ソフトウェア基礎論　12－3
（1985. 2. 22）

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

高橋　直久　小野　諭　雨宮　真人
（日本電信電話公社武蔵野電気通信研究所）

"A Bug Location Method for Functional Programs using Function Dependency Graph Transformation"

Naohisa TAKAHASHI, Satoshi ONO and Makoto AMAMIYA
(Musashino Electrical Communication Laboratory, N.T.T.)

A new algorithmic bug detection method for functional programming languages named Function dependency Graph Minimization (FGM) method is proposed. This method detects locations of bugs by analyzing the programmer's answers to the queries which are automatically generated by the debugging system. The method analyzes a function dependency graph which reflects the static dependency of the functions, as well as the execution history that is dynamically created by test-run of the program to be debugged. Since this method adopts a new strategy deduced from the program structure and bug property, it is especially suitable for debugging recursive programs. This paper also clarifies the effectiveness of the proposed method by showing the results of the experimental debugging system for dataflow machines.

Keywords: algorithmic debugging, graph reduction, dataflow analysis, dataflow machine

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。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〔1〕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〔2, 3〕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，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の問題を軽減する試みも行われている〔4〕。

関数性をもつプログラム言語では、誤りを含む実行履歴をもとにプログラマへの質問を繰り返し、それらの応答を解析すること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〔4, 5〕。これらのシステムは，質問すべきデータを探索空間の中から選択する手続きと、応答に基づいて探索空間を簡約化する手続きを定めており、実行履歴が大規模な場合でも機械的かつ効率的にバグを検出することを可能にしている。しかし、従来のシステム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を繰り返す場合があり、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た。また、関数に与えられたパラメータが誤っている場合でも、その誤ったデータを生成した計算のみを取り出して原因を探索することができなかった。

本稿では，上のような問題を解決するため，プログラムの静的な構造を表す関数依存グラフを探索空間とする新たなバグ検出アルゴリズムである関数依存グラフ最小化（Function Dependency Graph Minimization：FGM）法を提案する。関数依存グラフとは，関数の呼び出し関係，および関数のパラメタの依存関係を表す有向グラフである。FGM 法は、関数依存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、プログラマの応答に基づき関数依存グラフを変形・簡約化するという一連の操作を繰り返すことにより，関数依存グラフをバグ発生源を表すノードに縮約する。このため，実行結果に誤りが現れた場合、バグの伝播方向の解析に基づいて探索空間を簡約化することができる。また、探索空間を単純に二分する従来の戦略に比べ、プログラムの構造とバグの性質から導かれる新たな戦略を加えているため、質問回数の減少が期待できる。

2．背景

2．1　プログラム診断システム

プログラム診断システムは，図1に示すようにデバッガ

とデータベースからなる〔4〕。このシステムでは，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データベースに蓄積している。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。そして再び新たに質問を生成し、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。

（1）インスタンス

プログラムの実行時に関数呼出しにより新しく生成される実行環境をインスタンスと呼ぶ。関数呼出しの際の関数名f，入力パラメータの並びx，演算結果yからなる3つ組（f,x,y）を具現値と呼ぶ。今，インスタンスiの具現値を（f,x,y）とすると，f（x）=yが予期した通りの正しい入出力関係を与えている場合には，iは真であるという。また，xが予期した入力であるが，yが予期しない誤った出力である場合に，iは偽であるという。他方，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，iは未定義であるという。与えられたインスタンスが真，偽，未定義のいずれであるかをプログラマに問い合わせることをインスタンスの検査と呼ぶ。なお，インスタンスiの実行中に起きた関数呼出しによりインスタンスjが生成された場合に，iをjの親，あるいはjをiの子と呼ぶ。

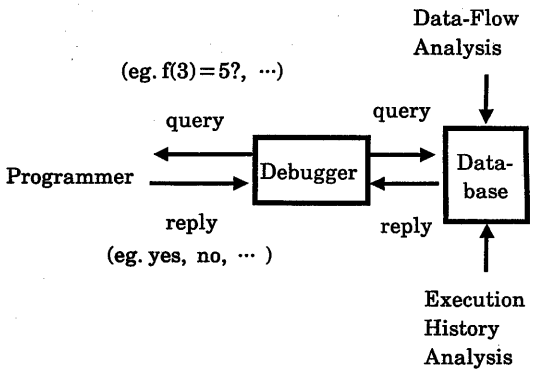


図1　プログラム診断システムの構成

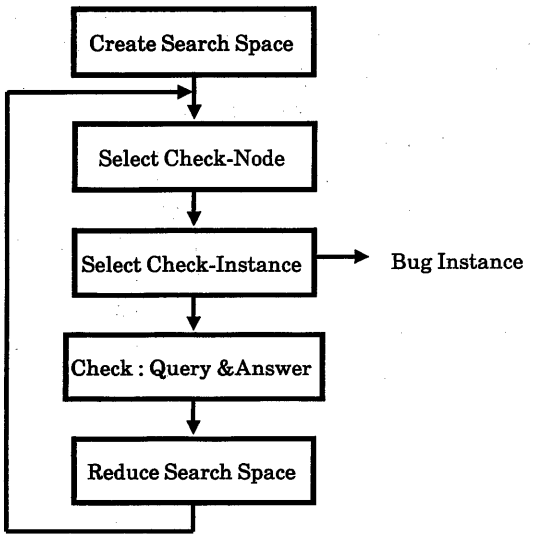


図2　バグ検出法の枠組み

（2）バグ検出アルゴリズム

次の2つの条件を同時に満たすインスタンスbをバグ発生インスタンスと呼ぶ。

・bは偽である。

・bの子の中に偽のものが存在しない。

バグ検出アルゴリズムは，図2に示すように，探索空間Sを生成した後，与えられた実行履歴を以下の手順で繰り返し検査してバグ発生インスタンスを求める。

①Sから1点sを選択して取り出す（検査点の選択）。

②sに対応するインスタンスを実行履歴から選択して取り出す（検査インスタンスの選択）。

③iを検査する（質問・応答）。

④検査結果に基づきSを変換する（探索空間の簡約化）。

バグ検出アルゴリズムでは，上の①から④の繰り返しの回数をできるだけ少なくするように探索空間と各手続きを定める。

2．2　バグ検出アルゴリズム

本稿では、グラフGは（N，A，I）の三つ組であり、Nはノード集合を、Aはふたつのノード間を接続するアークの集合を、またI∈Nは初期ノードをさす。また関数$\mathcal{N}$と$\mathcal{L}$を、　$\mathcal{N}$（G）=N,　　$\mathcal{L}$（G）=I
と定義する。

グラフGにおいて、v∈$\mathcal{N}$（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、$\mathcal{L}$（G）を初期ノードとするGの極大部分グラフであり、Pv と表記する。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り、Svと表記する。また、vの従属成分 Dv とは、$\mathcal{L}$（G）からw∈$\mathcal{N}$（Sv）へのすべてのパスがノード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。

これまでに提案されているバグ検出アルゴリズムを表1に示す。このように、従来のバグ検出アルゴリズムは実行履歴に基づいて生成されるインスタンス依存グラフ〔4〕

表1　従来のバグ検出アルゴリズム

	divide-and-query法〔5〕	射影グラフ最小化法〔4〕
探索空間	インスタンス依存グラフG	G で具現値の等しいノードを1点に射影したグラフG′
検査点の選択法	s∈Gなるsで，sの後続成分とそれ以外のノードの数の差が最小になる点s	s∈G′ なるsで，sの後続成分とそれ以外のノードの数の差が最小になる点s
検査インスタンスの選択法	検査点s の値を検査インスタンスの具現値とする。	
探索空間の簡約化 検査結果 真の場合	sの後続成分を取り除いて得られる部分グラフ	
探索空間の簡約化 検査結果 偽の場合	sの後続成分以外の点を取り除いて得られる部分グラフ	

のみを用いてバグの探索を行っており、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を繰り返す場合があり、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た。たとえば，再帰関数ひとつだけからなるプログラムを考える。この場合，バグは再帰部分と停止部分のいずれかに存在する。しかし，従来のアルゴリズムでは，実行履歴に基づいた探索空間を二分する点を選択するので，繰り返し再帰部分を検査して，最後にはじめて停止部分を検査することになる。この方法は，停止部分にバグがある場合には明らかに悪い選択法である。また，再帰部分にバグがある場合にも，再帰部分のすべてのインスタンスが偽となる可能性が高いと考えられるので停止部分のインスタンスを先に検査すべきである。しかし、従来の方法では、こうしたプログラムの静的な構造に依存した探索戦略を採用できなかった。

更に、関数に与えられたパラメータ自体が誤っているインスタンス（たとえば，ソートされたデータのリストがパラメータとして与えられるべきなのに、そうでないリストが与えられた場合）が検査された場合に、誤ったデータを生成した可能性のある計算のみを取り出して原因を探索することができなかった。また、プログラマにとっても、常に入力データが正しいとしてインスタンスの真偽いずれかを答えなければならないため、関数が複雑な場合には判断が難しくなるという問題点も存在した。

3．関数依存グラフの変換によるバグ検出

従来のバグ検出アルゴリズムの問題点は、実行履歴のみを用いてバグを探索していることに起因する。本章では，このような問題を解決するため、プログラムの静的な構造を表す関数依存グラフを探索空間とし、実行履歴を参照しながらバグ探索を行うバグ検出法を提案する。ここでは，まず関数依存グラフを用いたバグ検出法の基本的な考え方，およびバグ検出アルゴリズムを構築する際に必要な関数依存グラフの表現法と変換規則について述べる。具体的なバグ検出アルゴリズムは，次章に示す。

3．1　関数依存グラフによるバグ検出法

関数依存グラフとは、プログラム中で定義される関数をノードとし，関数間の静的な依存関係に従い各ノードを接続した有向グラフである。このグラフをバグ検出アルゴリズムの探索空間とすると次のような利点が得られる。

（1）検査点の取り出し法において，探索空間をできるだけ等しい2つの部分木に分割するような従来の選択法に加えて，プログラムの構造を考慮した選択法を容易に組み込める。

（2）検査インスタンスが未定義であることが分かった場合には，関数依存グラフから入力パラメタの生成に関与する関数を求め，その関数だけを探索空間に残せる。

（3）検査すべき関数を静的に定められる。このため，すべての実行履歴を保持せずに，検査の時にプログラムを実行させて必要な具現値だけを取り出すこともできる。

一方，関数を探索空間のノードとした場合には，次のような問題が生じるので，検査インスタンスの取り出しや探索空間の簡約化には、従来とは異なった方法が必要である。

・同じ関数が何度も呼ばれ，種々の入力パラメータが与えられた場合には，検査点（関数）に対応する具現値が一意に定まらない。このため，ある検査インスタンスの真偽が判明しても，検査点を探索空間から取り除けない場合がある。

・インスタンス依存グラフを用いるとバグのある関数名とバグを発生した入力パラメータの双方を求めることができる。これに対し、関数依存グラフでは、関数名のみしか特定できない。従って、どのような計算が実行されてバグのある結果を与えたのかが明らかにならない。

本稿で述べるバグ検出法では，このような問題を解決するため、具現値をノードに保持できる拡張した関数依存グラフの概念を導入している。以下に、この関数依存グラフの表現法と変換規則を示す。

3．2　関数依存グラフの表現法

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、呼出しアークとパラメータ・アークの二種類がある。

呼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に至る呼出しアークを張り、ｆ→ｇと図示する。たとえば、例１のプログラムでは、図３（ａ）の関数依存グラフとなる。

（例１）　ｆ＝λ〔〔ｘ〕ｇ（ｘ）＋１〕；
　　ｇ＝λ〔〔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⇢ｇと図示する。たとえば，例２の関数依存グラフは図３（ｂ）となる。

（例２）　ｈ＝λ〔〔ｘ〕ｇ（ｆ（ｘ））〕；
　　ｇ＝λ〔〔ｘ〕ｘ＊ｘ〕；
　　ｆ＝λ〔〔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の呼出し後続成分とは、呼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る。これに対し、単にｖの後続成分という場合は、呼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。

関数依存グラフに実行履歴の情報も保持できるようにするため、今後、関数依存グラフでは、以下に示す具現値を付加されたノード、再帰表現されたノード、および、ノードを畳み込まれたノードもノード集合に含められているものとする。

（１）具現値を付加されたノード

図３（ａ）のグラフを考える。関数ｇのノードに具現値（g,3,1）が付加された場合は、図３（ｃ）に示すように、ノードを　ｇ｛(g,3,1)｝　と表す。また、さらに具現値（g,5,3）が付加された場合は、ノードを

ｇ｛(g,3,1)，(g,5,3)｝

と表現し、以下、同様に付加していく。ここで、具現値の並びの順序は意味を持っていない。

（２）再帰表現されたノード

（例３）

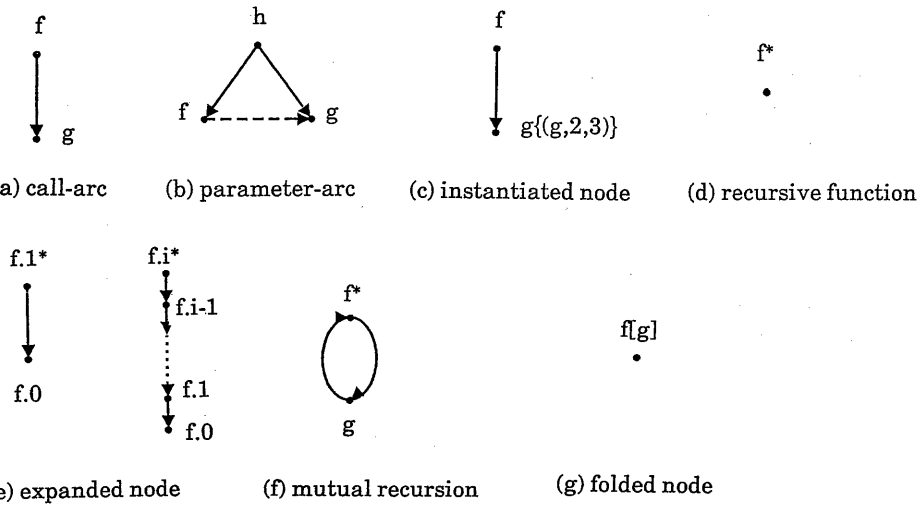
ｆ＝λ〔〔ｘ〕if ｘ>1 then ｆ（ｘ-1）＊ｘ else 1〕

例３のような再帰関数ｆの参照関係をそのまま表すと，ｆ→ｆ→ｆ→・・・　となる。関数依存グラフでは，再帰関数ｆを図３（ｄ），（ｅ）のように，１点ｆ＊，あるいはｉ＋１個の点からなるグラフで表現する。このように＊を付けたノードを再帰表現されたノードという。ここで，ｆ＊はｆ.0＊の省略形で、再帰関数ｆの基本形と呼ぶ。

（例４）

ｆ＝λ〔(x, y) if ｘ>0 then ｆ(x-1, y+1)
　　else ｇ(x, y)〕；
ｇ＝λ〔(x, y) if ｘ==0 then y
　　else ｆ(-x, -y)〕；

例４のように相互再帰を含む場合には、関数依存グラフ



(a) call-arc　(b) parameter-arc　(c) instantiated node　(d) recursive function

(e) expanded node　(f) mutual recursion　(g) folded node

図３　関数依存グラフの表現法

は図3（f）のようにサイクルを使って表される。

（3）畳み込まれたノード

ある関数gがバグを持たないと推定される場合、gをその関数を参照している関数fの一部とみなして扱えることが望ましい。しかし，この場合には，与えられた実行履歴の中にgのインスタンスで偽のものがなく，真にgが正しいと確定するまで，gのノードを関数依存グラフから取り除けない。このため，gをfの一部と見なしたことを陽に示すため，　f〔g〕　と表現し、このようなノードを畳みこまれたノードという。

たとえば、図3（a）においてgをfの一部と見なすと図3（g）が得られる。

3．3　関数依存グラフの変換規則

本節では、関数依存グラフGとインスタンス依存グラフDが与えられた場合にGを変換する規則を示す。なお、ノード集合HおよびEを、　H＝$\mathcal{N}$（G），　E＝$\mathcal{N}$（D）と定義する。

（1）具現値の付加

ノードp∈Hが与えられた時，Eの中でpが示す関数に入力パラメータを適用した具現値の集合を Ep　(Ep⊂E)とする。この時、次のような具現値q（q∈Ep，q∉C）を取り出してノードpに付加することを，pの具現化（instantiation）といい、そのようなqが存在することをノードpは具現化可能であるという。ここで，C（C⊂E）は、これまでにEから取り出されている具現値の集合である。

①pが再帰関数でない場合：　Dにおけるqの先行成分をSqとした場合，　i∈$\mathcal{N}$（Sq）∩ Ep，i∉C　を同時に満たす具現値iが存在しないような任意のqを取り出す。すなわち，まだ取り出されたことがないpの具現値のうち、$\mathcal{L}$（D）にもっとも近いものを取り出す。

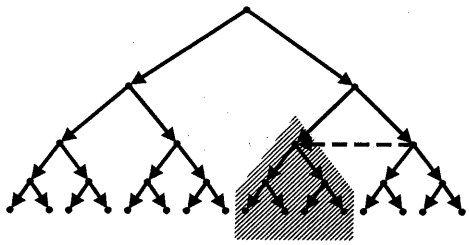
②pが再帰関数の場合：　pが再帰表現されている時には①と同じ方法でqを取り出す。他方，pが　f.i と表現されている時には，Eにおけるpの具現値のみをノード集合とする有向部分グラフ Gp を考える。 Gp のノードqについて，葉までの距離の最大値がiであり，かつCに含まれないような任意のqを取り出す。

（2）ノードの分解・合成

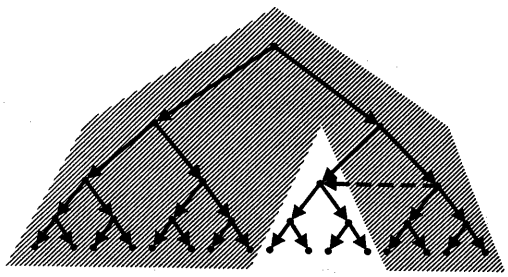
ノードgが与えられた時，Gの中のすべてのf→gをノードf〔g〕に置換することを、ノードgの畳み込み（folding）と呼ぶ。置換後の関数依存グラフについて，ノードfはノードgを畳み込んでいるという。また，畳み込まれたノードf〔g〕をもとに戻すことをfの展開（unfolding）と呼ぶ。Gの中に他の点を畳み込んでいる点がある

表2　関数依存グラフの変換規則

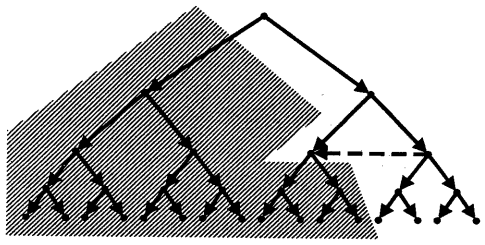
変換規則	機能（例）
instantiation : f	f ⇒ f {q}
folding : f	g → f ⇒ g〔f〕
unfolding : g	g〔f〕⇒ g → f
expansion : f.i*	f.i* ⇒ f.i+1* → f.i
inner-removal : g	k←f→g→h ⇒ k←f
outer-removal : g	k←f→g→h ⇒ g→h
non-input-removal : g	k←f→g→h ⇒ f



(a) inner-removal



(b) outer-removal



(c) non-input-removal

図4　関数依存グラフのノードの除去規則

時，Gは展開可能であるという。

再帰表現されている点 f.i* を f.i+1* → f.iに置換することを点 f.i* の伸張（expansion）と呼び，iを伸張番号と呼ぶ。ただし，f * が与えられた場合には f.1* → f.0 に置換する。Gの中に再帰表現されているノードがある場合に，Gは伸張可能であるという。

（3）ノードの除去

Gにおいて，ノードpの呼出し従属成分を取り除いた部分グラフを求めることを、pの内側除去（inner-removal）と呼ぶ。また，pの呼出し後続成分に含まれないノードを取り除いた部分グラフを求めることを、pの外側除去（outer-removal）という。今、pの呼出し先行成分のノード集合をS，S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Tとする。グラフGにおいて、ノード集合Tのすべてのノードの呼出し後続成分のノード集合とSの和集合をpの入力決定成分という。この時，pの入力決定成分に含まれないノードをGから取り除いた部分グラフを求めることをpの非入力部除去（non-input-removal）という。Gが二進木の場合に上記の各規則を適用するとそれぞれ図4の斜線部のノードが取り除かれる。

表2に各変換規則をまとめる。

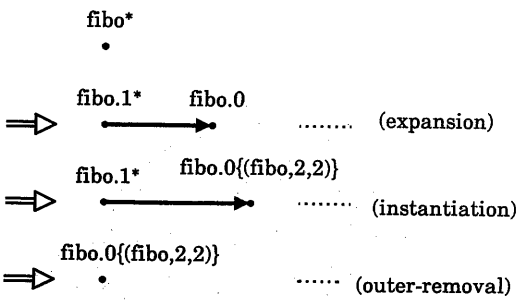


図5　関数依存グラフの変換例

4．関数依存グラフ最小化法

4．1　FGM法によるデバッグ例

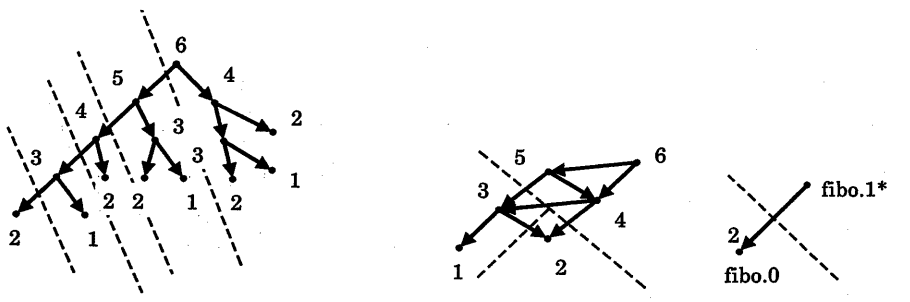
はじめに簡単な例により，3章で示した関数依存グラフの変換規則を用いたバグ検出法の概要を述べる。まず，次のようなバグのあるフィボナッチ数計算プログラム fibo を考える。

（例5）　fibo=λ〔〔n〕 if n≦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5のように関数依存グラフGを繰り返し変換するとバグを検出できる。図中，a⇒b　　・・（r）は，変換規則rによりグラフaをbに変換することを意味する。まず，fibo* を伸張する。これにより生成された点 fibo.0 を具現化し，具現値（fibo,2,2）を得る。次に，この具現値を検査する。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，Gに対して fibo.0 の外側除去を行う。これにより残されるノードがただひとつとなるので，その点 fibo.0 の具現値（fibo,2,2）の計算でバグが発生したことがわかる。関数 fibo に入力パラメータ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。

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6となる。この図から，プログラム fibo の場合，上述の変換ではプログラムの構造をより直接的に反映させた検査インスタンスの選択法を採っているので，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。

上の例を一般化し，FGM 法における関数依存グラフの変換過程を図示すると図7のようになる。ここで，Gi は，ノード数がiの関数依存グラフを表し，簡約化（reduce）は具現化で得られた具現値を検査し，その結果に基づき畳



(a) Divide-and-Query Algorithm　(b) PGM Algorithm　(c) FGM Algorithm

図6　バグ検出アルゴリズムの比較（関数fiboの探索空間の分割）

み込み，外側除去，内側除去，非入力部除去のいずれかの変換によりグラフのノード数を減らす操作である。FGM 法では，図に示すように，ノード数が1になるまでグラフを繰り返し簡約化する。そして、そのグラフG1 が伸張可能，あるいは展開可能であるなるば，伸張あるいは展開した後，

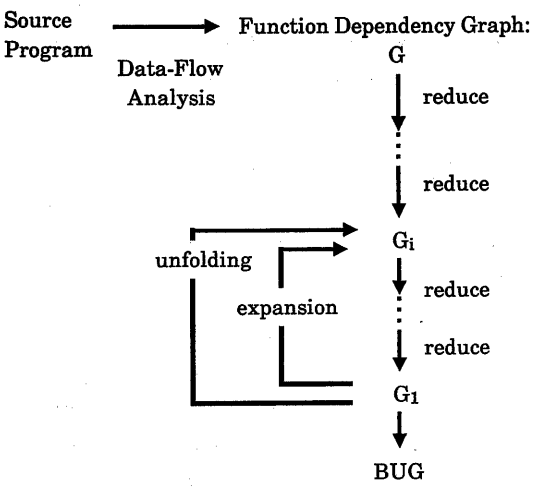


図7　関数依存グラフ最小化法の概念図

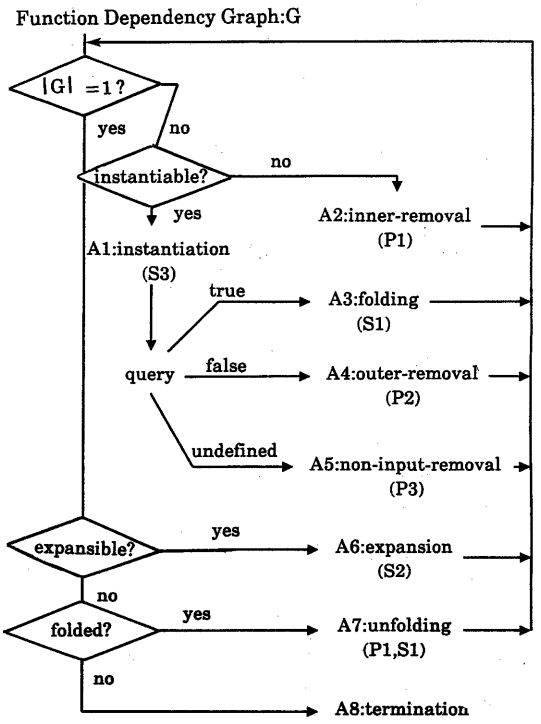


図8　関数依存グラフ最小化アルゴリズム

再度簡約化操作を繰り返す。もし，G1 が伸張・展開のいずれも可能でない場合には、残された唯一のノードｖがバグの発生源を示す。すなわち，ノードｖに付加された偽の具現値が，バグのある関数の名前，バグ発生時の関数の入力パラメータ，およびその計算結果を与える。

FGM 法においては，次節に示す適用則に従い 3.3節で述べた変換規則を繰り返し関数依存グラフに適用する。プログラマの役割は，簡約化の際に関数の具現値が真，偽，あるいは未定義であることを教え，適切な適用則の選択を可能にすることである。

4．2　関数依存グラフ最小化によるバグ検出

関数依存グラフＧはバグに関して次のような性質を持っている。

〔性質 P1 〕Ｇのノードｖが具現化不能の場合，ｖに付加されたすべての具現値が真であるならば，ｖの呼出し従属成分以外にバグのある関数が存在する。

〔性質 P2 〕Ｇのノードｖのある具現値が偽である場合．ｖの呼出し先行成分の中にバグのある関数が存在する。

〔性質 P3 〕Ｇのノードｖのある具現値が未定義である場合、ｖの入力決定成分の中にバグのある関数が存在する。

プログラムの構造とバグに関しては，次のような仮説を立てる。

〔仮説 H1 〕すべての関数において，ある具現値が真であると，その関数にバグがない可能性が高い。

〔仮説 H2 〕関数ｆを実行させて具現値ｑが得られたとする。この時，ｆにおいて実際に実行された式の集合をｑの必須式と呼ぶ。すべての関数において，ある具現値ｑが真である時，ｑと同じ必須式をもつようなすべての具現値は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。

上のような仮説に基づき，FGM 法では次のような戦略をとる。

〔戦略 S1 〕ある関数のある具現値が真であるならば，その関数を畳み込む。

〔戦略 S2 〕再帰関数ｆについては，ｆの停止部分の任意の具現値 q0，　q1 → q0 なるｆの再帰部分の具現値 q1 の順に具現値を調べる。

〔戦略 S3 〕次のような方策により，グラフＧを簡約化したグラフのノード数が最小になるように検査点を選択する。今，Ｇのあるノードｖ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 Tv，Fv，Uv とする。この時，(Tv+Fv+Uv) が最小となるｖの中から，Tv，Fv，Uvの最大値が最小となるような点ｖを選択する。

FGM 法では，上記の性質および戦略に基づき，図８に示すように A8 に到達するまで関数依存グラフＧに対して繰り返し変換規則を適用する。図において，ｉはＧのノード数，ｖは戦略 S3 に従い選択した点，ｑはｖを具現化して新たに付加された具現値である。 A1 ～A8 は適用則の識別子である。また，適用則に付した（ ）内の性質および戦略名はその変換規則を適用する理由を表す。たとえば，ｉ≧2 の時，まず，戦略 S3 によりＧから点ｖを選択し，具現化する。具現化できない場合には性質 P1 に従い，ノードｖの内側除去（inner-removal ）を施す。他方，具現化により具現値ｑが点ｖに付加された場合には，ｑについて質問する。この質問の答により，ｑの真，偽，未定義が決定すると，それぞれ畳み込み，外側除去，非入力部除去を施す。また，図のように，ｉ＝１の時には，Ｇ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，Ｇの唯一の点ｖに含まれる具現値がバグ発生インスタンスを与える。

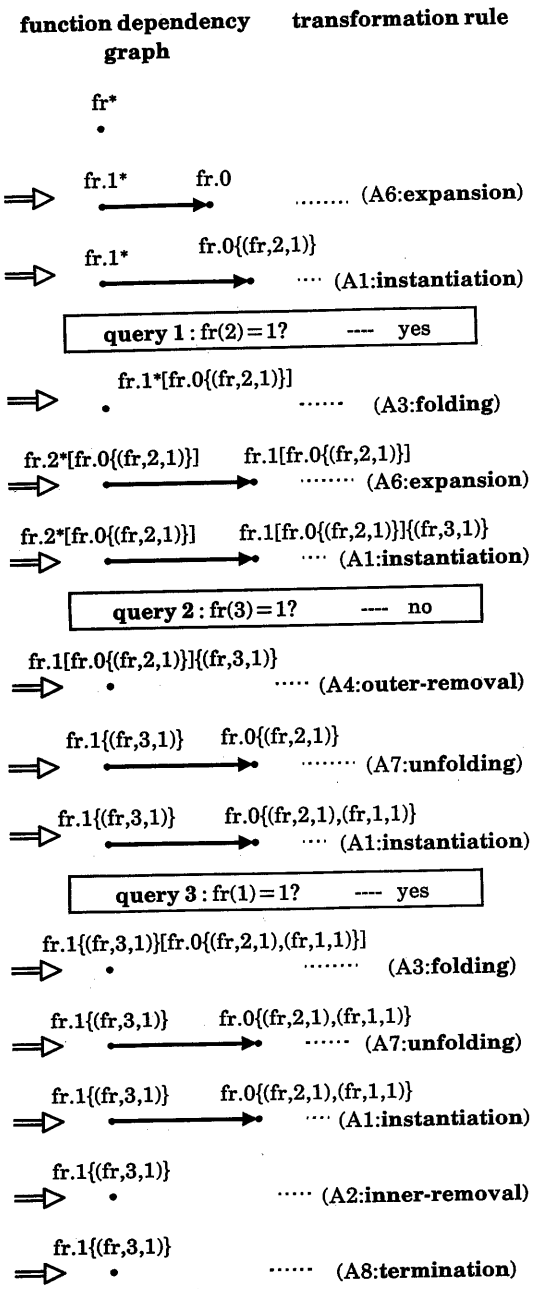


図９　関数依存グラフ最小化法によるバグ検出例

４．３　関数依存グラフ最小化法の適用例

一例として，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６〕

```
fr＝λ〔〔n〕 if n≦1 then 1
        else fr （n -1） *fr （n-2 ）〕
```

この場合には，前の例に比べて少し多い変換ステップを要する。しかし，図９に示すように繰り返し変換規則を適用することにより，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）にバグがあることが判明する。

ｎ ≧3 なる任意のｎに対してfr（ｎ）の実行履歴が与えられると，図９のように関数依存グラフが変換される。従って，ｎの値が大きく実行履歴のデータが多い場合でも，すべて同じ質問回数（３回）でバグを検出できる。

５．データフロープログラム診断システム

５．１　概要

並列処理環境下における関数型プログラムのデバッグ法としてのFGM 法の有効性と問題点を明確にするためにデータフロープログラムデバッグシステムを作成した。このシステムは先に報告したデータフロープログラムデバッグシ

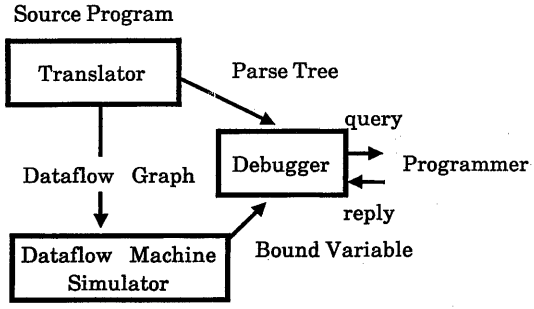


図10　データフロープログラム診断システムの構成

```
(func (member u v) (return m)
 (let m
  (cond
   ((null v) nil)
   ((equal u (car v)) T)
   (T (member u (cdr v))))]

(func (union u v) (return z)
 (let z
  (cond
   ((null u) nil)
   (T (clause
       (let x (union (cdr u) v))
       (let y (car u))
       (cond
        ((member y v) x)
        (T (cons y x))))]
```

(a) Union program with a bug

```
=>(?= union (2 3 1 9)(5 1 3 7))

=>fgm

Is (MEMBER 2 (5 1 3 7))=() true?  yes

Is (UNION () (5 1 3 7))=() true?  no

There is a bug in the following instance!
Modify the function.

[env (0 9 0)]        (UNION () (5 1 3 7))=()

<< 5 nodes  2 steps ( c = 2.50000^0 ) >>

( FUNC
  (UNION (U = ()) (V = (5 1 3 7)))
  (RETURN (Z = ()))
  ( LET
    (Z = ())
    ( COND
      ((NULL (U = ())) NIL)
      ( T
        ( \\
          ( CLAUSE
            (LET (X = *) (UNION (CDR (U = *)) (V = *)))
            (LET (Y = *) (CAR (U = *)))
            ( COND
              ((MEMBER (Y = *) (V = *)) (\\ (X = *) \\))
              (T (\\ (CONS (Y = *) (X = *)) \\)) ) )
          \\ ) ) ) ) )
```

(b) Program diagnosis for union

図11　集合の和の計算プログラムのバグ検出

```
(func (split x)(return s1 s2)
 (let (s1 s2)
  (cond
   ((null x)(tuple nil nil ))
   ((null (cdr x)) (tuple x nil ))
   (T (clause
       (let (u v) (split (cddr x)))
       (tuple
        (cons (car x) u)
        (cons (cadr x) v)))))]

(func (merge x y)(return s)
 (let s
  (cond
   ((null x) y)
   ((null y) x)
   ((< (car x) (car y))
     (cons (car x) (merge (cdr x) y)))
   (T (cons (car y) (merge x (cdr y)))))]

(func (mergesort x)(return s)
 (let s
  (cond
   ((null (cdr x)) nil)
   (T (clause
       (let (u v) (split x))
       (merge (mergesort u) (mergesort v))))]
```

(a) Mergesort program with a bug

```
=>(?= mergesort (8 2 1 6 4 7 5 3))

=>fgm

Is (SPLIT (8 2 1 6 4 7 5 3))
=((8 1 4 5) (2 6 7 3)) true?  yes

Is (MERGE () ())=() true?  yes

Is (MERGESORT (8))=() true?  no

There is a bug in the following instance!
Modify the function.

[env (0 26 0)]       (MERGESORT (8))=()

<< 4 nodes  3 steps ( c = 1.33333^0 ) >>

( FUNC
  (MERGESORT (X = (8)))
  (RETURN (S = ()))
  ( LET
    (S = ())
    ( COND
      ((NULL (CDR (X = (8)))) NIL)
      ( T
        ( \\
          ( CLAUSE
            (LET ((U = *) (V = *)) (SPLIT (X = *)))
            (MERGE (MERGESORT (U = *))
                   (MERGESORT (V = *))) )
          \\ ) ) ) ) )
```

(b) Program diagnosis for mergesort

図12　マージソートプログラムのバグ検出

ステム〔4〕を拡張し，FGM 法をバグ検出アルゴリズムとして用いたものである。このシステムは，図10のように，言語処理系，データフローマシンシミュレータ，デバッガの３つのプログラムからなる。言語処理系では，S式を用いて記述されたLisp風のソースプログラムをデータフローグラフに変換する。ここでは，言語処理系の簡単化のためにS式表現としたが，記述言語の計算構造はデータフローマシン用関数型言語 Valid 〔6〕のサブセット〔4〕である。

5．2　プログラムデバッグの例

簡単なプログラムを前節で述べたシステムで実行させ，デバッグした例を示す。以下の図では，下線部はプログラマの入力を表し，その他はデバッガの出力を表す。

（1）集合の和の計算

図11（a）のような集合の和を求める関数にバグを入れたプログラム unionを考える。このプログラムは，データフローグラフに変換されシミュレータにより実行される。実行結果に誤りがあるのでプログラマは診断システムを起動し，図11（b）のようにデバッガと質問・応答を繰り返すとバグを含むインスタンスを検出できる。

（2）マージソートの計算

図12（a）は，バグを含んだマージソートプログラムである。（1）と同様にして，このプログラムを実行させ，プログラム診断システムを起動すると図12（b）のようにバグを含むインスタンスが検出される。

6．むすび

本稿では，関数型プログラムを並列実行させるシステムに適し，関数依存グラフの変換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関数依存グラフ最小化（FGM）法を提案した。従来のバグ検出アルゴリズムが実行履歴より生成されるインスタンス依存グラフのみを用いているのに対し、FGM 法は、関数の静的な依存関係を表している関数依存グラフも使用して解析を行い、プログラマに質問を発する。そして，その応答を解析して関数依存グラフを簡約化する。この一連の操作を繰り返すことにより，関数依存グラフをバグ発生源を表すノードに縮約し、バグ発生インスタンスを同定する。FGM 法は，従来のバグ検出アルゴリズムに比べ次のような利点を持つ。

（1）バグの探索空間は関数の呼び出し関係，および関数のパラメタの依存関係を表す有向グラフである。このため，実行結果に誤りが現れるとバグの伝播方向を解析することにより探索空間を簡約化することができる。

（2）探索空間を単純に二分する従来の戦略にプログラムの構造とバグの性質から導かれる新たな戦略を加えたバグ検出アルゴリズムである。

（3）与えられたプログラムにおいて，プログラマに質問すべき場所を静的に定められるので，実行履歴をすべて保持する必要がない。

従って、本方式を用いると、再帰関数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。

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御意見を頂いた情報通信基礎研究部・第二研究室の後藤厚宏主任に深く感謝します。

参考文献

〔1〕Darlington, J., Henderson, P. and Turner, D. A. (eds.) "Functional Programming and its Application," Cambridge University Press, 1982.i
〔2〕Treleaven, P., Brounbridge, D.R. and Hopkins, R. P., "Data-Driven and Demand-Driven Computer Architecture," ACM Computing Surveys, Vol.14, No. 1, pp. 94-143,March 1982.
〔3〕Vendahl, S. R., "A Survey of Proposed Architectures for the Execution of Functional Languages," IEEE Trans. on Computers, Vol. C-33, No. 12, pp. 1050-1071, 1985.
〔4〕高橋，小野，雨宮，"並列処理環境下における関数型プログラムのデバッグ方式，"情処ソフトウェア基礎論研究会，11—4，1984.
〔5〕Shapiro, E. Y., "Algorithmic Program Debugging," MIT Press, 1983.
〔6〕雨宮，長谷川，小野，"データフロー計算機用高級言語 Valid，" 通研実報 Vol. 32, No. 6, 1984.