ソフトウェア工学 90−11
（1993. 2. 5）

プログラム変換履歴の一般化

鈴木秀明　西谷泰昭
群馬大学 工学部 情報工学科

概要

プログラム変換過程の再利用を目的として、変換履歴を一般化し自動的に変換履歴スキーマを作成する手法を与える。変換履歴は対話的に与えられるプログラミング手法と問題固有部分の知識を含んでいる。スキーマはその問題固有部分を一般化することで得られ、プログラミングの知識のみを含む。一般に問題固有部分を見つけることは難しく、自動化されていない。複数の履歴に共通な部分は一般性が高いという観点から、二つの似たような履歴を二階の照合によって比較し問題固有部分を見つける。また、同じプログラムを導出する変換履歴は多数存在するため、標準履歴を導入する。実験システムによって作成した「終端再帰」「二分法」を表す変換スキーマを示す。

Generalization of Program Transformation Histories

Hideaki Suzuki　Yasuaki Nishitani
Department of Computer Science, Gunma University

Abstract

For the purpose of reusing program transformational processes, we develop an automatic method for generalizing transformation histories into a scheme. Transformation histories contain the knowledges about programming techniques and particular problems. The schemes, which contain only the programming kowledge, are obtained by abstracting the parts of histories concerned with particular problems. No automatic methods to find such parts have beed presented in general. We compare two similar histories by the second order pattern matching and find such problem specific parts. Since there are various transformation histories that derive the same program, we introduce normal forms of histories. We present several transformation schemes derived by the experimental system, in which tail recursion and binary search techniques are embodied.

1 はじめに

プログラム変換は生産性向上や再利用を目指した研究の一つである。プログラム変換の一つの特徴は、プログラム生成の過程が連続的なルールの適用からなるため、他のプログラム生成法に比べて、その生成過程を正確にとらえることができることにある。このプログラムの生成過程を記録し、新たなプログラムの生成に再利用することで効率的にプログラムを作成することが提案されている [1][3]。

プログラム変換過程の記録を**変換履歴**と呼ぶ。変換履歴には、式のどの部分にどのルールがどのような順序で適用されたかが形式的かつ明示的に記録される。変換ルールの適用はプログラマによって対話的に行なわれるが、これはプログラマの知識を変換システムに与えていると考えられる。このため、変換履歴にはプログラマの知識が含まれている。履歴中のプログラマの知識は (1) プログラミングの知識と (2) 問題固有の知識の二つに大きく分けられる。例えば「二分法により平方根を求めるプログラム」では、「二分法」に関わる部分は一般的なプログラミングの知識であり、「平方根」に関わる部分は問題固有の知識である。

変換履歴は問題固有の知識を含んでいるため、他の問題への直接的な利用は難しい。このため、変換履歴の問題固有部分を一般化し、**変換履歴スキーマ**としてプログラミングの知識を抽出する。つまり、変換履歴スキーマはプログラム変換における一般的なプログラミングの知識（変換手法）を抽出したものである。

本稿では図1で示されるように、プログラム変換システムにおいてプログラマとの対話によって得られたプログラミングの知識を抽出し、変換履歴スキーマを自動的に生成することを目指す。これまで、問題固有部分を自動的に識別する手法がなく、ヒューリスティクに行なわれていた [1]。本稿では、これを自動的に行なう手法を与える。基本的な考え方は、複数の変換履歴に共通な部分は一般性が高いという観点から、似たような変換履歴をパターン照合により比較し、問題固有の部分を自動的に識別する、というものである。

パターン照合は関数の照合を考え Heute 等 [2] の二階の照合アルゴリズムをもとに、履歴を扱うことができるように拡張する。履歴をパターン照合によって比較するには、対応するルールが同じ順番で適用されている必要があるが、一般には対応するルールが同じ順番で適用されていない。このため、履歴をルールの適用順序に一定の規則を設けた標準形に変換し、その標準履歴の照合を行なうことでスキーマを作成する。

2 章でプログラム変換 (2.1 節) と変換履歴 (2.2 節) について簡単に述べる。3 章で変換履歴の一般化手法とその課題について述べる。特に、一般化における課題である標準履歴とパターン照合について 3.1 節 3.2 節で述べる。また、4 章で例として reverse と factorial の変換履歴から作成した、「終端再帰」を表す変換履歴スキーマと、search と sqrt の変換履歴から作成した「二分法」を表す変換履歴スキーマを示す。5 章では、変換履歴スキーマを利用した変換について述べる。

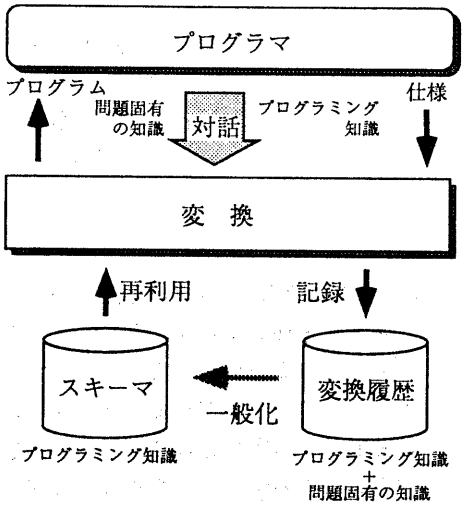

図 1: 変換、履歴、スキーマ

2 変換と履歴

2.1 プログラム変換

プログラム変換は、ある言語で書かれたプログラム Prog を変換ルール r により、「意味」を変えずにより効率的な同じ言語のプログラム Prog′ に変換することである。このとき、各式は環境 E を持つ。環境は、プログラムの適用条件、プログラム構造（条件文など）、適用されたルールにによって決まる。変換ルール r の適用によって環境が E′ になったとき、その変換を次のように表す。

$$(\mathrm{Prog}, \mathrm{E}) \xrightarrow{\mathrm{r}} (\mathrm{Prog}', \mathrm{E}')$$

ルールは以下のように定義される。

$$\mathrm{r} :< \mathrm{P}_1 \rightarrow \mathrm{P}_2 |\ \mathrm{Cond},\ \mathrm{Env} >$$

P_1 は変換前のパターン、P_2 は変換後のパターン、Cond はルールを適用する条件、Env は変換後に環境に加える条件である。ただし Cond と Env がないときは省略する。ルール r に対して変換前のパターン P_1 を prepat(r) で表し、変換後のパターン P_2 を postpat(r) で表す。また、次の条件を満たせば式 (exp,E) はルール r で変換可能であるという。つまり、対象式 exp と prepat(r) が代入 σ によって照合し、そのとき適用条件 Cond[σ] が証明可能であれば、変換可能である。

$$\exists\sigma\ \mathrm{exp} = \mathrm{prepat(r)}[\sigma] \text{ and } \mathrm{E} \vdash \mathrm{Cond}[\sigma]$$

2.2 変換履歴

変換過程を記録するために、変換履歴を定義する。変換は通常、(1) ルール、(2) ルール適用の位置、(3) ルール適用の順序によって決まる。したがって、これらの情報を明確に記録することで、変換の過程を記録することができる。

しかし、ルール、適用位置、適用順序だけを履歴として記録した場合、履歴を見てもルール適用の対象式が明示的でないため、どのような式のどの部分がどのように変換されたかがわからない。また、本稿では二つの変換履歴をパターン照合により比較し、変換履歴スキーマを自動的に生成することを目指しているため、ルール適用の対象となる式がルールの対応を考える上で重要な役割を果たす。このため、ルール、適用順、適用対象を明確に記録するように変換履歴を定義する。

[定義 1] 変換履歴

- 定数や変数 e が無変換であることを表す履歴は [e] である。
- 式 $(e_0 \cdots e_n)$ において、$e_0, \cdots, e_n$ をそれぞれ $e'_0, , \cdots, e'_n$ に変換したときの履歴を $R_0, \cdots, R_n$ とすれば、$(e_0 \cdots e_n)$ の $(e'_0 \cdots e'_n)$ への変換履歴を $[(R_0 \cdots R_n)]$ と表す。
- e_i の e'_i への変換履歴を $[R_i]$ とし、e'_i がルール r_i によって e_{i+1} に変換されるとする。このとき、e_1 の e'_n への変換履歴を次のように表す。

$$[R_1\ r_1\ R_2\ \cdots\ R_{n-1}\ r_{n-1}\ R_n]$$

特に、このタイプの履歴を「履歴フォーム」と呼ぶ。

□

簡単のために履歴フォーム以外は [ ] を省略することもある。また、式 exp から式 exp′ への変換履歴 R に対して、変換前の式を init(R)、変換後の式を final(R) で表す。

ルール r:< (append ?X nil) →?X > を用いた append に関する次の変換を考える。

$$(\text{append}_1\ (\text{append}_2\ \text{nil}\ \text{nil})\ (\text{append}_3\ \text{nil}\ \text{L})) \xrightarrow{*} \text{L}$$

$append_2$、$append_3$、$append_1$ の順にルール r を適用したときの変換履歴を 図 2(a) に示す。図 2(b) は $append_2$、$append_1$、$append_3$ の順にルール r を適用したときの変換履歴である。このように、同じ結果を得る変換過程は複数存在することがある。標準履歴については、3.1 節で述べる。

```
[ (append1                    [ (append1
  [ (append2 nil nil)           [ (append2 nil nil)
    r                             r
    nil ]                         nil ]
  [ (append3 nil L)             (append3 nil L))
    r                           r
    L ])                        (append3 nil L)
r                               r
L]                              L]

        (a)                           (b)
```

図 2: 変換履歴

3 変換履歴の一般化

変換履歴には個々の問題に固有な部分が含まれているため、他の問題に直接再利用するのは困難である。例えば、次の n! と $\sum_{i=1}^{n} i$ の仕様はそれぞれ似たような終端再帰プログラムに変換でき、それぞれの履歴に終端再帰に変換するプログラミングの知識を持っている[1]。

$$\text{n!} = (\text{fold}\ (\lambda\ (\text{X Y})\ (\times\ \text{X Y}))\ 1\ (\text{mklist}\ 1\ \text{n}))$$

$$\sum_{i=1}^{n} i = (\text{fold}\ (\lambda\ (\text{X Y})\ (+\ \text{X Y}))\ 0\ (\text{mklist}\ 1\ \text{n}))$$

しかし、× − + と 1 − 0 の部分はそれぞれ問題に固有の部分であり、適用されるルールが異なるため、n! の変換履歴を直接 $\sum_{i=1}^{n} i$ の変換には利用できない。そこで、変換履歴における問題固有部分を抽出し、その部分を一般化することで他の問題にも利用可能な変換履歴スキーマを作成する。

具体的には、問題個有の定数や関数をスキーマ変数に置き換えることで行なう。例えば、[2] では終端再帰へ変換するルールパターンが与えられているがその作成法については述べられていない。これは問題固有部分を自動的に判断するのが難しいためであり、[1] ではヒューリスティックに問題固有の部分が決められていた。

本稿では、スキーマの作成を自動的に行なうことを考える。つまり、複数の変換履歴に共通な部分は一般性が高いという観点から、似たような変換履歴をパターン照合により比較し、問題固有の部分を自動的に識別する。また、変換履歴をスキーマ化することで Heute 等 [2] の変換スキーマでは表現できない変換のスキーマを記述できる。

一般化の手続は次のような段階で行なう (図 3)。

(1) パターン照合を行なうために、一方の変換履歴をパターン化する。

[1] mklist は引数として二つの整数をとり、第一引数から第二引数までの整数のリストを作成する関数。
(fold f x_0 [$x_1 \cdots x_n$]) = (f x_1 ⋯ (f x_{n-1} (f x_n x_0)) ⋯)

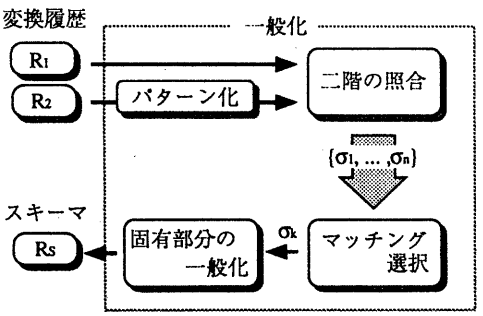

図 3: 変換履歴の一般化

(2) 履歴パターンと履歴との二階のパターン照合を行ない、マッチングの候補を作成する。

(3) マッチングの候補の中から、最適と思われるマッチングを選択する。

(4) 選択されたマッチングから、二つの履歴に共通する部分と固有の部分を識別し、その固有部分をスキーマ変数に置き換えることで履歴を一般化する。

このとき、次のような点が課題となる。

- 履歴におけるルール適用の順序
  履歴の照合を行なうためには、対応するルールが同じ順序で適用されている必要があるが、通常ルール適用の順序が異なり照合ができない。このため、ルールの適用の順序が一定である標準履歴を導入し、標準履歴を対象に照合を行なう必要がある。

- 変換履歴のパターン化手法
  式の中に同じ定数や関数が現れていたとしても、それらが意味的に同じであるのか一般には判断できない。このため、単純に同じ名前の定数や関数を同じパターン変数でパターン化できない。

- 照合アルゴリズム
  一般に、照合は式を対象としているので、履歴を扱えるように拡張する必要がある。また、通常の（一階の）照合では、例えば (P 3) と (+ 1 3) の照合はできない。しかし、パターン変数 P を関数と見なし λx.(+ 1 x) とすることで照合が可能となる。履歴の照合では異なる関数の対応をとる必要があるため、二階の意味での照合が必要である。

- 完全には照合に成功しなかった場合の扱い
  履歴の照合では、完全には照合に成功しなくても、照合に成功した部分の変換は共通の変換手法がとられている部分であり、有効な変換手法と考えられる。このため、照合可能な部分をとり出す必要がある。

以下、標準履歴、パターン化、照合について述べる。

3.1 標準履歴

一般に、仕様からプログラムを生成する過程は、ルールの適用順の違いから複数存在する。このため、人が見て同じ意味を持つと思われる変換履歴であっても、ほとんどの場合見かけ上は異なる変換履歴である。例えば、図 2で与えた

$$(\mathrm{append}_1\ (\mathrm{append}_2\ \mathrm{nil}\ \mathrm{nil})\ (\mathrm{append}_3\ \mathrm{nil}\ \mathrm{L})) \stackrel{*}{\rightarrow} \mathrm{L}$$

の変換では、append に関する変換をどのような順番で行なうかによって三通りの変換がある（変換履歴として表すと図 2(a)、図 2(b) の二通りとなる）。

直観的には、「できる限り外側の式に関する変換を先に行う」ようにルール適用の順序を制限した履歴を標準履歴とする。例えば、上記の変換は append_2、append_1、append_3 の順番で行なう（図 2(b)）。

ここで append_2 は append_1 の部分式であるため、append_1 を先に変換すべきであるが、append_1 の変換は append_2 の変換結果を利用しているため、append_2 の変換を先に行なう。一方、append_3 の式は append_1 の変換の前あるいは後のどちらでも変換できるが、外側の式である append_1 の変換を先に行なう。

つまり、標準履歴では各ルールについてルールの定数部と対応する部分の変換は、そのルール適用の前に行なわれ、パターン変数と対応する部分はそのルール適用の後に行なわれる。ただし、ルールでは同じパターン変数は同じ式と対応しなくてはならないので、この制約を満たすために行なわれる変換はそのルール適用の前に行なわれる。

標準履歴を形式的に定義するために、履歴の合成 (∘) を定義する。

[定義 2] final(R) = init(R′) である履歴 R と R′ について、履歴の合成 R ∘ R′ は次のように再帰的に定義される。

- R= 定数 (変数) R′= 定数 (変数) のとき
  R
- $R = (R_1 \cdots R_n)$ $R' = (R'_1 \cdots R'_n)$ のとき
  $(R_1 \circ R'_1 \cdots R_n \circ R'_n)$
- $R = [R_1\ r_1 \cdots R_n]$ $R' = [R'_1\ r'_1 \cdots R'_n]$ のとき
  $[R_1\ r_1 \cdots R_n \circ R'_1\ r'_1 \cdots R'_n]$

□

また、LCPF(Longest Common PostFix) は final が等しい履歴 $R_1, \cdots, R_n$ について、$R_i = R'_i \circ R_{lcpf}$ を満たすような R_{lcpf} のうちで、含まれるルールの数が最

も多いものを表す（付録参照）。標準履歴の定義は次のように与えられる。

[定義 3] 標準形

履歴 R のフォームにしたがって次のように標準履歴を定義する。

R= 定数 (変数)

R は標準形である。

R = (R_1 ⋯ R_n)

$R_1 \cdots R_n$ が標準形なら R も標準形である。

R = [R_1 r_1 $R_2 \cdots R_n$]

$R_1 \cdots R_n$ がすべて標準形で、R_i r_i の組が以下の条件を満たすとき R は標準形である。

条件： R_i r_i について、履歴の定義から final(R_i) = prepat(r_i)[σ] であるような σ が存在する。prepat(r_i) に出現するパターン変数を $X_1, \cdots, X_m$ とし、$\sigma = [e_1/X_1, \cdots, e_m/X_m]$ であるとする。さらに $X_j (j = 1, \cdots m)$ について、prepat(r_i) における X_j の出現をそれぞれ、$X_{j,1} \cdots X_{j,l}$ とし、$X_{j,k}$ と対応した、R_i の部分履歴を $R_{i\,j,k}$ とする (final($R_{i\,j,k}$) = $e_j (k = 1, \cdots, l)$)。このとき LCPF($R_{i\,j,1}, \cdots, R_{i\,j,l}$) = e_j がなり立つ。

□

3.2 パターン化と照合

パターン照合は、パターンと式（履歴）を用いて行なう。このため、照合のために与えられる二つの履歴の一方をパターン化する。このとき、同じ関数の意味の違いを自動的に判断するのは難しいために、最も適切なパターン化を自動的に行なうことは難しい。例えば、

1. (cons (cons X nil) nil)

2. (cons (+ X 0) nil)

が対応しているとすると、1 式に出現する二つの cons や二つの nil は意味的に異なるものである。このため、二つの cons や二つの nil を同じパターン変数でパターン化すると 1 式と 2 式の照合はできない。このような意味の違いを自動的に判断するのは困難であるので、基本的に同じ名前の関数や定数であっても異なるパターン変数でパターン化する。しかし、パターン変数が増えると照合の効率が落ちるため、できるだけ同じ変数を用いるために次のような方針をとる。

履歴においては各ルール適用において、変換前の式を exp 変換後の式を exp' とするとルール r に対して exp=prepat(r)[σ] exp'=postpat(r)[σ] という関係がある。このため、exp と prepat(r)、exp' と postpat(r) においては定数や関数に対応関係がある。このような、対応する定数や関数は同じパターン変数でパターン化する。

```
[ (P1 [ (P2 P3 P4) <(P2 P3 ?X) → ?X> P4 ]
      (P5 P6 L))
 <(P1 P4 ?X) → ?X>
 (P5 P6 L)
 <(P5 P6 ?X) → ?X>
 L ]
```

図 4: 変換履歴パターン

例えば、図 2(b) の変換履歴をパターン化すると、図 4となる。このようにパターン化することで、意味の異なる関数や定数は必ず異なるパターン変数でパターン化され、同じ名前で意味の異なる関数や定数を含む問題でも照合が可能となる。

パターン照合は、Heute 等 [2] による二階のパターン照合を履歴を扱えるように拡張したものを用いる。二階のパターン照合の大きな特徴は、関数を認識したうえで照合を行なうことにある。例えば、次のようなパターンと式について二階の照合を考える。

$$(P_1\ c\ (P_2\ b)) \leftrightarrow (f\ (g\ a\ b)\ c)$$

一階の意味での照合では照合に失敗するが、二階の意味での照合ではパターン変数を、例えば、次のような関数と見なせば照合に成功する。

$$P_1 = \lambda xy.(f\ y\ x) \quad P_2 = \lambda x.(g\ a\ x)$$

これはマッチングの一つの例であり、通常二階の照合では複数のマッチングが得られる。この例では 12 通りの P_1 P_2 の組が得られる。

二階のパターン照合を拡張した部分は、次の二点である。

- Heute 等による照合は式を対象としたものであるので、履歴を対象とするように履歴フォームを扱えるように拡張した。履歴フォーム [$R_1 \cdots R_n$] の照合は各要素毎の照合に分解して行なう。

- 履歴の照合では、完全には照合に成功しなくても、履歴のどの部分までが照合に成功したのかが重要である。照合の順序をルール適用の順序と等しくなるように制限し、さらに照合に成功したルールを全て記録する。これによって、マッチング毎に履歴のどの部分までが照合に成功したのかを明確にする。この情報をもとに照合に成功した部分の履歴をとりだしスキーマを作成する。履歴フォームの照合は $R_1 \cdots R_n$ の順で行う。

```
[ (SV1 L RESULT)
  ...
  (if (null L)
      [ (SV2 SV4 RESULT)
(1)     < (SV2 SV4 ?L) → ?L) >
        RESULT ]
    [ (SV2
       [ ...
         (SV2
          (fold (lambda (X0 X1) (SV2 X1 (SV3 X0)))
            SV4
            (cdr L))
          (SV3 (car L))) ]
       RESULT)
(2)  < (SV2 (SV2 ?X ?Y) ?Z)
       →
       (SV2 ?X (SV2 ?Y ?Z)) >
      (SV2
       (fold (lambda (X0 X1) (SV2 X1 (SV3 X0)))
         SV4
         (cdr L))
       (SV2 (SV3 (car L)) RESULT))
      ...
      (SV1 (cdr L) (SV2 (SV3 (car L)) RESULT)) ]) ]
```

図 5: 終端再帰の変換履歴スキーマ

4 一般化と再利用の例

4.1 終端再帰

factorial を計算するプログラムへの変換履歴と reverse のプログラムへの変換履歴から終端再帰のプログラミング手法を抽出した変換履歴を作成できた。実際には次のように補助関数が導入され、補助関数について変換が行なわれる。

```
(reverse L) = (reverse-aux L nil)
(reverse-aux L RESULT) =
(append
 (fold (lambda (X Y) (append Y (list X)))
       nil
       L)
 RESULT)

(factorial N) = (fact-aux (mklist 1 N) RESULT)
(fact-aux L RESULT) =
(* RESULT
   (fold (lambda (X Y) (* X Y)) 1 L))
```

reverse-aux と fact-aux の変換履歴から作成された変換履歴スキーマを図5 に示す（SV はスキーマ変数）。図5の変換履歴スキーマで一般化されているルールは (1) と (2) である。

$<(SV_2\ SV_4\ ?L) \rightarrow ?L>$
$<(SV_2\ (SV_2\ ?X\ ?Y)\ ?Z) \rightarrow (SV_2\ ?X\ (SV_2\ ?Y\ ?Z))>$

これらのルールはもともと append の単位元と結合律に関するルールであった。この問題固有である append や nil の部分を一般化している。上記のルールスキーマは、スキーマ変数で表された関数 SV_2 が単位元 SV_4 を持ちかつ結合律を満たすことを示している。これは一般のプログラミングにおいて終端再帰プログラムを作成するために必要な知識であり、変換履歴スキーマによってこの一般的なプログラミングの知識を抽出している。

実際にこのスキーマを利用してプログラムを作成する場合 SV_2 と対応する関数が単位元として SV_4 と対応する値をもち、さらに結合律を満たすかどうかが対話によって調べられる。

4.2 二分法

二分法を用いた平方根を計算するプログラム (sqrt) への変換履歴とソートされた配列要素の二分探索のプログラム (search) への変換履歴から二分法を示す変換履歴スキーマを作成した。実際には、補助関数としてそれぞれ sqrt-aux と search-aux が導入されこの関数についての変換履歴からスキーマが作成できた。それぞれの仕様を次に示す[2]。

```
(sqrt A) = (sqrt-aux A 1 (* 100 A))
(sqrt-aux A M N) =
(fold (lambda (X Y)
        (if (and (<= (square X) A)
                 (< A (square (+ 0.01 X))))
            X
            Y))
  nil
  (map (lambda (Z) (* 0.01 Z))
       (mklist M N)))

(search A ARR N) = (search-aux A ARR 0 N)
(search-aux A ARR M N) =
(fold (lambda (X Y)
        (if (equal-data X A) X Y))
  nil
  (map (lambda (Z) (aref ARR Z))
       (mklist M N)))
```

それぞれの補助関数 search-aux と sqrt-aux の変換履歴から得られた変換履歴スキーマを図6に示す。

通常、二分法はソートされた問題の対象に対して次のような方針で行なわれる。

1. 問題の対象を中間で二つに分割する。
2. 中間の値によって場合わけをして、分割された一方を問題の対象から除外する。
3. 残った問題の対象について同様に繰り返す。

[2] $(\mathrm{map\ f\ [x_1 \cdots x_n]}) = [(\mathrm{f\ x_1}) \cdots (\mathrm{f\ x_n})]$

```
[(SV1 A ARR M N)
 ...
 (if [... (> M N)]
     nil
   [...
    (if (SV2 [... (SV3 ARR M)] A)
        [... (SV3 ARR M)]
(1)   [(fold (lambda (X0 X1) (if (SV2 X0 A) X0 X1))
         nil
         (map (lambda (X2) (SV3 ARR X2))
              (mklist (1+ M) N)))
       ...
(2)    (fold (lambda (X0 X1) (if (SV2 X0 A) X0 X1))
(3)      (fold (lambda (X0 X1) (if (SV2 X0 A) X0 X1))
           nil
           (map (lambda (X2) (SV3 ARR X2))
                (mklist X3 N)))
         (map (lambda (X2) (SV3 ARR X2))
              (mklist (1+ M) (1- X3))))
       ...
(4)    (if (SV4
            (SV3 ARR [X3 3 (floor (+ (1+ M) N) 2)])
            A)
          [(fold (lambda (X0 X1)
                   (if (SV2 X0 A) X0 X1))
(5)         [(fold (lambda (X0 X1)
                     (if (SV2 X0 A) X0 X1))
                   nil
                   (map (lambda (X2) (SV3 ARR X2))
                        (mklist X3 N)))
                  ...
                  nil]
                 (map (lambda (X2) (SV3 ARR X2))
                      (mklist (1+ M) (1- X3))))
           2
           (SV1 A ARR (1+ M)
                (1- [X3 3 (floor (+ (1+ M) N) 2)]))]
(6)      [(fold (lambda (X0 X1)
                  (if (SV2 X0 A) X0 X1))
             (fold (lambda (X0 X1)
                     (if (SV2 X0 A) X0 X1))
               nil
               (map (lambda (X2) (SV3 ARR X2))
                    (mklist X3 N)))
             (map (lambda (X2) (SV3 ARR X2))
                  (mklist (1+ M) (1- X3))))
          ...
          (fold (lambda (X0 X1)
                  (if (SV2 X0 A) X0 X1))
            nil
            (map (lambda (X2) (SV3 ARR X2))
                 (mklist X3 N)))
          2
          (SV1
           A ARR [X3 3 (floor (+ (1+ M) N) 2)] N)])])])]
```

図 6: 二分法の変換履歴スキーマ

図 6の変換履歴スキーマでも、同様な手法がとられている。

(1) では、問題の対象は (1+ M) ～ N であるが、これを (1+M) < X3 ≤ N であるような X3 で分割する。これによって、(1) の問題が (1+ M) ～ (1− X3) を対象とした問題 (2) と X3 ～ N を対象とした問題 (3) に分割される。このとき、X3 は free な変数であり実際どのような点で分割するかは決められていない。また、これまでの変換では明示的に問題を二つに分割しただけで効率的には変っていない。

(4) で 分割点 X3 の値によって場合わけをしている。実際には次のような if 文を導入するルールが適用されている。

$$r :<?X \rightarrow (\text{if } ?C\ ?X\ ?X) >$$

ルールの意味としては?C がどのような式であってもよいが、以降の変換によって効率化を行なうためには ?C をどのような式にするかは重要である。search-aux の変換では ?C として (bigger-data (aref ARR X3) A) が変換の途中で対話的に与えられる。?C として対話的に与えられる部分は、そのように分割すれば効率的なプログラムへの変換ができるというプログラマの知識に基づいている。このプログラマの知識は問題固有であるので次のようにスキーマ化される。

$$r' :<?X \rightarrow (\text{if } (SV_4\ (SV_3\ \text{ARR X3})\ A)\ ?X\ ?X) >$$

(5) は $(SV_4\ (SV_3$ ARR X3) A) = T のもとで変換が行なわれ nil となる。つまり、二つの小問題のうち X3 ～ N を対象とした問題は簡単化され 実際に計算をする必要がないことがわかる。

同様に (6) は $(SV_4\ (SV_3$ ARR X3) A) = F のもとで変換され nil となる。つまり、(1+ M) ～ (1− X3) の問題は簡単化される。

これらはまさしく一般的な二分法の手法である。つまり、関数 SV_3, SV_4 で表される A と分割点 X3 との関係によって、分割された二つの問題の一方だけを考えればよいことを表している。

この変換履歴スキーマで、一般化されているルールは上記の r′ のルールだけである。再利用のために仕様の対応からこのルールを特定すると SV_4 がスキーマ変数として残る。このスキーマ変数には対話によって関数が与えられるが、どのような関数を与えたとしてもルールとしては正しい。しかし、以降の変換を行なうためには SV_4 にどのような関数を与えるかが重要である。これは問題固有の知識であり個々の問題毎にどのような関数を与えるかはプログラマの知識に委ねられる。

5 スキーマを利用した変換

変換すべきプログラム (仕様) と似たような仕様を持つ変換履歴スキーマを利用して、通常の変換よりも少

ない手続きで変換を行なう方法を与える。このとき、次の二つが課題となる。

- スキーマを利用した変換の正当性の保証
- 仕様の対応では特定化されないスキーマ変数の扱い。

仕様の比較から変換履歴スキーマを特定化し、その特定化された変換履歴をもとに変換を行なう。しかし、この特定化された変換履歴の正当性は保証されていない。その理由は、スキーマ変数を含むルールを特定化した場合、そのルールが正しいことの保証がないからである。例えば、+ の交換律を表すルールを一般化したルールスキーマ

$$< (SV_1\ ?X\ ?Y) \rightarrow (SV_1\ ?Y\ ?X) >$$

の変数 SV_1 を cons で特定化した場合次のような間違ったルールとして特定化されてしまう。

$$< (\mathrm{cons}\ ?X\ ?Y) \rightarrow (\mathrm{cons}\ ?Y\ ?X) >$$

このような特定化されたルールの正当性の保証は自動的にできないため対話によって正当性を保証する。

さらに、仕様の比較だけから特定化された変換履歴には、特定されていないスキーマ変数が含まれている可能性がある。 このスキーマ変数の部分は、変換のときに対話によって与えられた部分であり、まさに問題固有の部分である。例えば、次のような場合分けを行なうルールでは、ルール適用のとき対話によって ?C に対する式を与えている。

$$<?X \rightarrow (\mathrm{if}\ ?C\ ?X\ ?X) >$$

?C として与えられた式に含まれる関数や定数を一般化した場合、この部分は変換の途中で与えられた部分であるので、仕様の対応からは特定できない。したがって、このようなスキーマ変数が現れた場合、対話によってスキーマ変数を特定する。

実際の手続きは以下のように行なわれる。

(1) 変換履歴スキーマの仕様と与えられた仕様との二階のパターンマッチングを行ない、マッチングの候補を得る。

(2) マッチングの候補から、最適なものを選択する。

(3) 変換履歴スキーマを得られたマッチングによって特定化する。

(4) 特定化された全てのルールの正否の判定を行なう。特定化されたルールがすでにルールとして登録されていれば、正しいルールである。登録されていない場合、正否を対話によって人が判断する。正しいルールであれば新しいルールとして登録する。

(5) 得られた変換履歴（スキーマ）をもとに変換を行なう。ただし、スキーマ変数が出現した時点で、対話によってそのスキーマ変数を特定化する。

6 おわりに

変換の過程で用いられる知識を獲得するために変換履歴を定義し、これを一般化し変換履歴スキーマを自動的に作成する手法を与えた。一般化のために、標準履歴を導入し、二階のパターンマッチングを拡張した。一般化により、プログラミングの知識を抽出した変換履歴スキーマが作成できた。この変換履歴スキーマは Heute 等 [2] の変換スキーマよりも表現力が高い。この手法をインプリメントした実験システムによって、「終端再帰」「二分法」などのプログラミングの知識を獲得できた。

しかし、例えば二分法のスキーマ (図 6) を再利用したとき、SV_4 に適当な関数を対話的に与えなくてはならないが、どのような関数を与えるべきかの指針は与えられない。以降の変換との関係からどのような関数を与えるべきかの指針を示すことができれば、再利用によるプログラミングがより簡単になる。これは今後の課題である。

参考文献

[1] 小堀賢司, “変換履歴の再利用による高レベルのプログラム変換”, 第 44 回情処全国大会 2J-7, 1992.

[2] G. Heute and B. Lang, “Proving and Applying Program Transformations Expressed with Second-Order Patterns”, *Acta Infomatica*, 11, pp. 31–55, 1978.

[3] A. Goldberg “Reusing Software Developments”, *Proceeding of the Forth ACM SIGSOFT symposium on Software Development Environment*, 15, pp. 107–119, 1990.

[4] N. Dershowitz, “Program Abstraction and Instantiation”, *ACM Trans. Programming Languages and Systems*, 7, No. 3, pp. 446–477, July 1985.

付録

[定義 4] LCPF(R, R′)

- R = [定数 (変数)] R′ = [定数 (変数)] のとき R′
- $R = [(R_1 \cdots R_n)]$ $R' = [(R'_1 \cdots R'_n)]$ のとき $[(\mathrm{LCPF}(R_1, R'_1) \cdots \mathrm{LCPF}(R_n, R'_n))]$
- $R = [R_1\ r_1 \cdots R_n]$ $R' = [R'_1\ r'_1 \cdots R'_{n'}]$ のとき $R_n = R'_{n'}$ かつ $r_{n-1} = r'_{n'-1}$ であれば $[\mathrm{LCPF}([R_1 \cdots R_{n-1}], [R'_1 \cdots R'_{n'-1}])\ r_{n-1}\ R_n]$ そうでなければ $\mathrm{LCPF}(R_n, R'_{n'})$