人　工　知　能　99−22
（１９９５．　３．　７）

論理プログラムの自動検索・統合・発見

有馬 淳

株）富士通研究所

[概要] プログラム資産の再利用を念頭においた論理プログラムの自動合成、作成支援を目指す研究について報告する。本稿では、その課題達成のために行なった基本的考察、動作原理、理論的性質、および論理プログラムによるプロトタイプシステムで行なった評価実験についてその結果を報告する。この方式は、与える例の数が極めて少なくて済むこと、既存プログラムの再利用を基本にしていることなどこれまでにない利点を数多く持っている。例えば、評価実験では reverse プログラムの正例３個から新述語 concat 合成を含むプログラムを生成することに成功している。

Automatic Retrieval, Integration and Invention of Logic Programs

Jun ARIMA

FUJITSU Labs. LTD.

Abstract. This paper describes results of study aiming at automatic programming, from incomplete specification, with reusing existing programs and with inventing new programs. Results involve fundamental considerations to achieve the aims, principles, properties of a proposed method and experiments by a prototype system. This method provides many exceptional features: the number of examples needed is quite few, the method is based on reuse of existing programs, and etc. For instance, in the experiments, the prototype system succeeds in producing `reverse` program with inventing `concat` program, just from 3 positive examples for `reverse`.

1 導入

1.1 課題

プログラムを作成する人なら誰でも、「この機能を持つプログラムなら誰かが既に作っているだろう。それを利用できないか？」とか「昔（今しがた？）作ったあのプログラムの名前は何だったろう？」とか考えたことがあるだろう。本論文で扱う一つの主題はそれである。そしてもう一つの主題は、昔からある計算機科学者の夢 – 簡単な仕様を入力する。しばらくする。お望みのプログラムがプリントアウトされる。 – である。以上の困難な問題に対して、ここでは極めて簡単な技法を提案し、その可能性と限界を考えてみたい。

本研究の目的は大きく以下の２つで表される。

- 一般に不特定のプログラマーによって作られたプログラム資産を有効に活用すること。
- 不完全な、しかし与えやすい仕様から自動的にプログラムを生成できること。

ここで不特定のプログラマによるプログラム資産とはより具体的には、そのプログラムに関するさまざまな性質 – プログラムの機能、呼出名、引数の個数、そして使用法など – が不明の実行可能なコードの非構造的な集合を意味している。このような集合を、通常利用は困難という意味で以下では迷宮プログラム DB($\mathcal{LPD}$)と呼ぶことにする。このような資産の再利用法はこれまで考察されていないようである。しかしこの設定は不自然なものだろうか？

本論文は上述の研究の最初の報告として、プログラム検索、および、線形再帰関数で表現されるクラスのプログラム合成に関して行なった考察と提案、およびそのアイデアに基づくプロトタイプの結果について述べる。本章では、基本的なアイデアを示す。第２章ではより詳細な定義、議論を行ない、理論的な諸性質を調べる。第３章ではこれらの結果に基づくプロトタイプシステム ARDEX による実験結果を示す。

1.2 背景・議論・二つの鍵

検索: 機械によって必要なプログラムを検索する方法を大きく非形式的なものと形式的なものに分ける。前者の例としては、さまざまな言語のライブラリやオブジェクト指向言語において見られるようにプログラムを分類階層に構成したり、何らかのインデックスをつけ利用者が自身であるいはある部分は機械の力を借り検索、再利用する方法 [5] があげられる。これらの方法はすべて利用者の暗黙知識に依存しておりこの意味で非形式的である。非形式的手法は、領域依存度く、一般に個人による利用や限られた分野の使用に限られ、不特定のプログラム資産の利用には向かない。また、同時に機械による高度な支援を困難にしている。

不特定プログラムの再利用を行なう場合、形式的であることが必要である。目的プログラムの形式的仕様が与えられれば、検索に限らず、プログラミング作業の大部分を自動化することが原理的に可能で、しかも生産されるプログラムの仕様に対する正しさ（正当性）などさまざまな性質を保証することもできる [9]。しかしながら形式的な仕様に基づく検索には以下の三つの問題点がある。まず第一に検索の自動化には定理証明を使う方法がある [6] が、残念ながら一般に実際的な時間での完全な検索は難しく、扱えるプログラムのサイズは著しく制限される。第二は認知的距離の問題である。形式的な仕様は一般に人間にとって理解しにくい。このため与えにくい。（三番目の欠点は後述する。）

我々が求める方法は、形式的でありながら認知的な距離の短いもの、すなわち、人間が与えやすく理解しやすいものでなければならない。しかも機械による仕様としての理解、比較、処理が容易なものでなければならない。このことから、我々は次の一見馬鹿げているほど簡単な方法こそ最良の解に近いのではないかと信じるに至った。本研究ではプログラム検索の手段の原理として次のものを採用する。

外延的検索原理 (*extensional retrieval principle*): プログラム DB 中の各プログラムに適当な入力データを与え、意図する値を意図する計算資源で出力するプログラムを採れ。

すなわち、目的プログラムの具体的入出力の対を仕様とし、検索の手段としても利用する。この方法には明らかな欠点がいくつかある。まず、入出力の関係だけでなく副作用が重要なプログラムはこの検索方法に本

質的に向かない。また、仕様の長さは有限なので、一般に完全な仕様を与えることができない。

しかしながらこの方法は我々が掲げた問題に対する検索を考えた時、それらの欠点を償える利点がある。第一に具体的入出力例は仕様として人間がもっとも理解しやすく与えやすいものの一つであること。第二に、与えられた入力データを使って直接プログラムを実行させることにより、そのプログラムが現実的な性能を持つなら現実的な時間でその部分的仕様を満たす正当性の保証ができること。第三にプログラムの計算性能（時間、記憶域資源の使用量などからみた実性能）を選択の基準にすることができること。第四に、検索対象は有限の中での選択であり、現実上、完全な仕様は不要と考えられること。そして最後に、形式的であることにより、例えば、補助プログラムの検索やそれらの統合など、計算機による高度な機能の自動化が可能になることなどである。最後の点をプログラムの統合・発見の原理から議論する。

統合・発見: 既存プログラムや、後に述べる発見によって得た新しいプログラムを組合せ、目的のプログラムを合成することを統合と呼ぶ。統合の自動化を考えてみよう。

形式的仕様とプログラムの間に意味的な等価性が成り立つような完全な仕様を考えた場合、バグのないプログラムを書くこととバグのない形式的仕様を書くことは意味的に全く同様の難しさを持つことは明らかである。従って、現実的には部分的仕様で我々はなんとかやってゆく方法を見つけねばならない。我々はここでも外延的アプローチを採ってみよう。

具体的入出力の対を仕様と考え、プログラムをその入出力例から生成する方法自体は特に関数型言語の自動プログラミングの分野 (IAP と書く) [3, 13] で行なわれ、また論理型プログラムの学習 (ILP) の分野においても関連する多くの研究がある。しかし、いづれの場合も以下に述べるように不満な点を残している。

第一に目的のプログラムを導くのに必要な例の数は小さくなければならない。

例えば、リストを反転させるプログラムを得るのにも、ILP 研究の最良のものでも数十から数百の例が必要になる。これは例示の仕方が一般に無作為に与えられることが一つの原因と考えられる。自動プログラミングの観点からは、少なくとも例の入力に必要な手間が利用者が（手で）作りあげる手間に勝ってはならないことになり、現在の研究は残念ながらこの要請に対し満たしているものはない。

第二に、目的プログラム以外の情報を利用者に求めるべきではない [4]。

目的プログラムに必要な補助プログラムすべてが既にプログラム DB 中にあるとは限らない。この場合、新たに必要な補助プログラムを作成することを補助プログラムの発見と呼ぶ。目的プログラムついての質問とは違い、その補助プログラムに関して情報を与えることは教師や神でない利用者にとっては事実上期待できない。IAP および ILP 研究を通じて、発見機能を持たせることをこの意味で真に可能にしているものは残念ながらまだ極めてわずかである。

第三に、目的プログラムのデータの種類を限定すべきでない。

一般的なプログラムの作成支援ができるために、データの型、リスト、項、数値に依存しない一般的な自動プログラミング技法であることが望ましい。リストや項の処理を行なう目的プログラムを対象とする技法は個別に提案されているものの、IAP, ILP 研究を通じて、統一的に数値処理まで扱える技法はほとんどない。

GAP [3] は上記要請の 2,3) を, CIGOL [10] は 1,2) を、CLINT[2] と FOIL[11] は 1,2,3) を、CHAMP [7] は 1,3)、FILP [1] は 2) のそれぞれの要請を満足していない。

これらの要請に答えるためには、例だけにプログラムの情報を頼ることができない。我々は例の情報の他にプログラムの構造に関する情報をプログラムの再帰スキーマとして、また例示の順序を再帰スキーマによる依存順（計算順）に与えられることを仮定すると、極めて有利な幾つかの利点が生じることに気がついた。両方の仮定が必要である。IAP 研究のほとんどは依存順の例のみ使い (GAP [3] は例外的にスキーマを使うが後者は使わない)、ILP 研究は両方を使わない (CLINT[2] は例外的にスキーマを使うが、後者を使わない)。CLINT は言語スキーマを使ったが、我々は再帰スキーマの 計算論的な面を使う。つまり、我々の 2 番めの鍵は 再帰的定義、例の依存順提示である。 以下にこの原理が如何に働きどのような利点が生じるかを説明する。

再帰スキーマとはある値が、前に計算された自分自身の値に基づいて決まる再帰的定義の型を表したものである（図 1.2参照）。

$$f(X) \Leftarrow \text{if } e(X) \text{ then } b(X) \text{ else } a(X, f(c(X)))$$

但し、e, i は述語、f, b, a, c は関数。

図 1.2.a: （線形）再帰スキーマの例

このスキーマの論理型プログラム表現は図 1.2の f における出力変数を Y, PY と置くことによって以下のように表現される。ここでいづれのスキーマにおいても、X, PX は有限の長さの変数列を表す。

```
F(Y,X) :- E(X),B(Y,X).
F(Y,X) :- C(PX,X), F(PY,PX), A(Y,PY,X).
```

図 1.2.b: 論理型プログラムでの線形再帰スキーマの例

図 1.2.a に対応して図 1.2.b の論理型プログラム表現は、述語 E を除いてすべての第一引数は「出力」としての性格を（以下を通じ）与えるものとする。入力 X に対し Y の値が一意である必要が無いという点で関数的ではないが、構成された値が出力変数によって取り出せなければならない。この計算論的な制限が「計算できるプログラム」の効率的な探査を可能にする。

さて、このような再帰的定義によるある関数の値の決まり方に従って順序を考えることができる。この順序をここでは依存順と呼ぶことにする。先のスキーマとその依存順に基づいた例が与えられると仮定してみよう。例えば、階乗を計算するプログラム fact の自動合成を考える。fact という述語の入出力例として、ある適当な例 '$fact(24,4)$' と、入力 '4' においてその「一つ前」の入力 '3' を持つ例を与える。

$$\{fact(24,4),\ fact(6,3)\}$$

上記スキーマの再帰節（第2節）の F および Y, PY, X, PX のこの例に基づく特殊化（$F \leftarrow$ fact, $(Y,X) \leftarrow (24,4)$, $(PY, PXs) \leftarrow (6,3)$）は前者の例を導く具体的な節を表すことになる。つまり、

```
fact(24,4) :- C(3,4), fact(6,3), A(24,6,4).
```

を得る。この時、我々は既に $fact$ の構成に必要な補助プログラムの外延的仕様を手にしている！ 一つは 4 を入力し 3 を出力するプログラム (C) であり、もう一つは 6 と 4 から 24 を出すプログラム (A) である。ここで上述の外延的検索原理に基づいた外延探査を行なえば、$\mathcal{LPD}$ に 1 を引くプログラムと乗算のプログラムがあれば容易に引きだせるであろう。これにより精密化した結果、

```
fact(Y,X) :- PX = X - 1, fact(PY,PX), Y = PY * X.
```

が得られる。これは確かに我々が求める階乗計算をする論理プログラムの一部である。

このように、帰納スキーマと依存整列した例の入力方法は、まず、例の数を大幅に少なくすることに貢献する。この説明で明らかなとおり、各補助プログラムに既存プログラムが使える場合、スキーマ中に現れる目的プログラム名が R 個とすると高だか例の数も R 個で済むことになる。今の場合、fact が現れるのは再帰節に 2 個であり、停止節の 1 個をあわせて 3 例で － 停止節の例が再帰節の例に含まれるよう与れば最低 2 例で － fact プログラムの合成が可能になった。次に、目的プログラムに関する例（仕様）を補助プログラムに関する例（仕様）に容易に変換できる。これは、非形式的検索方法ではできない大きな利点になる。すなわち、階乗計算プログラムが $\mathcal{LPD}$ 中に存在しなくても、それに必要な補助プログラムが何であるかを知ることなしに自動的に探査し、しかも合成することが可能になる。第三に、理論的にも実際的にも計算効率の向上をはかることができる。入出力関係を持つ再帰的スキーマは、あらかじめ「計算」できるプログラムを探査することになり、探査空間を大幅に減少させる。第四に、プログラムの扱うデータの種類（リスト、項、数値）に依存しないガイド法である。最後に、暗黙的ヒュウリスティックのかわりに、獲得できるプログラムのクラスを明示的に与えられるなどといった利点がある。

プログラムのガイドとなるスキーマが詳細であれば、あらかじめプログラムの形を予測することの困難さや、スキーマの数の増大にともなう選択の問題が生じる。しかし、実際上は既存研究で使われる例題のほとんどがここで示す一種類の線形スキーマをあらかじめ与えておくだけで目的プログラムの獲得に成功することが実験により確かめられた[1]。

[1] この研究では自動合成の計算上の問題から、1 関数変数は 1 関数を表すことに対応し、一般の関数合成を許していないが、そのような制約がなければ、図 1.2 のスキーマは繰り返し型のスキーマのクラスに対応するほど十分に広いことが知られている。

2 構成

2.1 準備

本論文では 1 出力、$n-1$ 入力 ($n \geq 1$) の場合を定義する。複数出力の場合は類推によって容易に拡張できよう。また、(複数の) 入力はリストで表現し、出力を y、入力をリスト x_s とする時、出力をリストの head に、入力をその tail に持つリスト $[y|x_s]$ によって入出力組 (IO-list) を表すものとする[2]。また、通常のアトムの表記法に加え、n-引数述語 p のアトムを p^t で表す。ここで t は長さ n のリストである $(p(x_1,\cdots,x_n) = p\hat{}[x_1,\cdots,x_n])$。

定義 1 交換関数、除去関数、変換関数: 関数 < . > が交換関数であるとは、< . > がリスト l を入力とし、l の要素のある並べ替えを行った結果得られたリストを出力する関数であることをいう。

関数 $w_k(.)$ が除去関数であるとはリスト l ($|\,l\,| \geq k$) の入力に対し、$k+1$ 番め以降の要素を任意に除去したリストを出力する関数 (順序は不変) であることを言う。本論文では $k \geq 1$ の除去関数のみに興味がある。このような場合、単に w と書く。

関数 $a(.)$ が ADB 上での変換関数であるとは、リスト l ($l = [e_1,\cdots,e_n]$) の入力に対し、l の各要素 e_i にそれぞれ任意に ADB の引数関数 a_i を適用したリスト ($[a_1(e_1),\cdots,a_n(e_n)]$ ここで $a_i \in ADB$) を出力とする関数である。但し、本論文では a_1 は恒等関数とする。

除去関数 w、変換関数 a によってリストの第一要素 (出力用変数) の値は変わらないことに注意せよ。

本論文で「入出力」という言葉は Prolog プログラムのように入力、出力の区別がない場合や入力がない場合も含め、また、一般に関数的でない、すなわちある入力に対して出力が複数ある場合も含めて使うため、通常の「入出力」を拡張した定義を与える必要がある。

定義 2 正負入出力例: < . > を交換関数とする。この時、[y|xs] は < . > に関するプログラム p の (正) 入出力例 *((positive) IO-pair)* であるとは、

$$\exists \mathtt{Y}.\ (\vdash \mathtt{p\hat{}} < [\mathtt{Y}|\mathtt{xs}] >, \mathtt{Y} == \mathtt{y}),$$

が成り立つことである。また、これが成り立たない時、< [y|xs] > は負入出力例 *(negative IO-pair)* という。ここで p^ < [Y|xs] >, Y == y は論理型プログラムのゴール列として読み、xs に具体化された時、p の他の変数が Y に具体化され、かつそれが文字どうり y に等しいことを表す。また、交換関数 < . > が恒等関数の時 *(forall list l, $< l >= l$)*、単に [y|xs] はプログラム p の (正/負) 入出力例という。正の入出力例はそのリストの前に + 、負は − をつけて表す。

定義 3 論理型プログラムスキーマ、再帰、線形: $F\hat{}[Y|Xs], \beta_i$ を原始式を *range* とする変数もしくは原始式とする。この時、$F\hat{}[Y|Xs]: -\beta_1,\cdots,\beta_n.$ ($n \geq 0$) の形を持つ節を**論理型プログラムスキーマ片**、その有限の長さの並びを**論理型プログラムスキーマ** (以下では単にそれぞれ**プログラムスキーマ片**、**プログラムスキーマ**) と呼ぶ。また、すべての $F\hat{}[Y|Xs], \beta_i$ が具体化している時、そのプログラムスキーマをプログラムと呼ぶ。Y を出力変数、Xs を入力変数の組とするプログラムスキーマ片において、Y を引数に持たない (入力変数のみに依存する) 原始式 β_i は条件式と呼ぶ。スキーマにおいてある述語が $[Y, Z_1,\cdots,Z_j|Xs]$ の引数をとる時、$Y, Z_1,\cdots,Z_j$ を**必須引数** と呼ぶ。あるプログラムスキーマ S 中のすべての頭部が同じ述語 f である時、f を S の**定義述語**と呼ぶ。$\beta_1,\cdots,\beta_n$ (C の本体) 中に f が m 個 *($0 \leq m \leq n$)* 現れる時、C は m **の依存度を持つ**と言う。C が 0 の依存度を持つ時、**非再帰的** と言い、それ以外の時、**再帰的** と言う。C がちょうど依存度 1 を持つ時、C は**線形である** と言う。f を定義述語とするプログラムスキーマ S の依存度は S 中の C の持つ依存度の最大の値と定義する。依存度 *1* を持つプログラムスキーマ S は**線形である**という。また S において再帰的でない節は**停止節**と呼ぶ。

[2]従って、この論文では入出力組の第一要素のみが出力を表す。例えば、$[a, b|x_s]$ を入出力組とすると、出力が a、入力が $[b|x_s]$ であることを表す。

定義 4 依存した、(線形) 依存列: 述語 H が現れないあるプログラムの有限集合を $\mathcal{K}$、H の原始式を h, h_i、H のある原始式の集合 (空集合を含む) を $\mathcal{H}$ と書く。この時、$\mathcal{K}$ を背景とした時、H を定義述語とするプログラム P に関して、h が集合 $\mathcal{H}$ に依存している とは、ある 節 $C = (h : -\beta_1, \cdots, \beta_n.) \in P$ があって、$\mathcal{H} = \{h_i\theta \in \{\beta_1\theta, \cdots, \beta_n\theta\} \mid h_i \text{ is an atom of } H.\}$ かつ $\mathcal{K}, C, \mathcal{H} \vdash h$ であることを言う。また、H の原始式の並び $\mathcal{E}^* = \{h_n, \cdots, h_0\}$ $(n \geq 0)$ が、H を定義述語とするあるプログラム P に関する (線形) 依存列である *(linearly dependently ordered)* とは、$0 \leq i \leq n$ なる任意の i に関して、P に関して h_i が h_{i-1} に依存することをいう。

定義 5 外延仕様、整列外延仕様: $\mathcal{E}$ が プログラム $\mathcal{P}$ の正または負の入出力例の有限集合である時、$\mathcal{E}$ を $\mathcal{P}$ の 外延仕様 と言い、プログラム $\mathcal{P}$ は入出力仕様 $\mathcal{E}$ を満たすと言う。さらに、$\mathcal{E}$ が $\mathcal{P}$ の依存列である時、$\mathcal{E}$ を $\mathcal{P}$ の 整列外延仕様 と言う ($\mathcal{E}$ は $\mathcal{E}^*$ と表す)。

定義 6 スキーマ $\mathcal{S}$ の $< . >, w(.), a(.)$ による精密化とは、$\mathcal{S}$ のすべての原始式に対し次の変換を行なうことである。 $A\hat{}[Y_1, \cdots, Y_k|Xs] \rightarrow A'\hat{} < a(w_k([Y_1, \cdots, Y_k|Xs])) >$

2.2 検索

外延探査プログラム: Ariadne は、プログラム名、引数の順序が不明のプログラムの探査を目的とする。Ariadne は入力された入出力例に任意の交換関数 $< . >$ を順次適用し得られた入出力例に関して、$\mathcal{LPD}$ から有限の計算資源の下で満たすプログラムを順次取り出す。

2.3 統合・発見

スキーマ誘導合成プログラム: EXN はあらかじめ与えられた再帰的プログラムスキーマの集合 SDB が与えられると仮定する。整列外延仕様 $\mathcal{E}^*$ が入力されるとこのプログラムは $\mathcal{E}^*$ を満足するプログラムを合成することを目的とする[3]。本論文では、図2.3 に表す一般的な論理型線形再帰スキーマで得た結果を報告し、これをもとに今後の展開を考えたい。

```
ps0:  F^[Y|Xs] :- E^Xs,B^[Y|Xs].        | 但し、
ps1:  F^[Y|Xs] :-                       | C(PXs,Xs) =
                  C(PXs,Xs),            |     true     (if |PXs| = 0)
                  F^[PY|PXs],           |     C1^[PX1|Xs],...,Cn^[PXn|Xs]
                  A^[Y,PY|Xs].          |              (if PXs = [PX1,...,PXn])
```

図 2.3: 論理プログラムの2つの再帰スキーマ片の例

先の fact の例は、ps0 を特殊化した1停止節と ps1 を特殊化した帰納節からなり、しかも大変都合のいい例であった。ここでは再帰節が複数ある一般的な例をとり、必要な拡張を行ないながら説明することにする。第3引数のリストの要素から第2引数の要素を完全に取り除く *del* というプログラムを考えよう。外延的仕様として例えば次のものを与える。

$\mathcal{E}(del) = \{+[[a,c,d],b,[a,b,b,c,b,d]], +[[c,d],b,[b,b,c,b,d]], +[[c,d],b,[b,c,b,d]], \cdots, +[[],b,[]]\}$

以上の整列した外延的仕様の連続2例 (第1例と第2例、第2例 + 第3例) を対応する入出力変数に代入することにより ps1 中の述語は次の外延的仕様が与えられることになる。

$[Y,X1,X2]+[PY,PX1,PX2]$	$C1\hat{}[PX1,X1,X2]$	$C2\hat{}[PX2,X1,X2]$	$A\hat{}[Y,PY,X1,X2]$
第1例 + 第2例	[b,b,[a,b,b,c,b,d]]	[[b,b,c,b,d],b,[a,b,b,c,b,d]])	[[a,c,d],[c,d],b,[a,b,b,c,b,d]]
第2例 + 第3例	[b,b,[b,c,b,d]]	[[b,c,b,d],b,[b,b,c,b,d]]	[[c,d],[c,d],b,[b,b,c,b,d]]

表 2.3 : $\mathcal{E}(del)$ による ps1 の特殊化

[3]従って $\mathcal{E}^*$ には正の入出力例のみからなることになる。負例は EXN によるプログラム合成の後利用することができるが、本稿では触れない。

ここで、$C1(PX_1,X_1,X_2) \leftarrow (PX_1 = X_1)$、$C2(PX_2,X_1,X_2) \leftarrow (X_2 = [\_|PX_2])$ が実は望ましい特殊化になるが、$C1,C2$ のような特殊な述語が $\mathcal{LPD}$ に入っていることは期待できない。そこで、外延探査プログラムの適応性を大幅に高めることをまず考える。補助述語に与えられる外延的仕様の特徴は、1) 入力引数は冗長である（例：$C1$ における X_2、$C2$ における X_1）。そして 2) 簡単な変換で既存述語を活かせることが多いことである（例：$C2$ における cdr 関数の X_2 への適用により等号述語が再利用できる）。そこで、引数を落す除去関数 $w(.)$、各引数に適応変換用関数データベース ADB の関数を適用する変換関数 a を使うことで既存述語の再利用性を大幅に高められると考えられる。

補題 1 正負 *IO-pair* の有限集合 $\mathcal{E}'$ の各要素が $a(w(.))$ により変換された結果の集合を $\mathcal{E}$ とする。有限集合 $\mathcal{E}$ を外延的仕様とする任意の述語 p に対して、述語 $P\hat{}Zs \equiv p\hat{}a(w(Zs))$ は $\mathcal{E}'$ を満たす。

Lemma 1 は $\mathcal{E}'$ は $a(w(.))$ によって $\mathcal{E}$ に変換される時、$\mathcal{E}$ を仕様とする p によって作られる $p\hat{}a(w(Zs))$ がもとの $\mathcal{E}'$ を外延的仕様として満たすことを保証している。このことは第一要素不変である限り、どんな変換関数、除去関数でも成り立つ。この意味で健全なアルゴリズムを構成できる。a,w の可能な候補を試すことによって拡張された外延探査アルゴリズムを拡張外延探査アルゴリズムと呼ぶ。

変換関数 a が適用する関数の選択肢を広げることは再利用可能なプログラムの数を増やすことになるが、一方でシステム全体の計算量を指数関数的に増大させるため（次章において計算量の考察がある）、変換プログラム・データ・ベース (ADB) は極めて限定的に設定する必要がある。ADB は一般的な基本関数 (car,cdr,+,* など) および必要に応じて利用者やプログラムが各使用領域で使う特殊な合成関数を陽に定義し容易に格納することを可能にする。これはの変換能力を利用者が自在に拡張、改良することを可能にし、また、ADB への登録を限定的に行なうことで、探索の場合の数を効果的に押え、計算時間を有効に使うことを可能にする。

del の例に戻る。拡張された外延検索アルゴリズムによって、望ましい C_1,C_2 を得ることができる。だが、表 2.3 の外延仕様を満たす A は意図するものではない。2 例のみ（線形再帰フラグメントの場合）で完全な探索が行なえる利点を生かすことができる。まず、第 1 例＋第 2 例のみでスキーマ片 ps1 の精密化を行なう：$A(Y,PY,X_1,X_2) \leftarrow (X2 = [A|\_], Y = [A|PY])$。この結果次のプログラム片 p1 を得る。

```
p1:  del(Y,X1,X2) :-                    % 部分計算によって
          PX_1 = X1, X2 = [_|PX2],      %  p1:  del([A|PY],PX1,[A|PX2]) :-
          del(PY,PX1,PX2),              %                       del(PY,PX1,PX2).
          X2 = [A|_], Y = [A|PY].       %
```

今、p1 に関して第 1 例は第 2 例に依存している。外延仕様は依存列である前提だから、同様に、第 i 例が第 $i+1$ 例に現在のプログラム片の並びのもとで依存しているかを調べる。この場合、第 2 例は第 3 例に依存していないので、新たなプログラム片を第 2 例＋第 3 例から p2 を作ろうとする。p1 によって第 2 例から好ましくない計算をしないようにするために、以下のような処理を行なう。まず、第 2 例を満足し、かつ、p1 によって依存していた例を満足しないような条件式 I を以下の外延的仕様から拡張外延検索プログラムによって求める。

$$\mathcal{E}(I) = \{-(b,[a,b,b,c,b,d]),+(b,[b,b,c,b,d])\}$$

この結果、$I(X_1,X_2) \leftarrow (X_2 = [X_1|\_])$ を得る。これから求める p2 を p1 の前に置き、p2 の本体部に I を置けばよい。この結果、正しい複数の節ができあがる。（3 章に del の合成結果がある。）

2.4 発見

リストを反転させるプログラム rev の合成を例にとる。

$\mathcal{E}(rev) = \{+[[a,b,c],[c,b,a]],+[[a,b],[b,a]],+[[a],[a]],+[[],[]]\}$ を与える。ps1 同様に

$[X,Y]+[PX,PY]$	$C\hat{}[PX,X]$	$A\hat{}[Y,PY,X]$
第 1 例＋第 2 例	[[b,a],[c,b,a]]	[[a,b,c],[a,b],[c,b,a]]
第 2 例＋第 3 例	[[a],[b,a]]	[[a,b],[a],[b,a]]

表 2.4：$\mathcal{E}(rev)$ による ps1 の特殊化

補助述語 A に対し、拡張外延探査アルゴリズムによる既存プログラムの探査・再利用が失敗する場合（すなわち、concat $\notin \mathcal{LPD}$ の場合）を考える。この場合、新プログラムの生成を試みる。適応関数 $a(w(.))$ 適用後の出力が新述語のための仕様になる。プログラム発見のためにまず新しいプログラム名を与る。次に再帰的にスキーマに基づく合成プログラムを呼びだし新プログラムの生成をはかる。今、適応関数が第3引数にのみ car を適用しているとする。この時、新プログラム $newp$ が満たすべき入出力例は

$$\mathcal{E}(newp) = \{+[[a,b,c],[a,b],c],+[[a,b],[a],b],+[[a],[],a]\}$$

ところが、これは実際には依存列になっていない。一般に入力される外延的仕様は必ずしも依存列ではない。このような場合に対する完全な答はまだ持っていないが、以下の処理によって、ある程度依存整列させることができる。それぞれの例は、個体項に関して独立であることに着目する。すなわち、個体項は各例において rename してもその意味をこの例では変えない。そこで、適当に rename する（3章参照）。(但し、rename された仕様で得られたプログラムは必ずしももとの仕様に対し正しくないので、得られたプログラムがもとの仕様を満たすかを後で調べる必要がある。) この rename によって

$$\mathcal{E}(newp) = \{+[[a,b,c],[a,b],c],+[[b,c],[b],c],+[[c],[],c]\}$$

が出力され、これを（再帰的に）スキーマに基づく合成プログラムに渡すことで、newp プログラムの生成に成功する。

このスキーマによる合成アルゴリズムと拡張外延探査アルゴリズムからなるプロトタイプシステムの名を **ARDEX** と呼ぶ。このように **ARDEX** は補助プログラムの生成に関して外部からの情報を一切必要としないプログラムになっている。

2.5 計算量と考察

発見を含まない計算量に関する考察を行なう。拡張外延探査アルゴリズムの計算量はほぼ $\Sigma_{n=0}^{m}(mPn * c^n * D * 1Ret * e) \simeq m! * c^m * D * 1Ret * e$ であらわせる。ここで、m は 拡張外延探査アルゴリズム に求める述語の引数の数、n は 外延探査アルゴリズムに求める述語の引数の数、c は 構成述語の数、D は $\mathcal{LPD}$ にある n-引数述語の数（n に対し一様とする）、$1Ret$ は 直接実行にかかる時間の上限（有限）、e は入力された外延仕様 $\mathcal{E}$ の大きさ ($e = \mid \mathcal{E} \mid$) である。この解析からまず、ADB の大きさ c は極めて限定的でなければならないことがわかる。現在のところ、ADB には小数の一般的な変換関数を置くだけで高い効果が上がっており、このことは実際上、大きな制限にならないのかもしれない。

しかし、もう引数の数 m に対する鋭敏性は本質的制限になりうる。既存プログラムの各引数の意味がわからない一般的な状況では、この方法による支援は実際上、引数の数が少ない場合に限られるだろう（現プロトタイプシステムでは6、7引数が限度？）。一方、引数の数は実際上限定的だという見方からすれば、$\mathcal{LPD}$ の大きさに関する効率がもっとも重要であろう。この点、本研究の Algorithm は $O(n)$ であり決して悪くないことになる。

次にスキーマに基づく合成アルゴリズムの計算量を考える。論理型プログラムの統合において、スキーマを与えることによって生ずる効果を計ることによってどのようなスキーマを与えることが望ましいかを考察する。図 2.3 のスキーマの具体化の一つであるプログラムを獲得する場合の数をスキーマがある場合とない場合を簡単に比較する。$\mid X_s \mid = n$ とする。ボディー部の述語の個数は $n+3$。個々の述語に対し候補となる述語の数を一様に D 個とする。まず、スキーマを与えない場合を扱う。Prolog の論理積は交換則が成り立たないので述語の出現順には意味がある。よって、D^{n+3} の Prolog 節が述語の出現位置の場合分けで生じる。これが大まかな検索回数になる。D は利用する既存プログラムの集合の濃度であり、多数の既存プログラムを利用する立場からするとこの探査空間は十分に否定的な意味を持つ。一方、このスキーマを与えた場合の探査を考える。すべての述語は Y, X_s, PY, PX_s のみに依存しており、個々の値は決まっているので、ある述語の選択が他の述語の選択に影響を与えることはない。従って、個々独立に検索するのみで良い。従って、高だか $(n+3) * D$ 回の検索で済むことになる。

定義 7 ある述語 p に対して、ある節 C が透明な *(transparent)* 構造を持つとは、C において現れる変数の集合を V_C、C において p の述語が引数として持つ変数の集合を $V_C(p)$ とした時、$V_C(p) = V_C$ となること

を言う。

C が透明な構造を持つ時、C における p の具体化により各述語に対する外延的仕様が与えられる。C に関して依存関係にある p の例を与えることで、C において述語変数が M 箇所あると D^M の探索空間を $M*D$ の探索空間に縮小させることができる。プログラムスキーマと依存順の外延仕様を与える方法だけがこの利点を受け継ぐ。我々は既存方法ではなぜ背景プログラムの利用がうまくいかないかの理由がここにあると考えている。

3 実験

本章では以上の考察をもとに作成したプロトタイプシステムとその実験による性能評価を報告する。

プロトタイプシステム ARDEX はここで述べられなかった機能が幾つかある。その主だったものを下に述べる。詳しくは別の機会に述べたい。

- rename と領域構成関数の ADB への登録： 一般にスキーマに基づく合成プログラムに入力される外延的仕様は必ずしも依存整列していない。本プロトタイプでは、例間の項の部分マッチによって個体項を rename し、またその差分を ADB へ登録する。この登録によって $\mathcal{S}(even) = \{[4],[2]\}$ に対し $PX = X-2$ や、$\mathcal{S}(nat) = \{[s(s(0))],[s(0)]\}$ に対し $s(PX) = X$ など特殊な変換関数が一時的に ADB へ追加される。

- 停止条件探査機能： 一般に停止例付近では情報が少なくなり多くの候補プログラムが出てしまう問題がある。例えば、先の rev の例ですべてのサブプログラムが既知プログラムから再利用可能な場合最小の例の数は 2 であり、これは停止例を含む必要があるため、{ rev$^+$([a],[a]), rev$^+$([],[]) } を例示することになる。しかしながら、これを満足するプログラムは X = Y, X == Y,... など非常に多い。この問題を解決するため、停止例から離れた連続例のみから、得られた帰納節を後向きに適用し、停止条件を求める機能を設けている。

客観的実験: $\mathcal{LPD}$ = 5 1 個の Prolog 組み込み述語 + Quinlan の例 [11] + ADB, SDB = {ps0,ps1}
ADB = List 関係 7 個 + 数値関係 5 個、 言語: Quintus Prolog、Mashine: Sun 4/370

system	class	invention	target: reverse CPU sec.	target: reverse no. of ex.
GOLEM(90)	determinate	no	2.6(C)	(12)
FILP(93)	determinate	no	4.4(CP)	(4)
FOIL(90)	discriminate	no	247(C)	4761
CHAMP(92)	list tail rec.	yes	24.2(SP)	256(16)
ARDEX(94)	linear rec.	yes	3.8/1.6 (QP)	3/2

CP: C-prolog
SP: Sicstus Prolog
QP: Quintus Prolog
invention / no invention

```
% E*(nat) = {[s(s(0))], [s(0)]}
nat(0) :- true.
nat(s(X)) :- nat(X).
%0.87sec.

%E*(reach) = {[n0,n8],[n3,n8]} %Quinlan's%
reach(X,Y) :- linked_to(X,Y).
reach(X,Y) :- reach(Z,Y),linked_to(X,Z).
%4.3sec. after 2 unintended clause.

% permutation : invention of 'select'
% E*(perm) = {[[b,a,c],[a,b,c]],
%             [[b,c],[b,c]],[[c],[c]]}
perm(X,X) :- true.
perm(X,[Y|Z]) :- perm(A,Z),newpred1(X,A,Y).
```

```
| % E*(fact) = {[24,4],[6,3]}
| fac(1,0) :- true.
| fac(X,Y) :- add(1,Z,Y),fac(A,Z),mpy(A,Y,X).
| % 1.72sec.

|%E*(gcd) = {[6,48,18],[6,18,12],[6,12,6]}
|gcd(X,X,0) :- true.
|gcd(X,Y,Z) :- A is Y mod Z,gcd(X,Z,A).
|%3.72sec

|%sub_list: invention of 'prefix'
|%E*(sl) = {[[b,c,d],[a,b,c,d,e]],
|%         [[b,c,d],[b,c,d,e]],[[c,d],[c,d,e]]}
|sl([X|Y],[X|Z]) :- newpred0([X|Y],[X|Z]).
|sl(X,[Y|Z]) :- sl(X,Z).
```

```
newpred1([X|Y],Y,X) :- true.
newpred1([X|Y],[X|Z],A) :- newpred1(Y,Z,A).
% 7.0sec.
```

```
|newpred0([],X) :- true.
|newpred0([X|Y],[X|Z]) :- newpred0(Y,Z).
|%7sec. after unintended 6th clause.
|%1st is 'included' pred in 2.6sec.
```

```
%insert::
%E*(ins) = {[[3,5,6,7],5,[3,6,7]],
%           [[5,6,7],5,[6,7]],[[5,7],5,[7]]}
ins([X,Y|Z],X,[Y|Z]) :- X<Y,!.
ins([X|Y],Z,[X|A1]) :- ins(Y,Z,A).
%2.45sec
```

```
|% delete::
|%E*(del) = {[[a,c,d],b,[a,b,b,c,b,d]],
|% [[c,d],b,[b,b,c,b,d]],[[c,d],b,[b,c,b,d]]}
|del([],Y,[]) :- true.
|del(X,Y,[Y|Z]) :- !,del(X,Y,Z).
|del([X|Y],Z,[X|A]) :- del(Y,Z,A).
|%2.93sec
```

主観的実験: 上記条件に加え、$\mathcal{LPD}$ に [12] の basic library と lists library および開発中の **ARDEX** 自身を加え、**ARDEX** 開発ツールとして使う実験を行なった。自分の書いたプログラムの名前や引数の順序を忘れることがしばしばあったので、そのような時 **ARDEX** は確かに役に立った。また、何度か著者の知らないプログラムをライブラリから見い出すのに成功し、プログラムの労が省かれた。

4 結論

本研究は不特定多数の背景知識の利用を自動プログラミングの観点から考察した初めての研究と思われる。部分仕様としての例からの自動プログラミングは例からの学習としてもとらえられる。機械学習は計算量の観点から多くの限界があることが理論的に知られ、学習機構そのままでは実用的なプログラム合成は困難である。検索機構を持つことによって実用的なプログラムを目的プログラムの中に取り込むことができるため、これにより目的プログラムの実用性を高めることができ、学習の理論的限界を緩和できる可能性が出た。

また、ここで提示したスキーマと依存整列した例示による方法は、理論的にも実験的にも優れた効率を持つことが確かめられた。また、合成に必要な例の数は従来法に比べ著しく少なくない。この他、補助述語の発見能力を持つこと、領域に依存しない能力（弁別的・決定的性格を問わない。文字・数値処理を問わない）を持つこと、合成されるプログラムの限界がはっきりしており（ヒューリスティックによらない）停止性が保証された完全な探査を行なうことができること等、優れた特徴がある。

参考文献

[1] Bergadano,F. and Gunetti,D. 1993. An Interactive System to Learn Functional Logic Programs, in *13th International Joint Conference on Artificial Intelligence*, pp 1044-1049.

[2] De Raedt,L. and Bruynooghe,M. 1989. Towards Friendly Concept-Learners, in *11th International Joint Conference on Artificial Intelligence*, pp.849-854.

[3] Hardy,S. 1975. Synthesis of Lisp Functions From Examples, in *4th International Joint Conference on Artificial Intelligence*, pp. 240-245.

[4] Ishizaka, H. 1990. Private communications.

[5] IMSL 1987. *IMSL Math/Library User's Manual. 1.0 Edition*, Tex.

[6] Katz,S.,Richter,C.A., and The,K. 1987. PARIS: A system for reusing partially interpreted schemas, in *9th International Conference on Software Engineering*, IEEE Computer Society Press, CA, pp 377-385.

[7] Kijsirikul,B., Numao,M. and Shimura,M. 1992. Discrimination-Based Constructive Induction of Logic Programming, in *the National Conference on Artificial Intelligence*, pp 44-49.

[8] Krueger,C.W. 1992. Software Reuse, *ACM Computong Surveys*, Vol.24, No.2, pp 131-183.

[9] Martin-Löf,P. 1984. *Intuitionistic Type Theory*, Bibliopolis, Napoli.

[10] Muggleton,S. and Buntine,W. 1988. Machine Invention of First Order Predicates by Inverting Resolution, in *the 5th International Workshop on Machine Learning*, Morgan Kaufmann, MI, pp 339-352.

[11] Quinlan,J.R. 1990. Learning Logical Definitions from Relations, in *Mashine Learning* 5, Kluwer Academic Pub., pp 230-266.

[12] Quintus Prolog 1990. Release 2.5, *Quintus Prolog Development Environment.*

[13] Summers,P.D. 1977. A Methodology for LISP Program Construction from Examples, in *J. of the Association for Computing Machinery*, Vol.24,No.1,pp. 161-175.