並行プログラミングのための作用的通信機能

山野紘一　木谷有一　渡辺　坦
（日立システム開発研究所）

1　はじめに

最近、作用的プログラミング言語が2つの面より注目されている。1つは高度の並行処理の実現を目ざすことからである。従来、実行効率の点でさけられていたが、新しいハードウェア技術とVLSIとの発達によって現実のものとなってきたことによる。

もう1つは、プログラムの記述・検証能力の観点からである。ハードウェアコストが次第に安価になるのに比べて、ソフトウェアのコストは益々高くつくようになってきている。その要因の大きなものは、プログラムの誤りによる虫とりのコストと修正のコストである。

この問題の解決には、誤りを起こしにくくすると共に、検証を容易にし、修正が簡単にできるような記述方式が求められている。構造化プログラミングは、主として、プログラムの制御構造の面からのアプローチであった。作用的プログラミング言語の特徴は、プログラムの計算過程における評価方式からのアプローチで、副次的効果を許さないことにあり、プログラマに透明な環境をもたらす。

代表的な作用的言語には、純粋なLispやBackusのFP[1]、プログラムの証明を目的とするLucid[4]、データフローマシーン向き言語のVal[2]、Id[3]、Valid[19]、帰納方程式にもとずくHOPE[3]などがある。

我々は、並行プログラミングに適合する言語VULCAN[5]（Value oriented Language for Reliable Calculation）を作成した。VULCANは、並行プログラミングのために、プロセスの概念を導入したもので、プロセスは、通信と函数から構成される。ここでは、プロセス間の通信機能を作用的構造のわく組の中で行なうために考案したbracketと呼ぶブロックについて述べ、帰納方程式にもとずく言語による記述との比較を行なう。

2．言語作成の目的

2.1．信頼性の向上

(1)作用的構造の実現

プログラムの理解や正しさの検証において，明晰性（referential transparency）の原理が非常に重要である。この原理にもとづく言語構造は、作用的（applicative）構造と呼ばれている。作用的であるとは、式（函数と値から構成）の評価においては、副次的効果を伴なわないことを意味し、プログラムの理解の容易性と信頼性の向上につながる。

(2)検証の容易性

プログラムの記述方式を考えるとき、検証面からの考慮が必要である。検証方法には、静的検証、動的検証、正当性の証明がある。副次的効果を伴なう言語に対しては、静的検証におけるデータフロー解析の技法を適用するにしても、副次的効果の不透明性がその解析能力の障害となっている。動的検証においてもテスト環境の設定、値のトレースなどが非常に複雑なものになっている。作用的言語であるからといって、必ずしも検証が容易になってい

るとは限らないので検証の容易性を考慮する。

2.2. 並行プログラミング

通常、並行プログラミングというと多重プロセッサに対するプログラムやオペレーティングシステムでの多重タスクのプログラムを作成する方式であるが、ここでは、プログラムの強力なモジュール化構造の1つとして、仮想的に並行しているプロセスを用いたプログラム作成方式をも含むものとする。これは、順次プログラムの作成においても、各モジュールの相互関係をより明確にする手段としてプロセスを活用することを考える。

3. 言語の概要

プログラムの基本単位はプロセスである。プロセスは、通信と函数より構成する。

3.1. 函数

VULCANの基本要素は、次のように函数と値より構成する。

y = = f : x

函数fは、対象xに作用し、結果の値yを生成する。fへの入力情報はxのみで、それ以外の情報は一切使用できず、その結果はyのみで、それ以外の効果は生じないという作用的構造をもつ。従って、同じ入力に対しては、同じ出力を生成する。

VULCANには代入文はない。作用的言語において、計算の結果を定義する名前を許さないもの(BackusのFP)と名前付けを許すもの(Val, Id, Hendersonの記述[7]など)がある。VULCANは、型付きの値をもち、名前付けを許す形式とし、型チェックの可能性とプログラムの分かり易さとの配慮をしている。

函数の引数の評価は、引数を先に評価(作用的順序)する値渡しの方式をとる。繰り返し函数の中での値の再定義には、それを明示するために、Lucidや[8]Idと同じ考えを採用して、newという再定義函数を用いる。

3.2. プロセス

プロセスは型とし、act函数によって、具現化と起動を行ない、終了はdeact函数で指示する。起動から終了までのプロセス生存期間では、プロセス内で直接定義された値が生存する。この意味において、プロセスは状態をもつ。

プロセス間通信の方式は、HoareのCSP[9]にもとづいたものにした。プロセスには、端子(port)を設け、端子と端子を結びつけたものを通信路と呼ぶ。通信路の設定は、connect函数で行ない、CSPのように通信の相手を入出力コマンドで直接指定する方式とは異なる。これは、プロセスの結合を外部において明示指定できるようにし、プロセスの独立性を高めることをねらったものである。情報交換は、通信路を流れるメッセージで行ない共有領域は存在しない。また、通信によって、値の受け渡しと共に同期機能をもはたすことにし、端子には、非決定性機構をもたしている。

通信路によって結合されたプロセス群はネットワーク[12]を形成する。このネットワークは、プロセスの合成を代数的に取り扱うことも可能となる[9]。

4. 通信機能

4.1. 入出力函数

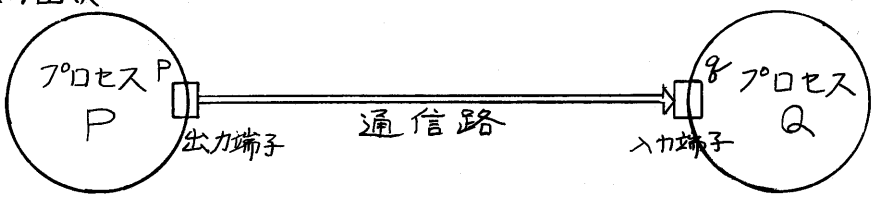

図4.1. 通信路

プロセス間の通信路は、値のストリーム[11]（無限長の列）と考えられる。図4.1.に示す通信路は、端子Pと端子qとを結ぶものである。プロセスPからみるときは、出力端子Pがそこに形成されているストリームの値名を、プロセスQからは入力端子qがストリームの値名を示すものと考える。

端子の定義は、

```
port  out   p : T'value   (プロセスP)
port  in    q : T'value   (プロセスQ)
```

で与える。ここで、T'valueは、ストリームを構成する要素の値の型を示す。

ストリームxを、$[x_1, x_2, \cdots]$とし、空ストリームを[ ]で、未定義を⊥で表わして、ストリーム上の演算を与える。

```
first(x : stream -> T'value) ==
        if x=[ ] then ⊥ else  x1 ∈ x=[x1 , x'] endif
rest (x : stream -> stream) ==
        if x=[ ] then ⊥ else  x' ∈ x=[x1 , x'] endif
append(x : stream , v:T'value -> stream) ==
        if x=[ ] then [v] else [x , v] endif
```

VULCANの入出力は、HoareのCSPと同じ記法を用いて、

- ○入力函数　　q？：<nil>
- ○出力函数　　p！：<v>

と記述する。これらの函数の意味は、ストリームに対する演算を使って、次のように定義される。

- ○入力函数　？(q : stream , v : T'value -> stream , T'value) ==
  if q=[ ] loop else <rest:<q>, first:<q>> endif
  ここで、loopは、q=[ ]が成り立つ間中、qの評価を繰り返すことを示すもので、その間にqに対する出力によって効果があるものとする。
- ○出力函数　！(p : stream , v : T'value -> stream , T'value) ==
  <append :< p, v>, v >

入力函数に対して、領域<nil>があるのは、出力函数と対称にするためのものであって、特別の意味はない。

出力函数は、端子で示されているストリームへのappendであり、出力されたストリームを直接、同じプロセス内で参照することはできない。従って、出力

によって副次的効果は生じない。
入力函数は、入力しようとしているプロセスの外の世界で生成されたストリームからの先頭の取り出しである。従って、現プロセスには直接依存しないストリームの値の切り出しであり、副次的効果はない。
VULCANでは、通信の制御方式として、同期方式と非同期方式とを設けた。
○同期方式
入力と出力の待ち合わせを実現するもので、入力と出力の取扱いは同等とする。
○非同期方式
これは、出力側で、入力の待ち合わせを行なわない方式で、出力は、順次行なうことができる。入力は、ストリームが空のとき、待ち合わせることになるが、それ以外のときは、順次入力することができる。

通信の制御方式は、ストリームの流れの制御に関するもので、値に関係するものではない。

4.2. 通信と函数
ある函数の中で、入出力を許すと、その函数は、作用的構造の性質をもたなくなる。

```
func  g (x: port of integer, y: integer -> integer)==
        +:<y, x?:<nil>>
endf ;
```

gは、端子xより入力して、yを加えた結果を返す函数である。ここで、Pをport of integer型の引数とした函数呼び出しを考える。

```
a==  g:<P,2>
b==  g:<P,2>
```

一般に、aとbは等しいとはいえない。従って、gは作用的函数でない。
入出力には、時間(順序)が本質的に伴なうことであり、何んらの制御が必要である。特に入力においては、要求駆動であるので、函数における値駆動とは異なる。そこで、VULCANでは、次のようにした。
(1)函数の引数として、端子の値を渡すことはできない。
(2)入出力を制御するbracketというブロックを導入する。

4.3 bracket
函数の基本操作は、合成にあり、その記述形式が言語の特徴を表わすことになる。VULCANでは、次のような合成函数の記述ができる。

```
<u, v> == <+:<x, y>, *:<+:<x, y>>>
```

しかし、一般に、上の形式のものだけとすると分かり難くなる場合がある。そこで、bracket というブロックを使うことができ、braの次には入力を、ketの次には出力を示すものである。前の記述は、次のようになる。

```
bra (x, y);
    u == +:<x, y>;
```

```
        v == * ; <x, u>
    ket (u, v);
```

このbracketの形式は、前の記述より読み易くなる。但し、u, v は、値の定義を示すもので、通常の言語の代入文とはちがう。

この例で、yを端子Pより入力して、vを端子Qに出力するときには、次のように、braあるいはketの次に、withという句をつけることにする。with句の中には、通信（入力と出力）のみを書くことができる。

```
bra (x) with   y == p ? : <nil> endw ;
               u ==   + : <x, y> ;
               v ==   * : <x, u>
ket (u) with   q ! : <v> endw ;
```

このbracket は、合成函数の表現に通信を持たせるように拡張したもので、通信付きのbracket と呼ぶ。

通信付きbracketの意味は、次のものである。

(1) 入口（braに続くwith句）において、通信函数を評価する。その順序は、記述された順に行なう。待ち合わせがある時には、その時点で待ち合わせる。

(2) 入口の通信函数がすべて評価されたら、bracketの中の函数の評価に移る。

(3) bracketの中の函数の評価がすべて終了してから、出口（ketに続くwith句）の通信函数を評価する。

(4) 出口の評価がすべて終了したときを、bracketの評価の終了とする。

bracket において、副次的効果が生じる要因はないので、作用的構造は保持されている。

4.4. 非決定性機能

通信によって送られてくる値は、それを入力するプロセスと独立に到着する。複数のプロセスを相手にするとき、言語機能として非決定性の表現ができると便利である。ガードを用いて、

```
guard
  //   B1    => 値定義 ;
  //   B2    => 値定義 ;
               :
```

と表わす。ガードB1, B2・・・は、boolean値をとる函数である。通信は許さない。このガードによって生じる非決定性を局所的（プロセス内の状態による。）非決定性という。

VULCANでは、プロセス間の通信で生じる大域的非決定性もある。これは通常、非決定性マージと呼ばれる非決定性で、ストリームのマージがランダムに行なわれることを意味する。同期方式の場合は、あるプロセスに対して、2つ以上のプロセスからの送信があるとき、待ち合せの選択は、ランダムに行なわれ、どれか1つが選ばれ、残りのものは、次の機会まで待たされる。非同期方式の場合は、ランダムにappendされていく。

入力側において、出力元を知りたいときがある。ストリームとして出力される値には、自動的に出力側のプロセスの識別名を付加することにしている。従って

送り主を知りたいときは、入力函数？？を用いて、送り主を知ることができる。
また、出力端子に対して、送り先を指示するときには、出力函数！！を用いて実現することができる。

4.5 帰納方程式によるプログラムとの比較

図4.2に示すプログラム例は、Hendersonによって作成されシステム[13]（図中＊で示す行）をVULCANで記述したものである。Hendersonのプログラム（HPで略す）は帰納方程式（recursion equation）を基礎にした作用的言語で記述されている。このプログラムは2つのキーボードより打ち込まれたコマンドを処理し、データベースからの応答を対応するそれぞれのスクリーンに表示することを記述している。

(1)入出力

HPは、keyboard, screen はそれぞれストリームとして端末を抽象化しているので明示されない。一般に、帰納方程式言語ではそこまでモデル化するものはない。[10]

(2)同期

HPでは、同期は暗黙的に実現されるものと仮定している。すなわち、キーボードより打ち込まれたコマンドは、次々とSYS2に送り込まれ、処理され、その結果はスクリーンに出されるものとしている。VULCANでは、入出力によって、明示的に記述している。

(3)非決定性

元来、Hendersonがこのプログラムを作った目的は、函数"interleave"が作用的構造のわく組に含まれるかを議論することにあった。"interleave"を要求駆動と考えるならば、このプログラムは作用的ではあるが函数ではないということである。VULCANでは、"interleave"に対するものは、非同期方式の端子に対する入力函数で表わされ、基本的に"interleave"と同じ機構である。

帰納方程式をベースにした言語[10,15]は、記述力が非常に強力であるが、その意味を明確にしたりするには、annotationsが必要である。

5. プログラム仕様の記述

VULCANとプログラムの設計法とを関連づけることは、ソフトウェア工学の面から有効である。プログラムの設計法として非常に明解であり、基本的事項を含んでいるものにJackson法[17]がある。Jackson法は、入力と出力のデータ構造を分析、それらを対応づける函数を見い出す方法であるが、重要なのは、構造不一致（structure clash）の問題がある場合の解決法の提案である。

このような構造不一致は、VULCANのプロセスを用いた並行プログラミングによって、自然な形で解決することができる。すなわち、作成するプログラムが逐次プログラムであっても、設計のツールとして活用できる[18]。

作用的言語と呼ばれているものは一般に記述能力が強力であるので、従来のプログラミング言語では簡潔に表現できないものも容易に記述することができる。この特徴を活かし、プログラムの仕様記述言語として用い、実用のプログラムに変換するプログラム変換技法[19]を適用すれば、ソフトウェア工学で問題となってい

```
frame PFOS;-- Purely Functional Operating Systems described by VULCAN.
  type SeqStr == seq(String);
       String == seq(char);
  --Following shows the original version described by P.Henderson.
  --* sys2(keyboard1,keyboard2)=screen1,screen2
  --*   where screen1 = untag(1,s)
  --*     and screen2 = untag(2,s)
  --*     and s = tdbf(interleave(tag(1,keyboard1),tag(2,keyboard2)))
  --*   tdbf(c)==tdbf1(c,NIL)
  --*   tdbf1(c,db)==cons(m,tdbf1(tl(c),db1))
  --*     where m,db1 = tdbstep(hd(c),db)
  --*   tdbstep((t PUT f s),db) = (t DONE),put(f,s,db)
  --*   tdbstep((t GET f  ),db) = (t get(f,db)),db

  type Sys2 == process(Keyboard1,Screen1
                      ,Keyboard2,Screen2:String);
    port in  Keyboard:SeqStr;
    port out Screen  :String;
    type Database: seq(SeqStr);
    val  KB,SC,S,M:String;
         Command,Com:SeqStr;
         DB:Database==nil;
    func Tdbstep(C:SeqStr,D:Database -> SeqStr,Database)==
          if #1:<C>='PUT' then <'DONE',put:<#2:<C>,#3:<C>,D>>
      elseif #1:<C>='GET' then <       get:<#2:<C>,D>,D>
                          else <'MalCommand',D> endif endf;
    func Scid(Kbid,Kb1,Kb2,Sc1,Sc2:String -> String)==
           if Kbid=kb1 then Sc1
      elseif Kbid=Kb2 then Sc2 endif endf;
    guard
    //fill:<Keyboard> =>
      bra  <Keyboard1,Keyboard2,Screen1,Screen2,DB>
             with <KB,Command>==Keyboard??:<nil> endw;
        <M,new DB>==Tdbstep:<Command,DB>;
               SC ==Scid:<KB,Keyboard1,Keyboard2,Screen1,Screen2>;
      ket <DB> with Screen!!:<SC,M> endw;
    endp;
  type KB301==process; -- A hardware model for the keyboard.
    port out Kchar:SeqStr;
      :
    endp;
  type SC401==process; -- A hardware model for the screen.
    port in Schar:String;
      :
    endp;
  bra;-- The body of PFOS is defined as follows.
    type SYS:Sys2;
         KBA,KBB:Keyboard;
         SCA,SCB:Screen  ;
         R1,R2,R3,R4:boolean;
    -- process activation
    KBA==act:<KB301>; -- keyboard A
    KBB==act:<KB301>; -- keyboard B
    SCA==act:<SC401>; -- screen A
    SCB==act:<SC401>; -- screen B
    SYS==act:<Sys2(id(KBA),id(KBB),id(SCA),id(SCB)>;
    -- port connection
    R1 == connect:<KBA,Kchar ,SYS,Keyboard>;
    R2 == connect:<KBB,Kchar ,SYS,Keyboard>;
    R3 == connect:<SYS,Screen,SCA,Schar,>;
    R4 == connect:<SYS,Screen,SCB,Schar,>;
  ket;
endframe PFOS;
```

図4.2. VULCANによる記述例

るプログラムの保守等の1つの解法になる。
仕様記述の場合には、対象の正確な規定も必要であり、たとえば、並行プログラムの対象とするものは、実際の装置や端末、既存のOS等の実世界をモデル化できる能力も必要となる。この点、DULCANのプロセスの概念は、非常に有効である。

6. おわりに

並行プログラミングにおいて、プログラムの信頼性を高める上から、副次的効果の伴なわない作用的構造をもった言語DULCANの通信機能を中心に述べた。プロセス間の通信は、メッセージ渡しの方式で行なうようにして、プロセスの独立を図り、モジュラリティのよい並行プログラミングを可能にした。
通信に対する入出力には、順序関係が発生する。この問題を作用的構造のわく組の中で取り扱うために、函数と通信との調整を図るbracketというブロックを導入した。
また、帰納方程式に基礎をおく言語によるプログラムと比較し、DULCANの有効さを示した。さらに、プログラム設計法ともうまく融合できることを述べた。
現実のハードウェア環境を考えるとき、作用的言語から命令的言語へのプログラム変換技法を開発することが必要である。そして、この変換技法の開発が、プログラムの信頼性を着実に高めてゆくための確実な方法となるように思われる。

7. 参照文献

1 Backus,J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. CACM, 21, 8, pp.613-641, 1978
2 Ackerman,W.B. and J.B.Dennis VAL-A Value-Oriented Algorithmic Language : Preliminary Reference Manual. Technical Report MIT/LCS/TR-218, 1979
3 Arvind, K.P.Gostelow and W.Plouffe The Id Report : An Asynchronous Programming Language and Computing Machine. University of California, Irvine, Technical Report #114, 1978
4 Ashcroft,E.A. and W.W.Wadge Lucid, a Nonprocedural Language with Iteration. CACM, 20, 7, pp.519-526, 1977
5 Kitani,Y, K.Yamano and T.Watanabe Functional-Style Programming with Communication. Proc. 6th Int. Conf. on Software Engineering, Poster Session, pp.17-18, 1982
6 Hoare,C.A.R. Communicating Sequential Processes. CACM, 21, 8, pp.666-777, 1978
7 Henderson,P. Functional Programming Application and Implementation. Prentice-Hall, 1980
8 Burstall,R.M., D.B.MacQueen, and D.T.Sannella HOPE:An Experimental Applicative Language. 1980 LISP Conf., pp.136-143, 1980
9 Pratt,V.R. On the Composition of Processes. Ninth Annual ACM Symposium on POPL. ACM, pp.213-223, 1982
10 Schwarz,J. Using Annotations to Make Recursion Equations Behave. IEEE SE-8, 1, pp.21-33, 1982
11 Burge,W.H. Recursive Programming Techniques. Addison-Wesley, 1975
12 Kahn,G. and D.B.MacQueen Coroutines and Network of Parallel Processes. Information Processing 1977 (ed. by B.Gelchrist), North-Holland, pp.993-998, 1977
13 Henderson,P. Purely Functional Operating Systems. Functional Programming and its Applications (ed. by J.Darlington, P.Henderson and D.A.Turner), Cambridge University Press, pp.177-192, 1982
14 Burstall,R.M. and J.Darlington A Transformation System for Developing Recursive Programs. JACM, 24, 1, pp.44-67, 1977
15 Turner,D.A. Recursion Equation As A Programming Language. Functional Programming and its Applications (ed. by J.Darlington, P.Henderson and D.A.Turner), Cambridge University Press, pp.1-28, 1982
16 Jackson,M.A. Principles of Program Design. Academic Press, 1975
17 Jackson,M.A. Information Systems:Modelling Sequencing and Transformations. Proc. 3rd Int. Conf. on Software Engineering, pp.72-81, 1978
18 Yamano,K. and Y.Matsumoto Unified Functional Design Technique Based on Data Flow Concept. COMPSAC81, pp.369-377, 1981
19 雨宮,尾内 データフロー用高級言語VALIDについて 信学会,電子計算機研究会 EC82-9, 1982