プログラミング言語　24－4
（1990. 2. 9）

プロセス間でのメッセージ交換を用いた項書き換え系のリダクション

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，関数的なプログラミング言語の抽象インタプリタである項書き換え系（ＴＲＳ）のリダクションを，各関数記号に割り当てられたプロセス間でのメッセージの交換を用いて実行する方式を述べる．

本方式はＴＲＳ上の各関数記号に対して一つのプロセスを割り当て，それらの間の通信で実行が進む方式であり，コンパイル方式であるが，最外戦略の実現に関して，全体を制御するためのランタイムルーチンの必要のない方法である．したがって，関数記号の増加に対して，単にその関数に対するプロセスを追加するのみでよく，システムへの関数記号の登録の手間がない．そのためプロセスが分散されて配置されている場合でも効率よく実行できる可能性がある．また，他のリダクションシステムへの応用も容易である．

本稿では，各関数に割り当てるべきプロセスの外部構造，プロセス間のメッセージの交換の仕方（リダクションのためのプロトコル），及びプロセスの内部構造を明確に定めることによって，ＴＲＳの新たなreducerを提案する．また，その正しさを示す．

The Reduction in Term Rewriting System based on Message Passing

Hiroshi Nunokawa and Shoichi NOGUCHI

Research Institute of Electorical Communcation TOHOKU University

2-1-1 Katahira, Sendai 980, JAPAN

Abstract

In this paper, we propose a new type of reducer for Term Rewriting Systems(TRS). This new reducer carries out reductions by message passing between processes assigned to the each function symbols in a given TRS. This method is easily adaptable to distributed parallel processing systems.

We give the external structure of process , the communication protocol for message passing and the internal structure of processes. We conclude this paper by showing the correctness of our method.

1 はじめに

本論文では，関数的なプログラミング言語の抽象インタプリタである項書き換え系（ＴＲＳ）のリダクションを，各関数記号に割り当てられたプロセス間でのメッセージ交換を用いて実行する方式を述べる．

一般に，プログラミング言語の計算モデルには制御フロー，データフロー，リダクションがあるといわれている[14]．現在までに提案されてきた，ＴＲＳにおける書き換えを行うシステム（ＴＲＳ処理系，reducer）はＴＲＳの定義にしたがって素直に作成されたものが多く，ＴＲＳの操作的意味であるリダクションをそのまま実現している．本方式はこの分類によればリダクションであるが，リダクションをプロセス間でのメッセージ交換によって実現している．そのため，分散的にプロセッサーが配置されたシステム上での処理にも適する．

ＴＲＳ自身は，それをプログラミング言語としてみると，記述の容易さ，意味の把握のしやすさ等優れた言語である[1][5]．また，記述の抽象度が高いため，多くの関数的な言語のための汎用の処理系として使うこともでき，ＴＲＳによるＬｉｓｐインタプリタの記述もすでに行なわれている[6]．また，仕様としてみると，すべての関数の実現を行わなくとも，実現されていない関数を単なる記号として扱うことにより実行することができる実行可能仕様でもある．

reducerの実現方法には，(1)そのreducerを直接作成する方法（インタプリタ方式）[5][8]．(2)ＴＲＳを，一度他の言語のプログラムへ変換し，変換後にその言語上で実行する方法（コンパイル方式）[9][11][12][13]．(3)ＴＲＳそれ自身を扱うメタなreducerを記述する方法等がある[16][17]．

最外戦略は多くのＴＲＳの正規化戦略であるが，コンパイル方式で最外戦略を実現することは難しい．これは，変換先のプログラミング言語が基本的に最内戦略であるためであり，最外戦略の実現のためには，実行時に全体に渡って書き換えの順序を制御するための関数をランタイムルーチンとして付加する必要がある[9]．そのため，関数記号の追加の際には，それに合わせてシステム全体を制御するためのこのルーチンを変更する必要があり，分散処理には不向きである．

本方式は，ＴＲＳ上の各関数記号に対して一つのプロセスを割り当て，それらの間のメッセージ交換で実行が進む方式であり，上記によればコンパイル方式に分類されるが，最外戦略の実現に関して，全体を制御するためのランタイムルーチンの必要のない方法である．したがって，関数記号の増加に対して，単にその関数に対するプロセスを追加するのみでよく，システムへの関数記号の登録の手間がない．そのため，プロセスが分散されて配置されている場合でも効率よく実行できる可能性がある．また，他のリダクションシステムへの応用も容易である．

本論文では，各関数に割り当てられたプロセス間でのメッセージの交換の仕方（プロトコル），及びプロセスの内部を明確に定めることによって，ＴＲＳの新たなreducerを提案する．２章で従来のＴＲＳについて述べ，そのreducerを定義する．３章でプロセス間のメッセージ交換の仕方について述べ，４章で，メッセージ交換を実現するために必要な，各関数記号に割り当てるべきプロセスの内部を定義する．６章は本論文のまとめである．

2 項書き換え系

2．1 項書き換え系（Term Rewriting System,TRS）

［定義］（項，Term，Term0，arity）

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∈Var の時 x∈Term

(2) a()∈Const の時 a()∈Term，arity(a)=0

(3) f∈Func，t1,...,tn∈Term の時

f(t1,...,tn)∈Term，arity(f)=n □

(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．

定義中，(2)において，定数a()は単にaと略記することがある．また，通常用いられている関数記号の中置表現を適宜用いることにする．

［定義］（書き換え規則，項書き換え系，ＴＲＳ）

P▷Qの形をしたものを書き換え規則という．ここで，Pは変数でない項であり，Qに出現する変数は必ずPにも出現するものとする．項書き換え系（ＴＲＳ）とは，書き換え規則の有限集合である． □

［定義］（定義関数記号(DefinedFunc)，ｺﾝｽﾄﾗｸﾀ(ConstFunc)）

ＴＲＳ中の書き換え規則P▷Qにおいて，P≡f(t1,...,tn)の時，fを定義関数記号といいその集合をDefinedFuncで表わす．コンストラクタとはFuncの要素であり，DefinedFuncの要素でないものである．その集合

をConstFuncと呼ぶ．

［定義］（リデックス，→，正規項）
ＴＲＳの書き換え規則Ｐ▷Ｑが項Ｍに適用可能であるとは，ある置換θが存在しＭの部分項Ｍ'に対して Ｍ'≡Ｐθ となることである．この時，Ｍ'をＭのリデックスと呼び，Ｍ→Ｎと書く（ＭがＮにリダクションされたという）．ここで，ＮはＭ中のＭ'をＱθで置き換えることによって得られた項である．→ の推移的閉包を$\overset{*}{\rightarrow}$と書く．
項Ｎにリデックスがない場合，Ｎは正規形と呼ばれる．Ｍ$\overset{*}{\rightarrow}$ＮかつＮが正規形であるの時，ＮはＭの正規形と呼ばれ，Ｍ↓と書く． □

本稿で扱うＴＲＳは以下の２つの条件を満たすものする．この程度の制限はＴＲＳをプログラミング言語として使用する際の大きな制限とはならない．これら２つの条件は，ＴＲＳが合流性を有するための十分条件である[10]．

条件１（重なりがないこと）
ＴＲＳのいかなる２つの規則$P_1 \triangleright Q_1$，$P_2 \triangleright Q_2$ においても，P_1がP_2の変数でない部分項（P_2自身も許す）と単一化可能でないこと．すなわち，２つの規則の間に重なりがないこと． □

条件２（線形であること）
ＴＲＳのすべての規則 Ｐ▷Ｑ において，Ｐ中に同じ変数が２度以上出現しないこと． □

２．２　書き換え戦略
一般に項Ｍは複数個のリデックスをもち，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ってくる．しかしながら，本論文で考えている合流性を満たすＴＲＳにおいては，存在すれば結果として得られる正規形は唯一である．複数のリデックスから書き換えるべきリデックスを指定する方法を書き換え戦略（リダクション戦略）という．リダクション戦略により，有限回の書き換えで求められる場合と求められない場合があり，また，正規形を求めるために要する書き換えの回数もパスも異なってくる．

［定義］（最外，最内リデックス）
項Ｍのリデックス'が他のリデックスＭ"の部分項になっていない時，Ｍ'を最外リデックス（outermost redex）という．また，Ｍ'に他のリデックスＭ"が出現していないときＭ'を最内リデックス（innermost redex）という． □

一般に最外，最内リデックスともに複数存在する．各書き換えにおいて最外（最内）のリデックスのいずれかを選択し書き換えるリダクションを最外（最内）リダクションという．特に，最も左側の最外（最内）リデックス１個を書き換える戦略を最左最外（最内）リダクションという．また全ての最外（最内）リデックスを書き換える戦略を並行最外（最内）リダクションという．一般に，逐次的な実行しかできないプログラミング言語を用いて並行リダクションを実現する場合は，各最外（最内）リデックスを左から右へ順にリダクションすることによりシミュレートする事が多い．正規形が存在するときに，必ず求めることのできるリダクション戦略を正規化戦略という．本論文で扱っている重なりのない線形のＴＲＳについては，次のことが知られている．

［命題］[2]
重なりのない線形ＴＲＳにおいて，正規形が存在すれば，最外リダクションが存在する． □

２．３　リデューサ（reducer）
書き換えを実際に行なうためのインタプリタ（ＴＲＳプログラムの処理系）をreducerという．並行最外リダクションを行なうためのreducerであるRpo（reduce parallel outermost）は以下にように定義される．定義中Rpoの領域はTerm→Term でもよいが，３章，４章に合わせ，この領域で定義する．

［定義］（並行最外ﾘﾀﾞｸｼｮﾝ戦略のreducer，Ｒｐｏ）

```
Rpo : Term0 → Term0
Rpo(t) = if nf(t) then t
        elseif match(t) then Rpo(rewrite(t))
        else   Rpo(f(Rpo-one(t1),
                          ... ,
                     Rpo-one(tn)))
               where t≡f(t1, ... ,tn)

Rpo-one(t) = if nf(t) then t
        elseif match(t) then rewrite(t)
        else f(Rpo-one(t1), ... Rpo-one(tn))
               where t≡f(t1, ... ,tn)
```
□

定義で用いている言語は通常のLispのような評価順序を持つものとする．すなわち，if-then-elseは，ま

ず条件部を評価しfalseであった時にelse部を，それ以外の時にthen部を評価する逐次条件文である．また，他の関数の評価に関しては，左から順に内側から評価されるものとする．定義中match(t)はすでに与えられているＴＲＳから，ｔ自身に適用可能な規則Ｐ▷Ｑを求め，Ｐθ≡ｔとなる置換θ（θ：Var→Term0）を返す関数である．本論文で扱っているＴＲＳは合流性を満たすためmatchが適用できる規則を探す順序を特に考慮する必要はない．ｔ自身に適用可能な規則がない時はfalseを評価結果として返す．rewrite(t)はmatch(t)によって求められた規則Ｐ▷Ｑ，置換θを用いＱθを返す関数である．以下に示す最左最内リダクション戦略のreducer Ｒｌｉ（reduce leftmost innermost）でも同様の関数を用いている．

［定義］（最左最内ﾘﾀﾞｸｼｮﾝ戦略のreducer，Ｒｌｉ）

```
Rli : Term0 → Term0
Rli(t) = if nf(t) then t
         elseif t∈Const then Rli-sub(a)
         else   Rli-sub(f(Rli(t1), ... ,Rli(tn))
                        where t≡f(t1, ... ,tn)

Rli-sub(t) = if nf(t) then t
         elseif match(t) then Rli(rewrite(t))
                        else t                    □
```

定義中，t∈Const はｔがConstの要素か否かを判定する関数である．また，nf(t)によるチェックは必要ないが，Rliに関する説明のしやすさのため，挿入してある．Ｒｌｉの定義中，Rli-sub(f(Rli(t_1), ... ,Rli(t_n))）において各Ｒｌｉが並列に動作すれば，これは並行最内戦略のreducerとなる．これを各々Ｒｐｉ（reduce parallel innermost），Ｒｐｉ－ｓｕｂと呼ぶ．

3 プロセス間でのメッセージの交換を用いたreducer

3．1 プロセスの外部構造

本方式では，与えられたＴＲＳの各関数記号ｆ（∈Func）に対して，一つのプロセス proc(f) を割り当てる．各プロセスproc(f)は，ｆに関しての書き換え規則の処理のみを行なうものとする．これにより，関数記号の追加は，単にそれに対応するプロセスの追加のみでよく，定義関数記号ｇについての規則の追加は，１つのプロセス proc(g) へのみでよい．また，各プロセスは，ネットワークにより結合され，分散的に配置された各プロセッサーに割り当てられてもよく，この場合，本方式は分散処理システム上でのリダクションとなる．

図３．１に arity(f)=n である関数記号ｆに対して割り当てられるプロセスproc(f)の外部構造（外部からみたプロセスの構造）を示す．

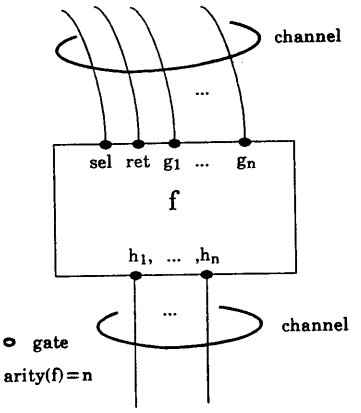

図3.1 プロセスの外部構造

プロセスは２つのゲート群（(sel,ret,g_1, ... ,g_n) 及び (h_1, ... ,h_n)）を持つ．selをセレクタゲート，retをリターンゲート，g_1, ... ,g_n 及び h_1, ... ,h_nを引数ゲートと呼ぶ．プロセスを関数とみれば，ゲートは関数の仮引数に対応している．リダクションはこれらのゲートを通してのメッセージの交換によって実行される．メッセージは下記の構造を持ち，これらは関数の実引数に対応する．

メッセージの構造

<ﾒｯｾｰｼﾞ>:=<ｾﾚｸﾀﾄｰｸﾝ>,<ﾘﾀｰﾝﾄｰｸﾝ>,
<ｱｰｸﾞﾒﾝﾄﾄｰｸﾝ$_1$>, ... ,<ｱｰｸﾞﾒﾝﾄﾄｰｸﾝ$_n$> |
<ｱｰｸﾞﾒﾝﾄﾄｰｸﾝ$_1$>, ... ,<ｱｰｸﾞﾒﾝﾄﾄｰｸﾝ$_n$>

<ｾﾚｸﾀﾄｰｸﾝ>は各プロセスがどの様な動きをすればよいかを外部から指示するために用いられる．並行最外戦略でリダクションすることを指示する'po'，一度だけ最外戦略でリダクションすることを指示する'po-one'，並行最内戦略でのリダクションを指示する'pi'がある．使用法の詳細は３．３節で示す．<ﾘﾀｰﾝﾄｰｸﾝ>はセレクタトークンによって示された計算の結果を返すべきプロセス，及びそのプロセスのゲートを指示するために用いる．<ｱｰｸﾞﾒﾝﾄﾄｰｸﾝ>はリダクションを行なうときの関数の引数に対応する．

プロセスの状態には，生成，活動，待ち，消滅，がある．プロセスｐが活動状態にあることをactive(p)で，待ち状態にあることをwait(p)で表わす．

プロセスが自分自身を再帰的に呼ぶことは許さない．

本方式では，リダクションの制御を，２章に示したreducerのように再帰によるのではなく，プロセス間のメッセージ交換で行なうためである．また，一般に，再帰的に自分自身を呼ぶためには，多くのメモリ等資源を必要とするため，プロセスが大きくなる欠点があり，本方式にはそぐわない制御法である．

３．２ プロセス間のメッセージ交換

プロセスは生成されたあと，上部のゲート群にメッセージが到達したとき（すなわち，全てのトークンがそろったとき）起動され，活動状態となる．これはデータフロー計算モデルと同様である．また，待ち状態になっているプロセスの下部のゲート群にメッセージが到達すると，プロセスは再始動され，待ち状態から活動状態に変わる．プロセス間のメッセージ交換の記述には図３．２に示す記法を用いる．

```
  {}q1                 ﾌﾟﾛｾｽpはn個のプロセスq1～
p    ...                        qnとチャネルを設定
  {}qn

       {A1→}q1     pは各qiに対してメッセージ
=> Sp(p)  ...      Aiを同時に送信し状態Spになる
       {An→}qn

       {}S1(q1)    各qiはチャネルを通してメッセ
=> Sp(p)  ...      ージAiを受け取り,各々状態Si
       {}Sn(qn)                        になる
```

図３．２　プロセス間でのメッセージ交換の記述法

本方式おけるメッセージの送信には，新たにプロセスproc(g)を生成し，そのproc(g)に対してメッセージを送信，自分自身（proc(f)）は消滅する場合（図３．２(a)）と，自分自身（proc(g)）の起動元である，待ち状態に状態になっているプロセス（proc(f)）へメッセージを送り，自分自身は消滅する場合（図３．２(b)）がある．

```
proc(f) => proc(f) proc(g) proc(g)
        => active(proc(g))
                    (a)

proc(f) proc(g)
 => wait(proc(f)){}proc(g))
 => wait(proc(f)){}active(proc(g))
 => wait(proc(f)){}
 => active(proc(f))
                    (b)
```

図３．３　プロセス間でのメッセージ交換の例

３．３ メッセージの交換を用いたリダクションの例

２章で定義したＲｐｏ，Ｒｐｉでは，，f(0+s0)の正規形は各々Rpo(f(0+s0))，Rpi(f(0+s0))で求められる．本式では，これを，fに割り当てたプロセスproc(f)に対してメッセージ 0+s0,<結果の送り先>,'po' または 0+s0,<結果の送り先>,'pi' を送ることによって求められる．

ＴＲＳプログラム１

```
      f(0) ▷ s0              x+0 ▷ x
f(s(x)) ▷ f(x)*s(x     x+s(y) ▷ s(x+y)
```

ＴＲＳプログラム１では，Var={x,y}，Func={f,+,s,0}，DefinedFunc={f,+}，ConstFunc={*,s,0} である．図３．４，３．５に，ＴＲＳプログラム１を用いたリダクションの例を示す．porc(KeyBoard)，porc(CRT)は各々，入力及び出力のためのプロセスである．

```
・f(0+s0)→f(s(0+0))→(0+0)*f(0+0)→...→s0*s0
proc(KeyBoard){0+s0,CRT,'po'→}proc(f)
=> wait(proc(f)){0,s0,h,'po-one'→}proc(+)
=> wait(proc(f)){}active(proc(+))
=> wait(proc(f)){←s(0+0)}
=> active(proc(f))
=> active(proc(f)) proc(*)
=> {0+0,f(0+0),CRT,'po'→}proc(*)
                ...
=> {s0*s0→}proc(CRT)
```

図３．４　ﾌﾟﾛｾｽ間通信による並行最外ﾘﾀﾞｸｼｮﾝの例

```
・f(0+s0)→f(s(0+0))→f(s0)→...→s0*s0
proc(KeyBoard){0+s0,CRT,'pi'→}proc(f)
=> wait(proc(f)){0,s0,h,'pi'→}proc(+)

                                {'',h1,'pi'→}proc(0)
=> wait(proc(f)){}wait(proc(+))
                                {'',h1,'pi'→}proc(s)

                                    {←0}
=> wait(proc(f)){}wait(proc(+))
                {}wait(proc(s)){'',h,'pi'→}proc(0)

                                    {←0}
=> wait(proc(f)){}wait(proc(+))
                                {}wait(proc(s)){←0}
```

{←0}

=> wait(proc(f)){}wait(proc(+))

{←s0}

=> wait(proc(f)){}wait(proc(s))

{0,0,h,'pi'→}proc(+)

...

=> wait(proc(f)){←s0}

...

=> {s0*s0→}proc(CRT)

図３．５　ﾌﾟﾛｾｽ間通信による並行最内ﾘﾀﾞｸｼｮﾝの例

4 プロセスの内部構造

3章では，各プロセスの外部構造を示した後，それらの間でのメッセージ交換の仕方（プロトコル）を定め，リダクションの方法について述べた．本章では3章で示されたプロトコルを用いてメッセージの交換をするための，プロセスの内部構造について述べる．

4．1 並行最外リダクションのためのメソッド

並行最外リダクションを行なうためには，セレクタトークン'po'，'po-one'に対するプロセス内部での処理（メソッドと呼ぶ）を定める必要がある．Ｒｐｏからの類推により，arity(f)=n である proc(f)が'po'を受け取った場合の処理は以下のようになる．

・match(f(g_1, ... ,g_n)) が FALSE以外の時

書き換えを行ない，その結果に対応するプロセスに対して，並行最外戦略での書き換えを指示し，消滅する．

・match(f(g1, ... ,gn)) が FALSEの時

1．各部分項に対応するプロセスに対して，1度だけ並行最外戦略でリダクションすることを指示し，その結果を待つ．

2．待ち状態でメッセージを受け取ったら新たにproc(f)を生成，そのproc(f)に対して並行最外戦略でのリダクションを指示し，消滅する．

すなわち

・match(f(g_1, ... ,g_n)) が FALSE以外の時

{g_1, ... ,g_n,ret_token,'po'→}proc(f)

=>{s_1, ... ,s_m,ret_token,'po'→}proc(g)

(where rewrite(f(g_1, ... ,g_n)≡g(s_1, ... ,s_m))

(PO-1)

・match(f(g_1, ... ,g_n)) が FALSEの時

{g_1, ... ,g_n,ret_token,'po'→}proc(f)

{s_{11}, ... ,s_{1m1},h_1,'po-one'→}proc(s_1)

=>wait(proc(f))　　...

{s_{n1}, ... ,s_{nmn},h_n,'po-one'→}proc(s_n)

(where g_i≡s_i(s_1, ... ,s_{mi}))

(PO-2)

・proc(f)が待ち状態の時

{h_1→}

... wait(proc(f))

{h_n→}

=>{h_1, ... ,h_n,ret_token,'po'→}proc(f)

(PO-3)

また，Rpo-oneからの類推により，'po-one'に対するメソッドは以下のようになる．

・match(f(g1, ... ,gn)) が FALSE以外の時

書き換えを行ない，その結果を送り先であるプロセスへ返し，消滅する

・match(f(g1, ... ,gn)) が FALSEの時

1．各部分項に対応するプロセスに対して，1度だけ並行最外戦略でリダクションすることを指示し，その結果を待つ．

2．待ち状態でメッセージを受け取ったら，受け取った結果にｆをつけてretで示されたプロセスのゲートに返し，消滅する

すなわち

・match(f(g_1, ... ,g_n)) が FALSE以外

{g_1, ... ,g_n,ret_token,'po-one'→}proc(f)

=>{g(s_1, ... ,s_m)→}ret_token

(where rewrite(f(g_1, ... ,g_n)≡g(s_1, ... ,s_m))

(PO_ONE-1)

・match(f(g_1, ... ,g_n)) が FALSE

{g_1, ... ,g_n,ret_token,'po-one'→}proc(f)

{s_{11}, ... ,s_{1m1},h_1,'po-one'→} to proc(s_1)

=>wait(proc(f))　　...

{s_{n1}, ... ,s_{nmn},h_n,'po-one'→} to proc(s_n)

(where g_i≡s_i(s_1, ... ,s_{mi}))

(PO_ONE-2)

・proc(f)が待ち状態の時

{h_1→}

... wait(proc(f))

{h_n→}

=> {f(h_1, ... ,h_n)}ret_token

(PO_ONE-3)

4．2 正規形の判定

4．1で示した内部構造をもつプロセスを用いれば，並行最外戦略のリダクションを行なうことが出来るが，正規形が求まった時に停止する処理が含まれていないため，例えばＴＲＳプログラム1で，{s0,s0,'po'→}proc(*) はデッドロックに陥る．一般に正規形か否かの判定は，定義通り作成しようとすれば，すべての規則を必要とする．しかし，本方式ではproc(f)はfに関する規則についてのみの処理を行なうことを基本とするため，fに関する規則のみで正規形か否かを判定する必要がある．そのため，本方式では，以下の性質を利用する．

［性質4．1］

f(t_1, ... ,t_n)) が 正規形であることと，以下の2つを同時に満足することは同値

1．すべてのt_1, ... ,t_n が 正規形である

2．match(f(t_1, ... ,t_n)) が FALSE □

一度正規形と判断された項tはその旨のカラーリングを行なうものとする（NF(t)と表記）．したがって，proc(f)では，match(f(t_1, ... ,t_n))がFALSEを返した時，アーグメントトークンのすべてがNFでカラーリングされているかで，f(t_1, ... ,t_n)が正規形かを判定できる．図4．1に(PO-1)～(PO-3)，(PO_ONE-1)～(PO_ONE-3)を満たし，性質4．2を用いた正規形判定の機能を組み込んだプロセスの内部構造を示す．図中，各文の逐次実行を；で，同時実行を｜で表わしている．また，メッセージの送信をsendで，ゲートh_1, ... ,h_nで待ち状態になることをwait(h_1, ... ,h_n)で，消滅することをKillMeで表わしている．また，isNF(t_1, ... ,t_n) により，性質4．1の1に関する判定を行なう．

4．3 並行最内リダクションのためのメソッド

［性質4．2］

Ｒｐｉは以下のように書き換えることが出来る．

a∈Const の時

Rpi(a) = if match(a) then Rpi(rewrite(a))

else NF(a)

```
process f (sel,ret,g1, ... ,gn)
case sel of
  'po':
      match(f(g1, ... ,gn)) が FALSE以外
          send (s1, ... ,sm,ret,'po') to proc(g);
               (where rewrite(f(g1, ... ,gn)≡g(s1, ... ,sm))
          KillMe
      match(f(g1, ... ,gn)) が FALSE
          isNF(g1, ... ,gn) が TRUE
              send NF(f(g1, ... ,gn)) to ret;
              KillMe
          isNF(g1, ... ,gn) が FALSE
              send (s11, ... ,s1m1,h1,'po-one') to proc(s1)|
                                  ...
              send (sn1, ... ,snmn,hn,'po-one') to proc(sn);
                       (where gi≡si(s1, ... ,smi))
              wait(h1, ... ,hn);
              send (h1, ... ,hn,ret,'po') to proc(f);
              KillMe
  'po-one':
      match(f(g1, ... ,gn)) が FALSE以外
          send g(s1, ... ,sm) to ret;
               (where rewrite(f(g1, ... ,gn)≡g(s1, ... ,sm))
          KillMe
      match(f(g1, ... ,gn)) が FALSE
          isNF(g1, ... ,gn) が FALSE以外
              send NF(f(g1, ... ,gn)) to ret;
              KillMe;
          isNF(g1, ... ,gn) が FALSE
              send (s11, ... ,s1m1,h1,'po-one') to proc(s1)|
                                  ...
              send (sn1, ... ,snmn,hn,'po-one') to proc(sn);
                        (where gi≡si(s1, ... ,smi))
              wait(h1, ... ,hn);
              send f(h1, ... ,hn) to ret;
              KillMe
```

図4．1 並行最外戦略のためのプロセスの内部構造

fがConstの要素でないとき

Rpi(f) = if isNF(t_1, ... ,t_n)

then if match(f(t_1, ... ,t_n))

then Rpi(rewrite(f(t_1, ... ,t_n)))

else NF(f(t_1, ... ,t_n))

else Rpi(f(Rpi(t_1), ... , Rpi(t_n)) □

性質4．2に基づき，セレクタ'pi'に対するメソッドは以下のようになる．

・a∈Const の時

・match(a) が FALSE以外の時

{'',ret_token,'pi'→}proc(a)

=>{s_1, ... ,s_m,ret,'pi'→}proc(g)

(where rewrite(f(g_1, ... ,g_n)≡g(s_1, ... ,s_m))

(PI-1)

・match(a) が FALSEの時

{'',ret_token,'pi'→}proc(a)

=> {NF(a)→}ret_token

(PI-2)

・ f がConstの要素でないとき

・isNF(g_1, ... ,g_n) が TRUE, かつ

match(f(g_1, ... ,g_n)) が FALSE以外の時

{g_1, ... ,g_n,ret_token,'pi'→}proc(f)

=>{s_1, ... ,s_m,ret,'pi'→}proc(g)

(where rewrite(a)≡g(s_1, ... ,s_m))

(PI-3)

・isNF(g_1, ... ,g_n) が TRUE, かつ

match(f(g_1, ... ,g_n)) が FALSE

{g_1, ... ,g_n,ret_token,'pi'→}proc(f)

=>{NF(f(g_1, ... ,g_n)→}ret_token

(PI-4)

・isNF(g_1, ... ,g_n) が FALSE

{g_1, ... ,g_n,ret_token,'pi'→}proc(f)

{s_{11}, ... ,s_{1m1},h_1,'pi'→} to proc(s_1)

=>wait(proc(f)) ...

{s_{n1}, ... ,s_{nmn},h_n,'pi'→} to proc(s_n)

(where g_i≡s_i(s_1, ... ,s_{mi}))

(PI-5)

・proc(f)が待ち状態の時

{h_1→}

... wait(proc(f))

{h_n→}

=> {f(h_1, ... ,h_n)}ret_token

(PI-6)

図４．２に上記の規則を満たす，セレクタpiに対するメソッドを示す．

４．４ 変換の性質

本方式は，与えられたＴＲＳプログラムの各関数記号ｆ∈Ｆｕｎｃに対して，４．２節及び４．３節で述べたメッソドを合わせ持つプロセス porc(f) へ変換する．その後，３章で述べたプロトコルに沿ったメッセージ交換によりリダクションが行なわれる．この意味において，本方式はＴＲＳプログラムをプロセスへ変換するコンパイラである．この変換において以下の性質を有する．

・ a∈Const の時

```
process a (sel,ret)
case sel of
  'pi':
      match(a) が FALSE以外
          send (s1, ... ,sm,ret,'pi') to proc(g);
                (where rewrite(a)≡g(s1, ... ,sm))
          KillMe
      match(a) が FALSE
          send NF(a) to ret;
          KillMe
```

・ f がConstの要素でないとき

```
process f (sel,ret,g1, ... ,gn)
case sel of
  'pi':
      isNF(g1, ... ,gn) が TRUE
         match(f(g1, ... ,gn)) が FALSE以外
             send (s1, ... ,sm,ret,'pi') to proc(g);
               (where rewrite(a)≡g(s1, ... ,sm))
             KillMe
         match(f(g1, ... ,gn)) が FALSE
             send NF(f(g1, ... ,gn)) to ret;
             KillMe
      isNF(g1, ... ,gn) が FALSE
             send (s11, ... ,s1m1,h1,'pi') to proc(s1)|
                                ...
             send (sn1, ... ,snmn,hn,'pi') to proc(sn);
             wait(h1, ... ,hn);
             send (h1, ... ,hn,ret,'pi') to proc(f);
             KillMe
```

図４．２　並行最内戦略のためのプロセスの内部構造

[性質４．３]

(1) Rpo(f(t_1, ... ,t_n)) => ... => s

iff

{t_1, ... ,t_n,R,'po'→}proc(f) =>

... => {NF(s)→}proc(R)

(2) Rpi(f(t_1, ... ,t_n)) => ... => s

iff

{t_1, ... ,t_n,R,'pi'→}proc(f) =>

... => {NF(s)→}proc(R)

□

（証明の手法）

１回の書き換えはrewriteの実行によってのみ行なわれることに注目し，一方の１回のリダクションを，他方がシュミレートとしていることを示す．このとき，項の構造にしたがい，項が正規形の時と，そうでない時に分類し証明することができる．

また，コンパイル時には，個々の関数記号により，

無駄なメソッドを省いたプロセスを割り当てることが出来る．例えば，ConstFuncの要素では，常にmatchがFALSEを返すため，matchによる判定を必要としない．そのためmatchがFALSE以外を返すときのメソッドを必要としない．また，a∈Constに対しては，ＴＲＳプログラムが与えられた時点で，match(a)の結果が分かるため，FALSE以外の場合と，FALSEの場合のいずれか一方に対するメソッドのみでよい．

5　まとめ

本論文では，ＴＲＳのリダクションを，各関数記号に割り当てられたプロセス間でのメッセージの交換を用いて実行する方式を述べた．各関数に割り当てるべきプロセスの外部構造，リダクションのためのプロトコル，及びプロセスの内部構造を明確に定めることによって，ＴＲＳの新たなreducerを提案した．また，本方式はコンパイル方式であるが，その変換の正しさについても示した．

本方式は，リダクションをメッセージの交換によって実行しているため，ＴＲＳを用いた証明システム等の作成に於いても，そのシステムをメッセージ交換に基づいて作成できる利点がある．また，本方式はリダクションによって実行するものであれば，高階関数を扱う一般の関数型言語，Ｓｃｈｅｍｅ，Ｌｉｓｐへも応用が可能である．

さらに，プロセスが分散されて配置されている場合でも効率よく実行できる可能性があり，リダクションのためのアーキテクチャへの応用も可能と思われる．具体的な手法については今後の課題である．ＡＬＩＣＥ[3]のような手法も興味深い．

参考文献

[1]Burstall,R.: Hope:An Experimental applicative Language, Conferrence Record of 1980 Lisp Conference(1980), pp136-13

[2]O'Donnell,M. :Computing in Systems Described by Equations. Lecture Notes in Comput. Sci. No.58, Springer(1977)

[3]Darlington,J. and Reev,M. :ALICE:A Multi-processor Reduction Machine for the Parallel Evaluation for Applicative Languages, Proc. Functional Programming Languages and Computer Architecture, pp65-75(1981)

[4]二木厚吉，中川中 :抽象データ型とＯＢＪ２，ｂｉｔVol.20, No.9(1988), pp67-80

[5]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(1983), pp.133-146

[6]Hoffmann,C.M and O'Donnel,M.J :Programming with Equations. ACM Trans. on Prog. Lang. Syst. Vol.4, No.1(1982), pp.83-112

[7]稲垣康善,坂部俊樹:抽象データ型の仕様記述法の基礎(1)－多ソート代数と等式論理－，情報処理，Vol.25, No.1(1984), pp.47-53

[8]長田博康:等式システムとそのインタプリタ，理研シンポジウム関数プログラミング資料(1986), pp.28-33

[9]布川博士，黒田清隆，富樫敦，野口正一:項書き換え系の関数型言語への変換による実現，コンピュータソフトウェア Vol.4 No.4(1987), pp5-15

[10]Rosen,B.K. :Tree-manipulating Systems and Church-Rosser Theorems. J. ACM, Vol.20(1973), pp.160-187

[11]酒井正彦，坂部俊樹，稲垣康善:抽象データ型直接実現システムCdimple，コンピュータソフトウェア Vol.4 No.4(1987), pp16-27

[12]杉山裕二，鈴木一郎，谷口健一，嵩忠男:あるクラスの項書き換え系の効率のよい実行，信学論 J65-D No.7(1982)pp858-865

[13]戸村哲，二木厚吉:項書き換えシステムからLispプログラムへの変換系，通学技報，SS86-9(1986)，pp15-20

[14]Treleaven,P.C.,Brownbridge,D.R. and Hopkins, R.P.:Data- Driven and Demand Driven Computer Architecture, Comput. Surv. Vol.14, No.1(1982), pp93-143.

[15]Vuillemin,J. :Correct and Optimal Implementation of Recursion in a Simple, Programing Language. J. Comput. Syst. Sci. Vol.9(1974), pp.332-354

[16]山中英樹,直井徹,坂部俊樹,稲垣康善:万能項書き換えシステムと部分計算,信学技報，SS86-12(1986)，pp.21-27

[17]矢野博之,布川博士,富樫敦,野口正一:項書き換え系のメタインタプリタE-TRS，コンピュータソフトウェア，Vol.5，No4(1988)，pp40-51