人工知能と対話技法 15－3
（1980. 5. 19）

データ構造処理プログラムの合成について

間野 暢興（電子技術総合研究所）

1. はじめに

従来提案されているプログラムの合成方式を検討して感じるのは、機能的な物の見方およびデータの構造把握の概念、の両者が全く欠けていることである。人間のプログラム作成・新しいアルゴリズム発見などは基本的にはこれらによって導かれて行われていると思われるだけに、これは奇妙なことである。機能的な見方を導入すれば、必要なデータ構造操作のあつまりに適合したデータ構造の選択、合成の手係りを与える基本操作の導入、合成における意味論的なガイドが可能となり[1),8)]「何のためにこれが用いられているのか」という疑問に答えるかたちでの存在・役割に対する理由づけができる。また、データの構造を把握すれば、入力や出力のデータの構造とプログラム・プランの論理的構造との対応をつけることができ[1)]、複合データ構造のプログラム・プランの合成も容易となる。またプランを修正して他の目的のためのプランに変化させるときも、データの部分構造ごとに対応するプランの部分が容易に同定できるので都合が好い。ここで述べるプログラムの合成方式はこれら二つの点を強調した合成・証明方式である。関数型および副作用を含むデータ構造処理プログラムで、シーケンス文・条件文・再帰文からなるものの合成を目的とする。

2. 機能にもとづくデータ構造の選択

2.1 関数型プログラムと副作用のあるプログラム

関数型プログラムとはpure-LISPのプログラムのような副作用のない関数の値の受け渡しにより演算を行うプログラムで、最近新しいコンピュータ・アーキテクチャとして注目されているデータフロー計算機は、そのような計算方式でできる限りやってみようという試みである。これに対して副作用のあるプログラムは我々が日常よく書き用いているプログラムで、変数の値の書き変え、配列の要素の割り付け、データ構造内のポインタの変更、による記憶領域の状態の変化を利用している。関数型プログラムは並列（半順序）に実行できる可能性があり、単純で数学的に扱いやすい。副作用のあるプログラムは、現在のフォン・ノイマン型計算機では、記憶領域・計算時間ともに能率的であるが、複雑で"虫"の発生をひき起しやすい。探索プログラムは、どちらの計算方式にしても、関数型プログラムとなる。関数型プログラムか副作用のあるプログラムか、どちらを合成したいのかでとられる手段は非常に異るが、挿入・除去などの副作用のあるプログラムも探索プログラムを基本としていることから分るように、一般に関数型プログラムは副作用を利用したプログラムよりより基本的と考えて良いであろう。

2.2 プログラムの機能別分類

データ構造を扱うプログラムを眺めてみると、特定のデータ構造によらず意味的・機能別に分類できる[1),3)]ことに気がつく。それは入力のデータと出力のデータの集合的関係にもとづく分類を核としている（もちろん、他の要素も類別の項目と

して現われるが)。関数型プログラムの場合の、この観点からの分類およびそのプログラム・プラン合成におけるガイド的働きについては文献1)に述べたのでここでは省略する。副作用のあるプログラムの場合の分類を表1表に示す。この機能の観点は、次節で述べる抽象表現から具体表現への精製における適合した実現データ構造の選択や、プランの合成のガイドとして、重要な役割を果す。後者の例として、集合保存の演算では一定の記憶領域で副作用を活用すべく、配列の要素の交換(bubble sort, exchange sort, heap sort)やポインタの回転[2](リストのreverse, Deutsch-Schorr-Waiteのグラフ・マーキング・アルゴリズムなど)の手段をとることが挙げられる。

	スタック	配列	ハッシュ表	2進探索木	木構造	2-3木
集合保存演算						
登録	push	arraystore	insert	insert	insert	insert
除去	pop		delete	delete	delete	delete
探索	top	arrayfetch	member	member	member	member
最小値見出し				min		min
2つの集合の合併					union	union
要素の属する集合の検出					find	find

表1表　副作用のある場合のデータ構造別操作演算分類表

2.3　データ構造の選択

各データ構造にはその構造に適合した操作が定まっている。ゆえにあるデータ構造に対しどのような操作のあつまりが施されるかによって、それの具体データ構造として何を選んだら良いかを決められる。例えば簡単には表1表に記されている操作演算のあつまりでcoverできるデータ構造を用いれば良い[3]。データ構造に関するコンサルタント・システムあるいは処理プログラムの自動合成システムは当然表1表のような知識を内蔵し、ユーザの能率的なシステム実現に協力できねばならない。

ひとつの具体的なデータ構造だけではcoverしきれないときは、複数個のデータ構造を直積結合として用いてcoverできるようにする。例えばスタックを配列で実現する場合、スタックの入口を指すのに配列のインデックスを表わす変数と配列の直積結合を用いる。

副作用を利用したデータ構造操作においては、データ構造の構成要素間の関係(および構成要素の属性[4])が求められる機能を実現するのに重要な役割を果す。この構成要素間の関係は次の3つの方法のいずれかで実現される。

① データ構造内のポインタ
② プログラムの変数の値の利用
③ プログラムで構造をたどる途中で取り出しあらかじめ用意したデータ構造に記入・修正して計算に利用するもの。

双方向リストでは、ある構成要素の次と直前の要素がそれぞれのポインタで指示されているため、挿入・除去が簡単に行える。これに対し一方向リストでは直前の要素は、②のように構造のtraverseのための変数の前の値を受け継ぐ補助変数

により指示される。Deutsch-Schorr-Waiteのグラフ・マーキング・アルゴリズムでは、ある構成要素の左息子・右息子・親を示す3つの関係のうち、マーキングの進行に従って構成要素内の2つのポインタの表わすものの割り付けを変化させ、残りのものを補助変数を使って指示している。③は例えば、無向グラフの双連結性成分検出のための縦型探索アルゴリズム[3]における親の関係、さらにもっと複雑な目的のための関係などを格納する。

以上から分るように、データ構造の選択導入は必要な機能の実現のためのデータの構造の各部分(あるいは全体)に対する役割の割り付け(あるいはその利用)の決定と解釈できる。複雑なデータ構造が複雑な目的に用いられる場合、これは単純なことでは無くなるであろうが、それはシステムに内蔵された(やや高級な)知識によって導入されることとなろう。

3. データの構造の記述およびデータとプログラム・プランの構造の対応

3.1 データの構造の記述

データの構造の記述を、構造の構成要素の記述のレベルからデータ構造の共有結合のレベルまで、部分から全体へと記述を与えるとすると次のようになろう。

① 構造の構成要素あるいはレコードの記述

関係と属性で構造の構成要素の内容を表現するものが提案されている[4]。レコードの記述は一般にその成分の役割名とタイプ名のペアを並べたもので表わされる。

② データ構造の内部構成の記述

再帰データ構造の構成を disjoint union (du) と Cartesian product (cp と略す)[5]を用いて再帰的に定義したもの、あるいはJackson法におけるJackson木[6]など。Natural Programming Calculus[7]におけるように述語論理表現によるものもある。

③ データ構造毎の記述

これはひとまとまりのデータ構造のその構成要素を述べる記述で、代数的仕様記法のタイプ宣言がこれにあたる。

④ 複合データ構造の記述

③のデータ構造を複数個結合して用いるもので、構造を共有するものと、構造は共有せず値のみ共有するものがある。前者では通常階層結合がとられることが多い。後者は直積結合という。階層結合は②あるいは③を用いてnestして表現できる。直積結合は各データ構造における①の記述の役割名を用いて宣言することが必要である。

以上は具体・抽象のレベルの区別を特にしていないが、さらにデータベースの関係モデルにおける関係の記述のように論理レベルにおける記述がある。

3.2 プログラム・プランにおける入出力データ構造の部分との対応づけ

前節に示したようなデータの構造の記述をシステムは内部に持つと、プログラム・プランの合成や修正を行うのに大変助けになる。入出力のデータの構造全体の構成を把握すれば、与えられた問題においてその各構成部分要素におけるなすべきこととそれらの間の関係を定めることができる。たとえば、上の②のduとcpを用いて入出力のデータの構造を記述し、それにさらに入出力の要素間の対応やそれらの内部条件などを問題の意味の指定と共に与えれば、関数型のプログラム・プランがこれらをもとに合成できる(文献1および4.参照)。この場

合、構造記述に基くプラン内の条件判定の場合分けと順序づけ、およびデータフローによるデータ構造の各部分要素とその処理との対応づけができることを活用している。副作用を利用したプログラムの場合は、記憶領域の状態変化が起るのでプランの中でコントロール・フローが重要となって来るが、それはデータの構造のtraverseを骨組としており上に記した関数型のものの土台の上に各部分要素処理において副作用のある場合の演算を追加修正したものと考えられる。

なおduやcpによるデータの構造の記述は一般にひとつのデータに対し幾通りか考えられ、その分割表現の仕方はデータをどう扱うかあるいはどんな目的の問題かにより選ばれるものであり、計算の戦略指定によって定められるべきものでもある。例えば、2進木のtraverseにおけるpre-, in-, post-orderはそれぞれ次の構造記述が対応する。duとcpの引数の順序は意味がある。cp()は空を表す。

pre-order　btree A $\stackrel{def}{=}$ du(cp(), cp(A, btree A, btree A))
in-order　btree A $\stackrel{def}{=}$ du(cp(), cp(tree A, A, btree A))
post-order　btree A $\stackrel{def}{=}$ du(cp(), cp(tree A, tree A, A))

グラフはduとcpでその構造は記述できないが、その処理アルゴリズムの多くは木の構造によるものを基本としそれに修正を加えたものである。縦型探索はpre-orderであり、横型探索は木の構造を根を出発点に各レベルのnodeの順に全順序の見方で把えている。

4. プログラム・プランの合成

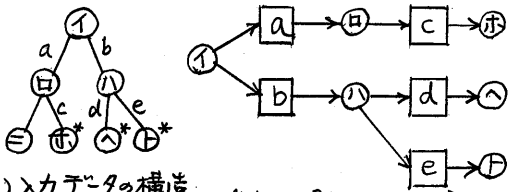
4.1 データの構造記述に基く関数型プログラムの合成

関数型の再帰データ構造処理プログラムのinformalな合成法については文献1でも述べた。ここではファイル処理のようなより巨視的なレベルにも適用する合成法を一般的・抽象的に述べる。入出力の構造はdu, cpあるいはジャクソン木により与えられているとし、ジャクソン法でいうstructure clashは無く試行による問題解決による合成は不要（あるいは済んだ後）と仮定する（入出力の要素間の対応はついている）。

合成のための手続きの要点を次に記す。

1）出力の構造の要素のうち入力の要素に演算を施して得られるものは、その演算を行うセグメントをプラン・エリアに導入し、そのセグメントの入力と出力のデータを入出力のデータ構造要素と対応づける。

2）出力に必要な入力の要素を入力からそのselectorセグメント(群)を用いて取り出す。第1図にその例を示す。入力の構造記述において必要な要素から根へとたどり途中でてくるデータの部分要素名とそれをとり出すためのselectorを(b)のようにプラン・エリアに展開する。途中既に展開された要素にあえば、プランの対応するデータのところから取り出すようにする。



(a) 入力データの構造　(b) プランの一部

第1図 Selectorによる必要なデータの取り出し
（○はデータ、□はselectorを表す）

3）出力の組立は出力の構造の記述のレベル毎のconstructorを用意して、出

力データの構造記述における末端の要素から根に至るまでのデータフローをプランの中に作る。

4) 入力データの構造記述の中にduによる選択表現が存在するか、演算において合成判断が含まれるとき、条件文とそれが働くscope表現がプラン中に現われる。条件文とそのscopeの間にはコントロール・フローがつけられる。

5) 最適化のためや、物理的拘束条件からセグメントあるいはプラン内のあるまとまった範囲をなすscope間にコントロールの優先順位がつけられることもある。

6) プラン内のデータ・フロー(およびコントロール・フロー)をたどってシーケンス文に直せるところはコントロール・フローをつける。条件文ごとに再帰的に働くtopological sortのプログラムでこれはやれる。

4.2 定理証明の方法に基く関数型プログラムの合成

データの構造記述にプランのデータフローの対応をつけ、問題の意味の解釈をシステムに貯えられた知識によって導かれながら、定理証明の方法によって関数型プログラムのプランの合成を行う方法について述べる[8]。これは文献1のinformalな方法を定理証明の形式に一般化したものである。データの構造の記述はやはりduとcpを用いて与えられるものとする。問題ひとつにプランひとつが対応しそれは上位セグメントとよばれ、知識ベースに貯えられたセグメントと公理の写しをプランエリアに取りだし、問題の入出力仕様を充すようにデータ・フロー、ロジカル・リンク、コントロール・フローからなる論理的なプログラム・プランの構造を合成する。セグメントの入出力仕様はpre-conditionとpost-conditionを含み、公理はantecedentとconsequentよりなる。それらの間の論理的つながりをロジカル・リンクとよんでいる。上位セグメントのpost-condition, 下位セグメントのpre-condition, 公理のantecedentがgoalあるいはsubgoalに相当し、それらを充すのは下位セグメントのpost-condition, 上位セグメントのpre-condition, 公理のconsequentである。ひとつの問題において関連するデータの構造(定義)が複数個あるときは、それぞれに対応するセグメント名をつけて知識ベースの中で管理しておくと、あるプランの中のデータフローの対応するデータ構造名からそのセグメントの写しを下位セグメントとして導入できる。

合成のための問題解決法を次に述べる。入力と出力両方向からセグメントと公理の論理的対応をとることで問題解決は進行する。入出力の構造記述の各部分に問題ごとに固有の名前をつけ、プランの上下位セグメントや公理に現われる変数がとりうる値とする。これがプラン中のデータ・フローと入出力の構造の部分との対応に相当している。この変数と値の束縛を通して入出力のデータの部分どうしの対応がつく場合がある。

1) 条件判定の場合分けがある場合には、次の3つが源となっている。

① 入力のデータ構造記述にdisjoint unionの表現があるとき。

② 2つの入力相互の関係が不定なとき考え得る全ての関係を取り出し、各場合分けにおいてその関係を仮定として持たせる。

③ 問題の意味によりデータ内部の要素の相互関係の場合分けが必要となるとき。

これらを入力の構造記述の順に従ってひとつずつ順番に2)以下に述べるようにサブプランを合成し全部の場合分けが終了すれば成功となる。なお問題の前提条件から起り得ない場合分けはあらかじめ取り除いておく。

2) 上位セグメントのpost-conditionに複数のsimultaneous goalの条件があるときはデータの集合関係を表わしているものを先に問題解決を行う。1)に記した場合分けごとに両方のgoalが充足されなければならない。

3) 上位セグメントのpost-conditionをgoalに、公理(あるいはレンマ)を用いてsubgoalを展開していく。公理はMannaとWaldingerの論文の変換規則(文献9)になぞらえると次の形をしている。

<consequent> ⇒ <subgoals> if <condition>

<consequent>はgoal-directedに整合されるのに対し、<condition>は判断のセグメントのpostconditionと整合がとられ、それらをpairとして問題解決が試みられる。複数個の公理が候補として存在するときは、原則として<subgoals>の簡単なものが優先し、不成功に終ったとき複雑なものが試みられる。<subgoals>の部分に出て来る述語は、部分関係を示すもの、組立関係を示すもの、などがあり、これらはそれぞれ入力や出力のデータの構造分割を表現している。

4) セグメントには次に記すようなものがある。

① 入力の構造記述をたどるときのそのセレクタ

② 1)に述べたような条件判定セグメント

③ データフローに基く再帰あるいは他のデータ構造処理のセグメント

④ 問題の意味、すなわち上位セグメントのpostconditionにより導入されるもの。これには具体的に決ったセグメントと、副問題セグメントがある。後者は該当するセグメントが知識ベース中に見つからない場合とか、複数の候補がある場合である。入力が集合の場合などが多い。

⑤ データ・フローにおけるタイプを調節するためのセグメント

5) セグメント間のデータフローは変数の値が定まったとき、あるいは問題の意味に基いて導入される(組立など)。データフローをたどってその上に出現するセグメントのpost-conditionを集積し、subgoalの公理のantecedentと比較すると、副問題の仕様決定やさらに導入すべき公理を選択するのに役立つ。

6) 問題解決の途中の公理のantecedentに、条件判定における仮定に矛盾する述語が現われたときは、その試行は失敗とし後戻り(backtrack)して他の組合せを試みる。

7) 成功した試行には、公理を逆に上位セグメントまでたどってその出口へとコントロール・フローをつける。

4.3 関数型プログラム・プランの合成例

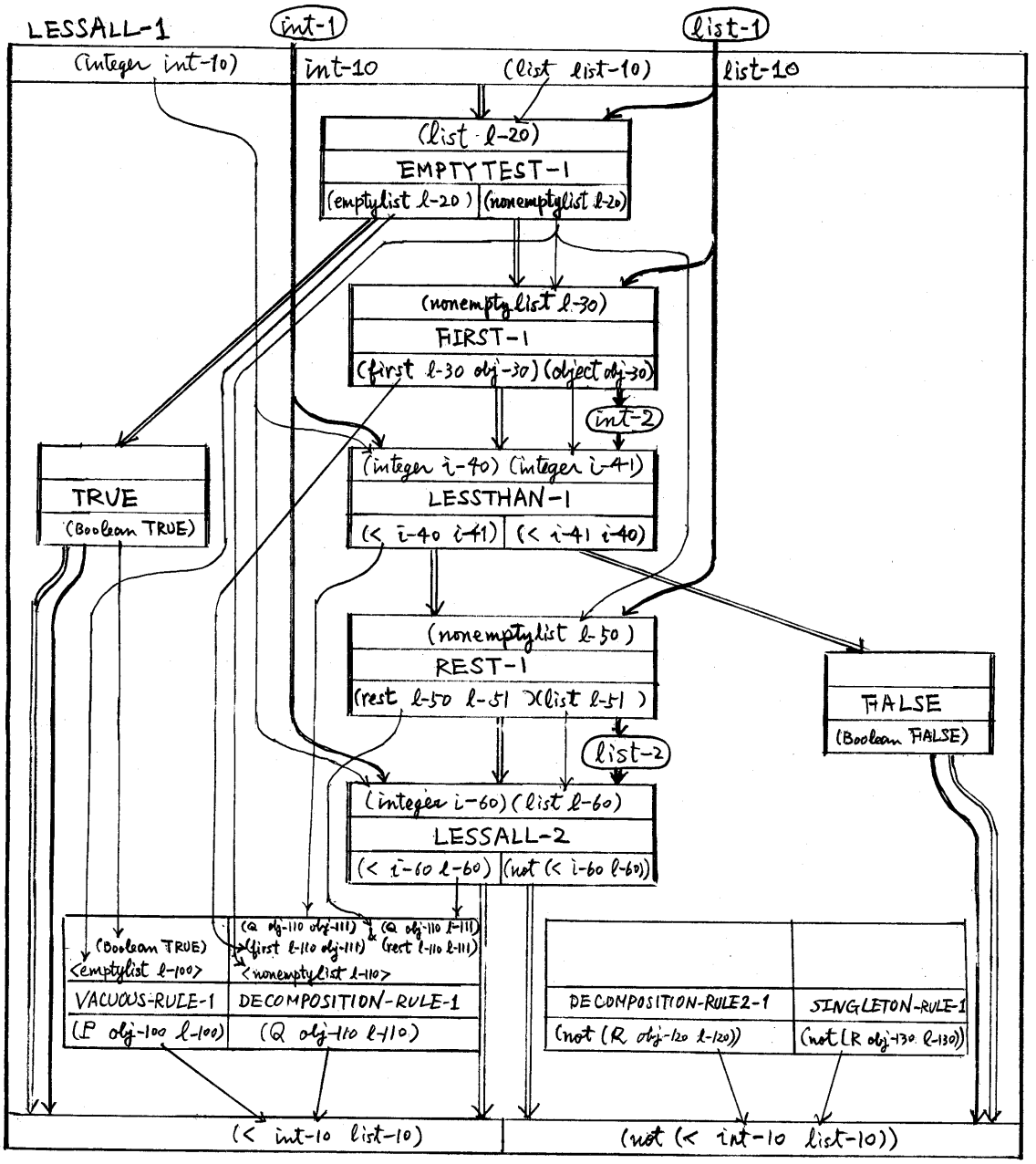
第2図に合成されたプランの例を示す。これはManna & Waldingerの文献9にのっている例題である。

lessall (x, l) ⇐ if empty (l) then true
else if x > first(l) then false
else lessall (x, rest(l))

lの構造の記述: list int def du(cp(), cp(int, list int))

名前: list-1 int-2 list-2

取扱いを簡単にするために、
(forall (integer-10)(member list-10 integer-10) ⇒ (< integer-11 integer-10)
を (< integer-11 list-10) と表現している。同じ文献のallall問題(xがリ



precondition
セグメント名
postcondition

antecedent
<condition> (subgoals)
公理名
consequent

data-flow ↓
control-flow ⇓
logical link ↓

第2図 LESSALLプログラムのプラン (一部略)

ストである）とその副問題 lessall は本方法では全く一様な扱い方で解決され，同文献の不自然な一般化（generalization）など不要である。

5. 副作用のあるプログラムの合成と修正

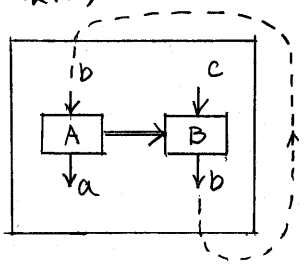
5.1 副作用の表現

副作用を起すoperationは，変数・配列への値の割り付け，データ構造内のポインタの変更などがある。変数への値の割り付けは，関数型のプログラム・プランではセグメントの出力から複数本データフローをつけることで足りたが，副作用のある場合は割りつけのactionをセグメントとして表現する必要がある。また記憶の状態を破壊しないで次の状態に滑らかに計算を続けるには，値の"slide"[2]が必要となる。これは第3図のようにbを他の目的に使う場合に，

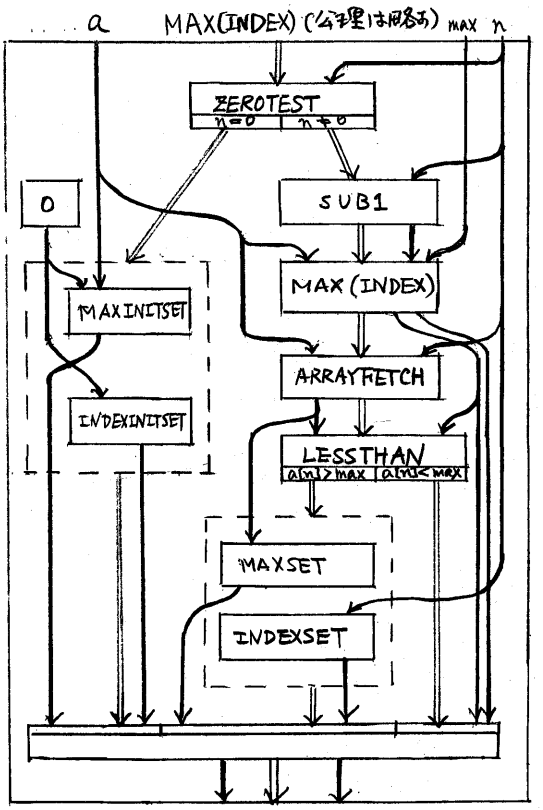
A: a←b　　B: b←c

の順で割り付けのoperationを行い，bの値をaに退避させてから使用するものである。配列における二つの要素の値の交換やポインタの回転はこの特殊な例で，補助の変数をひとつ利用している。この"slide"はプログラムに修正を施すときにも気をつけねばならぬことで，逆にどこに新たなセグメントを挿入したら良いかを容易に決める手掛りとなる。副作用により入力と出力では状態の変化したデータ・オブジェクトはその現われるセグメントの入出力仕様で宣言し，かつ入力と出力で実際は同一のデータ・オブジェクトはおのおのの名前が同一のものを表わしていることを宣言する[10]。副作用を利用して繰返し計算に用いられるもの（例えば第3図のb）は，そのプランを表わす下位セグメントを含む上のプランにおいて，そのデータフローがループとなり初期設定が必要になることがある（tail recursionの場合）。

第3図　値のslide
（A，Bはassignmentを表わす）



第4図　配列の最大値を求めるプラン



5.2 合成方法

副作用のある場合の合成もgoal-directedであることには変りがないが，2.1にも述べたように関数型のプログラムに比べず，と機能設計や状態変化の中間stepを与える必要があり難しい面が多い。また最初から副作用のプログラムを考える問題（第4図の最大値を求める問題など）と

関数型プログラムを基本としこれに副作用を考慮した修正を加えて目的とするプログラムを得る問題（探索プログラムを修正して得られるリストの挿入や除去のプログラム、あるいは木をpreorderにたどる関数を基本としたグラフの縦型探索マーキング・プログラムさらにはそれを土台とした無向グラフの双連結部分・有向グラフの強連結部分を取り出すプログラム[3]、など）がある。いずれの場合も気がつくのは、関数型のプログラムの場合と同じくデータの構造記述に対応したプログラム・プランの考え方が非常に有効なことである。第4図に示した配列a[0:n]の最大値を求めるプログラム・プランでは、抽象シンタックスの考え方に基いて、 list A def du(cp(), cp(list A, A)) とまず考えてリストの最大値を求めるプランを作り、次にリストと配列の構造の対応をつけて、配列向きのプランに書き直す[8]。このプランにせよ、上に述べたグラフ処理のプログラムにせよ、データの構造の各部分を処理しているプラン上の部分が明白なのでプランの修正は非常に能率的に行える。

5.3 修正の箇所と修正法

修正を加える箇所としては、初期設定の部分、条件文の部分、シーケンス文の部分、再帰セグメントの前あるいは後などがある。新たな副作用のあるセグメントを加えたり、条件判定の場合分けを追加したり、場合分けにおける実行内容をすっかり新しい目的のために書き換えたり（探索プログラムを挿入プログラムに変えるときなど）、上位セグメントに新たなgoalを追加するに合せて再帰セグメントにも追加したりする。新たな変数やデータ構造の追加もある。

修正を加えるとき注意すべき事柄としては次のことが挙げられる。

① 副作用のあるプログラムの場合はプログラム・プラン中のコントロール・フローが重要な役を果すので、挿入の箇所に注意する。

② 土台となるプランの機能を破壊しないよう値のslideに気をつける。

③ データの構造が挿入・除去のあとも全体として前とかわらないように（要素どうしの関係やreachabilityがinvariantに保たれるように）修正する。このときデータの構造を考慮して端や中間を意識して行う。

④ 構造を共有しているとき、複合構造などではとくに副作用のoperationに注意。

第4図の最大値を求めるプランに新たに最大値の存在するインデックスを求めるgoalを追加（同図）したときは、このプラン表現では新しいデータ・オブジェクトが既存のものに全く干渉を及ぼすことなくその関連セグメントを追加できるのは容易に分る。このプランの合成をも含めて、Manna & Waldingerの論文[9]にあるようなweakest-precondition operatorを用いた複雑なsimultaneous-goal principleに基いて合成や修正を行う必要は実際のところほとんど無いといって良いであろう。これはプラン[10]が次の性質を持つことによる。

① プランは2次元的広がりを有する。

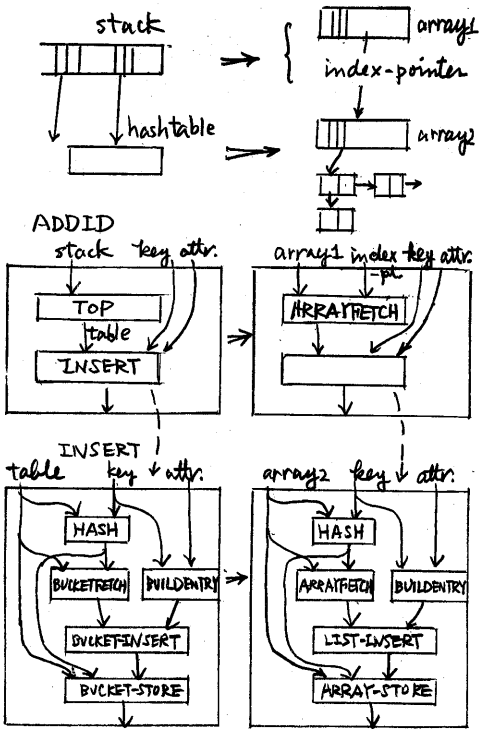
② データフローが明確な独立した存在として表現されている。

③ データの構造の部分とそれを処理する部分との対応がつくこと。

④ （図には略されているが）goalを達成する公理とセグメントのpostconditionへのロジカル・リンクがついており、コントロール・フローを出口の方から逆にたどることにより、容易に直接影響するものを見出せること。

6. 複合データ構造とデータの抽象表現から具体表現への変換

複合データ構造では、それを構成するデータ構造のおのおのについて用意された、一般性のあるプランを持つセグメントを組合せて用いて、かなり容易に全体の構造に対するoperationを組み立てることができる。階層結合の場合、末端のデータ構造に構造変化を施すために途中のデータ構造は探索のoperationで通過されるが、特に末端とそのひとつ前のデータ構造のつなぎ目は構造変化の影響を受けるので注意が必要となる。構造の共有にしろ値の共有にしろ、構造の記述から全体像を把握して副作用に対処することが大切である。なお、全体の構造に対するプラン合成では、あいまいさが残りユーザの指示が必要となることもある。



第5図 SymboltableとADDIDの抽象ー具体表現の対応

複合構造では抽象データ構造による表現がはじめにとられ、具体データ構造へと段階的に表現が変換されることが多い。第5図にstackとhashtableで表現されるsymboltableにkeyとそのattributeを登録する場合のプラン、およびその具体データ構造表現におけるそのを示す。抽象セグメントの具体プランを用意しておくと役立つ。プランに現われるデータ・オブジェクトが変っても使われるセグメントの意味は共通性がある。入出力仕様の実現変換については述べなかったが、これがどの程度自動的に行えるかは今のところ不明である。なお第5図のようにデータ構造が抽象レベルと具体レベルで類似しているときは、文献8の方法が応用できる。

7. おわりに

機能設計とそれをプランにもちこんで実現するための"extrinsic"ー"intrinsic"[10][11]な情報の対応づけが、今後のプログラムの合成の研究課題で、これができて初めてプログラムの自動合成は実用となりうるものと思われる。最後に日頃御世話頂く石井ソフトウェア部長、榛上情報システム研究室長と室員の方々に感謝します。

参考文献

1) 間野："再帰データ構造処理プログラムの知識にもとづく合成について"電子通信学会オートマトンと言語研究会資料AL79-120 (1980). 2) N. Suzuki："Analysis of Pointer Rotation", submitted to 1980 ACM POPL Conf.
3) A.V. Aho et.al："The Design and Analysis of Computer Algorithms", Addison-Wesley (1975)
4) 大石："抽象アルゴリズムの記述と具体プログラムへの変換"情報処理, vol. 19, no. 11 (1978)
5) W.H. Burge："Recursive Programming Techniques", Addison-Wesley (1975)
6) J.H. Hughes："A Formalization and Explication of the Michael Jackson Method of Program Design", Software-Practice and Experience (1979). 7) A. Hansson and S. Tarnlund："A Natural Programming Calculus", Proc. of the 6th IJCAI (1979). 8) 間野："データフローの概念に基づくプログラム合成方式"電子通信学会オートマトンと言語研究会資料AL79-70 (1979). 9) Z. Manna et al："Synthesis: Dreams ⇒ Programs", STAN-AIM-302 (1977). 10) C. Rich et.al："Initial Report on a LISP Programmer's Apprentice", IEEE Trans. SE-4 (1978)