集合指向言語 SOL の拡張とフローグラフのインターバル解析への応用

重松保弘† 吉田 将††

集合指向言語 SOL は，アルゴリズムの自然なプログラム化を目的として筆者らが開発した言語であるが，有向グラフなどのデータ構造を操作するアルゴリズムの記述が複雑化し，計算時間の増加や記憶領域使用効率の低下をもたらす場合が出てくるという問題点があった．これは，SOL が多値関数の記法を言語仕様に含まないことに起因する．また，集合式の制限が強いという問題点もあった．そこで SOL に多値関数である対応の概念を導入することによって言語仕様を拡張し，言語の記述能力の改善を行った．具体的には，対応，逆対応および逆写像の記法の導入，対応の導入に伴う対応定義文の拡張，対応操作文の導入などである．集合式については，内包記法の拡張と範囲指定形式の追加を行った．また，拡張された言語仕様に対応して，新たに言語処理系の開発を行った．本論文では，SOL の拡張言語仕様について述べると共に，フローグラフのデータフロー解析に利用されるインターバル解析アルゴリズムと，これを用いて簡約可能なフローグラフの導出順序を計算する手続きを取り上げ，これが拡張した SOL で，旧 SOL と比べ簡潔かつ効率よくプログラム化できることを示す．

An Extension of the Set Oriented Language SOL and Its Application to Interval Analysis of Flow Graphs

Yasuhiro Shigematsu† and Sho Yoshida††

SOL is one of the set oriented very high-level class languages designed and developed for writing algorithms as computer programs in a simple and natural way. However, it was not efficient for SOL to represent algorithms which manipulate complex data structures, such as directed graphs. This is caused by the fact that notations of multi-valued functions were not included in its language specification. Restrictions of the notation of set data forms were another problem. To solve these problems, we have introduced the concept of correspondence relation into SOL. To be concrete, we have introduced correspondence and inverse correspondence notations, and also introduced several statements for the definition and manipulation of correspondence relations. We have also introduced a subrange form and a new set-builder form into set data forms. In this paper, the language specification of extended SOL is provided in detail. Using extended SOL, we also show that algorithms based on the interval analysis of flow graphs are able to be expressed as simple and efficient SOL programs.

1. まえがき

抽象的なデータとその集合を対象とするアルゴリズムの自然なプログラム化を目的として，筆者らは集合指向言語 SOL とその言語処理系を設計・開発した[6]．SOL の主な特徴は，① PASCAL 風な手続き型言語，②集合間に写像が定義できる，③写像の動的な変更が可能，④集合・写像データの入出力が可能，⑤集合族が扱える，⑥外延・内包記法による動的な集合の生成が可能，⑦集合演算子，全称・存在論理記号などに数学上の記法が使える，などである．

筆者らは，SOL をグラフアルゴリズム[5]や関係データベース言語[9]へ応用し，その有効性を確認してきたが，問題点も発見された．最も大きい問題点は，SOL では写像，すなわち一価関数，しか定義できないことである．このため，有向グラフなどのデータ構造は節集合から直接後行（先行）節集合族への写像という形式で表現せざるをえなかった．しかし，こうした表現形式をとると，有向グラフ上の節探索，節や矢の追加・削除処理などの操作手順の記述が複雑になるうえ，計算時間の増加や記憶領域使用効率の低下を招く

† 九州工業大学工学部電気工学科
Department of Electrical, Electronic and Computer Engineering, Faculty of Engineering, Kyushu Institute of Technology
†† 九州工業大学情報工学部知能情報工学科
Department of Artificial Intelligence, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology

ことになる.

そこで，筆者らは，SOL に多値関数である対応[1]の概念を導入することによって，この問題点を解決した[8]．具体的には，対応，逆対応の記法，および対応定義文と2つの対応操作文を言語仕様に取り入れ，有向グラフを対応記法で表現できるように拡張した（以後，必要に応じて，これまでの SOL を旧 SOL，拡張した SOL を拡張 SOL と呼ぶ）.

SOL の“対応”は SETL[4] の multi-valued map と同じ概念である．しかし，SOL では写像と対応のいずれについても始集合と終集合の指定ができるのに対して SETL では single-valued map，multi-valued map 共に定義域と値域の型（mode と呼ばれる）を指定することになっている．mode には base set と呼ぶ特殊な集合を指定することもできるが，この集合は SETL の通常の集合としては扱えない（式や実行文中に現れることも許されない）．したがって，SOL の写像と対応は SETL の map に比べ，応用プログラムの記述においてより柔軟であるといえる．また，逆写像と逆対応にあたる記法は SETL には存在しない.

拡張 SOL では，対応記法の導入のほかに，集合の表現形式として範囲指定形式の導入と内包記法の拡張を行った．また，写像（対応）の始集合と終集合の（代入文などによる）再定義がもたらす写像（対応）関係への影響を明確にした．この他，構文規則の整理[10]を行った（付録参照）.

SOL の言語仕様の拡張にともない，SOL 言語処理系についても変更・拡張を行った．その主なものは，対応の導入に対する内部データ構造の変更と対応操作文などの処理の追加，集合記法の拡張に伴う処理の追加，中間コードである S コードの追加，内部データ構造の基本となるセルの塵集め処理の追加，などである[10].

本論文では，拡張 SOL の言語仕様について述べると共に，フローグラフのインターバル解析への応用例を通じて，拡張 SOL がフローグラフの表現能力と処理効率の点で旧 SOL より優れていることを示す.

2. 拡張 SOL の言語仕様

2.1 集合の表現形式

集合は次の5通りの形式で表現される．このうち，**範囲指定形式**と**内包形式2**は新たに追加した形式である.

（外延形式）　{式, …}
（範囲指定形式）　{定数～定数}
（内包形式1）　{束縛式|論理式}
（内包形式2）　{式|束縛式, …, 論理式}
（空集合定数）　ϕ

外延形式（tabular form）は，集合の要素を列挙するものである．範囲指定形式の定数は，整数型または文字型でなければならない．束縛式は，”束縛変数 ∈ 式”の形式をとる．内包形式（set-builder form）1は，論理式の値を満足する束縛変数の値の集合を返す．内包形式2は，論理式を満足する束縛変数の値について式を評価し，式の値の集合を返す．以下に例を示す.

（外延形式）　{1, 3, 6, 7, 8}, {$m+n$, $m-n$, $m*n$}
（範囲指定形式）　{−10～10}, {'a'～'z'}
（内包形式1）　{$x \in \{1\sim100\} \mid x \bmod 3=0$}
（内包形式2）　{$m*n \mid m \in \{2\sim7\}$, $n \in \{2\sim25\}$, $m*n<51$}

2.2 写　像

2.2.1 写像の定義

写像に関しては，その定義方法について意味的な一貫性が保てるよう部分的な変更を加えた.

写像を宣言する場合は，次に示すように，あらかじめ始集合と終集合を集合変数の形で宣言しておかねばならない（以下，特に断わらない限り D, R, f は，この形式で宣言されているとする）.

```
var D : setof integer;
    R : setof string;
map f : D→R;
```

ここで f は，整数集合型変数 D を始集合とし，文字列集合型変数 R を終集合とする写像として宣言されている．このとき，写像 f は defmap 文，代入文，または，入力関数のいずれかを用いて定義できる．例を次に示す.

（1） **defmap** f(1)=”one”
（2） f←{[1, ”one”], [2, ”two”], [3, ”one”]}
（3） read(f) または read (”filename”, f)

（1）の defmap 文は，これまでと同じ形式であり，1と”one”の間に写像関係が定義される．（2）の代入文では，これまで矢印の右辺には特別な形式の写像定数を置くことにしていた[6]．しかし，写像は定義域の要素と像の組集合で表すのが自然であることから，構文を変更して2項組の集合を評価値とする式を置くことにした．（3）の前者は標準入力から，後者はファイル filename から，それぞれ写像データを f に入力する.

旧 SOL では，写像を代入文によって定義する場合，定義域の要素と像の値は，各々，始集合と終集合に含まれていなければならないと規定していた[6]．これは，誤った写像定義文の実行によって始集合や終集合が変更されないために与えた制限であった．しかし，写像の始集合と定義域☆，または，終集合と値域が一致するときはかえってプログラムが冗長になること，および，始集合や終集合が変更できた方が都合がよい場合があることなどから，拡張 SOL では，この制限は撤廃することにした．したがって，(2)の代入文の実行前に始集合と終集合の値を定義する必要はなくなった．(1)と(3)の場合も，同様の理由から始集合と終集合の値を定義しておく必要はない．

2.2.2 写像の変更

既に定義された写像は，次のいずれかの方法で変更できる．

(1) defmap 文，代入文，入力関数による再定義

(2) 始集合または終集合の再定義

defmap 文による再定義はこれまでと同じであり，既に存在する写像関係を削除した後，新たな写像関係を定義する[6]．代入文の例を次に示す．

(1-1) $f \leftarrow \Phi$ または $f \leftarrow \{\ \}$

(1-2) $f \leftarrow$ {[1, "two"], [5, "five"]}

(1-1) は写像関係 f をすべて消滅させる．ただし，始集合 D と終集合 R の値は変化しない．(1-2) は，写像関係 f をすべて消滅させた後，再度，写像関係を定義する．

(2)の場合の例を次に示す．**図1**の状態で，次の文を実行するとする．

(2-1) $D \leftarrow D - \{2\}$ または $D \leftarrow \{1, 3, 4\}$

(2-2) $R \leftarrow R -$ {"one"}

または $R \leftarrow$ {"two", "three"}

(2-1) は始集合から要素2を取り除く代入文である．

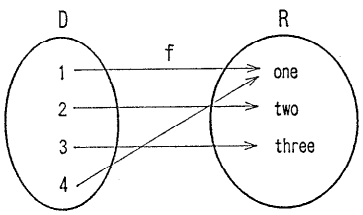


図 1 写像の例
Fig. 1 An example of mapping relation.

☆ 数学上，写像の始集合は定義域に一致することになっているが，SOL では map 宣言された写像は対応としても使用できるので，SOL 処理系はこの一致を検査しない（取扱いは利用者に任されている）．

これにともなって2から"two"への写像関係も削滅する．ただし，"two" は R の要素として残る．(2-2) は終集合から要素"one"を削除する代入文である．この場合は，1および4から "one" への写像関係が同時に消滅する．

拡張 SOL では，N が写像の始集合または終集合の場合，代入文 "$N \leftarrow M$" によって $N \cap M$ の要素に関する写像関係の保存を保証し，$N \cup M - N \cap M$ の要素に定義されていた写像関係の削除を保証する（旧 SOL では保証していなかった）．

2.3 対　　応

2.3.1 対応の宣言

対応は，写像の宣言によって暗黙のうちに宣言される．たとえば，2.2.1 項における map 宣言は写像（写像名 f）の宣言であるが，これによって暗黙のうちに対応（対応名 $f*$）を宣言したことになる．プログラム中で f と $f*$ のいずれを使用するかは，定義される関係によって決まる．すなわち，一価の関係を扱うときは f を，多値の関係を扱うときは $f*$ を，各々，使用する．なお，一価の対応は写像を意味するので，このときは f と $f*$ のいずれを用いてもかまわない（ただし，f は終集合の要素を返すが，$f*$ は終集合の部分集合を返す）．

2.3.2 対応の定義

対応関係は defmap 文，代入文，入力関数のいずれかで定義する．次に例を示す．

(1) **defmap** $f*(1) =$ {"one", "two"}

(2) $f* \leftarrow$ {[1, {"one", "two"}], [2, {"two"}],
[3, {"one", "three"}]}

(3) read($f*$) または read ("filename", $f*$)

(1)の defmap 文によって1と "one"，および1と "two" の対応関係が定義される．(2)の代入文によって生成される対応のグラフを**図2**に示す．この対応関係は多値なので，写像 f は無意味なものとなる（すなわち，f の宣言は $f*$ の宣言のために利用されたにすぎない）．

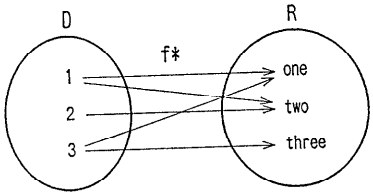


図 2 対応の例
Fig. 2 An example of correspondence relation.

2.3.3 対応の変更

対応は，次のいずれかで変更される．

（1） defmap 文，代入文，入力関数による再定義

（2） addmap 文による対応関係の追加

（3） delmap 文による対応関係の削除

（4） 始集合または終集合の再定義

次に例を示す．

(1-1) **defmap** $f*(1)=\phi$ または { }

(1-2) **defmap** $f*(1)=$ {"one", "three"}

(1-1) は写像の場合と同様，対応関係をたんに消去するものである．また，(1-2) の defmap 文は，いったん対応関係を消去した後，改めて指定された対応関係を生成する．したがって，この文を実行すると，図2の対応関係が**図3**のように変更される．代入文についても，写像の変更の場合と同様の結果が得られる．次に例を示す．

(1-3) $f*\leftarrow\phi$ または $f*\leftarrow$ { }

(1-4) $f*\leftarrow$ {[3, {"one", "two"}], [5, {"five"}]}

(1-3) は対応 $f*$ をすべて削除するものである．(1-4) は対応関係をすべて削除した後，再定義するものである．

addmap 文と delmap 文の一般的な書式は次のようになっている．

addmap f(式$_d$)＝式$_r$

delmap f(式$_d$)＝式$_r$

また，これらの文は，各々

defmap $f*$(式$_d$)＝$f*$(式$_d$)∪{式$_r$}

defmap $f*$(式$_d$)＝$f*$(式$_d$)−{式$_r$}

と等価である．以下に例を示す．

(2-1) **addmap** $f(1)=$"three"

(3-1) **delmap** $f(1)=$"two"

これらの文は，各々，次の defmap 文に等しい．

defmap $f*(1)=f*(1)\cup$ {"three"}

defmap $f*(1)=f*(1)-$ {"two"}

したがって，図2の状態で (2-1) と (3-1) の文が順次実行されるが，図3の対応関係に変更される．

始集合と終集合の再定義による対応関係の変更は，2.2.2 項で述べた写像関係の変更と同じ効果をもたらす．

2.4 逆写像と逆対応

集合 D から R への写像 f が宣言されているとき，R から D への逆対応 $f*^{-1}$ を次のように定義する．

$f*^{-1}(r)$：集合 $\{d \mid d\in D,\ r\in f*(d)\}$ を返す

逆写像は，写像が1対1対応のときのみ意味をもつが，便宜上，逆写像 f^{-1} を次のように定義する．

$f^{-1}(r)$：$f*^{-1}(r)$ の1つの要素を返す

2.5 写像名と対応名の参照

式のなかで写像名，対応名，逆写像名および逆対応名を参照すると，それぞれ，定義域の要素と値域の要素(像)の組集合，定義域の要素と値域の要素集合(像)の組集合，値域の要素と定義域の要素（原像）の組集合，値域の要素と定義域の要素集合（原像）の組集合，を返す．**図4**に，対応名と逆対応名の参照の例を示す．

3. インターバル解析への応用例

3.1 インターバル解析アルゴリズムの記述

ここでは，フローグラフ (flow graph) のインターバル解析 (interval analysis) アルゴリズム，および，このアルゴリズムを用いて簡約可能なフローグラフの導出順序を計算する手続きをSOLで記述する例を示す．

フローグラフは，プログラムを制御の流れに着目してグラフ化した制御フローグラフ (Control flow

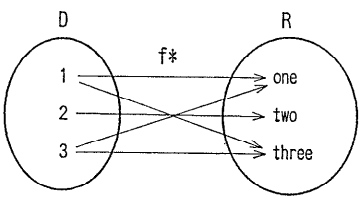


図3 図2の変更によって得られた対応関係
Fig. 3 New correspondence relation obtained by a modification of Fig. 2.

(プログラム)

```
program inverse;
var A,B : setof integer;
map f : A → B;
begin
  f* ← {[1,{10,20}],[2,{20,30,40}]};
  writeln("f*  =",f*);
  writeln("f*-1 =",f*-1);
  f* ← f*∪{[3,{10,30}]};
  writeln("f*  =",f*);
  writeln("f*-1 =",f*-1);
end.
```

(出力結果)

```
f*  ={ [  1, {  10,  20 } ], [  2, {  20,  30,  40 } ] }
f*-1={ [  10, {  1 } ], [  20, {  1,  2 } ], [  30, {  2 } ], [  40, {  2 } ] }
f*  ={ [  1, {  10,  20 } ], [  2, {  20,  30,  40 } ], [  3, {  10,  30 } ] }
f*-1={ [  10, {  1,  3 } ], [  20, {  1,  2 } ], [  30, {  2,  3 } ], [  40, {  2 } ] }
```

図4 対応名と逆対応名の参照の例
Fig. 4 An example of reference of correspondence name and inverse correspondence name.

graph) の抽象モデルである．フローグラフ G は節(node) の集合 N, 矢 (arc) の集合 A および入口節 (entry node) s の3つ組 (N, A, s) で表現される．インターバルは，フローグラフを，そのループ構造に着目して分割した部分グラフであり，データフロー解析 (data flow analysis) に利用される．G の1つの節を h とするとき，ヘッダ (header) を h とするインターバル $I(h)$ は**図5**(a)のアルゴリズムで構成される[2]．この定義によって，フローグラフをインターバルに分割するアルゴリズム[2]を図5(b)に示す．**図6**は，フローグラフを，このアルゴリズムによってインターバル分割した例である．

図5のアルゴリズムを SOL で記述したプログラム例と実行結果を**図7**に示す．図7のプログラム中で使用しているデータは図6のフローグラフである．フローグラフ G の構成要素 N, A, s は，図7では集合変数 N, 対応 $A*$, 整数型変数 s で，各々，表現される．$A*$ への最初の代入文によって図6のフローグラフが生成される．また，フローグラフにおける直接後行節 (immediate successor) 集合と直接先行節 (immediate predecessor) 集合は，節番号の入っている変数を m とすると $A*(m)$ と $A*^{-1}(m)$ で，各々，表現できる．図5(a)における条件 "all arcs entering m leaves nodes in $I(h)$" は，条件 "m の直接先行節集合が $I(h)$ の部分集合である $(A*^{-1}(m) \subset I(h))$" と同値である．なお，図7中の関数 max は引き数となる集合の最大値を返し，getel は引き数となる集合の要素を1つ返す (要素は集合から削除される)．

フローグラフ $G=(N, A, s)$ が与えられたとき，次の3つの性質を持つフローグラフを G の導出フローグラフ (derived flow graph) と呼び，$I(G)$ で表す[2]．

(1) $I(G)$ の節は G のインターバルである．

(2) インターバル J と $K(J \neq K)$ について，J 中の節から K のヘッダへの矢があれば，J に対応する $I(G)$ の節から K に対応する $I(G)$ の節への矢が存在する．

(3) $I(G)$ の入口節は $I(s)$ である．

$G(=G_0)$ から導出フローグラフ $G_1(=I(G_0))$ を構成し，さらに G_1 から導出フローグラフ $G_2(=I(G_1))$ を構成するという手順を繰り返すことにより導出順序 (derived sequence) $G_0, G_1, \cdots G_k$ が構成される．ここ

```
I(h) := {h} /* initially */
while ∃ a node m such that m∉I(h) ∧ m≠s ∧ all arcs entering m
  leave nodes in I(h) do
I(h) := I(h)∪{m}
endwhile
```

(a) インターバルの構成アルゴリズム
(a) Algorithm for constructing an interval.

```
procedure FIND$INTERVALS(flow graph G=(N,A,s))
  sets H, /* set of potential header nodes */
       L, /* set of intervals */
  H := {s}
  L := ∅
  while H≠∅ do
    Select and delete a node h from H.
    Compute I(h) from the definition of interval.
    Add I(h) to L. /* L is a set of sets. */
    Add to H any node that has a predecessor in I(h), but that is not already in
    H or in one of the intervals of L.
  endwhile
  Output L.
return
```

(b) フローグラフをインターバルに分割するアルゴリズム
(b) Algorithm for partitioning a flow graph into intervals using the algorithm of (a).

図5 フローグラフのインターバル分割のアルゴリズム (参考文献 2) より抜粋)
Fig. 5 Algorithm for partitioning a flow graph into intervals.

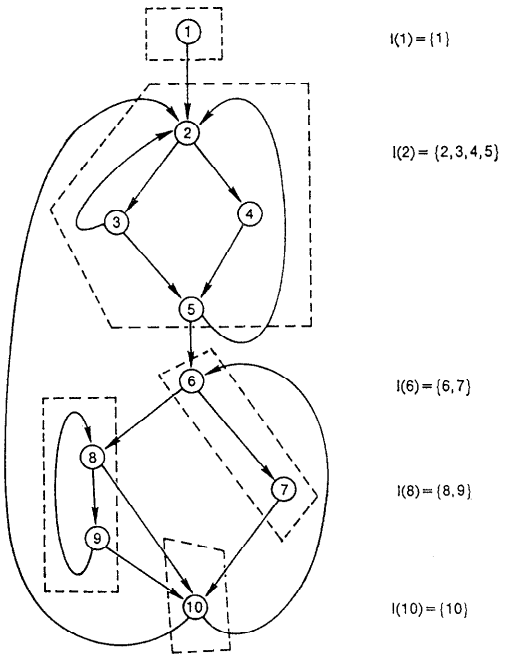


図6 フローグラフのインターバル分割の例
Fig. 6 An example of interval partitioning of a flow graph.

で，$I(G_k)=G_k$ であり，このとき G_k は極限フローグラフ（limit flow graph）と呼ばれる．極限フローグラフが単一の節から成り，矢を持たないとき，もとのフローグラフは簡約可能（reducible）と呼ばれる[2]．**図8**は，簡約可能なフローグラフの導出順序の例である．

導出フローグラフを構成する手続きは，図7のプログラムを拡張することで実現できる．**図9**に，簡約可能なフローグラフの導出順序を計算する SOL プログラム例と図8のフローグラフ G_0 をデータとして与えたときの実行結果を示す．図9において，写像 D は，導出フローグラフの節から対応するインターバルのヘッダへの写像を記憶しておくために用いる．手続き find_intervals では，インターバルを検出するごとに導出フローグラフの節（変数 n）を作り，n からインターバルのヘッダ（変数 h）への写像を定義している（これは，文 defmap $D(n)=h$ で実現される）．導出フローグラフの節 n を N に追加すると "s から N のすべての節への経路が存在する" というインターバル分割の前提条件が成り立たなくなる．そのため，インターバル分割時に孤立節を除外する目的で条件 "$A*^{-1}(m)\neq\Phi$" を追加した．手続き generate_graph は，導出フログラフの矢を生成するものである．導出フローグラフの節 j に対応するインターバルは $I(D(j))$ で表されるので，その要素（変数 m）の直接後行節集合（$A*(m)$）に節 k のヘッダ（$D(k)$）が含まれれば j から k への矢を生成する（これは，文 addmap $A(j)=k$ で実現される）．図9の結果から，図8のフローグラフ G_0 から正しく導出フローグラフと導出順序が計算されていることがわかる．

3.2 旧 SOL との比較

ここでは，旧 SOL と拡張 SOL を用いてフローグラフを記述する場合の比較を行う．フローグラフの記述は大別してフローグラフの表現と処理に分けることができる．処理の基本的なものは，先行節と後行節の計算，矢の追加・削除，節の追加・削除などである．

旧 SOL の言語仕様では，フローグラフは節集合から直接後行節集合族（または直接先行節集合族）への写像として表現せざるを得なかった[6]．この形式で記述したインターバル解析プログラムの例を**図10**に示す．図10において，N は節集合，NS は直接後行節集合族，写像 succ は矢を各々表している．また，直接先行節集合は関数 pred で計算する．この表現方法における問題点は，pred のような手続きを陽に記述しなければならないという表現

```
 0 program  find_intervals;
 0 const    MAXNODE=10;
 0 var  m, h, i, s, last : integer;
 0      N : setof  integer;          /* set of nodes                  */
 0      H : setof  integer;          /* set of potential header nodes */
 0      I : indexedset(1~MAXNODE) of setof integer;
 0                                   /* set of intervals              */
 0 map  A : N → N;                   /* set of arcs                   */
 0 begin
 0   A* ← { [1,{2}],[2,{3,4}],[3,{2,5}],[4,{5}],[5,{2,6}],
 3          [6,{7,8}],[7,{10}],[8,{9,10}],[9,{8,10}],[10,{2,6}] };
99   s ← 1;  last ← max(N);
107  H ← {s};                       /* FIND INTERVALS                */
112  while H≠Φ do
117    h ← getel(H);                /* Select and delete a node h    */
121                                 /* from H.                       */
121    I(h) ← {h};                  /* Compute I(h) from the         */
128                                 /* definition of interval.       */
128    while ∃(m∈N)( (m∉I(h)) and (m≠s) and (A*^-1 (m)⊂I(h)) ) do
163      I(h) ← I(h) ∪ {m}
171    od;
177    writeln("I(",h:2,")=",I(h));
191    forall m∈I(h) do             /* Add to H any node that has    */
199      H ← H ∪ A*(m)              /* a predecessor in I(h),        */
208    od;
210    for i ← s to last do         /* but that is not already in    */
214      H ← H - I(i)               /* one of the intervals.         */
224    od
225  od
226 end.

I( 1)={  1 }
I( 2)={  2,  3,  4,  5 }
I( 6)={  6,  7 }
I( 8)={  8,  9 }
I(10)={  10 }

Used cells =       97
```

図7 インターバル解析プログラムと実行結果
Fig. 7 Interval analysis program written in SOL, and its execution result.

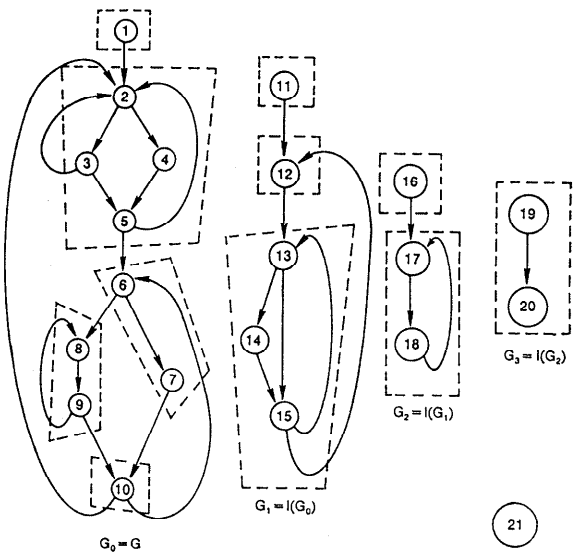


図8 簡約可能なフローグラフの導出順序の例
Fig. 8 An example of the derived sequence of a reducible flow graph.

上の冗長さに加えて計算時間の点で不利になることである（直接先行節を計算するために，図7の $A*(m)$ は始集合から m を探索するだけで済むのに対して，図 10 の pred(m) は終集合の各要素（集合）から m を探索しなければならない）．計算時間を短縮するために直接先行節集合族（NP）を宣言し，関数 pred を N から NP への写像に置き換える方法もあるが，これはフローグラフの二重定義になるので2倍近い記憶領域（消費セル数）を必要とするという点でやはり不利になる．

矢の追加・削除は，旧 SOL では defmap 文を用いて実現できる．たとえば節 p と q の間に矢を追加・削除するには次の文を各々，使用すればよい．

defmap succ(p)=succ(p)∪{q}

defmap succ(p)=succ(p)−{q}

関数 pred を写像に置き換えた場合は，さらに次の文が各々，必要になる．

defmap pred(q)=pred(q)∪{p}

defmap pred(q)=pred(q)−{p}

これらの defmap 文を用いる場合，問題となるのは終集合に不要な集合が蓄積されることである．前述の矢の追加・削除の場合を例にとると，defmap 文の等号の右辺の succ(p) の値を Q とすると，defmap 文の実行によって succ の終集合には集合 Q∪{q} または Q−{q} が追加される（Q は終集合から削除されない）．defmap 文の実行によって Q が succ の値域の要素でなくなった場合，Q の存在は（succ をフローグラフと解釈すれば）無意味であり削除されるべきである（Q は処理系のガーベジコレクションの対象外となるので，記憶領域の使用効率の点でも削除されるべ

```
0 program  interval_order;
0 const    MAXNODE=100;
0          MAXLEVEL=10;
0 var  s : indexedset (0 ～ MAXLEVEL) of integer;
0      I : indexedset (1 ～ MAXNODE)  of setof integer;
0      N : setof integer;
0      over : boolean;
0      level, last : integer;
0 map  A, D : N → N;
0
0   procedure find_intervals(s : integer);
0   var  h, i, m, n, last : integer;  H : setof integer;
0   begin
0     last ← max(N);      /* Remember the last node of a flow graph.   */
5     n ← last;
8     H ← {s};
13    while H≠Φ do
18      h ← getel(H);
22      n ← n+1;            /* Define mapping relation from higher level */
27      defmap D(n)=h;      /* flow graph to lower level flow graph.     */
34      I(h) ← {h};
41      while ∃(m∈N)( (m∉I(h)) and (m≠s) and
58                    (A*⁻¹(m)≠Φ) and (A*⁻¹(m)⊂I(h)) ) do
83        I(h) ← I(h) ∪ {m}
91      od;
97      writeln("D(",n:2,") = I(",h:2,") = ",I(h));
116     forall m∈I(h) do
124       H ← H ∪ A*(m)
129     od;
135     for i ← s to last do
139       H ← H - I(i)
145     od
150   od
151 end;
151
151 procedure generate_graph(first : integer; var over : boolean);
152 var  j, k, m, last : integer;
152 begin
152   last ← max(N);
157   for j ← first to last do
161     for k ← first to last do
165       if j≠k then               /* If there is any arc from  a node  */
169         forall m∈I(D(j)) do     /* in J to the header of K, and J≠K, */
180           if  D(k)∈A*(m)  then /* then make an arc from the node    */
191             addmap A(j)=k;      /* representing interval J to that   */
198             break               /* representing K.                   */
199         fi
199       od
201     fi
201   od
202 od;
203 for k ← first to last do
207   writeln("successor of node ",k:2," = ",A*(k))
219 od;
221 if first=last then over ← true fi
228 end;
228
```

```
228 begin
229   A* ← { [1,{2}],[2,{3,4}],[3,{2,5}],[4,{5}],[5,{2,6}],
232          [6,{7,8}],[7,{10}],[8,{9}],[9,{8,10}],[10,{2,6}] };
326   level ← 0;   s(level) ← 1;   last ← max(N);
339   over ← false;
342   while not over do
345     level ← level + 1;
350     writeln;  writeln("level = ",level:2);
357     find_intervals(s(level-1));
365     s(level) ← last+1;
372     last ← max(N);
377     generate_graph(s(level),over);
384   od
385 end.
```

```
level =  1
D(11) = I( 1) = {  1 }
D(12) = I( 2) = {  2,  3,  4,  5 }
D(13) = I( 6) = {  6,  7 }
D(14) = I( 8) = {  8,  9 }
D(15) = I(10) = {  10 }
successor of node 11 = {  12 }
successor of node 12 = {  13 }
successor of node 13 = {  14,  15 }
successor of node 14 = {  15 }
successor of node 15 = {  12,  13 }

level =  2
D(16) = I(11) = {  11 }
D(17) = I(12) = {  12 }
D(18) = I(13) = {  13,  14,  15 }
successor of node 16 = {  17 }
successor of node 17 = {  18 }
successor of node 18 = {  17 }

level =  3
D(19) = I(16) = {  16 }
D(20) = I(17) = {  17,  18 }
successor of node 19 = {  20 }
successor of node 20 = {  }

level =  4
D(21) = I(19) = {  19,  20 }
successor of node 21 = {  }

Used cells =       397
```

図 9 簡約可能なフローグラフの導出順序を計算するプログラム例とその実行結果

Fig. 9 An example SOL program for finding derived sequence of a reducible flow graph, and its execution result.

きである)．しかし，削除を実行するためには Q を像とする写像関係が全く存在しないことを確認しなければならず，このための手続きを利用者の責任で用意しなければならない．同じ問題は，写像 pred についても発生する．

これに対し，拡張 SOL ではフローグラフは対応 ($A*$) で表現されるので，矢の追加・削除は次の addmap 文と delmap 文を用いてより簡潔に実現される（図9のプログラム参照)．

addmap $A(p)=q$

delmap $A(p)=q$

この2つの文は，2.3.3 項で述べたように対応に対する defmap 文でも表現できるが，効率の点で defmap 文より優れている (defmap 文は，p と p の像との対応関係をすべて削除した後，再定義する)．また，これらの文は，いずれも写像に対する defmap 文と異なり不要な集合を生成することがない．

節集合変数 N への節 p の追加は，旧 SOL，拡張 SOL いずれも次の代入文で実現される．

$N \leftarrow N \cup \{p\}$

節の削除は，旧 SOL では表現上も計算時間上も不利である．たとえば，節集合変数 N から節 p を削除するには，次の3ステップが必要である．

(1) $N \leftarrow N - \{p\}$

(2) 写像 succ の終集合 (NS) のすべての要素（集合）から p を削除する．

(3) 関数 pred を写像に置き換えた場合は，写像 pred の終集合 (NP) のすべての要素（集合）から p を削除する．

これに対し，拡張 SOL では，上記3ステップ中，ステップ(1)のみで済むので表現上も計算時間上も効率的である．

```
 0 program  find_intervals;  /* old SOL version */
 0 const    MAXNODE=10;
 0 type setofint = setof integer;
 0 var  m, h, i, s, last : integer;
 0      N : setof  integer;            /* 節集合                              */
 0      NS: setof setof integer;       /* 直接後行節集合族                    */
 0      H : setof  integer;            /* set of potential header nodes       */
 0      I : indexedset (1~MAXNODE) of setof integer;
 0                                     /* set of intervals                    */
 0 map  succ : N → NS;                /* 節集合から直接後行節集合族への写像 */
 0
 0   function  pred(m:integer) : setofint;
 0   var  p : setof integer;
 0        n : integer;
 0   begin
 0     p ← Φ;
 3     forall  n∈N  do
 8       if  m∈succ(n)  then   p ← p ∪ {n}  fi
24     od;
26     pred ← p
27   end;
30
30 begin
31   N    ← {1,2,3,4,5,6,7,8,9,10};
54   NS   ← { {2},{3,4},{2,5},{5},{2,6},{7,8},{10},{9,10},{8,10},{2,6} };
111  succ ← { [1,{2}],[2,{3,4}],[3,{2,5}],[4,{5}],[5,{2,6}],
114           [6,{7,8}],[7,{10}],[8,{9,10}],[9,{8,10}],[10,{2,6}] };
210  s ← 1;  last ← MAXNODE;
216  H ← {s};                          /* FIND INTERVALS                  */
221  while H≠Φ do
226    h ← getel(H);                   /* Select and delete a node h      */
230                                    /* from H.                         */
230    I(h) ← {h};                     /* Compute I(h) from the           */
237                                    /* definition of interval.         */
237    while ∃(m∈N)( (m∉I(h)) and (m≠s) and (pred(m)⊂I(h)) ) do
271      I(h) ← I(h) ∪ {m}
279    od;
285    writeln("I(",h:2,")=",I(h));
299    forall m∈I(h) do                /* Add to H any node that has      */
307      H ← H ∪ succ(m)               /* a predecessor in I(h),          */
312    od;
318    for i ← s to last do            /* but that is not already in      */
322      H ← H - I(i)                  /* one of the intervals.           */
328    od
333  od
334 end.

I( 1)={  1 }
I( 2)={  2,  3,  4,  5 }
I( 6)={  6,  7 }
I( 8)={  8,  9 }
I(10)={  10 }

Used cells =      105
```

図 10 旧 SOL で記述したインターバル解析プログラムと実行結果
Fig. 10 Interval analysis program written in old version SOL and its execution result.

4. あとがき

SOL に対応の概念を導入し，言語仕様に対応，逆対応などの記法，および対応操作文を取り入れることにより，インターバル解析の例にみられるように，フローグラフを対象とするアルゴリズムを旧 SOL と比較して簡潔かつ効率的なプログラムとして実現できるようになった．SOL の応用上の目的の1つは，プログラムのデータフロー解析 (data flow analysis) アルゴリズムや最適化アルゴリズムを自然な形で簡潔にプログラム化することである．インターバル解析はデータフロー解析に応用される[2]ので，SOL の拡張によってこの目的は，ある程度，達成できたと思われる．

また，インターバル解析の例では使用されなかったが，新しい内包記法の導入によって，これまで forall 文による繰り返し形式で記述しなければならなかったプログラムを，集合式の形で簡潔に記述できるようになった．内包形式を構文規則に導入しているプログラミング言語には SETL，Lorel-2[3]，KRC[7] などがある．SETL と Lorel-2 は共に set former と呼ぶ記法を導入しているが，Lorel-2 は SETL に比べて構文上の制限が強い．KRC は ZF (Zermelo-Frankel) 式と呼ぶ集合抽象記法を導入しており，構文の最後の論理式に制限がない点では

Lorel-2 より柔軟性があるが，構文の種類の豊富さでは SETL に及ばない．SOL の内包形式は SETL に匹敵する豊富さを持っているうえ，SETL と比べて構文規則が数学上の記法により近いという特徴を持っており，集合処理プログラムの記述に有効であると思われる．

今後の課題は，データフロー解析アルゴリズムや最適化アルゴリズムを SOL プログラムの形で実現し，SOL の言語としての実用性を調べることである．なお，本文中の図4，図7，図9，図 10 は，いずれも Apollo 社の DN4000 ワークステーション上で稼働している拡張 SOL 処理系の出力結果である．

謝辞 日頃，ご討論いただく九州工業大学工学部の安在弘幸教授に感謝します．

参 考 文 献

1) 松坂和夫：集合・位相入門，pp. 22-29，岩波書店，東京 (1968).
2) Hecht, M. S.: *Flow Analysis of Computer Programs*, p. 232, North-Holland, New York (1977).
3) 榎本 (進)，宮地，片山，榎本 (肇)：Lorel-2 言語について，情報処理，Vol. 19, No. 6, pp. 522-530 (1978).
4) Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. and Schonberg, E.: *Programming with Sets (An Introduntion to SETL)*, p. 493, Springer-Verlag, New York (1986).
5) Shigematsu, Y., Yoshimi, K. and Yoshida, S.: Set Oriented Language SOL and Its Application to Graph Algorithms, *Proc. of ICS '88*, pp. 1135-1140 (1988).
6) 重松，吉見，吉田：集合指向言語 SOL とその言語処理系の開発，情報処理学会論文誌，Vol. 30, No. 3, pp. 357-365 (1989).
7) Gray, P. M. D.，田中ほか (訳)：論理・代数・データベース，pp. 101-102，産業図書，東京 (1990).
8) 松浦，重松，吉田：集合指向言語 SOL の拡張とプログラムフローグラフへの応用，情報処理学会九州支部研究会報告，pp. 51-60 (1990).
9) 重松，與那覇，吉田：集合指向言語のデータベースへの応用，情報処理学会データベース・システム研究会資料，78-6，pp. 53-62 (1990).
10) 重松：SOL の言語仕様 (Ver. 2)，九州工業大学工学部電気工学科資料，p. 50 (1991).

付録 SOL の構文規則

以下の記法において，[，]，{，}，|，＝はメタ記号である．[ ]は，選択してもしなくてもよい．{ } は，1回選択する．[ ]* は，0回以上の繰り返し，[ ]† は，1回以上の繰り返しを，各々，意味する．メタ記号と紛らわしい記号は，“と” で囲んで区別してある．

```
プログラム－program プログラム名; ブロック.

ブロック＝
  [const [定数名 “＝” 定数;]†]
  [type  [型名  “＝” 型;]†]
  [var   [変数名 [, 変数名]* : {型|型名};]†]
  [map   [写像名 [,写像名]* : 変数名→変数名;]†]
  [[procedure 手続き名 [(引き数の並び)]; ブロック;]*|
   [function 関数名  [(引き数の並び)] : {基本型| 型名}; ブロック;]*
  ]*
  begin 文 [; 文]* end

引き数の並び = 引き数 [, 引き数]*
引き数 = [var] 変数 [,変数]* :{基本型|型名}

型 = 基本型|組型|集合型|添数付き集合型|ファイル型
基本型 = integer|real|char|string|boolean
組型 = tupleof “[” 欄の並び “]”
欄の並び =  欄名[,欄名]* :{基本型|組型|集合型}
          [; 欄名[,欄名]* :{基本型|組型|集合型}]*
集合型 = setof {基本型|組型|集合型}
添数付き集合型 = indexedset (定数～定数 [,定数～定数]*) of {基本型|組型|集合型}
ファイル型 = file

文 = {変数|関数名|写像名|対応名} ← 式|
     手続き名 [(式[,式]*)]|
     addmap 写像名(式) “＝” 式|
     begin 文 [; 文]* end |
     break |
     case 式 of [定数 [,定数]* : 文;]† esac |
     defmap {写像名|対応名} (式) “＝” 式|
     delmap 写像名(式) “＝” 式|
     for 変数←式 to 式 do 文 [;文]* od |
     forall 束縛式 do 文 [;文]* od |
     if 式 then 文 [else 文] fi |
```

repeat 文 [; 文]* until 式 |
while 式 do 文 [; 文]* od

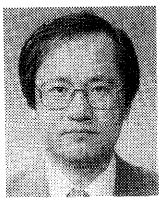
式 = 単純式 [{=|<|>|≠|≦|≧|∈|∉|⊂}単純式]
単純式 = [+|−] 項 [{+|−|∪|or} 項]*
項 = 因子 [{*|/|∩|and|div|mod} 因子]*
因子 = 符号のない定数|(式)| "[" 式[, 式]* "]"
|not 因子|変数|
関数名 (式[, 式]*)|外延集合式|範囲指定集合式|
内包集合式1|内包集合式2|
述語論理式|{写像名|対応名} [$^{-1}$] [(式)]

変数 = 変数名 [(式[, 式]*)|. 欄の名前]*
外延集合式 = "{" 式[, 式]* "}"
範囲指定集合式 = "{" 定数〜定数 "}"
内包集合式1 = "{" 束縛式 "|" 式 "}"
内包集合式2 = "{" 式 "|" 束縛式 [, 束縛式]* [, 式] "}"
述語論理式 = {∀|∃} (束縛式) (式)
束縛式 = 変数名 ∈ 式

対応名 = 写像名 "*"
定数 = [+|−] {符号のない定数|定数名}
符号のない定数 = 定数名|符号のない数|'文字'|"文字[文字]*"|true|false|ϕ
符号のない数 = 符号のない整数 [. 符号のない整数] [e[+|−]符号のない整数]
符号のない整数 = 数字$^{+}$

プログラム名, 定数名, 変数名, 型名, 写像名, 手続き名, 関数名, 欄名 =
英字[英字|数字|_]* [英字|数字]

(平成4年1月23日受付)
(平成4年12月10日採録)



重松 保弘 (正会員)

昭和22年生. 昭和45年九州工業大学工学部電子工学科卒業. 昭和47年九州工業大学工学部情報工学科助手. 昭和61年同講師. 現在, 同電気工学科助教授. 工学博士. プログラミング言語, 計算機ネットワークなどの研究に従事. 著書「基礎 BASIC プログラミング」, 「ネットワークアーキテクチャの基礎」など. 訳書「ネットワークと分散処理」. 電子情報通信学会会員.

吉田 将 (正会員)

昭和8年生. 昭和33年九州工業大学電気工学科卒業. 昭和35年九州大学大学院工学研究科修士課程修了. 工学博士. 昭和37年九州大学工学部講師, その後, 九州工業大学教授, 九州大学工学部教授を経て, 昭和61年10月九州工業大学情報工学部長. この間, 九州工業大学および九州大学情報処理教育センタ長, 九州大学大型計算機センタ長を歴任. 自然言語処理の研究に従事. 人工知能学会, 日本認知科学会, 米国 ACL などの会員.