戦略的なグラフ変換演算を用いた関数型プログラムのデバッグ法†

高橋直久†† 小野 諭†† 雨宮真人††

本論文では，関数型プログラムのバグ検出アルゴリズムである戦略的射影グラフ最小化 (SGM) 法を提案する．SGM 法は，実行履歴をプログラムの静的な構造に従って射影し，戦略的射影グラフ (SPG) と呼ぶバグの探索空間を生成する．そして，バグの性質や仮説から導かれる戦略に基づいて，プログラマへの質問の設定，および，SPG に対する変換演算と部分グラフの切り出し（選択）演算の適用を繰り返すことによりバグを検出する．このため，実行履歴のみから探索空間を生成し，それを単純に二分して探索する従来のバグ検出法に比べて，SGM 法では，再帰関数を含むプログラムによって大規模な実行履歴が生成された場合でも機械的かつ効率的にバグを検出することができる．本論文では，まず，SPG とその上の演算系を定義し，関数型プログラムのバグの伝播に関する性質に基づいてこれらの演算を SPG に適用することによりバグを検出する方法を示す．さらに，プログラムの構造とバグに関する仮説に基づいて SPG への変換演算の適用を加えることによって，より効率的なバグ検出が可能となることを具体例とともに示す．

1. まえがき

関数型言語は，数学的な関数の概念にその基礎を置き，記述の簡潔さ，読みやすさ，プログラム変換の容易さなど多くの優れた性質を備えている[1)~5)]．また，副作用がなく，参照に透明性のあるその計算構造（セマンティクス）は並列処理に適している．このため，関数型プログラミングの研究とともに，関数型プログラムの並列実行を指向した計算機アーキテクチャの研究[6)~11)]も盛んに行われている．さらに，上記のような計算構造はプログラムデバッグの際にも有効であり，筆者らは，この性質を用いて並列処理環境下でのプログラムデバッグを容易にする方式を検討している[12),13)]．

関数性を持つプログラム言語では実行時の誤りの伝播が簡明であるので，実行履歴の解析およびプログラマとの簡単な質問・応答により機械的にプログラムのバグを検出するプログラム診断システムを実現できる[12)~14)]．プログラムのバグは，プログラムテキスト中に存在し，実行結果の不正として現れる．したがって，プログラムを効率良くデバッグするためには，プログラムテキストおよびプログラムの実行履歴の双方を組み合わせて解析し，バグを探索していくことが望ましい．しかし，これまでに提案された機械的バグ検出法では実行履歴のみを解析し，プログラムを静的に解析して得られる関数依存関係を使用していなかった．このため，プログラム上では同じテキストに対応する実行結果について冗長な質問が繰り返される場合があり，再帰関数が数多く呼び出されている時には特に問題となっている．また，関数に与えられたパラメータが誤っている場合に，その誤ったデータを生成した計算の過程を追跡していくことができなかった．

本論文では，上記の問題を解決するため，プログラムの動的な実行履歴と静的な構造の双方を利用して探索空間を表す戦略的射影グラフを生成し，そのグラフに対する変換や部分グラフの切り出し（選択）などの演算を繰り返すことによりバグを検出する戦略的射影グラフ最小化 (Strategically Projected Graph Minimization : SGM) 法を提案する．SGM 法は，戦略的射影グラフで選択されたノードに対応する実行履歴を参照してプログラマに質問を発し，プログラマの応答およびバグの性質と仮説から導かれる戦略に基づき戦略的射影グラフに変換・選択演算を施す．そして，この一連の操作を繰り返すことにより，戦略的射影グラフをバグ発生源ノードのみからなるグラフに変形していく．このように，SGM 法は実行履歴と静的な構造の双方を利用して探索空間のグラフを生成し，プログラムの構造やバグの性質から導かれる探索戦略を採用しているため，実行履歴から導かれる探索空間を単純に二分して探索するだけであった従来の方法の上記の欠点を解決することができる．

† An Algorithmic Bug-Location Method for Functional Programs Using Strategic Graph Transformation System by NAOHISA TAKAHASHI, SATOSHI ONO and MAKOTO AMAMIYA (NTT Electrical Communications Laboratories).

†† NTT 電気通信研究所

2. 背　景

2.1 並列処理環境における関数型プログラムの診断システム

関数型プログラムではプログラムの記述順序と演算の実行順序が一致せず，しかも実行制御の流れが多数存在するので，プログラムにバグが入った場合には，トレースやメモリダンプなど従来のデバッグ法[15]でバグを検出するのは困難である．また，関数型プログラムを並列実行させるデータフローマシンでは，スタックやメモリの概念が排除されているので，従来の手法をそのまま使うことはできない．データフローマシンでは，演算結果がトークンと呼ばれるデータパケットの形でシステム内を流れる[6)~9)]ので，トークンを監視することによりプログラムの実行経過を把握することができる．しかし，プログラムの並列度が高い場合には，トークンの時系列が複雑となり，トークンを監視してバグを検出するのは難しい問題である．

この問題を解決するためには，データ依存関係の解析に基づいたプログラム診断システムが有効である[12]．このシステムは，図1に示すようにデバッガとデータベースからなり，プログラムの実行履歴，データフロー解析の結果などデバッグの際に必要なデータを有向グラフの形でデータベースに蓄積する．デバッガはプログラマに質問を出し，その応答に基づいてデータベースを更新する．そして再び質問を生成し，プログラマから応答を得る．このような質問と応答を繰り返すことによりデバッガはデータの検索範囲を狭めていき，最終的にバグの発生源を同定する．

2.2 デバッグに使用される依存グラフ

本論文では，プログラムの実行履歴や静的な依存関係を有向グラフを用いて表現する．以下では，これらのデータ構造と関連する用語について説明する．

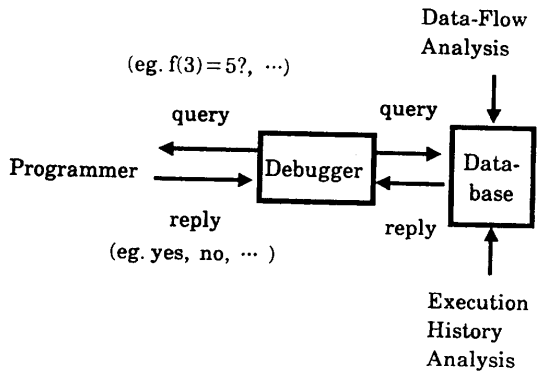


図1 プログラム診断システム
Fig. 1 Program diagnosis system.

（1） 有向グラフ

有向グラフ G は，(N, A, I) の三つ組であり，N はノード集合を，A は二つのノード間を接続するアークの集合を，また $I \in N$ は初期ノードをさす．グラフ $G=(N_g, A_g, I_g)$ において，$v \in N_g$ とした時，v の**先行成分**とは，G のアークにより v に到達可能なノード集合からなり，I_g を初期ノードとする G の極大部分グラフである．v の**後続成分**とは，G のアークにより v から到達可能なノード集合からなり，v を初期ノードとする G の極大部分グラフである．また，v の後続成分を $S_v=(N_v, A_v, I_v)$ とすると，v の**従属成分**とは，I_g から $w \in N_v$ へのすべてのパスが v を通るようなノード w をノード集合とし，v を初期ノードとする G の極大部分グラフである．

（2） インスタンス依存グラフ

関数呼び出しの際の関数名 f，パラメータの並び x，演算結果 y からなる三つ組 (f, x, y) を**具現値**と呼び，具現値と識別子の二つ組を**インスタンス**と呼ぶ．識別子は，同一の具現値を与える計算が複数回実行された場合にそれらを区別するために使用される．また，インスタンス i の具現値の計算にインスタンス j の具現値が使用された場合，j を i の子インスタンスと呼ぶ．**インスタンス依存グラフ**とは，実行中に生成される全インスタンスをノード集合とし，すべてのインスタンスについて自分の子インスタンスに向かってアークを張ることによりできる有向グラフである．初期ノードは，実行開始時の関数適用に対応するインスタンスである．

（3） 具現値依存グラフ

関数性を持つプログラムのデバッグでは，インスタンスの識別子は意味を持たない．具現値依存グラフは，インスタンス依存グラフにおいて同一の具現値を持つノードを一点に縮退して得られる準同型グラフである[12]．インスタンスの場合と同様に，ある具現値の計算に使用された具現値を子具現値と呼ぶ．また，関数 $\mathcal{F}$ は，具現値 $v=(f, x, y)$ に対し，その関数部分 f を与えるものとする．

（4） 関数依存グラフ

関数依存グラフのノードは，プログラムで定義されている各関数に対応する．また，初期ノードは最初に実行する関数に対応するノードである．アークには，呼び出しアークとパラメータ・アークの二種類がある．

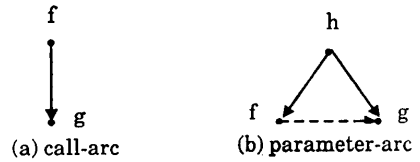


図 2　関数依存グラフの表現法（例）
Fig. 2　Function dependency graph representation.

呼び出しアークは関数相互の参照関係を表し，関数 f の中に関数 g の値を参照する式がある場合には，ノード f から g に至る呼び出しアークを張り，$f \rightarrow g$ と図示する．たとえば，例1のプログラムでは，f は g を呼び出し，g は基本関数のみを使用しているので，図2(a)の関数依存グラフとなる．

(例1)　$f(x)=g(x)+1;$
　　$g(x)=x-2;$

パラメータ・アークは関数の入力パラメータの定義・参照関係を表し，ある関数において，関数 g の入力パラメータを求める際に関数 f の値を使う場合には，ノード f から g に至るパラメータ・アークを張り，$f \dashrightarrow g$ と図示する．たとえば，例2の関数依存グラフは，g のパラメータの計算に f が使用されているので，図2(b)となる．

(例2)　$h(x)=g(f(x));$
　　$g(x)=x*x;$
　　$f(x)=x+1;$

関数依存グラフは上記の2種類のアークを使うので，先行成分，後続成分，従属成分もそれぞれのアーク対応に定義される．たとえば，ノード v の**呼び出し後続成分**とは，呼び出しアークのみを用いて v から到達可能なノード集合からなる部分グラフであり，v が直接または何重かの関数呼び出しの後に呼び出す可能性のあるすべての関数をノードとする．また，v の**パラメータ後続成分**とは，パラメータ・アークのみを用いて v から到達可能なノード集合からなる部分グラフであり，v を呼び出している関数本体のプログラムテキストにおいて v の結果を入力パラメータの計算に使用している関数，さらにその関数の結果を使用している関数，…と順次たどった場合に遭遇するすべての関数をノードとする．これに対し，単に v の後続成分という場合は，呼び出しアークとパラメータ・アークの双方を用いて到達可能なノード集合からなる部分グラフをさすことにする．先行成分や従属成分についても，同様に定義する．

2.3　関数型プログラムにおけるバグ

関数 f がプログラムにおいて関数定義されているとする．今，任意の具現値 v を (f, x, y) とする．x が関数 f の入力として予期しない誤ったパラメータを含んでいる場合には，v は**未定義**であるという．x が正しいパラメータである場合に，$f(x)=y$ ならば v は**真**であるといい，$f(x)\neq y$ ならば v は**偽**であるという．

関数型言語は履歴依存性を持たないので，プログラムに含まれるバグは，正しいパラメータに対し誤った答えを返す，という形で表面化する．したがって，次の二つの条件を同時に満たす具現値 b を (f, x, y) とすると，$f(x)$ を計算した際に実行した関数 f の式の中に誤りがあると断定できる．

・b は偽である．
・b の子具現値はすべて真である．

上記 b を**バグ発生具現値**と呼び，デバッガが実行履歴の中からバグ発生具現値を求める手続きを**バグ検出アルゴリズム**と呼ぶ．

3. 戦略的射影グラフとその変換・選択

本章では，戦略的射影グラフ，および，その上の演算について述べる．

3.1　戦略的射影グラフ

あるプログラム P の関数依存グラフ $G=(N_g, A_g, I_g)$ と具現値依存グラフ $V=(N_v, A_v, I_v)$ を考える．今，$v_1 \rightarrow v_2$ $(v_1, v_2 \in N_v)$ なるアーク $a \in A_v$ がグラフ V に存在したとする．そして，各具現値の関数部分，すなわち $\mathcal{F}(v_1), \mathcal{F}(v_2)$ をそれぞれ f_1, f_2 とする．この時，$f_1 \rightarrow f_2 (f_1, f_2 \in N_g)$ なる呼び出しアーク $c \in A_g$ がグラフ G に存在する．また，$\mathcal{F}(I_v)$ は I_g となる．したがって，グラフ V の各ノードのうち，同一の関数部分を持つノードを一点に縮退してできる準同型グラフは，グラフ G の部分グラフになっている．そこで，N_g から N_v のべき集合への関数 $\mathcal{D}$ を次のように定義する．

$$\mathcal{D}(f) \triangleq \{v \mid v \in N_v \wedge \mathcal{F}(v)=f\}$$

$\mathcal{D}$ は，関数名 f に対し，f を使用して生成された具現値の集合を与える関数であり，$\mathcal{D}(f)$ をノード f の全体具現値集合と呼ぶ．

プログラム P の関数依存グラフ G と具現値依存グラフ V から生成される戦略的射影グラフ $S=(N_s, A_s, I_s)$ とは，次の初期グラフ S_0 に対し，3.2〜3.3節に示す変換・選択演算を任意回施して得られるグラフの

ことである．

初期グラフ $S_0=(N_{s_0}, A_{s_0}, I_{s_0})$ は，グラフ G の部分グラフであり，以下のように定められる．

ノード集合 N_{s_0} は，関数依存グラフのノード集合のうち，インスタンス依存グラフで実際に関数適用された関数名の集合とする．すなわち，

$$N_{s_0}=\{f \mid f\in N_g \wedge \mathcal{D}(f)\neq\phi\}.$$

アーク集合 A_{s_0} は，関数依存グラフのアーク集合（呼び出しアークおよびパラメータ・アーク）のうち，アークの始点と終点が共に集合 N_{s_0} に属するものとする．また，初期ノード $I_{s_0}=I_g$ とする． □

次節で導入される構造を持ったノードに対し，これまでのノードを基本ノードと呼ぶ．戦略的射影グラフの基本ノード f に対し，関数 $\mathcal{D}$ は，関数依存グラフの場合と同様に定義される．また，S のすべてのノードの全体具現値集合の和集合をグラフ S の全体具現値集合と呼び，$\mathcal{D}(S)$ と表す．明らかに，$\mathcal{D}(S_0)=N_v$ である．

SGM 法では，バグ探索において，まず戦略的射影グラフのあるノード f を選択し，次に $\mathcal{D}(f)$ の中から任意のノードを選択して，その真偽をプログラマに質問する．このとき，$\mathcal{D}(f)$ 中の一部のノードのみを選択対象にすることができるようにするため，選択対象となるノード集合を与える関数 $\mathcal{S}$ を定める．ノード集合 $\mathcal{D}(f)$ のうち，どの部分を $\mathcal{S}(f)$ に含めるかは，戦略に従って決定される．本論文においては，基本ノードについて，$\mathcal{S}(f)$ は，$\mathcal{D}(f)$ のノードのうち，初期ノード I_v から，$\mathcal{D}(f)$ 中の他のノードを経由しないで到達可能なノードの集合とする．すなわち，

$$\mathcal{S}(f)\triangleq\{v \mid v\in N_v \wedge \mathcal{A}(v)\cap\mathcal{D}(f)=\{v\}\}$$

と定める．ここで，$\mathcal{A}(v)$ は，ノード v の先行成分のノード集合をさす．

$\mathcal{S}(f)$ をノード f の探索具現値集合と呼び，グラフ S のすべてのノードの探索具現値集合の和集合をグラフ S の探索具現値集合と呼ぶ．

3.2 戦略的射影グラフに対する変換演算

プログラム P の関数依存グラフ G と具現値依存グラフ V から生成される戦略的射影グラフ $S=(N_s, A_s, I_s)$ を考える．戦略的射影グラフのノード集合として，基本ノードのほか，以下の演算で生成される構造を持ったノードも認めることにする．

（1） 具現化演算

探索具現値集合および全体具現値集合から具現値 j を取り出し，グラフ S において関数 $\mathcal{F}(j)$ のノードに j を付加する演算を具現化 (instantiation) 演算と呼ぶ．今，$g\in N_s$ とし，また $j\in\mathcal{S}(g)$ とする．この時，グラフ S に対し，ノード g に具現値 j を付加して得られるグラフは，ノード g を $g\{j\}$ と変更する以外は，グラフ S と同一である．$g\{j\}$ を具現値付加ノードと呼び，

$$\mathcal{D}(g\{j\})\triangleq\mathcal{D}(g)-\{j\}$$
$$\mathcal{S}(g\{j\})\triangleq\mathcal{S}(g)-\{j\}$$

と定義する．また，$i\in\mathcal{S}(g)$ として，ノード $g\{i\}$ に具現値 j を付加する場合は，$g\{i, j\}$ と表現し，さらに具現値を付加する場合も同様とする．

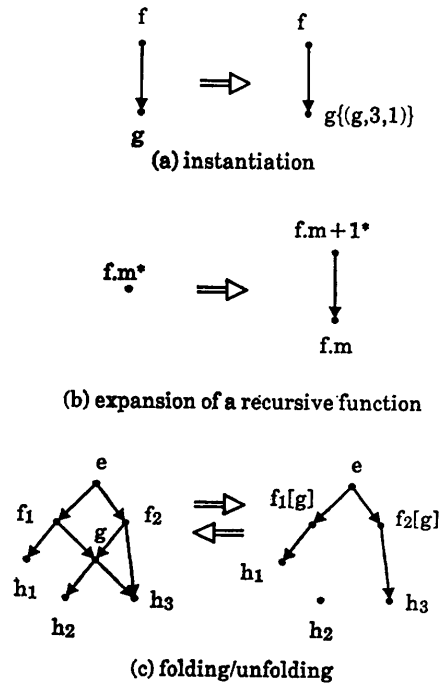


図 3 戦略的射影グラフの変換演算
Fig. 3 Transformation operators for strategically projected graph.

たとえば，戦略的射影グラフとして図 3（a）左側の関数依存グラフを考え，$(g, 3, 1)\in\mathcal{D}(g)$ とする．この時，ノード g に具現値 $(g, 3, 1)$ を付加する演算を施すと同図右側のようにグラフが変換される．

$\mathcal{S}(f)\neq\phi$ の時，ノード f は具現化可能という．

（2） 再帰表現伸張演算

再帰関数の具現値を分類して後述の戦略に従って具現値の検査順序を制御できるようにするため，再帰関数を表すノードと具現値を分離する演算を導入する．プログラム P において，関数 f が再帰的に定義されているとする．この時，具現値依存グラフ V の初期ノード I_v からある葉までのパスの中に関数部分が f

である具現値が複数現れうる．戦略的射影グラフの基本ノード f と自然数 m について，再帰表現ノード $f.m$ および $f.m^*$ を以下のように再帰的に定義する．

$\mathcal{D}(f.1^*) \triangleq \mathcal{D}(f)$

$\mathcal{S}(f.1^*) \triangleq \mathcal{S}(f)$

$m \geqq 1$ に対し

$\mathcal{D}(f.m) \triangleq \{j \mid j \in \mathcal{D}(f.m^*) \wedge \mathcal{M}(j, f) = m\}$

$\mathcal{D}(f.m+1^*) \triangleq \{j \mid j \in \mathcal{D}(f.m^*) \wedge \mathcal{M}(j, f) \geqq m+1\}$

$\mathcal{S}(f.m) \triangleq \mathcal{D}(f.m)$

$\mathcal{S}(f.m+1^*) \triangleq \{j \mid j \in \mathcal{S}(f.m^*) \wedge \mathcal{M}(j, f) \geqq m+1\}$

ここで，$\mathcal{M}(j, f)$ は，具現値依存グラフ V において，ノード j から任意の葉までのパスの集合を P_j，また，パス p $(p \in P_j)$ に存在する具現値のうち関数部分が f のものの個数を $\mathcal{E}(p, f)$ としたとき，

$$\mathcal{M}(j, f) \triangleq \max_{p \in P_j} (\mathcal{E}(p, f))$$

とする．明らかに，

$\mathcal{D}(f.m^*) = \mathcal{D}(f.m) \cup \mathcal{D}(f.m+1^*)$　$(m \geqq 1)$

が成立する．

再帰表現ノード $f.m^*$ $(m \geqq 1)$ がグラフ S に存在すると仮定する．（基本ノード f は，関数 f がプログラムPにおいて再帰的に定義されている場合は，再帰表現ノード $f.1^*$ とみなす．）この時，ノード $f.m^*$ を伸張 (expansion) して得られるグラフとは，ノード $f.m^*$ を $f.m$ と $f.m+1^*$ とに分割し，かつ，ノード $f.m^*$ に出入りしているアークは，すべて $f.m$ と $f.m+1^*$ とに出入りさせて得られるグラフのことである．たとえば，ノード $f.m^*$ を伸張すると図3（b）のようになる．

$\mathcal{D}(f.m+1^*) \neq \phi$ の時，ノード $f.m^*$ は伸張可能であるといい，グラフ S が伸張可能なノードを含む時，S は伸張可能であるという．

（3） 畳み込み・展開演算

SGM 法では，戦略に従って仮説を立てて探索を進める．畳み込み (folding) 演算は，仮説に基づき，あるノード集合を探索範囲から除外することに対応し，また，展開 (unfolding) 演算は，その仮説を放棄してノード集合を元に戻すことに対応する．

戦略的射影グラフ S において，ノード g $(g \in N_s, g \neq I_s)$ を畳み込んでできるグラフとは，$f \to g$ なる呼び出しアーク $a \in A_s$ を持つそれぞれのノード $f \in N_s$ を $f[g]$ と変換し，かつ，g を始点とする呼び出しアークとパラメータ・アークをすべて除去してできるグラフのことである．そして，$f[g]$ の形式のノードを，畳み込みノードと呼び，

$\mathcal{D}(f[g]) = \mathcal{D}(f) \cup \mathcal{D}(g)$

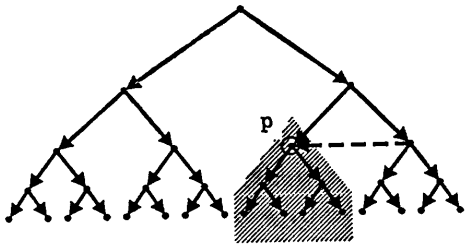
$\mathcal{S}(f[g]) = \mathcal{S}(f)$

と定義する．

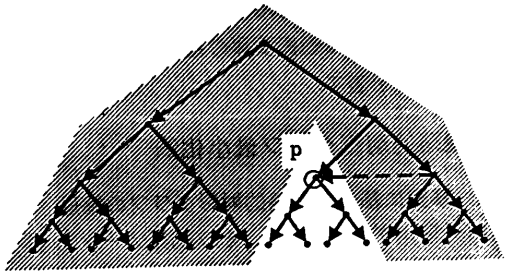
畳み込みノード $f[g]$ に畳み込みの逆演算を施して，f と g の独立したノードに分解する演算を展開と呼ぶ．図3（c）に畳み込み・展開演算の例を示す．

3.3 戦略的射影グラフに対する選択演算

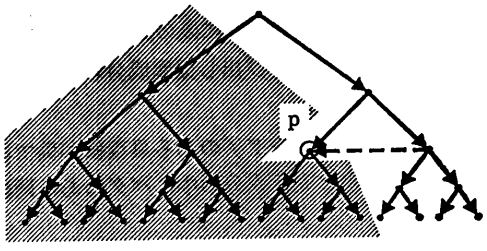
戦略的射影グラフ $S=(N_s, A_s, I_s)$ を考える．戦略的射影グラフのノード集合の一部を除去して部分グラフを求める選択演算として，以下の3種類の演算を定義する．なお，各ノードの全体具現値集合および探索具現値集合はいずれの演算の場合も変化しない．



(a) inner-removal



(b) outer-removal



(c) non-input-removal

図 4　戦略的射影グラフの選択演算
Fig. 4　Selection operators for strategically projected graph.

(1) 内側除去演算

$p \in N_s$ かつ $p \neq I_s$ とする．グラフ S に対しノード p の内側除去 (inner-removal) を行って得られるグラフとは，I_s を初期ノードとし，グラフ S からノード p の呼び出し従属成分を取り除いて得られる極大部分グラフのことである．

(2) 外側除去演算

$p \in N_s$ とする．$v \in \mathcal{S}(p)$ なる具現値 v について，グラフ S に対しノード p の外側除去 (outer-removal) を行って得られるグラフとは，p を初期ノードとする p の呼び出し後続成分のことである．この演算の結果，v を初期ノードとする v の後続成分が新たな具現値依存グラフ V となる．

(3) 非入力部除去演算

p の呼び出し先行成分のノード集合を L，L 中のすべてのノードのパラメータ先行成分についてのノード集合の和集合を M とする．グラフ S において，ノード集合 M のすべてのノードの呼び出し後続成分のノード集合と L の和集合を p の入力決定成分という．この時，グラフ S に対しノード p の非入力部除去 (non-input-removal) を行って得られるグラフとは，I_s を初期ノードとし，p の入力決定成分に含まれないノードを S から取り除いた極大部分グラフのことである．

図4に各選択演算の適用例を示す．図4(a)～(c)において，斜線部は選択演算の適用により取り除かれるノード集合を表す．

4. 戦略的射影グラフ最小化法

4.1 事実に基づく戦略的射影グラフの縮退

戦略的射影グラフ S は，その初期ノードの具現値が偽である時，バグに関して次のような性質を持っている．

[性質 P1] S のノード f が具現化不能の場合，すなわち $\mathcal{S}(f)=\phi$ の場合，f に偽の具現値が付加されていないならば，f の呼び出し従属成分以外にバグのある関数が存在する．

[性質 P2] S のノード f のある具現値が偽である場合，f の呼び出し先行成分の中にバグのある関数が存在する．

[性質 P3] S のノード f のある具現値が未定義である場合，f の入力決定成分の中にバグのある関数が存在する．

今，S の全体具現値集合 $\mathcal{D}(S)$ とバグ発生具現値 b ($b \in \mathcal{D}(S)$) を考える．f の具現値が真または未定義の場合に具現化演算を適用するものとする．この時，S の任意のノード f について，具現化不能の場合に内側除去，具現値が偽の場合に外側除去，具現値が未定義の場合に非入力部除去の各演算を適用すると，上記性質 P1～P3 により $b \in \mathcal{D}(S)$ を保証したまま $\mathcal{D}(S)$ を小さくすることができる．したがって，S の任意のノードの具現値を調べ，その結果により具現化演算，および内側，外側，非入力部除去の選択演算を適用する操作を $\mathcal{D}(S)$ の要素がただ一つになるまで繰り返し行って S を縮退させていくと，最終的に $\mathcal{D}(S)=\{b\}$ となり，バグ発生具現値を求めることができる．

4.2 仮説に基づく戦略的射影グラフの変換

前節に示した方法で，バグ発生具現値を求めるために必要な操作の繰り返し回数は，戦略的射影グラフ S から取り出すべきノードの選定法と調べるべき具現値の選定法に依存する．本節では，この回数を少なくすることを目指して，S の変換戦略を定める．まず，プログラムの構造とバグに関する次のような仮説を定める．

[仮説 H1] すべての関数において，ある具現値が真であると，その関数にはバグがない可能性が高い．

[仮説 H2] 今，p, q をある関数 f の具現値とし，かつ p は真であるとする．この時，p, q それぞれの計算の際に f 内を通過するパスが等しい場合には，q も真である可能性が高い．

[仮説 H3] プログラム中の関数の数が増えるとその中にバグのある関数が含まれる可能性が高い．

上の各仮説から以下に示す戦略を採用する．

[戦略 S1] ある関数のある具現値が真であるならば，仮説H1に従い，その関数を畳み込んで，その関数を関数部分に持つ具現値を探索具現値集合から除去する．バグ検出中にすべての関数が畳み込まれて，「すべての関数にバグがない」という仮説 H1 に起因する矛盾に到達した場合には，展開演算を適用し仮説を覆す．

[戦略 S2] 再帰関数では，再帰する場合と停止する場合で実行するパスが異なる．このため，再帰関数 f について，S のノード f の全体具現値集合にバグ発生具現値が含まれていることがわかった場合にノード f を伸張する．これにより再帰が停止するパスを通過した具現値が探索具現値集合に現れる．同様に，$f.n$ ($n \geq 1$) を $n=1$ から順次伸張していくと，再帰が停止

する直前の再帰，さらにその直前の再帰，…に対応する具現値が探索具現値集合に次々に現れる．この結果，f 内を通過するパスが異なる可能性の高い具現値を順次探索具現値集合に入れることができる．

[戦略 S3] 今，S のあるノード f の具現値が真，偽，未定義のそれぞれの場合について，バグのある関数が存在する領域として戦略 S1 および性質 P2, P3 で指定される部分グラフを考え，そのノード数をそれぞれ T_f, F_f, U_f とする．この時，$(T_f+F_f+U_f)$ が最小となる f の中から，T_f, F_f, U_f の最大値が最小となるような点 f を選定する．今，f の具現値が真・偽・未定義である確率がすべて等しいとする．この時，この選定法に従うと f の具現値の真・偽・未定義が判明し，その結果により演算を適用してできる S のノード数の期待値が最小となる．

4.3 戦略的射影グラフ最小化法

戦略的射影グラフ最小化 (SGM) 法は，戦略的射影グラフ S に対する事実に基づく縮退と仮説に基づく変換とを統合した方法である．すなわち，**図5**に示すように，4.1 節，4.2 節で述べた適用法に従って各演算を繰り返し施し，$\mathcal{D}(S)$ に最後に残った要素がバグ発生具現値であると断定する．図において，A1 から A7 は適用演算の識別子であり，適用演算に付した ( )内の性質および戦略名はその演算を適用する理由である．また，$|S|$ は S のノード数を表す．$|S|\geqq 2$ の時，まず，戦略 S3 により S からノード f を選定する．ノード f が具現化できない場合，すなわち $\mathcal{S}(f)=\phi$ の場合には性質 P1 に従い，ノード f の内側除去を施す．他方，f が具現化可能な場合には，$\mathcal{S}(f)$ の任意の要素 v について質問する．この質問の答えにより，v の真，偽，未定義が決定すると，それぞれ，畳み込み，外側除去，非入力部除去の演算を施す．この時，v が真または未定義の場合には f に具現化演算を施す．図のように，$|S|=1$ の時には，S が伸張可能ならば伸張，展開可能ならば展開を施す．いずれも可能でない場合には，$\mathcal{D}(S)$ がバグ発生具現値ただ一つからなる集合になる．

与えられたプログラムのトップレベルの（最初に呼び出した）関数の具現値が偽である場合に，上記操作によりバグのある関数とバグを発生した入力パラメータの値を知ることができる．この結果を用いてバグ発生時の変数の束縛状態を解析し[12]，プログラマは少なくとも一つのバグを検出して修正することができる．複数のバグが存在する場合には，バグを一つ修正した

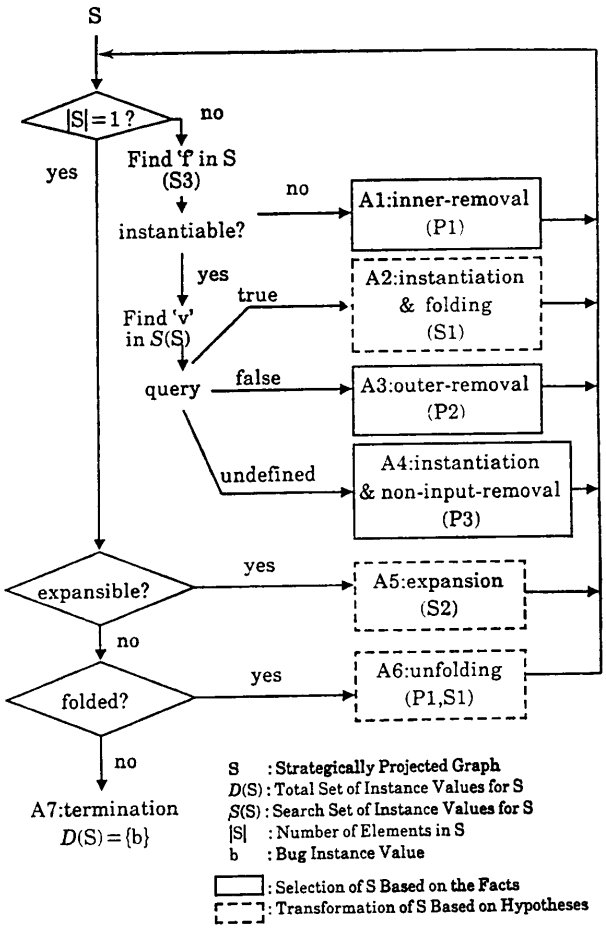


図 5 戦略的射影グラフ最小化法
Fig. 5 SGM algorithm.

プログラムを再実行させ，別のバグを検出・修正するという一連の操作を繰り返して，一つずつバグを修正することにより正しいプログラムを得ることができる．SGM 法の特徴は，戦略に基づいて具現値依存グラフの射影法を変えることにより探索空間を制御している点にある．この結果，実行履歴と静的なプログラムの構造の双方の情報が探索空間に埋め込まれ，かつ，プログラムの構造やバグの性質から導かれる探索戦略が用いられる．このため，実行履歴のみから探索空間を生成し，性質 P3 が反映されない従来の方法[12),14)]に比べて次のような利点が得られる．

（1） プログラム上で同じテキストに対応する実行結果について冗長な質問が繰り返されない．

（2） 関数に与えられたパラメータが誤っている場合に，その誤ったデータを生成した計算過程を追跡して探索していくことができる．

（3） 仮説や戦略に基づく，より高度な探索方策を

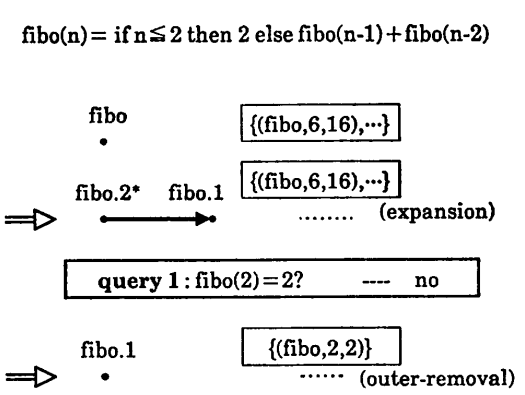


図 6 戦略的射影グラフを用いたバグ検出
Fig. 6 Bug location using strategically projected graph.

組み込むことができる.

以下に，簡単な例により SGM 法によるバグ検出手順の概要を示す．まず，次のようなバグのあるフィボナッチ数計算プログラム fibo を考える.

(例 3) fibo(n)=if $n\leqq 2$ then 2
else fibo($n-1$)+fibo($n-2$)

今，fibo(6) の計算の結果として 16 を得たとする．この時，**図 6** のように戦略的射影グラフ S を繰り返し変換・選択するとバグを検出できる．図で a T1 ⇨ b T2 …(r) は，グラフ a に演算 r を適用し，この結果 a が b に変換・選択され，全体具現値集合が T1 から T2 に変更されることを意味する.

戦略的射影グラフの初期グラフは，ノード fibo のみからなるグラフである．そこで，まず，fibo を伸張する．これにより生成された点 fibo.1 の具現値 (fibo, 2, 2) を検査する．すなわち，"fibo(2)=2" の真偽をプログラマに問い合わせ，偽であるという応答を得る．この結果，具現値 (fibo, 2, 2) について，S に対して fibo. 1 の外側除去を行う．これにより全体具現値集合の要素がただ一つとなるので，その具現値 (fibo, 2, 2) の計算でバグが発生したと断定する．関数 fibo にパラメータ 2 を与えた場合に実際に実行される fibo の式を求めると再帰関数 fibo の停止部分 (if の条件部と then 部) にバグがあることが判明する.

SGM 法を従来のバグ検出アルゴリズムと比較するため，上述の変換，および従来の方法での探索空間の分割を図示すると**図 7** となる．Divide-and-Query 法[14]，PGM 法[12] はそれぞれインスタンス依存グラフ，具現値依存グラフを質問に基づいてノード数ができるだけ等しい二つの部分グラフに分割する方法である．図において，各ノードに付与した値は具現値の入力パラメータを表し，①…は第 i 回目の質問応答の結果に基づいてグラフを 2 分する箇所を示す．この図から，プログラム fibo の場合，SGM 法ではプログラムの構造をより直接的に反映させた方法で具現値を選択しているので，従来のアルゴリズムより少ない分割回数（質問回数）でバグを検出できることがわかる.

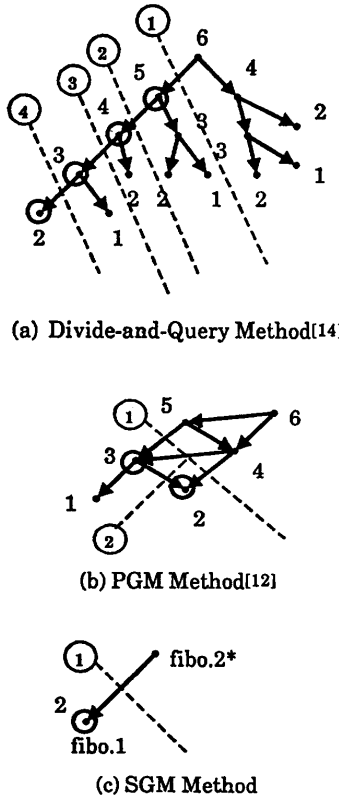


図 7 バグ検出法の比較
Fig. 7 Comparison of bug location algorithms.

5. 戦略的射影グラフ最小化法の適用例

(1) フィボナッチ数の計算プログラム

次のようなフィボナッチ数計算プログラムで再帰部分にバグがある場合のバグ検出過程を示す.

(例 4)

$fr(n)$=**if** $n\leqq 2$ **then** 1
else $fr(n-1)\times fr(n-2)$

この場合には，前の例に比べて変換ステップ数が多くなる．しかし，図 8 に示すように繰り返し変換・選択演算を施すことにより，$fr(3)=1$ の計算がバグの発生源であることがわかり，関数 fr の再帰部分 (if の条件部および else 部) の中にバグがあることが判明する．従来のように探索空間を単純に二分する方法では，上のような再帰関数では再帰部分の具現値が繰り

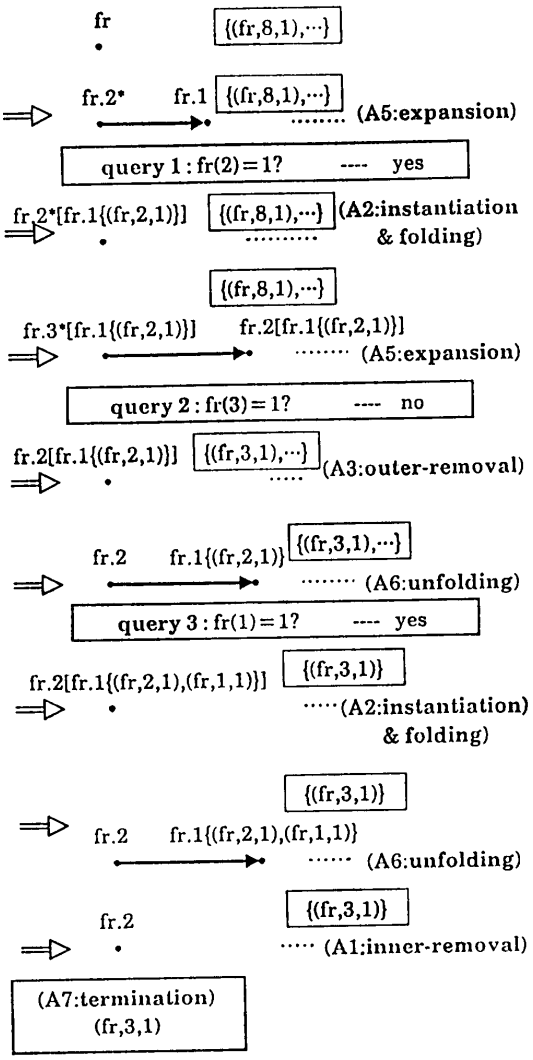


図 8 SGM 法によるフィボナッチ数計算プログラムのバグ検出
Fig. 8 Bug location of erroneous Fibonacci program using SGM algorithm.

返し調べられ，n が大の場合に冗長な質問が繰り返されるという問題が生じる．これに対して SGM 法では n の値と独立に図 8 の変換・選択が行われ，関数の停止部，再帰部の順に具現値が調べられるのでこのような問題が解決されている．

（2） マージソートプログラム

リストデータをマージソートアルゴリズムによりソートするプログラムを考える．このプログラムは，リストを分割する関数 split，ソートされた二つのリストをマージする関数 merge，これら二つの関数および自分自身を再帰的に使ってリストをソートする関数 mergesort の三つの関数からなる．今，（例 5）のように関数 mergesort の停止部分にバグがある場合には，図 9 のように戦略的射影グラフの変換が行われて，バグの位置が検出される．

（例 5）

```
split(x)=if null(x) then (nil, nil)
         elsif null (cdr(x)) then (x, nil)
           else {(u, v)=split (cddr(x));
                 (cons (car(x), u),
                  cons (cadr (x), v))} ;
merge(x, y)=if null(x) then y
            elsif null(y) then x
              elsif car(x)<car(y)
                then cons(car(x),
                  merge(cdr(x), y));
mergesort(x)=if null(cdr(x)) then nil
             else {(u,v)=split(x);
                   merge(mergesort(u),
                     mergesort(v))}
```

6. むすび

本論文では，関数型プログラムを並列実行させるシステムに適し，静的な依存関係と実行履歴の解析を結合させた新たなバグ検出アルゴリズムである戦略的射影グラフ最小化 (SGM) 法を提案した．

SGM 法は，実行履歴と静的な構造の双方を利用して探索空間のグラフを生成し，プログラムの構造やバグの性質から導かれる探索戦略を採用している．このため，実行履歴のみから探索空間を生成し，それを単純に二分して探索する従来の方法に比べて，

（1） プログラム上で同じテキストに対応する実行結果について冗長な質問が繰り返されない．

（2） 関数に与えられたパラメータが誤っている場合に，その誤ったデータを生成した計算過程を追跡して探索していくことができる．

（3） 仮説や戦略に基づく，より高度な探索方策を組み込むことができる．

などの利点を持っている．

以上の理由から，本方式を用いると，再帰関数を含むプログラムにより大規模な実行履歴が生成された場合でも，機械的かつ効率的にバグを検出することが可能になる．本稿では，戦略に基づきグラフを変換する演算系を用いて探索空間を絞り込む手法に議論を集中させて，すべての具現値に対して，その真偽の判定が等しく容易にできるものとした．しかし，実際的な局

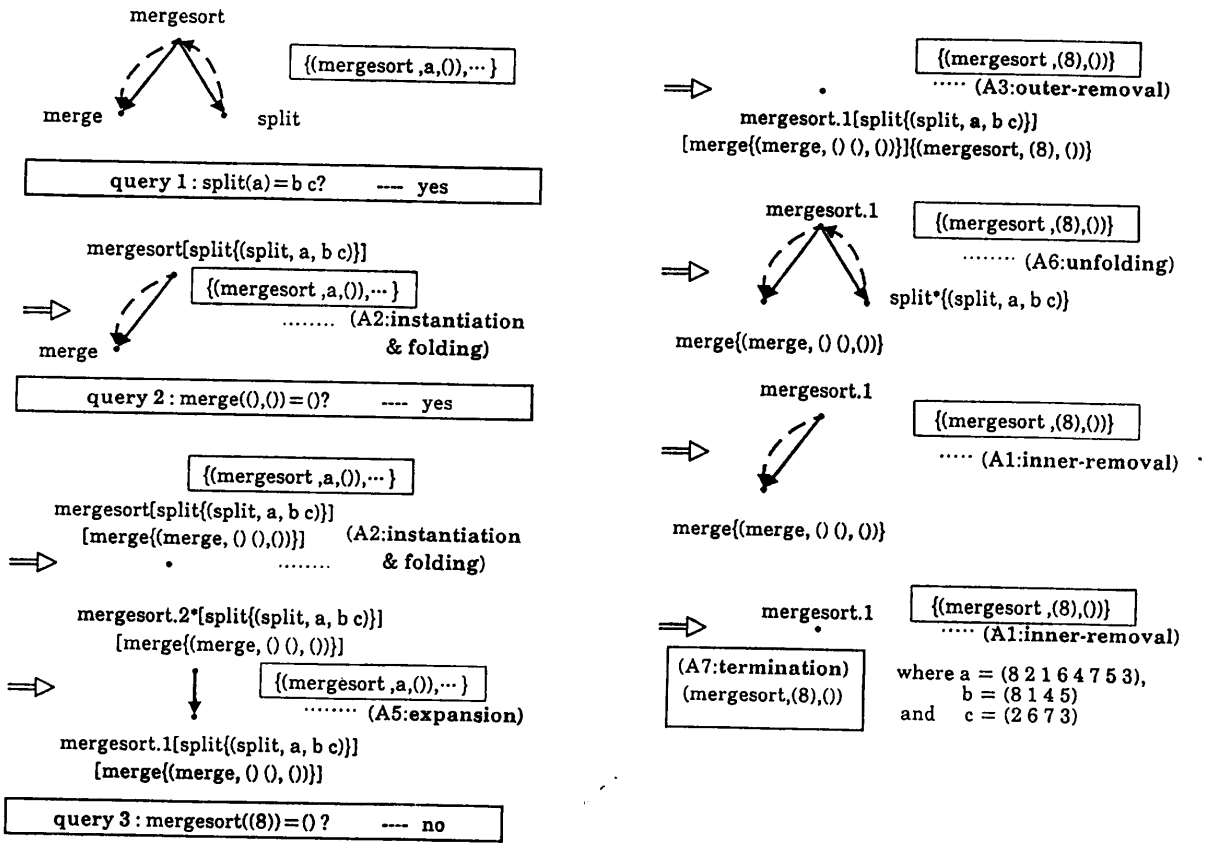


図 9 SGM 法によるマージソートプログラムのバグ検出
Fig. 9 Bug location of erroneous mergesort program using SGM algorithm.

面を考えると，任意に選択された具現値の真偽の判定が容易でない場合もある．このため，筆者らは，プログラムの階層的な構造に着目した具現値の選択法[16]や簡単な仕様により具現値の真偽を間接的に指定する方法などを検討している．これらについては，別途報告する予定である．

本稿では，再帰関数のバグ検出に提案手法が有効であることを直観的に理解できるように簡単な例を示したが，その有効性を明示するにはさらに議論が必要である．このため，現在，本方式を組み込んだデータフロープログラムデバッグシステム[13]を用いて方式評価を行っている．本手法では，実行履歴を使わずにプログラムテキストの解析から検査すべき関数を決定できるので，関数を決定してから再計算を行い具現値を取り出すことにより，実行履歴を保持するために必要なメモリ量を大幅に減少させることが期待できる．今後，実際の応用プログラムをデバッグする場合に必要な質問・応答回数，CPU 時間，メモリ量などの観点から各バグ検出アルゴリズムの性能評価を進める予定である．

謝辞 本稿に関して貴重なご意見を頂いた後藤厚宏氏，並びに，日頃ご討論頂く情報通信基礎研究部第二研究室諸氏に深く感謝します．

参 考 文 献

1) Darlington, J., Henderson, P. and Turner, D. A. (eds.): *Functional Programming and Its Application*, Cambridge University Press, Cambridge (1982).
2) Turner, D. A.: A New Implementation Techniques for Applicative Languages, *Softw. Pract. Exper.*, Vol. 9, pp. 31-49 (1979).
3) Keller, R. M.: FEL (Function-Equation Languages) Programmer's Guide, AMPS Technical Memorandum No. 7, University of Utah, Salt Lake City (1982).
4) Amamiya, M., Hasegawa, R. and Ono, S.: Valid, A High-Level Functional Programming Language for Data Flow Machine, *Rev. ECL*, Vol. 32, No. 5, pp. 793-802 (1984).

5) Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Comm. ACM*, Vol. 21, No. 8, pp. 613-641 (1978).
6) Arvind and Kathail, V.: A Multiple Processor Dataflow Machine That Supports Generalized Procedures, *Proc. 8th Annual Symposium on Computer Architecture*, pp. 291-302 (1981).
7) Gurd, J. and Watson, I.: Data Driven System for High Speed Parallel Computing (1 & 2), *Comput. Des.*, Vol. 9, No. 6 & 7, pp. 91-100 & 97-106 (1980).
8) Takahashi, N. and Amamiya, M.: A Data Flow Processor Array System: Design and Analysis, *Proc. 10th Annual Symposium on Computer Architecture*, pp. 243-250 (1983).
9) Amamiya, M., Hasegawa, R., Nakamura, O. and Mikami, H.: A List-Processing-Oriented Data Flow Machine Architecture, *Proc. 1982 National Computer Conference, AFIPS*, pp. 143-151 (1982).
10) Keller, R. M., Lindstrom, G. and Patil, S.: A Loosely Coupled Applicative Multiprocessing System, *Proc. 1979 National Computer Conference, AFIPS*, Vol. 49, pp. 613-622 (1979).
11) Darlington, J. and Reeve, M.: ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Proc. 1981 ACM/MIT Conference on Functional Programming Language and Computer Architecture*, pp. 65-75 (1981).
12) 高橋直久，小野 諭，雨宮真人：並列処理環境における関数型プログラムのデバッグ方式，情報処理学会論文誌，Vol. 27, No. 4, pp. 425-434 (1986).
13) 高橋直久，小野 諭，雨宮真人：関数依存グラフの変換による関数型プログラムのデバッグ法，情処ソフトウェア基礎論研究会，12-3 (1985).
14) Shapiro, E. Y.: *Algorithmic Program Debugging*, MIT Press, Cambridge (1983).
15) Johnson, M. S.: A Software Debugging Glossary, *SIGPLAN Notices*, Vol. 17, No. 2, pp. 53-70 (1982).
16) Takahashi, N. and Ono, S.: Strategic Bug Location Method for Functional Programs, 京都大学数理解析研究所講究録，586, pp. 196-223 (1986).

(昭和61年1月13日受付)
(昭和61年6月18日採録)

高橋 直久（正会員）

昭和26年生．昭和49年電気通信大学応用電子工学科卒業．昭和51年同大学院修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，並列計算機のアーキテクチャと言語の研究に従事．現在，日本電信電話(株) NTT 基礎研究所主任研究員．電子通信学会，ACM 各会員．

小野 諭（正会員）

昭和29年生．昭和52年東京大学工学部電子工学科卒業．昭和57年同大学院工学系博士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，並列計算機および関数型言語の研究に従事．現在，日本電信電話(株) NTT 基礎研究所主任研究員．工学博士．電子通信学会，IEEE 各会員．

雨宮 真人（正会員）

昭和17年生．昭和42年九州大学工学部電子工学科卒業．昭和44年同大学院工学研究科修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，プログラミング言語・処理系，自然言語理解，データフロー計算機，並列処理，関数型／論理型言語，知能処理アーキテクチャの研究に従事．現在日本電信電話(株)NTT 基礎研究所情報通信基礎研究部第一研究室室長．工学博士．電子通信学会，ソフトウェア科学会，IEEE 各会員．