

## 関数型プログラムのデバックにおけるある手法

川崎 治 夫\*

### Some Technics on Debugging for Functional Programs

HARUO KAWASAKI\*

**Synopsis:** A bug detection method for list processing programs is described which is written by functional programming languages. The methods and thoughts of errors in the numerical analysis are applied to debugging. The propagation of bugs has similar properties like that of errors in the numerical analysis, so this paper presents a new method used these properties.

**要旨:** 関数型プログラム言語で書かれたリスト処理プログラムのデバックを支援する方法について述べる。これらのプログラムについて数値計算上の誤差に対する方法、考え方をデバックに応用する。バグの伝播は、数値計算上の誤差の伝播に似た性質を持っているので、この性質を利用した方法を述べる。

#### 1. ま え が き

関数型プログラムのデバックについては、機械的にプログラムのバグを検出するシステム[1, 2]の研究がなされている。実際の処理環境としては、システムと対話型に処理を進めていくことによりバグを指摘されるように構成されている。[1]においては、[2]を基礎としたシステムにおいてプログラマへの質問回数を減少させる為の試みがなされている。

しかし、取り扱うことができるプログラムの範囲、バグの種類という点においては十分ではない。関数性をもつプログラム言語では、その持つ性質により他のものに比べれば少し簡単に機械的にバグを検出するシステムを実現できるが、プログラムの誤りの程度がひどくなったり、検証すべきプログラムの複雑さが増してくるとプログラム言語の性質を超えたソフトウェアの検証に共通する問題となってくる。バグの発生ということをもう一度根本から考え直す必要がある。本稿においては、その為に数値計算上の誤差に対する考え方、方法論を関数型プログラムのデバックへ適用することについて述べる。本稿で取扱う関数型プログラムはリスト処理プログラムに限定するが、この一見関数係が全くなさそうなリスト処理プログラムと数値計算上の問題が「誤り」という世界において共通性を持っていることを示しデバックの一方法を示す。このような異質な分野に共通性を見出しての研究は現在のところ全くなされていない。

---

\*国士舘大学電子計算機センター助教授

Assistant professor, Computer Center

い。数値計算の誤差の問題は完全に解明しつくされた問題ではないが、プログラムのバグに対する研究よりは、はるかによく研究されている。従って方法論的にも関数型プログラムのデバッグに利用できるものが多いと思われる。

## 2. 数値計算上の誤差と関数型プログラムのバグについて

通常数値計算上で誤差を検討する場合は、その計算を実行するプログラムの正当性は保証されているものとして議論が進められている。しかしここで取り上げる関数型プログラムには全くそのような保証はない。前提条件においてこのような差があるのであるが、誤差の研究[4]における誤差混入、誤差増大のメカニズムを調べるということは、この「誤差」という言葉を「バグ」とかえると関数型プログラムの世界と関りがでてくる。

### 2.1 情報落ちとバグ発生

本稿で扱っているリスト処理用関数型プログラムは、実数、整数を定義域とするそれとはちがって、出力値が履歴を含んでいるという利点がある。数値計算上では「桁落ち」という現象が起きるが、これによく似たことが関数型プログラムにおいてもバグの発生によって起きる。

例 1 バグのある  $l_1$  と  $l_2$  の和集合を求めるプログラム

```
union1 [ $l_1$ ;  $l_2$ ]  $\equiv$  if  $l_1 = \text{nil}$  then nil
                     else if member[car[ $l_1$ ];  $l_2$ ] then union1 [cdr[ $l_1$ ];  $l_2$ ]
                     else otherwise cons[car[ $l_1$ ]; union1 [cdr[ $l_1$ ];  $l_2$ ]]
```

union1 [(A B); (A C D)] の値を求める。

union1 [(A B); (A C D)]

= union1 [(B); (A C D)]

= cons[B; union1 [nil; (A C D)]]

①

= cons[B; nil]

②

=(B)

上の展開において誤った出力値が得られた原因は①から②へと移る際に (A C D) が全く落ちてしまったからと考えられる。union1 [(A B); (A C D)] が union1 [(B); (A C D)] となった時ではまたわからないが、①から②へ移ることによって全く正しい出力値が得られる可能性がなくなっている。このような状態を「情報落ち」と呼ぶことにする。この場合、情報落ちは①から②へ移る際つまり union1 [nil; (A C D)] = nil によって生じている。

勿論、「情報落ち」風な現象は、正当性を保証されたプログラムにおいてもバグを含んだプログラムにおいても見かけ上は出現する可能性はある。しかし機械的にバグを検出するシステ

ムを作成するときには，探索空間を狭める戦略としては有効性を持つものと思われる。

リスト処理プログラムにおいては，途中の計算過程を考えに入れなくても入力パラメタのみから出力結果の可能性を推測できる場合が多いので，このような種類のプログラムのバグの検出に対しては有効だと思われる。

## 2.2 情報落ちのチェックと探索空間の分割

誤った計算結果を得た場合には，デバッガがバグの発生源を求めて動き出すのであるが，むやみに探しまわったりしないように考えなければならないし，システムが使用者に与える質問の回数も少ない方がよい。

### 2.2.1 入力履歴系列

問題としては，入力データは正しいものが与えられてプログラムが動きだし，予期しない誤った結果を出力して停止する場合を考える。 $n$ 変数関数型プログラムを以下のように表現する。

$$\begin{aligned} f[\alpha_1; \alpha_2; \dots; \alpha_n] &\equiv \text{if } P_1 \text{ then } e_1 \\ &\quad \text{else if } P_2 \text{ then } e_2 \\ &\quad \dots\dots\dots \\ &\quad \text{else if } P_n \text{ then } e_n \\ &\quad \text{else otherwise } e_{n+1} \end{aligned}$$

一つの入力データに対して結果を出力してプログラムが停止するということは，実行履歴を見ると  $e_i (i=1, 2, \dots, n+1)$  の各変数に適当なデータが与えられた形式  $e'_i$  の有限列である。

入力履歴系列は次の手順で得られる。

- ① 入力データに対する実行履歴の中で出現する  $e'_i (i=1, 2, \dots, n+1)$  に対して入力引数となっている  $S$ -式から，それを構成しているアトム集合を作る。
- ② ①の操作を実行履歴に出現するすべての  $e'_i$  に対しておこない，実行履歴における計算順に並べて①で得られた集合の列を作る。

このようにして得られた集合の列を入力履歴系列と呼ぶ。

$f[(A(B)); (A\ C)]$  から①のレベルで得られる集合は  $\{A, B, C\}$  である。

誤った出力結果を得た場合には，本来の正しい値を入力履歴系列上でチェックしてどこで情報落ちが生じているか調べてやればよい。後は，必ずしも情報落ちしてもバグの発生源とならない場合があるので，情報落ちに関わる  $e'_i$  の最終の値を求めて，正しい場合には，バグの発生源でなくバグの発生源は自分の親あるいは，もっと上のレベルにあることになる。

正しくない場合は，ここを探索空間分割のポイントにできる。また複数個の場所で情報落ちしているときは，実行順序に従ってチェックすればよい。



### 3. 実 例

例 3.1 例1のプログラムを使用する。

今  $\text{union1}[(A\ B); (A\ C\ D)] = (B)$  を得たとする。この場合入力履歴系列は以下のようになる。

$\{A, B, C, D\} \{A, B, C, D\} \{A, B, C, D, \text{nil}\} \{B, \text{nil}\}$

本来ならば  $(A\ B), (A\ C\ D)$  という入力データに対しては  $(B\ A\ C\ D)$  という値を出力しなければならないのだから,  $\{A, B, C, D, \text{nil}\}$  から  $\{B, \text{nil}\}$  へ移るとき情報落ちしている。

$\text{cons}[B; \text{union1}[\text{nil}; (A\ C\ D)]]$  の値を  $\text{cons}[B; \text{nil}]$  としたところである。

また  $\text{union1}[\text{nil}; (A\ C\ D)] = \text{nil}$  と誤った結果となるので, この部分つまり  $\text{union1}[\text{nil}; (A\ C\ D)] = \text{nil}$  の出力原因となった停止部が, おかしいことがわかる。

例 3.2 バグのある  $l_1$  と  $l_2$  の和集合を求めるプログラム

```
union2 [ $l_1; l_2$ ]  $\equiv$  if  $l_1 = \text{nil}$  then  $l_2$ 
      else if member[car[ $l_1$ ];  $l_2$ ] then union2 [cdr[ $l_1$ ];  $l_2$ ]
      else otherwise cons[cdr[ $l_1$ ]; union2 [cdr[ $l_1$ ];  $l_2$ ]]
```

今  $\text{union2}[(A\ B); (A\ C)]$  の値を求める。

```
union2 [(A B); (A C)]
= union2 [(B); (A C)]
= cons[nil; union2 [nil; (A C)]]
= cons[nil; (A C)]
= (nil A C)
```

本来の正しい値は  $(A\ B\ C)$  のはずであるから, 入力履歴系列を作る。

$\{A, B, C\} \{A, B, C\} \{\text{nil}, A, C\} \{\text{nil}, A, C\}$

入力履歴系列を見ると  $\{A, B, C\}$  から  $\{\text{nil}, A, C\}$  へ移るとき情報落ちしている。

$\text{union2}[(B); (A\ C)]$  の値を  $\text{cons}[\text{nil}; \text{union2}[\text{nil}; (A\ C)]]$  としたところである。

$\text{union2}[(B); (A\ C)]$  の値はまちがっているので結局プログラム上の **else otherwise** 以下がおかしい。

例 3.3 バグのある  $l_1$  と  $l_2$  の共通集合を求めるプログラム

```
int1 [ $l_1; l_2$ ]  $\equiv$  if  $l_1 = \text{nil}$  then nil
      else if member[car[ $l_1$ ];  $l_2$ ] then cons [car[ $l_1$ ]; int1 [cdr[ $l_1$ ]; cdr[ $l_2$ ]]]
      else otherwise int1 [cdr[ $l_1$ ];  $l_2$ ]
```

今  $\text{int1}[(A\ B); (B\ A)]$  の値を求める。

```
int1 [(A B); (B A)]
=cons[A; int1 [(B); (A)]]
=cons[A; int1 [nil; (A)]]
=cons[A; nil]
=(A)
```

本来の正しい値は  $(A\ B)$  のはずであるから、入力履歴系列を作る。

$\{A, B\}\{A, B\}, \{A, \text{nil}\}\{A, \text{nil}\}$

入力系列を見ると  $\{A, B\}$  から  $\{A, \text{nil}\}$  へ移るとき情報落ちしている。しかしこの場合は、 $\text{int1}[(B); (A)]$  の値は正しいので、バグの発生源は自分の親または、それ以上のレベルにあることになる。従ってこの場合は、親または、それ以上のレベルというのは  $\text{int1}[(A\ B); (B\ A)]$  の値を求めて  $\text{cons}[A; \text{int1}[(B); (A)]]$  とした部分ということになる。 $\text{int1}[(B); (A)]$  は、 $\text{int1}[(A\ B); (B\ A)]$  の出力を誤らせた原因ではない。従って最初の時に既にまちがった情報が入っていたことになる。つまり  $\text{cons}[\text{car}[l_1]; \text{int1}[\text{cdr}[l_1]; \text{cdr}[l_2]]]$  の部分がおかしい。

#### 4. バグの修正

数値計算においては、桁落ちで下位の桁の有効情報が落ちないようにデータを倍長語の形で扱うという方法をとることがある。関数型プログラムのバグを修正するときに、これと似た方法を使うことができる。

例 4

例 3.3 のプログラムのバグの修正について考える。

$\text{int1}[(A\ B); (B\ A)] = (A)$	(誤)
$\text{int1}[(A\ B); (C\ B\ A)] = (A\ B)$	(正)
$\text{int1}[(A\ B\ C); (D\ C\ B\ A)] = (A\ B)$	(誤)
$\text{int1}[(A\ B\ C); (E\ D\ C\ B\ A)] = (A\ B\ C)$	(正)
$\text{int1}[(A\ B); (B\ A)] = \text{cons}[A; \text{int1}[(B); (A)]]$	①
$\text{int1}[(A\ B); (C\ B\ A)] = \text{cons}[A; \text{int1}[(B); (B\ A)]]$	②
$\text{int1}[(A\ B\ C); (D\ C\ B\ A)] = \text{cons}[A; \text{int1}[(B\ C); (C\ B\ A)]]$	③
$\text{int1}[(A\ B\ C); (E\ D\ C\ B\ A)] = \text{cons}[A; \text{int1}[(B\ C); (D\ C\ B\ A)]]$	④

$\text{int1}[(A\ B); (B\ A)]$  は正しい値とならないが、第2引数の長さを1殖やし  $(C\ B\ A)$  とすると正しい値を出力する。また  $\text{int1}[(A\ B\ C); (D\ C\ B\ A)]$  についても同様のことがいえる。

修正の方針としては、①の右辺を②の右辺で置き換える(③の右辺を④の右辺で置き換える)

という方法を採用。結局  $\text{int1}$  の第2引数だけに変化があって①については  $(A)$  を  $(BA)$  に、③については  $(CBA)$  を  $(DCBA)$  に換えればよい。

よって  $\text{int1}[\dots; \text{cdr}[l_2]]$  を  $\text{int1}[\dots; l_2]$  と修正してやればよい。

## 5. ま と め

数値計算の世界とリスト処理の世界は、通常無関係と思われているが、「誤り」という点においては、類似点を持っていることがわかる。まだまだ調べなければならない事は多いが、次の段階としては、適用範囲を拡張することと、数値計算上の手法で他に応用できるものがないか調べていく。誤差の解析については、プログラムのデバッグよりはるかによく研究されているので、そちらからのバックアップによる進展を期待したい。

(1985年10月31日 受理)

## 参 考 文 献

- 1) 高橋, 小野, 雨宮。“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” 情報処理学会ソフトウェア基礎論研究会, 72-3, 1985
- 2) Shapiro, E. Y., “Algorithmic Program Debugging” MIT Press, 1983
- 3) Anderson, R. B., “Proving Programs Correct” (演習プログラムの証明 有澤訳 近代科学社1980)
- 4) 戸川隼人 “計算機のための誤差解析の基礎” サイエンス社, 1974
- 5) J. H. Wilkinson “Rounding Errors in Algebraic Process”, 1963 (基本的演算における丸め誤差解析 一松, 四条訳 培風館)