作用型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[†]
アート ミデルドープ[††] 井田哲雄[††]

プログラミング言語Evは，関数型言語の特徴である高階関数，遅延評価，およ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兼ね備えた言語である．Evは，作用型条件付き項書換え系に対して，ナローイングを効率良く実行する計算系LNCを適用するという計算モデルを持つ．構文的には，作用型条件付き項書換え系を基本に，多くの糖衣構文を用意した．これにより，可読性の高いプログラムを書くことが可能である．本論文では，関数論理型言語Evの(1)概要，(2)理論的背景，(3)処理系の実装方法，特に構文処理上の技法について述べる．処理系は，大きく分けて，構文処理を行うフロントエンド部と，実際に計算を行うインタプリタ部，これらを統合するユーザインタフェース部からなる．フロントエンド部では，局所定義の持ち上げや，レイアウトルールの適用を行う．局所定義の持ち上げには，ラムダ持ち上げを応用した技法を用いている．計算系LNCは推論規則の形で記述してあるため，LNCに基づくインタプリタをProlog上に容易に実現することができる．

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

MAKOTO HAMANA,[†] TOMOYUKI NISHIOKA,[†] KOICHI NAKAHARA,[†]
AART MIDDELDORP [††] and TETSUO IDA [††]

In this paper we introduce a functional logic language Ev that amalgamates functional and logic languages. The language Ev has both features of functional languages such as higher-order functions and lazy evaluations, and of logic languages such as logical variables and nondeterministic computations. The computational model of the languages is based on applicative term rewriting systems with narrowing calculus called LNC. LNC simulates lazy narrowing efficiently. We describe in detail the language Ev, its implementation techniques, syntactic processing and interpreter of programs. The implemented system consists of three parts: frontend, interpreter, and user-interface. The frontend carries out the syntactic processes such as applying a layout rule and lifting local definitions similar to lambda-lifting. We further show that Ev interpreter which is a main part of the system is naturally derived from LNC.

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，しかも両者を一つの理論的枠組で融合した言語である．近年，関数論理型言語についての研究が，理論面，実装面共に盛んに行われ，いくつかの言語が提案されている[1)~4)]．

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．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．

簡約および導出の二つの概念は，ナローイングという概念によって統合することができる．本論文では，ナローイングに基づく一つの関数論理型言語を提示し，その実装について述べる．この言語は著者らの間でEvと呼ばれており，本論文でもEvと呼ぶ．

言語Evは，条件付き項書換え系(Conditional Term Rewriting System，以後，CTRSと略す)を言語の理

† 筑波大学工学研究科
Doctoral Course of Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

* 本論文は情報処理学会記号処理研究会で発表した論文「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」(情報処理学会研究報告 94-SYM-73，pp. 25-32)の修正・改良版である．

本研究の一部は，筑波大学TARAプロジェクト研究費および文部省科学研究費一般(C)06680300の支援を得て行われた．

論的基礎としている．Evの操作的意味は，CTRSを用いたナローイングにより与えられる．Evでは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．さらに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できる．また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．

本論文の構成は次のとおりである．2章で，Evのプログラミング例として論理回路のシミュレーションをあげる．3章で，Evの構文を基本構文とそれを拡張した糖衣構文で定義する．4章で，実装したEv処理系の各構成部分の実現方法を述べ，さらに局所定義を変換する部分については，その変換手続きを与える．5章でEvの理論的基礎について述べ，Ev処理系がこれらの理論に基づいて実装されたことを示す．最後に6章で，結論と今後の研究課題について述べる．

2. 関数論理型プログラム—Evのプログラム例

まずEvのプログラミング例として，伝搬遅延付のゲートを扱う論理回路のシミュレーションをあげる．本節は言語Evの特徴を見てとるのが目的である．言語のより詳しい記述は3章に与える．

この例では，論理回路のゲートを関数として，ゲート間の配線の状態を関数の結合としてモデル化する．配線の電位の状態を，構成子*HとL，Undefで表し，時刻tの時の配線上の電位がvであることを，構成子項*At t vで表す．回路上の配線の電位状態は，時刻によって移り変わる．時刻$t_1, \cdots, t_n$における電位を$v_1, \cdots, v_n$とすると，この電位の履歴をリスト[At t_1 v_1, …, At t_n v_n]で表す．なお，：で（中置の）リストの構成子を，[]で空リストを，$[a_1, \cdots, a_n]$でリストa_1:(a_2:(…a_n:[])…)を表す．ゲートを表す関数は，この電位の履歴を引数として取り，ゲートを通して変化した電位の履歴を返す．

ここでは，ゲートを表す関数を直接に定義するのではなく，高階関数の機能を使ってつくり出すことにする．ゲートをつくり出す関数makeGate1，makeGate2を図1に示す．ここで，def?は入力の電位が確定したかどうかを見る関数である．関数makeGate1，makeGate2は，それぞれ1入力，2入力のゲートを表す関数をつくり出す高階関数である．これらは，適用する演算（引数op1あるいはop2）と伝搬遅延時間（引数delay）を引数とし，「確定した（Undefでない）入力に対して遅延時間後に演算結果を出力するゲート」を表す関数を返す．

関数makeGatalの内部には，1入力のゲートをシミュレートする関数opl_gateが定義されている．関数opl_gateの定義は，ゲートへの電位の入力履歴が

```
 1: def? L    = True
 2: def? H    = True
 3: def? Undef = False
 4:
 5: -- The argment op1 is a unary logical function (e.g. notF).
 6: makeGate1 op1 delay = op1_gate
 7:  where
 8:   op1_gate []             = []
 9:   op1_gate ((At tx x):rest) = op1_gate rest                          :- not (def? x) |
10:   op1_gate ((At tx x):rest) = At (tx+delay) (op1 x) : (op1_gate rest) :- def? x |
11:
12: -- The argument op2 is a binary logical function (e.g. andF).
13: makeGate2 op2 delay = op2_gate (Undef,Undef)
14:  where
15:   op2_gate (a,b) [] ws = []
16:   op2_gate (a,b) vs [] = []
17:
18:   op2_gate (a,b) ((At tx x):vs) ((At ty y):ws)
19:     = op2_gate (a,b) vs ((At ty y):ws)                              :- not (def? x) |
20:
21:   op2_gate (a,b) ((At tx x):vs) ((At ty y):ws)
22:     = op2_gate (a,b) ((At tx x):vs) ws                              :- def? x, not (def? y) |
23:
24:   op2_gate (a,b) ((At tx x):vs) ((At ty y):ws)
25:     = At (tx+delay) (op2 x b) : (op2_gate (x,b) vs ((At ty y):ws))  :- def? x, def? y, tx < ty |
26:
27:   op2_gate (a,b) ((At tx x):vs) ((At ty y):ws)
28:     = At (tx+delay) (op2 x y) : (op2_gate (x,y) vs ws)              :- def? x, def? y, tx == ty |
29:
30:   op2_gate (a,b) ((At tx x):vs) ((At ty y):ws)
31:     = At (ty+delay) (op2 a y) : (op2_gate (a,y) ((At tx x):vs) ws)  :- def? x, def? y, tx > ty |
```

図1 Evによるゲートをつくり出す関数の記述
Fig. 1 The functions creating logical gates.

* 定義は3.1節を参照のこと．

なくなった時にゲートを停止させるための部分（8行目），入力の電位が確定するまで待つ部分(9行目)，出力を計算し，ゲートの出力履歴に加える部分(10行目)からなる．

関数makeGate2内部で定義されている関数op2_gateは2入力のゲートをシミュレートする．第一引数は，演算待ちの入力電位の対(2入力のゲートのため）を示す．第二，第三引数は，ゲートの二つの入力履歴である．関数op2_gateの定義もopl_gateと同様に，停止する部分(15, 16行目)，待つ部分(18～22行目)，出力を加える部分（24～31行目）からなる．

図1のプログラムから分かるように，Evのプログラムは条件付き書換え規則の集合として記述する．条件付き書換え規則 $s=t:-C$ は，条件 C が成立するとき，s から t への書換えが起こることを意味する．

この関数makeGatel, makeGate2を使って，遅れ3のNOTゲート（関数notGate)，遅れ2のANDゲート（関数andGate）を次のようにしてつくり出すことができる．

```
notF H=L
notF L=H
notGate=makeGatel notF 3

andF L L=L
andF L H=L
andF H x=x
andF L Undef=Undef
andF Undef x=Undef
andGate=makeGate2 andF 2
```

同様に，遅れ1であるORゲート（関数orGate）も定義することができる．こうしてできたゲートを組み合わせて，さらに大きな回路をつくることができる．たとえば，半加算器は次のように定義できる．

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

halfAdder a b= (c, s)
  where d=orGate a b
        c=andGate a b
        e=notGate c
        s=andGate d e
```

次に，このプログラムのもとでの質問の実行を示す．質問の実行とは，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，次のような等式からなる質問を出す．これは時刻0で半加算器の双方の入力に電位Hを与え，その出力を求める質問である．時刻9で双方の入力を電位Lにしている．

```
?−halfAdder  [At 0 H, At 9 L]
             [At 0 H, At 9 L]
   ==result.
```

すると次の解を得る．

result=([*At 2 H*, *At 11 L*],
[*At 3 Undef*, *At 7 L*, *At 12 L*])

これは時刻2で半加算器のキャリー側の出力がH，時刻7で和の側の出力がLとなったことを示している．このような質問は，Evを関数型言語的に使っているといえる．

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に，質問の中に変数を含めて，その変数が具体化される値を求めるという，いわば逆向きの計算も可能である．次の質問は，先の質問で得られた加算器の出力を与え，それを出力するような加算器への入力を求めるものである．

```
?−halfAdder  [At tx x, At ty y]
             [At 0 H, At 9 L]
   ==([At 2 L, At 11 L],
             [At 3 Undef, At 7 H, At 12 L])
```

これに対して，確かに先の入力と同じ解が求まる．

tx=*0*, *x*=*H*, *ty*=*9*, *y*=*L*;

さらに他の解も求めることができる．

tx=*0*, *x*=*H*, *ty*=*9*, *y*=*H*;
tx=*0*, *x*=*H*, *ty*=*n*+*10*, *y*=*H*;
⋮

ここで n は変数である．5.1節で述べるように，Evはデータ項代入を求めるため，このように変数を含む解が求まることもある．

3. 関数論理型言語 Ev

次に言語Evについてより詳しく述べる．Evの構文は基本構文，およ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章で述べる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**図2**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．

書換え規則 r 中の $f\ t_1 \cdots t_n$ を左辺，t を右辺，$e'_1, \cdots, e'_m$ をガード部，$e_1, \cdots, e_l$ 条件部と呼ぶ．ガード部，条

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
$r ::= f\ t_1 \ldots t_n = t :- e'_1, \ldots, e'_m \mid e_1, \ldots, e_l$
プログラム $p ::= \{ r_1. \ldots r_n. \}$

図2 Evの基本構文
Fig. 2 Basic syntax of Ev.

件部は厳格等式（定義は5.1節を参照のこと）の列である．図2で定義される項のうちで，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，書換え規則の左辺の最左にある記号（f の部分）が同一である書換え規則の集まりを（f の）関数定義と呼ぶ．

3.2 糖衣構文

次に，基本構文に対する拡張として，Evの糖衣構文を定義する．

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，字句 **where** 以後の {} で囲まれた部分がwhere節である．関数gは関数fの右辺で有効な局所関数である．関数gを定義する書換え規則の右辺に現れる変数xは，fの仮引数として現れた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．

$r ::= f\ t_1 \cdots t_n = t :- e'_1, \cdots, e'_m \mid e_1, \cdots, e_l$ **where** p

書換え規則中の仮引数および局所関数の有効域を以下で定める．まず，いくつか用語の定義をする．

{} 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 r の右辺，ガード部，条件部をまとめて，右部と呼ぶ．さらに書換え規則 r の右部と，r の持つwhere節以下にあるすべてのブロックの右部をまとめて右下部と呼ぶ．書換え規則

r: $f\ t_1 \cdots t_n = t :- e'_1, \cdots, e'_m \mid e_1, \cdots, e_l$ **where** p

に対する変数，関数記号の有効域は次のとおりである．

$t_1, \cdots, t_n$ 中の変数の有効域：
　r の右下部
関数記号 f の有効域：
　r の右下部，f と同ブロックにある他の書換え規則の右下部
$e'_1, \cdots, e'_m, e_1, \cdots, e_l$ 中の変数で $t_1, \cdots, t_n$ 中には現れていない変数の有効域：
　$e'_1, \cdots, e'_m, e_1, \cdots, e_l$
p 中で定義される関数 $f_1, \cdots, f_m$:
　r の右下部

局所定義を持つEvのプログラムは4.1.2項に示される局所定義の持ち上げ手続きにより，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，省略しても復元は可能である．Evでは，この字下げの情報をも考慮に入れた構文でプログラムを書くことができる．

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．この補完を行えば，字下げ情報付き構文の字句列（これは位置情報を持つ）は，通常の文脈自由文法に従う構文の字句列となる．

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（現在と同じモードであっても）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．

レイアウトルールを以下に示す．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の字句，あるいは字句 **where** の場合，
　その直後の字句（桁位置 s から始まるものとする）が
　字句 { の場合：
　　通常処理モードに入る．
　字句 { でない場合：
　　字句 **where** の直後に字句 { 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場合，
　t の次の字句，すなわち t の次行にある字句が，

桁位置 s から始まっている場合：

t のある行と次行は分離されている行とみなし，t の次に字句 . を挿入する．

桁位置 $s'(>s)$ から始まっている場合：

t のある行の継続行とみなすので何もしない．

桁位置 $s''(<s)$ から始まっている場合：

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

3. t が } の場合，

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 がその他の字句の場合は，何もしない．

3.2.3 その他の糖衣構文

Ev の糖衣構文では，中置記法の式を書くことができる．Ev における中置演算子の集合を Op とする．中置記法 (t_1 op t_2)，$op \in Op$ 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 が＋のとき，op' は plus に対応する．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り容易に扱うことができる．これらデータの表記法と，基本構文への展開法を**表 1** に示す．

ガード部，条件部，質問に現れる厳格等式は s＝＝True の形式ならば，単に s と略記が可能である．例えば，図1では def? x＝＝True と書くべきところを def? x と書いた．

4. Ev 処理系の実現方法

この章では，前章までに述べてきた言語 Ev の処理系の実現方法を示す．Ev 処理系は，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 3** に示す．

表 1 データの表記法と展開法
Table 1 Notations for data and its expansions.

	糖衣構文	基本構文
組	(a_1, ..., a_n)	Tuple_n a_1 ... a_n
リスト	[a_1, ..., a_n]	Cons a_1 (Cons ··· (Cons a_n Nil))
文字	'c'	Char_c
文字列	" c_1 ... c_n "	Cons Char_c_1 (Cons ··· (Cons Char_c_n Nil))

処理系は，フロントエンド，インタプリタ，ユーザインタフェースの三つの部分からなる．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．

フロントエンドは Ev のプログラムを構文解析したのち，本章で述べる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．構文処理系の実現には，字句解析生成系 Rex[5]，構文解析器生成系 Gentle[6] を用いた．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，実際の計算を行う部分である．インタプリタの実現は，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 上に実現した．

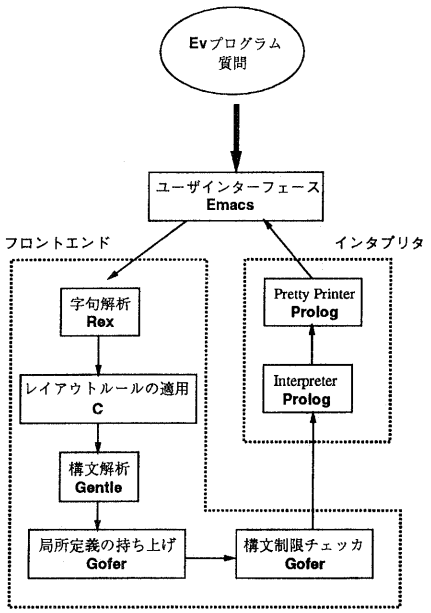


図 3 Ev 処理系の構成
Fig. 3 Structure of Ev system.

以下では，フロントエンドの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3）．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項で示したレイアウトルールを適用し，適切な位置に字句{,},.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に持ち上げる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上げる．例えばプログラム

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

は

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

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ラムダ持ち上げ（lambda lifting）として知られている技法である[7),8)]．この技法において関数型プログラムは，λ式の糖衣としてみなされる．ラムダ持ち上げは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であるため，これを関数型プログラム（の糖衣を展開したλ式）に適用すると，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，Evにおける局所定義の持ち上げに対して，ラムダ持ち上げをそのまま用いることはできない．我々はラムダ持ち上げを参考にして，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）の右部に現れる変数 x は，ブロック p をwhere節として持つ規則（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 p' より上のブロックにある書換え規則の左辺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．同様に，ある書換え規則（ブロック p にあるとする）の右部に現れる関数記号 f は，ブロック p をwhere節として持つ規則のあるブロックか，それよりも上のブロックで定義された関数記号であるならば，間接束縛関数と呼ばれる．

持ち上げ手続きは次の三つのステップから構成されている．

Step 1：名前の付け換え

大域ブロックと局所ブロックのすべての書換え規則について，変数名，関数名を有効域を考慮しつつ，衝突がないように一意に名前を付け換える．

Step 2：変数の注記

プログラム中の任意のwhere節であるブロック p に対して，p 中の書換え規則の間接束縛変数をすべて求め，これを集合 B としてブロック p に注記する．

次に再びプログラム中の任意のwhere節であるブロック p に対して以下を行う．p 中の間接束縛関数をすべて求め，これを集合 $\{g_1, \cdots, g_l\}$ とする．各関数記号 g_i が定義されたブロックに注記されている間接束縛変数の集合を B_i とする．ブロック p に注記されている変数の集合を B とすると，これに代えて $B \cup \cup B_i$ をブロック p に注記する．

Step 3：持ち上げ

大域ブロック中の任意の書換え規則が持つ，where節であるブロック p に対して以下の手続き **lift** を行う．

lift：ブロック p には変数の集合 $\{z_1, \cdots, z_k\}$ が注記されているとする．ブロック p 中の各書換え規則 $f_i\ t_1 \cdots t_n = t :- g_1, \cdots, g_m | e_1, \cdots, e_l$ 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 $(f_i\ z_1 \cdots z_n)$ に置き換える．この結果，t は t' に，g_i は g_i' に，e_i は e_i' に変換される．さらにブロック q 中の各書換え規則について再帰的に手続き **lift** を行うとすると q 中の書換え規則はすべて大域ブロックに持ち上げられる．そして $f_i\ z_1 \cdots z_k\ t_1 \cdots t_n = t' :- g_1', \cdots, g_m' | e_1', \cdots, e_l'$ 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ブロックへ持ち上げられ，すべての書換え規則はwhere節を持たないものとなる．

5. Evの理論的基礎

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[9]．計算系 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．本章では，この計算系 LNC を含むEvが基づく理論について述べる．まず一般的なCTRSについて述べ，次にEvの計算モデルとして使われる作用型CTRSについて述べる．

5.1 CTRSとナローイング

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

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

この書換え規則は，条件部$c_1,\cdots,c_m$がすべて成立するときに項$f(l_1,\cdots,l_n)$をrに書き換えることを表す．記号fは関数記号の集合$\mathcal{F}$に属する．さらに記号の集合には，変数の集合$\mathcal{V}$，構成子記号の集合$\mathcal{C}$があり，これらはそれぞれ互いに素である．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要求される様々な性質を得るために，我々が用いるCTRSは，次の四つの条件を満たすものとする．

書換え規則に対する制限：

1. すべての書換え規則$l\to r\Leftarrow C\in\mathcal{R}$において，$l$には同じ変数が二度以上現れない．
2. 任意の二つの書換え規則は重なりをもたない．
3. すべての書き換え規則$l\to r\Leftarrow C\in\mathcal{R}$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，二つの項sとtが同じ閉じた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，いくつかの関数論理型言語で採用されている[1),2)]．

ナローイングは，項書換え系を用いた，項に対する操作である．関数論理型言語の有力な計算モデルとして注目されている[1),3),4)]．以下でナローイングの概略を説明する．解を求めたい厳格等式の列をゴールと呼ぶ．まずプログラムとして，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

- [on] 最外ナローイング

$$\frac{f(s_1,\ldots,s_n)=t,E}{s_1=l_1,\ldots,s_n=l_n,C,r=t,E}\qquad t\notin\mathcal{V}$$

ただし，書換え規則$f(l_1,\ldots,l_n)\to r\Leftarrow C$が$\mathcal{R}$に含まれる時に限る．

- [d] 分解

$$\frac{f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n),E}{s_1=t_1,\ldots,s_n=t_n,E}$$

- [v] 変数消去

$$\frac{t=x,E}{\sigma E,\text{ここで }\sigma=\{x\mapsto t\}}\quad\text{または}\quad\frac{x=t,E}{\sigma E,\text{ここで }\sigma=\{x\mapsto t\}}\qquad t\notin\mathcal{V}$$

- [ons] 厳格等式のための最外ナローイング

$$\frac{f(s_1,\ldots,s_n)\equiv t,E}{s_1=l_1,\ldots,s_n=l_n,C,r\equiv t,E}\quad\text{または}\quad\frac{s\equiv f(t_1,\ldots,t_n),E}{t_1=l_1,\ldots,t_n=l_n,C,s\equiv r,E}$$

ただし，書換え規則$f(l_1,\ldots,l_n)\to r\Leftarrow C$が$\mathcal{R}$に含まれる時に限る．

- [ds] 厳格等式のための分解

$$\frac{c(s_1,\ldots,s_n)\equiv c(t_1,\ldots,t_n),E}{s_1\equiv t_1,\ldots,s_n\equiv t_n,E}$$

ただし，cは構成子記号．

- [ims] 模倣

$$\frac{c(s_1,\ldots,s_n)\equiv y,E}{\theta(s_1\equiv y_1,\ldots,s_n\equiv y_n,E)}\quad\text{または}\quad\frac{y\equiv c(t_1,\ldots,t_n),E}{\theta(y_1\equiv t_1,\ldots,y_n\equiv t_n,E)}$$

ただしcは構成子記号，$\theta=\{y\mapsto c(y_1,\ldots,y_n)\}$．

- [ts] 自明な厳格等式の除去

$$\frac{x\equiv y,E}{\sigma E}\qquad\text{ここで}\quad\sigma=\begin{cases}\{x\mapsto y\} & x\neq y\text{のとき}\\ \emptyset & \text{その他．}\end{cases}$$

図4 LNCの推論規則
Fig. 4 Inference rules of LNC.

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，最終的には，ゴールを満たすデータ項代入が求まることになる．

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，書換え可能な部分項の選択に対して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[10)~12)]．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[13)]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適用することで，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[9)]．

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[14)]．図4に計算系 LNC の推論規則を示す．この計算系は，ある与えられたCTRSの下で用いられる．推論規則

[on], [d], [v]は，遅延評価を実現するためのもの，[ons], [ims], [ds], [ts]は，効率よく厳格等式を処理するためのものである．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．計算系LNC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，計算機上に実装しやすく，実行効率のよい実現が可能である．

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ような解をすべ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[14]．すなわちLNCを用いて，与えられた等式を満たすすべてのデータ項を求めることができる．

5.3 作用型CTRS

計算系LNCで扱う5.1節であげた条件を満たすようなCTRSは，一階の関数の定義であるとみなすことができる．しかしEvは高階関数を扱える言語であるので，Evの実現のためには，CTRSのうちでも特に作用型CTRS*（applicative CTRS）と呼ばれる書換え系が必要となる．作用型CTRSは，関数記号として'Ap'および0引数のいくつかの関数記号を持つものである．項の表現に対しては，変数，構成子記号，0引数の関数記号，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，作用型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．

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

項は左結合であると仮定すれば，Apといくつかの括弧およびコンマは，省略したとしても混乱は生じない．上の例は，簡潔に

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

と書ける．そして容易にこの例は，高階関数mapを定義していることがわかる．

* 著者らは従来はコンビナトリーCTRSという用語を用いてきたが，本論文では作用型CTRSと変更した．コンビナトリーという言葉は我々の計算系の特徴を適切に表しているが，最近ではλ式にみられる関数の抽象化表現を含んだ書換え系を，コンビナトリーシステムと呼ぶことが多い．このため我々の用いるCTRSを，関数の（高階な）作用を特徴とするという意味で，作用型と呼ぶこととした．

5.4 作用型CTRSとしてのEv

Evのプログラムは，次のようにして作用型CTRSとして解釈される．Evの0引数の関数記号は，CTRSにおいても関数記号と解釈される．Evの $n(>0)$ 引数の関数記号，任意の構成子記号は，ともに作用型CTRSにおいては構成子記号と解釈される．Evにおける関数記号，構成子記号という語と，作用型CTRSにおける関数記号，構成子記号という語は，同じ用語を用いているが意味は異なることを注意しておく．

項の適用表現 $(t_1\ t_2)$ は $\mathrm{Ap}(t_1, t_2)$ と解釈する．また字句＝＝をCTRSでは厳格等号 ≡ として，Evの書換え規則における字句＝をCTRSにおける記号→として，字句：－を記号 ⇐ 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．

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を，対応するEvのプログラムに対して若干弱めた形で課す．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の計算機構として用いることができる．

6. 結　論

本論文では関数型言語Evとその処理系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基礎を持ち，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．処理系の中核とな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り容易に行うことができた．

計算系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．この変換の方法を考案し，実装を行った．さらにユーザとの対話を行うためのインタフェースを実現し，これらを統合することによってEv処理系を構築した．

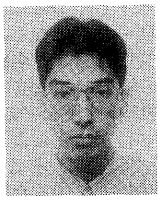
今後の研究課題としては，Evのための抽象機械の設計および，これに対してのコンパイラの開発がある．またEvの拡張として，型を導入し，型検査機構を処理系に組み込むことを検討している．

参考文献

1) Moreno-Navarro, J. J. and Rodríguez-

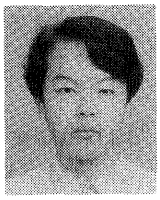
Artalejo, M.: Logic Programming with Functions and Predicates: The Language BABEL, *J. of Logic Programming*, Vol. 12, pp. 191-223 (1992).
2) Giovannetti, E., Levi, G., Moiso, C. and Palamidessi, C.: Kernel-LEAF: A Logic Plus Functional Language, *J. of Computer and System Sciences*, Vol. 42, No. 2, pp. 139-185 (1991).
3) Fribourg, L.: SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting, *Proceedings of the 2nd IEEE Symposium on Logic Programming, Boston*, pp. 172-184 (1985).
4) Hanus, M.: Compiling Logic Programs with Equality, *Proceedings of the 2nd Workshop on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science 456*, pp. 387-401 (1990).
5) Grosch, J.: Rex—A Scanner Generator, Technical Report, GMD Research Group at the University of Karlsruhe (1991).
6) Vollmer, J.: The Compiler Construction System GENTLE Revision 3.8, Technical report, GMD research Group at the University of Karlsruhe (1992).
7) Hughes, R.: Super-combinators: A New Implementation Method for Applicative Languages, *Proceedings 11th ACM Symposium on Principles of Programming Languages*, pp. 1-10 (1982).
8) Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations, *Proceedings of the 1985 Conference of Functional Programming and Computer Architecture, LNCS 201*, pp. 190-205 (1985).
9) Ida, T. and Okui, S.: Outside-In Conditional Narrowing, *IEICE Transactions on Information and Systems*, Vol. E77-D, No. 6 (1994).
10) Antoy, S., Echahed, R. and Hanus, M.: A Needed Narrowing Strategy, *Proceedings of 21st ACM Symposium on Principles of Programming Languages*, pp. 268-279, Portland (1994).
11) Bockmayr, A., Krischer, S. and Werner, A.: Narrowing Strategies for Arbitrary Canonical Rewrite Systems, Technical Report, Universität Karlsruhe (1993).
12) You, J.-H.: Enumerating Outer Narrowing Derivations for Constructor-Based Term Rewriting Systems, *J. of Symbolic Computation*, Vol. 7, pp. 319-341 (1989).
13) Huet, G. and Lévy, J.: Computations in Orthogonal Rewriting Systems, I, in Lassez, J.-L. and Plotkin, G. eds., *Computational Logic: Essays in Honor of Alan Robinson*, pp. 395-414, The MIT Press (1991).
14) Ida, T., Nakahara, K. and Hamana, M.: Leftmost Outside-In Conditional Narrowing for Functional-logic Programming Languages, Technical Report ISE-TR-94-108, University of Tsukuba (1994).

(平成6年4月27日受付)
(平成7年5月12日採録)



浜名　誠

1971年生. 1993年東京電機大学理工学部情報科学科卒業. 現在, 筑波大学博士課程工学研究科電子・情報工学専攻在学中. 関数論理型言語の実装および意味論の研究に従事.



西岡　知之（学生会員）

1966年生. 1991年筑波大学第3学群情報学類卒業. 現在, 筑波大学博士課程工学研究科電子・情報工学専攻在学中. 項書換え系, プログラミング言語, 特に関数論理型言語の設計と実装に興味を持つ. 日本ソフトウェア科学会会員.

中原　鉱一

1970年生. 1993年筑波大学第三学群情報学類卒業. 1995年同大大学院工学研究科, 修士号取得後中退. 同年(株)キヤノンに入社. 関数論理型言語, 項書換え系, 並行処理, 言語処理系に興味を持つ.

アート　ミデルドープ

1963年生. 1986年アムステルダム自由大学卒業. 1990年同大学大学院情報科学専攻博士課程修了. 同年オランダ数学・計算機科学センター(C. W. I)研究員. 1992年日立基礎研究所客員研究員. 1993年から, 筑波大学電子・情報工学系講師, 現在に至る. 理学博士. 記号計算, 特に項書換え系の分野での研究に従事. EATCS会員.

井田　哲雄（正会員）

1947年生. 1971年東京大学教養学部基礎科学卒業. 理学系研究科博士課程物理学専攻中退. 理学博士（東京大学）. 筑波大学教授. 電子・情報工学系および先端学際領域センター. 記号処理, 宣言型プログラミング, 計算モデルの研究に従事.