Vol. 27 No. 4 情報処理学会論文誌 Apr. 1986

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†

高橋直久†† 小野 諭†† 雨宮真人††

関数型プログラムは，副作用の排除，記述の簡潔さ，並列処理の自然な記述などプログラム記述上優れた性質を備えている．しかし，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，しかも並列処理環境下では実行制御の流れが多数存在するので，従来のデバッグ法を用いて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す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，トレースやメモリダンプなど従来のデバッグ手法をそのまま使うことはできない．本稿では，このような問題を解決するため，関数型プログラムを並列実行させるシステムでの新しいプログラムデバッグ法を提案し，その実現法について述べる．この方式の特徴は，(1)変数や関数の相互のデータ依存関数を双方向に検索する演算を基本操作とし，プログラマが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デバッグを進められること，(2)上記演算を用い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機械的，かつ効率的に検出できることの2点にある．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する．

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おり，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．このため，新たなデバッグ手法の開発が望まれている．

また，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従来のデバッグ機能[11)]は，逐次処理プログラムの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には適さない．すなわち，従来のデバッグ法では，プログラムテキスト上の式の順序に従ってプログラムが実行されることを基本的に想定しているので，プログラム実行時に発生する時系列データをメモリやスタックを介して監視する機能がデバッグの中心となっている．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，プログラムの並列実行時に発生する時系列データを追跡してデバッグを進めるのは困難である．

以上の問題を解決するため，本稿では，まずデータ依存関係の解析に基づいたプログラムデバッグ法を提案する．この方式では，実行履歴など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，各データの間の相互依存関係を探索する操作をデバッグの基本とする．これにより，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くことが可能となる．次に，依存関係の探索操作を発展させて機械的にバグを検出するプログラム診断システムについて述べる．このシステムは，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同定する．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．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．

† Parallel-Processing-Oriented Algorithmic Bug Location Method by NAOHISA TAKAHASHI, SATOSHI ONO and MAKOTO AMAMIYA (NTT Electrical Communications Laboratories).

†† NTT 電気通信研究所

2. データ依存関係の解析に基づくデバッグ法

2.1 背　景

従来の並列処理システムでは，複数の逐次型プロセスを並列実行させるものが多く提案されている．このため，並列処理プログラムのデバッグ法としても，各プロセスのプログラムデバッグとプロセス間の同期・通信に関するデバッグの組み合わせを中心とした方法が考えられてきた[12]．このようなシステムではプロセスの実行が逐次的であるので，逐次型プログラムのデバッグと同様にトレースやブレークなど時系列データを解析するツールが重要な役割を果たす．しかし，①非同期処理を含むのでトレースが難しい，②実行場所が分散しているので誤りを発見するのが難しい，③通信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，④一般に並列処理ではプログラムの実行環境が複雑でデバッグ時に解析するデータ量が多い，という並列処理プログラムのデバッグに特有な問題が残されている．

一方，関数型プログラムは，副作用の排除，記述の簡潔さ，読みやすさ，などバグの少ないプログラムを作成する上で有効となる性質を備えている．さらに，関数型プログラムでは特別な同期・通信命令を陽に用いずに自然な形で並列処理を記述するので，並列処理を考えた場合でもプログラムに混入するバグは従来に比べ比較的少なくなると期待できる．しかし，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従来のデバッグ法でバグを検出するのは困難になる．また，関数型プログラムを並列実行させる計算機として期待されてい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[1)~8)]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，トークンをモニタしてバグを検出するのは難しい問題である．

2.2 デバッグシステムの構成

前節に示した問題を解決するために時系列データの追跡を基本とする従来のデバッグ法に代わりデータ依存関係の解析に基づくデバッグ法を提案する．この節ではデバッグシステムの構成法と処理の流れを概観し，具体的なデバッグ法については次節で述べる．

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，コードの実行結果として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，データベース (DB) に登録す

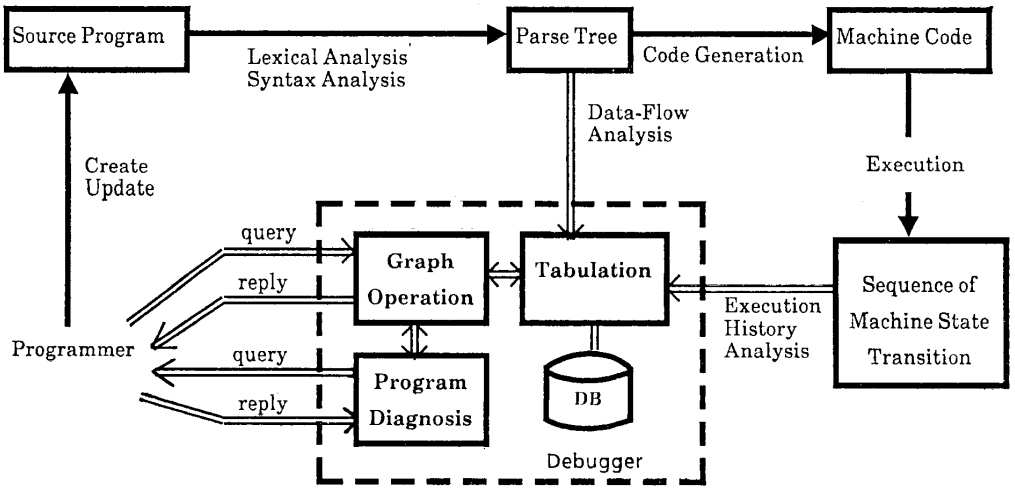


図1 データ依存関係の解析に基づくデバッグシステム
Fig. 1 Program debugging system based on data dependency analysis

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DBに登録する．登録されたデータについて相互依存関係に従ってデータを関連づけることにより実行履歴を表す依存グラフが得られる[13]．プログラマは，これらの依存グラフに関して注目するノードから双方向に探索することによりバグの発生源を検出する．また，バグ検出アルゴリズムに従い，依存グラフの探索・変換を繰り返し行うとバグ発生源を機械的に求められる．デバッグシステムを構成する主な機能は次のとおりである．

（1） タビュレーション機能

以下に示す五つの関係データをデバッグの際の基本データとして表に保持する．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．デバッグを進める上で変数や関数の値が必要になった場合には，表から必要な値を取り出して使用することにより無駄な計算が省かれる．

① 関係 SF (F, F_c)：**関数の静的な依存関係**

$(f, f_c)\in$SF は，関数 f が関数 f_c を呼び出すことを意味する．

② 関係 SV (F, V, V_c)：**変数の静的な依存関係**

$(f, v, v_c)\in$SV は，関数 f において変数 v_c 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, I_c)：**関数の動的な依存関係**

プログラム実行時に関数呼び出し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，$(i, i_c)\in$DD は，インスタンス i において関数呼び出しが発生しインスタンス i_c が生成されたことを意味する．

④ 関係 DI (I, Q)：**インスタンスの実行状態**

$q=(f, x, y)$ の時，$(i, q)\in$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 のパラメータおよび結果であり，$q=(f, x, y)$ を**具現値**と呼ぶ．

⑤ 関係 DV (I, V, W)：**変数の束縛状態**

$(i, v, w)\in$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） 依存グラフ操作機能

本稿では，有向グラフ G をノード集合 N とアーク集合 A およびノードの値を与える関数 value $(n_i)=v_i$ により表す．ここで，v_i はノード n_i の値であり，$(n_i, n_j)\in A$ ならば，ノード n_i からノード n_j に至るアークが存在する．また，関数 node, arc を node$(G)=N$, arc$(G)=A$ と定義する．

表 1 依存グラフの基本操作
Table 1 Primitive operations on dependency graphs.

記　法	機　能
find(G, n)	グラフGのノードnの値を取り出す．
child(G, n)	グラフGでノードnから距離1で到達可能な全ノードを取り出す．
descendent(G, n)	グラフGでノードnから到達可能なノードからなる極大な部分グラフを取り出す．
parent(G, n)	グラフGでノードnへ距離1で到達可能な全ノードを取り出す．
ancestor(G, n)	グラフGでノードnへ到達可能なノードからなる極大な部分グラフを取り出す．

上記関係データを用いて，次のような有向グラフを生成する．

① インスタンス依存グラフ G

node$(G)=\{i\mid(i, q)\in$DI$\}$

arc$(G)=$DD

value$(i)=q$ iff $(i, q)\in$DI

② インスタンス i の変数値依存グラフ V_i

node$(V_i)=\{v\mid(i, v, w)\in$DV$\}$

arc$(V_i)=\{(v_1, v_2)\mid(i, q)\in$DI $\wedge$ $q=(f, x, y)$ $\wedge$ $(f, v, w)\in$SV$\}$

value$(v)=w$ iff $(i, v, w)\in$DV

有向グラフに対する基本操作として，**表1**に示すようにノードの値の取り出し，特定ノードから到達可能なノードからなる部分グラフの取り出し，特定ノードに到達可能なノードからなる部分グラフの取り出しを行う操作を定める．これらの操作を上に示した依存グラフに適用し，その結果を表示すると，**表2**に示すようにデバッグ支援機能が実現される．すなわち，インスタンス依存グラフに各操作を施すと，特定インスタンスの実行状態のモニタ，およびインスタンス依存グラフを双方向に辿る機能が与えられる．また，変数値依存グラフに各操作を施すと特定変数の束縛状態のモニタ，および関数内の各変数の値の相互依存関係を双方向に辿ることができる．表2に示したデバッグ機能名は，文献 11) の定義に従っている．

（3） プログラム診断機能

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機能をプログラム診断機能と呼ぶ．今，具現値が正しい（予期したとおりの

表 2 デバッグ支援機能
Table 2 Primitive functions for debugging functional programs.

依存グラフ	グラフ操作	機能	デバッグ機能名[11]
インスタンス依存グラフ G	find(G, i)	インスタンス i の実行状態を求める.	snapshot dump
	child(G, i)	i が生起したインスタンスの実行状態を求める.	
	descendent(G, i)	i の全子孫のインスタンスの実行状態を i から順に辿る.	subroutine trace
	parent(G, i)	i を生起したインスタンスの実行状態を求める.	
	ancestor(G, i)	プログラムを起動してから i に至る全インスタンスを i から逆向きに辿る.	retrospective trace
変数値依存グラフ V	find(V, v)	変数 v の束縛状態を求める.	variable trace
	child(V, v)	変数 v を右辺に持つ式の値を求める.	stepwise execution
	descendent(V, v)	変数 v の値が影響を与えた変数の束縛状態を v から順に辿る.	execution trace
	parent(V, v)	変数 v の値の計算で用いた変数の値を求める.	
	ancestor(V, v)	変数 v の計算に影響を与えた変数の束縛状態を v から逆向きに辿る.	flowback analysis

結果である）インスタンスは真であり，そうでないインスタンスは偽であるという．この時，関数型プログラムでは副作用がないので，次の二つの条件を同時に満たすインスタンス b がバグを発生させたインスタンスであると断定することができる．

・b が偽である．

・b の子の中に偽のものが存在しない．

バグ発生インスタンス b を求める最も単純な方法は，**図2**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真偽を調べるものである．この図で○は真のインスタンス，●は偽のインスタンス，→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．上のアルゴリズムは，検査すべきインスタンスの数がグラフのノード数に比例して多くなるという問題がある．この問題を解決するアルゴリズムについては3章で述べる．

2.3 デバッグ法

次のように依存グラフを探索することによりバグの発生箇所を特定する．

（1） プログラマ主導型のバグ検出

バグの存在する関数を検出するためにデバッガの応答に基づいてプログラマがインスタンス依存グラフを探索する．今，関数 f のあるインスタンス i のパラメータ x および結果 y がプログラマに与えられたとする．この時，$f(x)=y$ が予期した（正しい）結果であるとプログラマが判断した場合には，インスタンス

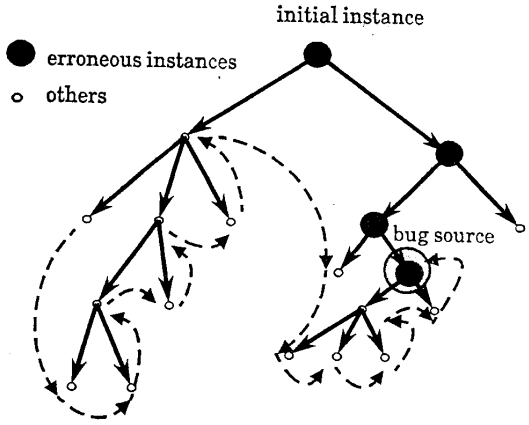
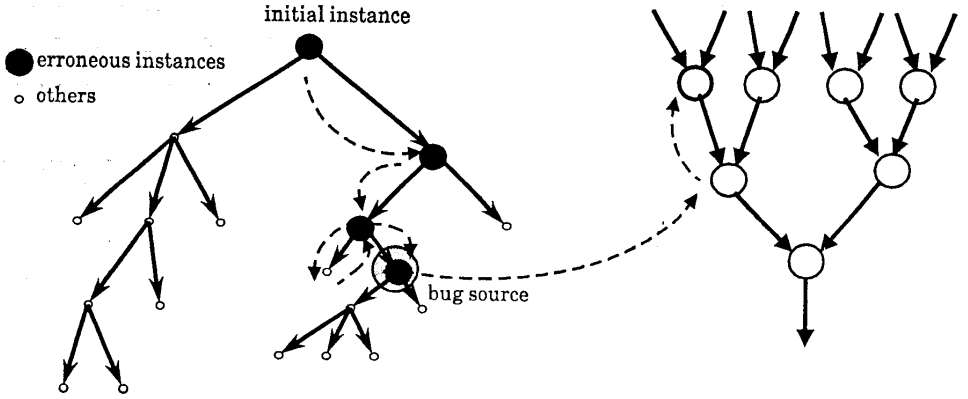


図 2 単純なバグ検出アルゴリズム
Fig. 2 Simple bug location algorithm.

依存グラフをノード i から逆方向に辿るとバグのある関数のインスタンスに到達する．反対に，$f(x)=y$ が予期しない（正しくない）結果であると判断した場合には，インスタンス依存グラフをノード i から順方向に辿るとバグを探し出せる．したがって，**図3**（a）の矢印で示すようにインスタンス依存グラフを双方向に探索することによりバグを発生した際のインスタンスを求められる．同様にして，図3（b）のように，このインスタンスの変数値依存グラフを双方向に検索するとバグのある式を求められる．

（2） デバッガ主導型のバグ検出

プログラム診断機能を用いたデバッガでは，デバッガが実行履歴を解析してプログラマに質問を発する．そして，その応答を解析してバグの探索空間を狭め



(a) Instance dependency graph. (b) Value dependency graph.

図 3 依存グラフの探索によるバグ検出
Fig. 3 Bug location by traversing dependency graphs.

る．この一連の操作を繰り返すことにより，デバッガはバグ発生インスタンスを同定する．したがって，プログラマは，デバッガの質問に繰り返し答えることにより，バグのある関数の名前，およびバグを発生させた時に与えられたパラメータの値を知る．さらに，そのインスタンスの変数値依存グラフを調べることにより，誤りのある式を検出する．

依存グラフの解析に基づくデバッグ法の特徴をまとめると次のようになる．

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の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を容易にする．

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 タビュレーション手法[14]をデバッグに適用することにより，トレース，バックトレース[11]に必要な再計算量とメモリ量を減少させることができる．一例として，関数の内部のバックトレースを考える．この場合，任意のインスタンスのバックトレースを許す場合にはすべてのインスタンスの変数値依存グラフが必要となる．今，インスタンスの数を m，1インスタンス当たりの変数の数の平均値を n とする．この時，関数内部のバックトレースのために必要な変数値の個数は，タビュレーション機能を使わずに単純にすべての変数値を保持する場合には $m \times n$ のオーダになる．これに対してインスタンス依存グラフだけを保持しておき，必要なインスタンスをタビュレーション機能を使いながら再計算して変数値依存グラフを求めると，保持すべき変数値の個数は $m+n$ のオーダに減少する．

⑤ 依存グラフでバグが伝播する範囲を解析することにより機械的にインスタンスを探索してバグを検出するアルゴリズムを与えることができる．これにより大量にデバッグ用データが生成された場合でも機械的かつ効率的にバグを検出できる．

3. 射影グラフ最小化法によるバグ検出

3.1 プログラム診断システム

2.2 節で示したプログラム診断機能の実現例ではインスタンス依存グラフの各ノードを一つずつ検査していくので，実行結果が大量に生成された場合には質問回数が極めて多くなるという問題がある．すなわち，インスタンスの数を m とすると，この方法では質問回数が 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，という二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．彼のアルゴリズムでは，インスタンス依存グラフについてノード数が全体のほぼ半

分になる部分木を選び，その根のインスタンスの真偽を質問する．真と答えた場合には，その部分木を取り除いた残りの木にバグ発生インスタンスが存在する．偽と答えた場合には，その部分木の中にバグ発生インスタンスが存在する．いずれの場合にも1回の質問応答でバグの存在範囲がほぼ1/2に狭められる．したがって，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．

・ 探索木に同じ具現値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．

・ すでに出した質問で使用した具現値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．

本章で示す射影グラフ最小化 (PGM: Projected Graph Minimization) 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 P を探索木とし，P 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．

3.2 射影グラフ

今，h を

$h(i)=q$ iff $(i, q)\in$DI,

と定義すると，任意の (i_0, q_0), $(i_1, q_1)\in$DI に対して $q_0\neq q_1$ ならば $i_0\neq i_1$ であるので，h はインスタンス集合 I から具現値集合 Q への関数である．この関数 h は，具現値 q の値が等しいすべてのインスタンスを1点に射影するものである．グラフ G を関数 h で射影することによりできるグラフ P を射影グラフと呼び，次のように定義する．

$$\mathrm{node}(P)=\{q \mid q=h(i) \wedge i\in\mathrm{node}(G)\}$$

$$\mathrm{arc}(P)=\{(q_1, q_2) \mid q_1=h(i_1) \wedge q_2=h(i_2) \wedge (i_1, i_2)\in\mathrm{arc}(G)\}$$

$$\mathrm{value}(q)=q$$

ここで，$|S|$ を集合 S の要素数とすると，上の定義より直ちに次のことがいえる．

〔性質1〕 $|\mathrm{node}(P)|\leqq|\mathrm{node}(G)|$

〔性質2〕 $q_1\neq q_2$ ならば，$\mathrm{value}(q_1)\neq\mathrm{value}(q_2)$

〔性質3〕 $(i_1, i_2)\in\mathrm{arc}(G)$ かつ $q_1=h(i_1)$ かつ $q_2=h(i_2)$ ならば，$(q_1, q_2)\in\mathrm{arc}(P)$ である．

さらに，副作用のない関数型プログラムでは，同じ関数に同一パラメータを与えた場合には等しい結果が返され，かつ，その実行履歴は等しくなるので，次のことがいえる．

〔性質4〕 $(q_1, q_2)\in\mathrm{arc}(P)$ かつ $h(i_1)=q_1$ ならば，$h(i_2)=q_2$ かつ $(i_1, i_2)\in\mathrm{arc}(G)$ なる i_2 が存在する．

上記〔性質3〕および〔性質4〕より，グラフ P はグラフ G の準同型グラフであることが分かり，表1に示した操作を用いると次のことがいえる．

〔性質5〕 $i_1\in\mathrm{node}(\mathrm{descendent}(G, i))$ かつ $h(i)=q$ ならば，$h(i_1)\in\mathrm{node}(\mathrm{descendent}(P, q))$ である．

上の性質から，バグ発生インスタンスの具現値の存在箇所に関して次のことがいえる．

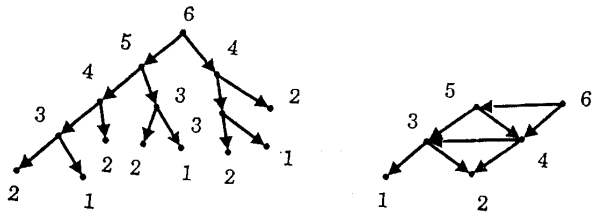
〔性質6〕 グラフ G で i を根とする部分木にバグ発生インスタンス i_1 が含まれる場合，かつ，この場合にのみ，グラフ P で $q=h(i)$ なるノード q から到達可能な極大な部分グラフにバグ発生インスタンスの具現値が存在する．

3.3 射影グラフ最小化法

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．プログラムが無限ループを含み停止しないような場合には，実行時にタイムアウトあるいはプログラマの指示により実行を中断させ，中断時の実行履歴に基づいてバグを検出することができる．しかし，議論を簡単にするため，以下では，プログラムが停止し得られた結果に誤りがある場合のバグ検出の問題について述べる．

PGM 法では，前節で述べた射影グラフ P を探索木とし，P 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．ここで，グラフ P においてバグ発生インスタンスの具現値の存在箇所に関して〔性質6〕が成立するので，P の中のあるノードの具現値の真偽が判明するとバグ発生インスタンスの具現値を必ず含むような部分グラフを P から取り出すことができる．したがって，Shapiro のアルゴリズムと同様に divide-and-conquer の考え方を適用して，P の2分割を繰り返すことにより P をバグ発生インスタンスの具現値に縮約できる．さらに，〔性質1〕，〔性質2〕より，PGM 法は，Shapiro の方式に比べ，次のような利点が得られる．

① 探索空間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



(a) Instance dependency graph. (b) Projected graph.

図 4 射影グラフ（フィボナッチ数列の計算）
Fig. 4 Projected graph (calculating Fibonacci numbers).

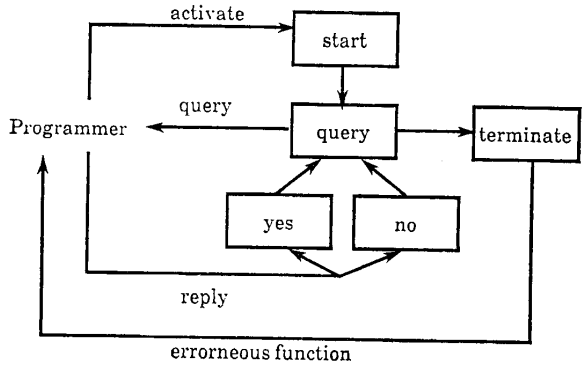


図 5 射影グラフ最小化法の構成
Fig. 5 Projected graph minimization method.

(6)の計算により作られるインスタンス依存グラフは**図4**の(a)から(b)のように射影され，探索空間が小さくなる．

② 具現値を一度質問に使用すると，その具現値は探索空間から取り除かれるので，同じ質問が出されることがない．

このアルゴリズムは，**図5**のように **start, query, yes, no, terminate** の5ステップから成り，プログラマが **start** を起動するとプログラム診断を開始する．各ステップの動作は次のとおりである．

start: プログラムのデータフロー解析により関係SVを求める．次に，プログラムを実行させ関係 DD, DI を求め，これにより射影グラフ P を生成する．この時，プログラムの実行で最初に起動した関数の具現値を r とする．次に，**query** に行く．

query: $d=|P|$ を求める．

$d=1$ の時，**terminate** に行く．

$d>1$ の時，式 (3.1), (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ような $q\in$ node(P) を求める．

$$n=\text{node}(\text{descendent}(P,q)) \quad (3.1)$$

$$y=\text{node}(P)-n \quad (3.2)$$

$q=(f, x, y)$ 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: グラフ P において，[$S=$descendent(P,q) なる部分グラフ S を求める．グラフ P から S に含まれるすべてのノードを取り除き **query** に行く．

no: descendent(P, q) を新たに P とし，r を q に変更して，**query** に行く．

terminate: 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．関係 DV, DI, SV よりインスタンス r の変数値依存グラフを求めて表示し，関数 f の修正をプログラマに依頼することにより，プログラム診断を終了する．

4. データフロープログラムデバッグシステムの実現

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6**のように，言語処理系，データフローマシンシミュレータ，デバッガの三つのプログラムからなる．各プログラムの機能概要は，以下のとおりである．

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，記述言語のセマンティクスはデータフローマシン用関数型言語 Valid の基本構造[16]に基づいている．

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．発火制御機構は，文献 7) と同等であるが，演算処理機能については次のように拡張している．①関数

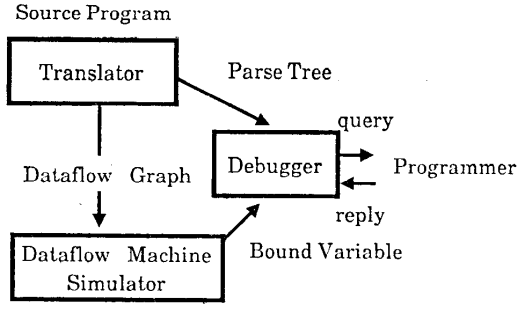


図 6 データフロープログラムデバッグシステム
Fig. 6 Experimental debugger for dataflow programs.

```
(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)]
```

図 7 例題プログラム（マージソート）
Fig. 7 Sample program (mergesort).

呼び出し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．

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 Lisp で記述され，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．

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．ここでは，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，図7に示すように三つの関数 split, merge, mergesort からなり，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．図のプログラムにはバグがあり，mergesort に入力データ (8 2 1 6 4 7 5 3) を与えると，ソートされていない誤った結果 (3 6 7 2 5 1 4 8) が返される．この時，プログラマがデバッガ PGM を起動して図8のようにデバッガと質問・応答を繰り返すと，関数 merge にパラメータ（2），（7）を与えた時にバグが発生したことが判明する．図で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．さらに，バグ発生時の関数 merge の変数値依存グラフを使って変数の束縛状態が表示されるので，プログラマはバグ発生時に実行された式とその値を知ることができる．ここで，斜線で示した部分は，バグ発生時には実行されなかった式を意味する．このようにプログラマが yes, no で答えるだけでバグのある関数が与えられるので，トークンをトレースする方式に比べ，バグ検出に関するプログラマの負担が大幅に軽減される．

```
=>(?= mergsort (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)>>

(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)))))))))
```

図 8 例題プログラムのバグ検出
Fig. 8 Bug location for sample program.

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，検索用のグラフ操作命令を与える．デバッグ法は，時系列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を容易にする．

（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，関数依存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．

謝辞 本研究の機会を与えられ，ご指導いただいた畔柳功芳情報通信基礎研究部長に深く感謝します．Lisp 処理系を開発し，筆者らの実験にご協力いただいた梅村恭司主任に深謝します．また，本稿に関して貴重なご意見をいただいた勝野裕文主任研究員，後藤厚宏氏，並びに，日頃ご指導，ご討論いただく情二室諸氏に深く感謝します．

参 考 文 献

1) Treleaven, P., Brownbridge, D. R. and Hopkins, R. P.: Data-Driven and Demand-Driven Computer Architecture, *ACM Comput. Surv.*, Vol. 14, No. 1, pp. 94-143 (March 1982).
2) Dennis, J.: A Preliminary Architecture for a Basic Data Flow Processor, Proc. 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, Proc. of IFIP 80, pp. 175-180 (1980).
4) Arvind and Kathail, V.: A Multiple Processor Dataflow Machine That Supports Generalized Procedures, Proc. of the 8th Ann. Symp. on Computer Architecture, pp. 291-302 (1981).
5) Gurd, J. and Watson, I.: Data Driven System for High Speed Computing, *Comput. Des.*, 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, Proc. of the 1982 NCC, AFIP, pp. 143-151 (1982).
7) Takahashi, N. and Amamiya, M.: A Data Flow Processor Array System: Design and Analysis, Proc. 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, Proc. COMPCON 84, pp. 486-490 (1984).
9) Keller, R. M., Lindstrom, G. and Patil, S.: A Losely-coupled Applicative Multiprocessing System, Proc. 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, Proc. of Conf. on Functional Programming and Computer Architecture, ACM, pp. 65-75 (1981).
11) Johnson, M. S.: A Software Debugging Glossary, *SIGPLAN Notices*, Vol. 17, No. 2, pp. 53-70 (1982).
12) Garcia-Molina, H., Germano, F., Jr. and Kohler, W. H.: Debugging a Distributed Computing System, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 2, pp. 210-219 (1984).
13) 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).
14) Bird, R. S.: Tabulation Techniques for Recursive Programs, *ACM Comput. Surv.*, Vol. 12, No. 4, pp. 403-417 (1980).
15) Shapiro, E. Y.: *Algorithmic Program Debugging*, MIT Press, Cambridge (1983).
16) 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, pp. 793-802 (1984).

（昭和60年 7 月 12 日受付）
（昭和60年12月 21 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．工学博士．昭和57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．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．