ソフトウェア基礎論 11－4
（１９８４．１２．１４）

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

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）

"Parallel-processing-oriented Algorithmic Bug-detection Method for Functional Programming Languages"

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

A new algorithmic debugging method for functional programming languages named Projected Graph Minimization (PGM) method is proposed. This method detects bugs in programs more efficiently than the conventional divide-and-query algorithm, and is especially suitable under highly-parallel computing environment.

Dynamic execution history data as well as static dependency data created by a compiler are stored in a relational data-base. Debugging information such as subroutine trace can be retrieved by applying relational algebra operators to the data-base. This approach frees programmers from annoying about where and when programs are executed. The method detects locations of bugs by analyzing the programmer's answers to the queries automatically generated by the debugging system. This paper also clarifies the effectiveness of the proposed method by showing the results of the experimental debugging system for dataflow machines.

Keywords: debugging, dependency graph, relational data-base, 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。このため，関数型プログラミングや関数型プログラムを高速実行する計算機アーキテクチャに対する関心が高まっている。特に，副作用を排除したその計算構造（セマンティクス）は，並列処理に適しているので，データフローマシン〔1 ～ 8〕，リダクションマシン〔1,9,10〕等の並列アーキテクチャの研究が活発に行われている。

データフローマシンについては，既にいくつかの実験システムが稼働している状態であり，関数型プログラムの高並列計算機の実現性は高いと考えられる。しかし，並列処理システムのためのプログラミング環境，特にプログラムデバッグ，の観点からは検討すべき問題が多く残されている。たとえば，データフローマシンでは，プログラムカウンタやメモリの概念を排除しているためトレースやメモリダンプなどの従来の手法をそのまま用いることができないので有効なデバッグ手法の開発が望まれている。

従来の多くのプログラムデバッガは，プログラムを実行させた時の計算機の状態変化を追跡することを基本としている。たとえば，プログラムの実行トレース，ブレークポイント設定，メモリやスタックのダンプなどの機能を備えている〔11〕。高級言語のプログラムデバッガでは，計算機のメモリやプログラムカウンタなどのハードウェア資源を意識させないレベルで状態変化をトレースできるものもある〔11〕。たとえば，INTERLISP や MACLISP などの言語では，プログラムの実行中断やスタックの読み出しのために便利な機能が取り揃えられている〔12,13〕。

上述のような機能は，逐次処理プログラムデバッグの際には強力なツールとなるが，並列処理プログラムのデバッグのためには不十分な点があった。たとえば，①並列計算機のプログラムでは，非同期処理を含むのでトレースが難しいこと，②実際に実行される場所が分散しているので誤りのあるプログラムを発見するのが難しいこと，③通信の遅延があるので実行してからモニタするまでに状態の変化が起こる場合があること，④一般に並列処理ではプログラムが大きくなりデバッグ時に解析すべきデータ量が多いことなどの点が挙げられる〔14〕。また，プログラムの実行時に発生する時系列データを監視する従来のトレース方式は，プログラムテキストの変数定義の系列に従って実行結果を追跡することを基本としている。これに対して，関数型プログラムでは変数定義の順序は意味を持たないので，従来のトレース方式により時系列データを追跡してデバッグを進めることは困難である。

本稿では，並列処理環境下における関数型プログラムに適したデバッグ方式を提案する。以下，まず，関係データベースを用いたデバッグ法について述べる。この方式ではプログラムの実行履歴などのデバッグに必要なデータをす

べて関係データとして扱い，デバッグの際にデータの発生に関して時間的要因と空間的要因を考慮する煩わしさをプログラマから取り除いている。また，本方式では，新たに提案する射影グラフ最小化法と呼ぶバグ検出アルゴリズムに従ってデバッガがプログラマへの質問を繰り返し，その応答を解析することによりバグのある関数を特定する。射影グラフ最小化法は，Shapiro が Prolog プログラムのバグ検出のために提案した方法〔15〕を基本とし，関係演算を用いてバグ検出に必要な質問回数を減少させたものである。最後に，データフロープログラムに対して上記デバッグ法を実現した実験システムについて，その構成およびデバッグ例を示す。

2. 関係データベースを用いたデバッグ法

2．1 関係データベースを用いたデバッグシステム

関数型プログラムのコンパイルおよび実行により得られる情報を関係データの形で管理し，このデータベースを用いてプログラムをデバッグする方式を提案する。この方式を用いたプログラムデバッグの流れを図1に示す。図において，→はプログラムのコンパイルと実行の処理の流れを示している。この処理は，従来と同様であり，構文解析とコード生成により作成されるコードが実行される。従来のデバッグ法では，コードを実行させた結果生じる計算機の状態変化を直接追跡する形式であった。これに対して本方式では，図において⇨で示す機構を加え，次のように関係データの検索操作に基づいたデバッグ作業を行う。まず，実行前に，プログラムの構文木をデータフロー解析して関数や変数の相互のデータ依存関係を求め，関係データベース（RDB ）に登録する。また，プログラムの実行履歴データの解析により，変数の束縛関係，あるいは，実行時の関数の相互依存関係などを関係データの形でRDB に登録する。プログラマは，RDB への問い合わせを繰り返し，その応答を解析することによりバグを検出し，ソースコードを更新する。本方式の特徴は，次の通りである。

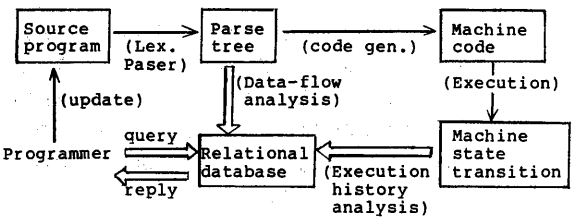


図1．関係データベースを用いたプログラムデバッグシステム

① プログラム中の変数と関数について局所的な相互のデータ依存関係を双方向に検索することにより実行結果を解析するので，変数定義の順序が意味を持たない関数型プログラムに適している。

② この方式は，従来デバッグ用データの基本となっていたスタック情報を抽象化しすべて「関係」として捉えることにより，プログラマが時間的要因（演算が行われた時刻と時間に依存する状態変化に関する情報）と空間的要因（演算が行われたプロセッサと演算器，および使用したメモリに関する情報）に煩わされずにデバッグを進めることを可能にしており，並列プログラムのデバッグの難しさを緩和している。

③ 関数型プログラムでは変数と値の対応関係に一意性が保証されているので実行履歴を関係データとして容易に保持できる。

④ タビュレーション技法〔16〕をデバッグに応用することにより効率的なシステムを実現できる。即ち，デバッグ用の解析データを得るために必要な計算の重複を避けること，および保持すべきデータの量を減少させることができる。一例として，ある関数 f の内部のトレースとバックトレース〔11〕を考える。この場合 f の全変数の束縛状態を知る必要がある。しかし，全変数の値を保持しなくとも，すべての関数の入出力関係を保持しておけば f の式のみを再計算するだけで良い。

2．2 基本関係データとそのオペレータ

次節に示すデバッグ用の各種データを求めるために必要な関係データと基本オペレータについて述べる。

（1）基本データ

以下に示す5つの関係データをデバッグの際の基本データとする。ここで，①，②はコンパイル時のデータフロー解析により求められる。また，③～⑤は，プログラム実行時に求められる。

① 関係SF (F,Fc) ：関数の静的な依存関係

(f,fc) ∈ SF は，関数f が関数fcを呼び出すことを意味する。

② 関係SV (F,V,Vc) ：変数の静的な依存関係

(f,v,vc) ∈ SV は，関数f において変数vcの定義式で変数v を参照していることを意味する。

③ 関係DD (I,Ic) ：関数の動的な依存関係

プログラム実行時に関数呼出が起こると新しい実行環境が生成される。その実行環境をインスタンスと呼ぶ。この時，(i,ic) ∈ DD は，インスタンス i において関数呼出が発生しインスタンスicが生成されたことを意味する。

④ 関係DI (I,F,X,Y) ：インスタンスの実行状態

(i,f,x,y) ∈ DI は，インスタンスi において，

y = f (x) の計算が行われたことを意味する。ここで，x,y は関数f の入力および出力パラメータの値の並びである。 (f,x,y) を関数の入出力関係と呼ぶ。

⑤ 関係DV (I,V,W) ：変数の束縛状態

(i,v,w) ∈ DV は，インスタンスi において変数v が値 w をとることを意味する。

(2) 基本オペーレータ

集合演算∪ (union) , ∩ (intersection) , − (difference) および以下に示す関係代数演算〔17〕を基本オペレータとする。

①射影 (projection) ： Rの属性 (A1,A2,・・・, An) への射影を R〔A1,A2,・・・, An〕と表す。

②結合 (join) ： ここでは，結合演算のうち自然結合 (natural-join) のみを用いる。R の属性A と Sの属性B との間の等結合によりできる関係について属性B を落として属性A のみを残すことを R〔A = B〕S と表す。特に，R とS に共通するすべての属性について上記演算を施す時，R ⋈ Sと簡略化して表す。

③制約 (restriction) ： ここでは，特定の属性がある値を持つ組を取り出すために制約演算を用いる。属性A が値c である組を関係R から取り出す操作を R〔A = c〕と表す。

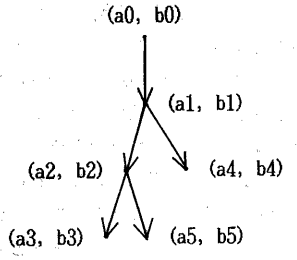
④個数 (cardinarity) ：R の要素の数を | R | と表す。

⑤閉包 (transitive closure) ：デバッグでは，関数や変数の相互の依存関係に基づいて双方向にデータを検索する操作が常用されるので，このために必要な演算の簡約記述法を導入する。

関係 R〔A,Ac,B〕において閉包演算を定義する。A の定義域をDaとすると,Ac の定義域はDa∪ {⊥} である。ここで，“⊥”は，以下で述べる親子関係において終端を表す特別な記号である。今，m0= (a,ac,b) ∈ R を考える。任意のm0= (a,ac,b) について， m1 ∈ R〔A =ac〕を満たすすべてのm1を，a の子供，あるいは a のAcに関する1代目の子孫と呼ぶ。また，各m1= (a1,ac1,b1) についての R〔A =ac1 〕の union をa のAcに関する2代目の子孫と呼ぶ。同様な操作をn 回繰り返して出来る関係をa のAcに関するn代目の子孫と呼びR ↓ (A = a | Ac,n) と表す。上記関係演算を用いると， R↓ (A =a | Ac,n) は，次のように再帰的に定義される

$$R \downarrow (A=a \mid Ac,n) = \begin{cases} (R\downarrow (A=a \mid Ac,n-1)\ 〔Ac〕)\ 〔Ac=\underline{A}〕\ R & \cdots\ n\geqq 1 \text{ の場合} \\ R〔A=a〕 & \cdots\ n=0 \text{ の場合} \end{cases}$$

A	B	Ac
ao	b0	a1
a1	b1	a2
a1	b1	a4
a2	b2	a3
a2	b2	a5
a3	b3	⊥
a5	b5	⊥
a4	b4	⊥



(a) Relation E　　(b) Graph g (E,A,Ac,B)

図2．親子関係を持つ関係データとそのグラフ

一例として図2 (a) の関係を考える。この時,A=a1のAcに関する1代目の子孫，および2代目の子孫は，それぞれ { (a2,a3,b2) , (a2,a5,b2) , (a4, ⊥,b4) } , および { (a3, ⊥,b3) , (a5, ⊥,b5) } となる。

a のAcに関するすべての子孫，R ⇓* (A =a | Ac) は，次式で与えられる。

$$R\Downarrow^{*}\ (A=a \mid Ac) = \bigcup_{n=0}^{\infty} R\downarrow (A=a \mid Ac,n)$$

A =a のB に関するn 代前の先祖 R↑ (A =a | Ac,n) , A =a のAcに関するすべての先祖 R⇑* (A =a | Ac) も上述の定義と同様に次のように定義される。

$$R\uparrow (A=a \mid Ac,n) = \begin{cases} (R\uparrow (A=a \mid Ac,n-1)\ 〔A〕)\ 〔A=\underline{Ac}〕\ R & \cdots\ n\geqq 1 \text{ の場合} \\ R〔A=a〕 & \cdots\ n=0 \text{ の場合} \end{cases}$$

$$R\Uparrow^{*}\ (A=a \mid Ac) = \bigcup_{n=0}^{\infty} R\uparrow (A=a \mid Ac,n)$$

2．3 デバッグ支援機能

本節では，デバッグのために必要な各種データを前節に示した基本関係データとそのオペレータを使って求める方法を示す。本方式によるデバッグでは，有向グラフで表される変数と関数の相互のデータ依存関係を双方向に検索する。従って，デバッグ用のデータ作成の中心は基本データから必要な有向グラフを生成することである。このため，まず関係データと有向グラフとの関連付けを行う。

(1) 関係データと有向グラフ

前節の閉包の定義において用いた関係 R (A,Ac,B) を考える。R の (A,B) への射影，および (A,Ac) への射影を求めることにより，次のように有向グラフが得られる。

(a, b) ∈ R〔A, B〕の時，aをノード識別子，bをノードの値とする。今，ふたつのノード (a1, b1)，(a2, b2) において，(a1,a2) ∈ R〔A,Ac〕を満たすならば，ノードa1からノードa2に向けたアークを張る。これによりできる有向グラフGをRのグラフと呼び g (R,A,Ac,B) と表す。前述の図2 (a) の関係Eに対して上記操作を施すと，図2 (b) のようなグラフg (E,A,Ac,B) を得る。

また，Rのグラフにおいてaを根とする極大部分グラフをsg (R,A,Ac,B, a) と表す。この時，閉包演算の定義より次式が成立する。

$$sg(R,A,Ac,B,a) = g(R\Downarrow^{*}(A=a \mid Ac), A,Ac,B)$$

(2) デバッグ支援機能

閉包演算により求められる主なデバッグ支援機能を表1に示す。表2では，デバッグ支援機能，その実現のために必要な関係データ，閉包演算の種類（子孫または先祖），および演算結果のグラフとその根，ノードの識別子 (id) と値を表している。機能名は，文献〔11〕の定義に従っている。たとえば，次章で使うインスタンス依存グラフは，ノードの値が (f,x,y)，ノードの識別子がi,であり，g (DF,I,Ic, (F,X,Y)) と表される。また，このグラフで，ノードi0を根とする部分木は，g (ST* (i0) ,I,Ic, (F,X,Y)) と表され，文献〔11〕のsubroutine traceの機能に対応する。

閉包演算を用いないで求められる機能としては，関数f内の変数vの束縛状態〔variable trace〕VT (f,v)，インスタンスiのすべての変数の束縛状態〔snapshot dump〕SD (i) などがあり，それぞれ次のように与えられる。

$$VT(f,v) = (DI \bowtie DD \bowtie DV)\,〔F=f, V=v〕\,〔I,W〕$$

$$SD(i) = DV\,〔I=i〕\,〔V,W〕$$

3. プログラム診断システム

3.1 プログラム診断システムの概要

本節では，プログラム診断システムの目的，構成，およびプログラム診断アルゴリズムの概要を示す。診断アルゴリズムについては，同様な目的を持つ従来の研究〔15〕と関連付けて本稿で採ったアプローチの特徴について述べる。

(1) 目的

2章で述べたように関係データを用いて関数と変数のデータ依存関係を双方向にトレースしてバグを検出する方式は，デバッグの際に時間的要因と空間的要因を考慮する煩わしさをプログラマから取り除く。しかし，関係データの検索手順は完全にプログラマに任されているので，大量のデバッグ用データが生成された場合にはバグに到達するまでに多くの試行錯誤を要しプログラマに多大な負担をかけることになる。プログラム診断システムは，この問題を解決するために，システムがバグ検出アルゴリズムに従って検査すべきデータを決定しプログラマに提示するものである。

本稿で述べるバグ検出アルゴリズムは，バグを発生したインスタンスを検出する問題に限定し，そのインスタンスにおいて実際にバグを発生させた式の特定はプログラマに任せるものとする。また，プログラムが停止し得られた結果に誤りがある場合のデバッグに限定し，コンパイラで誤りが検出される場合，或いは，無限ループを含みプログラムが停止しない場合は除外する。言い換えると，このアルゴリズムはプログラムの実行結果から，以下の条件を同時に満たすインスタンス bを求めるものである。ここで，bをバグ発生インスタンス，bを求めるアルゴリズムをプログラム診断アルゴリズムと呼ぶ。

① (f,x,y) ∈ DF〔I=b〕〔F,X,Y〕とする時，f(x)=y が成立しない。

② (f,x,y) ∈ S=ST (b,1)〔F,X,Y〕とする時，f(x)=y が成立するか，或いはS が空である。

表1 デバッグ支援機能

デバッグ機能		必要な演算とデータ		演算結果のグラフ			
機能名〔11〕	表記法 (n 代目，全要素)	演算データ D	閉包	意味	根	id	値
execution trace	ET (i,v0,n)，ET* (i,v0)	DV′ = (DV⋈DI⋈SV)〔I=i〕〔V,W,Vc〕	子孫	インスタンスiの変数依存グラフ	v0	v	w
flowback analysis	FA (i,v0,n)，FA* (i,v0)		先祖				
subroutine trace	ST (i0,n)，ST* (i0)	DF=DI⋈DD	子孫	インスタンスの依存グラフ	i0	i	(f,x,y)
retrospective trace	RT (i0,n)，RT* (i0)		先祖				
call graph	CG (f0,n)，CG* (f0)	SF	子孫	フローグラフ	f0	f	f

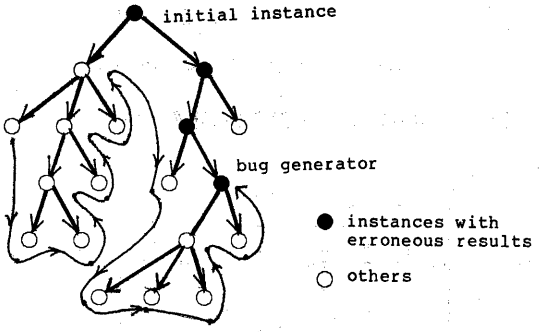


図4.　バグ発生インスタンスの検索

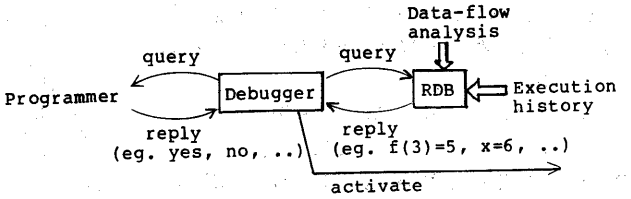


図3.　プログラム診断システムの構成

（2）構成

プログラム診断システムは，図3に示すようにデバッガと関係データベースから成る。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関係データベースを検索する。更に，デバッガは検索データを使って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検出する。

ここでは，プログラマが行う検査をできるだけ簡単にするため，デバッガとプログラマとの質問・応答は文献〔15〕と同様な形式とした。即ち，デバッガがある関数の入出力関係の真偽を質問しプログラマが yes または no と答える。

（3）プログラム診断アルゴリズム

バグ発生インスタンスを求める最も単純なアルゴリズムは，図4のようにインスタンス依存グラフを探索木とし，その葉の方から順にすべてのインスタンスの関数の入出力関係の正当性を問い合わせるものである。この図で，○は関数の入出力関係が正しいインスタンス，●は正しくないインスタンス，→ は検索順序を表す。この順序に従って検索し最初に到達する●がバグ発生インスタンスである。インスタンス依存グラフのノード数をmとすると，このアルゴリズムは質問回数がmのオーダとなり効率が悪い。質問回数を減少させるためには，①探索木を小さくする，②検査するインスタンスの選択法を改善する，という2つのアプローチが考えられる。

Shapiro が提案したProlog プログラムの診断アルゴリズム（divide-and-query アルゴリズム）〔15〕は，divide-and-conquer の考え方をインスタンスの選択に適用して質問回数を減少させるものであり，②に沿ったアプローチといえる。彼のアルゴリズムは，探索木の分割を繰り返すことによりバグ発生インスタンスを求めるものである。このため，必要な質問回数がlog m のオーダとなり〔15〕，この方法は質問回数を減少させる上で効果的な選択法である。しかし，彼のアルゴリズムでは，インスタンス依存グラフを探索木としているため，次のような問題が生じる。

・　探索木に同じ関数入出力関係を値として持つノードが複数存在するので，探索木が大きくなる。

・　すでに質問した関数入出力関係と同じ値のノードが選ばれる可能性がある。このため，質問を重複させないためには，質問・応答の履歴を別途管理する必要がある。

本稿で示す射影グラフ最小化（Projected Graph Minimizaion）法と呼ぶアルゴリズムは，インスタンスの依存グラフGの準同型グラフG'を探索木とし，G'に対して divide-and-conquerの考え方を適用するものである。この点で，このアルゴリズムは，①に沿ったアプローチを加えることにより上記問題点を解決するものである。次節に PGM 法の詳細を示す。

3．2　射影グラフ最小化（PGM）法

はじめに，PGM 法において探索木とする射影グラフについて述べ，次にこのグラフを用いたバグ検出アルゴリズムを明らかにする。

（1）射影グラフ

以下の議論では，Q = (F,X,Y) とし，インスタンスの実行状態をDI′ (I, Q) で表す。即ち，

DI′ (I, Q) = DI (I,F,X,Y)

とする。この時，Shapiro のアルゴリズムの探索木Gは，次式で与えられるDRを用いて，G＝g (DR,I,Ic,Q) と表現できる。

DR=DF′ 〔Ic=J〕 (DF′ (I →J, Ic →Jc, Q →Qc))
〔I,Ic,Q,Qc〕

DF′ =DI′ ⋈DD

ただし，DF′ （I →J，Ic →Jc，Q →Qc）は，名前の付け替えによりDF′ の属性I,Ic,Qをそれぞれ J,Jc,Qc とすることを表している。

さて，インスタンス集合I からQ への写像を

h （i ）=q　　iff　　(i,q) ∈DI′

と定義すると，任意の（i0，q0），（i1，q1）∈DI′ に対し，q0 ≠q1 ならば i0≠i1 であるのでh は関数となる。

今，（i,ic,q,qc ）∈DR の時，i $\otimes_{DR}$ icとすると，グラフG 上でノード識別子 i1 からi2へのアークが存在する条件は$i1 \otimes_{DR} i2$ である。

さて，G′ = g (DQ,Q,Qc,Q) ，DQ=DR〔Q,Qc〕

とする。また，（q,qc）∈DQを $q \otimes_{DQ} qc$とすると，グラフ G′ 上でノード識別子 q1 からq2へのアークが存在する条件は$q1 \otimes_{DQ} q2$ である。

〔定理〕上記のグラフ G′ は，関数値の依存グラフであり、次の性質を満たす。

i） $i \otimes_{DR} ic \Rightarrow q \otimes_{DQ} qc$

（ここで q = h (i) ， qc = h (ic) ）

ii） $q \otimes_{DQ} qc \Rightarrow$ h (i) =q を満たすすべてのi に対し，（i,ic,q,qc ）∈DRを満たすicが存在する。

(証明)

i）条件より（i，ic，q′ ，qc′ ）∈DRが存在する。またDRの性質より，h (i) =q ′ ，h (ic) =qc′ 。よって q′ =q，　qc′ =qc。DQはDRの射影であるから，（q,qc）∈DQ である。よって，$q \otimes_{DQ} qc$ 。

ii）（q,qc）∈DQ であるから

（i′ ，ic′ ，q，qc ）∈DR なる i′ ，ic′ が存在する。

a） i= i′ の時，ic=ic′

b） i≠ i′ の時，関数型言語の関数値の依存関係の一意性より，G の性質として同じノード値q を持つ極大部分グラフは同値である。よって，

（i,ic,q,qc ）∈DR が必ず存在する。

上記定理より，G′ はインスタンス依存グラフG＝g（DF′ ,I,Ic,Q ）に対して同じq の値を持つノードを一点に射影してできる準同型グラフとなることが分かる。また，G′ において，r を根とする部分グラフを与える関係DP*（r）は，閉包演算により次のように求められる。

DP* （r）=DQ⇓* （Q =r ｜Qc）　・・ (3-1)

（2）　バグ検出アルゴリズム

PGM 法では，（1）で述べた関係DQのグラフG′ を探索木とし，G′ のノードの値の検査を繰り返すことによりバグを検出する。これに対して前述のShapiro の方式は，DF′ のグラフGを探索木とし，そのグラフのノードの値の

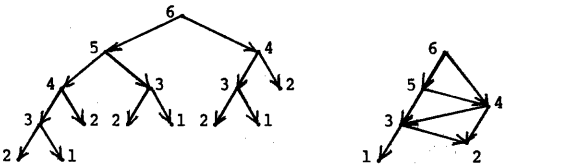


図5．　関係演算による探索木の縮退化の例

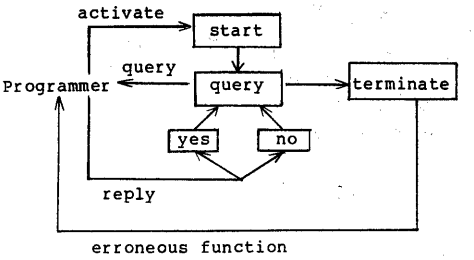


図6．PGM 法のデバッグ手順

検査を繰り返すことに対応する。従って,PGM法は,Shapiroの方式に比べ，次のような利点があるといえる。

① 探索木が小さくなり質問回数が減少する。たとえば，6番目のフィボナッチ数を求める関数fibo（6）の計算により作られるインスタンス依存グラフは図5の（a）から（b）のように射影され，探索木が小さくなる。

② 既に質問した関数入出力関係を値とするノードは，探索木から取り除かれるので，同じ質問が選ばれる可能性がない。

このアルゴリズムは，図6のように start，query,yes,no, terminate の5ステップから成り，プログラマの応答を介して接続されている。最初に，start でデータを初期化する。query では，分割後の探索木が最小になるようなインスタンスq を選択し，q の関数入出力関係をプログラマに提示する。次に，yes,または no で，プログラマの答えに応じて探索木を分割する。プログラマがq は正しいと判定した場合には，探索木においてq の子孫以外のノードの中にバグ発生インスタンスの関数入出力関係を値とするものが必ず存在することが分かる。この結果,yesが起動され，ノードq を根とする部分木を探索木から取り除く。他方，q を正しくないと判定した場合は，q の子孫にバグ発生インスタンスの関数入出力関係を値とするノードが必ず存在することが分かる。この結果,no が起動され，ノードq を根とする部分木を新しい探索木とする。このように分割された探索木に対して，再度query が起動される。query とそれに引き続くyes または　noのステップは，探

索木のノード数d が1になるまで繰り返される。　d=1 になると terminateが起動され，バグのある関数のテキストがバグ発生時の変数の束縛状態と共に提示される。以下に，各ステップの動作の詳細を示す。

start ：　プログラムのデータフロー解析により関係SF，SVを求める。次に，プログラムを実行させ関係DFを求める。式 (3 -1) より，DP* （r ）を求める。ここで，r は，探索木の根を表し，その初期値はプログラムで最初に起動したインスタンスの関数入出力関係である。 queryに行く。

query ：　d ＝｜DP* （r ）｜　を求める。

d ＝1 の時，terminate にいく。

d ＞1 の時，式 (3 -2) で与えられる n，y について ｜y ｜－｜n ｜の絶対値を最小にするq を求める。但し，q は式 (3 -3) を満たすものである。

$$n = DP^*(q) \qquad \cdots\cdots \quad (3\text{-}2)$$

$$y = DP^*(r) - n$$

$$q \in DP^*(r) \qquad (3\text{-}3)$$

q ＝ (f,x,y) とした時，プログラマに

“ f (x) ＝ y”　の真偽を問い合わせる。

プログラマが「真」と判定した場合にはyes に，「偽」と判定した場合にはnoに行く。

yes ：　DQ＝DQ－DP* （q ）　とすることにより，q を根とする部分木を探索木から取り除きquery に行く。

no　：r＝q とし，queryに行く。

terminate ：タビュレーション技法により関係DFを使いながら f (x) を再実行させ関係DVを作成する。SD (r) によりインスタンスr の全変数の束縛状態を求めて表示し，関数f の修正をプログラマに依頼する。

4．　データフロープログラム診断システムの実現

4．1　システムの概要

前章までに述べたデバッグ法の有効性と問題点を明確にするために実験用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，図7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。各プログラムの機能概要は，以下の通りである。

言語処理系　：　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 の基本構造〔18〕の中から，値定義，関数定義，マクロ定義，タプル式，if　式，return　式，値の参照，関数適用，マクロ展開からなるサブセットを取り出したものである。

データフローマシンシミュレータ　：　循環パイプライン型データフロープロセッサの機能をシミュレートする。プロセッサのアーキテクチャは，次の2点を除いて文献〔7〕と同等である。①関数呼出の際にインスタンス名を動的に生成する。②単項演算と二項演算の範囲で構造データ操作を含む任意の演算機能を定義し追加できる。現在，VAX／VMS のTAO-Lisp〔19〕の関数，及び，関数リンケージ，ゲート命令等のフロー制御命令〔7〕を演算機能として使うことができる。

デバッガ　：　2，3章で述べたデバッグ用データに関する関係データの管理機能，およびプログラム診断機能からなる。

上記プログラムはすべて VAX／VMS のTAO-Lisp〔19〕を用いて記述され，現在，データフロープログラムの実行とデバッグの実験が可能である。但し，関係データの管理機能については，2．3節に示したデバッグ用の各データを関係データから直接求める機能のみが現在までに実現され，2．2節に示した関係演算は開発中である。

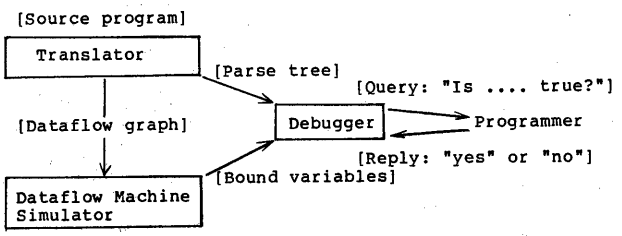


図7．　データフロープログラム診断システム

4．2　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。以下の例で示す図においては，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）階乗の計算

n の階乗を求める関数　fact　のバグを含んだ例として図8（a）のプログラムを考える。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8（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。図のように，プログラマはデバッグの際に yes，no　で答えるだけで良く，トークンをトレースする方式に比べプログラマの負担が大幅に軽減される。

（2）マージソートの計算

図9（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9（b）のようにバグを含むインスタンスが検出される。

（3）集合のunion の計算

図10（a）に示すバグを含んだ union　プログラムの診断過程を図10（b）に示す。

```
(func (fact n)(return result)
 (let result
  (cond
   ((<= n 1) 1)
   (T (+ n (fact (- n 1)))))))
```

(a) Factorial program with a bug

```
=>(?= fact 8)
[env (0 0 0)] (FACT 8)=36
=>pgm
Is (FACT 4)=10 true?  no

Is (FACT 2)=3 true?  no

There is a bug in the following instance!
Modify the function.
[env (0 6 0)] (FACT 2)=3
<< 8 nodes  6 steps ( c = 1.33333^0 ) >>

( FUNC
  (FACT (N = 2))
  (RETURN (RESULT = 3))
  ( LET
    (RESULT = 3)
    (COND
     ((<= (N = 2) 1) (\\ 1 \\))
     (T (+ (N = 2) (FACT (- (N = 2) 1))))) ) )
```

(b) Program diagnosis for FACT

図8．階乗計算プログラムのバグ検出

5．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おける新しいプログラムデバッグ法を提案し，その実現法を明らかにした。この方式は，次のような特徴を持つ。

（1）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すべて関係データとして扱い，検索用の関係演算を与える。デバッグ法は，時系列

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

(a) Megesort program with a bug

```
=>(?= mergesort (8 2 1 6 4 7 5 3))
[env (0 0 0)] (MERGESORT (8 2 1 6 4 7 5 3))
                =(3 6 7 2 5 1 4 8)
=>pgm
Is (MERGESORT (2 6 7 3))=(3 6 7 2) true?  no

Is (MERGESORT (6 3))=(3 6) true?  yes

Is (MERGESORT (2 7))=(7 2) true?  no

Is (MERGE (2) (7))=(7 2) true?  no

Is (MERGE (2) ())=(2) true?  yes

There is a bug in the following instance!
Modify the function.
[env (0 36 0)] (MERGE (2) (7))=(7 2)
<< 26 nodes  12 steps ( c = 2.16667^0 ) >>

( FUNC
  (MERGE (X = (2)) (Y = (7)))
  (RETURN (S = (7 2)))
  ( LET
    (S = (7 2))
    ( COND
      ((NULL (X = (2))) (\\ (Y = *) \\))
      ((NULL (Y = (7))) (\\ (X = *) \\))
      ( (= (CAR (X = (2))) (CAR (Y = (7))))
        (\\ (CONS (CAR (X = *))
                  (MERGE (CDR (X = *)) (Y = *))) \\) )
      (T (CONS (CAR (Y = (7)))
               (MERGE (X = (2)) (CDR (Y = (7)))))))) )
```

(b) Program diagnosis for MERGESORT

図9．マージソートプログラムのバグ検出

データの追跡を基本とする従来の方法に代わり，変数や関数の相互のデータ依存関係を双方向に検索する操作を基本としたものとなる。この結果，プログラマはデータ発生に関する時間的要因と空間的要因を考慮せずににデバッグを進めることが可能となり，並列プログラムのデバッグの難しさが軽減する。

（2）上記デバッグ法を基礎に，実行結果を解析して機械的にバグを検出するプログラム診断機能が実現されている。このプログラム診断では，新たに提案した射影グラフ最小化法と呼ぶバグ検出アルゴリズムに従って，デバッガがプログラマへの質問を繰り返し，その応答を解析することによりバグを検出する。この結果，プログラムの実行履歴が大量に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となる。

本稿では，また，データフローマシンのプログラムデバッグシステムへの本方式の適用実験を行い，方式の有効性を明らかにした。

現在，関数型プログラムの特徴を積極的に利用し，フローグラフの変換操作を基本としたバグ検出アルゴリズムを開発し実験を行っている。この結果については，別途報告する予定である。

```
(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 (3 1 8 2) (5 3 2 9))
[env (0 0 0)] (UNION (3 1 8 2) (5 3 2 9))=(1 8)
=>pgm
Is (UNION (2) (5 3 2 9))=() true?  no

Is (MEMBER 2 (3 2 9))=T true?  yes

Is (UNION () (5 3 2 9))=() true?  no

There is a bug in the following instance!
Modify the function.
[env (0 9 0)] (UNION () (5 3 2 9))=()
<< 5 nodes  2 steps ( c = 2.50000^0 ) >>

( FUNC
  (UNION (U = ()) (V = (5 3 2 9)))
  (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10. 集合のunion の計算プログラムのバグ検出

〔謝辞〕 TAO-lisp システムを開発し，筆者らの実験に御協力頂いた梅村恭司氏に深謝します。また，本稿に関して貴重な御意見を頂いた勝野裕文調査員，後藤厚宏主任，並びに，筑波大学清木康博士に深く感謝します。

参考文献

〔1〕 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.
〔2〕 Dennis, J., "A Preliminary Architecture for a Basic Data Flow Processor," Procs. of 2nd Ann. Symp. on Computer Architecture, pp. 126-132, 1975.
〔3〕 Comte, D., Hifdi, H. and Syre, J. C., "The Data Driven LAU Multiprocessor System : Results and Perspective," Procs. of IFIP 80, pp. 175-180, 1980.
〔4〕 Arvind and Kathail, V., "A Multiple Processor Dataflow Machine that Supports Generalized Procedures," Procs. of the 8th Annu. Symp. on Computer Architecture, pp. 291-302, 1981.
〔5〕 Gurd, J. and Watson, I., "Data Driven System for High Speed Computing," Computer Design,Vol. 9, No. 6& 7, pp. 91-100&97-106, 1980.
〔6〕 Amamiya, M., Hasegawa, R., Nakamura, O. and Mikami, H., "A List-processing-oriented Data Flow Machine Architecture," Procs. of the 1982 NCC, AFIP, pp. 143-151, 1982
〔7〕 Takahashi, N. and Amamiya, M., "A Data Flow Processor Array System : Design and Analysis," Procs of the 10th Ann. Symp. on Computer Architecture, pp. 243-250, 1983.
〔8〕 Shimada, T., Hiraki, K. and Nishida, K., "An Architecture of a Data Flow Machine and its Evaluation," Procs. COMPCON 84, pp. 486-490, 1984.
〔9〕 Keller, R. M., Lindstrom, G. and Patil, S., "A Losely-coupled Applicative Multiprocessing System," Procs. of NCC, AFIPS, pp. 861-870, 1978.
〔10〕 Darlington, J. and Reeve, M., "ALICE : A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages," Procs. of Conf. on Functional Programming and Computer Architecture, ACM, pp. 65-75, 1981.
〔11〕 Johnson, M. S., "A Software Debugging Glossary," SIGPLAN Notices, 17, 2,pp. 53-70, 1982.
〔12〕 Teitleman, W., "Interlisp Reference Manual," Xerox Palo Alto Research Center, 1978.
〔13〕 Pitman, K. M., "The Revised Maclisp Manual," MIT, 1983.
〔14〕 Garcia-Molina, H, Germano, F., Jr. and Kohler, W. H., "Debugging a Distributed Computing System," IEEE Trans. on Software Engineering, Vol. SE-10, No. 2, 1984.
〔15〕 Shapiro, E. Y., "Algorithmic Program Debugging," MIT Press, 1983.
〔16〕 Bird R. S., "Tabulation Techniques for Recursive Programs," ACM Computing Surveys, Vol. 12, No. 4, pp. 403-417, 1980.
〔17〕 上林弥彦，"データベースの基礎理論(4) : 関係データベース言語，"情報処理 Vol. 24, No. 2, pp. 176-187, 1983.
〔18〕 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.
〔19〕 梅村恭司，"移植性の高い TAO-LISP システム，" 第27回情処学会全国大会，6E-1, 1983.