記 号 処 理 54－5
（1990. 3. 12）

関数型言語による論理型プログラムの記述

今泉正雄、寺島元章
電気通信大学 情報工学科

本論文は、論理型プログラミングと関数型プログラミングという２つのパラダイムの融合問題に対する一解決法を示す。そのアプローチとして、論理プログラムの関数型言語への変換を行ない、リダクションによる論理の推論を試みる。論理型言語は、pure Prolog を基本に、その言語仕様を強化したもの（以下、単に Prolog という）を用いる。

Prolog のプログラムは、推論の結果真か偽の値を返すが、ユニフィケーションの機構により、推論過程で変数値の決定が行なわれる。これらを関数のリダクションのみで行なうためには、述語を、定数、変数を含む引数リストからこれにユニフィケーションを施した後のリストを得る多価関数と考えれば良いことを示す。また、この多価関数としての値をストリームとしてとらえることにより、関数のリダクション戦略と Prolog の計算戦略との対応がとれることを示す。

Writing logic programs with a functional language

Masao Imaizumi, Motoaki Terashima
Department of Computer Science
The University of Electro-Communications

This paper represents a solution of the problem how to fuse two paradigm : functional programming and logic programming. As an approach to it, we try to replace 'reasoning' of the latter with 'reduction' of the former by source translation. A target of logic programming language is pure-Prolog with extended features (or Prolog for short).

Prolog program returns true or false by reasoning, and values of variables are determined during reasoning by unification. We show that Prolog action can be simulated using multiple value functions for Prolog predicates and reduction of them. The multiple value function accepts an argument list made of constant(s) and variable(s), and returns a unified list. We also show that there is correspondence of strategy of reduction with that of Prolog by introducing 'stream' into multiple value handling.

1 はじめに

1.1 本論文で使用する言語

本論文では論理型言語として pure-Prolog[1] を考えるが、構文については Dec10-Prolog[5] を参考にする。

関数型言語としては、既存の言語を用いずに、λ計算が念頭におけて、しかも読みやすさを考えた言語Λ を設定する。

1.2 言語Λ

言語Λにおけるプログラムは、関数定義の集合である。ここで、関数定義とは、次の構文により表される。

<関数名>⇔ λ <変数> .{λ <変数> .} <式>
<式> ::= <定数> | <変数>
| <関数名> (<式> {,<式>})
| <式><演算子><式>
| *if* <式> *then* <式> *else* <式>
| *begin* <式> {;<式>}*end*
<演算子> ::= '=' | '+' | '−' | '×' | '÷' | '&' | '|'

言語Λの扱えるデータ型は、論理型、文字（文字列）型、整数型、関数型、リスト型、S-リスト型である。

リストは、[...]という形で表され、その操作に次の関数が用意されているものとする。

$$car([a_1, a_2, \ldots, a_n]) \rightarrow a_1$$
$$cdr([a_1, a_2, \ldots, a_n]) \rightarrow [a_2, \ldots, a_n]$$
$$cons(a_1, [a_2, \ldots, a_n]) \rightarrow [a_1, a_2, \ldots, a_n]$$
$$list(a_1, a_2) \rightarrow [a_1, a_2]$$
$$listp(X) \begin{array}{ll} \rightarrow true & ;if\ X\ is\ list \\ \rightarrow false & ;otherwise \end{array}$$
$$null(X) \begin{array}{ll} \rightarrow true & ;if\ X\ is\ [\,] \\ \rightarrow false & ;otherwise \end{array}$$
$$append([a_1, \ldots, a_m], [b_1, \ldots, b_n]) \rightarrow [a_1, \ldots, a_m, b_1, \ldots, b_n]$$
$$nth(i, [a_1, \ldots, a_i, \ldots, a_n]) \rightarrow a_i$$

S- リストとは、< ... >という形で表されるリストである。その操作のために、

$$scar, scdr, scons, slist$$

が用意される。これらは、

$$car, cdr, cons, list$$

に対応するものである。

関数型データは、関数 apply の引数となるものである。

$$apply(f, X) \rightarrow f(X)$$

論理型には、演算子 '&'(論理積)、'|'(論理和) の他、否定*not*が関数として与えられているものとする。

2 述語の関数化

2.1 Prolog の述語と論理型関数

Prolog のプログラムは、論理式の集まりであるから、これらを 論理型の関数とみなすことは、ごく自然である。

例 2.1

$$f(X) : -g(X).$$
$$g(a).$$
$$\Rightarrow \quad \bar{f} \Leftrightarrow \bar{g}$$
$$\bar{g} \Leftrightarrow \lambda x.x = a$$

例 2.1に対し、 Prolog では次のようなゴール節が記述できる。

例 2.2

$$: -f(a).$$
$$: -f(X).$$

例 2.2の前者のゴール節については、論理型として変換された関数と同様の結果、*true*を得る。しかし後者の場合、$X = a$ という結果を求めながら、*true*を得るところに、プログラミング言語としての Prolog の意味がある。こうした機能は、単なる論理型関数の置き換えでは実現できない。

2.2 Prolog の述語と多価関数

2.1の問題を解決するため、述語gの関数化$\bar{g}$を次のような関数として考える。

例 2.3

$$\bar{g}([a]) \rightarrow < [a] > \quad /*\ true\ */$$
$$\bar{g}([b]) \rightarrow < > \quad /*\ false\ */$$
$$\bar{g}(*) \rightarrow < [a] > \quad /*\ true\ */$$

ここで、'*'は変数を表し、3.1節でみるように、任意の定数とパターンマッチする。

引数をリスト型とするのは2引数以上の述語の考慮である。これを引数リストと呼ぶことにする。

また、結果をSリストとするのは一般に述語の関数化は多価関数となるので、これらをひとまとめに扱うためである。

3 ユニフィケーション

3.1 ユニフィケーション関数

ユニフィケーションの動作は、推論過程を計算と見なす仕組みにおいては、本質的役割を持っている。

ここで、ユニフィケーションを行なう関数を定義する。

定義 **3.1** ε-関数

$$
\begin{array}{l}
\varepsilon \Leftrightarrow \lambda x.\lambda y. \\
\quad if\ x = *\ then\ y \\
\quad else\ if\ y = *\ then\ x \\
\quad else\ if\ x = y\ then\ x \\
\quad else\ if\ listp(x)\ \&\ listp(y)\ then\ \varepsilon'(x,y) \\
\quad else\ fail
\end{array}
$$

$$
\begin{array}{l}
\varepsilon' \Leftrightarrow \lambda Lx.\lambda Ly. \\
\quad if\ null(Lx)\ then \\
\qquad (if\ null(Ly) \mid Ly = *\ then\ [\,]\ else\ fail) \\
\quad else\ if\ null(Ly)\ then \\
\qquad (if\ Lx = *\ then\ [\,]\ else\ fail) \\
\quad else\ if\ not(listp(Lx)) \mid not(listp(Ly))\ then \\
\qquad \varepsilon(Lx, Ly) \\
\quad else\ if\ \varepsilon(car(Lx), car(Ly)) = fail\ then\ fail \\
\quad else\ if\ \varepsilon'(cdr(Lx), cdr(Ly)) = fail\ then\ fail \\
\quad else\ cons(\varepsilon(car(Lx), car(Ly)), \\
\qquad\qquad \varepsilon'(cdr(Lx), cdr(Ly)))
\end{array}
$$

例 **3.1**

$$
\begin{array}{ll}
\varepsilon([a],[a]) = [a] & \varepsilon([a,b],[a,*]) = [a,b] \\
\varepsilon([a],[b]) = fail & \varepsilon([a,b],[c,*]) = fail \\
\varepsilon([a],*) = [a] &
\end{array}
$$

3.2 変数を含まない述語の変換

ユニフィケーション関数εの値は、関数化された述語のリダクション値ではない。そこで新たに次の関数を定義する。

定義 **3.2** υ-関数

$$
\begin{array}{l}
\upsilon \Leftrightarrow \lambda L1.\lambda L2. \\
\qquad if\ \varepsilon(L1, L2) = fail\ then\ <> \\
\qquad else\ slist(\varepsilon(L1, L2))
\end{array}
$$

上述のυ-関数を用いれば、述語の引数が全て定数であるような確定節 (definite clause) は、次の規則によって変換可能である。

変換規則 **3.1**

$$
\begin{array}{l}
f(a_1, a_2, ..., a_n) \\
\Rightarrow \bar{f} \Leftrightarrow \lambda L.\upsilon(L, [a_1, a_2, ..., a_n])
\end{array}
$$

4 and/or 定義の述語の変換

4.1 or 定義の述語とその変換

定義 **4.1** σ-関数

$$
\begin{array}{l}
\sigma \Leftrightarrow \lambda S_1.\lambda S_2. \ldots .\lambda S_n. \\
if\ n = 0\ then\ <> \\
else\ sappend(S_1, \sigma(S_2, \ldots, S_n))
\end{array}
$$

$$
\begin{array}{l}
sappend \Leftrightarrow \lambda S_x.\lambda S_y. \\
if\ S_x = <>\ then\ S_y \\
else\ scons(scar(S_x), sappend(scdr(S_x), S_y))
\end{array}
$$

上述の関数を用いれば、or定義される述語に関しては、次の変換規則が成立する。

変換規則 **4.1**

$$
\begin{array}{l}
f(X) : -g_1(X). \\
f(X) : -g_2(X). \\
\qquad \vdots \\
f(X) : -g_n(X). \\
\Rightarrow \overline{f} \Leftrightarrow \lambda L.\sigma(\overline{g_1}(L), \overline{g_2}(L), ..., \overline{g_n}(L))
\end{array}
$$

定理 **4.1**

$$f = \lambda L.\sigma(g_1(L), g_2(L), \ldots, g_n(L))$$

において各 g_i が有限個の解を持つとき、$f(*)$をリダクションすればその解が求まる。

(証明)

n に関する帰納法による。□

4.2 and 定義の述語とその変換

定義 4.2 π-関数

$$\pi \Leftrightarrow \lambda f.\lambda S.$$
$$if\ S =< >\ then\ < >$$
$$else\ sappend(apply(f, scar(S)),$$
$$\pi(f, scdr(S)))$$

上述のπ-関数と、前節のσ-関数の間には、次の関係がある。

定理 4.2

$$\pi(f, < L_1, L_2, \ldots, L_m >)$$
$$= \sigma(f(L_1), f(L_2), \ldots, f(L_m))$$

(証明)
定義より明らか。□

and 定義される述語に関しては、次の変換規則が成立する。

変換規則 4.2

$$f(X) : -g_1(X), g_2(X), \ldots, g_n(X).$$
$$\Rightarrow \overline{f} \Leftrightarrow \lambda L.\pi(\overline{g_n}, \pi(\overline{g_{n-1}}, \ldots, \pi(\overline{g_2}, \overline{g_1}(L))\ldots))$$

定理 4.3

$$f = \lambda L.\pi(g_n, \pi(g_{n-1}, \ldots, \pi(g_2, g_1(L))\ldots))$$

において各 $g_i(*)$ が有限個の解を持つとき、$f(*)$ をリダクションすればその解がもとまる。

(証明)
nに関する帰納法、及び定理 4.1、定理 4.2による。□

5 リダクション戦略

定理 4.1および定理 4.3は、変換後の関数はリダクションの方法にかかわらず、解が求められることを示している。しかし、再帰的に定義される述語は、この定理の範囲外となる。

そこで、次に述べるような Prolog の計算戦略に対する工夫を行なうことで、Prolog の述語の関数化は意味を持つことになる。

5.1 Prolog の計算戦略

Prolog の述語は、その置かれた位置が、重要な意味を持つ。

例 5.1

$$f(X) : -g_1(X), g_2(X), g_3(X).$$
$$g_1(a_1).$$
$$g_1(a_2).$$
$$g_1(a_3).$$
$$g_2(a_1).$$
$$g_2(a_2).$$
$$g_3(a_2).$$

例 5.1のプログラムにたいし、

$$: -f(X).$$

なるゴール節を発行する。このとき、Prolog インタプリタは

$$g_1(X) = g_2(X) = true;\ \ g_3(X) = false$$
$$where\ X = a_1$$
$$g_1(X) = g_2(X) = g_3(X) = true$$
$$where\ X = a_2$$

という調べ方をし、a_3については調べずに停止する。ここで、$g_3(X) = false$となる時、次の候補を調べにいくのが、バックトラックである。

5.2 ストリーム計算の利用

5.1節に説明される Prolog インタプリタの動作は、計算の順序を考えていることから、対応するリダクションにも、順序を定義する必要がある。

そこで、述語の変換された多価関数としての値を、ストリーム [6][7] として取り扱うことを考える。これは関数のリダクションに時間の概念を導入することに他ならない。

例 5.1の述語fは、次のように変換される。

$$\overline{f} \Leftrightarrow \lambda L.\pi(\overline{g_3}, \pi(\overline{g_2}, \overline{g_1}(L)))$$

ここで、$\overline{g_1}(*)$の解が時間とともに

$$< [a_1] >, < [a_1], [a_2] >, < [a_1], [a_2], [a_3] >$$

と求まっていくならば、$[a_1]$を求めた時点で、

$$\pi(\overline{g_2}, < [a_1] >) \rightarrow < >$$

なるリダクションが可能となる。この場合は解がないので、さらに

$$\pi(\overline{g_2}, < [a_2] >) \rightarrow < [a_2] >$$

なるリダクションが行なわれ、その結果

$$\pi(\overline{g_3}, < [a_2] >) \rightarrow < [a_2] >$$

がリダクションされ、$\overline{f}(*)$の値として、$[a_1]$を得ることができる。

5.3 遅延評価

5.2節のストリーム計算の実現のために、言語Λに遅延評価 [8] の仕組みを設ける。すなわち、*delay*によって遅延評価体が作られ、これが *force*による評価を受けると計算が行なわれる。

変換規則 4.1にみられる各 $g_i(X)$を、*delay*によって遅延評価体とし、*sappend*、π、σ等に*force*による評価順序の規定を行なう。*scons*は、半遅延 cons[7] とする。これは car 部のみ評価し、cdr 部が遅延評価体となるものである。

これにより、関数の値は一度に一つだけ計算され、これを car 部とし、cdr 部は残りの（未評価の）遅延評価体とするようなリストを作ることが可能となる。

例 **5.2**
$f(*) =< a_1, a_2, ..., a_n >$のとき、
$\delta_0(*) = delay(f(*))$ を遅延評価体とすると、

$$force(\delta_0(*)) \rightarrow < a_1.\delta_1(*) >$$
$$force(\delta_i(*)) \rightarrow < a_{i+1}.\delta_{i+1}(*) >$$

5.4 カットの実現

Prolog プログラムにおけるカットオペレータ '!' は、プログラマがプログラムの制御に関する操作を意識的に行なうという意味で手続き的である。しかし遅延評価の応用として、このカットの実現を考えることができる。

変換規則 **5.1**

$$f(X) : -g_1(X), \ldots, g_i(X), !, g_{i+1}(X), \ldots, g_n(X).$$
$$\Rightarrow \overline{f} \Leftrightarrow \lambda L.\pi(\overline{g_n}, \ldots, \pi(\overline{g_{i+1}}, \pi(cut, \pi(\overline{g_i}, \ldots, \pi(\overline{g_2}, \overline{g_1}(L)) \ldots))) \ldots)$$

ここで、*cut*はカットオペレータに対する処理をする関数であり、次のように定義される。

定義 **5.1** *cut*

$$cut \Leftrightarrow \lambda S.$$
$$if\ S =< >\ then\ < >$$
$$else\ slist(!, scar(S))$$

すなわち、*cut*関数は、これまでに計算された値1つのみを残し、遅延評価体を全て切り捨てる。また、記号! をその解に挿入する。これは、or 定義される述語の場合、その関数化の中で、*cut*があっても、それによって捨てられない遅延評価体ができるため、関数σに対し、それらを切り捨てるように指示するためである。

6 2 引数以上の述語の変換

これまでに扱ってきた述語は、同一の引数リストをもつ述語のみで構成されるため、次の例のような述語の変換はできない。

例 **6.1**

$$f(X, Y) : -g(Y, X).$$

例 **6.2**

$$f(X, Y) : -g(X, Z), h(Y, Z).$$

6.1 パターン変換関数の導入

例 6.1の関数変換を考える。$\overline{f}$に与えられる引数リストは、$[X, Y]$なるパターンであり、また$\overline{g}$に与えられる引数リストは、$[Y, X]$なるパターンである。

そこで、次のような引数リストのパターン変換関数ωを導入する。

例 **6.3**

$$\omega([2,1],[X,Y]) \rightarrow [Y,X]$$
$$\omega([2,0],[X,Y]) \rightarrow [Y,*]$$
$$\omega([2,[3]],[X,Y]) \rightarrow [Y,3]$$

ω- 変換のための関数を次に定義する。

定義 **6.1** ω- 関数

$$
\begin{array}{l}
\omega \Leftrightarrow \lambda N.\lambda L. \\
if\ null(L)\ then\ [\,] \\
else\ cons(\\
\quad if\ car(N) = 0\ then\ * \\
\quad else\ if\ listp(car(N)) \\
\quad\quad then\ car(car(N)) \\
\quad\quad else\ nth(L, car(N)), \\
\quad \omega(cdr(N), L) \quad)
\end{array}
$$

次に定義する関数Ωは、第2引数として関数の値（すなわち、$< \ldots >$の形のもの）をとるものである。

定義 **6.2** Ω- 関数

$$
\begin{array}{l}
\Omega \Leftrightarrow \lambda N.\lambda S. \\
if scdr(S) =< > \ then\ slist(\omega(N, scar(S))) \\
else\ scons(\omega(N, scar(S)), \Omega(N, cdr(S)))
\end{array}
$$

ここで、例6.1は、次のように変換できる。

例 **6.4**

$$
\begin{array}{l}
f(X, Y) : -g(Y, Z). \\
\Rightarrow \overline{f} \Leftrightarrow \lambda L.\Omega([2,1], \overline{g}(\omega([2,1], L)))
\end{array}
$$

6.2 π- 関数の拡張

6.2のような一般の2引数以上の述語を関数変換するためには、定義4.2のπ- 関数にω変換を導入する必要がある。

定義 **6.3** 拡張π- 関数

$$
\begin{array}{l}
\pi \Leftrightarrow \lambda f.\lambda F.\lambda S. \\
if\ S =< > \ then\ < > \\
else\ sappend(\\
\quad sinsert(apply(f, \omega(F, scar(S))), \\
\quad\quad \pi(f, F, scdr(S))))
\end{array}
$$

$$
\begin{array}{l}
\pi' \Leftrightarrow \lambda f.\lambda F.\lambda L. \\
sinsert(apply(f, \omega(F, L)), L)
\end{array}
$$

$$
\begin{array}{l}
sinsert \Leftrightarrow \lambda S.\lambda L. \\
if\ S =< > \ then\ < > \\
else\ scons(append(scar(S), L), \\
\quad sinsert(scdr(S), L))
\end{array}
$$

上のπ- 関数及びπ'- 関数を用いれば、一般の2引数以上の述語の変換を行なうことができる。

変換規則 **6.1**

$$
\begin{array}{l}
f(L_0) : -g_1(L_1), g_2(L_2), \ldots, g_n(L_n). \\
\Rightarrow \overline{f} \Leftrightarrow \lambda L.\Omega(F_{n+1}, S_n)
\end{array}
$$

$$
S_k = \begin{array}{ll} \pi'(\overline{g_1}, F_1, L_0) & ; k = 1 \\ \pi(\overline{g_k}, F_k, S_{k-1}) & ; k \geq 2 \end{array}
$$

$F_i (i = 1, 2, \ldots, n)$ 決定アルゴリズム

1. $L := append(L_{i-1}, L_{i-2}, \ldots, L_0)$
2. $(L_i) = (X_1, X_2, \ldots, X_m)$ とする（ただし、$L_{n+1} = L_0$）。
   このとき、$F_i = [N_1, N_2, \ldots, N_m]$ とは、

$$
\begin{array}{ll}
N_j = 0 & ; X_j \notin L \\
N_j = 1, 2, \ldots & ; X_j \text{が} L \text{の} N_j \text{番目} \\
N_j = [X_j] & ; X_j \text{が定数}
\end{array}
$$

6.3 引数同値の明示化

Prolog 述語の、右節のない確定節の多くはυ- 関数を用いて変換できる。

例 **6.5**

$$f(X, X).$$

これは、引数として与えられたものが、同値であることを表している。こうした述語を関数のリダクションとして処理するために、組み込み述語としてeq を定義しておく。

組み込み述語とその変換 **6.1** eq

$$
\begin{array}{l}
eq(X, Y) \\
\Rightarrow \overline{eq} \Leftrightarrow \lambda L. \\
if\ \varepsilon(car(L), car(cdr(L))) = fail\ then\ < > \\
else\ slist(\varepsilon(car(L), car(cdr(L))), \\
\quad\quad \varepsilon(car(L), car(cdr(L))))
\end{array}
$$

例 **6.6**

$$
\begin{array}{ll}
\overline{eq}([a, a]) & \rightarrow < [a, a] > \\
\overline{eq}([a, *]) & \rightarrow < [a, a] >
\end{array}
$$

この eq を用いて例6.5は、関数への変換に先立ち次のように変換される。

例 6.7

$$f(X,X). \Rightarrow f(X,Y) : -eq(X,Y).$$

一般に次の変換前処理が関数への変換前に行なわれるものとする。

変換前処理規則 6.1

$$f(\ldots,X,\ldots,X,\ldots).$$
$$\Rightarrow f(\ldots,X_1,\ldots,X_2,\ldots) : -eq(X_1,X_2).$$

ここで、変数を含まない述語に対しても、eqを用いた述語に変換することを考える。

変換前処理規則 6.2

$$f(a_1,a_2,\ldots,a_n).$$
$$\Rightarrow f(X_1,X_2,\ldots,X_n)$$
$$: -eq(X_1,a_1),eq(X_2,a_2),\ldots,eq(X_n,a_n).$$

7 リストデータの取り扱い

7.1 cons の明示化

Prologでは、リストを$[X_1|X_2]$のように表す。本論文では、これを述語として明示化する。

$cons$ は、次のように関数化される。

組み込み述語とその変換 7.1 $cons$

$$cons(X,Y,Z)$$
$$\Rightarrow$$
$$\overline{cons} \Leftrightarrow \lambda L.$$
$$begin$$
$$c := \varepsilon'(cons(car(L),(car(cdr(L))),$$
$$car(cdr(cdr(L))));$$
$$if\ c = fail\ then\ <>$$
$$else\ slist(list(car(c),cdr(c),c))$$
$$end$$

例 7.1

$$\overline{cons}([a,[b],*]) \quad \rightarrow < [a,[b],[a,b]] >$$
$$\overline{cons}([*,[b],[a,b]]) \quad \rightarrow < [a,[b],[a,b]] >$$
$$\overline{cons}([a,*,[a,b]]) \quad \rightarrow < [a,[b],[a,b]] >$$

変換前処理規則 7.1 リストが左節にあるとき

$$f(\ldots,[X_1|X_2],\ldots) : -\ldots.$$
$$\Rightarrow f(\ldots,X,\ldots) : -cons(X_1,X_2,X),$$
$$\ldots,cons(X_1,X_2,X).$$

変換前処理規則 7.2 リストが右節にあるとき

$$f(\ldots,[X_1|X_2],\ldots)$$
$$\Rightarrow cons(X_1,X_2,X),f(\ldots,X,\ldots),$$
$$cons(X_1,X_2,X)$$

例 7.2

$$append([\,],Y,Y).$$
$$append([A|B],Y,[A|C]) : -append(B,Y,C).$$

$$\Rightarrow \quad append(X,Y,Z) : -eq(X,[\,]),eq(Y,Z).$$
$$append(X,Y,Z) : -cons(A,B,X),$$
$$cons(A,C,Z),append(B,Y,C),$$
$$cons(A,B,X),cons(A,C,Z).$$

8 整数データの取り扱い

整数データ型はプログラミング言語としては必要な型であるが、論理の枠内でこれを取り扱うことは効率が悪い。

そこで、一般には、整数の演算は述語の取り扱いとは別に行なうが、本論文では既に述べている変換規則を適用できるよう、形式上は述語として扱う工夫をする。

8.1 加減乗除

変換前処理規則 8.1

$$X\ is\ a_1 + a_2 \Rightarrow add(a_1,a_2,X)$$
$$X\ is\ a_1 - a_2 \Rightarrow add(a_2,X,a_1)$$
$$X\ is\ a_1 * a_2 \Rightarrow mul(a_1,a_2,X)$$
$$X\ is\ a_1/a_2 \Rightarrow mul(a_2,X,a_1)$$

例 8.1

$$fact(0,1).$$
$$fact(X,Y) : -Z\ is\ X-1,fact(Z,U),$$
$$Y\ is\ X*U.$$
$$\Rightarrow \quad fact(X,Y) : -eq(X,0),eq(Y,1).$$
$$fact(X,Y) : -add(Z,1,X),$$
$$fact(Z,U),mul(X,U,Y).$$

8.2 比較

すでに6.2節において等価を表す述語eqとその関数化$\overline{eq}$について述べたが、これはPrologのプログラム中、比較演算子'='が現れる箇所にも用いる。

変換前処理規則 8.2

$$a_1 = a_2 \Rightarrow eq(a_1, a_2)$$

同様に、次のような比較演算子

$$>,\ <,\ \geq,\ \leq,\ \neq$$

についても、

$$gt,\ lt,\ ge,\ le,\ ne$$

などの述語を考え、上記のような変換規則を与える。

9 結論

本論文では関数型言語の処理系における論理型プログラミングの可能性と、論理型プログラミングと関数型プログラミングの融合を示した。すなわち

- Prolog の述語は、多価関数とみなすことができ、論理プログラミングにおける推論及び計算の機構を純粋に関数のリダクションに置き換えられる。
- 多価関数としての値を求めていく過程にストリーム計算を応用することで、Prolog の計算戦略に対応するリダクションを考えることができる。

また、この結果、

- Prolog において計算（推論）可能な述語は、これを関数化したものでも計算（リダクション）可能である。

本論文の応用としては、

- Lisp マシン上でのリレーショナルデータベースの構築
- 論理型の記述容易性を活かした関数型プログラムのプロトタイピング

等が考えられる。

参考文献

[1] Kowalski,R.A. *Predicate Logic as Programing Language, Proc. IFIP-74 Congr., North-Holland, Amsterdam (1974)*

[2] Steele,G.L.Jr and Sussman,G.J. *The Revised Report on SCHEME A dialect of LISP. AI Memo 452, MIT Artificial Intelligence Lab. (1978)*

[3] Church,A. *The calculi of lambda conversion. Princeton University Press (1941)*

[4] Barendregt,H.P. *The Lambda Calculus, North-Holland (1984)*

[5] Clocksin,WF. and Mellish,C.S. *Programming in Prolog Springer-Verlaag Berlin Heidelbeerg New York (1981)*

[6] Landin,P.J. *The Next 700 Programming Languages, Comm. ACM, Vol.9, No.3 (1966)*

[7] Friedman,D.P. and Wise,D.S. *CONS should not evaluate its arguments, automata, languages and programming, Edinburgh University Press (1976)*

[8] Henderson,P. and Morris,J.H. *A Lazy Evaluator,Conference record of the third ACM symposium on principles of programming languages, Atlanta, Georgia (1976)*