プログラミング言語　24－5
（1990．2．9）

表明付き項書き換え系によるストリームプログラミング

古賀信哉　布川博士　野口正一

東北大学電気通信研究所

あらまし

本稿では我々がすでに提案，実現している戦略の表明をもつ項書き換え系A－TRSを用いてトークンモデルに基づくストリーム（並行プロセス）の記述を行なう．

ストリームは項書き換え系のような，いわゆる関数型言語に順序関係を導入するものであり，プログラムの構造化手法としても重要である．さらに，ストリームを並行に動作するモジュール間のデータの流れとして捉えることで，プログラムの中に並列性を陽に表現することが可能となる．本稿で述べるプログラムはプロセスの動作をすべてリダクションでシミュレートしており，処理系に対して通信機能を加えるなどの変更をなんら施すことなく実行が可能であるという特長をもっている．

Stream Programing in Strategy Annotated Term Rewriting System

Shinya Koga, Hiroshi Nunokawa, Shoichi Noguchi

Research Institute of Electorical Communucation TOHOKU University

2-1-1 Katahira, Sendai 980, JAPAN

Abstract

We have proposed and implemented the strategy annotated term Rewriting System(A-TRS). In this paper, we show the *stream programming* based on *token model stream* as an example of concurrency modularization method in A-TRS.

We use rewriting rules to describe each process, and the connections between processes(*channels*) is also described by rewriting rules. The strategy annotation is used to pass an data tokens to the processes connected by the *channel*. We can use a A-TRS reducer to simulate the processes communication.

This shows that A-TRS is useful in describing and simulating the communication between processes by reduction.

1．はじめに

項書き換え系（Term Rewriting System; TRS）は関数型言語の計算モデルとしても用いられるが，それ自体をプログラミング言語とする見方もできる．この場合，TRS は記述や意味の把握が容易であるといった優れた特長を持つ．また，実現されていない関数については関数記号を定数とみなして可能な部分の計算を行なうため，プログラムのトップダウンな実現が可能である．

TRSの処理系を構築する際に，従来は正しさを重視して正規化戦略を採用するか，または実行効率を重く見て最内戦略を採るという，二者択一的な立場がとられてきた． われわれはこれに対し，TRSをプログラミング言語として有効に利用するためには単一の戦略を用いるのでは不十分であり，ユーザが恣意的に戦略を定められる必要があるという観点から，プログラム中に記述した表明によって戦略が規定される項書き換え系 A-TRSを提案しており，その処理系を定義および実現したことについてすでに報告している[1][2]．

本論文では，A-TRSを用いてトークン・モデルに基づくストリーム（並行プロセス）を記述する．ストリームは関数型言語に順序関係を導入するものであり，プログラムの構造化手法としても重要である．さらにストリームを並行に動作するモジュール間のデータの流れとして捉えることで，プログラムの中に並列性を陽に表現することが可能となる．また，この論文で紹介するプログラムはプロセスの動作をすべてリダクションでシミュレートしており，処理系に対して通信機能を加えるなどの変更をなんら施すことなく実行が可能である．

2．項書き換え系と戦略

2．1項書き換え系

項書き換え系（Term Rewriting System; TRS）は，項から項への書換えを定める書き換え規則の有限集合である．一般に項（term）は関数の適用を表しており，従って各々の書き換え規則を関数定義とみることもできる． このとき，TRSは一つの関数型プログラムとして扱われる．以下に，項の集合Termの定義を示す．

［定義］　（項，Term，Term0）
項の集合Termは，変数の集合Var，関数記号の集合Func，及びFuncの部分集合である定数の集合Constより以下の帰納的定義によって定まる．
（１）　$*x \in Var$ の時 $*x \in Term$
（２）　$a() \in Const$ の時 $a() \in Term$
（３）　$f \in Func$ であり，且つ $t_1, \cdots, t_n \in Term$ の時 $f(t_1, \cdots, t_n) \in Term$

（２）と（３）からのみ構成される，変数を含まない項を基底項（ground term）といい，その集合をTerm0で表わす．
※　定義（２）の定数a()は単にaと略記することがある．また通常用いられる関数記号については，中置表現を用いている（例えば"+"について"+(a,b)"ではなく"(a + b)"と表記する）．

TRSにおける計算は，与えられた項に書き換え規則を適用して書き換える操作，すなわち項の簡約（リダクション）である．書き換え規則が適用可能であるとは，書き換え規則を$P \triangleright Q$，対象となる項をMとしたとき，Mの部分項M'に対し$M' \equiv P\theta$となるような置換θが存在することである．このとき，MについてM'を$Q\theta$で置き換えて新たな項Nを得ることを，MがNにリダクションされたという．また､M'をMのリデックスと呼ぶ．どの書き換え規則も適用できない項，つまりリデックスを持たない項を正規項と呼ぶが，与えられた項 Mのリダクションを行って得られる正規項（Mの正規形）がTRSでの計算結果である．

項が複数のリデックスを持つ場合，どのリデックスを書き換えるかによって最終的な結果は異なる可能性がある．この問題に対し，与えられた項についてその正規形が一意であること，つまり計算結果の一意性を保証するためにTRSが備えるべき性質が，合流性（Church-Rosser性）である．合流性について，線形かつ重なりの無いTRSはこれを満たすことが知られており，本論文では以下TRSとして 基本的に線形かつ重なりの無いものを扱う．

2．2書き換え戦略

書き換えようとする項が複数個のリデックスを持つ場合に，そのうちのどれを書き換えるべきか指定する方法を書き換え戦略と呼ぶ． TRSが合流性を満たしていれば，正規項が存在する場合にその一意性は保証される．しかしながら，正規項が存在しても戦略によっては有限回の書き換えで求められない場合があり，

また求められるとしてもそれに要する書き換えの回数は戦略によって異なってくる．

任意の項Mのリデックスについて，M'が他のリデックスの部分項になっていないときM'をMの最外リデックス（outermost redex）と呼び，またM"に他のリデックスが出現していないときこれを最内リデックス（innermost redex）と呼ぶ．一般にM'およびM"，即ち最外リデックスおよび最内リデックスは，ともに複数個存在する．

書き換え戦略のうち，最外リデックスのいずれかを選んで書き換えを行なうものを最外戦略，最内リデックスのいずれかを選択するものを最内戦略と呼ぶ．特に，本論文で扱うものとして最も左側の最内リデックス1個を書き換える最左最内戦略と，すべての最外リデックスを書き換える並行最外戦略とがある．

正規項が存在するならばそれを必ず求めることのできる書き換え戦略を正規化戦略と呼ぶが，本論文で扱っている重なりのない線形のTRSについては，並行最外戦略は正規化戦略であることが知られている．

3．表明付き項書き換え系A-TRS

3．1 A-TRSにおける書き換え

A-TRSとはプログラムである書き換え規則に対し，戦略を規定するための 表明をつけられたTRSのことである．表明（Annotation）は規則の右辺または左辺の部分項に対して与えられ（ただし左辺の場合は変数に限る．），最外並行戦略を指定するlazy表明（"[]"）と最左最内戦略を指定する eager表明（"{}"）とがある．

A-TRSプログラムの実行順序，即ち書き換え戦略は，正規化戦略である並行最外戦略を原則としている．書き換えようとする項の部分項が表明項の場合には，これを指定された戦略により正規形まで書き換えたのち全体の書き換えを行なう．プログラムの例として，階乗を計算するfactorial，および二つの引数を取りその対を返すpairに関する規則を図1に示す．

```
factorial(*x)
    ▷ if(eq(*x,0) ,1
                 ,(*x × factorial(*x - 1)))

pair(*x,*y) ▷ (*x . *y)
```

（実行例）pair(factorial[3],factorial(5))

```
→ pair(6,factorial(5))
→ (6 . factorial(5))
…→ (6 . 120)
```

図1 A-TRSプログラム（1）

3．2 A-TRSのリデューサ

まず，A-TRSで用いる項の集合A-Termを定める．A-TermはTermを部分集合としている．

［定義］ （表明付き項，ATerm，ATerm0）

A-TRSで用いる項の集合ATermは，変数の集合Var，関数記号の集合Func，及びFuncの部分集合である定数の集合Constより以下の帰納的定義によって定まる．

（1） $*x \in Var$ の時 $*x$，$[*x]$，$\{*x\} \in ATerm$

（2） $a() \in Const$ の時 $a()$，$a[]$，$a\{\} \in ATerm$

（3） $f \in Func$ であり，且つ $at_1,\cdots,at_n \in ATerm$ の時 $f(at_1,\cdots,at_n)$，$f[at_1,\cdots,at_n]$，$f\{at_1,\cdots,at_n\} \in ATerm$

[]，{}を表明と呼び，表明を含む項を表明付きの項という．特に[*x]，a[]，f[…]をlazyな表明を持つ項（lazy annotated term），{*x}，a{}，f{…}をeagerな表明を持つ項（eager annotated term）という．

（1）を用いずに構成された，変数を含まない基底項の集合をATerm0で表わす．

A-TRSの処理系は表明項の処理を行なう関数Arpと，基本的に並行最外戦略をとりながら，Arpを使うことで表明に従った書き換えを実現する関数Ratとから成る．Arpは表明項に対し，表明の種類に応じたリデューサを割り当てる．このうち，eagerな表明に対応するものとして最左最内戦略を実現するRat-Rliが用意されておりまたlazyな表明に対してはRat自身が割り当てられる．

［定義］ （Arp ；annotation replacement）

Arp: Aterm0→Term0

```
Arp(at)
  = a, Rat(a), Rat-Rli(a)
      where
        at≡a, a[], a{} resp.
    f(Arp(at1),…,Arp(atn))
      where
        at≡f(at1,…,atn)
    Rat(f(Arp(at1),…,Arp(atn)))
      where
        at≡f[at1,…,atn]
    Rat-Rli(f(Arp(at1),…,Arp(atn)))
      where
        at≡f{at1,…,atn}
```

ここで，置換 $\theta=\{at_1/x_1,\cdots,at_m/x_m\}$ について
$Arp(\theta)=\{Arp(at_1)/x_1,\cdots,Arp(at_m)/x_m\}$
と定義する．

［定義］ (A-TRSのリデューサRat
; Reduce Annotated Term)

```
Rat: ATerm0→Term0

Rat(at)
  = if nf(at)
      then at
      else if match(at)
             then
               Rat(Arp(AQ·Arp(θ)))
                   where  match(at)=θ,
                          at=AP·θ,
                          AP ▷ AQ
      else  Rat(f(Rat-one(at1),···,Rat-one(atn)))
                   where  at≡f(at1,···,atn)
                          または at≡f[at1,···,atn]
                          または at≡f{at1,···,atn}


Rat-one(at)
  = if nf(at)
      then at
      else if match(at)
             then
               Arp(AQ·Arp(θ))
                   where  match(at)=θ,
                          at=AP·θ,
                          AP ▷ AQ
      else  f(Rat-one(at1),···,Rat-one(atn))
                   where  at≡f(at1,···,atn)
                          または at≡f[at1,···,atn]
                          または at≡f{at1,···,atn}
```

※上の定義において，matchは at (∈ATerm0) と適用しようとする規則の左辺から表明を外したものとの間での置換を求め，その後左辺に付けられていた表明を求めた置換に付けて返すものとする． 例えば f({*x},[*y]) ▷ … という規則があるとき，match(f(g(A),h(B))) の結果として {g{A}/x, h[B]/y} が返される．

［定義］ (Rat-Rli)

```
Rat-Rli: ATerm0→Term0

Rat-Rli(at)
  = if nf(at)
      then at
      else if at≡f(c1,···,cm)
             then Rat-Rli(Rat-one(at))
                    where ci∈AConst

      else  Rat-Rli(Rat-one(f(Rat-Rli(at1),
                              ···,Rat-Rli(atn))))
                 where  at≡f(at1,···,atn)
                        または at≡f[at1,···,atn]
                        または at≡f{at1,···,atn}
```

３．３ Lisp関数の呼び出し機能

前節までは，A-TRSに関する基本的な事柄の紹介を行なった．本節ではA-TRSの応用的な側面，すなわち組み込み関数について述べる．

プログラミング言語A-TRSは，組み込みの基本関数をまったく持たない．言い換えれば，すべてをA-TRSのプログラムとして記述する必要がある． この問題の解決策として Lisp関数の呼び出し機能を新たに加え，導入すべき組み込み関数は Lispによって実現するという手法をとった． 適切な組み込み関数を定めて処理系に導入することをしなかったのは， プログラム作成者が必要な組み込み関数を追加しようとする際に 処理系の変更を行なう必要がない， といった点で呼び出し機能を設ける方が有利だと考えたからである．なおA-TRSからの呼び出し相手としてLispを用いたのは，A-TRSの処理系（リデューサ）が現在 Lispによって実現されていることから， Lispを利用しやすい環境にあるためである．

Lisp関数の呼び出し機能を用いた例として，"-"，"×"，"eq0"という３つの組み込み関数を導入して階乗関数factorialを定義した．これを以下に示す．

```
factorial([*x])
    ▷ if(eq0(*x) ,1
               ,(*x × factorial(*x - 1)))

 [*x] - [*y] ▷ (MINUS *x *y)

 [*x] × [*y] ▷ (TIMES *x *y)

 eq0([*x]) ▷ (COND ((ZEROP *x) 'true)
                   (t 'false))

 if(*cond, *then, *else)
   ▷ if1([*cond], *then, *else)
  if1(true, *x, *y) ▷ *x
  if1(false, *x, *y) ▷ *y
```

書き換え例

```
f(1)
→ if(eq0(1) ,1
          ,(1 × factorial(1 - 1)))
→ if1(eq0[1], 1, (1 × factorial(1 - 1)))
→ if1(false, 1, (1 × factorial(1 - 1)))
→ (1 × factorial(1 - 1))
        → if(eq0(0) ,1
                   ,(0 × factorial(0 - 1))
        → if1(eq0[0], 1, (1 × factorial(0 - 1)))
        → if1(true, 1, (1 × factorial(0 - 1)))
        → 1
→ (TIMES 1 1)
→ 1
```

上に示した組み込み関数は規則の右辺がＳ式で記述されており，これらのＳ式はすべてLispの処理系で処理される．Lisp関数を呼び出すためには A-TRSのリデューサに対し拡張を施す必要がある．呼び出し機能を持つように拡張されたA-TRSのリデューサ，すなわちRatの定義を以下に示す．

［定義］　(Lisp関数の呼び出し機能を持つRat)

```
Rat: ATerm0→Term0

 Rat(at)
   = if nf(at)
       then  t
       else if  match(at)
              if isLisp(AQ)
                then
                  Rat(term(evalLisp(AQ·
                               s-exp(Arp(θ)))))
                else
                  Rat(Arp(AQ·Arp(θ)))
                  where  match(at)=θ,
                         at=AP·θ,
                         AP ▷ AQ
       else  Rat(f(Rat-one(at1),···,Rat-one(atn)))
                  where  at≡f(at1,···,atn)
                         at≡f[at1,···,atn]
                         at≡f{at1,···,atn}
```

A-TRSに加えられたLisp関数の呼び出し機能を使うことで，プログラマは必要な組み込み関数を自由に導入できる．組み込み関数を変更したり，また新たに導入する場合にも処理系の変更はいっさい行なう必要がい．

```
fact([*x]) ▷ (FACT *x)
```

実際，上のように階乗関数のfact自体をLispで実現しても構わない．このとき，factに関するこの規則は A-TRSとLispとのインタフェースを記述していると見ることができる． つまり，A-TRS上のfactがやるべきことはLisp上のFACTに引数を評価した後に渡すことであるが，この事を左辺の表明が表わしているのである．表明に対する評価により，FACTには必要な情報だけが渡される．表明のこのような効果は，４章で取り上げるストリームの記述にも関わってくる．

4． A-TRSによるストリーム・プログラミング

4．1ストリーム処理

ストリームは流れを表わす概念であり，関数型言語に順序関係を導入するものである．ストリームを扱うことによって，例えば繰り返し制御が必要な処理を逐次的に処理が加えられるべき要素の並びに対する操作として抽象化し，処理の類似した部分を捉えてプログラムの部品化を進めることが容易になる．

```
NatGen(*x) ▷ next(*x,NatGen(*x + 1))

FactPair(next(*x,*st))
          ▷ next(pair(*x,fact(*x)),FactPair(*st))

Reduce(*num,next(*x,*st))
          ▷ if(eq0(*num) ,*x
                         ,Reduce((*num - 1),*st))
```

図３．１　ストリーム・プログラム（１）

```
Reduce(1,FactPair(NatGen(1)))

→ Reduce(1,FactPair(next(1    ···①
                           ,NatGen(1 + 1)})))
→ Reduce(1,next(pair(1,fact(1))   ···②
                  ,FactPair(NatGen(1 + 1))))
→ if(eq0(1) ,pair(1,fact(1))
             ,Reduce((1 - 1)
                        ,FactPair(NatGen(1 + 1))))
···→ Reduce((1 - 1),FactPair(NatGen(1 + 1)))
→ Reduce(0,FactPair(next((1 + 1)    ···①'
                           ,NatGen((1 + 1) + 1))))
→ Reduce(0,next(pair((1 + 1),fact(1 + 1))   ···②'
                  ,FactPair(NatGen((1 + 1) + 1))))
→ if(eq0(0) ,pair((1 + 1),fact(1 + 1))
             ,Reduce(FactPair(NatGen((1 + 1) + 1))))
→ pair((1 + 1),fact(1 + 1))
→ ((1 + 1) . fact(1 + 1)) → (2 . 2)
```

図３．２　書換え例

図３．１は，ストリーム・プログラムの例である．このプログラムは自然数のストリームを生成するNatGen，ストリームを受け取り，各要素の階乗と要素自身の対からなるストリームへの変換を行なうFactPair，及び指定された番数の要素を取り出すReduceの定義からなっている．このプログラムの実行例を図３．２に示す．ここでストリームは無限リストであって，その要素は必要とされた時点で漸次具体化されていくものである．このような無限リスト構造を扱うには最内戦略を用いることができないため，図３．２の例では最外戦略によって実行を行なっている．

ストリームはまた，モジュールの間のデータの流れとして捉えることもできる．例えば図３．２の書換え例において①の"FactPair(next(1,NatGen(1 + 1)))"は，モジュールFactPairが モジュールNatGenの出力した自然数 1を受け取った状況を表わしていると解釈される．しかし，①'～②'についてはそのような解釈をすることができない．つまり本来モジュールの中で評価が完了し，正規形（データ・トークン）として出力されるべき項が未評価のまま出力されているのである．モジュールが未評価の項を出力したのでは，その項がエラーを含んでいた場合にエラーは出力先で生起することになり，デバッグが困難になる，といった弊害がある．このように，ストリームを扱うには最内戦略でも，また単なる最外戦略を用いたのでも不十分である．この問題を解決するには出力されるべき項に表明を付ければよい．そのような考えに基づいて図３．２のプログラムを書き換えたものを，以下に示す．

```
NatGen([*x]) ▷ next(*x,NatGen(*x + 1))

FactPair(next(*x,*st))
        ▷ next(pair[*x,fact[*x]],FactPair(*st))

Reduce(*num,next(*x,*st))
        ▷ if(eq0(*num) ,*x
                       ,Reduce((*num - 1),*st))
```

図３．３　ストリーム・プログラム（２）

```
Reduce(1,FactPair(NatGen(1)))

→ Reduce(1,FactPair(next(1   ・・・①
                   ,NatGen(1 + 1))))
→ Reduce(1,next(pair[1,fact[1]]
              ,FactPair(NatGen(1 + 1))))
・・・→ Reduce(1,next((1 . 1)   ・・・②
              ,FactPair(NatGen(1 + 1))))
→ if(eq(1,0) ,next((1 . 1)
          ,Reduce((1 - 1)
                 ,FactPair(NatGen(1 + 1))))
```

図３．４　書換え例

上のA-TRSプログラムでは，モジュールにデータ（正規形）として渡されるべき項に表明が付けられている．すなわちNatGenの規則で左辺に表明があるのは，このモジュールがストリームを生成するために，核となる自然数を受け取ることを表わしており，またFactPairの規則で右辺の表明は，FactPairが受け取った自然数とその階乗との対をデータとして出力することに対応している．従って，図３．３のプログラムを図的に表現するならば下の図４のようになる．

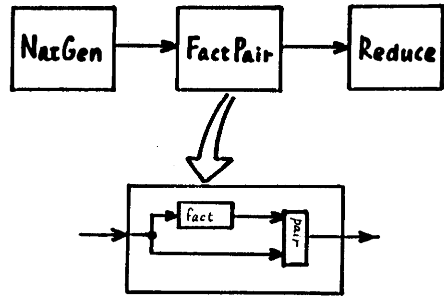

図4

4．２ストリームと並行プロセス

最内戦略を持つ関数型言語においては，従来遅延CONS及び半遅延CONSをアド・ホックに導入することでストリームの実現がなされてきた　．半遅延CONSとは，cdr部の評価を行なわずcarのみを評価するようなリスト構成子を用いるものであり，例えばSchemeでは遅延評価を実現するためにdelayとforceという演算子が導入されている．

これに対し，A-TRSの評価戦略は 基本的に最外戦略であり，遅延CONSが自然に実現されている．また，表明を用いることで半遅延CONSの実現も容易に行なうことができる．つまり，A-TRSは きわめて自然にストリームを扱うことができる言語である．

遅延CONS：　cons(*x,*y) ▷ (*x . *y)

半遅延CONS：　cons(*x,*y) ▷ ([*x] . *y)

ここまでは，ストリームを表わすのにリスト状のデータ構造を用いてきた．実際，これまで例示してきたA-TRSのプログラムは nextという構成子によって結合されたデータ構造を生成し，変換あるいは消滅させるものであった．これに対し，ストリームをモジュール間のデータの流れとして実現することもできる．データ構造として表現されたストリームをストラクチャ・モデル，データ・トークンの流れとして表わされたものをトークン・モデルと呼ぶ[3]．トークン・モデルの代表的なものとしてはデータフロー・モデルが挙げられる．ストリームをトークン・モデルで記述した

場合には，同じプログラム構造の中をデータが流れるためにメモリ効率が向上する，またパイプライン的な処理を行なうことができるので処理速度も向上するといった，プログラム実行上の利点がある．

さらに，ストリームではなくモジュールを主体とする記述を行なえば，ストリームはモジュールの間で受け渡されるデータ，すなわちモジュール間の通信でやり取りされるメッセージとして現れ，いわゆる並行プロセスのプログラムとなる．例えば HoareのCSPや，CSPに基づいて設計されたプログラミング言語Occamなどは，このようにしてストリームを扱った計算モデルと見ることができる[3][5]．

4．3 A-TRSを用いた並行プロセスの記述

並行プロセスを記述するための言語は，通信機能をプリミティブとして与えられているのが普通である．これらの言語の目的が，モジュール同士がいかに通信を行ないながら協調して処理を進めていくのか記述することにあるのを考えれば，この事は全く妥当であろう．

しかしながら，A-TRSの評価戦略と，TRSの持つリダクションによる計算という性質を利用すれば，処理系に通信機能を導入しなくとも並行プロセスのプログラムを実行できる．

以下の議論では，A-TRSを用いた並行プロセスの記述について（1）並行プロセスに対するモデル，（2）モジュール間通信のモデル，（3）記述の例，（4）記述で用いる構文，（5）A-TRSの評価戦略が果たす役割，という順序で話を進める．

（1）モデルの特徴づけ

並行プロセスの処理において，各モジュールの動作の調整（例えば通信を行なう際の同期）を司る機構には，言語によって異なるいくつかのモデルが与えられている．ここではそのような調整機構として，一つの並行プロセスを構成するモジュール群に対し，その管理を行なう関数を用意した．各モジュール間での同期，及び通信のためのデータの受渡しはこの関数の働きによって実現される．このモデルの概念図を図5に示す．

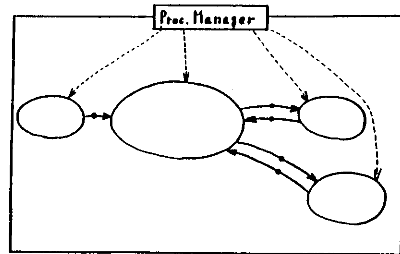

図5 平行プロセスのモデル

（2）プログラムの形式

並行プロセスをA-TRSで記述する際のスタイルは，Occamに似たものにした．従って，プログラムを書くうえではCSPに似たモデルを念頭におくことになる．このモデルには

（a）通信を行なおうとするモジュールは，相手の準備が整い，データの受渡しが完了するまでは次の動作に移れない．（通信方式は同期式）

（b）プログラムの実行中に，動的にモジュールが生成されることはない．（プロセス・ダイナミクスが静的）

（c）モジュール間の通信は1対1．（1対多，あるいは多対多の通信は基本通信としない）

といった制約がある反面，これによって通信の構図が捉えやすいという特長を持つ．また，通信相手の指定はモジュール間に通信路（チャネル）を設け，チャネルの名前を明示することで行なう．

ここに述べたモデルではモジュール間を流れるデータはチャネルを通り，またモジュール間の同期もチャネルが実現すると想定しているが，プログラムの実行にあたってはチャネルの働きを（1）で述べたプロセス・マネージャによってシミュレートする．言い換えれば，ここで述べたモデルに基づいて設計されたプログラムに対し，チャネルの機能を（1）のモデルで解釈することで実行するのである．

このような段階を踏むことにより，通信機能を持たない処理系で並行プロセスの処理をシミュレートできる．また，モジュールの記述とプロセス・マネージャの記述は独立しており，従ってモジュールの記述においては実際にチャネルがあると思えばよく，プロセス・マネージャの存在を大きく意識する必要はない．

（３）記述例

以下に，記述の具体例を示す．この例は４．１で取り上げたプログラムに対応するものであるが，ここでは各モジュールに対し，ストリームを受け取る関数ではなくチャネルを通じて他のモジュールとデータをやり取りするものとして記述を行なう．つまり，４．１の二つのプログラムがストラクチャ・モデルに基づいているのに対し，以下のプログラムはトークン・モデルに基づく記述である．

(A) モジュールの定義

（ａ）NatGen

```
NatGen(0,(*x)) ▷ out(ch1(*x),NatGen(1,(*x)))
NatGen(1,(*x)):done
              ▷ NatGen(0,([*x + 1]))
```

（ｂ）FactPair

```
FactPair(0,()) ▷ in(ch1(),FactPair(1,()))
FactPair(1,()):receive(*x)
          ▷ out(ch2(pair[*x,fact[*x]])
                ,FactPair(2,()))
FactPair(2,()):done ▷ FactPair(0,())
```

（ｃ）Reduce

```
Reduce(0,(*num)) ▷ in(ch2(),Reduce(1,(*num)))
Reduce(1,(*num)):receive(*y)
                 ▷ if[eq0(*num)
                      ,End(*x)
                      ,Reduce(0,[*num - 1])]
```

(B) プロセス・マネージャの定義

```
Proc(out(ch1(*x),*mod1),in(ch1(),*mod2),*mod3)
      ▷ Proc(*mod1:done,*mod2:receive(*x),*mod3)

Proc(*mod1,out(ch2(*x),*mod2),in(ch2(),*mod3))
      ▷ Proc(*mod1,*mod2:done,*mod3:receive(*x))

Proc(*mod1,*mod2,End(*x))
   ▷ *x
```

（４）記述の構文

並行プロセスを記述する場合に，ここでは３つの構文要素を用いている．一つはモジュールの本体を表わすものであり，他の二つはチャネルを介した通信の状況を表わすものである．

（ａ）モジュール本体

モジュール名（遷移入力，（状態変数リスト））

（例）NatGen(1,(*x))

（ｂ）通信要求

要求名（チャネル名（データ）
，モジュール本体）

（例）out(ch1(*x),NatGen(1,(*x)))

（ｃ）メッセージ受理

モジュール本体：メッセージ

（例）NatGen(1,(*x)):done

モジュール本体の構文で，遷移入力はモジュールが行なう一連の逐次的な動作を，正しい順序で行なうためのものである．（例えば，モジュールFactPairはチャネル１からデータを受け取る前にこのデータを使って計算される値を送出することはできない．）

（ｂ）の通信要求は，名前を用いて指定されるチャネルに対し，入力または出力を要求するものである．

（ｃ）のメッセージ受理は，通信を要求したモジュールに対しチャネル（実際にはプロセス・マネージャ）からメッセージが与えられた状況を表わす．メッセージには，送信の完了を示す"done"と，到着したデータを知らせる"receive(ﾃﾞｰﾀ)"の２種類がある． 通信を要求したモジュールはチャネルからメッセージが与えられるまで次の動作に移ることができず，これにより同期が実現される．

記述の具体的な説明のために，FactPairの動作をOccamの構文で書いたものと 記述例で与えたものとの対応を図６に示す．

```
WHILE TRUE
  SEQ
    ch1 ? x                 ･･･①
    ch2 ! pair(x,fact(x)) : ･･･②
```

```
FactPair(0,()) ▷ in(ch1(),FactPair(1,()))
  ：初期状態から①への遷移

FactPair(1,()):receive(*x)
        ▷ out(ch2(pair[*x,fact[*x]])
              ,FactPair(2,()))
  ：①の終了後，②へ遷移

FactPair(2,()):done ▷ FactPair(0,())
  ：②の終了後，初期状態へ遷移
```

図６　FactPairの動作

（４）並行プロセスと評価戦略

モジュール間の同期通信をシミュレートしてプログラムを実行するためには，データの受渡しに関する規則を与える必要がある．ここで与えられる規則の集合がプロセス・マネージャの記述であり，この記述をTRSの書換え規則として使うことで，計算が実行される．

（３）に示したプロセス・マネージャの記述は，ch1を介したNatGenとFactPairの通信，ch2によるFactPairとReduceの通信，そしてReduceの終了によるプロセス全体の終了，のそれぞれに対応する３つの規則から成っている．プロセス・マネージャの役割は，互いに通信可能な一対のモジュールを見つけてデータの受渡しを行ない，そして同期のために中断していたそれらのモジュールの処理を再開させることであるが，この動作はそのまま並行最外戦略に対応する．つまり，まずトップレベルでリダクションを試み（通信可能なモジュール対の探索），失敗すれば引数を評価する（各モジュールへ制御を移す）．

また，並行最外戦略のみを用いたのでは，４．１で述べたのと同じ問題が生じる．つまり，モジュールが評価すべき項を未評価のまま出力するという事態が起こり得るが，これが表明項の評価によって回避される．結局，並行プロセスの動作を並行最外戦略と表明項の評価による値渡しとによってシミュレートしているのであり，並行プロセス，すなわちトークン・モデルのストリームを扱ううえでもA-TRSの評価戦略が有効であることを示している．

以下に，記述例で与えたプログラムの実行される様子（書き換え例）を示す．

```
Proc(NatGen(0, (1)),FactPair(0, ()),Reduce(0, (1)))
→ Proc(out(ch1(1),NatGen(1, (1)))
       ,in(ch1(),FactPair(1, ()))
       ,in(ch2(),Reduce(1, (1))))
→ Proc(NatGen(1, (1)):done
       ,FactPair(1, ()):receive(1))
       ,in(ch2(),Reduce(1, (1))))
→ Proc(NatGen(0, ([1 + 1]))
       ,out(ch2(pair[1,fact[1]])
           ,FactPair(2, ()))
       ,in(ch2(),Reduce(1, (1)))

···→ Proc(NatGen(0, (2))
          ,FactPair(2, ()):done
          ,Reduce(1, (1)):receive((1 . 1)))

→ Proc(out(ch1(2),NatGen(1, (2)))
        ,FactPair(0, ())
        ,if[eq0(1)
            ,End((1 . 1))
            ,Reduce(0, ([1 - 1]))
            ])

···→ Proc(out(ch1(2),NatGen(1, (2)))
          ,FactPair(0, ())
          ,in(ch2(),Reduce(1, (0))))
→ Proc(out(ch1(2),NatGen(1, (2)))
       ,in(ch1(),FactPair(1, ()))
       ,in(ch2(),Reduce(1, (0))))

→ Proc(NatGen(1, (2)):done
       ,FactPair(1, ()):receive(2)
       ,in(ch2(),Reduce(1, (0))))
→ Proc(NatGen(0, ([2 + 1]))
       ,out(ch2(pair[2,fact[2]])
                ,FactPair(2, ()))
       ,in(ch2(),Reduce(1, (0))))

···→ Proc(NatGen(0, (3))
          ,FactPair(2, ()):done
          ,Reduce(1, (0)):receive((2 . 2)))

→ Proc(out(ch1(3),NatGen(1, (3)))
       ,FactPair(0, ())
       ,if[eq0(0)
           ,End((2 . 2))
           ,Reduce(0, [0 - 1])
           ])

···→ Proc(out(ch1(3),NatGen(1, (3)))
          ,FactPair(0, ())
          ,End((2 . 2)))
→ (2 . 2)
```

4．4プログラムの構造化と並行プロセス

前節では，並行プロセスの動作をリダクションでシミュレートできる事を示した．4章の主な目的は，並行最外戦略と表明項の評価による値渡しの組合せという A-TRSの評価戦略が有効な領域の一つとして，ストリームを取り上げて議論することであった．そこで得られた結論として，ストラクチャ・モデルのストリームだけではなく，トークン・モデルのストリーム，すなわち並行プロセスを扱うのにも A-TRSの評価戦略が有効であると述べた．

並行プロセスのようにモジュールを主体として記述する場合には，データの一方向の流れにとどまらず，双方向の流れ（コルーチン）や複数の流れを同時に扱うことができる．従って，関数型言語として主従関係の明確な表現に偏りがちなTRSにおいて，並行プロセスを扱うことで表現の幅を広げることが期待できる．これに対しストラクチャ・モデルのストリームでは一方向の流れに偏る傾向があり，コルーチンなどを表現するのが難しいと思われる．

並行プロセスを導入するということは，プロセスを構成するモジュール群の各々を単位としたプログラムの構造化であるとも考えられる．本論文でトークン・モデルのストリームを扱ったのは，A-TRS に適したプログラミングの対象としてだけではなく，プログラムの構造化手法を与えるものとしても並行プロセスに興味があったからである．

5．まとめ

われわれがすでに提案している表明を持つ項書き換え系 A-TRSでは，表明を付けられた部分項に対する先行評価と並行最外戦略の組合せで書き換え戦略が決定される．このような戦略が適したプログラミングの例としてトークン・モデルのストリーム，すなわち並行プロセスの記述を取り上げ，A-TRSの持つ戦略でプロセスの動作（モジュール間の同期通信）をシミュレートできる事を示した．

参考文献

[1]布川博士，富樫敦，野口正一： 表明付き項書き換え系A-TRS
ーリダクション戦略の表明ー，信学技報，Vol.88，No.143，pp.77-86(COMP 88-43) (1988).

[2]古賀信哉，布川博士，野口正一： 戦略の表明を持つ項書き換え系A-TRSの実現と評価，情報処理研究会報告，Vol.89，No.12，89-PL-20 (1989).

[3]田中二郎： 関数型プログラムにおけるストリーム計算，情報処理，Vol.29，No.8，pp.836-844 (1988).

[4]Robert E. Filman, Danil P. Friedman： COORDINATED COMPUTING－Tools and Techniques for Distributed Software－，McGraw-Hill (1984).
邦訳：雨宮真人，尾内理紀夫，高橋直久 共訳，マグロウヒルブック (1986).

[5]尾内理紀夫：Occamとトランスピュータ，共立出版 (1986).

[6]C.A.R.Hoare：CommunicatingSequentialProcesses, CACM，Vol.21，No.8，pp.666-667 (1978)

[7]山野紘一，木谷有一，渡辺担：並行プログラミングのための作用的通信機能，情報処理学会研究報告 (1982)

[8]二木厚吉，外山芳人：項書き換え型計算モデルとその応用，情報処理，Vol.24，No.2，pp.144-146(1983)