解　説

関数型プログラムの作成支援システム†

高　橋　直　久††

1. はじめに

関数型プログラムは，関数の定義の集まりとして記述され，代入などの副作用がなく，参照の透明性をもっている[1),2)]．このような性質は，理解しやすいプログラムの作成，作成者の意図に合った正しいプログラムの作成，あるいは，作成したプログラムが正しいことの検証，誤りの検出，などについて，従来の手続き型プログラムに比べ，プログラマをより強力に支援できる可能性をもたらす．

プログラム作成支援技術について関数型プログラムに係わる話題はきわめて多岐にわたっているが，本稿では，関数型プログラムの実行時に現れる誤りの検出に焦点をあて，その関連技術を紹介したい．したがって，デバッグしやすいプログラム，あるいは，デバッグ手法などの問題を取り上げることになる．

本解説では，まず，2. で，以下の議論の見通しをよくするために，手続き的アプローチ，および，非手続き的アプローチと呼ぶ二つのデバッグのモデルを設定し，必要な支援技術を概観する．3. では，両モデルに関連する技術として，デバッグしやすいプログラムという観点から，プログラム構造の理解を助ける表現法について述べる．4. では，手続き的アプローチにおいてデバッグ情報を収集する手法，いわゆるデバッギングツールについて述べる．5. では，支援系がバグ検出手順を考えて情報の収集・解析を行い，プログラマはプログラムに込めた自分の意図を宣言的に支援系に教える，非手続き的なアプローチについて紹介する．

2. デバッグのモデルと支援技術

本章では，コンパイル時には検出されず実行時に初めて表面化するようなバグについて，その検出と修正の手順を考え，これらの手順を効率的に行うために必要な支援技術を概観する．

2.1 デバッグのモデル

（1） 手続き的なアプローチ

デバッグ時にプログラマが通常よく行う，プログラムの実行状態の変化を調べてバグを修正する仕事は，次のような手続きとしてモデル化される[3)]（図-1(a)参照）．プログラマは，まず，プログラムを実行させて，関数や変数の値の変化や制御の流れなどを観測して情報収集する．次に，収集した情報を解析し，いつ，どのような場合に誤りが生じているか調べあげる．さらに，誤りの原因がどこにあるか推定し，仮説をたてる．再び情報収集・解析を行い，仮説が正しい，あるいは，誤っていることを検証する．このような手順を繰り返して，正しい仮説が得られると，プログラムを修正する．そのプログラムを再度実行させ，誤りが現れないことを確認して，デバッグを終える．

（2） 非手続き的なアプローチ

上記の手続きに従い，バグの箇所を予測し，検証する仕事は，豊富な経験を必要とし，さらに，多くの試行錯誤をともなう．このような仕事からプログラマを解放することをねらい，プログラムの解析手順をデバッグシステムが与えるような試みがなされている[4)-10)]．この場合，ユーザはバグの検出手順を考えるというよりも，システムが行う解析を補助するために，プログラムで実行しようとした自分の意図を宣言的に教えることになるので，このようなアプローチを非手続き的なデバッグ手法と呼ぶことにする（図-1(b)参照）．

2.2 支援技術

一般には，エディタ，コンパイラ，デバッガなど多数のツールを含む，いわゆるプログラミング環境[12)]すべてが，デバッグの支援技術に関連するが，本稿では，図-1 のモデルに従い次の 3 点について議論する．

（1） デバッグ情報の解析支援

デバッグ時の情報解析では，得られた情報とプログラムとを対応づけて，実行状態を正しく，かつ，素早く，把握することが重要になる．プログラムの構造の

† Support Systems for Functional Programming by Naohisa TAKAHASHI (NTT Software Laboratories).
†† NTT ソフトウェア研究所

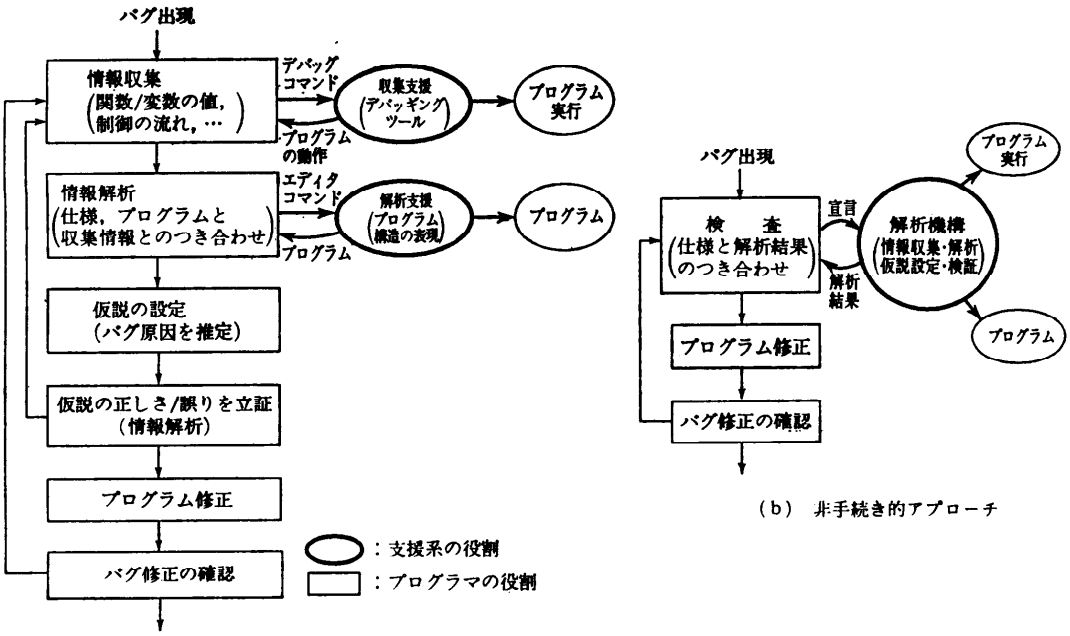


図-1 デバッグのモデル

理解が容易で，実行の様子を把握することが容易な，プログラムの表現法を与えることが基本的要件となる．

関数型プログラムの参照の透明性[1]は，このような表現法を実現する上で望ましい性質である．さらに，関数的な性質を基礎にした，図式表現法[14)~18)]や階層的な表現法[19),20)]は，この性質から得られる効果をさらに高めるものである．

(2) デバッグ情報の収集支援

手続き的なアプローチで情報を収集する場合に，トレーサ，ステッパなどのデバッギングツール[11),13)]が，基本的な道具立てとなる．関数型言語では，効率的実行，並列実行などのため，従来の手続き型プログラムとは大きく異なった実行方式を採る場合がある．この場合には，実行方式に応じて，新たな考え方に基づくツールが必要になるであろう[8),21)]．また，一般に，デバッギングツールでは，プログラムを実行させて，副作用を観測する．関数型言語でデバッギングツールを記述する場合には，副作用を用いない手法[21)~23)]を与えることが必要になる．

(3) 非手続き的なデバッグにおける解析機構

プログラマがデバッグの手続きを考えなくても良いようにするためには，システムがデバッグ情報の収集，解析，および，バッグの探索範囲の絞り込み操作を行う必要がある．たとえば，システムがバグの位置を推定し，必要な情報の収集・解析を自動的に行う機構[4)~10)]が要求される．

副作用のない非手続き的なプログラムは，計算の構造（セマンティクス）が簡明なため，プログラムの状態変化の機械的な解析が容易となる．さらに，バグの伝播が明解であり，実行履歴を機械的に解析しバグの存在する範囲を絞り込むことを強力に行えるので，上のような機構の実現に適している．

以下の各章では，上記各項目に関連した技法，すなわち，プログラム構造の表現法，手続き的なアプローチでのデバッギングツール，非手続き的なデバッグ手法についてそれぞれ紹介する．

3. プログラム構造の表現法

関数型プログラムの構造を理解し，実行状態を把握する上で有効な表現法として，図式表現と階層的表現を取り上げ，それらの手法を紹介する．

3.1 図式表現

プログラムの図式表現は，手続き型言語でも広く行われている．たとえば，フローチャートもその一種である．また，プログラムの構造を分かりやすく表現す

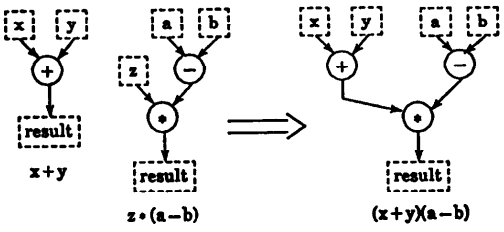
る図式も各種提案されている[24]．これらの図式は，ループの構造，サブルーチンの呼応関係，分岐の状態などの把握を容易にする．一方，関数適用に基づいてプログラムを図式表現する，データフロー図式では，これに加えて，次のような利点も期待できる[14]．

（1） データ依存関係の記述と並列性の表現

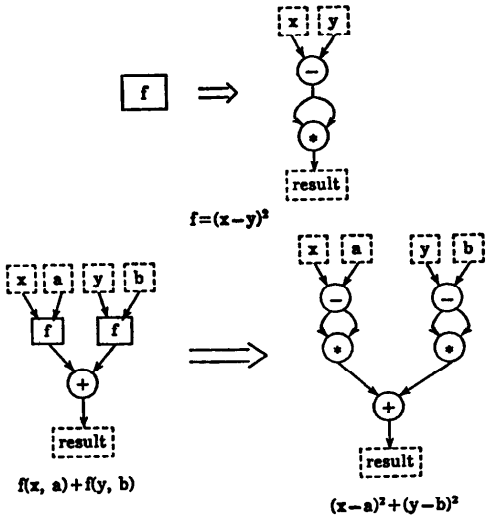
プログラムは，関数を表すノード，および，関数間でのデータの参照関係を表すアークからなる有向グラフで記述される．この結果，実行順序を意識せずに，単にデータの依存関係を考えればプログラムが理解でき，プログラムの構造の把握が容易になる．また，アークの接続関係から，ノード間で実行を独立（並列）に行えるか直感的に把握できる．

（2） 図式の合成・展開

二つのグラフを結合する，あるいは，グラフ中のノードを別のグラフで置き換える操作を行っても，操作対象の二つのグラフが互いに副作用を与えないので，それぞれ元のグラフの意味や性質が損なわれない．したがって，このような操作により，プログラムの合成・展開が容易に行える．図-2 に合成・展開の例を示す．



（a） グラフの合成



（b） グラフの展開

図-2 データフローグラフの合成・展開

（3） 形式的記述

グラフが表す計算の意味が形式的に定められているので，プログラムとしてそのまま実行することができる．あるいは，プログラムの表記と実行との対応がとりやすい．このため，たとえば，グラフを使ってハンドシミュレーションを行い，バグの検出，並列度の解析，性能のボトルネックの検出などを効率的に進めることができる．

並列計算モデル，中間言語，高級言語など目的に応じて上記の特性を具現化したデータフロー図式言語システムが開発されている[14)~18)]．たとえば，FGL[15]，GPL[16] は，良好なユーザインタフェースを与える高級言語として設計されたシステムである．D³L[17] は，中間言語として開発され，問題適応の高級言語[25]を上位の図式言語として想定している．D³L では，非決定性を回避可能な図式表現により履歴依存処理を行う方式を提示している[17]．また，このほかに，実行モデル，あるいは，抽象的なレベルでデータフローマシンの機械語を表す表記法として，データフロー図式表現が広く用いられている[26)~28)]．さらに，ラムダ計算[29]の図式表現を与え，関数型言語の視覚的なプログラミング環境の開発が試みられている[18]．

3.2 階層的表現

一般に，関数型言語を用いたプログラミングでは，より基本的な機能（関数）を簡単な操作で組み合わせることにより，新たな機能（関数）が作り上げられる．この意味で，関数型言語は階層的な機能表現に適したツールである．また，データフロー図式でのグラフの展開は，階層的なプログラム作成の基本概念である．属性文法型プログラミング言語 AG のモジュールとモジュール分割の概念[19]は，さらに積極的にプログラマを階層的な構成法に導くプログラミングスタイルを提示している．以下に，AG における階層的プログラミングを紹介する．

AG は，属性文法に基づいた，rule-based な純関数的な言語である[20]．プログラムは，階層的にモジュー

```
X(↓a, ↑b) = Y(↓c, ↑d), Z(↓e, ↑f)
            when P(a)
            with c = F(a)
                 e = G(a, d)
                 b = H(f)
```

図-3 プログラムの階層的表現[20]

ルと呼ぶ処理単位に分割される．モジュールの内容はその入出力関係により規定される．たとえば，図-3 のプログラム例[20]では，モジュール X がモジュール Y, Z に分割されることを表している．図において，↓，↑ はそれぞれモジュールの入力と出力を表し，F, G, H は，基本関数，あるいは，すでに定義されているモジュールの名前であり，関数として使用されている．when 句はモジュール分割を行う条件を与え，with 句は，モジュールの特定の属性（入力あるいは出力データ）を他のモジュールの属性を用いて関数的に定義する式を与えている．

親モジュールを作る際には，子モジュールは，入力と出力だけを考えたブラックボックスである．親モジュールの分割が終わると，はじめて，子モジュールの内容が記述される．子モジュールの記述では，親モジュールと同様に下位モジュールに分割される．このような分割を繰り返し，基本関数やすでに定義されたモジュールで表せるモジュールに到達するとプログラムができあがる．

4. デバッギングツール

デバッグに必要な情報を収集するツールについて，基本機能，および，関数型言語に特有な問題とその解決へのアプローチを紹介する．

4.1 Lisp におけるデバッギングツール

Lisp は，関数型言語として出発したが，その後，言語仕様が従来の手続き的言語にかなり近づいている．Lisp のプログラミング環境として蓄積されてきた豊富なツール類[13]は，手続き型言語としての側面を強く感じさせるものも多いが，関数や変数の値，関数の呼応関係，制御の流れなどを調べる基本的な道具立てともなる．以下に，主な機能を簡単に紹介する．

（1） デ バ ッ ガ

Lisp では，実行時の制御情報はフレームと呼ぶ単位で次々にコントロールスタックに積まれる．デバッガは，実行が中断された後に，コントロールスタック上のフレームを辿りながら，実行された式，および，実行時の環境を調べるツールである．

（2） ト レ ー サ

トレーサは，プログラムを実行させながら，指定された関数について，呼び出し時に入力パラメータの値を，実行終了時に結果の値をそれぞれ表示するツールである．図-4 に VAX-Lisp でのトレース例を示す．この例からも分かるように，関数の呼応関係を把握できるように表示上の工夫がなされている．

```
Lisp>(defun fact (n)(if (= n 0) 1 (* n (fact (1-n)))))
FACT
Lisp>(trace fact)
(FACT)
Lisp>(fact 4)

#21: (FACT 4)
. #28: (FACT 3)
. . #35: (FACT 2)
. . . #42: (FACT 1)
. . . . #49: (FACT 0)
. . . . #49⇒ 1
. . . #42⇒ 1
. . #35⇒ 2
. #28⇒ 6
#21⇒ 24
24
Lisp>
```

図-4 トレース機能の実行例（階乗の計算）

（3） ス テ ッ パ

ステッパでは，各式を評価する前に，どのように実行を進めるかプログラマから指示を受ける．この結果，1ステップずつ実行する，あるいは，通常に実行するなどの制御を行い，バグに関係がありそうな部分に焦点を絞って詳しく追いかけることができる．

4.2 関数型言語の計算機構とデバッギングツール

関数型プログラムの実行方式として，データフロー方式，および，コンビネータを用いた方式を取り上げ，デバッグに関する問題点，ならびにデバッギングツールの実現例を紹介する．

4.2.1 データフロー型計算とデバッギングツール

データフローマシンは，関数型プログラムを並列実行する計算機として期待され，すでにいくつかの実験システムが稼働している[28]．データフローマシンでは，プログラムカウンタ，スタック，メモリといった逐次型計算機で基本となっている概念がないので，デバッグについて新たなアプローチが要求される．

一方，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[28]ので，トークンを観測するとプログラムの実行経過を把握できる．データフロー図式で表したプログラムの上にトークンを表示するシステム[30]は，このような場合にプログラムの実行状態を把握する上で有効な支援系となろう．

また，データフロープログラムでは，データの依存関係の正しさを調べることが重要であるので，変数の束縛状態と相互依存関係，あるいは，関数の値と相互依存関係などを関係データの形で保持し，これらのデータから構成される依存グラフを双方向に辿るツー

ルが開発されている[8]．データフローマシン用高級言語 ID のプログラム開発環境 ID World でも，関数の値を依存関係に従って順次調べていくコマンドが用意されている[31]．

4.2.2 コンビネータを用いた計算機構とデバッギングツール

通常の Lisp 処理系のように，環境（変数とその値の対）を用いて計算を進める方式では，高階関数や遅延評価の扱いが複雑になる．これに対して，コンビネータ論理[32]に基づく方法で関数型プログラムの計算を行う方式は，項の簡約時に変数を取り扱う必要がないので，環境も不要であり，高階関数や，遅延評価の取り扱いが容易になる[33]．さらに，少数のコンビネータ（結合子）で関数を表現できる，コンビネータを使うと式の簡約化を簡単な機械的操作で実現できる，といった利点がある．このため，関数型プログラムの言語処理系の実現法として，コンビネータを用いた方式が注目され，システムが実現されている[33)~35]．

コンビネータを用いた計算系の問題点として，デバッグの難しさがしばしば指摘されている[33)~35]．すなわち，コードが実行時に簡約のため変形していくので，誤りが生じたときに，コードと元のプログラムとの対応が分かりにくいという問題がある．また，関数へ渡される引数が値でなく，未評価の式を表すグラフであるので，Lisp のトレーサのように引数や結果を値の形で分かりやすく表示することが難しい．

関数型言語 pfp では，関数のトレーサ，および，コンビネータに変換されたコードを扱うデバッガが提供されている[35]．トレーサは，指定された関数の評価時に，関数名とポインタ形式の引数を表示し，計算の流れを教える．また，デバッガは，ポインタが指すグラフを図式的に表示して，グラフの内部表現を知らなくても簡約化の過程を容易に把握できるようにしている[36]．

4.3 関数型言語によるデバッギングツールの記述

デバッグの対象となるプログラムと同じ言語（関数型言語）でデバッギングツールを記述可能な場合には，次のような利点が期待できる[21),22]．

①プログラマにデバッグのための特別な“モード”を意識させない．

②プログラマが新たな関数を定義することにより，自分に必要なデバッグ機能を容易に与えられる．

③インタプリタや専用マシンなどの実行系に対して，デバッグのために特別な変更を加える必要がない．

一方，従来のデバッギングツールは，実行時の状態変化を観測して出力するもので，基本的に副作用を利用している．さらに，デバッグ時のプログラムの修正は，関数を書き換えることを意味する．

関数型言語 Daisy のプログラミングシステムでは，次のようにして上記問題の解決を図っている[22]．

（a） 同じ関数に対して複数の定義を許し，それぞれバージョンを付けて管理する．これにより，一つの関数について，デバッグ用の関数，修正した関数などを切り替えて使うことができる．

（b） ソースプログラムをデバッグ用プログラムに変換する．変換後のプログラムでは，関数の結果だけでなく，関数と変数の参照関係などのデバッグ情報も引数として関数間を伝播させる．

この手法では，デバッグシステムを実行方式と独立に関数的に実現できるという利点があるが，デバッグ用のプログラムが元のプログラムに比べて大きくなる，デバッグ機能ごとにプログラムを変換しなければならないのでデバッグの柔軟性に欠けるという問題がある．これらの問題に着目し，ユーザとの対話機能をもった特別なインタプリタとしてデバッガを実現する方式も提案されている[23]．

データフロープログラムについても，上記①～③と同様な目標の実現に向けたデバッグ法が議論され，データフローマシン用高級言語 ID のマネージャ[27]としてデバッガを構成するモデルが提案されている[21]．

5. 非手続き的なデバッグ手法

関数型プログラムのデバッグを非手続き的に進める手法について，基本的な考え方を具体例により紹介する．

5.1 アルゴリズムによるバグ検出

プログラムの実行時の振舞いを表すために，インスタンス依存グラフ[8]を考える．このグラフのノードは，関数のインスタンスを表し，具現値（関数 f，入力パラメータの並び x，結果 y の三つ組み (f, x, y)）を

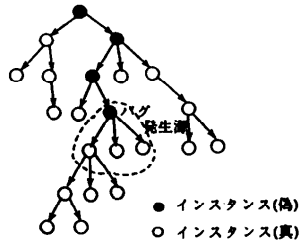


図-5 バグ発生源のパターン

値とする．アークはインスタンス相互の呼応関係を表す．このとき，y が式 $f(x)$ の結果として期待した値である場合に，インスタンスは真であるといい，そうでない場合には偽であるという．

インスタンス依存グラフにおいて，自分が偽で，子供がすべて真であるインスタンス（図-5 参照）を考え，その具現値を (g, a, b) とする．副作用のない関数的なプログラムでは，誤りが局所化され，関数が正しいパラメータに対し誤った答えを返すという形でバグが表面化する．このため，関数 g にバグがあり，かつ，$g(a)$ の計算でバグが現れて以後の計算に伝播したといえるので，このインスタンスをバグ発生源と呼ぶ．実行履歴から探索空間を作りだし，以下の一連の操作を機械的に繰り返して，図-5 のパターンを得るまで探索空間を絞り込めば，バグ発生源が検出される（図-6 参照）．

①探索空間からインスタンスを選択する．

②プログラマに選択したインスタンスを提示し，その真偽を答えさせる．

③バグの探索空間から，図-5 のパターンを少なくとも一つは含むことが保証される，できるだけ小さな領域を切り出して，新たな探索空間とする．

バグ発生源を効率よく見つけるためには，プログラマへの質問の順序と探索空間の絞り込みの方法が重要である．両者を定めたアルゴリズムを診断アルゴリズム，あるいは，バグ検出アルゴリズムと呼ぶ．

Shapiro が提案した副作用のない Prolog プログラムの診断アルゴリズム（Divide-and-Query 法）[4]では，分割統治法に基づいてインスタンスを選択し，質問を生成している．インスタンス依存グラフを探索空間として，ノード数が全体のほぼ半分になる部分木を選び，その部分木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，部分木を取り除き，偽と答えた場合には，部分木だけを残す．いずれの場合にも1回の質問応答でノード数がほぼ 1/2 になる．インスタンス依存グラフのノード数を m とすると，$\log m$ のオーダの質問応答でバグの発生源が検出される．

5.2 戦略的なバグ検出

戦略的射影グラフ最小化法[9]では，Shapiro の診断アルゴリズムを基礎に，関数型プログラムのバグ検出法として，次のような視点を加えている（図-7 参照）．

（1） プログラム構造の静的解析

デバッグ対象のバグはプログラムの中に記述時に埋め込まれたものである．このため，ソースプログラム

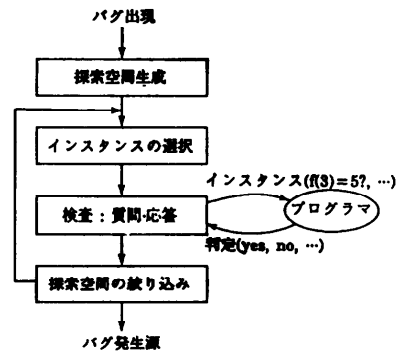


図-6 アルゴリズムに従ったバグ検出

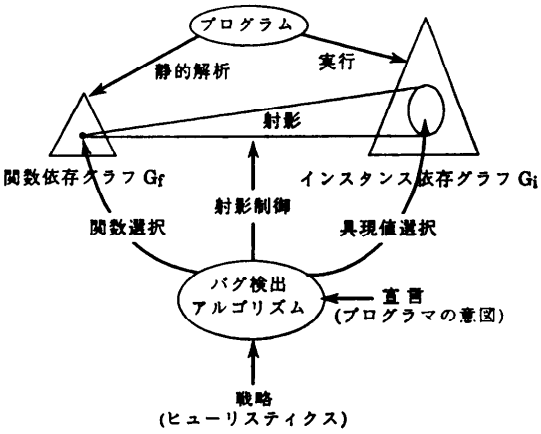


図-7 戦略的なバグ検出

の静的な解析結果は，デバッグ時にバグの存在箇所を推定するために有効な情報を与える．さらに，関数型プログラムは，従来の手続き型プログラムや論理型プログラムに比べて，静的な解析が容易であり，かつ，解析から多くの情報が得られる．

（2） デバッグのヒューリスティクス

手続き的なデバッグでは，優れたプログラマは自分の経験に基づいて確度の高い仮説をたてて，比較的少ない手続きでバグを検出してしまう場合も多い．アルゴリズムに従ってバグを検出する場合でも，単に実行履歴を解析するよりも，プログラムの性質やバグの性質から導かれる仮説に従って戦略的にバグを検索したほうが，より効率的になると期待される．

（3） プログラムの性質の宣言

バグのあるプログラムを実行させた場合には，そもそも予期しない式の計算が現れたり，あるいは，一つのインスタンスを取り出したときに入出力関係が正しいか判断し難いことがある．このため，特定の具現値

の真偽を答える二者択一ではなく，プログラムの性質を宣言するという役割に近づけたほうが，プログラマの負担を軽くし，かつ，バグの探索空間をより効率的に絞り込める可能性がある．

この手法でも図-6 に示した枠組みを使う．ただし，インスタンスの選択は，関数の静的な呼応関係を表す関数依存グラフ G_f，および，インスタンス依存グラフ G_i を用いて次のように行う（図-7 参照）．まず，G_f から検査すべきノード（関数）を選択する．次に，G_i において，そのノードに射影されているインスタンスの中から具現値を選択する．ここで，関数とインスタンスの選択は，デバッグに関するヒューリスティクスから導かれた戦略に基づいて行われる．また，どのインスタンスを関数のノードに射影するかも戦略により定められる．射影関係を制御することにより，各関数についてバグを効率的に検出する上で“重要な”インスタンスだけが探索空間に入れられる[9]．

上記のインスタンス選択法は，タビュレーション技法[37]を少容量メモリで実現する機構と結びついて，限られたメモリでのバグ検出アルゴリズムの実現法，および，停止しないプログラムにおけるバグ検出の支援法を与えるという効用ももたらす[10]．

```
; -- set operation 1. : union --
; -- union program with a bug Ver. 1 --

(defun xmember (u v)
  (cond
   ((null v) nil)
   ((equal u (car v)) T)
   (T (xmember u (cdr v)))))

(defun xunion (u v)
  (cond
   ((null u) nil)
   (T (let ((x (xunion (cdr u) v))
            (y (car u)))
        (cond ((xmember y v) x)
              (T (cons y x)))))))

(diagnoser '(xunion (1 4 5)(2 9 4 6)))

—— SHELL evaluator (show)   File BUGFUNC.LSP ————
```

（a） 初期画面

```
Query : (XMEMBER 1 '(2 9 4 6)) = NIL;  True?

0 : (XUNION '(1 4 5) '(2 9 4 6)) = (1 5);
 9 : (XMEMBER 1 '(2 9 4 6)) = NIL;
  10 : (XMEMBER 1 '(9 4 6)) = NIL;
   11 : (XMEMBER 1 '(4 6)) = NIL;
    12 : (XMEMBER 1 '(6)) = NIL;
     13 : (XMEMBER 1 nil) = NIL;
 6 : (XMEMBER 4 '(2 9 4 6)) = T;
  7 : (XMEMBER 4 '(9 4 6)) = T;
   8 : (XMEMBER 4 '(4 6)) = T;
 1 : (XMEMBER 5 '(2 9 4 6)) = NIL;
  2 : (XMEMBER 5 '(9 4 6)) = NIL;
   3 : (XMEMBER 5 '(4 6)) = NIL;
    4 : (XMEMBER 5 '(6)) = NIL;
     5 : (XMEMBER 5 nil) = NIL;

———————— (Information Window) ——
(XMEMBER 1 '(2 9 4 6)) = NIL;  True?
```

（b） 第1の質問の提示

```
Query : (XUNION nil '(2 9 4 6)) = NIL;  True?

Assertion : (XMEMBER 1 '(2 9 4 6)) = NIL;
FOLDING : XMEMBER
EXPANSION : XUNION ⇒ XUNION.2, XUNION.1
Examine : XUNION.1
0 : (XUNION '(1 4 5) '(2 9 4 6)) = (1 5);
99#: (XMEMBER 1 '(2 9 4 6)) = NIL;
1 : (XUNION '(4 5) '(2 9 4 6)) = (5);
 2 : (XUNION '(5) '(2 9 4 6)) = (5);
  3 : (XUNION nil '(2 9 4 6)) = NIL;

———————— (Information Window) ——
(XUNION nil '(2 9 4 6)) = NIL;  True?
```

（c） 第2の質問の提示

```
Assertion : (XUNION nil '(2 9 4 6)) /= NIL;
OUTER-REMOVAL : XUNION.1
UNFOLDING : XMEMBER
Examine : XMEMBER
INNER-REMOVAL : XMEMBER

There is at least one bug in the instance :
(xunion nil '(2 9 4 6)) = nil;
where
(defun xunion (u v)
  (cond ((null u) nil)
        (t
         (let ((x (xunion (cdr u) v))
               (y (car u)))
           (cond ((xmember y v) x)(t(cons y x)))))))
…… evaluated expression ……
  (null u): (null nil) = t;

———————— (Information Window) ——
Result value: T
```

（d） 最終画面

図-8 宣言的デバッグシステムの実行例

5.3 宣言的デバッグシステム

宣言的デバッグシステム[10]は，戦略的なバグ検出の考え方に基づいて作られたデバッグシステムである．非手続き的なデバッグの具体的なイメージを与えるため，このシステムでの簡単なデバッグ例を示す．

図-8は，二つの関数 xmember, xunion からなる和集合を求めるプログラム（構文は Common Lisp[38]）のデバッグ時の端末画面の変化を示している．図-8(a)では，関数の定義とシステムを起動する式が表示されている．この式を実行してシステムを起動すると，最初の質問，および，システムが選択した関数 xmember のすべての具現値が表示される（図-8(b)）．

最初の質問は正しいのでプログラマが yes と答えると，第2の質問と xunion の具現値が質問の理由とともに表示される（図-8(c)）．質問の理由とは次のことを意味する．システムは，最初の質問の答えから関数 xmember にはバグがないという仮説をたて，xmember を基本関数のように呼び出し元の関数の一部とみなす（FOLDING : XMEMBER）．残された関数 xunion が再帰関数であるので，詳細に調べるために再帰部分と停止部分に分解する（EXPANSION : XUNION ==> XUNION. 2* XUNION. 1）．さらに，XUNION の具現値の中から停止部分の具現値を取り出し質問として表示する．

第2の質問は誤りであるので no と答えると，バグ発生源のインスタンスが検出され，その関数の定義，および，そのインスタンスで実行された式が表示される（図-8(d)），表示は，xunion に nil と '(2 9 4 6) を与えるとバグが現れ，条件式の ((null u) nil) に原因があることを示している．

上の例では質問が簡単なため，プログラマは，単に yes, no と答えている．質問に直接答えないで，表示された具現値から任意の具現値を選び，真偽を答えたり，あるいは，入力パラメータに予期しない値が含まれているなどと宣言することもできる[10]．

6. おわりに

関数型プログラムの作成支援技術のうち，実行時に現れる誤りの検出に焦点をあて，その関連技術として，デバッグしやすいプログラムやデバッグ手法などの問題を取り上げて紹介した．一般に，支援技術は，本来の目的（この場合にはプログラムの作成やデバッグ）の遂行を助けるためのツールとして提供され，どのような機能が備わっているか，どの程度使いやすいか，効果がどの程度得られるか，などが重要な話題となるであろう．

一方，ここで紹介した手法の多くは，プログラミングやデバッギングの新しい方法論を探索している研究で提示・実験されているものであり，現段階ではツールとして広く用いられているものではない．このため，この解説では，支援ツールの機能を整理し，吟味するという立場ではなく，関数型プログラムとの関連から新たに生じた問題意識をデバッグという観点から整理する立場で支援手法を紹介するようにつとめた．

紹介した手法の基礎となる考え方は，特定の言語システムの支援ツールに制限されるものではなく，関数的な性質をもつシステムに対して広く適用できるものが多い．また，支援系に対する新たな考え方は，システムの新しい設計法をもたらす可能性もある．この小論が，新たなシステムの支援系，あるいは，システム自身を考える上で参考になれば幸いである．

参考文献

1) Henderson, P. : Functional Programming, Application and Implementation, Prentice-Hall (1980).
2) Darlington, J., Henderson, P. and Turner, D. A. (eds.) : Functional Programming and its Application, Cambridge University Press (1982).
3) Fairley, R. : Software Engineering Concepts, McGraw-Hill Book Company (1985).
4) Shapiro, E. Y. : Algorithmic Program Debugging, MIT Press (1983).
5) Plaisted, D. A. : An Efficient Bug Location Algorithm, Proc. 2nd Inter. Logic Programming Conference, pp. 151-157 (1984).
6) 佐藤，玉木：デバッグ支援システムの試み，日本ソフトウェア科学会第1回大会，2C-2 (1985).
7) Takeuchi, A. : Algorithmic Debugging of GHC Programs and its Implementaion in GHC, ICOT Technical Report TR-185, Institute for New Generation Computer Technology (1986).
8) 高橋，小野，雨宮：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 (1986), pp. 425-434.
9) 高橋，小野，雨宮：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，情報処理学会論文誌，Vol. 27, No. 9, pp. 869-876 (1986).
10) 高橋，小野：関数型言語の宣言的デバッグ支援システムについて，信学会データフローワークショップ，pp. 255-262 (1987).
11) Johnson, M. S. : A Software Debugging Glossary, SIGPLAN Notices, Vol. 17, No. 2, pp. 53-70 (1982).

12) Barstow, D. R., Shrobe, H. E. and Sandwall, E. Eds.: Interactive Programming Environments, McGraw-Hill (1984).
13) 奥乃，丸山：Lisp のプログラミング環境，情報処理，Vol. 26, No. 7, pp. 741-749 (1985).
14) Davis, A. L. and Keller, R. M.: Data Flow Program Graphs, IEEE Computer, Vol. 15, No. 2, pp. 26-41 (1982).
15) Keller, R. M. and Yen, W. J.: A Graphical Approach to Software Development Using Function Graphs, Compcon 81 Spring, IEEE, pp. 156-161 (1981).
16) Davis, A. L. and Lowder, S. A.: A Sample Management Application Program in a Graphical Data-Driven Programming Language, Compcon 81 Spring, IEEE, pp. 162-167 (1981).
17) 西川，寺田，浅田：履歴依存処理を許すデータ駆動図式，信学論(D)，Vol. J 66-D, No. 10, pp. 1169-1176 (1983).
18) 布川，富樫，野口：図式をシンタックスにもつ関数型言語，信学技法，COMP 87-4, pp. 69-80 (1987).
19) Katayama, T.: HEP: A Hierarchical and Functional Programming based on Attribute Grammar, 5th Int. Conf. Software Engineering, pp. 343-352 (1981).
20) 片山卓也：属性文法による在庫管理システムの記述，情報処理，Vol. 26, No. 5, pp. 478-485 (1985).
21) Bauman, N. B. and Iannucci R. A.: A Methodology for Debugging Data Flow Programs, MIT Tech. Rep. 5-37-84 (1984).
22) Hall, C. V. and O'Donnel, J. T.: Debugging in a Side Effect Free Programming Environment, ACM Symp. on Programming Languages and Programming Environments, pp. 60-68 (1985).
23) 廖，武市：関数的プログラムによる関数プログラムのデバッガ，ソフトウェア科学会関数プログラミング研究会資料 (1986).
24) 二村良彦：構造化プログラム図式，コンピュータソフトウェア，Vol. 1, No. 1, pp. 64-77 (1984).
25) 西川，浅田，寺田：超高位図的言語処理システムの設計思想，第33回情報処理学会全国大会，4F-9 (1986).
26) Dennis, J. B.: First Version of a Data Flow Procedure Language, Lecture Notes in Computer Science No. 19, pp. 362-376 (1974).
27) Arvind, Gostelow, K. P. and Plouffe, W.: An Asynchronous Programming Language and Computing Machine, Tech. Rep. 114 a, Department of Information and Computer Science, University of California, Irvine (1978).
28) 雨宮真人：関数型言語とリスト処理向きデータフローマシン，情報処理，Vol. 26, No. 7, pp. 765-779 (1985).
29) Barendregts, H. P.: The Lambda Calculus-It's Syntax and Semantics, North Holland (1984).
30) Nishikawa, H., Asada, K. and Terada, H.: A Decentralized Controlled Multi-Processor System, Proc. 3rd Int. Conf. on Distributed Computing Systems, pp. 639-644 (1982).
31) Morais D. R.: ID WORLD: An Environment for the Development of Dataflow Programs Writen in ID, MIT Tech. Rep. TR-365 (1986).
32) Curry, H. B., Feys, R. and Craig, W.: Combinatory Logic, Vol. I, North-Holland (1958).
33) Turner, D. A.: A New Implementation Technique for Applicative Languages, Soft. Pract. and Exper. Vol. 9, pp. 31-49 (1979).
34) Jones, S. L. P.: The Implementation of Functional Programming Languages, Prentice-Hall International (1987).
35) 大黒，荒木：関数型言語 pfp の処理系の実現，ソフトウェア科学会関数プログラミング研究会資料 (1986).
36) 川上員譲：関数型言語 pfp 処理系のコード最適化とデバッガの整備，九州大学卒業論文 (1987).
37) Bird R. S.: Tabulation Techniques for Recursive Programs, ACM Computing Surveys, Vol. 12, No. 4, pp. 403-417 (1980).
38) Steele, G. L. et al.: Common LISP, Digital Press (1984).

(昭和63年5月2日受付)