ソフトウェア基礎論　17－11
(1986. 7. 4)

項書換えシステムからＬｉｓｐプログラムへの変換系

Term Rewriting System on Common Lisp

戸村　哲　　二木厚吉
Satoru TOMURA　Kokichi FUTATSUGI
電子技術総合研究所
Electrotechinical Laboratory

概要　リスト表記に基づく項書換えシステムTRS(a Term Rewriting System)をCommon Lisp上に作成した。TRSでは最内最左戦略を拡張した戦略リストによる簡約化を実行する。

TRSの実現には項書換え規則による演算子定義をLispの関数定義にコンパイルする方式を採用した。項をLispのプログラムとして実行すると項書換えシステムでの簡約化が行われる。

項のリスト表記とLispへのコンパイル方式を採用したことによってTRSのプログラムからLispのプログラムを自然に呼び出すことができる。また逆にLispのプログラムからTRSのプログラムも自然に呼び出すことができるので、TRS CompilerはLispにパターン照合機能とパターン照合に基づく呼び出し機能を追加したことにもなる。

TRSでの2進木表現にLispでのcons表現を用いた場合には、2進木表現を用いたリストの連結をするappendはLispで直接定義した場合の3分の1の速度で実行する。

Abstract　A techinique for compiling TRS (a Term Rewriting System) into Lisp is proposed. A strategy list can be attached to each operator of TRS for controlling the strategy of rewriting. By giving appropriate strategies to operators, the rewriting strategis which are not leftmost innermost can be realinzed.

Using this technique, TRS can be completely embedded into Lisp. That is, terms are represented by Lisp's list notation and evaluating the terms in Lisp have the same effect as reducing the terms using rewriting rules. Moreover, Lisp's function can be used freely in TRS, and the TRS's operators can be used freely in Lisp.

This technique is also seen as showing how to add the mechanism for defining and invoking functions using pattern matching.

The function defined in TRS and compiled by this method runs 10 to 100 times slower than the same function defined in Lisp.

0.はじめに

プログラミング言語設計支援システム「つくばね」[1]の一環としてリスト表記に基づく項書換えシステムTRS(a Term Rewriting System)をCommon Lisp[2]上に作成した。TRSは簡約化のための戦略を最内最左戦略を拡張した戦略リストで指定する。実現方式としては書換え規則による演算子定義をLispの関数定義にコンパイルする方式を採用している。

本報告ではTRSとTRSで採用した戦略リストによる簡約化戦略について説明し、Lispへのコンパイラ方式について述べる。

1.項のリスト表現

項書換えシステムの対象となる項のリスト表記の構文を拡張BNFを用いて定義する。ただし拡張BNFでは非終端記号は大小記号<,>に囲まれた記号を用い、終端記号には引用符"に囲まれた記号を使う。また[ ]は0回または1回の出現を表し、*は零回以上の繰り返しを表すものとする。

<項>　::= <演算項> | <定数> | <変数>
<演算項> ::= "(" <演算子名> <項>* ")"

字句単位(lexical unit)はLispの字句単位に準ずる。ただし、<定数>はLispにおいて定数として扱えるものであり、数値やt,nilなどを含む。<変数>は名前(symbol)の特殊なものであり、先頭の文字が下線(under score)"_"で始まるものである。<演算子名>は<変数>でない名前である。特に変数を含まない項を定数項という。

定義により

```
(cons _a 2) , 1 , (a)
```
は<項>であるが、一般のS式である
```
(1 . 2) , ((cons) 1 2) , (_fn 1 2)
```
などは<項>ではない。

2.項の書換え規則

書換え規則(rewriting rule)とは、項の対であり、リスト表記の構文を

```
<書換え規則> ::=
        "(" "==>" <左辺項> <右辺項> ")"
```

と定義する。但し、右辺項に出現する変数は左辺項に出現したものだけである。従って例えば
```
(==> (f _a) (g _a _b))
```
は右辺項の変数_bが左辺項に現れていないので書換え規則ではない。

定数項tに対して書換え規則(==> l r)の左辺項lとのパターン照合を行い、合致した場合にtとlとを等しくする変数置き換えsを右辺項rに適用した項s(r)を得ることを書換えという。

定数項およびその部分項に書換えが可能なときに1回以上の書換えを行うことを簡約化という。

定数項tに対して簡約化を行った結果、それ以上簡約化できない項sが得られた場合、sをtの既約形といい、定数項tから既約形sを求めることを既約化という。

3.書換え規則を用いた演算子定義

例えば階乗を計算する演算子factの定義は、二つの書換え規則
```
( ==> (fact 0) 1)
( ==> (fact _n) (* _n (- _n 1)))
```
によって表現できる。ただし*,-に適当に定義されているものとする。3!の値を計算するには(fact 3)の書換えを行い、その既約形を求めることになる。

書換え規則は演算子の写像関係を定義域と値域に対応する項の対として表現したものであり、演算子の名前を含む形式で表現している。書換え規則を演算子の名前に依存しない写像自身で記述する記法として、lambda形式を拡張した拡張lambda形式を導入する。拡張lambda形式はlambda形式と同様に引数リストとその値とからなる。リスト表記での構文は

```
<拡張lambda形式> ::=
        "(" "lambda" "(" <項>* ")" <項> ")"
```

と定義する。拡張lambda形式とlambda形式の相違は、lambda形式では引数リストは要素がすべて異なる変数でなければならないのに対し、拡張lambda形式では任意の項のリストとなっている点である。

演算子factを定義する書換え規則を拡張lambda形式で表現すると、それぞれ
```
(lambda (0) 1)
(lambda (_n) (* _n (- _n 1)))
```
となる。一番目の拡張lambda形式は引数が0という定数となっているのでlambda形式ではない。

演算子を拡張lambda形式の組によって定義する。

```
<演算子定義> ::= "(" "defrule" <演算子仕様>
                    <拡張lambda形式> ")"
<演算子仕様> ::=
     "(" <演算子名>
        [":strategy" <戦略指定>] ")"
<戦略指定>  ::=
  "(" ( <自然数> <戦略リスト> )*  ")"
<戦略リスト> ::= "(" <非負整数>* ")"
```

<戦略指定>とは演算子の簡約化戦略を指定するものであり、演算子の引数個数とその引数個数の演算子に対する<戦略リスト>とを並べてリストにしたものである。

<戦略リスト>とは非負整数のリストであり、0から演算子の引数個数nまでの数字を並べたものである。3引数の演算子の場合(1 2 3 0),(3 0 1 0), ()などが戦略リストである。

引数個数nに対する戦略リストが省略されている場合には、1からnまでの数字を昇順に並べ、最後に0を並べた戦略リストを指定したものと解釈する。4引数の演算子の場合、省略時の戦略リストは(1 2 3 4 0)である。

演算子factの拡張lambda形式による定義は
```
(defrule (fact :strategy ( 1 (1 0) ))
  (lambda (0) 1)
  (lambda (_n) (* _n (- _n 1))))
```
と表現する。

4.簡約化戦略

戦略指定を用いた簡約化戦略はOBJ2におけるE-strategy[2]に基づくものであり、簡約化の手続きは次の通りである。

簡約化手続き

a.項tが定数の場合はその定数が簡約化の結果である。

b.項tがk引数の演算項である場合には、その演算子fの戦略指定からk引数の場合の戦略リストを取り出す。

1.戦略リストが空の場合にはその時点での項が簡約化の結果である。

2.戦略リストが空でない場合には戦略リストの先

頭の要素kを取り出す。

7.要素kが0でない場合には、演算項tの第k番目の引数を簡約化し、b.1へ戻って残りの戦略リストの処理をする。

イ.要素kが0の場合には演算子名fの書換え規則の左辺項と演算項tとのパターン照合を行う。

A.パターン照合に合致する規則がある場合には、その規則の右辺項の変数をパターン照合が合致したときの変数置き換えに従って置き換えた項を簡約化した項を簡約化の結果とする。。

B.パターン照合に合致する規則がない場合には、b.1へ戻って残りの戦略リストの処理をする。

5.戦略リストの記述能力

戦略リストを省略した場合の簡約化戦略は最内最左戦略に相当する。即ち簡約化を行う項の部分項の中で最も内側でかつ最も左側にある書換え可能な部分項から順に書換えを行う。

最内最左戦略以外の簡約化戦略を指定する必要のある例としては、条件判断に対応する演算子ifがある。ifを書換え規則で定義すると、

```
(defrule (if)
  (lambda (t   _1 _2) _1)
  (lambda (nil _1 _2) _2))
```

となる。しかしこのままでは第2引数と第3引数を常に簡約化することになる。これを避けるためには、

```
(defrule (if :strategy ( 3 (1 0 2 3) ))
  (lambda (t   _1 _2) _1)
  (lambda (nil _1 _2) _2))
```

と定義すればよい。この場合には、まず第1引数を簡約化し、それがtであるときは第2引数だけを簡約化し、第1引数を簡約化したものがnilであるときには第3引数だけを簡約化することになる。但し第1引数を簡約化したものがtでもnilでもない場合には第2引数と第3引数の両方を簡約化し、それぞれ第2、第3引数に代入したものが簡約化した結果となる。

戦略リストは最内最左戦略を表現できるけれども、一般の最外最左戦略を表現することはできない。しかし、特殊な場合として書換え規則が次の4条件

1.左辺項の引数は定数または変数だけである。

2.左辺項の引数の定数は既約形である。または他の書換え規則と重なりがない。

3.左辺項には同じ変数は高々1回しか現れない、つまり左線形である。

4.左辺項の引数を左から順にみて一旦変数が出現するとその後は変数のみである。

を満たす場合には、戦略リストで最外最左戦略を指定できる。最外最左戦略に対応する戦略リストは1から引数個数までの数字を昇順に並べ、各書換え規則の引数に出現する定数の個数の数字の右隣に0を入れたものである。例えば

```
(defrule (f1)
  (lambda (a _1 _2 _3) _1)
  (lambda (b  c  d _1) _1))
```

の場合にはこの4条件を満足しており、最外最左戦略のための戦略リストは(1 0 2 3 0 4)である。

6.戦略リストの安全性

TRSの簡約化戦略は項の既約形を求めるとは限らない。例えば演算子lconsを

```
(defrule (lcons :strategy (1)))
```

と定義するとlconsの簡約化では第1引数だけしか簡約化しないので、

```
(lcons (+ 2 3) (+ 4 2))
```

を簡約化しても

```
--> (lcons 5 (+ 4 2))
```

のような結果が得られるだけで、既約形

```
--> (lcons 5 6)
```

とはならない。簡約化が既約化になるかどうかは戦略リストの性質である。

いまある戦略リストに従って任意の項を簡約化した結果が既約形である時、その戦略リストの指定は安全であるということにする。この安全性は項書換えシステムの戦略リスト全体に対する性質であるが、一つの演算子の戦略リストの局所的性質として安全性を次のように定義する。ある演算子の書換え規則に対する戦略リストが安全であるとは、項書換えシステムの他の戦略リストが安全であるときにその戦略リストを付け加えても全体が安全であることをいう。

n引数演算子に対する戦略リストが安全であるための十分条件は、戦略リストに1からnまでのすべての数字が出現し、左から順に戦略リストを見たときに1からnまでの数字がすべて出現した後に1回以上0が出現することである。例えば3引数演算子の戦略リスト(0 1 0 3 0 2 0 1)は安全な戦略リストである。

この条件は安全であるための必要条件ではない。例えば先の例で上げた3引数のifに対する戦略リスト(1 0 2 3)はこの条件を満たさないが、安全な戦略リストである。

先に示したlconsの戦略リストは安全ではないが、注意深く使うことで遅延計算（ストリーム、無限リストなど）を実現することができる。

7.TRS Compiler

TRS Compilerは、TRS上での簡約化をLispのプログラムの実行に変換するものである。リスト表記の項tをLispのプログラムとして実行した結果sが項tを簡約化したものとなるように、TRSでの演算子定義をLispでの関数定義に変換する。

TRS CompilerはTRSで戦略リストに基づく簡約化作業を展開してLispの関数定義に変換する。TRS Compilerが行っているコード生成について

1.不定個引数の処理、
2.パターン照合の処理、
3.構成子の処理、
4.戦略リストの処理

に分けて述べる。

まず演算子fのコンパイル結果を例に示す。

```
(defrule (f)
   (lambda (a)     (g (b))
   (lambda (_x)    (h _x))
   (lambda (_x _y) (i 1 _x)))
```

TRS Compilerはこの演算子定義をLispの関数fの定義に変換する。

```
(defun f(&optional
         (_a1 nil ?s1)(_a2 nil ?s2)
         &rest ?rest)
   (cond ((and ?s2 (null ?rest))
          (match (_a1 _a2)
            ((_x _y) (i 1 _x))
            ( t (list 'f _a1 _a2))))
         ((and ?s1 (null ?s2))
          (match (_a1)
            ((a)  (g b))
            ((_x) (h _x))
            ( t (list 'f _a1))))
         ( ?rest (list_ _a1 _a2 (copy-list ?
         ( (not ?a1) '(f))))
```

a.不定個引数の処理

TRSではとくに演算子の引数個数を演算子毎に固定していないので、演算子定義のコンパイル結果である関数は任意個数の引数による呼び出しを処理する必要がある。そのためコンパイル・コードである関数は、引数すべてがoptional引数であり、また最後には必ずrest引数を指定する。

optional引数の宣言形式は3組形式(var init svar)を用いる。3組の最初であるvarは引数の変数名である。二番目のinitは引数に対応する実引数が与えられなかった場合に引数varの初期値を与える形式であるが、この場合実引数が与えられなかった時の引数の値には意味がないのでnilとしている。最後の変数svarは引数varに実引数が与えられていたかどうか示すboolean変数である。例えば関数fを引数なしで呼び出したときの?s1の値はnilであり、引数をひとつ以上付けて呼び出したときはtとなる。

すべての引数をoptional引数としているので引数の個数が少ないときを処理できる。引数が多い場合はoptional引数で処理した引数の残りがリストとしてrest引数に与えられる。例えば関数fを(f 1 2 3 4 5)で呼び出すと、?restの値は第3引数以降をリストにした(3 4 5)であり、?restの値がnilかどうかで引数が実引数が3個以上あるかどうかが分かる。

これらの情報を用いると実引数の個数を判定できるので、例の示すように実引数の個数に応じて書換え規則とのパターン照合を行っている。

b.パターン照合の処理

パターン照合を行うLispの特別形式(special form)としてmatchを追加した。macthの構文は

```
<match> ::= "(" "match" <テキスト>
              <節>* ")"
<節> ::= "(" <パターン> <本体> ")" |
         "(" "t"       <本体> ")"
```

である。ここで<テキスト>,<パターン>はパターン変数を含むS式である。パターン変数の構文は項のリスト表記の<変数>の場合と同じで、最初の一文字が下線"_"で始まる名前である。

matchはまず<テキスト>に出現するパターン変数をその値で置き換えたS式を作る。そしてこのS式と<パターン>とのパターン照合を上から下の順で節単位に行う。パターン照合が合致すると合致したときの変数置き換えの環境で、<本体>を実行する。ただしパターン"t"は任意のテキストと合致する特別なパターンである。パターン変数の有効範囲は、<テキスト>及び<節>に対して局所的である。つまり<テキスト>や異なる<節>に出現するパターン変数はすべて異なる変数である。<テキスト>中に変数が出現する場合には、それらのパターン変数はmatchの外側でLispの変数として値を持つものとする。

<テキスト>にパターン変数を含む場合は<テキスト>に現れるパターン変数をその値に置き換えたデータを作ってからパターン照合したのと意味は同じである。例えば、

```
(match (_a . _b)
   ((cons _x _y) _x)
   ((pair _x _y) _y))
```

は最初に_aと_bとをconsしてからmatchする、

```
(let ((_val (cons _a _b))
 (match _val
   ((cons _x _y) _x)
   ((pair _x _y) _y))))
```

と意味的には同じである。ただし上の書き方と下の書き方では実際の実行コードは異なる。上の書き方をした場合には<テキスト>のデータを実際に作ることはせずにパターン照合を行う。

matchはマクロで実現しており、パターン照合作業

をコンパイルしてLispのコードに展開する。
例えば先の例の場合はそれぞれ

```
(cond ((and (eq _a 'cons) (consp _b)
            (consp (cdr _b))
            (null (cdr (cdr _b))))
       (car _b))
      ((and (eq _a 'pair) (consp _b)
            (consp (cdr _b))
            (null (cdr (cdr _b))))
       (car (cdr _b))))
```

および

```
(let ((_val (cons _a _b)))
  (cond ((and (consp _val)
              (eq 'cons (car _val))
              (consp (cdr _val))
              (consp (cdr(cdr _val))
              (null (cdr(cdr(cdr _val)))))
         (car(cdr _val)))
        ((and (consp _val)
              (eq 'pair (car _val))
              (consp (cdr _val))
              (consp (cdr(cdr _val))
              (null (cdr(cdr(cdr _val)))))
         (car(cdr(cdr _val))))))
```

に展開する。

c.構成子(constructor)の処理

書換え規則の定義がない演算子をデータ構造を構成するものという意味で構成子を呼ぶ。TRSでは構成子も関数に変換する必要がある。演算子が積極的に構成子として定義している場合、つまり書換え規則のない演算子として定義している場合は、対応する関数定義がされるので特別な処理は必要はない。例えばpairを構成子として定義するには、

```
(defrule (pair))
```

と定義すればよい。

問題は演算子定義が全くない場合の処理である。特別な処理をしないとすると、構成子全てを演算子定義しなければならないので、TRS Compilerは演算子定義のコンパイル中に構成子と思われるものが出現すると、つまり演算子として使われいるが、まだ演算子として定義されていないものを見つけると、構成子としての定義を自動的に行う。演算子fの例の場合に関数iが未定義であれば、

```
(defun i (&rest ?rest)
  (list* 'i (copy-list ?rest))
```

という関数定義を自動的に追加する。こうして書き換え規則中に出現する構成子を自然に実現している。

d.戦略リストの処理

演算子に戦略リストの指定がないときは、実引数を評価して渡せばよいが、戦略リストの指定があると、実引数の評価を遅らせる必要がある。しかしながらCommon lispのlambda引数の指定方法には実引数を評価しないで渡す方法がない。そこで演算子定義に戦略指定がある場合は、演算子名のマクロを定義し、実引数すべてをquoteしてから本来の処理をする関数を呼び出している。例えば、演算子ifを次のように定義する。

```
(defrule (if :strategy (3 (1 0 2 3))))
  (lambda (t   _s1 _s2) _s1)
  (lambda (nil _s1 _s2) _s2))
```

すると対応するLispのプログラムは

```
(defmacro if (&rest ?rest))
  (list* 'if*
    (mapcar #'(lambda (x) (list 'quote x))
            ?rest)))

(defun if* (&optional
      (_a1 nil ?s1)
      (_a2 nil ?s2) (_a3 nil ?s3)
      &rest ?rest)
  (cond((and ?s3 (null ?rest))
        (setf _a1 (eval _a1))
        (match (_a1 _a2 _a3)
         ((t   _s1 _s2) (eval _S1))
         ((nil _s1 _s2) (eval _S2))))
       ((not ?s1) '(if))
       ((not ?s2) (list 'if (eval _s1)))
       ((not ?s3) (list 'if (eval _s1)
                            (eval _s2)))
       (t (list* 'if (eval _a1) (eval _a2)
                 (eval _a3)
                 (mapcar #'eval ?rest)))))
```

というマクロifと関数if*の二つの定義となる。マクロifは引数すべてにquoteをつけてif*への呼び出しに変換する。if*では戦略リストに従って引数の評価とパターン照合を行っている。

このように戦略リストの処理はマクロ機能を用いて実現しているので、TRSを使う上で制約となることがある。TRS Compilerがコンパイルしたコードをlispのインタープリタで使用する場合には制約はないが、TRS Compilerがコンパイルしたコードをlispのコンパイラでコンパイルして使用するときに制約がある。これは関数の呼び出しをコンパイルするときには呼び出す関数が定義されていなくてもよく、実際に呼び出しを実行する時点で定義されていればよい。これに対して、マクロの呼び出しをコンパイルするときにコンパイル時に定義されている必要があるからである。このためTRS CompilerとLisp compi

lerとを併用する場合で、しかも演算子に最内最左戦略以外の戦略を指定する場合には、その演算子が参照されるより以前に定義しておく必要がある。

e.実行速度

TRS Compilerの処理例としてリストの連結を行うappendについて出力コードを示す。

リストを2引数構成子pairを用いて表現した場合の演算子appendの定義は次の通りである。

```
(defrule (append)
   (lambda (nil _x) _x)
   (lambda ((pair _a _b) _x)
           (append _a (pair _b _x))))
```

構成子pairは先に述べたようにLispの関数

```
(defun pair ( &rest ?rest)
   (list 'pair (copy-list ?rest)))
```

に変換する。演算子appendの方は

```
(defun append ((_a1 nil ?s1) (_a2 nil ?s2)
               &rest ?rest)
   (cond
    ((and ?s2 (null ?rest))
     (match (_a1 _a2)
       ((nil _x)  _x) )
       ((pair _a _b)_x)
        (append _a (pair _b _x)))
       ( t (list 'append _a1 _a2)))
    ((null ?s1) '(append)
    ((null ?rest) (list 'append _a1)
    (t (list_ 'append _a1 _a2
              (copy-list rest))))))
```

に変換する。更にmatchの部分をマクロ展開すると、

```
(defun append ((_a1 nil ?s1) (_a2 nil ?s2)
               &rest ?rest)
   (cond
    ((and ?s2 (null ?rest))
     (cond
      ((null _a1) _a2)
      ((and(consp _a1)(eq (car _a1) 'pair)
           (consp (cdr _a1))
           (consp (cdr(cdr _a1)))
           (null (cdr(cdr(cdr _a1)))))
       (append (car(cdr _a1))
               (pair (car(cdr(cdr _a1)
                     _a2)))))
      (t (list 'append _a1 _a2)))
    ((null ?s1) '(append)
    ((null ?rest) (list 'append _a1)
    (t (list* 'append _a1 _a2
              (copy-list rest))))))
```

がappendのオブジェクト・コードである。

このappendの実行速度は直接Lispのプログラムでappendを定義したものに比べると100分の1程度であった。この予想外に実行速度が遅い原因は、2進木をあらわすpair(X,Y)を表現するのにLispのconsセル3個を必要とするリスト表現(pair _X _Y)を用いているためである。

そこで2進木を構成子pairで表現する代わりにLispのconsセルを直接用いてappendを定義した。つまりappendを

```
(defrule (append)
  (lambda (nil _x) _x)
  (lambda ((_a . _b) _x)
          (append _a (cons _b _x))))
```

と定義する。オブジェクト・コードは、

```
(defun append ((_a1 nil ?s1) (_a2 nil ?s2)
               &rest ?rest)
   (cond
    ((and ?s2 (null ?rest))
     (cond
      ((null _a1) _a2)
      ((consp _a1)
       (append (car _a1)
               (cons (cdr _a1) _a2)))
      (t (list 'append _a1 _a2)))
    ((null ?s1) '(append)
    ((null ?rest) (list 'append _a1)
    (t (list_ 'append _a1 _a2
              (copy-list rest))))))
```

となる。この場合の実行速度はLispのappendの3分の1程度であった。

参考文献

[1]戸村、二木:「『つくばね』計画の概要」、情報処理学会研究会資料、85-PL-1-1(1985).

[2]Guy L. Steele Jr.:"Common Lisp The language,"Digital Press,(1984).

[3]K. Futatsugi et al. :"Principles of OBJ2," Proc. of 12th ACM Sympo. on POPL, pp.52-66(1985).

[4]二木、外山:「項書換えシステムとその応用」、情報処理、Vol.24(2), pp.133-146(1983).