F}\sigma = \boldsymbol{G}\sigma = (\boldsymbol{F}\theta)\rho = (\boldsymbol{G}\theta)\rho$$

となる代入 ρ が存在することをいう．最汎単一化代入を求める，もしくは存在しないことを知るアルゴリズムは存在することが知られている[1]．

$(f\ x)$ と $(f\ (d\ x))$ との単一化代入を重複のないように変数を呼び換えてから求めると $[x' \leftarrow (d\ x)]$ が得られ d を抽出することができる．単一化のアルゴリズムが成功するような部分が見つかったとき，全体に於けるその部分の位置から h を抽出することができる．

このような d に相当するフラグメントを**引渡しフラグメント**, h に相当するフラグメントを**差分フラグメント**と呼び, これらを求めることを**差分をとる**という. また $(f\ x) = (f\ (d\ x))$ となるとき, 差分フラグメント h は恒等関数であるという.

実際には繰り返しが有限回で終るので, 繰り返すと仮定される部分を適当な記号に置き換える必要がある (また, 有限回で終らなければ単一化アルゴリズムが終了しない). したがって実際には, 局所的に h が繰り返されていることを発見し, その繰り返しが継続することを推測するのである.

フラグメント F とその部分フラグメント G との差分をとる手順は以下の通りである.

1. F と G の変数を重複のないように置き換える. このように置き換えたものを, それぞれ F', G' とし、F' の変数をそれぞれ $x_{F1}, x_{F2}, \ldots, x_{Fn}$, G' の変数をそれぞれ $x_{G1}, x_{G2}, \ldots, x_{Gn}$ とする.

2. F' と G' の最汎単一化代入子が存在するときこれを θ とし, 差分フラグメントを恒等関数であるとして 6へ行く. このような最汎単一化代入子が存在しないとき, 次へ行く.

3. F 中の G を `Self` に置き換え F' と同じように変数を呼び換えたものを F'' とする.

4. G 中で F 中の G の出現する位置と同じ位置にあるものを `Self` に置き換える. ここで G 中に F 中の G の出現する位置と同じ位置にあるものが存在しないとき, 置き換えは失敗してこのアルゴリズムも失敗して終了する. 置き換えが成功するとき, 置き換えて G' と同じように変数を呼び換えたものを G'' とする.

5. F'' と G'' の最汎単一化代入子が存在するときこれを θ とし, 差分フラグメントを λ`Self`.F'' として次へ行く. このような最汎単一化代入子が存在しないとき, このアルゴリズムは失敗して終了する.

6. θ 中に x_{Fi} を $(D_i\ x_{G1}\ x_{G2}\ \ldots\ x_{Gn})$ の形のフラグメントに置き換える代入しか存在しないときは, D_i が引渡しフラグメントである.

以上のようにして F とその部分である G との差分をとることができるとき, F と G との関係の繰り返しによって F を定義できると推測する.

例として図2.1の入出力フラグメントから帰納的な関係を推測する過程を以下に示す.

$$
\begin{aligned}
&F = \\
&\quad \lambda x.(cons\,(cons\,(car\ \ x)\,nil) \\
&\qquad\qquad (cons\,(cons\,(car\,(cdr\ x))\,nil) \\
&\qquad\qquad\quad (cons\,(cons\,(car\,(cdr\,(cdr\ x)))\,nil) \\
&\qquad\qquad\qquad nil)))
\end{aligned}
$$

このとき G を F のもっとも外側の関数 $cons$ の第1引数である $(cons\ (car\ x)\ nil)$ とすると,

$$
\begin{aligned}
&F'' = \\
&\quad \lambda x_F.(cons\ \texttt{Self} \\
&\qquad\qquad (cons\,(cons\,(car\,(cdr\ x_F))\,nil) \\
&\qquad\qquad\quad (cons\,(cons\,(car\,(cdr\,(cdr\ x_F)))\,nil) \\
&\qquad\qquad\qquad nil))) \\
&G'' = \lambda x_G.(cons\ \texttt{Self}\ nil)
\end{aligned}
$$

となり, F'' と G'' との単一化に失敗する. 次に G を F のもっとも外側の関数 $cons$ の第2引数とすると,

$$
\begin{aligned}
F'' &= \lambda x_F.(cons\ (cons\ (car\ x_F)\ nil)\ \texttt{Self}) \\
G'' &= \lambda x_G.(cons\ (cons\ (car\ (cdr\ x_G))\ nil)\ \texttt{Self})
\end{aligned}
$$

となり, このとき単一化が成功して

$$\theta = [x_F \leftarrow (cdr\ x_G)]$$

である. したがって,

$$
\begin{aligned}
&\text{差分フラグメント} = \\
&\quad \lambda\texttt{Self}.\lambda x_F.(cons\ (cons\ (car\ x_F)\ nil)\ \texttt{Self}) \\
&\text{引渡しフラグメント} = cdr
\end{aligned}
$$

となる. このとき, F の帰納的な定義を以下のように推測する.

$$F \simeq \lambda x.(cons(cons(car\ x)\ nil)(F\,(cdr\ x)))$$

4 矛盾点の修正

合成された関数に入力例を適用して入出力フラグメントと異なるものが得られる場合, 関数に入力例を順次適用していくことで, どのような入力に対して関数の出力が入出力フラグメントと異なり, そのときどのような出力をすべきかを知ることができる. このような矛盾する入力を識別する述語を合成できれば, 適当な条件文を用いて関数を修正することができる[5].

以下に矛盾点を発見するアルゴリズムについて説明するが，ここで $((\lambda x.f)\ a)$ という記号列と関数$\lambda x.f$にaを適用して得られる記号列を区別するために，関数を簡約した形は $f[x \leftarrow a]$ と代入を明記する．

ここで求めるものは，矛盾が生じる入力例と矛盾が生じない入力例，および矛盾が生じたときに出力となるべきものである．このアルゴリズムで考えられる入力例は元の入力例に何回か引渡しフラグメントを適用したものである．そこで引渡しフラグメントを何回か合成したものを**適用フラグメント**と呼び，矛盾が生じるときの入力例を表す適用フラグメントを**負事例適用フラグメント**，矛盾が生じないときの入力例を表す適用フラグメントを**正事例適用フラグメント**と呼ぶ．また矛盾が生じたとき出力すべきものは与えられた入出力フラグメントの一部で表され，これを**終端フラグメント**と呼ぶ．

合成された関数を$\lambda x.f$，入力例をa，入出力フラグメントを$\lambda x.\boldsymbol{F}$とし，適用フラグメントを表すために$\boldsymbol{C}$を用いて，最初$\boldsymbol{C}$はxであるとする．

1. $f[x \leftarrow \boldsymbol{C}]$ と $\boldsymbol{F}$を比較する．もっとも外側の関数名から(記号列として)比較し，等しいときはその各引数を比較する．比較の結果でそれぞれ以下のように分かれる．

2. すべて等しければ無矛盾という結果でこのアルゴリズムを終了する．

3. 両者がフラグメントであれば，$\lambda x.\boldsymbol{C}$は負事例適用フラグメントであり，$\lambda x.\boldsymbol{F}$は終端フラグメントである．

4. yを引渡しフラグメント$\lambda y.d'$で抽象されている変数であるとしたとき，合成された関数の適用 $(g\ (d'[y \leftarrow x]))$ とフラグメント$\boldsymbol{G}$との比較で等しくない場合は，$\lambda x.\boldsymbol{C}$は正事例適用フラグメントであり，gの定義を新たにfとし，$\boldsymbol{G}$を新たに$\boldsymbol{F}$とし，$d'[y \leftarrow x]$を新たに$\boldsymbol{C}$として1へいく．

5. 条件分岐文とフラグメントを比較して等しくない場合は，$\boldsymbol{C}[x \leftarrow a]$が満たす条件に対応する関数を新たに$f$として1へいく．

このアルゴリズムが終了するとき，無矛盾であるか，そうでなければ正事例適用フラグメント(の集合)，負事例適用フラグメント，終端フラグメントが得られる．

例として前節で帰納的に定義された関数の矛盾点を発見する過程を以下に示す．

入力例:$(A\ B\ C)$.

合成された関数:

$\lambda x.f = \lambda x.(cons(cons(car\ x)\ nil)(f\ (cdr\ x)))$.

入出力フラグメント:

$$\lambda x.(cons(cons(car\ \ x)nil)\ (cons(cons(car(cdr\ x))nil)\ (cons(cons(car(cdr(cdr\ x)))nil)\ nil))).$$

まず，$\boldsymbol{C} = x$であるから，

$$(cons(cons(car\ x)\ nil)(f\ (cdr\ x)))$$

と

$$(cons(cons(car\ \ x)nil)\ (cons(cons(car(cdr\ x))nil)\ (cons(cons(car(cdr(cdr\ x)))nil)\ nil)))$$

とを比較する．よって記号列が等しくならない部分，すなわち

$$(f\ (cdr\ x)) \neq\ (cons(cons(car(cdr\ x))nil)\ (cons(cons(car(cdr(cdr\ x)))nil)\ nil))$$

が分かる．したがって，$g \equiv f$, $d' \equiv cdr$とすると4に相当するので，

$$\lambda x.f = \lambda x.(cons(cons(car\ x)\ nil)(f\ (cdr\ x)))$$
$$\boldsymbol{F} = (cons(cons(car(cdr\ x))nil)\ (cons(cons(car(cdr(cdr\ x)))nil)\ nil))$$
$$\boldsymbol{C} = (cdr\ x)$$

とする．ここで$\lambda x.x$は正事例適用フラグメントである．

再び，$f[x \leftarrow \boldsymbol{C}]$ と $\boldsymbol{F}$とを比較する．このとき，

$$(cons(cons(car\ (cdr\ x))\ nil)(f\ (cdr\ (cdr\ x))))$$

と

$$(cons(cons(car(cdr\ x))nil)\ (cons(cons(car(cdr(cdr\ x)))nil)\ nil))$$

との比較であり，

$$(f\ (cdr\ (cdr\ x))) \neq \begin{array}{l}(cons(cons\ (car\ (cdr\ (cdr\ x)))\ nil)\\ \qquad nil)\end{array}$$

となる．したがって，$g \equiv f$, $\lambda y.d' \equiv \lambda y.(cdr\ (cdr\ y))$ とすると 4 に相当するので，

$$\lambda x.f = \lambda x.(cons(cons(car\ x)\ nil)(f\ (cdr\ x)))$$
$$\boldsymbol{F} = (cons\ (cons\ (car\ (cdr\ (cdr\ x)))\ nil)\ nil)$$
$$\boldsymbol{C} = (cdr\ (cdr\ x))$$

とする．ここで$\lambda x.(cdr\ x)$ は正事例適用フラグメントである．

さらに，$f[x \leftarrow \boldsymbol{C}]$ と $\boldsymbol{F}$ とを比較する．このとき，

$$(cons(cons(car\ (cdr\ (cdr\ x)))\ nil)(f\ (cdr\ (cdr\ (cdr\ x)))))$$

と

$$(cons\ (cons\ (car\ (cdr\ (cdr\ x)))\ nil)\ nil)$$

との比較であり，

$$(f\ (cdr\ (cdr\ (cdr\ x)))) \neq nil$$

である．したがって，

$$\lambda x.f = \lambda x.(cons(cons(car\ x)\ nil)(f\ (cdr\ x)))$$
$$\boldsymbol{F} = nil$$
$$\boldsymbol{C} = (cdr\ (cdr\ (cdr\ x)))$$

とし，$\lambda x.(cdr\ (cdr\ x))$ は正事例適用フラグメントである．

次に f と $\boldsymbol{F}$ の比較をすると，fのもっとも外側の関数 $cons$ と $\boldsymbol{F}$である nilが両方ともフラグメントであって等しくないので 3 に相当する．したがって，このとき$\lambda x.(cdr\ (cdr\ (cdr\ x)))$ が負事例適用フラグメント，$\lambda x.nil$が終端フラグメントである．

すなわち，入出力例 $(A\ B\ C) \rightarrow ((A)(B)(C))$ を満たす関数を $f = \lambda x.(cons\ (cons\ (car\ x)\ nil)(f\ (cdr\ x)))$ であると推定すると，この入力例から矛盾点を発見して $(f\ (cdr\ (cdr\ (cdr\ '(A\ B\ C))))) = nil$ となるべきであることがわかるのである．

入力例に正事例適用フラグメントを施して得られるものと，負事例適用フラグメントを適用して得られるものとを区別する述語を発見し，条件文を用いて関数を修正する．もちろん，このアルゴリズムの結果が無矛盾である場合は，関数は修正されない．

$$\begin{array}{l}(atom\ '(A\ B\ C)) \neq T\\ (atom(cdr\ '(A\ B\ C))) \neq T\\ (atom(cdr(cdr\ '(A\ B\ C)))) \neq T\\ (atom(cdr(cdr(cdr\ '(A\ B\ C))))) = T\\ \Rightarrow\\ \quad f =\\ \quad \lambda x.(cond((atom\ x)\ nil)\\ \qquad\qquad (t\ (cons(cons(car\ x)\ nil)(f\ (cdr\ x))))\\ \qquad\qquad)\end{array}$$

図 4.1　関数の修正

こうして修正された関数と入力例が入出力フラグメントに対して矛盾しないので，入出力例を満たす関数が合成される．

参考文献

1. Robinson, J. A. : *A Machine-Oriented Logic Based on the Resolution Principle*, JACM, 12(1), 1965, pp. 23–41.

2. Steven HARDY: *Synthesis of LISP Functions from Examples*, Proceedings of IJCAI 1975, pp. 240–245.

3. Phillip D. SUMMERS: *A Methodology for LISP Program Construction from Examples*, JACM, Vol. 24, No. 1, pp. 161–175, January 1977.

4. ANGLUIN, D., AND SMITH, C. H.: *Inductive Inference: Theory and Methods*, ACM Comput. Surv., Vol. 15, No. 3, pp. 237–269, Sep., 1983.

5. E. Y. SHAPIRO 著 有川 節夫 訳: 知識の帰納的推論, 共立出版, 知識情報処理シリーズ 第 3 巻, 1986.

6. Nobuhiro INUZUKA, Kenichi TAKAHASHI AND Naohiro ISHII: *Representative sample of LISP program inference from examples*, International Journal of Systems Science, Vol. 23, No. 8, pp. 1321–1334, 1992.