記　号　処　理 73－4
（１９９４．３．１８）

コンビナトリー項書換え系に基づく関数論理型言語の設計と実装

浜名 誠　西岡 知之　中原 鉱一　井田 哲雄

筑波大学

関数論理型言語とは関数型言語と論理型言語を融合したプログラミング言語である．本稿で述べる関数論理型言語 Ev は，関数型言語の特徴である高階関数，遅延評価，及び論理型言語の論理変数や非決定的な実行といった特徴を全て備えている．Ev はコンビナトリー項書換え系とナローイングという理論的背景を持ち，効率の良い実行のために設計された計算系 LNC に基づいている．

本稿では関数論理型言語 Ev とそのインタプリタの実現方法について述べる．特に Ev のプログラムをコンビナトリー項書換え系とみなすための変換方法を与え，Ev と理論とのつながりをを示す．

A Design and Implementation of A Functional-Logic Language Based on Combinatory Term Rewriting Systems

Makoto Hamana, Tomoyuki Nishioka,
Koichi Nakahara, Tetsuo Ida

University of Tsukuba

Functional-logic language is a programming language amalgamating functional and logic languages. The functional-logic language Ev described in this paper has both features of functional languages such as higher-order functions and lazy evaluations, and of logic languages such as logical variables and nondeterministic evaluations. The language Ev is theoretically based on combinatory term rewriting systems and narrowing. For efficiency, we employ the narrowing calculus LNC as the computational model of Ev.

Here, we describe the functional-logic language Ev and its implementation techniques. Especially, we offer the transformation methods to regard a program of Ev as a combinatory term rewriting system, and next we show a connection between Ev and the theories.

1 はじめに

関数論理型言語は関数型と論理型言語の特徴を兼ね備え，しかも両者を融合した言語である．

関数型言語における項の簡約，論理型言語における節の導出は，両者を特徴付ける最も基本的な概念であるが，本稿で述べる我々の関数論理型言語 Ev ではこの二つの概念がナローイングという概念によって統合されている．

関数型言語の特徴である高階関数，遅延評価などは，数学的に厳密に定義された簡約の概念を基にしている．また論理型言語の場合の論理変数や非決定的な実行といった特徴は，節の導出の概念に由来する．関数論理型言語 Ev は，このような特徴をすべて備えたプログラミング言語である．

Ev は，**条件付き項書換え系** (conditinal term rewriting system，以後，CTRS と略す) を言語の基礎とし，ナローイング計算系 LNC を計算機構とする．先に述べたような関数論理型言語の特徴，すなわち論理変数を用いた計算や，高階関数の取り扱いが可能である．遅延評価により計算を行うため，無限長のデータを扱うことも可能である．さらに各種の糖衣構文により，可読性の高いプログラムを作成することができる．

2 Ev のプログラム例

まず Ev のプログラミング例として，論理回路のシミュレータをあげる．

回路中の電位の状態を，`Hi` と `Low`，`Undef` で表すとする．時刻 t の時の電位が v であることを，組 (t,v) で表す．回路の配線上では，このような電位の変化が連続して発生している．そこで，組 (t,v) のリストを使って配線を表すことにする．

次に，回路中の各ゲートの動作について考える．各ゲートは，確定した (`Undef` でない) 入力を受けとってから一定の時間の後に，入力に対する演算結果を出力する．つまり，各ゲートは，ゲートで行なわれる演算と，演算にかかる時間 (ディレイ)，入力をパラメータとしてモデル化することができる．この考え方に基づくと，ゲートをつくり出す関数は，図 1のように書くことができる．ここで，`valid?` は入力が確定したかどうかを見る関数，`gate1`，`gate2` は，それぞれ 1 入力，2 入力のゲートに相当する関数ををつくり出す高階関数である．

`gate2` 内部で定義されている関数 `f` は，3 つの引数を持つ．ひとつは，入力線のその時点での状態を示す引数で，あとの 2 つは入力線である．`f` の定義は，入力がなくなった時に停止するための部分 (1,2 行目)，入力が確定するまで待つ部分 (3,4 行目)，出力を計算し，出力の変化を出力線にのせる部分 (5-7 行目) からなる．

このように，Ev のプログラムは，(条件付き) 等式の集合として記述する．`:-` の右側が条件部分であり，直観的には，条件部分が成立する時に，その等式が有効であるということを意味している．

この関数を使って，特定の機能を持つゲートを次のようにして作ることができる．ここで，1 入力ゲートの遅れを 1，2 入力ゲートの遅れを 2 と仮定した．

```
notF Hi  = Low
notF Low = Hi
notGate  = gate1 notF 1

andF Low x  = Low
andF Hi  x  = x
andGate     = gate2 andF 2
```

`xorGate` や `orGate` 等も同様に定義することができる．こうしてできたゲートは，組み合わせてさらに大きな回路を作ることができる．たとえば，半加算器と全加算器は次のように書くことができる．

```
sumOf   (s,c) = s
carryOf (s,c) = c

halfAdder in1 in2 = ( s, c )
  where s = xorGate in1 in2
        c = andGate in1 in2

fullAdder in1 in2 inc = ( s, c )
  where si = sumOf (halfAdder in1 in2)
        ci = carryOf (halfAdder in1 in2)
        s  = sumOf (halfAdder si c)
        c  = orGate ci (carryOf (halfAdder si c))
```

次に，プログラムの実行例を示す．Ev におけるプログラムの実行とは，与えられたプログラムのもとで，与えられた質問を成立させるような，質問中の変数の値を求めることである．先のプログラムのもとで，次のような質問を出す．これは，全加算器に入力を与え，その出力を求める質問である．

```
?- fullAdder [(2,Hi), (3,Low),(5,Hi)]
             [(2,Hi), (4,Low),(6,Hi)]
             [(2,Low),(4,Hi), (6,Low)]
       == result
```

result = ([(4,Low), (6,Hi), (7,Low), (8,Low)],
[(6,Hi), (7,Low), (8,Low), (9,Hi)])

この例の場合，等式を満たすような変数 `result` の値を計算している．これは，関数型言語的な使い方である．

```
valid? Hi    = True
valid? Low   = True
valid? Undef = False

gate1 func delay = f
  where f []                = []
        f ((t,v) : rest) = f rest                          :-  valid? v == False |
        f ((t,v) : rest) = (t+delay, func v) : (f rest) :-  valid? v == True |

gate2 func delay = f (0,0)
  where
    f (0,0) []   w2              = []
    f (0,0) w1   []              = []
    f (p1,p2) ((t1,v1) : w1s) ((t2,v2) : w2s) = f (p1,p2) w1s ((t2,v2) : w2s)
                :- valid? v1 == False |
    f (p1,p2) ((t1 v1) : w1s) ((t2,v2) : w2s) = f (p1,p2) ((t1,v1) : w1s) w2s
                :- valid? v1 == True, valid? v2 == False |
    f (p1,p2) ((t1,v1) : w1s) ((t2,v2) : w2s) = ((t1+delay), func v1 p2) : f (v1,p2) w1s ((t2,v2) : w2s)
                :- valid? v1 == True, valid? v2 == True, t1 < t2 |
    f (p1,p2) ((t1,v1) : w1s) ((t2,v2) : w2s) = ((t1+delay), func v1 v2) : f (v1,v2) w1s w2s
                :- valid? v1 == True, valid? v2 == True, t1 == t2 |
    f (p1,p2) ((t1,v1) : w1s) ((t2,v2) : w2s) = ((t2+delay), func v1 v2) : f (p1,v2) ((t1,v1) : w1s) w2s
                :- valid? v1 == True, valid? v2 == True, t1 > t2 |
```

図 1: ゲートをつくり出す関数の Ev による記述

一方，Ev は論理型言語的な側面も持つので，以下の例に示すような逆向きの計算も可能である．

```
?- fullAdder x y [(2,Low),(4,Hi),(6,Low)] ==
     ( [(4,Low), (6,Hi), (7,Low), (8,Low)],
       [(6,Hi), (7,Low), (8,Low), (9,Hi)])

x = [(2,Hi),(3,Low),(5,Hi)],
y = [(2,Hi), (4,Low),(6,Hi)]
```

これは，先ほどの例で得た答を質問として与え，そのような出力を持つ入力の値を計算させる質問である．

3 Ev の構文

この節では Ev の構文を定義する．Ev の構文は基本構文，及びそれを拡張した糖衣構文によって定義されている．基本構文は 5節で述べる CTRS の構文と対応している．糖衣構文は 4.1節で述べる技法により，基本構文に変換される．

3.1 基本構文

Ev の基本構文は次の文法で与えられる．ここで *Var* は変数の集合，*Con* は構成子記号の集合，*Fun* は関数記号の集合で，互いに素なものであるとする．*Var* と *Fun* の要素は英小文字から始まる文字列，*Con* の要素は英大文字から始まる文字列で書く．□で囲んだ語は，Ev の字句 (token) を示す．

以下の文法中で定める r 中の $f\ \ t_1 \ldots t_n$ を左辺，t を右辺，$e'_1, \ldots, e'_m$ をガード部，$e_1, \ldots, e_l$ を条件部と呼ぶ．最左に出現する記号が，関数記号である項を関数項，構成子記号である項を構成子項と呼ぶ．また，f の部分が同一の書換え規則の集まりを関数定義と呼ぶ．

Evの文法

変数	v	$\in$	Var
構成子記号	c	$\in$	Con
関数記号	f	$\in$	Fun

項	t	::=	$(t_1\ t_2) \mid v \mid c \mid f$
厳格等式	e	::=	t_1 [==] t_2

書換え規則

$$r ::= f\ t_1 \ldots t_n\ [=]\ t\ [:-]\ e'_1\ [,] \cdots [,]\ e'_m\ [|]\ e_1\ [,] \cdots [,]\ e_l$$

$$\text{プログラム}\quad p ::= [\{]\ r_1\ [.] \ldots r_n\ [.]\ [\}]$$

3.2 糖衣構文

基本構文に対する拡張として，Ev の糖衣構文を定義する．始めに局所定義の構文及び有効域について述べる．次にレイアウトを考慮した構文を定義する．最後に中置演算子とその他のデータの表現方法について述べる．

3.2.1 局所定義

プログラムのモジュール性の向上のためにも，ある関数のみで使われるような補助的な関数は，その関数の中で局所的に定義し，外からは使用できないことが望ましい．Evではこのような局所的な関数の定義を行うために，where節を用意している．

例えば次のEvのプログラム

```
{ f x = g 8
    where { g y = x*y.}. }
```

において，関数 g は関数 f の右辺のみで有効な局所関数である．g の右辺に現れる変数 x は，f の仮引数の x である．

where節を持つ書換え規則の構文は，基本文法の書換え規則の部分を拡張して次のように表される．

$r ::=\ f\ t_1 \ldots t_n$ [=] t
[:-] e'_1 [,] $\cdots$ [,] e'_m [|] e_1 [,] $\cdots$ [,] e_l
[where] p

以下でいくつかの用語の定義の後，書換え規則中の仮引数および局所関数の有効域を定める．

[{] [}]で囲まれている書換え規則の集まりをブロック，あるブロック b 中に入れ子になって含まれているブロック b' を b からみて下のブロック，逆に b を b' からみて上のブロックと呼ぶ．プログラム中で最も上にあるブロックを大域ブロック，それ以外を局所ブロックと呼ぶ．ある書換え規則の右部とは，次のものをまとめて総称したものである．

- その規則の右辺，ガード部，条件部，
- where節以下にあるすべてのブロックの右辺，ガード部，条件部．

変数，関数記号の有効域

$t_1, \ldots, t_n$ 中の変数の有効域:
r の右部

関数記号 f の有効域:
r の右部，f と同ブロックにある他の書換え規則の右部

$e'_1, \ldots, e'_m, e_1, \ldots, e_l$ 中の変数で $t_1, \ldots, t_n$ 中には現れていない新出の変数の有効域:
$e'_1, \ldots, e'_m, e_1, \ldots, e_l$

p 中で定義される関数 $f_1, \ldots, f_m$:
r の右部

局所定義を持つEvのプログラムは4.1.2節の変換により，局所定義を持たないプログラムに変換される．

3.2.2 レイアウトルール

プログラム作成段階においてある決まった法則で字下げを行い，そして構文解析器が字下げの位置を認識するならば，ブロックを示す字句[{]と[}]，各書換え規則を分離するための字句[.]は，省略されていたとしても復元は可能である．我々はこの字下げの情報をも考慮に入れた構文を定める．

この字下げ情報付きの構文は，以下で述べる字句の補完規則(これをレイアウトルールと呼ぶ)によって定義する．すなわち字下げ情報を考慮する構文の字句列[1]に対して順次次のレイアウトルールを適用し，通常の文脈自由文法に従う構文になるよう字句を補完する．そうすればその後は通常の構文解析が適用できる．

ルール適用時には，レイアウトを処理するレイアウト処理モードかレイアウトは無視する通常処理モードのどちらかの状態にある．新たにモードに入る前には(現在と同じモードあっても)現在のモードを退避する．モードから出た際にはこれを復帰したモードにいることになる．

レイアウトルール

現在注目している字句を t とする．

1. t がプログラムの最初，あるいは[where]の時，
   その直後の字句(桁位置 s から始まるものとする)が

   [{]の時:
   通常処理モードに入る．

   [{]でない時:
   [where]の直後に[{]を挿入し，レイアウト処理モードに入る

2. t が行の終りの字句でレイアウト処理モードの時，
   t の次の字句，すなわち t の次行にある字句が，

   桁位置 s から始まっていた時:
   t のある行と次行は分離されている行とみなし，t の次に[.]を挿入する．

   桁位置 $s'(> s)$ から始まっていた時:
   t のある行の継続行とみなすので何もしない．

[1] 位置情報を持っている．

桁位置 $s''(<s)$ から始まっていた時:

ブロックの終了とみなし，t の次に`}`を加えて．レイアウト処理モードから出る．

3. t が`}`の時，

通常処理モードならば:

通常モードを出る．

レイアウト処理モードならば:

モードを出て，もしまだレイアウト処理モードなら，通常処理モードになるまでモードを出続ける．

4. t がその他の字句の時は，何もしない．

3.2.3 その他の糖衣構文

Ev における中置演算子の集合を Op とし，Op の要素は + や * などの記号文字の列で書く．$op \in Op$ とすると，中置記法 $(t_1\ op\ t_2)$ は，二引数の関数項 $(op'\ t_1\ t_2)$ の別記法であるとみなす．ここで $op' \in Fun$ は op と同じ演算をする関数の関数記号である．(例えば $op = +$, $op' =$ `plus` のように対応している)．演算子には結合の仕方と優先順位とが指定できる．

実際のプログラミングにおいては，数，文字，文字列，組，リスト，などのデータの使用が不可欠である．Ev においてこれらは構成子からなる項の糖衣とみなすことによって容易に扱うことができる．具体的には以下のようにする．

- 組 $(a_1,\ldots,a_n)$ は，構成子項 (`Tuple_`n a_1 ... a_n)
- リスト $[a_1,\ldots,a_n]$ は，構成子項 (`Cons` a_1 (`Cons` ... (`Cons` a_n `Nil`)))
- 文字 c は，構成子 `Char_`c
- 文字列は，文字のリスト

4 実現方法

この節では前節までで述べてきた言語 Ev の実現方法を示す．現在 Ev は，UNIX 上にインタプリタとして実現されている．インタプリタの全体構成を図 2 に示す．以下でこれらを概観する．

処理系は，大きく分けて，フロントエンド，インタプリタ，ユーザインターフェースの 3 つの部分からなっている．各構成部分は，それぞれ別々のプログラミング言語を用いて記述してあり，実行時には別々のプロセスとして動作する．

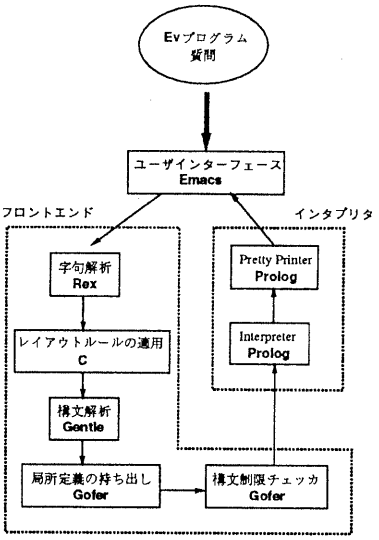

図 2: Ev インタプリタの構成

フロントエンドは Ev のプログラムを構文解析したのち各種変換により糖衣構文を基本構文に展開する．そして 5.1節で述べる制限をチェックし，インタプリタのために構文木をコード化して出力する．構文解析器の実現には，字句解析生成系 Rex[5]，構文解析器生成系 Gentle[11] を用い，文法記述から生成した．レイアウトルール適用プログラムは C 言語を用いてコーディングを行った．糖衣構文の展開，および構文的制限のチェック部分は関数型言語 Gofer を用いてコーディングした．

インタプリタは，フロントエンドの出力を受け，実際の計算を行なう部分である．これの実現には，5.2節で述べる計算系 LNC を Prolog のプログラムとしてコーディングすることにより行った．用いた Prolog は SICStus Prolog である．

ユーザインターフェースは，上の二つの部分を統合し，ユーザとの対話を提供する部分である．ユーザに対して，Prolog とよく似たインターフェースを提供している．ここでは，複数のプロセスを扱うこと，ユーザインターフェース部分の変更の容易さなどから，Emacs Lisp を用いて GNU Emacs 上に実現されている．

以下では，フロントエンドの特に主な部分についてそ

の実現方法を述べる．

4.1 フロントエンドの実現

4.1.1 レイアウトルールの適用

レイアウトルール適用プログラムは，字句解析器と構文解析器の中間に位置する (図 2)．レイアウトルール適用プログラムは字句解析が終了した後の字句の列を受け取り，これに対して 3.2.2節で示したレイアウトルールを適用し，適切な位置に{, }, .を挿入する．構文解析器はこのレイアウトルール適用済みの字句列に対して構文解析を行う．

4.1.2 局所定義の持ち上げ

where節を用いた局所的な関数の定義は，フロントエンドで大域ブロックの関数の定義に変換される．ただし単に定義を大域ブロック持ち上げただけでは変数や関数の有効域が意図するものでなくなってしまうため，有効域も考慮して定義を大域ブロックへ持ち出す．例えばプログラム

```
{ f x = g
    where { g y = x*y. }. }
```

は

```
{ g2 x1 y3 = x1*y3.
  f0 x1    = g2 x1 8. }
```

のように変換される．

このような変換は，関数型言語の実装においてはlambda liftingとして知られている技法である [8]．この技法において関数型プログラムは，λ式の糖衣されたものとしてみなされる．そしてlambda liftingをλ式から等価なスーパーコンビネータの定義をつくりだす技法として定義し，これを関数型プログラム (の糖衣を展開したλ式) に適用する．すると結果的に局所関数の定義は大域ブロックへ持ち出されることになる．

Evはλ計算ではなくCTRSに基づく関数論理型言語であるから，この変換にlambda liftingをそのまま用いることはできない．我々はこの技法を参考にしてEvのために新たに変換手続きを考案した．

以下ではこの局所定義の持ち上げ手続きを示す．

ある書換え規則 (ブロック p にあるとする) の右部の変数 x は，ブロック p をwhere節として持つ規則 r'(ブロック p' にあるとする) の左辺に現れた変数か，あるいは r' より上のブロックに現れた変数ならば，**間接束縛変数**と呼ばれる．上記の例ではwhere節中の書換え規則 `g y = x * y` の右辺に現れる変数 `x` が間接束縛変数である．

持ち上げ手続きは次の3つのステップから構成されている．

Step 1: 名前の付け換え

大域と局所のすべての書換え規則について，変数名，関数名を衝突がないように一意に名前を付け換える．

Step 2: 間接束縛変数の注記

プログラム中の任意のブロック p に対して，そのブロックの間接束縛変数をすべて求め，これらを列にして，p に注記する．

Step 3: 持ち上げ

大域のすべてのwhere節 p に対して以下の手続き lift を行なう．

lift: p には間接束縛変数列 $z_1, \ldots, z_k$ が注記されているとする．p 中の各書換え規則

$$f_i\ t_1 \ldots t_n = t \mathbin{:-} g_1, \ldots, g_m \mid e_1, \ldots, e_l \ \texttt{where}\ q$$

に対して，関数記号 f_i の有効域中に現れる関数記号 f_i をすべて項 $(f_i\ z_1 \ldots z_n)$ で置き換える．この結果，t_i は t'_i に，g_i は g'_i に，e_i は e'_i に変換される．さらに q 中の各書換え規則について再帰的に手続き lift を行なう (すると q 中の書換え規則は全て大域ブロックにに持ち出される)．そして

$$f_i\ z_1 \ldots z_k\ t_1 \ldots t_n = t' \mathbin{:-} g'_1, \ldots, g'_m \mid e'_1, \ldots, e'_l$$

のように新たに引数を加えた規則をつくり，これを大域ブロックの書換え規則として加える．where節中の元の書換え規則は削除する．

以上の3ステップを経るとwhere節の中にあった書換え規則は適切な形で大域に持ち出され，すべての書換え規則はwhere節を持たないものとなる．

5 理論的背景

Evの設計にあたって，その計算モデルとなるナローイング計算系LNCを考案した．LNCは推論規則の集合で与えられるため，これを計算機上に実現すればEvのインタプリタが得られることになる．我々はこの目的のためにPrologを用いることにより，推論規則ほぼ

そのままの形でコーディングを行い，インタプリタ部分を実現することができた．本節では，このLNCを含むEvの背景にある理論について述べる．

5.1 項書換え系とナローイング

CTRSは，次のような形式の書換え規則の集合$\mathcal{R}$で定義される．

$$f(l_1,\ldots,l_n) \to r \Leftarrow c_1,\ldots,c_m$$

この規則は，直観的には，条件部$c_1,\ldots,c_m$が全て成立するときに項$f(l_1,\ldots,l_n)$をrに書き換えることを表す．この書換え規則で定義される記号fを関数記号，それ以外の記号を構成子記号と呼ぶ．構成子記号と変数のみからなる項をデータ項と呼ぶ．

関数論理型言語の計算モデルとして様々な望ましい性質を得るために，条件付き書換え規則には，次の4つの条件を加える．

書換え規則に対する制限:

1. すべての書換え規則$l \to r \Leftarrow C$において，lには同じ変数が二度以上現れない．
2. 任意の2つの書換え規則は重なり(overlap)をもたない．
3. 書き換え規則$l \to r \Leftarrow C \in \mathcal{R}$において，$r$に出現する変数は$l$にも出現する．
4. 条件部には，$s \equiv t$の形式の厳格等式のみを許す．

厳格等式$s \equiv t$は，2つの項sとtが同じデータ項正規形をもつときに成立する等式である．これを次に述べるナローイングと併用すると，プログラミング言語の計算解として望ましいデータ項代入を求めることができるため，多くの関数論理型言語で採用されている[10, 4]．

ナローイング(narrowing)は，項書換え系に対する操作である[3, 6]．これは大まかに言えば次のものである．解を求めたい厳格等式の列をゴールと呼ぶ．まず計算のためのプログラムとして，あるCTRS $\mathcal{R}$を与える．ナローイングはゴールの列(のうちのある厳格等式の部分項)に対して，$\mathcal{R}$の書換え規則を用いて書換えを行う．ただしこの書換えは通常の簡約とは異なり，書換え可能な部分項の特定を，書換え規則とのパターンマッチではなく，単一化を用いて行う．このため，書換え中にゴールを満たすデータ項の(ための中間的な)代入が，部分的に求まっていき，最終的にゴールを満たすデータ項代入を求めることができる．

5.2 ナローイング計算系LNC

ナローイングは，非常に強力な計算機構であるが，その反面，非常に多くの非決定性を持つためプログラミング言語の計算モデルとしては効率が悪い．このためナローイングに戦略を導入し効率化をはかる研究がなされてきた[1, 2]．我々は，Evの遅延評価機構を実現するために項書換え系の標準簡約戦略[7]に着目した．この戦略は項書換え系で必要呼び(call-by-need)を実現する．この標準簡約戦略をナローイングに導入することで，ナローイングに関してもある種の必要呼びが実現される．

さらに，我々は計算機上で効率よくこの戦略を実現するためにナローイング計算系LNCを設計した．この計算系は，CTRSを対象言語とし，7つの推論規則を持つ．これを用い与えられたゴールに対して，適用可能な推論規則を適用していけば，ナローイングを効率的にシミュレートすることができる．

一般にはあるゴールを満たす解は複数個存在するが，関数論理型言語の計算モデルとしてはこの様な解を全て求められることが望ましい．計算系LNCは，与えられたCTRSが5.1節の制限を満たすとき，データ項代入に対して完全性をもつことが証明されている[9]．すなわちLNCを用いて，与えられた等式を満たす全てのデータ項を求めることができる．

この計算系は，適用すべき推論規則が最左の等式の形で決まるように設計されているため，計算機上に実装し易くなっている．

5.3 コンビナトリーCTRS

LNCで扱う5.1節であげた条件を満たすようなCTRSは，一階の関数を定義しているとみなすことができる．Evは高階関数を扱える言語であるので，Evの実現のために，CTRSのうちでも特にコンビナトリーCTRS(combinatory CTRS)と呼ばれる書換え系が必要となる．コンビナトリーCTRSは関数記号としては'Ap'のみを持つCTRSで，項の表現に対しては，変数，構成子記号，作用表現$\mathrm{Ap}(t,s)$(ここでtとsは項である)いずれかの形をとる．次の例は，コンビナトリーCTRSである．ここで map，Nil，Consは構成子記号，f，

x, xs は変数である.

$$\begin{cases} \mathrm{Ap(Ap(map},f\mathrm{),Nil)} \rightarrow \mathrm{Nil} \\ \mathrm{Ap(Ap(map},f\mathrm{),Cons(}x,xs\mathrm{))} \\ \qquad \rightarrow \mathrm{Ap(Ap(Cons,Ap(}f,x\mathrm{))},\mathrm{Ap(Ap(map},f\mathrm{),}xs\mathrm{))} \end{cases}$$

項は左結合であると仮定すれば, Ap といくつかの括弧は省略したとしても混乱は生じない. 例えば先の上の例は, 簡潔に

$$\begin{cases} \mathrm{map}\ f\ \mathrm{Nil} \rightarrow \mathrm{Nil} \\ \mathrm{map}\ f\ (\mathrm{Cons}\ x\ xs) \rightarrow \mathrm{Cons}\ (f\ x)\ (\mathrm{map}\ f\ xs) \end{cases}$$

と書ける. そして容易にこの例は, 高階関数 map を定義しているとみなせる.

5.4 コンビナトリー CTRS としての Ev

我々は, Ev のプログラムをコンビナトリー CTRS であると解釈する. すなわち, 項の適用表現 $t_1\,t_2$ を $\mathrm{Ap}(t_1,\ t_2)$ と解釈する. また `==` を CTRS では厳格等号 $\equiv$ として, Ev の書換え規則における, 字句 `=` を CTRS における記号 $\rightarrow$ として, `:-` を記号 $\Leftarrow$ とそれぞれみなし, ガード部と条件部は両方合わせて CTRS の条件部であると解釈する.

またこれに伴い 5.1節に挙げた CTRS の書換え規則に関する制限は, 対応する Ev のプログラムに対して, 若干弱めた形で課せられる. すなわち重なりを含んだ関数定義でも, 排他的なガードが付けられていれば重なりがないとみなす.

このように解釈すれば, CTRS に対して設計したナローイング計算系 LNC を高階関数を扱う Ev の計算機構として採用することができる.

6 まとめ

本稿では関数型言語 Ev とそのインタプリタの実現方法について述べた.

Ev は CTRS とナローイングという理論的背景を持ち, 効率の良い実行のために設計された計算系 LNC に基づいている. インタプリタの実装はこの LNC の推論規則をほぼそのまま Prolog のコードに書き下すことによって行った. LNC は CTRS に対して設計されているので, 糖衣構文を用いた Ev のプログラムは, CTRS の書換え規則の形式に変換する必要がある. よってこの変換の方法を考案し, 実装を行った. さらにこれらを統括し, ユーザとの対話を行うためのインターフェースを実現した.

今後は Ev のための抽象機械を設計し, これに対するコンパイラを開発する予定である. また Ev の拡張として, 型を導入し型検査機構を処理系に組み込むことを検討している.

参考文献

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proceedings of 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[2] A. M. Bockmayr. *Beiträge zur Theorie des logisch-funktionalen Programmierens.* PhD thesis, Universität Karlsruhe, 1990.

[3] L. Friborg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the 2nd IEEE Symposium on Logic Programming, Boston*, pages 172–184, 1985.

[4] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.

[5] Josef Grosch. Rex - A Scanner Generator. Technical report, GMD research Group at the University of Karlsruhe, 1991.

[6] M. Hanus. Compiling logic programs with equality. In *Proceedings of the 2nd Workshop on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science 456*, pages 387–401, 1990.

[7] G. Huet and J. Lévy. Computations in orthogonal rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honor of Alan Robinson*, pages 395–414. The MIT Press, 1991.

[8] R.J.M. Hughes. Super-combinators: A new implementation method for applicative languages. In *Proceedings 11th ACM Symposium on Principles of Programming Languages*, pages 1–10, 1982.

[9] T. Ida and S. Okui. Outside-in conditional narrowing. *IEICE Transactions on Information and Systems.* to appear.

[10] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[11] Jürgen Vollmer. The Compiler Construction System GENTLE Revision 3.8. Technical report, GMD research Group at the University of Karlsruhe, 1992.