

## ボトムアップ型のプログラム抽象化の一方法 A Method for Bottom-up Program Abstraction

古 東 馨

Kaoru Kotoh

東京電機大学理工学部数理学科

Department of Mathematical Sciences, Tokyo Denki University

あらまし プログラムを抽象化することは、単にプログラムの技法としてだけではなく、プログラムの自動検証、自動作成への貢献も大きいものと思われる。しかし、多くの議論ではプログラムの機能面での抽象化に向けられ、実装に関する面はプログラムの抽象化の過程で排除すべきものとして扱われている。ここでは、抽象化の過程で避けて通れない階層間の関係づけは、機能の面からだけではなく、実装と機能の相関した問題であると考ええる。そこで、プログラムの実装を抽象化したものとしてスキーマを導入し、スキーマと計算機能との係わりを定式化した。また、この考え方に立脚したボトムアップ型のプログラム検証システムの構築を試みる。

**Abstract** The automatic program abstraction is not a method only for computer programming but also a powerful tool for realizing the automatic program verifier. The program abstraction is said to extracting a functional property from a program. However, it must be important to recognize the role of program implementation when we argue the relation between abstract programs in different hierarchical levels.

In this report, a schema is defined as an abstracted implementation and the relation between schemata and functional properties are formalized.

Finally, it is mentioned about structure of a planning bottom-up program abstracting system.

### 1. はじめに

プログラムの抽象化は、複雑巨大化するプログラムに対する有効なプログラミングの一技法として注目され、近年では種々のプログラミング言語の中に、抽象化の機能が導入されて来つつある。

プログラムの抽象化の技法は、単にプログラムの考え方の方法としてばかりではなく、機械によるプログラムの検証の可能性を広げ、機械の援助の下でのプログラミングすなわちComputer Aided Programmingの可能性をも広げた。この分野の進展はプログラミング作業を、人はプログラムの仕様を

作成し、後は機械による自動プログラミングにまかせるという自動化への道をも示唆している。

筆者等は、いろいろな離散構造の計算用のプログラム・システムを開発して来たが、抽象化を進める過程で以下の2点が注目された。

- (1) 我々の実装を意識しない抽象構造をプログラムとして実装する場合には、それよりも下位の抽象構造を用いるのであるが、それらの下位抽象構造を操作する実装に用いられるのは、常套手段とでもいうべき簡単な制御方法(スキーマ)を用いる。

- (2) 対象となる計算システムに特有のスキーマがある場合もあるが、全体として多種類ではない。

この結果を踏まえて、我々はスキーマを抽象化されたプログラムの階層の間を関係付ける実装の方法に深く関連したものとして位置付けをした。即ち、トップダウン的手法では、上位の抽象構造を下位の構造で記述するというプログラムの実装を、下位の構造に適当なスキーマを適用することと解釈し、逆にボトムアップ的プログラミング又はプログラムの解析では、下位構造にスキーマを適用することにより上位の構造の演算の定義を付けをしている。

前に述べたとうり、問題の対象範囲をある程度制限した場合に、スキーマの数が少ないならば、heuristicな方法を用いても適用されたスキーマを検出することが出来る。各スキーマには、それに特有な検証事項を付随的に記憶しておけば、プログラムの自動検証での“何を検証すべきか?”という大きな問題[1]の部分的解決にも役立つものと考えられる。

本報告では、上に述べた仮定の下でのスキーマの関数的定義と、ボトムアップ的プログラムの抽象化システムの構築を試みる。

## 2. 抽象的プログラムとスキーマ

プログラムの抽象化の過程には、2種類の方法があるものと思われる。その1つは、プログラムのfunctionalな部分に注目したものであり、これは、抽象型データとその上の演算とによる代数的構造として捕らえることが出来る。他方は、functionalな機能とはかかわりなく、プログラムの実行順序にのみ注目したschematicalな部分の抽象化をしたモデルである。

プログラムを抽象化するには抽象化の程度によりfunctionalな代数構造に階層構造を生ずることは良く知られている。これらの階層間を結び付けているのは、実装としてのスキーマである。

1例として、筆者等がオートマトンやグラフなどの離散構造計算のためのプログラム・システム[2]の開発時に注目されたスキーマCONVLISTを紹介しよう。

このスキーマは、一般にsubset constructionとして知られている計算方法をより抽象化したものであ

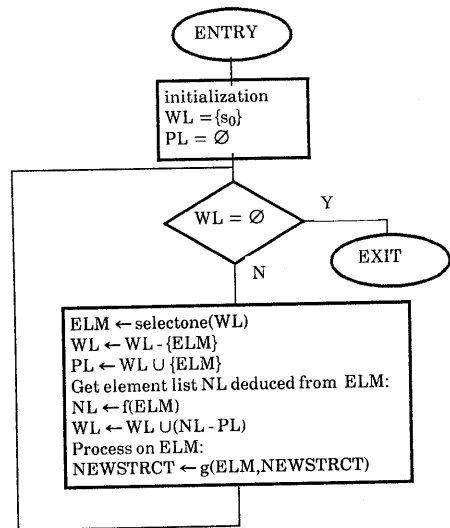


図1. CONVLISTの制御流れ図

る。実際のシステムに実装されたものは少々複雑であるが、簡単に言えば、システムのある初期状態から派生的に生ずる状態(図1の関数  $f$  によって得られる)をその派生の及ぶ限り順次処理(図1の関数  $g$  によって示される)を加えるというもので、処理で引用されている操作  $f, g$  は下位の構造(例えば、状態遷移表)上で定義されている演算であり、このスキーマによって表現されているものは、上位構造(例えば、オートマタ空間)上の1つの演算となっている。このスキーマは、前述のプログラム・システムでは、7種類のオートマタまたはグラフの間の演算を定義するのに利用されている。また、このスキーマは単にオートマタまたはグラフに特有のものではなく、ある種の並べ換え、探索のためのスキーマとしても有用であることが判明している。

上の例のように、抽象化されたスキーマはプログラムの実行順序に関する抽象モデルで、データの特性やその計算システムに依存しない。具体的な計算は、関数引数として与えられる。しかしながら、与えられる関数引数には、ある特定の条件を満たすことが要求されることは当然であり、スキーマにもその関数引数の満たすべき抽象計算システム(代数構造)

が必要である。これをスキーマに付随する抽象計算システム (schematic system) と呼ぶことにする。例えば、ある並べ換えのスキーマには、並べ換える順序を規定するために線形空間が定義されていなければならない。実際の計算では、この付随する抽象計算システムは具体的な計算システム (concrete system) で置き換えられる。詳しくは、後章にて触れる。

### 3. プログラムの関数表現

以後の議論におけるプログラムはPascalで記述されているものとする。

#### 3-1. 関数の表現

ある関数の適用されるプログラム環境における変数の集合をVとする。関数Fnは、

$$Fn = [p, E] \quad (a)$$

または

$$Fn = [p, E, D, R] \quad (b)$$

で表現される。(a)の定義は関数をプログラム環境の様相間の写像と考える文献[3]のものと同一である。ここで、pはV上の述語で、EはV×V上の関係である。Vの値a,bにたいし、(a,b)∈Eは、P(a) = trueのとき、Fn(a) = bを意味する。また、Eの簡易な表現として値aにたいして(a,b)∈Eなるbを与える式を用いることもある。このときは、b = E(a)の関係にある。

例えば、整数型変数 x,y で x = 1,2,3 の値にたいし、y = 3,5,7 を与える関数は

$$Fn = \{[1 \leq x \leq 3], [y = 2x + 1]\}$$

となる。

とくに、fの内部でどの変数の値が参照され、どの変数の値が定義されたかを問題とするときには、(b)の定義を用いる。ここで、DはpおよびEの計算で参照される変数の集合であり、Rはこの計算の途上で値が再定義される変数の集合である。また、考慮している範囲がプログラムの全域に渡っているときには、関数内では値が定義しなおされているけれども、以後その値が参照されないということも有り得る。このような変数をRから除いて真の意味での値域という意味を持たしたい時には、RのかわりにR<sub>ef</sub>を用いることにする。

#### 3-2. 関数の演算

関数間につぎの3種の演算を導入する。これらの演算の定義は、文献[3]とほぼ同じであるので詳細は省略する。

##### (1) 複合演算 (composition)

$$[p, E] * [p', E'] = [p \wedge p', E \circ E']$$

ここで、E∘E'でしめされる演算はEを実行した結果にさらにE'を適用することで、E'の中の変数を、Eを代入して得られる式である。

##### (2) 統合演算 (union)

$$[p, E] \cup [p', E'] = [p \vee p', \text{ite}(p, E, E')]$$

ここではp∧p' = falseが常に成り立っているものとする。(文献[3]の定義と多少異なる。) ite(p, E, E')はp = trueなる引数にはEを、p = false のときにはE'を適用する式である。

##### (3) 繰り返し演算 (loop)

$$f = q[p, E] \text{ とおくと、}$$

$$f = [\neg p, \text{id}] \cup [q, E] * f$$

と再帰的に定義される。[q, E] \* fで表現される関数の定義域は、この式を意味あるものとする値の集合、即ち q = true から [q, E] \* f を終端させない値を除いた範囲と定義する。

関数式の中での暗黙の優先順位は。> \* > ∪ の順とする。

#### 3-3. プログラムの関数化

プログラムの実行部分の関数化については、Pascalの各々の実行文について関数化が可能であるが、ここでは空文、代入文、IF文、WHILE文、procedureの引用だけに制限して話を進める。実際のプログラムでは、データ・フロー解析を可能とするために、すべてのプログラムは既約でなければならないという制限がつけられている。以下の式では左辺の関数記号内のPascalの文は、右辺の関数のかたちで記述される。

$$(a1) \quad [] = [\text{true}, \text{id}]$$

ここでidはVの全域で定義され何もしない恒等式とである。

$$(a2) \quad [\text{begin st end}] = [\text{st}]$$

$$(a3) \quad [\text{st1; st2}] = [\text{st1}] * [\text{st2}]$$

$$(a4) \quad [\text{if p then st1 else st2}] \\ = [p, \text{id}] * [\text{st1}] \cup [\neg p, \text{id}] * [\text{st2}]$$

$$(a5) \quad [\text{while p do st}] = p * [\text{st}]$$

$$(a6) \quad [\text{pfname}] = [\text{begin st1; st2; } \dots, \text{stn end}]$$

但し、上式の右辺は、関数または手順の定

義の中の実行部分であり、適当な環境の変換と仮引数と実引数の引き渡しが行なわれているものとする。

#### 4. スキーマの表現

##### 4-1. スキーマの付随計算システム

$A = (X, OP)$  をあるスキーマに付随する抽象計算システムとする。 $X$  は対象となるデータの集合、すなわち1つの台、 $OP$  は  $X$  の上の演算の集合である。

スキーマの中では、 $A$  における計算のみが許されているから、 $A$  の階層はそのままスキーマの階層となって表れる。スキーマに複雑な付随計算システムを許せば、一般のプログラムとスキーマとの間には明確な境界はなくなってしまう。そこで、ここではスキーマに付随する計算システムを次に3種類のものに制限した。

##### (1) 基本計算システム BS (basic system)

一般のPascalにおけるデータの集合であり、これらのデータの上にはデータ間の同等性(=)と、そのデータを直接構成している要素へのアクセスのための記法 $v$ 、および全てのPascalの基本型データの上の演算(整数型では、 $+$ ,  $-$ ,  $*$ ,  $div$  その他では、 $=$ ,  $<$ ,  $=$ ,  $pred$ ,  $succ$  など)が定義されている。 $v$  記法は  $X$  質異本型のときは定義されず、 $X$  が配列型のとき  $X3$  は  $X[3]$  を、 $X$  がレコード型のとき  $XvA$  は  $X.A$  をあらわしている。はこのシステムは各々の基本データ型での演算が定まっているので、単に台となるデータ型を規定するだけでやれば良いので

$BS = (X)$

で与えられる。 $X$  は対象となる要素の型定義か型名を表現するための仮引数である。この計算システムは、file型を除くPascalで扱い得る全てのデータ型が対象となりうる。

##### (2) 抽象集合計算システム SS (abstract set system)

Pascalにおける集合型とは異なって、集合の要素となるデータには何の制限も無い。(1)における基本計算システムの上に定義することが出来る。但し、基本要素の集合  $X$  が表現する要素は無限集合であっても良いが、その上の集合は有限でなければならない。 $null$  は空集合を表現している0項演算としてあつてはいる。

$SS = (X, \{null, union, intersect, elem, add, delete\})$

また、 $X$  を具体的な計算システムに対応づけたとき、 $X$  の表現している集合が有限集合を表現している時には、演算に  $not$  が追加される。

##### (3) リスト計算システム LS (list system)

リスト計算システムは、前の抽象集合システムとほぼ同じような計算システムであるが、集合の各要素に線形な順序関係を定義し、要素のアクセスの方法を制限している。

$LS = (X, \{null, cons, car, cdr, equal\})$

ここで、 $X$  の各要素はリストの要素となり得るものを定義している。 $null$  は空リストを表わす仮引数である。リストへの各演算はLISPの各関数と同じ名前を使用しており、機能も同じである。

##### 4-2. スキーマの表現

スキーマは、Pascalには無い機能であるが、procedureと類似していることから、説明上同じような記法を用いる。

スキーマはprocedureと同様にプログラムの中に定義部と幾つかの引用部とがある。定義は、

```
schema name ( argument_list );
...
begin
...
end
```

と記述される。仮引数のリストには、普通のPascalで許されているものの外に、スキーマに付随する計算システムが許されている。また、**begin** **end** 間の実行部分のプログラムで使用される演算は全て仮引数の付随計算システムで定義されたものでなければならない。

スキーマの適用はPascalのprocedureと同じように

$name(argument\_list)$

で表現され、仮引数が付随計算システムの位置には、具体的な計算システム(concrete system)が配置される。仮引数と実引数とのデータの引き渡し時には、普通の型のデータの場合に型のチェックが行われるように、実引数の計算システムが仮引数の付随計算シ

システムの構造を包含しているか否かのチェックが行われる。この計算システムの引数の引き渡しをシステム的具体化(system specification)と呼んでいる。

スキーマの実際上の表現方法は、引数の引き渡し時にシステム的具体化を必要とする新しい構造型が導入されたものとすれば、一般のプログラムと同じように関数表現が可能である。

例

リストの長さを求めるリスト計算システム上のスキーマは次のように与えられる。

```

schema length(l: list; var n: integer;
               LS: list__system);
  var lst: list;
  begin
    if l = null then n := 0
    else n := length(cdr(l)) + 1
  end;

```

これにたいして、リスト上の具体的計算システムは、

```

type
  personlist = ↑ personrec;
  personrec = record
    p: person;
    next: personlist
  end;

systemdef A; list__system;
  X = personrec;
  null = nil;
  ...
  cons = function consperson
    (var lst: personlist; a: person);
    var brec: personrec;
    begin
      new(x);
      brec ↑ .p := a;
      brec ↑ .next := lst;
      lst := brec
    end;
  ...
enddef;

```

として、この具体的なシステムAによってLSを具体化すると、

```

X ↔ personrec
null ↔ nil
...
cons ↔ consperson

```

...

と対応づけられスキーマlengthの中のnullはnillとして、また演算consは動作時にはconspersonが実行される。(例終)

## 5. プログラムの変換とスキーマの抽出

プログラムの変換の方法は、色々な手法で検討されているが[4],[5]、規則を効率よく適用するアルゴリズムはおろか、完全(complete)な変換システムさえ求められていない。

プログラムの抽象化のために有用なプログラムの書き換え規則がいくつか求められているが、ここでは、スキーマ抽出に重点を置くこととして、\*演算に関する可換性の法則だけを紹介しておく。

2つの関数

$[p, E, D, R]$  および  $[p', E', D', R']$

にたいして、

$$\begin{aligned}
 R \cap D' &= \emptyset \\
 R' \cap D &= \emptyset \\
 D_{ef} \cap D'_{ef} &= \emptyset \\
 \frac{[p, E, D, R] * [p', E', D', R']}{[p, E, D, R] : [p', E', D', R']}
 \end{aligned}$$

ここで記号:は

$$f: g \leftrightarrow f * g = g * f$$

を意味している。

例

```

a _____
  sum := 0;
  for i := 1 to maxindex do
    sum := sum + A[j, i];
  colsum[j] := sum
b _____

```

なるプログラム部分に対応する関数を

$$f_{ab} = (p, E, D, R)$$

とおくと、この区間での変数集合Vは{sum, i, A[j, 1], A[j, 2], ..., A[j, maxindex], colsum[j], maxindex} となる。

$$D = \{A[j, 1], A[j, 2], \dots, A[j, \text{maxindex}], \text{maxindex: integer}\}$$

$$R = \{\text{colsum[j]}\}$$

であるからsum, iは地域変数とでき、また i によって区別されるデータはまとめて次のスキーマが得られる。

```

schema example (datatype:basic__system;
  A:datatype;var y:integer;maxindex:integer;);
var
  A:datatype; sum:integer;i:1..maxindex
begin
  sum := 0;
  for i:=1 to maxindex do
    sum := sum + A[i];
  y := sum
end;

```

(例終)

## 6 ボトムアップ型のプログラム抽象化システム

図2にこの計画のシステムのブロックダイアグ

ラムを示した。既に開発されたフロントエンド部は、LL1コンパイラコンパイラZuseを基本とし、構文解析の理論に出来るだけ忠実に動作することを目的としている[6]。また、symbol tableは、従来のトークンなどの情報の外に、プログラムブロックの構造を始め、プログラムにおいて宣言された全ての情報が格納されている。このことは、プログラムの関数化などの後の処理においてプログラムの実行部だけを対象にできることを意味している。

スキーマ抽出の鍵となるデータフロー解析部は、原理に基づくアルゴリズムでは動作するが、実用的に利用するには計算時間がかかりすぎるきらいがある。その主たる原因は、多くのアルゴリズムがただ1つの変数を対象にしたものであるからで、現在複数の変数に対して処理可能なアルゴリズムを検討中である。

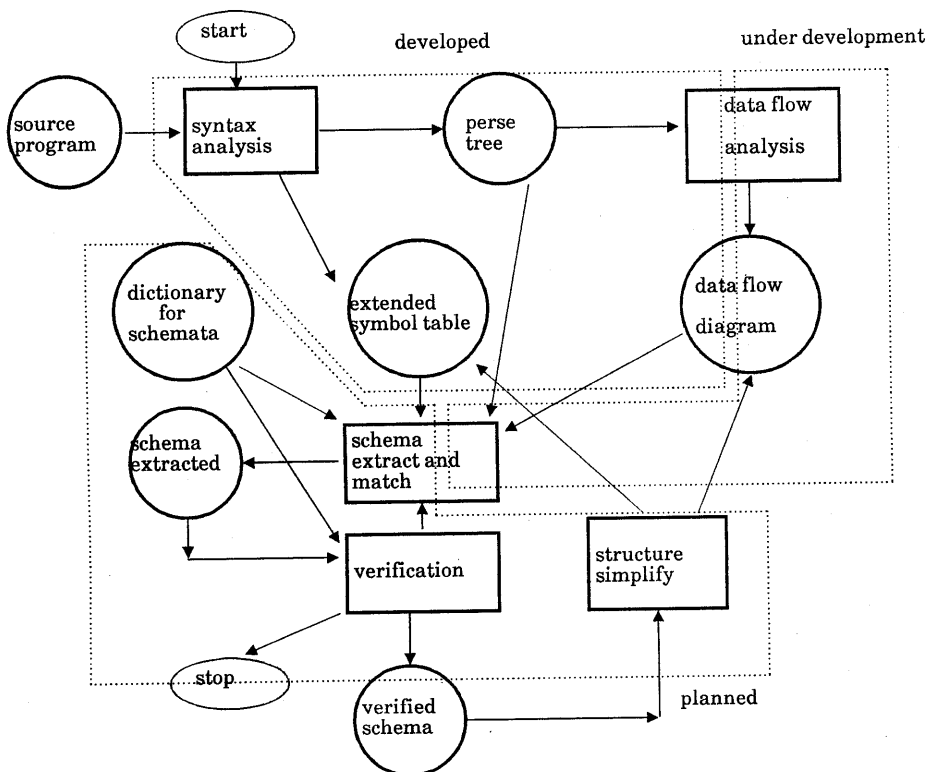


図2 ボトムアップによるプログラム検証システム

スキーマ抽出部は、現在前述の $f_{ab}$ を求める部分を実装中である。ここではスキーマの付随計算システムが小さくとってあるので、抽象化の階層が多くなる。このことは、関数化するプログラムの境界 $a, b$ の範囲を大きく取れないことになる。 $a, b$ の最適な選び方の指針としては、許される範囲内で出来るだけ多くの部分変数がとれ、データの抽象化の効果が大きいところを選ぶべきであるが、明確な解析はされていない。データフローやコントロールフローのループの起点(規約プログラムでは起点と終点とは同一で唯一である)を中心に探索するのが得策のようである。

スキーマアの辞書についてはその大部分が計画中で論理的構造も定まっていない。しかし、スキーマの付随計算システムを小さく押えることにより、スキーマの分類がしやすくなり、不十分な抽出スキーマからもそれに近似した標準スキーマが検索出来ると予測している。抽出スキーマと標準スキーマとのmatchingには、論理的なプログラム検証の方法を用いるが、検証の対象となるassertionは標準スキーマに付随して用意されている。全てのassertionが検証された時点で1段階の抽象化が完成する。

## 7. おわりに

プログラム抽象化の方法については、プログラムの機能を中心としたトップダウン的アプローチが多いようであるが、抽象的なプログラムの階層間の関係は、単に機能の細分だけではなく実装の方法も深く係わっているという前提がこの研究の出発点である。

プログラムの実装は、使用される言語に大きく依存するが、ある特定の言語に固定すればやはり抽象化、階層化の対象となり得る。実装の抽象モデルとしてスキーマを導入し計算の機能としての抽象モデルとの関係を定式化した。単なる仮定から出発し、厳密な体系の下に構築されたしすてむではなく、実証システムも未だ開発途上であることからシステムの有効性、実用性は全く不明であるが、初学者程度のプログラムの検証の実用化を目指している。

諸氏のきたんのない御意見、御助言が戴ければ幸いである。

## 参考文献

- [1] Reynolds, C. and Yeh, R.T. : Induction as the basis for program verification, IEEE Trans. SE-2, pp 224-252 (1976).
- [2] Kotoh, K. and Yamada, H. : Reference manual for STOP: A structure oriented processor, Tech. Rep. 76-08, Dept. Inf. Science, Univ of Tokyo (1979).
- [3] Mili, A. : An introduction to formal program verification, Van Nostrand (1984).
- [4] Cheatham, T.E. and et al. : Program refinement by transformation, 5-th Conf. on Soft. Eng., pp 430-347 (1981).
- [5] Burstall, R.M. and Darlington, J.L. : A transformation system for developing recursive programs, JACM 24, pp 44-68 (1977).
- [6] 宮寺、古東、片岡 : 意味構造解析用パーサーを用いたプログラムの抽象化, 情報処理学会第32回全大, pp 571-572 (1986).