論　文

抽象アルゴリズムの記述と具体プログラムへの変換*

大　石　東　作**

Abstract

Our target in this paper is not the development of a new structured programming language but the establishment of a new methodology in software engineering.

The core of this methodology is both the description of abstract algorithms and its translation to concrete grograms.

These algorithms written by an Algol-like high level language "Metal" manipulate abstract data-structures. These structures are composed of a set of abstract data-units which have information about relations and properties of themselves.

Abstract algorithms can not be executed directly because of their abstract characteristic; so these are translated to concrete programs which manipulate concrete data-structures such as arrays, linked lists, etc. by A PROGRAM SYNTHESIZER (APS).

Transformation rules for APS are generated automatically by A DATA OPERATION GENERATOR (ADOG).

Some examples of the description and transformation of abstract algorithms are presented.

1. はじめに

ソフトウェアの生産性・信頼性の向上を目的として自動プログラミングの研究が行われている．

Biermann, A. W.[1]はこの研究を次の5種類に分類している．

i　高水準言語または目的別専用プログラムの生成
ii　例題からのプログラム合成
iii　形式的な入力・出力述語仕様による合成
iv　自然語によるコマンドの変換
v　知識にもとづく発見的プログラム合成

iの手法が従来のプログラム言語の単なる拡張であるなら，自動プログラミングの名に今日ではもはやふさわしくないかもしれない．しかし実用性の見地からは十分価値がある．ii，iiiの手法はその形式性のゆえにプログラムの正当性の検証が容易であるが，実用との間にはギャップがある．iv[2]の自然語によるものは一部で高く評価されているが，一般的なプログラム合成の点からは自然語そのものの採用の是非，実用性に問題が残る．vの手法は遠い未来においては有効であっても，現状ではその基盤も確立されておらず評価自体が時機尚早であろう．

高水準言語による方法は斬新ではないが堅実であるから，この方法をさらに検討しよう．Fortran を始祖とする高水準プログラム言語がソフトウェアの普遍性・生産性・信頼性の向上，文書管理・保守の容易さをもたらしたことは言を待たない．しかし，お仕着せのデータ型・構造や制御構造の使用を余儀なくされてプログラムを記述し，コンパイラによって一様な形式のみで目的プログラムに変換するのでは，各種の細部仕様に適合した目的プログラムを得ることはできない．またアルゴリズムをできるだけ普遍的に記述したい意図にも反する．またコンパイラを極度に汎用化することは，その処理の時間的・空間的な効率を著しく悪くする．ここに従来の高水準言語による方法の大きな限界がある．特に高水準言語によってシステム・プ

* The Description of Abstract Algorithms and Its Translation to Concrete Programs by Tosaku OISHI (Electrotechnical Laboratory).

** 電子技術総合研究所

ログラムを直接書き下すことの是非は議論のわかれるところである．上のような高水準言語による方法の欠点は，プログラム記述における高水準な形式と目的プログラムにおいて要求されるさまざまな具体性が，一つのプログラム言語システムとして無理に結合されている点にその根拠があると考えられる．現在開発中の各種の構造的プログラム言語システムにおいても，上の欠点を十分に解消しているとはいいがたい．

抽象アルゴリズムを記述し，状況に応じてプログラムへと変換する我々の方法では，手続きの記述においてまず抽象化をめざし，個々の問題や目的プログラムのレベルで要求される具体性を極力除去することが意図される．その後，プログラムへ変換する際にこれらの具体性に適応するのである．このように手続きの記述とプログラムへの変換を分離することによって，高水準言語による記述の抽象性・一般性という課題と目的別専用プログラムにおいて必要とされる具体性・個別性という互いに相反する課題を解決する方法の一つとなっている．手続き表現と抽象化の重視は，この方法の構造的プログラミングやデータ抽象化技法との親近性をしめしている．特にデータ抽象化技法におけるプログラムの階層化・モジュール化に対して抽象データ構造を鍵とする分割が一般的な方式の一つであることを示唆する．また具体プログラムへの変換は段階的詳細化の一部が機械化できることを意味する．このような定式化・機械化の可能性が，我々の方法を自動プログラミングの一段階として位置づけると信じる．

我々の方法は汎用問題解決システム (Meta-88)[2]のプログラム合成サブシステムにおいて実現される．現在中間言語による抽象アルゴリズムとプログラムの具体性についての仕様から，具体プログラムを合成するシステムが試作されている．

2. 抽象アルゴリズムと抽象単位データ[2,3]

抽象アルゴリズム*は，有限個の要素(抽象単位データの実現値）の集合からなる抽象データ構造を対象とする演算・操作および連結・条件判定・繰り返し等からなる手続き制御操作の有限個の列である．通常のアルゴリズムやプログラムとの異なりは，抽象化を極力はかっている点にある．特にデータとその操作の抽象化に重点がおかれる．より具体的には，整数・実数・文字等のデータ型やリスト・配列等の具体的なデータ構造に関知せず，もっぱら抽象単位データという一様な形式を操作する．

抽象アルゴリズムの記述には後述するように Metal 言語を用いることを原則としている．しかし我々の方法の主眼は，たんに新しい構造的プログラミング言語の開発にあるのではなく，抽象アルゴリズムの記述法とそれを具体プログラムに変換する方法を確立するところにある．

まずこの手法の根幹ともなる抽象単位データについて述べよう．

2.1 抽象単位データと基本関数

抽象単位データ (Abstract Data Unit) は抽象データ構造を構成する原始要素であり，次の情報からなる．

1. 抽象単位データ間の2項関係を表現する成分の順序集合 (Relations：以下では関係と略す)
2. 抽象単位データそれ自体の性質・属性を表現する成分の順序集合(Properties：以下では属性と略す)

これらを次のネストされた順序組で表現する．

$$((REL_1 \quad REL_2 \cdots\cdots REL_m) \quad PRO_1 \cdots\cdots PRO_n)$$

Relations　　Properties

この表現は抽象単位データの形式(Form)を定義し，これに名称をつけることができる．内部の各項は成分の名称となっている．各成分の個数は任意であり，関係と属性のいずれか一方の集合が空であってもかまわない．

上の形式を持つ抽象単位データの実現値 (Instance) の集合によって，抽象単位データが構成される．単位データの実現値の関係成分は，他の実現値をさす情報であり，形式の成分名称とともに二つの実現値間の2項関係を表現する．この関係成分は通常のポインタに相当すると考えられるが，具体データ構造に変換された際にもポインタとして存続するとは限らない．このために2項関係として抽象化されている．属性の各成分はいわゆる変数の値(Value) に相当し，ベクトルないしは構造（各成分のデータ型が一致しない）になっており，その実現値の性質・属性をあらわす．

抽象アルゴリズムおよび抽象単位データにおいては，変数のいわゆるデータ型にはこだわらない．特に属性の各成分については型自由 (Type Free) とし，それらの間の演算等は自明とする．データ型の指定が必要になるのは抽象アルゴリズムを具体プログラムに変換するときである．型自由を許すのは，抽象アルゴ

* 正確には停止性の保証されていない半アルゴリズムである．

リズムでは単位データの関係成分によって形成される抽象データ構造の形とか操作に焦点があり，属性は従属的に扱われるからである．

抽象単位データの集合によって，さまざまの抽象データ構造を表現できる．これらは従来のプログラムの操作対象であった配列および表，リスト，木，グラフ等に関係づけうるから対応する抽象アルゴリズムが記述可能になる．このことは，種々のデータ構造の処理が，単位データの操作として定式化できることを意味し，『手続き処理』という点から広い汎用性をもっている．

一例として名札付き2進木をとりあげる．これを有向グラフによれば **Fig. 1**(a) のように節は名札と順序をあらわし，弧には関係 LSON, RSON を弁別する2種類がある．名札について2項関係として表現したのが同図（b）である．抽象単位データによると，形式は

LNODE=((LSON RSON) LABEL NUMBER)

となる．関係は LSON, RSON, 属性が LABEL, NUMBER である． 実現値として表現したのが同図(c)であり，名札つき2進木をあらわす抽象データ構造となっている．

単位データによる表現はあくまでも抽象的であり，計算機上での実現法は多様である．すなわち単位データとその具体化における関係は1対多であり，個々の具体化には状況に応じた得失がある．これが具体データ構造の選択による最適化の根拠となる．

単位データを操作する抽象基本関数は大別すると以下の4種類が存在する*.

i 単位データの実現値を生成・消去する関数
CREATENODE, DELETENODE

ii 単位データの各成分を参照する関数
LSON, RSON, LABEL, NUMBER

iii 単位データの各成分を更新する関数
SETLSON, SETRSON 等

iv 単位データの状態を判定する述語関数
NULLNODE, EQUAL

例示した各関数名は，前述の（名札付き2進木）抽象単位データの例に対応するものである．また単位データの間の2項関係をたどる（一方向に限定される）の

* 抽象基本関数は，抽象単位データの形式によってのみ決定される．実現値の集合によって形成される抽象データ構造，たとえば2進木，有向グラフ等からは独立している．しかし単位データの関係・属性名の命名法は任意であるから，形成される抽象データ構造と関連させて，これらの関係・属性名ひいては基本関数名を命名することは，抽象アルゴリズムの理解を容易にするのに役立つ．

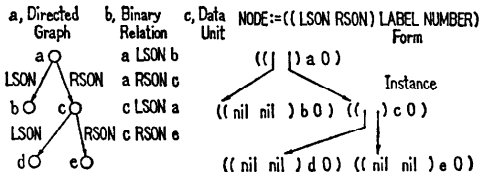

Fig. 1 Some representations of an abstract data structure.

は，iiの参照する関数を代入文の右辺とするか，関数呼び出しの実引数として使用することで可能となる．

2.2 抽象アルゴリズムの記述

抽象アルゴリズムは，高水準プログラム言語によって記述されているが，その抽象化は二つの方向になされている．第一は具体的な問題からの抽象化である．問題領域の異なる二つの問題に対しても，それらを支配する法則が同じであれば一つの抽象的な解法（抽象アルゴリズム）によって解決をはかる．また抽象アルゴリズムには本質的な部分のみが記述されているから，細部についての簡単な追加・修正によって適用範囲を広げることができる．第二はプログラムの具体性からの抽象化である．データの型や構造のように具体プログラムの実現 (Implement) に必要な仕様や最適化についての情報が抽象アルゴリズム中に記述されていないことは，具体プログラムへの変換時にかえって広範な具体性を付加する可能性をもたらす．

抽象アルゴリズムを記述する目的には，次の三つがある．第一に抽象的に記述することにより，アルゴリズムとしての本質を明らかにし，計算の複雑さについての抽象的な測度を定める．第二にアルゴリズム・ライブラリに登録・蓄積し，問題に応じて取り出しユーザの細い仕様に合せて修正し，機械的に具体プログラムへ変換可能とする．第三に正当性の保証されたアルゴリズムをユーザに提供する．正当性の検証自体も具体性を除去しているので容易になる．さらに具体プログラムへの変換過程の正当性を保証することによって，変換された複数種の具体プログラムの正当性を一般的に保証する．これは従来のように個々のプログラムをいちいち検証する方法とは異なる．このような目的と抽象化指向を達成することで，従来の *ad hoc* なプログラム作りから脱け出し，より普遍的なプログラム作成法の確立が我々の目標である．

抽象アルゴリズムにも記述言語が必要である．新たな内容をもりこむには，新しい言語が望ましい．

Algol 流言語 Metal[14]がそれである．しかし，現在

```
RECURSIVE INORDER NUMBERING METAPROCEDURE

LNODE :=  ( ( LSON RSON ) LABEL NUMBER )

begin
    COUNT := 1;
    INORDER ( ROOT )
end

metaprocedure INORDER ( NODE ):
  begin
      if  NODE = nilu then return;
      INORDER ( LSON ( NODE ) );
      NUMBER ( NODE ) := COUNT;
      COUNT := COUNT + 1 ;
      INORDER ( RSON ( NODE ))
  end
```

Fig. 2 An INORDER algorithm written by Metal.

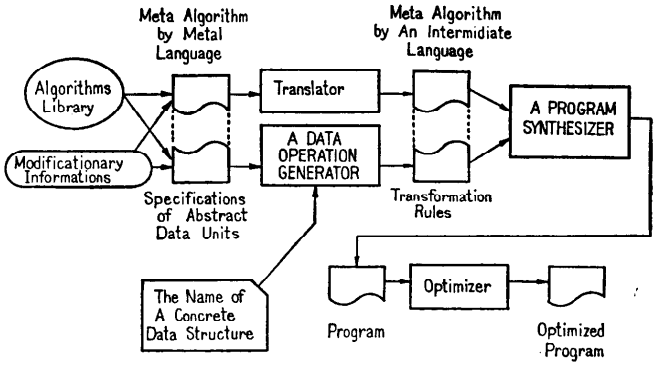

Fig. 3 The FURNANCE system.

のところその仕様には未確定の部分が残されている．この章の例題では Metal を先験的に用いるが，後の章では Lisp 1.6 を用いる．その使用理由はプログラム合成システムの親言語が Lisp であることにとどまらず，抽象アルゴリズムの手法が古い形式のプログラム言語にも適用できることをしめすためでもある．また Lisp が型自由であり再帰呼び出し機構を持つことは抽象アルゴリズムの記述に適合している．とはいえこの言語が構造的プログラミングに望ましいとされる各種の制御構造を持たず，その読みにくさもまた定評がある．しかし我々の手法の焦点はあくまでもデータと手続きの相互の対応を明らかにすることにあり，Lisp の使用は本質的な困難をもたらすものでない．これは経験的にも確められている．

抽象アルゴリズムの記述例として，これまでにも取り上げた名札つき2進木の各節に中順序（左木・親木・右木の順：inorder）で番号をふるアルゴリズム INORDER[6]を **Fig. 2** にしめす．番号を保持する大局変数 COUNT が，アルゴリズムの外で初期設定される．この手続きの評価に際して2進木の先頭が ROOT 引数によってしめされる．手続き自体は再帰形式で定義され，終了条件は未定義判定述語による．2進木の節が未定義すなわち2進木の葉より先をたどろうとするときに終了する．節が未定義でなければ，参照関数 LSON (NODE) によって左部分木に対してこの手続きを再帰的に評価させる．その後に属性を更新する関数 NUMBER (NODE) を用いて中順序番号を代入し，COUNTの更新，右部分木の評価を行ってこの手続きを終わる．

このアルゴリズムの例でも，抽象基本関数を主体として記述され，実現に関する記述が一切含まれていないのが容易に理解されよう．この点が他の構造的プログラム言語やデータ抽象化技法を用いた言語，たとえば CLU[7]，ALPHARD[9]と著しく異なっている．

3. プログラム合成システムの概要

抽象アルゴリズムには，直接実行できるような具体性はなく，個別の問題に適合しているとも限らない．その修正と具体プログラムへの変換が不可欠である．これらは **Fig. 3** にしめすプログラム合成システム(以下 Furnace と略す）によって実施される．

Furnace システムでは，Metal で記述された抽象アルゴリズムがライブラリに蓄積されている．問題に応じてそのうちの一つが選択され，具体プログラムを合成するための素材となる．これに特定の抽象単位データを対応させる．それらの両者あるいはいずれか一方に問題に応じた修正・追加がなされる．これらの選択や修正は現段階では人手にたよるほかはない．我々の方法では，この後の過程が機械化される．

修正された抽象アルゴリズムはトランスレータに入力され Syntax Weak な中間言語*で記述されたアルゴリズムに変換され，プログラム合成器 (A Program Synthesizer：以下 APS と略す)の入力の一つとなる．一方，抽象単位データの仕様と具体データ構造名の指定の2つがデータ操作生成器 (A Data Operation Generator：以下 ADOG と略す）の入力となる．ADOG は前述の4種類の基本関数について，抽象関数と1対1に対応する具体データ構造操作関数を対の形式で出力する．この対集合を抽象・具体変換規則とよぶ．これが APS への残りの入力となる．APS では中間言語によるアルゴリズムを変換規則に従って具体プログラムに変換する．出力されたプログラムが最初の実行可能なものになる．しかしこれは抽象アルゴ

* 現在は Lisp 言語を使用している．

リズムの構造を強く反映しており，具体データ構造等に適合せず時間・空間的動作効率が悪いこともある．そのときには最適化器による処理をほどこす．データ・アクセスにともなって出現する定数計算や共通式の消去，コードの移動等にはコンパイラにおける最適化技法が利用できる．また実際の計算上で定数として扱える変数の処理には部分評価の技法が適用可能である．

現在までにデータ操作生成器 (ADOG) とプログラム合成器 (APS) が試作され，これらが連動して所定の機能をはたし Furnace システムと核となりうることが確認された．さらに Metal 言語トランスレータが設計の段階にある．

4. 具体データ構造の選択

抽象データ構造とそれを計算機上において実現する具体データ構造の間には一般に1対多の関係があり，いずれを選択するかは重要な問題である．

具体データ構造には多様のものが考えられるが，Von Neumann 型計算機を前提にすると配列と結合リストが重要である．両者では単位データの関係の表現能力に大きな差がある．配列では関係をたどるときに添字 (Index) の演算を用いる．対して結合リストではポインタによる．配列では特定の関係に表現が制限されるが，結合リストであれば任意の関係を表現できる．両構造とも多くの変種が存在する．現存の Furnance システムで用意されているのは，次の4種類である．

ⅰ 2進木配列(BTA)：1次元配列により2進木を表現．単位データの属性の個数は任意であるが関係は限定され，添字演算によって実現する．

ⅱ リスト化配列(LAR)：複数個の1次元配列による．関係は配列の添字値（ポインタに相当）によって，属性は値として各配列中に埋め込まれる．各成分へのアクセスは配列名と添字値の指定による．

ⅲ リスト化単配列 (LSA)：上のリスト化配列に似るが，1個の1次元配列によるのが特徴．各成分へのアクセスは，添字値と各成分順序に対応する偏移値の和による．

ⅳ Lisp の2進リスト (BTR)：Lisp の基本構造で1セル：2情報を基本単位とし，すべての成分をポインタで表現する．

名札つき2進木を各データ構造で表現したのが **Fig. 4** である．ⅰの2進木配列の存在は自明であろう．残りの3種類はともに結合リスト形式であるが，

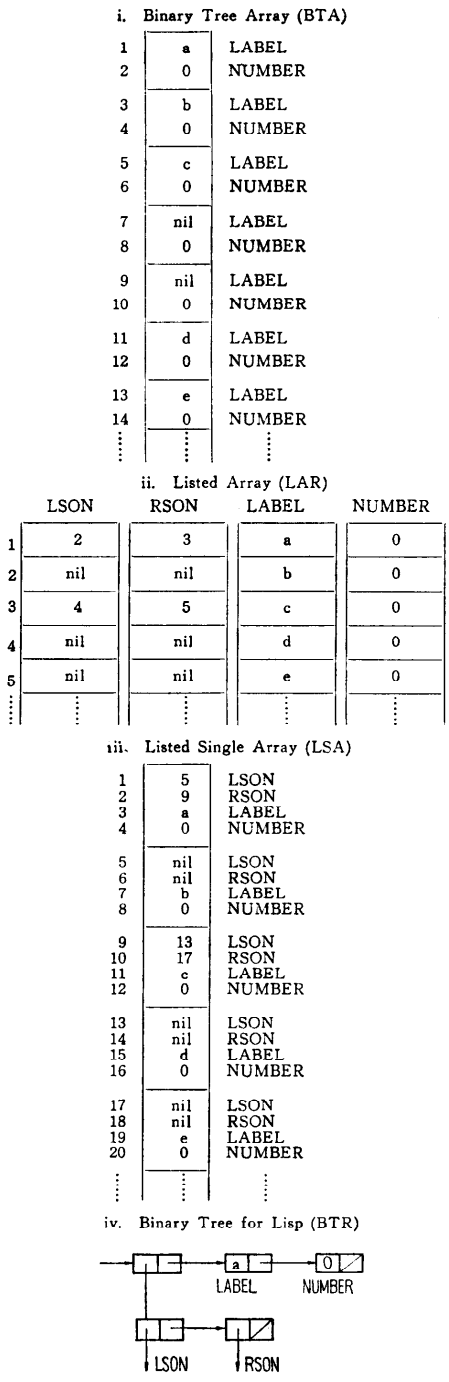

Fig. 4 Some concrete representations of abstract data units.

ⅱとⅲでは計算機方式（たとえば仮想記憶機であるか否か）によって各成分の参照・更新時間に差が出る．

またivの2進リストは Lisp の主たるデータ構造であり，任意の関係を表現する上でのある種の最小構造でもある．

どの具体構造を選択するかを決定するには質的・量的の両面から検討されねばならない．質的な面とは，選択された具体的データ構造によってもとの抽象データ構造を表現できるか否かである．たとえばiの2進木配列では任意の関係が表現不能なのは前述のとおりである．量的な面には，時間的・空間的動作効率がある．2進木配列では関係が実現上に直接表現されることがなく空間的効率は最も良く，かつ関係をたどるのも整数演算により迅速である反面，単位データが動的に生成・消去される状況には時間の点で適応できない．一方結合リストは配列とは逆の性質を持ち，特に Lisp 流2進リストの空間的効率は最も悪い．

いずれのデータ構造を用いるにしても，抽象単位データの関係・属性成分の名称・個数は任意であるから，上記の4種類のデータ構造によって広い応用範囲に適用できる．

5. 抽象・具体変換規則の生成

前にも述べた抽象具体変換規則は，抽象アルゴリズムとともにプログラム合成器 (APS) への入力の核である．この規則は原理的にアルゴリズムからは独立しており，抽象単位データの仕様と具体的データ構造から決定される．よって規則をこれらの仕様から自動的に生成することが重要になる．決して各データ構造に応じて変換規則を前もって用意しておくのではない．もしそうならば，単位データの各成分名等を任意に指定できない．指定法を制限すれば抽象アルゴリズムの書き易さは失われる．これをある程度避けえたとしても規則の数は膨大になり，また実現不能となる．各基本関数を具体データ構造ごとに先験的に準備する方法

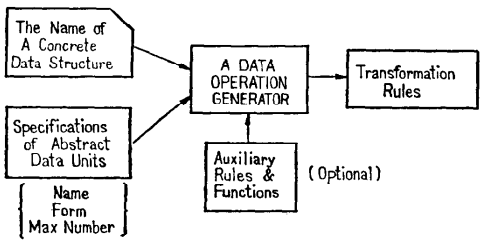

Fig. 5 Inputs and outputs of A DATA OPERATION GENERATOR.

* 最適化の余地が残されている．

```
((LSON N) (PLUS N N 2. -1.))
((RSON N) (PLUS N N 2. 2. -1.))
((LABEL N) (BTANODE N))
((SETLABEL D T) (STORE (BTANODE D) T))
((SWAPLABEL N1 N2) (DE SWAPLABEL
                       (N1 N2)
                       (PROG (T)
                             (SETQ T (BTANODE N1))
                             (STORE (BTANODE N1) (BTANODE N2))
                             (STORE (BTANODE N2) T))))
((NUMBER N) (BTANODE (ADD1 N)))
((SETNUMBER D T) (STORE (BTANODE (ADD1 D)) T))
((DELETENODE D) (FREE D 2.))
((NULLNODE N) (OR (ZEROP N)
                  (#GREAT N 40.))))
```

Fig. 6 Transformation rules for binary tree array (BTA).

が，かつて Balzer, R. M.[10]によって提案されているが，不十分な方法であるのは上の理由から明らかであろう．

このような変換規則を生成するのがデータ操作生成器 (ADOG) の役割りである．**Fig. 5** にしめすように，その入力は抽象単位データの仕様と具体データ構造名の指定が主体である．単位データの仕様にはその形式の指定の他に，その名称や属性各成分のデータ型の指定，実現値の最大個数の指定がある．これらは実現時の配列の名称やそのメモリ割り付けに使用される．ADOG への補助的な入力として，この他にマクロ展開用規則とメモリ管理等のコーティリティ関数があるが，これらは任意仕様である．また出力は当然変換規則である．

変換規則が生成できる根拠は，抽象単位データの関係・属性の各成分の記述順序や成分相互の独立性が直接的に具体データ構造に反映されるからである．このために具体データ構造の形式から，具体基本関数の中味が推論可能になる．

生成例としては，名札つき2進木の場合をしめす．具体データ構造をiの2進木配列としたのが **Fig. 6.** iv の Lisp 流2進リストとしたのが **Fig. 7** (次頁参照) である(いずれも規則の一部である)．単位データを参照する関数の一つは抽象レベルでは (RSON N) であるが，具体レベルが配列であるときは (PLUS N N 2 2 -1)* なる関数すなわち 2*NODE+3 なる添字の演算である．いわゆるポインタの概念が用いられていないのに注意されたい．一方リストのときは (CAR (CDR (CAR N))) のように Lisp のポインタ処理基本関数を使ってそのリスト構造をたどることになる．

6. 具体プログラムへの変換[5]

直接実行できるような具体性を持たない抽象アルゴリズムを具体プログラムに変換するのがプログラム合器成 APS) の役割りである．APS への入力は，前述

```
((LSON T1) (CAR (CAR T1)))
((SETLSON T1 T2) (RPLACA (CAR T1) T2))
((SWAPLSON T1 T2) (DE SWAPLSON
                      (T1 T2)
                      (PROG (T)
                            (SETQ T (CAR (CAR T1)))
                            (RPLACA (CAR T1)
                                    (CAR (CAR T2)))
                            (RPLACA (CAR T2) T))))
((RSON T1) (CAR (CDR (CAR T1))))
((SETRSON T1 T2) (RPLACA (CDR (CAR T1))T2))
((LABEL T1) (CAR (CDR T1)))
((SETLABEL T1 T2) (RPLACA (CDR T1) T2))
((NUMBER T1 ) (CAR (CDR (CDR T1))))
((SETNUMBER T1 T2) (RPLACA (CDR (CDR T1))) T2))
((DELETENODE D) (FREE D 4.))
((NULLNODE T1) (NULL T1)))
```

Fig. 7 Transformation rules for binary tree structure (BTR).

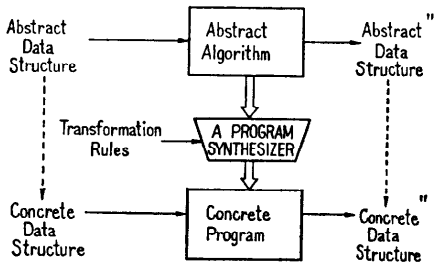

Fig. 8 Relationships between abstract and concrete objects.

のように2種類ある．この入出力関係を **Fig. 8** にしめす．データ構造とアルゴリズムのいずれにしろ抽象・具体間の変換や対応関係が，抽象から具体への一方向に統一されているのが大きな特徴である．Hoare[12]やBurstall[11]の両方向性と対照的である．Burstall の方法では，両方向性のゆえにプログラムを変換するときに発見的な情報を必要とし，機械的な変換は実現されていない．

APS の基本動作は，入力した抽象アルゴリズムを走査して構文中に出現する抽象基本関数を変換規則にしたがって具体基本関数に置き換えることにある．もし具体関数が複雑で置換不能であれば，その具体関数を実行可能な形式でシステムに登録する．

変換例として前述の INORDER アルゴリズムをとりあげよう．これを APS に適合するよう Lisp 言語で書き直したのが **Fig. 9** である．具体データ構造については，前述の4種類全部について変換可能である．対照的な変換例をしめすと，具体データ構造として2進木配列を操作するのが **Fig. 10** の INORDER-BTA である．(STORE IARY……)，(IARY……) はそれぞれ配列 IARY についての更新・参照の関数である．LSON，RSON の関係をたどる関数が，添字演算になっているのは変換規則の例でしめしたとおりである．具体データ構造を Lisp 流2進リストとしたものを **Fig. 11** にしめす．Lisp 基本関数の CAR，CDR の使用に注目されたい．

```
(SETQ COUNT 1)

EDE INORDER (NODE)
  (PROG NIL
        (COND ((NULLNODE NODE) (RETURN)))
        (INORDER (LSON NODE))
        (SETNUMBER NODE COUNT)
        (SETQ COUNT (ADD1 COUNT))
        (INORDER (RSON NODE)))]
```

Fig. 9 An INORDER algorithm written by Lisp(1.6).

```
EDE INORDER-BTA (NODE)
  (PROG NIL
        (COND ((OR (ZEROP NODE)
                   (*GREAT NODE 40.))(RETURN)))
        (INORDER-BTA (PLUS NODE NODE 2. -1.))
        (STORE (BTANODE (ADD1 NODE)) COUNT)
        (SETQ COUNT (ADD1 COUNT))
        (INORDER-BTA (PLUS NODE NODE 2. 2. -1.)))]
```

Fig. 10 An INORDER program for binary tree array (BTA).

```
(SETQ COUNT 1)

EDE INORDER-BTR (NODE)
  (PROG NIL
        (COND ((NULL NODE) (RETURN)))
        (INORDER-BTR (CAR (CAR NODE)))
        (RPLACA (CDR (CDR NODE)) COUNT)
        (SETQ COUNT (ADD1 COUNT))
        (INORDER-BTR (CAR (CDR (CAR NODE)))))]
```

Fig. 11 An INORDER program for binary tree structure (BTR).

7. おわりに

本論文においてデータ型やデータ構造に関する具体性を除去して抽象アルゴリズムを記述し，これに具体性を付加して具体プログラムに変換することにより，さまざまな状況に適合させうる方法をしめした．またこの方法の核ともなるデータ操作生成器 (ADOG) とプログラム合成器 (APS) を試作し，具体プログラムの変換の機械化が可能であり，かつ抽象から具体へと一方向に変換できることを実証した．

今後の課題としては，Furnace システムの残った部分の完成のほかに抽象アルゴリズムにふさわしく，かつ能率的な正当性の検証法の確立がある．

末筆ながら，この研究の機会をあたえて下さった石井治ソフトウェア部長，棟上昭男情報システム研究室長，また討論していただいた同研究室の皆様およびソフトウェアを作っていただいた皆様に深く感謝します．

参考文献

1) Biermann, A. W.: Approaches to Automatic Programming, Advance in Computers, Vol. 15, Academic Press (1977).
2) Winograd, T.: Understanding Natural Language, Edinburgh Univ. Press (1972).
3) 大石東作: 汎用問題解決システム (META-88) の基本構成, 291, 情報処理学会第 16 回大会 (1975).
4) 大石東作: 抽象アルゴリズムの記述と具体プログラムへの変換, 電子通信学会技術報告, AL 76-36 (1976).
65) 大石東作: 汎用問題解決システム (META-88) におけるプログラムの合成, 357, 情報処理学会第 17 回大会 (1976).
6) Aho, V. A., et al.: The Design & Analysis of Computer Algorithms, pp. 55-56, Addison-Wesley (1974).
7) Liskov, B. H., et al.: Programming with Abstract Data Types, SIGPLAN Notices, 9, 4, 50-59 (1974).
8) Liskov, B. H., et al.: Specification Techniques for Data Abstractions, IEEE Transactions on Software Engineering, Vol. SE-1, No. 1 (Mar., 1975).
9) Wulf, A. W., et al.: An Introduction to the Construction and Verification of Alphard Programs, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4 (Dec., 1976).
10) Balzer, R. M.: Dataless Programming, Proceedings of FJCC (1967).
11) Burstall, R. M., et al.: Some Transformations for Developing Recursive Programs, Proceedings of International Conference on Reliable Software (1975).
12) Hoare, C. A.R.: Proof of Correctness of Data Representations, Acta Informatica, 1, 271-281 (1972).
13) Suzuki, N.: Automatic Program Verification II, MEMO AIM-255, Stanford Artificial Intelligence Laboratory (1974).
14) 大石東作: 抽象アルゴリズム記述言語 Metal 仕様書, 第1版, 電総研内部資料 (1977).

(昭和52年6月3日受付)
(昭和53年3月20日再受付)