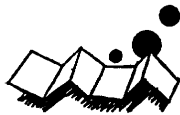
解 説

2. 方 式



# 2.4 プログラム変換による自動プログラミング†

吉 田 紀 彦††

## 1. はじめに

自動プログラミングのうちでも特に活発に研究が進められている分野の一つに，形式的仕様からプログラムを導出する過程の自動化がある．そのために用いられる手法として演繹推論によるもの，知識処理手法によるもの，部分合成によるものなどがあり，プログラム変換もその一つである．

プログラム変換とはプログラムをその意味を保存しつつなんらかの規則に則って書き換える手法をいう．ここで最初のプログラムが十分に抽象度の高いものであればそれを形式的仕様とみなすこともでき，この手法は仕様からプログラムを導出するものとなり得る．プログラム変換による自動プログラミングのイメージを，ごく簡単な例[1]を用いて示す．数値 $X$ と数値リスト $L$ について $X$ が $L$ のすべての要素より小さいか否かを判定するプログラムを考える．

lessall($X$, $L$) <= compute $X$ < all($L$)
where $X$ is a number and
$L$ is a list of numbers.

これは「何を」なすべきかを記述しているに過ぎない．ここで，*all* に関して次の二つの書換え規則が用意されているものとする．

$P$(all($L$))⇒true, if $L$ is the empty list.
$P$(all($L$))⇒$P$(head($L$)) and $P$(all(tail($L$))),
if $L$ is a non-empty list.

ここでは（関数を値としてもつ）2階変数 $P$ に "$X$<" を対応付けることができる．$L$ が空か否かが不明なので，とりあえず条件節の形を導入した上でこれらを適用する．

lessall($X$, $L$) <= if empty($L$)
then true
else $X$ < head($L$)
and $X$ < all(tail($L$))

*and* 以下は最初のプログラム本体の具体形になっているので，再帰呼び出しの形を導入することができる．

lessall($X$, $L$) <= if empty($L$)
then true
else $X$ < head($L$)
and lessall($X$,tail($L$))

これは最初のものと等価で，かつ「いかに」なすべきかを記述したプログラムになっている．

本稿はプログラム変換による自動プログラミングについて，その基本概念および代表的な研究例を紹介することを目的とする．プログラム変換に関しては，システムの網羅的かつ体系的なサーベイ[2]がすでに上梓されている．また，トピック的な解説[3)~6)]や，解説書[7]もある．ここでの解説は，自動プログラミングへの応用に焦点を当てるとともに，ある意味でそれらの文献を補完しようとするものである．したがって，参考文献も，それらが採り上げていないもの，ないし本文に直接関連するものや特に重要なものに限定している．プログラム変換に興味をもたれた場合には，上記の文献，さらにそこでの参考文献を適宜参照されることを薦める．

以下，まずプログラム変換の基本概念を解説した後，自動プログラミングへの応用について述べる．そして，代表的なシステム例をいくつか紹介する．最後に，将来へ向けての課題や展望について触れる．

## 2. プログラム変換

あるプログラムを別のプログラムに書き換えることがプログラム変換の基本である．しかし，書換えがプログラム変換として意味をもつためには，いくつかの要件がある．

変換前と後の二つのプログラムの間では意味が保存される．意味を保存しない書換えはプログラム変換で

† Automatic Programming with Program Transformation by Norihiko YOSHIDA (Department of Computer Science and Communication Engineering, Kyushu University).

†† 九州大学工学部情報工学科

はない．意味とは簡単に言えば機能や入出力間関係のことであるが，厳密にはなんらかのセマンティクスが定義されている必要がある．一方，変換前と後の間では構造が変化する．プログラムの構造をなんらかの評価基準の下で「望ましい」方向に向けて変化させることが，プログラム変換の目的である．この向きは，ほとんどの場合，抽象度の低下ないし実行効率の向上である．なお，一般には意味と同時に表現形式も保存される．コンパイルをプログラム変換の範疇に含めることはほとんどない．

変換はあらかじめ用意されている規則群に則ってなされる．ad hoc な書換えはプログラム変換ではない．おのおのの変換規則は，あるプログラム構造から別の等価なプログラム構造への写像を規定するものである．プログラム変換の具体的な手続きは，与えられたプログラムに変換規則を順次適用していくことである(**図-1**)．プログラムを「望ましい」方向に変換するためには，適切な部分に適切な規則を適切な順序で用いる必要がある．問題に応じて規則の適用方針を決定するものが変換戦略および戦術である．

プログラム変換にはさまざまな方式がある．それらを特徴付ける最大の要因は，変換規則の性質および形式である．変換規則の性質は汎用的なものと個別的なものに大別される．また，その形式は内包的（操作としての規則）なものと外延的（図式としての規則）なものに大別される．組合せは4通り考えられるが，汎用的規則は内包的であり，個別的規則は多くが外延的である．

以下，2種類の変換規則，および変換戦略と戦術について，項を改めて詳しく述べる．

### 2.1 汎用的規則による変換

これは汎用的な少数の変換規則を用意し，それらを結合して問題に応じた変換手続きを構成しようとするものである．個々の変換規則は全く機械的な書換え規則であり，問題への適合性といった評価基準を一切ともなわない．汎用的規則は，規則の一般性を確保するために，内包的形式を取る．これは変換規則を，プログラム構造を書き換える操作として定義する形式である．

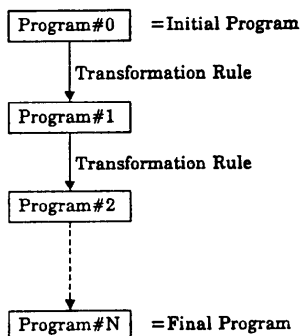



**図-1　プログラム変換の手続き**

汎用的（内包的）規則の代表例として，いわゆる unfold/fold 方式[8] がある．これは次の6つの変換規則から構成される．なお，対象として関数型プログラムを想定している．

① 定義 (definition)：新たな関数の定義を導入する．

② 具体化 (instantiation)：関数に現れる変数を具体的な値で置き換える．

③ 展開 (unfolding)：関数 $E \Leftarrow E'$ と $F \Leftarrow F'$ について $F'$ 内に $E$ の具体形が現れる場合に，それを対応する $E'$ で置き換えて $F \Leftarrow F''$ を得る．これはいわゆる「手続きのインライン展開」に等しい．

④ 畳み込み (folding)：関数 $E \Leftarrow E'$ と $F \Leftarrow F'$ について $F'$ 内に $E'$ の具体形が現れる場合に，それを対応する $E$ で置き換えて $F \Leftarrow F''$ を得る．これは展開と逆の操作である．

⑤ 抽象化 (abstraction)：関数の右辺（本体）に現れる関数を変数で置き換える．

⑥ 法則適用 (laws)：関数の右辺に可換則や結合則などの基本的法則を適用する．

実はこれら以外に暗黙のうちに用いられる規則として，「不要な関数の削除」がある．

unfold／fold 方式による変換の例は**図-2** に示されている（::はリスト構成子で，N::L は Lisp で言う cons (N L) のこと，*succ* は次の自然数を与える演算子）．

厳密に言うと，畳み込みは等価変換ではない．プログラムの停止性を保存しないからである．たとえば，関数を自分自身で畳み込むと次が得られる．

$$f(X) \Leftarrow f(X)$$

このプログラムは明らかに停止しない．停止性を保存するように規則を改めることも可能ではあるが，この方式の提案者は規則の厳密な正当性よりもその単純性を重視し，停止性の検証を人間に委ねている．

unfold／fold 方式は，規則構成を少し変更することにより，論理型プログラムを対象とすることも可能となる[9]．なお，論理型プログラムの変換について，そのプログラム導出への親和性も指摘されている[10]．

汎用的規則による変換は個々の規則が単純なため，理論的扱いが容易である．しかし，その上の戦術や戦略なしに実際の問題に応用することはできない．

### 2.2 個別的規則による変換

これはプログラミングのさまざまな局面ごとに個別の規則を用意し，問題に対処しようとするものである．規則にはプログラミング一般に関するもの，問題領域に関するものなどがあり，数百個以上用意されるのが普通である．個別的規則は，規則群の拡張性を確保するために，多くの場合外延的形式を取る．これは変換規則を，二つのプログラム構造に対応付ける図式の組として定義する形式である．

プログラムの図式とはその構造を抽象化したものであり，いわゆる２階変数を含む．外延的規則は<入力図式，出力図式，適用条件図式>の３要素から構成される．**1.** で取り上げた *all* に関する規則はそのごく簡単な例である．なお，外延的規則を適用する際には２階の照合アルゴリズムが必要となる[11]．

個別的規則としてはたとえば次のようなものがあり，その平板な（内部構造をもたない）集合としての規則群はいわゆる知識ベースに非常に近いものとなる．

・(プログラムの変換) ある種の再帰呼び出しがあったら，それを等価なループに書き換える．

・(データ構造の変換) 集合があったら，それを線形リストに写像する．

個別的規則による変換は規則が個々の局面に直接対応するため，問題領域に応じた変換も戦術や戦略なしにある程度行うことができる．しかし，多数の規則を用意しなければならず，どれだけあれば十分か，どのように収集すれば良いかなども解明されていない．また，理論的扱いも難しい．

### 2.3 変換戦略と戦術

変換戦略および戦術は，変換規則よりも広い視野，高い位置から規則適用の方針や必要な補助プログラムの導入を決定するものである．特に汎用的変換規則を採用する場合，個々の規則はプログラムの局所部分を機械的に書き換えるに過ぎず，規則のみで変換を「望ましい」方向に向けることはできない．たとえば，unfold／fold 方式では次の変換も可能である．

$$f(N) \Leftarrow N$$

$$\Downarrow$$

$$f(0) \Leftarrow 0$$

$$f(\mathrm{succ}\ N) \Leftarrow \mathrm{succ}\ f(N)$$

この変換は明らかな効率低下と無用の複雑化をもたらし，ほとんどの場合有害である．ちなみに，unfold／fold 方式ではこれの逆変換ができない．なぜならば，再帰的プログラムでいう基底形と再帰形しか導き出せないからである（基本的に，展開の連鎖によって基底形を，畳み込みによって再帰形を導き出す）．これがこの方式の大きな弱点である．

戦略および戦術にはプログラミング一般に関するもの，問題領域に関するものなどがある．問題がごく小さい場合には，これらを ad hoc ないし試行錯誤的に与えることも不可能ではない．しかし，一般の問題に対処するためにはこれらの整備と体系化が重要である．

実用規模プログラムの導出／効率化に有用な単純かつ汎用の戦略として，次のものがある[12]．すなわち，できるだけ細かい独立な部分に分割してプログラムを

```
sum(N :: L)       <= N + sum(L)
squares(N :: L) <= (N * N) :: squares(L)
sumsquares(L)     <= sum(squares(L))

sumsquares(N :: L)
        <= sum(squares(N :: L))                     <具体化>
        <= sum((N * N) :: squares(L))               <展開>
        <= (N * N) + sum(squares(L))                <展開>
        <= (N * N) + sumsquares(L)                  <畳み込み>
```

(a) 関数結合の除去

```
sum(N :: L)       <= N + sum(L)
product(N :: L) <= N * product(L)
sumproduct(L)     <= <sum(L), product(L)>

sumproduct(N :: L)
        <= <sum(N :: L), product(N :: L)>           <具体化>
        <= <N + sum(L), N * product(L)>             <展開>
        <= <N + U, N * V>
           where <U, V> == <sum(L), product(L)>     <抽象化>
        <= <N + U, N * V>
           where <U, V> == sumproduct(L)            <畳み込み>
```

(b) ループの併合

```
factorial(succ N) <= (succ N) * factorial(N)

factoriall(N, U)  <= U * factorial(N)               <定義>
factoriall(succ N, U)
        <= U * factorial(succ N)                    <具体化>
        <= U * ((succ N) * factorial(N))            <展開>
        <= (U * (succ N)) * factorial(N)            <結合則>
        <= factoriall(N, U * (succ N))              <畳み込み>
```

**図-2** 変換戦術の unfold/fold 方式における適用例

記述しておき，それらを併合する方向に変換を施して，問題に応じたプログラムを得る．この戦略を支援する戦術が次の三つである[12]．なお，これらの戦術のごく簡単な適用例を図-2 に示す．

① 関数結合の除去（パイプライン化）：複数のプログラムの結合構造（例における *sum* と *squares* からなる *sumsquares*）を一つに融合する．これは生産者ループ（*sum*）と消費者ループ（*squares*）のパイプライン化に等しい．

② ループの併合（疑似並列化）：同じ入力データを扱う複数のプログラム（例における *sum* と *product*）を一つに併合する．

③ プログラムの一般化：プログラムを，それを特殊な場合として含むようなより一般的なもの（例における *factorial* に対する *factorial* 1）で置き換える．すなわち，引数の追加，定義域／領域の拡張などを行う．

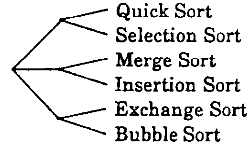
これらの戦術のごく簡単な適用例を図-2 に示す．この戦略および戦術を実用規模の問題に適用した例として，roff 型文書フォーマッタの導出[13]，Algol 系言語コンパイラの効率化[14] がある．

一方，より問題を特定化した戦略や戦術も，再帰呼び出しの効率化，プログラムの特殊化，非決定性の減少，分割統治法などに関して，提案がなされている[6),15),16)]．

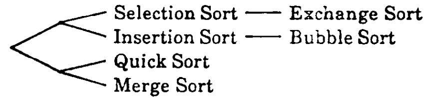
## 3. 変換によるプログラムの導出

プログラム変換による自動プログラミングとは，変換によるプログラム導出の自動化である．この応用の動機は次のような知見にある．すなわち，プログラムの明快さと効率とはしばしば相反するものであり，プログラミングの困難さは両者を同時に追求するところから生じる．明快さのみを意識してプログラムを記述することは容易なはずであり，機械的な変換によってその効率を改善することは自動化できるはずである．

変換前のプログラムが十分に抽象度の高いものであれば，その変換を導出とみなすこともできる，と **1.** で述べた．抽象度が高いとは，明快であると同時に，全くないし絶望的な効率でしか実行できないことを一般に意味する．しかし，そのプログラムはある処理系ではなんとか実行できるかも知れない．すなわち，プログラム導出のための変換は，プログラム効率化のための変換と本質的になんの相違もない．ある変換例を導出とみなすか効率化とみなすかは，多分に恣意的である．




(a) Darlington(と Clark)の結果




(b) Green と Barstow の結果

図-3 ソート・アルゴリズムの導出/分類

プログラム導出の最も洗練された形はアルゴリズム導出である．これについては，各種ソート・アルゴリズムの導出／分類実験が有名である．ソートの形式的仕様を次のように設定する．すなわち，与えられたリスト要素を並べるすべての組合せのうちから昇順のものを選び出すことを考える．これに対する変換手続きを選択肢ごとに分岐させることにより，一つの仕様からさまざまなソート・アルゴリズムを導出できる．さらに，その変換木の形状に従ってソート・アルゴリズムを分類できる．変換木は，用いる戦略により異なる形状をとる．Darlington（と Clark）の結果[17),18)]，および Green と Barstow の結果[19),20)] を**図-3** に示す．この例は問題が非常に限定されていて実験の域を出ない．また，高度な変換を含むため自動化は難しい．

より実際的で，かつ十分にプログラム導出と認め得る変換としては，たとえば次のものがある．

① 実行不能仕様の変換，特に陰関数の陽関数化[3)]：実行不能仕様として多く採り上げられるのは集合表現と陰関数表現である（ただし，前者については SETL や KRC など，これを直接実行できる言語も現れてきている）．陰関数表現は一般に実行不能であって，変換によってこれと等価でかつ実行可能な陽関数表現を導出することができる．

② 抽象データ型の具体化，および副作用の導入[3),21),22)]：抽象的で実現方法を意識しないデータ型があった場合に，変換によってそれからより具体的で直接実現できるデータ型を導出することができる．次のような例がある．

- リスト→ベクタ
- キュー→巡回バッファ，ベクタ
- 優先度付きキュー→2 進木

```
eqnlist.prexp
APPEND([], Y) <= Y
APPEND(A :: X, Y) <= A :: APPEND(X, Y)
APPEND3(X, Y, Z) <= APPEND(APPEND(X, Y), Z)
start
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST
append3(nil, y, z) end
APPEND3([], Y, Z) <= APPEND(Y, Z)
NOW TYPE IN RECURSIVE BRANCHES
append3(a :: x, y, z) end
HOW DOES
APPEND3(A :: X, Y, Z) <= A :: APPEND3(X, Y, Z)
LOOK TO YOU
ok
FINISHED
prexp
APPEND3([], Y, Z) <= APPEND(Y, Z)
APPEND3(A :: X, Y, Z) <= A :: APPEND3(X, Y, Z)
APPEND([], Y) <= Y
APPEND(A :: X, Y) <= A :: APPEND(X, Y)
```

**図-4** Darlington のシステムの実行例[23]

さらに，メモリ・セルに相当する変数を追加することにより，副作用（代入／参照）を導入することができる．

## 4. 代表的なシステム

プログラム変換の自動化を図ったシステムは数多く作成されている[2]．それらはその目的と性質によって二つに大別される．一つは研究指向のものであり，もう一つは実用指向のものである．

### 4.1 研究指向システム

これは純粋にプログラム変換の自動化の可能性を追究するためのシステムであり，実用性は考慮の対象ではない．かなり高度な変換を実行するシステムも存在するが，問題は微少なものに限られる．これは，問題が大きくなると試行錯誤が爆発的に増加するからである．また，システムを操作するに際しては，その内部動作や変換過程についての深い理解がユーザに要求される．

システムの例を紹介する．

（1） Darlington のシステム

これは unfold/fold 方式の自動化を目的とするシステムである．三つのリストを結合する関数 *append 3* の効率化の実行[23] を**図-4** に示す（小文字が入力，大文字が応答）．このシステムは「強制畳み込み（forced folding)」と呼ばれる変換機能をもつ．すなわち，畳み込みに失敗した場合の関数形の相違に着目して適切な補助関数を自動的に導入する．

（2） DEDALUS

Manna と Waldinger の DEDALUS は，個別的規則を採用したシステムであり，100 以上の規則を有する．**1.** で示した *lessall* の変換[1] は，このシステムの最も簡単な実行例である．

### 4.2 実用指向システム

これは自動プログラミングの実現を目指すためのシステムであり，あくまでもその手段としてプログラム変換を用いているに過ぎない．そして，おおむね次のような傾向をもつ．

・必ずしも完全な自動化を図ってはいない．ユーザとシステムの協同形態には，ユーザがシステムを主導する，ユーザがシステムに示唆を与える，などさまざまなレベルがある．

・手段をプログラム変換に限定せず，演繹推論によるもの，知識処理手法によるもの，部品合成によるものなど，他の手法との有機的結合を図っている．

・仕様からのプログラム導出だけでなく，形式的仕様の生成やプログラムの評価なども含む，プログラム開発過程の全般的な自動化を図っている．

システムの例を紹介する．

（1） ZAP

Feather の ZAP は，unfold/fold 方式の実用規模プログラムへの応用を目的とするシステムであり，研究指向の色彩が強い．特徴として，folding の機能をもたず，その代わりに変換後の予想されるプログラム図式をユーザが指示する．また，対話形式でなく，メタ・プログラム形式を採る．関数結合の除去を行うメ

```
start
def
var  N : num
var  L : list num
+++  sum(list num) <= num
---  sum(nil) <= 0
---  sum(N :: L) <= N + sum(L)
+++  squares(list num) <= list num
---  squares(nil) <= nil
---  squares(N :: L) <= (N * N) :: squares(L)
+++  sumsquares(list num) <= num
---  sumsquares(L) <= sum(squares(L))
end
context
  unfold   sumsquares sum squares
  using    sumsquares
  lemmas   commutative +
  transform
    goal   sumsquares(nil)
    goal   sumsquares(N :: L) <= $$(N, sumsquares(L))
  end
  delete   sumsquares(L)
end
stop
```

**図-5** ZAP のメタ・プログラム例[13]

タ・プログラム[13]を図-5に示す（*def* に続くのが対象プログラム，*context* に続くのが変換指示）．図中の *goal* で始まる行が *$$* を2階変数とする図式である．このシステムは図式記述，変換手続き記述の抽象化を目指している．しかし，十分達成されているとは言い難く，たとえば，別個のプログラムに同様な変換手続きを適用したい場合にも，メタ・プログラムを個別に記述しなければならない．

（2） CIP

Bauer らの CIP は，形式的仕様からアセンブリ言語までのレベルすべてを包括する言語を用いて，そのプログラム変換を基礎にプログラム開発の体系化を目指すプロジェクトである．変換規則は個別的であり，手続き型プログラムに対する規則なども含まれる[24]．なお，言語は CIP-L，変換システムは CIP-S と呼ばれる．このシステムは個々の規則の機械的な適用を代行するのみであり，単なる「工具」に過ぎない．

（3） PSI, CHI

Green らの PSI は，ソフトウェア開発の全般的な支援を目的とするシステムであり，仕様獲得部とプログラム合成部の二つから構成される．プログラム合成部はさらに，コーディング・エキスパート PECOS[25] と効率エキスパート LIBRA の二つを含む．この二つはいずれも知識ベースシステムであり（すなわち，個別的変換規則を採用している），協同して変換を行う．すなわち，変換に際して PECOS が複数の選択肢を提示すると，LIBRA が実行時間と必要領域の両面から個々の効率を評価して示唆を与える．

CHI は PSI の後継システムであり，より広範囲かつ統合的な支援を目的としている．知識の獲得による規則群の改良／拡張や一貫性管理などもその機能に含まれる．システムは現在その一部が完成している[26]．特に変換規則のコンパイ

**Rule Compiler Rules**

$$a{:}`P \rightarrow Q' \wedge \mathit{FreeVars}(P)=S_0 \wedge \mathit{FreeVars}(Q)-S_0=S_1 \\ \rightarrow a{:}`\mathit{Satisfy}(\forall S_0[P \Rightarrow \exists S_1[Q]])'$$

$$a{:}`\forall S[C \Rightarrow Q]' \wedge y \in \mathit{conjuncts}(C) \wedge \mathit{NoVarsOf}(y,S) \\ \rightarrow a{:}`C_1 \Rightarrow \forall S[C \Rightarrow Q]' \wedge \mathit{class}(C_1)=\mathrm{conjunction} \\ \wedge y \in \mathit{conjuncts}(C_1) \wedge y \notin \mathit{conjuncts}(C)$$

$$a{:}`\forall S_0[C \Rightarrow Q]' \wedge y \in \mathit{conjuncts}(C) \wedge y{:}`x=t' \wedge x \in S_0 \wedge \mathit{NoVarsOf}(t,S_0) \\ \rightarrow a{:}`\forall S_0[C \Rightarrow Q]/S_1' \wedge z \in S_1 \wedge z{:}`x/t' \wedge x \notin S_0 \\ \wedge y \notin \mathit{conjuncts}(C)$$

$$y{:}`\forall S_0[C \Rightarrow Q]' \wedge a \in \mathit{conjuncts}(C) \wedge a{:}`x \in t' \wedge x \in S_0 \wedge \mathit{NoVarsOf}(t,S_0) \\ \rightarrow y{:}`\forall x[x \in t \Rightarrow [\forall S_0[C \Rightarrow Q]]' \wedge x \notin S_0 \wedge a \notin \mathit{conjuncts}(C)$$

$$a{:}`\mathit{Satisfy}(C \Rightarrow R)' \rightarrow a{:}`\mathit{if}\ \mathit{Test}(C)\ \mathit{then}\ \mathit{Satisfy}(R)'$$

$$a{:}`\mathit{Satisfy}(P/S_0)' \wedge y \in S_0 \wedge y{:}`x/t' \\ \rightarrow a{:}`\mathit{bind}\ S_1\ \mathit{do}\ \mathit{Satisfy}(P)' \wedge z \in S_1 \wedge z{:}`x \leftarrow t'$$

$$a{:}`\mathit{Satisfy}(\exists S_0[P])' \wedge y \in S_0 \\ \rightarrow a{:}`\mathit{bind}\ S_1\ \mathit{do}\ \mathit{Satisfy}(P)' \wedge x \in S_1 \wedge x{:}`y \leftarrow (\mathit{NewObject})'$$

$$a{:}`\mathit{Satisfy}(C)' \wedge \mathit{class}(C)=\mathrm{and} \wedge P \in \mathit{conjuncts}(C) \\ \rightarrow \mathit{class}(a)=\mathrm{block} \wedge Q \in \mathit{steps}(a) \wedge Q{:}`\mathit{Satisfy}(P)'$$

$$a{:}`\mathit{Satisfy}(\forall x[x \in S \Rightarrow R])' \rightarrow a{:}`\mathit{enumerate}\ x\ \mathit{in}\ S\ \mathit{do}\ \mathit{Satisfy}(R)'$$

$$a{:}`f(u)' \rightarrow a{:}`(\mathit{GetMap}\ u\ f)'$$

$$a{:}`\mathit{Satisfy}(f(u)=v)' \rightarrow a{:}`(\mathit{ExtendMap}\ u\ f\ v)'$$

$$a{:}`\mathit{Satisfy}(p(u))' \rightarrow a{:}`(\mathit{ExtendMap}\ u\ p\ \mathit{True})'$$

$$a{:}`(\mathit{ExtendMap}\ u\ \mathrm{class}\ v)' \rightarrow a{:}`(\mathit{Ttransform}\ u\ v)'$$

$$a{:}`(\mathit{ExtendMap}\ u\ \mathrm{class}\ v)' \wedge a \in \mathit{steps}(P) \wedge x{:}`u \leftarrow (\mathit{NewObject})' \\ \rightarrow x{:}`u \leftarrow (\mathit{Tinstance}\ v)' \wedge a \notin \mathit{steps}(P)$$

$$\mathit{class}(a)=\mathrm{conjunction} \wedge \mathit{class}(C)=\mathrm{conjunction} \\ \wedge a \in \mathit{conjuncts}(C) \wedge P \in \mathit{conjuncts}(a) \\ \rightarrow P \in \mathit{conjuncts}(C) \wedge a \notin \mathit{conjuncts}(C)$$

$$a{:}`C \Rightarrow Q' \wedge \mathit{Null}(\mathit{conjuncts}(C)) \rightarrow \mathit{Replace}(a,Q)$$

**図-6** CHI の規則コンパイラ[27]

```
T1  : Define new relation ROUTER_NETWORK * (location, location)
T2  : Define code to 1) compute the transitive closure of the
router network, and 2) explicitly store each connection as a
ROUTER_NETWORK * relation
T3  : Replace each reference to LOCATION_ON_ROUTE_TO_BIN with
reference to ROUTER_NETWORK *
T4  : Remove LOCATION_ON_ROUTE_TO_BIN from program
  :
  :
T10 : Simplify expression (not not p => p)
T11 : Simplify expression (p and p => p)
T12 : Define new relation PACKAGE_LIST (packages, switch)
T13 : Define maintenance code for PACKAGE_LIST : add-package,
delete-package
T14 : Insert maintenance code at point where package destination
is created
T15 : Verify package entrance to router is equivalent to package
creation
T16 : Verify no change to PACKAGES_DUE_AT_SWITCH between package
creation and entrance
T17 : Move maintenance code from creation code to entry code
  :
  :
T30 : Define array of linked-lists, one for each switch
T31 : Replace relation access with linked-list access
```

図-7 GLITTER の変換手続き例[29]

ラはわずか十数個の変換規則で記述されているので[27),28)], **図-6** にそれを示す.

（4） SAFE/TI

Balzer らの SAFE/TI は PSI と目的を同じくするプロジェクトであり，SAFE が仕様獲得部，TI がプログラム合成部である．TI の目標は，仕様記述言語に自然言語と同等の表現力をもたせ，その形式的仕様から手続き型プログラムを変換によって導出することにある．変換の部分的な自動化を目指したシステムが GLITTER と呼ばれる知識ベースシステムであり，エキスパート・シェル Hearsay-III の上に作成されている．小包配送機プログラムの導出におけるこのシステムの変換手続きの一部[29]を図-7 に示す．なお，プロジェクト全体でどこまでシステム化が実現しているかは明確でない．

## 5. 課題と展望

自動プログラミングを実現する手法としての期待に対して，プログラム変換はある程度答えている．しかし，いまだ実験の域を越えてはいない．実用規模プログラムの導出にしても，個別の問題について発見的に成功しているに過ぎない．プログラム変換の実用化を達成する上では，①変換知識の獲得／構成法の確立，および②変換手続きの形式化と体系化，の二つが特に重要な課題である．

変換規則や戦術，戦略を作成するのに必要な知識について，十分な収集と体系化がなされているとは言い難い．自動プログラミングにおいては，対象分野，対象言語，対象マシン，プログラミング，数学，問題解決法，などに関するさまざまな知識が要求される．自動化のためには対象分野の特定および分野固有の知識の獲得が最も重要，との指摘もある[30]．知識の獲得／構成法の確立が必要である．その際に，知識工学の成果が大いに参考となるはずである．一方，プログラム変換の方式は，個別的規則の平板な集合から，汎用的規則とその上の戦術や戦略から成る階層的，体系的構成に移行すべきであり，実際その傾向にある．そのためには，変換手続き，すなわちプログラムのどの部分にどの規則をどの順序で用いるかの手順，の形式化と体系化が必要である[13),31)~35)]．変換手続き記述用のメタ言語も，Hope[32] などいくつか提案されている．

プログラム変換の研究のうちで今後の展開が最も期待されるものの一つに，並列処理プログラムの導出がある．逐次処理という枷を取り払うと同時に計算機の性能向上を目指すべく，並列処理の研究がソフトウェア，ハードウェアの両面から活発化している．プログラム導出において，並列処理は逐次処理とは全く異質の戦術，戦略に従う可能性がある．研究の萌芽[3),36)]に続いて，プロセスの融合／分離[37),38)]，細粒度高並列処理の導出[39),40)] などの研究例がある．しかし，同期や非決定性のセマンティクスの扱いなど，解決すべき課題も多い．

## 6. おわりに

自動プログラミングのための形式的手法として，プログラム変換は最も実用化に近いところにあると言える．プログラム変換に基づいてプログラミング方法論を構築しようという提案もある[41]．プログラム構造の抽象化という観点から，プログラム変換がアルゴリズム，プログラムについて新たな洞察を与えることも期待されている．このように，プログラム変換は研究，実用の両面においていまだ大きな可能性を残している．

## 参考文献

1) Manna, Z. and Waldinger, R. J.: Synthesis: Dreams => Programs, IEEE Trans. Softw. Eng., Vol. SF-5, No. 4, pp. 294-328 (1979).

2) Partsch, H. and Steinbrüggen, R.: Program Transformation Systems, ACM Comput. Surv., Vol. 15, No. 3, pp. 139-236 (1983); 玉木訳：プログラム変換システム, bit 別冊コンピュータ・サイエンス, pp. 74-106 (1984).

3) Darlington, J.: Program Transformation, *Functional Programming and Its Application* (Darlington, J. et al., eds.), Cambridge Univ. Press, pp. 193-215 (1982).

4) Darlington, J.: Program Transformation, BYTE, Vol. 10, No. 8, pp. 201-216 (1985).

5) 黒川, 外山：プログラム変換を論じる, bit, Vol. 16, No. 10, pp. 83-98 (1984).

6) 玉木：プログラム変換の戦略とその自動化, bit, Vol. 17, No. 12, pp. 28-34 (1985).

7) 古川, 溝口（編）：プログラム変換, 共立出版 (1987).

8) Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, J. ACM, Vol. 24, No. 1, pp. 44-67 (1977).

9) Tamaki, H. and Sato, T.: Unfold/Fold Transformation of Logic Programs, Proc. 2nd Logic Programming Conf., pp. 127-138, Uppsala (1984).

10) Hogger, C. J.: Derivation of Logic Programs, J. ACM, Vol. 28, No. 2, pp. 372-392 (1981).

11) Huet, G. and Lang, B.: Proving and Applying Program Transformation Expressed with Second-Order Patterns, Acta Inf., Vol. 11, pp. 31-55 (1978).

12) Feather, M. S.: A System for Assisting Program Transformation, ACM Trans. Prog. Lang. Syst., Vol. 4, No. 1, pp. 1-20 (1982).

13) Feather, M. S.: A System for Developing Programs by Transformation, Ph. D. Thesis, Univ. of Edinburgh (1979).

14) 吉田：実用規模プログラムの変換試行事例, プログラム変換（古川, 溝口編）第9章, 共立出版 (1987).

15) Smith, D. R.: The Design of Divide and Conquer Algorithms, Sci. Comput. Prog., Vol. 5, pp. 37-58 (1985).

16) Smith, D. R.: Top-Down Synthesis of Divide-and-Conquer Algorithms, Artif. Intell., Vol. 27, pp. 43-96 (1985).

17) Darlington, J.: A Synthesis of Several Sorting Algorithms, Acta Inf., Vol. 11, pp. 1-30 (1978).

18) Clark, K. L. and Darlington, J.: Algorithm Classification through Synthesis, Comput. J., Vol. 23, No. 1, pp. 61-65 (1980).

19) Green, C. C and Barstow, D. R.: On Program Synthesis Knowledge, Artif. Intell., Vol. 10, pp. 241-279 (1978).

20) Barstow, D. R.: Remarks on "A Synthesis of Several Sorting Algorithms" by John Darlington, Acta Inf., Vol. 13, pp. 225-227 (1980).

21) Darlington, J.: The Synthesis of Implementation for Abstract Data Types, Research Report DOC80/4, Imperial College (1980).

22) Moor, I. W. and Darlington, J.: Formal Synthesis of an Efficient Implementation for an Abstract Data Type, internal report, Imperial College (1981).

23) Darlington, J.: An Experimental Program Transformation and Synthesis System, Artif. Intell., Vol. 16, pp. 1-46 (1981).

24) Broy, M. and Pepper, P.: Program Development as a Formal Activity, IEEE Trans. Softw. Eng., Vol. SE-7, No. 1, pp. 14-22 (1981).

25) Barstow, D. R.: *Knowledge-Based Program Construction*, Elsevier North-Holland (1979).

26) Smith, D. R., Kotik, G. B. and Westfold, S. J.: Research on Knowledge-Based Software Environments at Kestrel Institute, IEEE Trans. Softw. Eng., Vol. SE-11, No. 11, pp. 1278-1295 (1985).

27) Green, C. et al.: Research on Knowledge-Based Programming and Algorithm Design-1981, Tech. Report Kes. U. 81. 2, Kestrel Institute (1982).

28) Green, C. and Westfold, S. J.: Knowledge-Based Programming Self Applied, *Machine Intelligence 10* (Hayes et al., eds.), Ellis Horwood/Halsted Press (1982).

29) Fickas, S. F.: Automating the Transformational Development of Software, IEEE Trans. Softw. Eng., Vol. SE-11, No. 11, pp. 1268-1277 (1985).

30) Barstow, D. R.: A Perspective on Automatic Programming, Proc. 8th IJCAI, pp. 1170-1179, Kahlsruhe (1983).

31) Bibel, W.: Syntax-Directed, Semantics-Supported Program Synthesis, Artif. Intell., Vol. 14, pp. 243-261 (1980).

32) Darlington, J.: The Structured Description of Algorithm Derivations, *Algorithmic Languages* (Bakker, J. W. d. and Vliet, H. v., eds.), Elsevier North-Holland, pp. 221-250 (1981).

33) Kin, T. S. C.: High Level Description of Program Transformation, internal report, Imperial College (1981).

34) Partsch, H.: Structuring Transformational Developments: A Case Study Based on Earley's Recognizer, Sci. Comput. Prog., Vol. 4, pp. 17-44 (1984).

35) 中川，中村：Prolog 等価変換エディタと変換戦略，情報処理学会論文誌，Vol. 26, No. 5, pp. 905-912 (1985).

36) Bush, V. J. and Gurd, J. R.: Transforming Recursive Programs for Execution on Parallel Machines, *Functional Programming Languages and Computer Architecture* (Goos, G. and Hartmanis, J. eds.), Lec. Notes Comp. Sci. 201, Springer-Verlag (1985).

37) Furukawa, K. and Ueda, K.: GHC Process Fusion by Program Transformation, 日本ソフトウェア科学会第2回大会論文集, pp. 89-92 (1985).

38) 柴山：並列オブジェクト指向プログラムの変換，日本ソフトウェア科学会第3回大会論文集，pp. 161-164 (1986).

39) King, R. M. and Brown, T. C.: Research on Synthesis of Concurrent Computing Systems, Tech. Report KES. U. 82. 10, Kestrel Institute (1982).

40) Yoshida, N.: Transformational Derivation of Systolic Algorithms in Relational Representation, Tech. Report, Kyushu Univ., to appear (1987).

41) Scherlis, W. L. and Scott, D. S.: First Steps towards Inferential Programming, Proc. IFIP, pp. 199-212 (1983).

(昭和62年4月2日受付)