作用的言語ＶＵＬＣＡＮの処理モデルについて

山野紘一，木谷有一
（日立システム開発研究所）

1　はじめに

作用的プログラミング言語に関心が寄せられているのは，主に次の２つの理由によるものである。１つは、参照透明性により、通常の言語のような副次的効果がないので、プログラミングの誤りを少なくすると共に、形式的な検証が容易に実現できる可能性がある。

オ２には、副次的効果がないために、式を、並行に評価することができ、高度な並行計算が可能となる。

しかし、作用的言語のもつ利点にもかかわらず、いままで広く受け入れられてこなかったのは、通常のマシーンでの効率が悪いのが大きな要因であると思われるが、最近のハードウェア技術の発達によって解決されつつある。

作用的言語を効率よく実働化する方式として、データフローとリダクションが提案されている。リダクション方式において、ラムダ算法と同じ意味をもつ結合子論理を用いたＳ－Ｋリダクションが Turner[2] によって提案された。この方式は、束縛変数をもつプログラムから、抽象化操作によって、変数を消去した結合子表現へと翻訳し、その表現を、グラフ操作で実行するものである。グラフ操作は、最左端の最も外側から計算を実行するもので，正規順序における遅延評価を自然な形において実現することができる。

Turner 方式には、次のような問題点がある。

(1)翻訳に時間がかかる。これは、一時に一変数の抽象化操作を行なうことを基本にしていることによる。また、プログラムがネストしているときには、多くのパスが必要である。

(2)結合子表現は、ソースプログラムの形からかけ離れたものとなる。このために、実行途中の情報を出すなどのデバッグ機能の実現が難かしい。

(3)実行は、一変数に対して作用する小さな単位に分割されているので、実行ステップ数が大きくなる。

本稿では、上記の問題点を解決する一方法を提案する。それは、一時に、多数の変数の抽象化操作を可能とするもので、結合子の対象として組を取り扱うことができるようにする。この結果、ソースプログラムに近い形の結合子表現（Ｖコード）を得ることができ、実行単位を、大きくすることが可能となる。

この方式をＶＵＬＣＡＮ[10]（Value oriented Language for reliable CALculatioN）に適用した場合について述べる。ＶＵＬＣＡＮは、並行プログラミングと関数との融合を作用的構造の枠組の中で実現することを狙ったデータ型をもつ作用的言語である。

以下、２章において、ＶＵＬＣＡＮ言語の概要を述べ、３章に列をベースとした結合子表現Ｖコードの定義を与え、４，５章に、ＶＵＬＣＡＮプログラムの翻訳方式と実行評価モデルについて議論する。

2．言語の概要

2.1．データ型

プログラムの静的な検証を行なう上でデータ型の機構は大いに役立つ。作用的言語に対して、データ型を導入することは、記述の簡潔さを損なうという欠点も生じてくるが、プログラムの信頼性の向上の観点から、ＶＵＬＣＡＮは、次のようなデータ型定義の機能を持たせている。

▷基本型 ··· integer, real, boolean, character, string
▷列型 ··· seq(型)
▷直積型 ··· prod(フィールド名:型; ···)
▷直和型 ··· union(タグ名:型; ···)

通常の言語にあるポインタ型の代りに、次のように帰納的データ型を使うことができる。

```
type STACK == union(empty: null; elem: prod(value: real; rest: STACK))
```

2.2. 式と関数

VULCANで計算を定義する基本要素は、式と関数から構成する。代入文のように変数を介しての値の更新はできないが、計算の結果の値に、名前を付けることができる。たとえば、

```
y == fac : x
```

においては、関数facは対象xに作用し、結果の値にyという名前を付けている。

関数の定義は、次の形である。

```
func fac (N : integer -> integer)
== if eq : <N, 0> then 1
   else N * fac : (N - 1)
   endif endf
```

基本関数には、次のものがある。

▷算術関数 ··· add(+), sub(−), mul(*), div(/)など
▷論理関数 ··· eq, ne, gt, ge, lt, le, and, or, など
▷列構成子 ··· sel, apndl, apndr, hd, tl, conc, など
▷直積構成子 ··· sel, rep, など
▷直和構成子 ··· is など

式の構成子には、次のものを設けている。

▷選択 ···

```
if P then E1 else E2 endif
```

▷繰り返し ···

```
for v==E0 while P apply E1 yield E2 endfor
```

▷bracket ···

関数の基本操作は、その合成にあり、その記述形式として、次のような合成関数を用いることができる。

```
<u, v> == < x+y, x*(x+y) >
```

しかし、一般にこの形式のものだけでは、分かりにくくなるので、bracketというブロックを使うことができる。braの次には、入力を、ketの次には出力を示すものが記述でき、前の例は次のようになる。

```
bra <x, y>;
  u == x+y;
  v == x*u;
ket <u, v>;
```

```
1  frame IntegerSet;
2
3    type Set == process;
4      port Has, Insert : in  integer;
5      port Reply       : out boolean;
6
7      type IntSeq == seq(integer);
8      val Content:IntSeq == nil;
9      val Size:integer == 0;
10
11     func Search
12       (<C:IntSeq; S, N:integer>
13                    -> integer)==
14       for I:integer==0
15         while
16           and:<le:<I, S>,
17                ne:<sel<C, I>, N>>
18         apply bra <I>;
19                new I==I+1 ket <I>;
20         yield I
21         endfor
22       endf;
23
24     guard
25     //fill:Has =>
26       bra <> with N==Has?:<> endw;
27         I==Search:<Content, Size, N>
28       ket <> with
29         Reply!:(lt:<I, Size>) endw;
30
31     //fill:Insert =>
32       bra <Content, Size>
33         with N==Insert?:<> endw;
34         I==Search:<Content, Size, N>;
35         P==and:<eq:<I, Size>,
36                  lt:<Size, 100>>;
37         new Content ==
38           if P then
39             rep:<Content, Size, N>
40             else Content endif;
41         new Size ==
42           if P then Size+1
43                else Size   endif;
44       ket <Content, Size>;
45   endp;
46
47   act:Set
48
49 endframe IntegerSet.
```

図1. VULCANプログラム例

2.3. プロセス

VULCANは、並行プログラミングを可能とするために、プロセスの概念を導入している。プロセスは、データ型とし、act関数によって、具現化と起動を行ない、終了はdeact関数で指示する。起動から終了までのプロセス生存期間では、プロセス内に値が保持されている。

プロセス間通信の方式は、Hoare[3]のCSPにもとづいたものである。プロセスには端子があり、端子と端子とを結びつけたものを通信路という。通信路の設定は、connect関数で行ない、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。

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果す。通信路によって結合されたプロセス群はネットワークを形成する。

図1のプログラムは、Hoareの整数集合のプログラムをVULCANで記述したものである。この例において、端子Has, Insertは入力を示し、端子Replyは出力を示す。このような入出力は、braあるいはketの次に、with という句をつけて表わす。with句の中には、通信(入力関数?と出力関数!)のみを書くことができる。

このbracketは、合成関数の表現に通信を持たせたもので、通信付きのbracketと呼ぶ。また、guardは、非決定性機能を表わしている。

3. 結合子表現

3.1. Turnerの方式

結合子は、論理において変数が不必要であることを示すために、Schönfinkelとcurryによって発明され、ラムダ算法と同様に計算の理論の基礎を与えている。Turner[2]は、結合子を作用的言語の新しい実働化技法として定式化した。

結合子を活用する利点には、次のものがあげられている。

(1) 正規順序における遅延評価を容易に実現することができると共に、リダクションにおいて、グラフを用いることにより、部分式の共有化を図ることができる。

(2) 結合子表現は、翻訳において、式から変数を消去するために、実行評価で環境を維持することが不要となる。

以下、Turner方式の概要を示す。一般に、多重引数の関数は、一変数の関数に変換して取り扱う。このことはCurry化と呼ばれる。

関数 $f(x_1, x_2, \cdots, x_n)$は、
$((\cdots(f'(x_1)x_2)\cdots)x_n)$
$\equiv f'(x_1)(x_2)\cdots(x_n)$ の形式にCurry化して考える。

基本的な結合子S, K, Iは次の方程式で定義される。

$$S\,f\,g\,x = f\,x\,(g\,x)$$
$$K\,x\,y = x$$
$$I\,x = x$$

式からすべての変数を取り除くために抽象化操作を行なう。この抽象化は、結合子S, K, Iを使って、次の規則で定義される。ここで、$[x]E$は、式Eからxを抽象化するという。

[抽象化規則]

$[x](E_1 E_2) \Rightarrow S([x]E_1)([x]E_2)$

$[x]\,x \Rightarrow I$

$[x]\,y \Rightarrow K\,y$ (ここでyはx以外の変数)

次に、例を用いて抽象化した結果を示す。

```
func f (x, y : integer→integer)
  == (x+1) * (y-1) end f
                    ・・・(1)
```

(1)の関数をCurry化したものを
mul(add x 1)(sub y 1)
とする。

```
[x]([y](mul(add x 1)(sub y 1)))
⇒S(S(KS)(S(KK)(S(K mul)
  S(S(K add)I(K 1))))) (
  K(S(S(K sub)I))(K1))
                    ・・・・(2)
```

この例のように、2つの変数x, yをもつときには、y, xの順に抽象化するので、原理的に2回のテキスト走査が必要である。また、ソースプログラムの大きさに比べて、かなり冗長な結合子表現となる。

そこで、次のような結合子B, Cが導入される。

$$Bfgx = f(gx)$$
$$Cfgx = fxg$$

結合子表現の冗長性を減らすための最適化規則が与えられている。

[抽象化の最適化規則]

$$S(KE_1)(KE_2) \Rightarrow K(E_1E_2) \quad \cdots(a)$$
$$S(KE_1)I \Rightarrow E_1 \quad \cdots(b)$$
$$S(KE_1)E_2 \Rightarrow BE_1E_2 \quad \cdots(c)$$
$$SE_1(KE_2) \Rightarrow CE_1E_2 \quad \cdots(d)$$

この最適化規則を(2)に適用すると次のようになる。

```
S(BS(BK(Bmul(Cadd 1)
 )))(K(Csub 1))  ・・・・(3)
```

なお、最適化規則の(c), (d)の適用は、それより上記の規則が適用できないときに初めて適用できるので、この最適化には、基本的に3回の走査を必要とする。

再帰的定義に対しては、Y結合子

$$Yf = f(Yf)$$

を用いる。

3.2 Vコード

VULCANの原則は、関数fが対象$\langle x_1, x_2, \cdots, x_n \rangle$に作用し、その結果として値を得るという概念にもとづいている。これは、次のように記述される。

$$f : \langle x_1, x_2, \cdots, x_n \rangle$$

ここで、引数$x_1, x_2, \cdots, x_n$は異なる型であるから、引数の直積と考えられる。

さらに、これを一般化して、

$$\langle f, g, \cdots, h \rangle : \langle x_1, x_2, \cdots x_n \rangle$$
$$= \langle f : \langle x_1, x_2, \cdots, x_n \rangle,$$
$$g : \langle x_1, x_2, \cdots, x_n \rangle, \cdots$$
$$h : \langle x_1, x_2, \cdots, x_n \rangle \rangle$$

が言語機能として与えられている。

さて、この$\langle x_1, x_2, \cdots, x_n \rangle$なるn組(n-tuple)に対応する結合子を導入する必要があるが、$\langle\ \rangle$の記号をそのまま組結合子と見なすことにする。

▷組結合子(T結合子)

$$\langle f, g, \cdots h \rangle \langle x_1, x_2, \cdots x_n \rangle$$
$$= \langle f \langle x_1, x_2, \cdots x_n \rangle,$$
$$g \langle x_1, x_2, \cdots x_n \rangle, \cdots$$
$$h \langle x_1, x_2, \cdots x_n \rangle \rangle$$

従って、言語上の$\langle\ \rangle$に対応する結合子は、組結合子$\langle\ \rangle$と考える。結合子表現の中で、組結合子は1つの対象である。Turnerの場合、結合子を適用するために、多重引数をもつ関数に対しては、Curry化をして考える必要があったが、組を単位とすることにより、Curry化を行なわないで結合子を適用する。

以下、VULCANにおける中間語としての結合子表現をVコードと呼ぶ。組$\langle x_1, x_2, \cdots, x_n \rangle \equiv \overline{x}$には、3.1節に述べた結合子はS, K, I, B, Cの定義は、$\overline{x}$を一つの対象とみなしてそのまま成り立つ。

組に対して、新たに必要となる結合子は、射影結合子J_iである。この結合子のラムダ算法との関係については Barendregt[1], Burge[9] を参照されたい。

▷射影結合子(J結合子)

$$J_i \langle x_1, x_2, \cdots, x_n \rangle = x_i$$
$$(1 \leq i \leq n)$$

次に、∇ULCANソースプログラムを∇コードに変換する抽象化規則を列挙する。

[∇コードの抽象化規則]

$[\bar{x}](E_1\ E_2) \Rightarrow S([\bar{x}]E_1)([\bar{x}]E_2)$

$[\bar{x}]\bar{z} \Rightarrow I$

$[\bar{x}]y \Rightarrow Ky \quad (1 \le i \le n \text{ に対して. } x_i \neq y)$

$[\bar{x}]x_i \Rightarrow J_i \quad (1 \le i \le n)$

$[\bar{x}]\langle y_1, y_2, \cdots, y_m\rangle \Rightarrow \langle[\bar{x}]y_1, [\bar{x}]y_2, \cdots, [\bar{x}]y_m\rangle$

3.1節に述べた関数fを、∇ULCANの本来の形式である作用形式に直すと、その本体は次のようになる。

$mul:\langle add:\langle x,1\rangle, sub\langle y,1\rangle\rangle$

この関数の抽象化は、

$[\langle x,y\rangle](mul\langle add\langle x,1\rangle, sub\langle y,1\rangle\rangle)$

$\Rightarrow S([kx,y]mul)([kx,y]\langle add\langle x,1\rangle, sub\langle y,1\rangle\rangle)$

$\Rightarrow S(K\,mul)(\langle[kx,y]add\langle x,1\rangle, [kx,y]sub\langle y,1\rangle\rangle)$

$\Rightarrow S(K\,mul)(S(K\,add\langle J_1,K1\rangle)(S(K\,sub\langle J_2,K1\rangle)) \quad \cdots(2')$

となる。この抽象化操作は原理的に、一回のテキスト走査でできる。すなわち、複数の変数を一時に抽象化の対象として行くことができる。

3.1節で述べた最適化規則は、そのまま適用でき、その結果は、次のものである。

$B\,mul\langle B\,add\langle J_1,K1\rangle, B\,sub\langle J_2,K1\rangle\rangle \quad \cdots(3')$

この例が示すように、一般に、Turnerの方式に比べて、∇コードは、短かくなり、テキストの走査回数も少なくてすむ。

4. 翻訳

4.1. 方式

∇ULCANプログラムの処理の流れを図2に示す。ソースプログラムを∇コードに変換することを翻訳といい、∇コードを評価することをリダクションという。

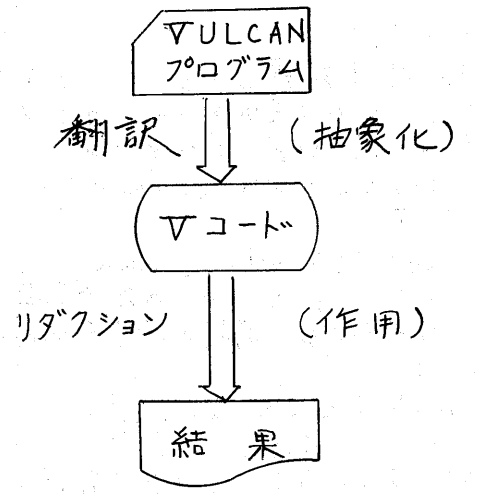


図2. 処理の流れ

ここで、∇ULCANの言語機能のうち、主な機能の∇コードについて述べる。

▷if式

if P then E_1 else E_2 endif

の∇コードは、

$Cond\langle P, E_1, E_2\rangle$

の形式とする。なお、Condは結合子でリダクションにおいて、Pを最初に評価し、その結果に従って、次の変換を行なう。

$Cond\langle true, E_1, E_2\rangle \Rightarrow E_1$

$Cond\langle false, E_1, E_2\rangle \Rightarrow E_2$

▷繰り返し式

for $v := E_0$ while P

apply E_1 yield E_2 endfor

まず、変数vは局所的変数であるから消去し、Y結合子を使うと次のように変換できる。

$h = (Y\,([\langle h\rangle]\,([\langle v\rangle]\,(\mathrm{Cond} \langle P,$
$h E_1, E_2 \rangle)))\,E_0$
$\Rightarrow (Y\,(B\,(B\,\mathrm{cond}) \langle K\,[\langle v\rangle]\,P,$
$C\,B\,[\langle v\rangle]\,E_1,\ K\,[\langle v\rangle]\,E_2 \rangle))\,E_0$

▷ bracket

bracketは、通信をもつものとそうでないものの２種があるが、そのVコードは、同じ形式のものとする。

bra <x> with E_1 endw ;
E_2
ket <y> with E_3 endw ;

ここで、E_1, E_3 は、入出力関数よりなる式である。特別の結合子bracketを設けて、

bracket << x , E_1 >, E_2
< y , E_3 >>

の形とする。リダクションにおいては、< x, E_1 > が最初に評価され、その結果が E_2 に作用され、最後に < y, E_3 > の評価を行なう。bracketの特徴はその入口の時点で、入力パラメタ x をも評価することである。

▷ 関数定義

関数名も、変数と同様に抽象化によって消去することが可能であるが、VULCANでは、関数定義は、独立なVコードに翻訳する。

たとえば、3.1節の例において、関数 f を呼び出すVコードが次のものであるとする。

B f < J_1 , J_2 > < 2 , 3 >
⇒ f < J_1 < 2 , 3 >, J_2 < 2 , 3 >>

この時点で、f のVコードをロードすればよい。グラフリダクションを行なうときには、f の位置に f のVコードを接続すればよい。

⇒ Bmul < Badd < J_1 , K1 >,
Bsub < J_2 , K1 >> < J_1
< 2 , 3 >, J_2 < 2 , 3 >>

関数定義が再帰性をもつ場合でグラフリダクションを行なうときは、Y結合子をそのまま用いないで、リダクションの途中で関数定義に出会えば、その都度Vコードをロードしながら接続してゆき、すでにロードされたものがあれば、その関数の先頭へのポインタを作成すれば、結果的にサイクリックグラフが作成でき、記憶域の節約となる。

▷ 大域的変数

VULCANの基本構成単位であるプロセスに直接属する大域的変数は、プロセス本体に記述されるbracketの中でのみ更新することができる。その参照に対しては、get という内部基本関数で値を取り出し、大域的変数に対する更新（new 関数が伴なう。）において、update という内部基本関数で更新する。

すなわち、大域的変数に対しては、端子への入出力と同様な取り扱いを行なう。

4.2 データ構造

データ型として、基本型以外に、列、直積、直和があるが、型のチェックは翻訳時に実施し、Vコードの段階ではすべて組（tuple）に変換する。

たとえば、

```
type date == prod(
        year  : integer ;
        month : integer ;
        day   : integer )
val   x : date ;
```

のとき、date のデータ構造は、

< ⟦year⟧, ⟦month⟧, ⟦day⟧ >

とする。ここで、⟦ ⟧ は、値を示す。

値へのアクセス date . day は、基本関数 sel を使って、

sel : < x , 3 >

とする。組の物理的な表現は、

T_3 | 1983 | 6 | 24 である。

4.3. 最適化

▷テキストの最適化

これは、通常の命令的言語で実施される最適化技法で　作用的言語においても、共通式の削除、ループ不変式の移動、翻訳時評価（定数のたたみ込みなど）等が考えられる。しかし、作用的言語の場合、命令的言語におけるような制御構造がないこと、副次的効果がないことにより、制御フロー及びデータフロー解析に対する処理が必要でなく、原理的には式のパターン照合によって行なうことができる。

（3＋x）＊（3＋x）

に対して、y＝＝3＋x　とおくと、

[<x>](([<y>]mul <y, y>)(3＋x))

⇒ B(B mul <J_1, J_1>)

(B add <K_3, J_1>)

となり（3＋x）の評価結果は共有される。

▷▽コードの最適化

Turner の方式では、最悪の場合長さnのプログラムは、n^2のオーダーの長さの結合子表現に変換される[7]。この処理にかかる翻訳時間はかなりのものである。▽ULCANの場合、言語構造自体が作用的な形式となっているのと、関数名を消去しないので、テキストの最適化の走査を除いて、原理的には一回の走査で抽象化を行なうことができる。

4.4 デバッグの支援

▽コードは、ソースプログラムの形に近く、関数呼び出しの場合、

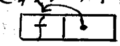
B f <引　数><――>

の形式になっているので、関数呼び出しが起った時で中断し、その引数を出力することができるし、その時点で、強制的に引数の評価を実施するようなデバッグ支援機能を設けることが容易にできる。また、関数定義は、モニタによって管理することにすれば、その実行トレースを記録することができる。

5. 評価方式

5.1 逐次評価

結合子を用いたときの逐次評価は、Turner によって提案されている。この方法は、結合子表現をTurnerグラフと呼ぶ2進木で表わす。そのグラフを走査するS－Kリダクションマシーンは、正規順序評価を制御するため、リダクションスタック（プッシュダウンオートマトン）を使用する。

評価は、常に、2進木の左側から実行するために、自然な形で、遅延評価が実現されている。また、共通式は、ポインタで共有されるので、一回評価するだけですむ。再帰的定義に対してはY結合子表現　Y f は  なる形のグラフに変換するだけでよい。

S－Kリダクションマシーンをマイクロコードで実現したものにSKIM[4]が開発されている。

▽ULCANの▽コードに対しても、▽コードをn進木に変換すれば、Turnerと同様の方法で逐次評価することが可能である。

5.2. 並行評価

作用的言語の高度な並行評価を実現するアプローチとして、データフローマシーン方式とグラフリダクション方式が数多く研究されている[6,8]。

以下、▽コードを正規順序に並行評価するモデルについて検討する。

3.1節に述べた関数fの▽コード（2）のグラフ表現は、図3のようになる。関数fに対して、次の作用があったとすると、

・・・f：<3, 4>・・・

このときのリダクションの過程は、次

に示すものとなる。

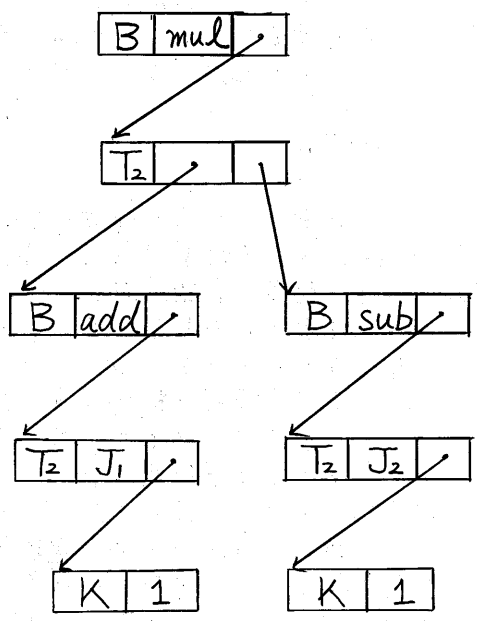


図3　∇グラフ

B mul < B add < J1, K1 >, B sub < J2, K1 >> <3,4> ・・・(4)
⇒ mul < B add < J1, K1 > <3,4>, B sub < J2, K1 > <3,4> > ・・・(5)
⇒ mul < add < J1 <3,4>, K1 <3,4> >, Sub < J2 <3,4>, K1 <3,4> > > ・・・(6)
⇒ mul < add <3,4>, sub <4,1> > ・・・(7)
⇒ mul <4,3> ⇒ 12

この評価の並行モデルを∇ULCANのプロセス機能を用いて表わす。2.3節に述べたように、∇ULCANでは、プロセスの生成／起動及び停止／消滅，プロセス間の端子の接続ができる。図3の∇グラフの各節には、ユニークな識別子を付加し、木のつながりを示すポインタは、この識別子で表現する。∇グラフは、このようなパケットの集まり（列構造）として保持されるものとする。

評価機は、この∇コードの列を取り出しながら、それぞれの結合子に対応するプロセスを生成して行く。たとえば、B結合子のプロセスは，図4（∇ULCANプログラムは図5）に示すものである。

B結合子（Bプロセス）の役割は、

B fgx ⇒ f(gx)

なる変換を実行することである。

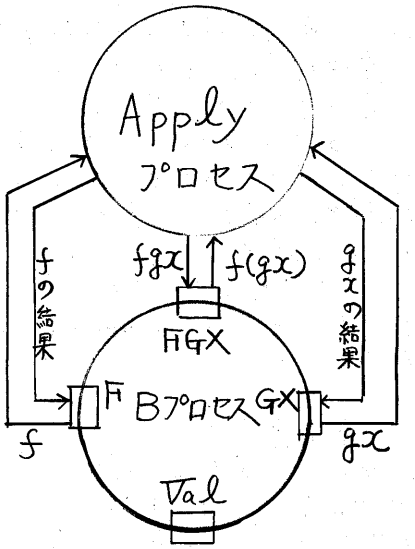


図4　Bプロセス

FGX端子より、fgxを受信し、F，GX端子より、Apply（評価機の主体）プロセスに、f、gxの評価を依頼する。Bプロセスは、評価結果をf(gx)とし、再びApplyプロセスに送信する。この時点で、fに対応するプロセスを生成し、そのプロセスの評価値を出力する端子とBプロセスの受信端子（∇al）の接続を行なう。

先の例（3）以下のリダクションについて説明する。

①Bプロセスは、入力として、

fgx・・・[mul, i2, <3,4>]

を受信する。ここで，[　]は列を

示すものとする。

② F端子への出力は、[mul]で、その評価結果は、mulである。

③ GX端子への出力は、[i_2,<3,4>]で、i_2に対応するTプロセスが生成される。そのTプロセスでは、(5)に対応して、[i_3,<3,4>]と[i_6,<3,4>]との処理が行なわれ、2つのBプロセスが生成される。

```
 1 frame Evaluator;
 2
 3   type Token    == string;
 4   type Exp      == seq(Token);
 5   type Prim     == union(N:null;
 6                          B:boolean;
 7                          I:integer;
 8                          R:real;
 9                          C:char );
10   type PrimSeq == seq(Prim);
11
12   type Apply == process;
13     . . .
14
15   type Bcombinator == process;
16     port FGX : in  Exp out Appl;
17     port F   : out Exp in  Token;
18     port GX  : out Exp in  PrimSeq;
19     port Val : in  Primseq
20                out PrimSeq;
21     type Appl == prod(FF:Token;
22                       XX:PrimSeq);
23
24     guard
25     //fill:FGX =>
26       bra <> with E==FGX?:<> endw;
27         FV ==sel:<E,1>;
28         GXV==tlr:E
29       ket <> with  F!:FV;
30                    GX!:GXV endw;
31
32     //and:<fill:F, fill:GX> =>
33       bra <> with  FV== F?:<>;
34                    GXV==GX?:<> endw;
35         AP:Appl==<FV,GXV>
36       ket <> with FGX!:AP endw;
37
38     //fill:Val =>
39       bra <> with V==Val?:<> endw;
40       ket <> with Val!:V      endw;
41     endp;
42
43   type Mult == process;
44     port Aport,Bport :in  integer;
45     port Cport       :out integer;
46     guard
47     //and:<fill:Aport,
48            fill:Bport> =>
49       bra <> with
50         A==Aport?:<>;
51         B==Bport?:<> endw;
52         C == A*B
53       ket <> with Cport!:C endw;
54     endp;
55
56   act:Apply
57 endframe Evaluator.
```

図5 Bプロセスプログラム

④ 以下、同様の過程が繰り返され、Bプロセスの GX端子にTプロセスの作成結果情報が送され、次にApplyプロセスに、mulプロセスの作成を依頼し、mulプロセスを接続する。

以上のプロセスによるリダクション過程を図式的に示すと図6のようなプロセスネットワークが形成される。

ユーザ定義の関数や列に対する基本関数 conc は遅延評価とするが、他の基本関数、bracket, cond の判定に対しては、その引数の評価は、強制しなければならない。

上記の例を Turner の結合子で記述すると3.1節の(3)のようになり、add と sub の評価を並行に行なうこ

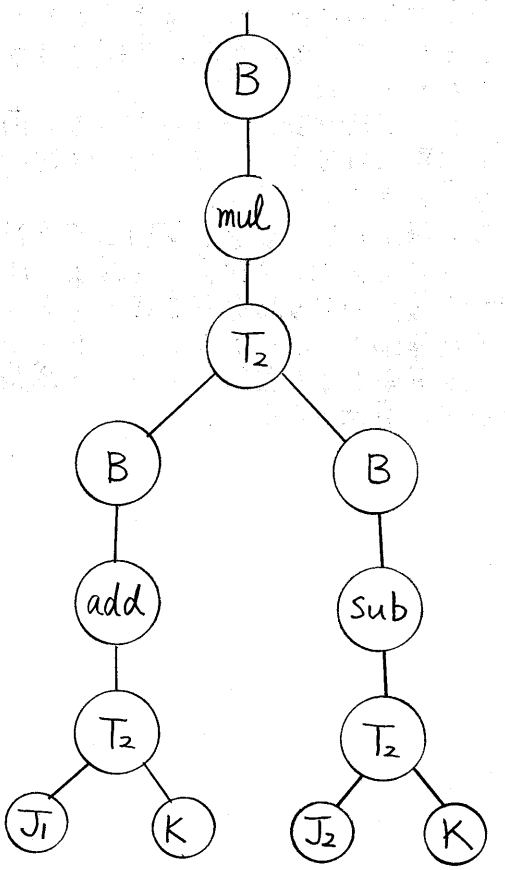


図6 プロセスネット

とは容易でない。それに比べると、並行評価において、引数を組で渡すことができるため、Tプロセスでは、その引数の評価を行なうプロセスの作成を容易に並行化することが可能となっている。

6. おわりに

作用的言語VULCANの処理モデルについて述べた。基本的本式は、中間語として、結合子表現を採用したものであるが、Turnerの方式とは異なって、関数の引数を1つの組(tuple)で表現することにし、Curry化を行なうことが特徴である。このようにすることによって、翻訳の速度は向上させることができると共に、中間語は、ソースプログラムに近い形にすることができ、デバッグ支援が容易に実現できることがわかった。

また、並行評価を行なう上でも、関数の引数に対するリダクションの並行度を高めることができた。

並行評価のモデルを、VULCANのプロセスを用いて記述した結果、作用的言語に対する結合子表現にもとづく並行評価モデルとして、メッセージ渡しを基本とする並行プロセスの機構が非常に有効である。

[参照文献]

1. Barendregt,H,P., The Lambda Calculus : Its Syntax and Semantics, North-Holland, 1981
2. Turner,D.A., A New Implementation Technique for applicative Languages, Software Practice and Experience, 9, pp.31-49, 1979
3. Hoare,C.A.R., Communicating Sequential Processes, CACM, 21, 8, pp.666-777, 1978
4. Clarke,T., P.Gladstone, C.Maclean, and A. Norman, SKIM - The S.K.I. Reduction Machine, Proc. 1980 LISP Conference, pp.128-135
5. Hughes,R.J.M., Super-Combinators, Proc. 1982 Symposium on LISP and Functional Programming, pp.1-10
6. Darlington,J. and M.Reeve, ALICE A Multi-processor Reduction Machine for The Parallel Evaluation of Applicative Language, Proc. 1981 Conference Functional Programming Languages and Computer Architecture, pp.65-75
7. Burton,F.W., A Linear Space Translation of Functional Programs to Turner Combinators, Information Processing Letters, 14, 5, pp.201-204, 1982
8. Treleven,P.C., D.R.Brownbridge and R.P.Hopkins, Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, 14, 1, pp.93-143, 1982
9. Burge,W.H., Recursive Programming Techniques, Addison-Wesley, 1975
10. 山野、木谷、渡辺、並行プログラミングのための作用的通信機能、情報処理学会、ソフトウェア基礎論3-4, 1982