プログラミング言語 20－6
（１９８９．２．１０）

# 戦略の表明を持つ項書き換え系 Ａ－ＴＲＳの実現と評価

布川 博士，古賀 信哉，野口 正一
（東北大学電気通信研究所）

**概要**

本稿では，我々がすでに提案している，ＴＲＳプログラム中に陽にリダクション戦略を表明でき，各ＴＲＳプログラムごとに戦略を定められることができる項書き換え系（Ａ－ＴＲＳ）の処理系作成の報告，それを用いたＡ－ＴＲＳの評価を行なう．

項書き換え系（ＴＲＳ）の処理系であるreducerでは，正規化戦略の実現のみでなく速度も要求される．従来の処理系はいずれか一方を犠牲にすることにより他方の利点を導入し作成されてきた．我々はすでに，単一の戦略を用いるのではなく，それぞれのＴＲＳプログラムに応じた戦略をＴＲＳプログラム中に記述できるＴＲＳ（Ａ－ＴＲＳ）及びその処理系を提案している．Ａ－ＴＲＳでは，各ＴＲＳプログラム中にそれ専用の戦略を定められることができるため，ユーザーが意図した通りに，書き換えが有効に行なわれるＴＲＳプログラムを作成できる．

本稿ではＡ－ＴＲＳのreducerを実際にLispを用いで作成し，その評価を速度，表明の効果の観点から行なう．また，それに基づきＴＲＳプログラムへの表明のいれ方について検討，議論を行なう．

The Implementation and Evaluation
of
Strategy Annotated Term Rewriting System(A-TRS)

Hiroshi Nunokawa, Shinya Koga, Shoichi Noguchi

Research Institute of Electrical Communication Tohoku University
Katahira 2-1-1, Sendai, 980 JAPAN

Abstract

In this paper, we show the implementation and evaluation of A-TRS.

We have proposed a new type of Term Rewriting System (TRS), named A-TRS, which has annotations for reduction strategies. The reducer of TRS should have the normalizing strategy(Normalizing strategy can obtain the normal form anytime, if it exists), and should also be high-speed. But these two things are mutually exclusive. For this reason, most reducers are implemented by compromising one of them. But, to efficiently use TRS as a programming language, it is very important to offer those features at the same time. A-TRS is one such attempt. In A-TRS, the programmer can control the order of reduction at will. And he obtain the normal form as he wishes.

This paper shows the implementation of a reducer of A-TRS written in Lisp, and evaluates A-TRS based upon the current implementation from the point of speed and effect of annotations. Finally we discus the methodology to use annotations.

## 1．はじめに

項書き換え系（ＴＲＳ）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．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．従ってプログラムの設計のみでなく，実現もトップダウンに行うことが可能である．またその意味は，代数的に明確にされており，等号論理との関連も多く知られている[4][7]．さらに操作的意味との対応もとれ，数学的意味と操作的意味のセマンティックギャップがない．また，理論的背景も整備されており，プログラムの実行結果である正規形を求めるための戦略（正規化戦略）もすでに与えられている[1][7]．

ＴＲＳにおける書き換えを行うシステム（ＴＲＳ処理系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）[3][5][14]．(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）[6][10][11]．(3)ＴＲＳそれ自身を扱うメタなreducerを記述する方法[13][15]等がある．実現のためには，正しさ（正規化戦略の実現）のみでなく速度も要求される．この２つは，互いに排反する条件であり，速度向上のため正規化戦略を実現しない方法もある[10][11]．また，速度向上を目的として対象とするＴＲＳ自身を制限した実現もある[10]

このように従来の処理系は正しさか，速度のいずれかを犠牲にすることにより他方の利点を導入し作成されてきた．しかし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，双方とも満足のゆく処理系が必要である．そのためには単一の戦略を用いるのではなく，それぞれのＴＲＳプログラムに応じた戦略を用いることが有効である．すなわち，各ＴＲＳプログラム中に，そのＴＲＳプログラムを有効に使うための戦略を定められることが望ましい．それによりユーザーが意図した通りに，書き換えが行なわれるＴＲＳプログラムを作成することができる．

本論文では，ＴＲＳプログラム中に陽に戦略を表明する事ができる項書き換え系を提案する．２章では従来のＴＲＳについて述べ，その処理系を定義する．３章では表明付きの項書き換え系（Ａ－ＴＲＳ）を２章と同じ枠組みで提示する．４章では，Ａ－ＴＲＳのreducerを実際にLispを用いで作成し，その評価を速度，表明の効果の観点から行なう．また，それに基づきＴＲＳプログラムへの表明のいれ方について検討，議論を行なう．また，評価の方法についても検討をする．５章は結論である．

## 2．項書き換え系

本章では，文献[1]に基づいてＴＲＳに関する諸定義と本論文に関連する基本的結果について簡単に触れた後ＴＲＳ処理系（reducer）について述べる．

### 2．1　項書き換え系（Term Rewritting System，ＴＲＳ）

［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∈Var　の時　x∈Term

（２）　a()∈Const　の時　a()∈Term

（３）　f∈Func，t1,...,tn∈Term　の時

f(t1,...,tn)∈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．

定義中，（２）において，定数a()は単にaと略記することがある．（３）においてｆ自身の取り得る引数の数（アリティ）ｎは重要でないので省略している．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とＱ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．Ｐ≡f(...)の時，fを定義関数記号という(Ｍ≡Ｎで，ＭとＮが構文的に等しいことを表わす)．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］（置換，リデックス，→．正規項）

ＴＲＳ，Ｒの書き換え規則Ｐ▷ＱがＭに適用可能

であるとは，ある置換$\theta$が存在しMの部分項M'に対して M'≡P$\theta$ となることである．この時M'をMのリデックスと呼び，規則P▷QがMに適用され項Nを得る（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Q$\theta$で置き換えることによって得られた項である．M →R Nで，Nが規則の１回の適用によってMから得られることを示す．また，その反射・推移的閉包を→*Rと書く．M →*R Nが成り立つとき，MはNに簡約可能であると言い，前後からRが明らかな場合→R，→*RのR を省略する．

項Nにリデックスがない場合，Nは正規項（正規形）と呼ばれる．M→*NかつNが正規項であるの時，NはMの正規項と呼ばれM↓と表記する．

ＴＲＳにおいて，正規項を計算結果と見なすと，結果の一意性を保証するのが次の合流性である．

［定義］（合流性）

ＴＲＳが合流性（Church-Rosser性)を満たすとは，M→*N1 かつM→*N2の時，ある項Lが存在して，N1→*LかつN2→*Lとなることである．

ＴＲＳが合流性を満足するなら，項Mの正規項は高々１個である．

［定義］（重なり）

M，Nを項，P▷Q，P'▷Q' を書き換え規則とする．

(１)MがNと重なりがあるとは，MがNの変数でない部分項N'（N自身も許す）と単一化可能であること．

(２)MがMと重なりがあるとは，MがMの真部分項M'（M自身でない部分項）と単一化可能であること．

(３)P▷QとP'▷Q'が重なりがあるとは，PとP'に重なりがあること．

(４)ＴＲＳに重なりがあるとは，ＴＲＳ中のある２つの規則（同一の規則の場合も許す）に重なりがあること．

［定義］（線形）

（１)項Mが線形であるとは，M中に同じ変数が２つ以上出現しないこと．

（２)規則P▷Qが線形であるとは，Pが線形であること．

（３)ＴＲＳが線形であるとは，ＴＲＳの全ての規則が線形であること．

ＴＲＳが合流性を満たすかどうかは一般に決定不能である．しかし，ある制限されたＴＲＳに対しては，次の十分条件が知られている．

［命題］[1]

ＴＲＳに重なりがなくかつ線形である時，ＴＲＳは合流性を満たす．

以下本論文では，重なりがなくかつ線形なＴＲＳのみを扱い，単にＴＲＳと呼ぶ．このようなＴＲＳに対して１節で述べたように，等号論理を用いたり，代数的に意味を考える立場と[4][7]，→Rを用いて操作的にのみ考える立場がある．ここでは操作的にのみ考え，ＴＲＳを書き換えのためのスキーマ，すなわちプログラムとみる立場に立つ．この時，ＴＲＳを特にＴＲＳプログラムと呼ぶ．

## 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項は唯一である．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M'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ュミレートする事が多い．

（注）

注

並行リダクションにおいて，すべての最外リデックスを書き換えることを１ステップのリダクションとする考え方もあるが，本論文では，逐次的にしか実行できないプログラミング言語，計算機での実現を考えているため，１回の書き換えを１ステップのリダクションと考える．

正規項が存在するときに，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][8]

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]（並行最外リダクション戦略のreducer）

```
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，Ｒからｔ自身に適用可能な規則を求める関数ある．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を用い，
ｔ≡Ｐθとなる置換θを求め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]（最左最内リダクション戦略のreducer）

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

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

Rliの定義中nf(t)によるチェックは必要ないが，Rliに関する証明のしやすさ及び実行時の効率を上げるために挿入してある．

ＴＲＳプログラム１

```
double(x) ▷ x*x
x*0  ▷ 0
x*s(y) ▷ x+(x*y)
0+y ▷ y
s(x)+y ▷ s(x+y)
```

ＴＲＳプログラムを１用いた，f(s0*ss0)のリダクション，Rpo，Rliによる評価の例を下記に示す．

並行最外リダクションの例
(PO→は並行最外リダクションを用いていることを示す)

```
double(s0+s0)
      PO→(s0+s0)*(s0+s0)
      PO→s(0+s0)*(s0+s0)
      PO→s(0+s0)*s(0+s0)
      PO→s(0+s0)+(s(0+s0)*(0+s0))
      PO→s((0+s0)+(s(0+s0)*(0+s0)))
      PO→s(s0+(s(0+s0)*(0+s0)))
            ・・・
```

Rpoでの評価例（rewrittenの表記の部分がrewriteが実行された直後であることを示す）

```
Rpo(double(s0+s0))
  1=>Rpo((s0+s0)*(s0+s0))
  2=>Rpo(Rpo-one(s0+s0)*Rpo-one(s0+s0))
  3=>Rpo(s(0+s0)*Rpo-one(s0+s0))       :rewritten
  4=>Rpo(s(0+s0)*s(0+s0))              :rewritten
  5=>Rpo(s(0+s0)+(s(0+s0)*(0+s0)))     :rewritten
  6=>Rpo(s((0+s0)+(s(0+s0)*(0+s0))))   :rewritten
  7=>Rpo(s(Rpo-one(0+s0)
              +Rpo-one((s(0+s0)*(0+s0))))
  8=>Rpo(s(s0+Rpo-one(s(0+s0)*(0+s0))))
                                    :rewritten
  9=>Rpo(s(s0+Rpo-one(s(0+s0))*Rpo(0+s0))))
                      . . .
```

最左最内リダクションの例（LI→は最左最外リダクションを用いていることを示す）

```
double(s0+s0)
      LI→double(s(0+s0))
      LI→double(ss0)
      LI→ss0*ss0
          . . .
```

Rliでの評価例（rewrittenの表記の部分がrewriteが実行された直後であることを示す）

```
Rli(double(s0+s0))
  1=>Rli-one(double(Rli(s0+s0)))
  2=>Rli-one(double(Rli-one(Rli(s0)+Rli(s0))))
  3=>Rli-one(double(Rli-one(s0+Rli(s0))))
  4=>Rli-one(double(Rli-one(s0+s0)))
  5=>Rli-one(double(Rli(s(0+s0)))    :rewritten
  6=>Rli-one(double(Rli-one(s(Rli(0+s0)))))
  7=>Rli-one(double(Rli-one(s(
              Rli-one(Rli(0)+Rli(s0))))
  8=>Rli-one(double(
              Rli-one(s(Rli-one(0+Rli(s0))))
  9=>Rli-one(double(Rli-one(s(Rli-one(0+s0))))
 10=>Rli-one(double(Rli-one(s(Rli(s0)))rewritten
 11=>Rli-one(double(Rli-one(ss0))
              . . .
```

この例から分かるようにrewriteに依って書き換えられた項がＴＲＳ上での各戦略におけるリデックスある．

従って，ＴＲＳ上からの定義とは逆に，これらreducerを用いて，ＴＲＳ上のリダクションを定義することができる．

最外リダクションでは書き換え時に，内部のリデックスをそのまま複数個コピーしてしまうため同じ書換えを複数回行なう必要がある．また，そのため書換えの回数が多くなり，実行時に出現する項も大きもなる欠点がある．最左最内リダクションでは書き換え回数，項の大きさとも並行最外リダクションに比べともに小さいことが分かる．他のＴＲＳプログラムに関しても同様の傾向があることが知られている[]．反面，最内リダクションでは存在する正規形を求められないことがある．

つぎにreducer間の能力について以下の定義する．

［定義］（reducerの能力，≤）
２つのreducer reduce-A，reduce-B　において
∀R:TRS，t∈Term，　reduce-B(t)=s
　　ならば　reduce-A(t)=s
が成立するとき　reduce-A ≥ reduce-B と書き
　　reduce-A は reduce-B よりも能力が高いという．
また，reduce-A≥reduce-B かつ reduce-A≤reduce-B の時 reduce-A＝reduce-B と書きreduce-A と reduce-B は同じ能力を有するという．

［性質２．１］（Rpo,Rliの能力）
（１）　Ｒｐｏ ≥ Ｒｌｉ
（２）　いかなるreducer reduce-A に対しても
　　reduce-A ∘ Rpo = Rpo
　　reduce-A ∘ Rli = pri

［性質２．２］（Full-Rpo）
reducer Full-Rpo を以下のように定義する．

```
Full-Rpo(t) = if nf(t) then t
      elseif t≡a then Full-Rpo(Rpo-one(t))
      else
        Rpo(f(Full-Rpo(t1), ... ,Full-Rpo(tn))
                    if t≡f(t1, ... tn)
      この時　Full-Rpo = Rli
```

３．表明付き項書き換え系

３．１　表明付き項書き換え系（Ａ－ＴＲＳ）の例
ＴＲＳプログラム１に加えて，次のプログラムが与え

られているする．

ＴＲＳプログラム２

f(0)▷s0

f(s(x))▷f(x)*s(x)

これを２章で定義した通常のＴＲＳでリダクションをする．並行最外リダクションでは以下のようになり，リデックスがそのままコピーされる現象が現われる．

f(so*ss0)

P0→f(s0+(S0*s0))

P0→f(s(0+(s0*s0)))

P0→f(0+(s0*s0))*s(0+(s0*s0))

P0→f(0+(s0*s0))+f(0+(s0*s0))*(0+(s0*s0))

...

上記のリダクションP0でのリダクションでは，リデックスがそのままコピーされまったく同じ書き換えを複数回行なう必要があり効率が悪い．そのため，書き換え回数の多いリデックスs0*ss0のように正規項が存在することが分かっており，その正規形を求めてから，fに関するリダクションをはじめたほうが，リデックスのコピーがされずに有効であること分かれば，そのような戦略でリダクションをするような表明入れればよい．その際，部分項s0*ss0を並行最外リダクションで求めるか，最左最内リダクションに依るかにより，f([s0*ss0]),f({s0*ss0})のいずれかを用いる．その表明に関してリダクションを行なった後に，全体が並行最外リダクションされる．この例では，s0*ss0のリダクションはリデックスのコピーがされない最左最内戦略のほうが効率がよい．以下に各々のリダクション例を示す．（下線を付した項がＡ－ＴＲＳのリデックスである）

リダクション例１

f([s0*ss0])

→f(s0+(s0*s0))

→f(s(0+(s0*s0)))

→f(s(s0*s0))

→f(ss(0+(s0*0)))

→f(ss(s0*0))

→f(ss0)

→f(s0)*ss0

→f(s0)+(f(s0)+s0)

→(f(0)*s0)+(f(s0)+s0)

→(f(0)*s0)+((f(0)*s0)+s0)

...

リダクション例２

f({s0*ss0})

→f(s0+(s0*s0))

→f(s0+(s0+(s0*0)))

→f(s0+(s0+0))

→f(s0+s(0+0))

→f(s0+s0)

→f(s(0+s0))

→f(ss0)

→f(s0)*ss0

→f(s0)+(f(s0)+s0)

→(f(0)*s0)+(f(s0)+s0)

→(f(0)*s0)+((f(0)*s0)+s0)

...

リダクション例１，２は書き換えを始める項（初期項）に，戦略に関する表明を入れた例である．これと似たような効果を得るために，ＴＲＳプログラム自身に表明を入れることも可能である．そのような戦略を表明したプログラムがＴＲＳプログラム３，４である．これらは，ｆの部分項ｘに代入された項をそれぞれプログラム３では並行最外リダクションで，プログラム４では最左最内リダクションで求めてから，右辺の項へ書き換えることを表明している．その他に関しては並行最外リダクションである．

ＴＲＳプログラム３

f(0)▷s0

f(s[x])▷f(x)*s(x)

f(s0*ss0)

→f(s0+(s0*s0))

→f(s(0+(s0*s0)))

→f(s[0+(s0*s0)])

→f(s(s0*s0))

→f(s(s0+(s0*0)))

→f(ss(0+(s0*0)))

→f(ss(s0*0))

→f(ss0)

→f(s0)*ss0

→f(s0)+(f(s0)+s0)

...

TRSプログラム4

f(0)▷s0

f(s{x})▷f(x)*s(x)

f(s0*ss0)
→f(s0+(s0*s0))
→f(s(0+(s0*s0)))
→f(s{0+(s0*s0)})
→f(s(0+(s0+(s0*0))))
→f(s(0+(s0+0)))
→f(s(0+s(0+0))
→f(s(0+s0))
→f(ss0)
→f(s0)*ss0
→f(s0)+(f(s0)+s0)
...

上記においてf(s0)*ss0→f(s0)+(f(s0)+s0)のリダクションが，これはf(s0)*ss0を並行最外リダクションに依り求めているためである．そのため，次の項が大きくなってしまう．f(s0)はいずれ正規形に到達することがわかるので，f(s0)*ss0全体のリダクションに先立って，f(s0)のリダクションを行なうのがよい．その際，部分項f(s0)に対して並行最外戦略を用いるか，最左最内戦略を用いるかを[]，{}により表明する．TRSプログラム5は並行最外戦略の使用を表明したものである．

TRSプログラム5

f(0)▷s0

f(s{x})▷[f(x)]*s(x)

f(s0*ss0)
→f(s0+(s0*s0))
→f(s(0+(s0*s0)))
→f(s{0+(s0*s0)})
→f(s(0+(s0+(s0*0))))
→f(s(0+(s0+0)))
→f(s(0+s(0+0)))
→f(s(0+s0))
→f(ss0)
→[f(s0)]*ss0
→([f(0)]*s0)*ss0
→(s0*s0)*ss0
→(s0*s0)+((s0*s0)*s0)
...

TRSプログラム5は，fに関してはLispのような通常の言語のもつ評価順序に近い戦略を表わしている．

## 3．2 A－TRSの定義

まずはじめにA－TRSで用いる項の集合ATermを定める．これは，Termを部分集合としている．

[定義]（A-TRSの項，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∈Var の時 x，[x]，{x}∈ATerm

（2） a()∈Const の時 a()，a[]，a{}∈ATerm

（3） f∈Func，at1,...,atn∈Termの時
f(at1,...,atn)，f[at1,...,atn]，
f{at1,...,atn}∈ATerm

[]，{}を表明とよび，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．

定義中，（2）において，定数a()，a[]，a{}は各々 a，[a]，{a}と表記することがある．（3）においてf自身の取り得る引数の数（アリティ）nは重要でないので省略している．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．例えば+[{a},x]は[{a}+x]のように表記する．

[定義]（A－TRSの書き換え規則，A－TRS）

AP▷AQの形をしたものをA－TRSの書き換え規則という．ここで，APとAQは変数自身でも表明付きの変数でもないA－TRSの項であり，AQに出現する変数（表明付きを含む）は必ずAPにも出現するものとする．またAP中の表明は変数にしか付いていないものとする．表明付き項書き換え系A－TRSとはこのような書き換え規則の有限集合である．

## 3．3 A－TRSのreducer

[定義] (Arp, Annotarion replacement)

Arp : ATermΩ → TermΩ

Arp(at) = a, Rpo(a), Rli(a),t≡a, [a], {a} resp.

f(Arp(at'1), ... ,Arp(at'n))

at≡f(at'1, ... ,at'n)

Rpo(f(Arp(at'1), ... ,Arp(at'n)))

at≡f[at'1, ... ,at'n]

Rli(f(Arp(at'1), ... ,Arp(at'n)))

at≡f{at'1, ... ,at'n)}

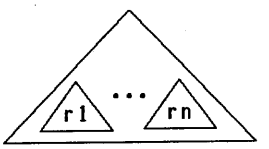
[定義] (A-TRSのreducer Rat,Reduce Annotated Term)

入力された項がatである時

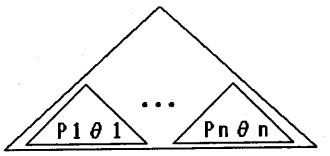
1 Ａｒｐ(at)を求める

2 1の結果において，リデックスを探す

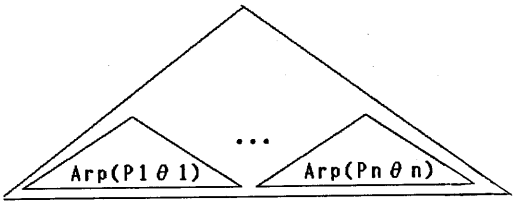
2．1 最外リデックスを求める．すなわち，ｒi≡Ｐiθiとなる，規則Ｐi▷Ｑi，θi＝ {at1/x1,...,atm/xm} を求める．



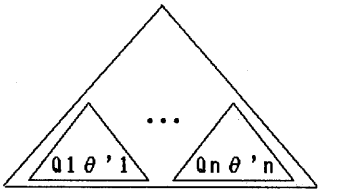

2．2 各ｒiをＰiθiで置き換える




3 θi'＝ {Arp(at1)/x1,...,Arp(atm)/xm} を求める．




4 Ｑiθi'に置き換える




[性質３．１] (Ratの能力)

(1)表明を含まない規則よりなるA-TRSにおいて

Rat = Rpo

(2)規則の右辺の項がeagerな表明を持つ項であるとき

Rat = Rli

(3)一般のＡ－ＴＲＳに対して

Rpo ≧ Rat ≧ Rli

このようにＡ－ＴＲＳのreducerはRpoとRliの中間的能力を持ち，その度合は，Ａ－ＴＲＳ中に表明をどの程度入れるかによって調節することができる．このようにして能力の度合，書き換えステップの調節をＴＲＳプログラムを書くユーザーが調節することができる．調節の仕方によっては，存在する正規形が得られない場合もあるが，それはユーザの責任である．それがユーザーの表明のいれ方によって求まらないのか，もともと求まらないのかは，いちどすべての表明を取り去った後にRatへ入力し（すなわちRatの能力をRpoと同等の能力にし）正規形が求まるか否かを確かめればよい．

３．５ deley表明

Ａ－ＴＲＳのシンタックスに次の（４）を加える．

（４） f∈Func, at1,...,atn∈Term の時

<f(at1,...,atn)>∈ATerm

このとき<f(at1,...,atn)>をdeley annotated termといい＜＞をdeley表明という．これに対するＡｒｐの規則は

Arp (<f(at1,...,atn)>) ＝ f(at1,...,atn)

と定め，ArpをATermΩ→AtermΩに拡張する．このdeley表明を用いると，ＴＲＳプログラム６が作成することができる．

ＴＲＳプログラム６

```
if(true,<x>,<y>)▷x
if(false,<x>,<y>)▷y
eq0(0)▷true
eq0(s(x))▷false
sub1(s(x))▷x
f(x)▷if(eq0{x},s0,<x*f[sub1[x]]>)
```

ＴＲＳプログラム６を用いた実行例.

```
f(ss0)→if(eq0{ss0},s0,<ss0*f[sub1[ss0]]>)
      →if(false,s0,ss0*f[sub1[ss0]])
      →ss0*f[sub1[ss0]]
      →ss0*f[s0]
      →ss0*if(eq0{s0},s0,<s0*f[sub1[s0]]>)
      →ss0*if(false,s0,s0*f[sub1[s0]])
      →ss0*(s0*f[sub1[s0]])
      →ss0*(s0*f[0])
      →ss0*(s0*(if(eq0{0},s0,<0*f[sub1[0]]>)
      →ss0*(s0*(if(true,s0,<0*f[sub1[0]]>)
      →ss0*(s0*s0)
      →ss0*(0+s(0+s0))
      →ss0*s(0+s0))
      →ss0+(ss0*(0+s0))
      →...
```

実行例から分かるように，この表明を用いると通常のif-then-elseと同じような戦略をもつプログラムを作成できる．deley表明はLisp Kit Lisp[2]におけるdeley関数に対応する．また，Lisp Kit Lisp のforce関数はＡ－ＴＲＳではArpに対応する．このようにdeley表明は実際にＴＲＳプログラムを作成する上で利用できる，ad hoc的機能である．しかしながらdeley表明をもつ規則対して，３．３節で述べた能力を明確にすることは困難である．また，deley表明は他のdeley表明と対になって（例えばプログラム６中では,if(...)▷...中の左辺のdeley表明と...▷if(eq0{x},s0,<x*f[sub1[x]]>中のdeley表明)その効果を発揮するため，他の表明のように一つの規則に着目して，その中の表明のみで効果を理解することが困難である．

## ４　Ａ－ＴＲＳの実現，評価

### ４．１　実現

Ａ－ＴＲＳのreducer Ｒａｔ はLispマシンFACOM-α上にUtiLispを用いてインタプリタ方式によって作成した．このreducer Ｒａｔはユーザーと対話的に実行を行なう機能は全く有しておらず，シュミレータ的要素の強いものである．例えばＴＲＳプログラム自身もreducer の一部として記述する必要がある．しかしながら，余計な機能を持たず，より忠実に定義に沿い実現されているため，上記Lispで約２００行（内部に保持しているＴＲＳプログラム部分をのぞく），２０数個足らずの関数で実現されている．内訳は，Ｒｐｏ独自の機能のために約２個，Ｒｌｉ独自の機能のために約３個，共通に用いる補助関数のために約１０個，またＲａｔの機能のために約６個の関数を用いている．

reducer Ｒａｔはほぼ定義に沿って記述されているが，実行効率の観点から以下のような手法を用いている．

・規則の格納方法

Ｒｐｏ，Ｒｌｉ，Ａｒｐ及びそこで用いられるmatchは表明の無いＴＲＳプログラムに対応しており，Ａ－ＴＲＳは表明の無い通常のＴＲＳを含んではいるものの，matchの適用のたびに表明を外していたのでは効率が悪い．そこで，これらの関数用にあらかじめ表明をすべて取り去ったＴＲＳプログラムも，表明付きのＴＲＳプログラムとともに内部にリストの形で保持している．表明付きのＴＲＳプログラムはＲａｔ，Ｒａｔ用のmatchでのみ用いられる．

・項の表現方法

項はLispのリストを用いて素直に実現している．従ってポインタの操作による部分項の共有，ポインタ操作による書き換えの模倣等の処置はいっさい行なっていない．しかしながら，一度正規形になった項に関しては正規項である旨のタグ付をつけ，Ｒｐｏ，Ｒｌｉ，Ｒａｔ等で複数回にわたり同一の正規項に対しての判断（nfによる判断）による無駄を避けている．これは，各reducerがそのはじめに当たって，常に正規形かをチェックすることによる効率の低下を防ぐためである．

### ４．２　評価

図４．１に３．１節で述べた各例に沿って，実行結果を挙げる．図中，Rpo by Rat とは表明をはずしたＡ－ＴＲＳでＲａｔによって実行した結果である．図４．２に他の例での結果を載せる．Rpo と Rpo_by_Rat の比較から，ＲａｔがＲｐｏと同様の結果を得るために

| | matchへの入力回数 M | 書き換え回数 rewriteへの入力回数 R | ヒット率 R／M (%) | 実行時間 (msec) |
|---|---|---|---|---|
| Rpo | 106 | 47 | 44.3 | 615 |
| Rpo by Rat | 106 | 47 | 44.3 | 681 |
| Rli | 38 | 21 | 55.3 | 225 |
| ﾘﾀﾞｸｼｮﾝ例1 | 63 | 27 | 42.9 | 327 |
| ﾘﾀﾞｸｼｮﾝ例2 | 58 | 27 | 46.6 | 338 |
| ﾌﾟﾛｸﾞﾗﾑ3 | 57 | 27 | 47.4 | 327 |
| ﾌﾟﾛｸﾞﾗﾑ4 | 59 | 27 | 45.8 | 340 |
| ﾌﾟﾛｸﾞﾗﾑ5 | 48 | 21 | 43.8 | 237 |

図４．１　評価結果（f(s0*ss0)）

| | matchへの入力回数 M | 書き換え回数 rewriteへの入力回数 R | ヒット率 R／M (%) | 実行時間 (sec) |
|---|---|---|---|---|
| Rpo | 2376 | 633 | 26.6 | 13.214 |
| Rpo by R | | | | 13.447 |
| Rli | 125 | 71 | 56.8 | 0.866 |
| ﾘﾀﾞｸｼｮﾝ例1 | 1665 | 339 | 20.4 | 7.164 |
| ﾘﾀﾞｸｼｮﾝ例2 | 1643 | 339 | 20.6 | 7.990 |
| ﾌﾟﾛｸﾞﾗﾑ3 | 1655 | 339 | 20.5 | 8.014 |
| ﾌﾟﾛｸﾞﾗﾑ4 | 1654 | 339 | 20.5 | 8.029 |
| ﾌﾟﾛｸﾞﾗﾑ5 | 1002 | 117 | 11.7 | 2.578 |

図４．２　評価結果（f(ss0*ss0)）

生じるオーバーヘッド測定される．わずかのオーバーヘッドではあるが，これにもかかわらず，各リダクション例，プログラム例とも速度が向上している．表明の速度向上に及ぼす効果がこのオーバーヘッドを打ち消すほど現われていることがわかる．

表明付きのリダクションは，ＲｐｏとＲｌｉの速度の中間に位置している．Ｒａｔは能力のみでなく速度もその中間的であることが分かる．また，図４．１，図４．２ともプログラム６の実行速度の向上が著しい．右辺の表明が全体の速度向上に寄与する度合が大きいことが分かる．

## ５．まとめ

本論文では，我々がすでに提案している，戦略に関する表明をＴＲＳプログラム中に記述することのできる項書換え系Ａ－ＴＲＳの処理系を作成，それに基づき評価を行なった．Ａ－ＴＲＳは，各ＴＲＳプログラムに応じた効率のよい戦略，書換えをＴＲＳプログラム中に記述できる．そのため，各ＴＲＳプログラムに応じた戦略を用いることができ，それにより，ユーザーが意図した通りに，書き換えが有効に行なえるＴＲＳプログラムする事ができる．評価の結果，表明により速度的にも有利であることが確認された．

今後，どの様に表明を入れるとうまいプログラムの作成が行えるか，Ａ－ＴＲＳを適切に適用できる応用，そのプログラミングスタイルを考察してゆくことは今後の課題である．またここでは，表明を単なる戦略の表明としてしか扱っていないが，並列にリダクションすることを表明していると見ることもでき，並列処理記述用ＴＲＳと見ることもできる．並列実行のための方式等，今後の課題である．

## 参考文献


[1]二木，外山：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

[2]Henderson,P : Functional Programming Language and Implementation. Printice Hall(1980)

[3]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

[4]稲垣,坂部: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

[5]長田：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

[6]布川,黒田,富樫,野口 :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)pp.5-15,

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

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

[10]酒井,坂部,稲垣：抽象データ型直接実現システムCdimple,第40回関数プログラミング研究会資料(1986)，pp.122-128

[10]杉山，鈴木，谷口，嵩：あるクラスの項書き換え系の効率のよい実行，信学論Ｄ(1982)pp.858-865

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

[12]山本，直井，坂部，稲垣 :項書き換えシステムにおける必須書き換え戦略の効率，信学技報Vol.87，No.237(1987)，pp21-30

[13]山中,直井,坂部,稲垣：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

[14]矢野：項書き換え型計算モデルの実現，１９８５年東北大学卒業論文

[15]矢野，布川，富樫，野口：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，No.4(1988)，pp40-51