（1982. 12. 10）

関数型言語 Valid における記号処理機能の拡充

小野 諭　長谷川隆三　雨宮真人

"Extension of Functional Programming Language VALID for Symbol Manipulation"

Satoshi ONO, Ryuzo HASEGAWA and Makoto AMAMIYA
(Musashino Electrical Communication Laboratory, N.T.T.)

The authors have proposed new functional programming language VALID for parallel numerical/list processing applications. VALID is designed to be executed on dataflow computers efficiently. In this paper, VALID is extended to improve symbol manipulation ability. Declaration, pattern matching and lenient/lazy evaluation for recursive data structures are introduced. To handle data abstraction and history sensitivity, concept of module is also proposed. Extended compiler is currently under development.

Key word: parallel processing, symbol manipulation, recursive data structure, pattern matching, lazy evaluation, module

1. まえがき

pure Lisp[1]に代表される関数型言語は記述の簡潔明瞭さ，検証の容易さ，並列性記述の平易さなどの点で従来型言語にない優れた性質を有しているがノイマン型マシンの上での実現効率の悪さから従来型言語ほど普及しなかった。しかしながら最近のＬＳＩ技術にみられるようなハードウェア素子技術の発展及びデータフロー計算機方式などの非ノイマン型計算機アーキテクチャ研究の進展とともに，並列処理記述のための実用的な言語[2][3]として、注目されるようになってきた。

従来型言語が抱えている種々の問題は、プログラムカウンタを用いた逐次的集中的制御概念に基づき、メモリセルの内容を書換えることで処理を進めるというノイマン原理に帰因している。例えば、逐次実行の概念は計算の順序を強制的に定めてしまい並列実行の可能性を欠う他、gotoのようなjump概念が入りプログラムの構造を不透明にする。またメモリセル概念は計算に本質的でないプログラム変数の概念をもたらし副作用など煩わしい問題を引起こす元凶になっている。

一方関数型言語では、プログラムは関数として扱い、その値は引数により一意に定まるものと考える。メモリセルの概念はなく変数はあくまで数学的変数である。副作用がないので関数計算は互いに独立になり並列処理の可能性が生じる，プログラムの意味は鮮明であり関数の合成・結合が容易に行えるなど多くの利点が存在する。

以上の観点から我々はデータフローマシン用高級言語として、関数型言語 Valid (Value identification language)[4]の開発を進め、データフローマシンと関数型言語の適合性を明確化してきたが[5]、実現上次の点が未整理・未解決な問題として残されている。

- 構造データをトークンとして流すのは現実的でないので、共有メモリを設ける必要がある。また単一代入則の遵守により構造データの変更の度にコピーが作られる。メモリ競合の解消とコピーの効率化が実現上の重要な課題となる。
- 履歴概念がないので、資源管理，データベース更新など履歴依存性を持つ処理をどう扱うかが大きな問題になる。

本稿では、これらの問題を共有資源に対する並列アクセスの問題として捉え、データ駆動計算原理の拡張及び Valid の処理機能の拡充を試みる。

先ず、Valid の特徴について概略を述

べ、次に、再帰データ構造の導入を行い、パターン照合によるデータ選択法，再帰データへの一般的な遅延評価法について論じる。最後に、データの抽象化，履歴依存性を扱うModuleの概念を提案し、Valid記述例を示す。

2. Validの特徴

Validは数値処理と記号処理を再帰概念という統一した枠組の中で捉えることを目的に設計された言語である。シンタクスはAlgol系言語を基調としながらセマンティクスはあくまで関数性を遵守するという設計思想に基づいており、以下の特徴を持っている。

(1) プログラム変数の概念はない。プログラムは式と定義のみからなる。定義には関数定義，マクロ定義，値定義，データ型定義等がある。

(2) 式或いは関数は複数の値を返すことを許す。特に値定義式では、

```
[x1,x2, ... ,xn] = <expression>
```

により複数の値に一度に名前を定義することができる。

(3) 各種定義式の集合であるブロック概念を持つ。ブロックは clause…end（または{…}）で定義される。ブロック自身も式であり、ブロックの生成する値はreturn式により示される。ブロック中ではAlgolのスコープルールに従う局所的な名前の定義が許される。ブロック内の式は；(セミコロン)で区切られるが、Algol系言語とは異なりデータ依存関係のみに基づき並列に評価される。

(4) 繰返しは全て再帰概念に基づいて記述する。従来型言語での繰返し構造はrecursive構造とfork-joinタイプのparallel構造に分けられる。recursive構造はさらにtail-recursiveとnon-tail-recursiveに分類されるが、Validではこれらの構造をそれぞれrecurrence式或いは関数の再帰呼出しにより記述する。recurrence式

```
for ( <value names> ) : <expressions>
do <expression including
                recur expression>
```

は無名の再帰関数を定義するものと解釈する。但し実行は効率向上のためループ制御で行われる。

parallel構造は次のようなparallel式

```
for each ( <value names> )
            in <expressions>
   do <parallel body>
   return <join expression>
```

により、parallel bodyの並列実行を陽に記述する。

(5) 値名，関数の定義にはデータ型の指定を許す。これによりコンパイル時に於けるデータ参照関係の整合性チェックを可能にする。

(6) 関数名を引数とする高階関数の定義を許す。このため関数名はデータ型を持つ値名として扱う。但し関数名に対する演算は認めない。

(例)

```
map: function(f:function,x:list)
      return (list)
 = if null(x) then nil
    else cons(f(car(x)),map(f,cdr(x)));
```

3. 再帰データ構造

従来の言語(ALGOL 68, PL/1, PASCAL等)では、referenceまたはpointer概念が言語構造の中に現われ、処理の中心的役割を担ってきた。reference概念を持つ言語では次のような問題が生じる。

(1) アドレスと内容の混乱を招きデータ型の整合性チェックが困難となり、プログラムの意味も不明瞭になる。この結果、プログラムの解釈・正当性の検証が著しく困難になる。

(2) dangling referenceの問題が生じ、変数のスコープルールがあいまいになる。

(3) データ構造を記述する場合、構造自身でなく、それがどうつくられるかを記述しなければならない。

これに対し再帰データ構造[9]はreference

概念ではなく再帰概念に基づいて、値の集合として構造を記述する。再帰データ構造を導入する利点は、

・プログラマはインプリメンテーションの詳細にとらわれることなく、データ構造の抽象的性質のみに注目できる，
・数学的形式化が容易でプログラムの意味は明瞭となり、正当性の検証が行い易い，

ことである。再帰データ構造の考え方は、メモリセル概念を排除した関数型言語のプログラミングスタイルを遵守するものといえる。データフローによる計算の観点からみると、先に提案した Lenient cons概念[6]によるリストのパイプライン処理，Lazy cons概念[7]による遅延評価，及び I-structure[8]を使った Stream処理は Leniency を持つ再帰データ構造に対する操作とその評価の問題として一般化して捉えることができる。

以下では Valid での再帰データ構造の定義・操作・評価法について述べる。

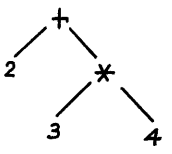
3.1 再帰データ構造の宣言と表現

再帰データ構造は Valid のデータ型定義 <type names> : type
= <type specification>
を使って定義することができる。
例えば hour,year : type = integer は hour, year が示す値のデータ型が integerであることを示している。

データ構造を構成する方法としては

(1) パーザによる自動生成

(例1)
```
a = 2 ;
b = 3 * 4 ;
c = a + b ;
```



(2) 生成関数を使う[9]

(例2)
```
a = const(2) ;
b = mult(3,4) ;
c = add(a,b) ;
```

(3) 型付きリストを使う

(例3)
```
a:const = 2 ;
b:term  = [3,'*,4] ;
c:expr  = [a,'+,b] ;
```

などの方法が考えられるが、コンパイル時の型の整合性チェックの容易さ,記号処理とのなじみ易さを考慮し、(3)の型付きリストを使う方法を採った。

図1-Aに、Valid に於ける数式のデータ型定義の例を示す。

```
-- Arithmetic expression type definition
aconst: type = integer | real ;
aident: type = identifier ;
aop1:   type = '+ | '- ;
aop2:   type = '* | '// ;
aop:    type = aop1 | aop2 ;
aterm:  type = aconst | aident |
        [ aexpr ] | [ aexpr,aop,aexpr ];
aexpr:  type = aterm | [ aop1,aexpr ];

-- Relational expression type definition
rop:    type = '< | '> | '<= | '>= |
               '= | '<> ;
rexpr:  type = [ aexpr,rop,aexpr ];

-- Logical expression type definition
lconst: type = 'true | 'false ;
lident: type = identifier ;
lop:    type = 'and | 'or | 'xor ;
lterm:  type = lconst | lident | rexpr |
        [ lexpr ] | [ lexpr,lop,lexpr ];
lexpr:  type = lterm | [ 'not,lexpr ];
```

図1-A 数式のデータ型定義

注)"|" は"または"を、"'" は quote を表わす。

上の例で、aconst は integer か real をとるデータ型であり、また aop1 は単項演算子(+または-)を、aop2 は2項演算子(*または/)をとるデータ型である。ここでは単項演算子を含まないものは aterm, 含むものは aexpr というデータ型として定義している。

型変換を一意的にするため、a:type=a や a:type=b; b:type=a のような循環定義は許さず、各データ型には優先順位を定める。(a:type=b の順序関係は a>b) 但しデータ型の集合は次の関係を満たす半順序集合である。

∀x,y,z ∈ T (set of types)

(1) x < y and y < z -> x < z
(2) x < y -> not (y < x)
(3) not (x < x)

データ型の集合の下界(e.g. integer,char, list)はシステム既知のデータ型でなければならない。

先の数式のデータ型定義の例では、図1-Bのような半順序関係が成立つ。

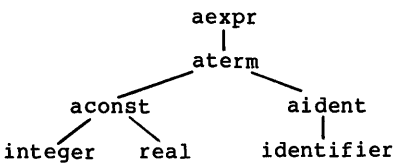


図 1-B 数式データ型の半順序関係

これにより、integer -> aconst -> aterm -> aexpr のように一意に型変換が行われる。(これを自明な変換と呼ぶ。)

再帰データ構造は図 1-C に示すように型付きリストの形で内部表現される。

```
    not (2+3*x > -5 and y=4)
                  |
                  v
[ 'not,
  [ [ [2,'+,[3,'*,x]:aterm]:aterm,
      '>,['-,5]:aexpr  ]:rexpr,
    'and,[y,'=,4]:rexpr ]:lterm ]:lexpr
```

図 1-C 再帰データ構造の内部表現

ここで [ ] はリストの constructor である。[a,b] は list(a,b), [a.b] は cons(a,b) を意味する。 [a.[b.[c.NIL]]] は [a,b,c] と書いてもよい。

データ構造の各成分には構造の生成過程を保存するため、内部ではデータ型名を識別するタグを付加する。但し型変換が自明な場合は省略してよい。

3.2 再帰データ構造の選択法

再帰データ構造の各成分は match 式

```
match <expression> with
    <matching pattern> -> <expression>
  {;<matching pattern> -> <expression>}*
            {;others  -> <expression>}
end
```

を使ってパターン照合により選択することができる。図 2 に match 式を用いて記述した数式の微分の例を示す。<matching pattern> で定義される変数(パターン照合の結果値名,例えばx)は形式パラメータに相当し、その有効範囲は → (右矢印) 以降 ; (セミコロン) 迄である。複数の <matching pattern> と照合がとれた場合、いずれか 1 つが選択され → 以降の <expression> が実行される。 others は don't care を意味する。

```
-- Symbolic expression differentiation
deriv: function (expr:aexpr,id:aident)
       return (aexpr)
=match expr with
  x:aconst -> 0 ;
  x:aident -> if eq(x,id) then 1 else 0;
  [x:aexpr]-> deriv(x,id);
  [x:aexpr,op:aop,y:aexpr] ->
   match op with
    opl:aopl ->
     omitl([deriv(x,id),opl,deriv(y,id)]);
    '*  ->omitl(
          [omitm([x,'*,deriv(y,id)]),'+,
           omitm([deriv(x,id),'*,y])]);
    '// ->omitd(
          [omitl(
           [omitm([deriv(x,id),'*,y]),'-,
            omitm([x,'*,deriv(y,id)])]),'//,
           omitm([y,'*,y])])
   end;
  [opl:aopl,x:aexpr] ->
        omitu([opl,deriv(x,id)])
 end;

omitl: function
         ([x:aexpr,op:aopl,y:aexpr]:aexpr)
       return (aexpr)
= if zerop(x) then omitu([op,y])
  elsif zerop(y) then x
  else  [x,op,y];

omitm: function
         ([x:aexpr,op:aop2,y:aexpr]:aexpr)
       return (aexpr)
= case
    or(zerop(x),zerop(y))-> 0 ;
    eq(x,1)-> y;  eq(y,1)-> x;
    others -> match y with
      ['-,p:aexpr]-> [omitu(['-,x]),op,p];
      others      -> [x,op,y]
              end
  end;

omitu: function([op:aopl,x:aexpr]:aexpr)
       return (aexpr)
 = if eq(op,'+) then x
   else match x with
          ['-,y:aexpr] -> y;
          others       -> ['-,y]
        end;
```

図 2 match 式を用いた数式微分

関数 deriv の match 式に於いて、最初の行は expr が定数 (aconst) なら 0 を返すことを、2 番目の行は expr が名前 (aident) なら if eq(x,id) then 1 else 0 の値を返すことを述べている。omit1, omitm, omitd, omitu はそれぞれ加減算,乗算,除算,単項演算 (eg. e+0, e*1) の簡約化を行う関数である。

パターン照合によるデータの選択はこの他に、関数の引数の授受, 値定義

に於いても行われる。複数の値定義 $[x_1, x_2, \cdots x_n] = \langle expression \rangle$ での使用法は以下の通り。

```
[p,q]=[a,b,c,d]   ==> p=a ;  q=b
[p.q]=[a,b,c,d]   ==> p=a ;  q=[b,c,d]
[p,q]=[a,[b,c]]   ==> p=a ;  q=[b,c]
```

3.3 再帰データ構造の評価

本節では、再帰データ構造に対する効率的操作と一般的な遅延評価について述べる。構造データ操作の問題は再帰データ構造に対する再帰関数の適用という観点で見るとわかり易い。

再帰データ構造は一般に<tag,item1,… itemn>(構造セル)を成分として持つ木構造で表現される。再帰データ構造操作のパイプライン化と遅延評価は、Leniencyを持つ構造の概念の導入により達成される。Leniencyとは各itemの値が求まっていなくても先に構造セルを生成し、これを使用する関数へ渡すことを意味する。この場合 itemは i) value (scalar / structure) ii) recipe (<delayed expression/function code, colour>) を値として持つ。各itemには未定義 / 定義/ recipe を示すタグが付される。itemを参照する関数は値が未定義であればwaitし、recipeであればそのcodeを評価する要求を出し値が求まる迄waitする。

図3-Aに遅延評価機構を使って n個迄の素数を求めるプログラムの例を示す。ここでは再帰データ構造は型指定のないリストで表現されている。遅延評価される関数は Lazyで指定される。Lazy指定がある場合は構造 itemとしてrecipeが作り出される。

図3-Bは型付きリストを用いた再帰データ構造の遅延評価の例である。(nこを計算するプログラム)

同じ構造を繰返し入力する場合は、計算の無駄を省くため出来るだけ共有した方が良い。これは値名の定義を一度行ってそれを複数の箇所で参照することに対応する。

```
-- Generating prime number sequence
primenumber: function(n:integer)
                 return (list)
= sieve(1,n,integerseqfrom(2));

-- Generate infinite integer sequence
intseqfrom: function(m:integer)
                 return (list)
= [m . lazy intseqfrom(m+1)];

-- Remove multiples of i-th prime number
sieve: function(i,n:integer,[p.q]:list)
         return (list)
= if i=n then [p.nil]
    else [p.sieve(i+1,n,deleteval(p,q))];

deleteval: function(x:integer,[p.q]:list)
             return (list)
= if remainder(p,x)=0
  then deleteval(x,q)
  else [p . lazy deleteval(x,q)];
```

図 3-A 型指定のないリストを用いた遅延評価

```
sequence: type = [ value . sequence ];

sum_to_n: function(n:integer)
            return (integer)
= {seq=intseqfrom(1);
   for (sum,[val . rest]):(0,seq) do
    if val=n then sum+val
             else recur(sum+val,rest)};

intseqfrom: function(m:integer)
              return (sequence)
= [m . lazy intseqfrom(m+1)] ;
```

図 3-B 型付きリストを用いた遅延評価

図3-Cは fibonacci 関数の定義と fibo(5)の再帰的な計算の過程を示したものである。通常、このように同一入力に対しても毎回同種の計算が行われ、関数の値の共有は行われない。

Valid では関数の値の共有を行うため図3-dに示すように、remind 属性を持つ関数の定義を許す。計算の過程は遅延評価と同様になるが、これを実現するには関数名と引数の値をキィとして連想アクセスする機構が必要である。

```
-- Definition of fibonacci function
fibo: function(n:integer) return (integer)
= if n=1 or n=0 then 1
                else fibo(n-1)+fibo(n-2);

   Computation of fibo(5)
fibo(5)
 ⇒ fibo(4)+fibo(3)
 ⇒ (fibo(3)+fibo(2))+(fibo(2)+fibo(1))
 ⇒ ((fibo(2)+fibo(1))+(fibo(1)+fibo(0)))
    + ((fibo(1)+fibo(0))+(1+1))
                 :
 ⇒ 8
```

図3-C 通常の fibonacci 関数の計算

```
-- Definition of fibonacci function
--     with 'remind' attribute
rfibo: remind function(n:integer)
       return (integer)
= if n=1 or n=0 then 1
       else rfibo(n-1)+rfibo(n-2);

   Computation of rfibo(5)
rfibo(5) ⇒ rfibo(4) + rfibo(3)
  rfibo(4) ⇒ rfibo(3) + rfibo(2)
    rfibo(3) ⇒ rfibo(2) + rfibo(1)
      rfibo(2) ⇒ rfibo(1) + rfibo(0)
               ⇒ 2
    rfibo(3) ⇒ 3
  rfibo(4) ⇒ 5
rfibo(5) ⇒ 8
```

図 3-d remind functionによるfibonacci計算

4. Module

入出力制御，資源管理，データベース管理などの応用では、過去の履歴に依存して処理が行われる。このような履歴依存性のある処理は、図4-1に示すように純関数とループの組合せで実現される履歴依存型関数(history sensitive function)で表わされる。実際には複数の関数f_i ($i=1,\cdots,n$)が内部状態を共用しうるのでこれらの関数群と状態をまとめて一つのインスタンスとして取扱わねばならない。Validではこれを Moduleと呼ばれるもので扱う。Moduleは図4-2に示すように、

- その存在がModule外にも知らされている exported functions
- Module内で局所的に使用される local functions
- recurrence式により過去の状態を保持している module body
- 上記要素間でデータの授受を行うためのキューである channel(10)

から構成される。

以下、Validの Moduleの概要を例を用いて説明し、次に Moduleのインスタンス管理について述べる。

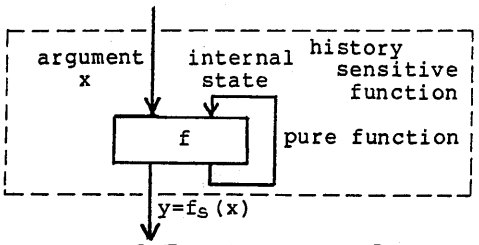


図 4-1 履歴依存性を持つ関数

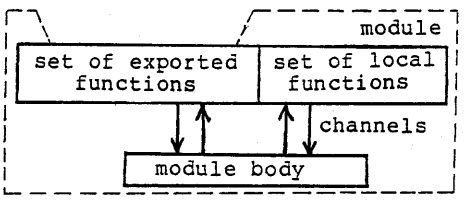


図 4-2 Moduleの構成

4.1 Module の概要

図4-3はスタックを Moduleで実現したものである。予約語 export の後には

```
(a) Definition of module 'stack'
stack:module()
       export (push,pop,add,sub,top)=
{pushstack:channel=(integer);
 pushed:   channel=(signal);
 popstack: channel=(signal);
 popped:   channel=(integer);
 topstack: channel=(signal);
 got:      channel=(integer);

 push:function(val:integer)
      return (signal)
 ={<<pushstack!(val); return pushed?>>};
 pop: function() return (integer)
 ={<<popstack!('sig); return popped?>>};
 add: function() return (signal)
 ={a=pop()+pop(); return push(a)};
 sub: function() return (signal)
 ={<<a=pop(); b=pop()>>; push(a-b)};
 top: function() return (integer)
 ={<<topstack!('sig); return got?>>};

 for (stack):(nil) do
  alt
   pushstack? ->
     {pushed!('done);
      recur [pushstack.stack]};
   popstack?  ->
     {[head.tail]=stack;
      popped!(head); recur tail};
   topstack?  ->
     {[head.*]=stack;
      got!(head); recur stack}
  end};

(b) Reference to module 'stack'
-- Expression evaluation of (3 + (4 - 5))
eval_expr:module import(stack)=
{x=create stack;
 <<x$push(3); x$push(4);
   x$push(5); x$sub; x$add; p=x$pop>>;
 return p};
```

図 4-3 module を用いたスタック定義

module 外から呼出すことが出来る関数名を示す。module を参照する時は、図4-3(b)に示すように module 名を import し、そのインスタンスを生成する。

関数呼出しは、

```
<instance name> $ <function name>
```

で表わす。channel は FIFO のキューであり、宣言された型のデータを扱う。channel への入力と出力はそれぞれ

```
<channel name> ! ( <actual arguments> )
<channel name> ?
```

で表現する。

非決定性を実現するため alt 式

```
<alternative expression>
::= alt <alternative element>
    { ; <alternative element> }*  end
<alternative element>
::= <channel name> ?
    { when <boolean expression> }
    -> <expression>
```

を新たに導入した。

alt 式では、空でない channel に対応する式が非決定的に選択，評価される。when 節がある場合、更に channel から試験的にデータを読出して論理式を評価し、真の時に選択の候補とする。

履歴依存性のある関数では、関数の評価順序をデータ依存性のみで定めることが出来ない。そのため、逐次的な実行を陽に指定する seq part を導入する。

```
<sequence part>
::= <<  <part element>
    { ; <part element> }*  >>
 <part element>
::= <expression> | <value definition>
```

seq part 内では、要素は並びの順に逐次的に評価される。

module body は recurrence 式を使って状態の保持と更新を行う。図4-3のスタックの例では、関数 push により整数型データが channel pushstack に出力されると、module body 内でそれが読出され、その値を元の stack の値に cons し、stack を更新する。

4.2 Module のインスタンス管理

図4-3の例では、各 stack のインスタンス（図4-3(b)中の x）は、それを生成した関数と module body 内でのみ用いられた。従ってインスタンスの存在期間は、インスタンス名によるアクセスの期間と同じである。しかし、資源管理など、インスタンスが複数の module から共有される場合には、インスタンスの生成／消滅と各インスタンス名によるアクセス期間とを分離して考える必要がある。このため Valid では、インスタンスの生成時にグローバルな名前を付けることを許す。(図4-4(b)参照) 他の module はグローバルな名前を用いて access 式を評価することにより、そのインスタンスを使用することが出来る。(図4-4(c)参照) インスタンス名を channel や他の module の関数への引数と

```
(a) Definition of module 'resource'
resource:module(n0:integer)
            export (get,free)=
{right:  type=integer;
 getreq: channel=(integer);
 got:    channel=(right);
 freereq:channel=(right);
 freed:  channel=(signal);

 get: function(m:integer) return (right)
 ={<<getreq!(m); return got?>>};
 free:function(m:right) return (signal)
 ={<<freereq!(m); return freed?>>};

 for (n):(n0) do
  alt
   getreq? when getreq<=n ->
     {got!('ok); recur(n-getreq)};
   freereq? ->
     {freed!('ok); recur(n+freereq)}
  end};

(b) Creation of named instance
-- Creation of the instance of 'resource'
def_res:module import(resource)
={x=create resource(I0) named 'res_name;
   ... };

(c) Access to named instance
-- Usage of the instance of 'resource'
use_res:module import(resource)
={<<x=access resource named 'res_name;
    r=x$get(2);
    -- use 2 units of 'res_name;
    x$free(r)>>; ... };
```

図4-4 module のインスタンスの共有

することは許さない。これは、インスタンス名が状態として保存されると、インスタンス名のスコープとアクセス可能期間が一致しなくなるからである。

図4-4(a)は、n_0個の均一な資源を管理するmoduleの定義例である。

図4-4(b)では、$n_0=10$としてインスタンスを生成し、named 'res_nameによりインスタンス名のグローバル宣言を行っている。図4-4(c)はそれを使用する例を示している。

以上述べたようにValidでは、module概念を導入することにより、データの抽象化と履歴依存性のある関数を実現している。

5. あとがき

再帰概念を基礎に置くValidは、数値処理や記号処理の並列演算の記述に適した関数型言語であり、データフローマシン上で効率良く実行しうるように設計されている。本稿では、関数型言語の課題となっている、構造データの効率的操作と履歴依存性の問題をとりあげ、主に記号処理機能に着目しValidの言語仕様の拡充を試みた。

先ず、再帰データ構造に対する型付きリストを用いた宣言法、パターン照合によるデータ選択法、要求駆動に基づく遅延評価の記述法について提案した。次に抽象データ構造や履歴依存性を扱うmoduleの概念を導入し、moduleの構成、記述法について述べた。

データフローマシン上での実現法については述べなかったが、ここで扱った問題は共有資源に対する並列アクセスという共通の問題を含んでおり、統一した機構を用いて実現することが可能である。これについては稿を改めて発表したい。最後に我々の研究に対して日頃叱咤激励して下さる山下紘一第一研究室長ならびに御討論頂いたデータフローマシン研究グループの諸氏に感謝します。

参考文献

(1) McCarthy J., "Recursive Functions of Symbolic Expression and Their Computation by Machine, Comm. ACM, 3(4) 1960, PP 184-195

(2) Ackerman W.B. and Dennis J.B., "VAL-A Value Oriented Algorithmic Language Preliminary Reference Manual", MIT/LCS/TR-218 Jun. 1979.

(3) Arvind, Gostelow K.P. and Plouffe W., "An Asynchronous Programming Language and Computing Machine," TR-114a Univ. of California, Irvine Dec 1978

(4) 雨宮, 尾内, "データフローマシン用高級言語VALIDについて", 電子計算機研究会 EC82-9, May 28 1982

(5) 雨宮, 長谷川, "データフローマシンと関数型プログラミング" 信学会'81 オートマトンと言語研究会 AL81-84, パターン認識と学習研究会 PRL81-63

(6) M.Amamiya, R.Hasegawa, O.Nakamura and H.Mikami, "A List Processing Oriented Data Flow Machine Architecture" submitted to AFIPS NCC 1982

(7) 長谷川, 雨宮 "データフローマシン上でのLazy evaluationの実現について" 情処学会第24回(昭57前期)全国大会

(8) Arvind and R.E.Thomas, "I-structures: an efficient data type for functional language", MIT/LCS/TM-178 Sept. 1980

(9) C.A.R. Hoare, "Recursive Data Structures" Stanford AI Lab. STAN-CS-73-400, Oct 1973.

(10) C.A.R. Hoare, "Communicating Sequential Process" Comm. ACM 21(8) Aug. 1978