等式システムとそのインタプリタ

長田博泰
（北大情報処理教育センター）

1. はじめに

等式は計算の過程を表現するための単純かつ便利な記法である。また、その意味も簡単かつ明瞭である。本稿では等式が有用なことを例をもって示すとともに、等式を直接実行するインタプリタを作成するときにであうパターンマッチングの問題を扱う。特に，上昇型マッチングのスペースと時間を改良する方法を検討する。

広くプログラミング言語として用いられている手続き的言語は多くの問題点を含んでいる。例えば、文の実行順序、副作用，記述能力の低いこと等を指摘することができる。特に、記述能力の低いこと、すなわち、プログラムと解こうとしている問題との間の関係を理解するのが非常に困難なことである。手続き的言語のこれらの問題点を克服するものとして非手続き的言語―関数型言語、論理型言語―が活発に研究されている[1,2他]。

非手続き的言語は，関数や関係式による数学的表現や述語論理のように計算の内容にまで立ち入らずに問題を数学的対象、関数および関係で記述しようとする。したがって，非手続き的言語は宣言的色合が強く，問題の見通しが得やすく，手続き的言語のもつ問題点を克服する可能性をもつように思われる。

ここでは、非手続き的言語の一つとして等式による言語、いわゆる等式言語を扱う。等式を採用する主な理由は2つある。第1に、'等しい'という概念は、最も基本的で、日常慣れ親んでいるものであり，等式だけを用いてわかり易く、また、その意味も明確なプログラムを書くことができる。第2に、計算の過程を左辺を右辺によって置き換えてゆくという単純な操作で実行することができる。

本稿では、まず、等式言語の有用性を例示する。つぎに、インタプリタを作成するために必要となるパターンマッチングの方法とその改良について述べる。

2では、3.以降の準備のため等式言語を形式的に定義する。3.では、等式を項書き換えシステムとして実行し，解を得るための条件等について述べる。4.では、等式言語による二三のプログラム例を掲げ，他の関数型あるいは論理型言語と比較しながら示し，その表現力が実用に充分耐え得るものであり、かつ、等式言語の方が理解し易いものであることを示す。5.では，パターンマッチングの方法について扱い，特に，上昇型マッチングのスペースを大中に減らし、前処理時間を減らす方法を提案する。6.では、試作したインタプリタの構成とその性能について述べる。

2. 等式言語

等式言語は，直観的に云えば，

項1＝項2

の形の式だけからなる言語である。左辺、右辺の項は通常の数式と考えてよい。例えば，1+2=3や階乗関数$f(n)=n!$の再帰的定義である

f(x)=if(zero(x),1,x×f(x-1))

はいずれも等式言語で許される式である。ここで，if(A,B,C)は条件式ψA then B else Cを表わし，zero(x)はxが0のときのみ真となる論理式である。

以下，等式言語を形式的に定義する。本来なら、多ソート代数と呼ばれるも

のを基礎に定義すべきであるが、ここでは、簡単化した定義を行う。等式言語の構文からはじめる。

関数記号の集合Σが与えられているとし，Σ-項を以下のように定義する。

[定義1]

(1) ランク0の記号$b\in\Sigma$はΣ-項である。

(2) $a\in\Sigma$がランクgで，$t_1,\cdots,t_g$がΣ-項なら$a(t_1,\cdots,t_g)$はΣ-項である。

(3) 以上のものだけがΣ-項である。□

[定義2] νをランク0の記号とする。$\nu\notin\Sigma$。$\Sigma\cup\{\nu\}$-項のνを変数という。□

[定義3] 等式とは二つの項を等号'='で結んだものである。□

この定義からわかるように、等式言語は1階論理の構文から等号以外の論理記号($\wedge,\vee,\neg,\forall,\exists$など)および関係(または述語)記号をとり除いたものになっている。

つぎに，等式言語における推論を形式化しよう。よく知られているように等式の世界では、反射律、対称律、推移律，代入および置換に基づいて推論が行われる。

[定義4] Γを等式の集合とする。

このとき，等式の集合$\mathcal{E}(\Gamma)$を次の条件を満す最小の等式集合Δと定める．

(1) $\Gamma\subseteq\Delta$

(2) $s=t\in\Delta$ (反射律)

(3) $s=t\in\Delta$ ならば $t=s\in\Delta$ (対称律)

(4) $s=t\in\Delta$ かつ $t=u\in\Delta$ ならば $s=u$ (推移律)

(5) $s=t\in\Delta$ ならば $s[u/x]=t[u/x]\in\Delta$ (代入)

(6) $s=t\in\Delta$ ならば $u[s/x]=u[t/x]\in\Delta$ (置換)

ここで，$t[u/x]$は項tの中に出現する変数xを同時に項uで置き換えて得られる項を表す。□

等式ψが$\mathcal{E}(\Gamma)$に属するとき，ψはΓから導出されるといい$\Gamma\vdash\psi$と書く。

例：Γを以下の三つの等式とする。

$\mathrm{Pred}(\mathrm{Succ}(x))=x$,

$\mathrm{Cond}(\mathrm{true},x,y)=x$,

$\mathrm{Cond}(\mathrm{false},x,y)=y$

このとき，たとえば

$\Gamma\vdash\mathrm{Cond}(\mathrm{true},\mathrm{Pred}(\mathrm{Succ}(\mathrm{zero})),\mathrm{Succ}(\mathrm{zero}))=\mathrm{zero}$

である。

最後に，等式論理の意味論についてふれる。等式の解釈を，等式の両辺の評価値が互いに等しいときかつそのときに限ると定める。Γのいかなる解釈の上でもψが成り立つとき$\Gamma\models\psi$と書く。等式に関する最も重要な定義は次の結果である。

[定理] 等式の集合Γと等式ψに対して$\Gamma\vdash\psi$であれば，$\Gamma\models\psi$である。(無矛盾性)，逆に，$\Gamma\models\psi$であれば、$\Gamma\vdash\psi$である。(完全性)。すなわち、$\Gamma\vdash\psi$と$\Gamma\models\psi$は互いに必要十分な条件である[1]。

3. 等式による計算
—項書き換えシステム—

等式の世界では左辺から右辺を得ることも、逆に右辺から左辺を得ることも同じように考えることができる。他方、計算の世界では、例えば，1+2の結果として3が得られるのであって、3の結果として1+2が得られることはない。このように，計算の世界では、等式を複雑な式から単純な式へ変換する書き換え規則とみなした方がよい．等式を以上のような書き換え規則とみなす考え方から発展してきた計算モデルが項書き換えシステムである。

ここでは、本稿に関わりがある範囲で項書き換えシステムの基本的性質を述べる。まず，基本的用語について定義する．

[定義1] 項Mの中で書き換え規則が適用可能な部分をリデックスと呼び、項Mのリデックスを書き換えて項Nが

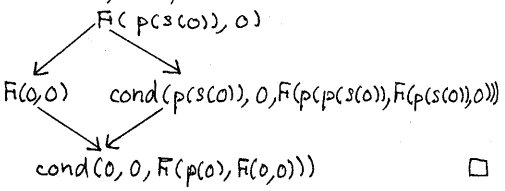
得られたなら、MはNにリダクションされたといい、M→Nと書く。Mから0回以上のリダクションでNに到達することができるときも、MはNにリダクションされたといい、M*→Nと書く。項Nがリデックスをもたないなら正規形と呼ぶ。□

項書き換えシステムによる計算で最も重要な性質の一つはリダクションの合流性である。これは、与えられた項にどのような順序で書き換え規則を適用してもよいことを示す。

例：等式の集合

$p(s(x))=x$, $cond(0,z,y)=x$,
$cond(s(x),y,z)=z$,
$F(x,y)=cond(x,0,F(p(x),F(x,y)))$

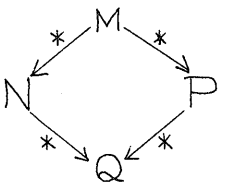
において、$F(p(s(s(0))),0)$は以下のように2つのリダクションは同じ結果をもたらす。すなわち、合流する。



□

一般の項書き換えシステムでは、このようなリダクションの合流は必ずしも成立しない。むしろ、リダクションが異なれば結果も異なることが多い。リダクションの合流性を保証する条件、すなわち、Church-Rosserの性質を見出す必要がある。

[定義2] 項書き換えシステムがChurch-Rosserの性質をみたすとは、任意の項M, N, Pにおいて、M*→N, M*→Pならば適当な項Qが存在してN*→Q, P*→Qとなることである。



□

[定義3]

項Mに2つ以上の書き換え規則が適用でき、それらが共通部分をもつなら項書き換えシステムは重なりをもつという。また、項Mが線形であるとは、Mの中に同じ変数が2回以上出現しないことである。□

Church-Rosserの性質が成立するための次の十分条件が得られている[1]。

[定理]

項書き換えシステム$\mathcal{R}$が重なりを持たず、かつ線形ならば、$\mathcal{R}$はChurch-Rosserの性質を満たす。□

項Mは一般に複数のリデックスをもつから書き換えられるリデックスが異なればその結果得られるリダクションも異なる。このとき、項が正規形をもつなら必ず正規形が得られ、なおかつ最小の書き換え回数で正規形が得られることが望ましい。重なりのない線形項書き換えシステムの場合、この問題に対する答がわかっている[1]。

[最適戦略]

外側のリデックスを常に書き換える。ただし、書き換えの際に部分項をコピーしないでポインタを用いて共有する。

4. 等式言語の例

等式言語の定義とこれを書き換え規則とみなし計算する方法を与えたので、ここでは、等式によるプログラム例を二三示す。これらの実例から、等式言語が直観的にきわめて理解し易いことがわかるであろう。

例1：記号微分

記号微分は、等式言語を用いれば、微分公式をほとんどそのまま書くことができる。比較のため、LISP, PROLOGによる記号微分も掲げる。等式によるプログラムは、他の言語に比べ、何も予備知識なしに理解できることがわかる。

[等式言語]

```
AXIOM
 FUNCTIONS
  IF:3;
  =,+,-,*,/,**,==:2;
  D,SIN,COS:1;
 CONSTANTS X;
 FOR ALL U,V,N:
  D(U)=IF(U=X,1,0) WHERE ATOM(U);
  D(U+V)=D(U)+D(V);
  D(U-V)=D(U)-D(V);
  D(U*V)=D(U)*V+D(V)*U;
  D(SIN(U))=D(U)*COS(U);
  D(X**N)=N*X**(N-1) WHERE N IN INTEGER;
END
```

これから以下のような結果が得られる。

```
D(X*X)?
=D(*(X,X))
=+(*(D(X),X),*(D(X),X))
=+(*(1,X),*(D(X),X))
=+(*(1,X),*(1,X))
=1*X+1*X

D(SIN(X+2))?
=D(SIN(+(X,2)))
=*(D(+(X,2)),COS(+(X,2)))
=*(+(D(X),D(2)),COS(+(X,2)))
=*(+(1,D(2)),COS(+(X,2)))
=*(1,COS(+(X,2)))
=1*COS((X+2))
```

例2：多項式の計算
右の等式は、多項式の加算，乗算等を行う。
P1，P2はそれぞれ、多項式
$2x^2+3x+1$，
$4x+5+4x$
を表わしている。

[PROLOG]

```
d(X,X,1) :- !.
d(C,X,0) :- atomic(C).
d(-U,X,-A) :- d(U,X,A).
d(U+V,X,A+B) :- d(U,X,A), d(V,X,B).
d(U-V,X,A-B) :- d(U,X,A), d(V,X,B).
d(C*U,X,C*A) :- atomic(C), C \= X, d(U,X,A), !.
d(U*V,X,B*U+A*V) :- d(U,X,A), d(V,X,B).
d(U/V,X,A) :- d(U*V^(-1),X,A).
d(U^V,X,V*W*U^(V-1)) :- atomic(V), C \= X, d(U,X,W).
d(log(U),X,A*U^(-1)) :- d(U,X,A).
```

```
?- d(x+1,x,X).

X = 1+0

?- d(x*x-2,x,X).

X = x*1+1*x-0
```

[LISP]

```
DEFINE((
(CADDR(LAMBDA (L)
     (CAR (CDR (CDR L))) ))
(DERIV(LAMBDA (E X)
    (COND ((ATOM E) (COND ((EQ E X) 1) (T 0)))
          ((OR (EQ (CAR E) (QUOTE PLUS)) (EQ (CAR E) (QUOTE DIFER)))
           (LIST (CAR E) (DERIV (CADR E ) X) (DERIV (CADDR E) X)))
          ((EQ (CAR E) (QUOTE TIMES))
           (LIST (QUOTE PLUS)
             (LIST (CAR E) (CADDR E) (DERIV (CADR E) X))
             (LIST (CAR E) (CADR E) (DERIV (CADDR E) X))))
          ((EQ (CAR E) (QUOTE QUOTIENT))
             (LIST (CAR E)
               (LIST (QUOTE DIFER)
                 (LIST (QUOTE TIMES) (CADDR E) (DERIV (CADR E) X))
                 (LIST (QUOTE TIMES) (CADR E) (DERIV (CADDR E) X)))
               (LIST (QUOTE TIMES) (CADDR E) (CADDR E))))
          ((EQ (CAR E) (QUOTE EXPT))
           (LIST (QUOTE TIMES)
             (LIST (QUOTE TIMES) (CADDR E)
               (COND ((EQUAL (CADDR E) 2) (CADR E))
                     (T (LIST (CAR E) (CADR E) (SUB1 (CADDR E))))))
             (DERIV (CADR E) X)))
          (T NIL)))) ))
```

```
AXIOM
 FUNCTIONS
  P1,P2,R,DEG:0;
  REDUCTUM,ISZERO,DEGREE,LDCF:1;
  REMTERM,ADD,MULT,MULTTERM,COEF,=,+,-,*,<,>,#:2;
  IF:3;
FOR ALL P,Q,A,B,C,D,X,Y:
  P1=[[2,2],[3,1],[1,0]];
  P2=[[4,1],[5,0],[4,1]];
  R=MULT(P1,P2);
  DEG=DEGREE(R);
  REMTERM([A,C]#P,D)
  =IF(C=D,REMTERM(P,D),[A,C]#REMTERM(P,D));
  REMTERM([],D)=[];
  MULTTERM([A,C],[B,D]#P)=[A*B,C+D]#MULTTERM([A,C],P);
  MULTTERM([A,C],[])=[];
  ADD([B,D]#P,Q)=[B,D]#ADD(P,Q);
  ADD([],P)=P;
  ADD(P,[])=P;
  MULT([B,D]#P,Q)=ADD(MULTTERM([B,D],Q),MULT(P,Q));
  MULT(P,[])=[];
  MULT([],P)=[];
  REDUCTUM(P)=REMTERM(P,DEGREE(P));
  ISZERO([A,C]#P)=IF(COEF(P,C)=-A,ISZERO(REMTERM(P,C)),F);
  ISZERO([])=T;
  COEF([A,C]#P,D)=IF(C=D,A+COEF(P,D),COEF(P,D));
  COEF([],C)=0;
  DEGREE([A,C]#P)
  =IF(C>DEGREE(P),C,IF(C<DEGREE(P),DEGREE(P),
   IF(COEF(P,C)=-A,DEGREE(REDUCTUM(P)),DEGREE(P))));
  DEGREE([])=0;
  LDCF(P)=COEF(P,DEGREE(P));
END
```

5. パターンマッチングの方法とその改良

3節で述べた最適戦略に従い，パターンマッチングによってリダクションを行うシステムの実現方法について述べる。一回のリダクションを行うために処理しなければならない問題は

(1) 式の効率的表現方法
(2) リデックスの見つけ方
(3) リダクションするリデックスの選択
(4) リダクションの実行

である。このうち，特に，リデックスを効率よく見つけることがシステムの高速化に最も基本的なので，その方法について説明し，その改良法を述べる。

5.1 パターンマッチングの方法

リデックスを見つける問題は木のパターンマッチングである。この問題に関してHoffmanら[5]は構文解析と同様に上昇型と下降型マッチングの方法を提案し，両方法の比較を論じている。ここでは上昇型の方法をとりあげ，これを改良する方法を検討する。

上昇型マッチングは，LR(k)文法等の上昇型構文解析のように与えられたパターン(構文)を事前に解析し，その結果得られる情報を用いてマッチングを行う。まず，その概略を例によって説明し，その後，厳密に述べる。

次の2つのパターン

$p_1 = a(a(y, y), b)$, $p_2 = a(b, y)$

が与えられたとき，各パターンの部分木をつくる。ただし，ここでyは変数を表す。

$\{y, b, a(y, y), a(b, y), a(a(y, y), b)\}$

これらの部分木の間のマッチング関係を調べ，この関係を有向グラフで表現する。上の5個の部分木のマッチング関係を有向グラフで表現すると図1のようになる。このグラフの各パスがマッチング関係を示している。例えば，

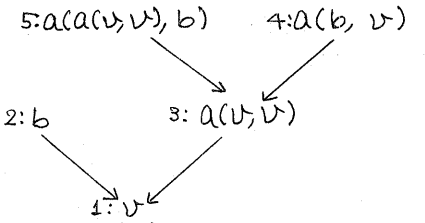


図1. 直接包含グラフ

$a(b, y) \rightarrow a(y, y) \rightarrow y$

は$a(b, y)$が$a(y, y)$, yにマッチし，$a(y, y)$がyにマッチすることを表わしている。また，この関係を

$a(b, y) > a(y, y) > y$

と表現し，簡単のために$a(b, y)$は$a(y, y)$より大きいということにする。

上昇型マッチングは，マッチングの対象となる木を下からトラバースし，図1に示されたマッチング関係を用いてマッチし得る部分木の中で一番大きいものを各ノードに割り付けてゆく．$a(a(a(b, c), b), c)$に適用してみよう．

$a(b, c)$は$a(b, y)$にマッチするのでaのノードに$a(b, y)$にマッチしたことを示す。ここでは，それを表すのに数4を割り付けている。次に，$a(a(b, y), b)$はマッチする部分木がないので，$a(b, y)$にマッチするもので一番大きい$a(y, y)$を考えると$a(a(y, y), b)$がマッチすることがわかるので，これを示す5を割り付ける。同様に，$a(5, 1)$も$a(1, 1)$すなわち3を割り付ける。この結果，図2が得られ，○で囲んだ位置でそれぞれパターンp_1, p_2とマッチすることがわかる。

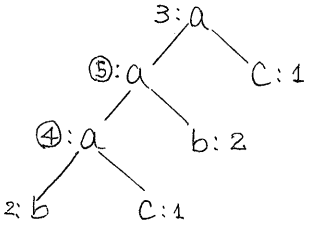


図2. 上昇型マッチング

以上の方法を定式化し、そのアルゴリズムを述べるために、いくつかの定義を与える。

[定義1] p, p'をパターンとする。

(1) pとp'が独立であるとは次の条件をみたす木t_1, t_2, t_3がある場合をいい、$p \sim p'$と書く。

a) t_1はpとマッチするがp'とマッチしない。

b) t_2はp'とマッチするがpとマッチしない。

c) t_3はp, p'とマッチする。

(2) pがp'を包含するとはpがある木にマッチするならp'もマッチすることをいう。 □

例：

(1) $a(b,v) \sim a(v,c)$

$t_1 = a(b,b), t_2 = a(c,c), t_3 = a(b,c)$

とすれば独立の条件をみたす。

(2) $a(b,v) > a(v,v)$。明らか。

[定義2]

パターンの部分木の集合が単純であるとは独立な部分木をもたないことである。 □

[定義3]

PFをパターンの部分木の集合とし、$p, p' \in$ PFのとき、pがp'を直接包含するとは $p > p'$で、$p > p'', p'' > p'$ であるような $p'' \in$ PFが存在しないことである。これを $p \succ_i p'$と書く。 □

例：パターン$a(a(v,v),b), a(b,v)$は独立な部分木をもたないから単純である。直接包含関係は以下の4つである。

$b \succ_i v$　　$a(v,v) \succ_i v$

$a(b,v) \succ_i a(v,v)$　　$a(a(v,v),b) \succ_i a(v,v)$

[定義4]

パターンの集合Fの直接包含グラフG_sは、Fの部分木をノードとし、$p \succ_i p'$ならpからp'への有向矢をもつグラフである。

例：上の例の関係から、図1に示す直接包含グラフが得られる。

直接包含グラフG_sを用いて、上昇型マッチングに必要な情報を生成するHoffmanのアルゴリズムを述べ、その後、改良方法を検討する。

[アルゴリズム－テーブル情報の作成]

入力：PFの直接包含グラフG_s

出力：上昇型マッチングを駆動するテーブルT_a：$T_a[p_1 \cdots p_m]$の各エントリには$a(p_1 \cdots p_m)$にマッチし得る最大の部分木$\in$PFが入る。

方法：

1. G_sをトポロジカルソートし、PFの木を包含関係の上昇順に並べる。
2. T_aのすべてのエントリをΛにする。
3. 上昇順に各パターン$p = a(p_1 \cdots p_m)$について4を実行する。
4. $p'_j \geq p_j$ $(1 \leq j \leq m)$であるような$\langle p'_1 \cdots p'_m \rangle$に対して $T_a[p'_1 \cdots p'_m] := p$

このアルゴリズムを図1のグラフに適用すると表1が得られる。図2をトラバースするとき、各ノードにマッチする木を容易に割り付けてゆくことができる。

表1. 図1のグラフから得られるテーブル

	v	b	$a(v,v)$	$a(b,v)$	$a(a(v,v),b)$
v					
b	$a(b,v)$	$a(b,v)$	$a(b,v)$	$a(b,v)$	$a(b,v)$
$a(v,v)$			$a(a(v,v),b)$		
$a(b,v)$			$a(a(v,v),b)$		
$a(a(v,v),b)$			$a(a(v,v),v)$		

空欄$= a(v,v)$

しかし、この方法は、3－4で$O(\text{patsize}^{\text{rank}} \times \text{sym} \times \text{ht})$のステップと$O(\text{patsize}^{\text{rank}} \times \text{sym})$のスペースを要する。ただし、patsizeはPF中の部分木の個数、symはΣ中の記号の個数、htは木の高さ、即ち、最長のパスである。

5.2 その改良

上記の方法を改良するために、その意味するところを図示してみよう。図3は直接包含グラフとマッチし得る部分木の関係を示している。すなわち、包含グラフ中の各ノードから、それをアーギュメントとしてもつ部分木へ矢印を書き込んだものである。

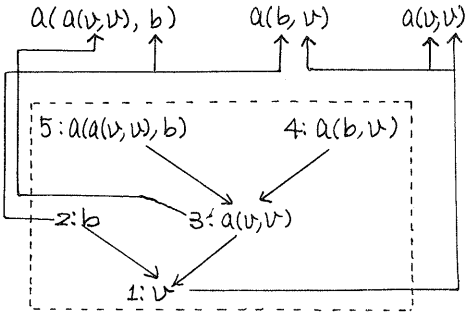


図3. グラフと部分木の関係(1)

テーブル情報を作成するアルゴリズムは、包含グラフの各ノードの組合せに対し、図3の各ノードから出ている部分木への矢印をたどってマッチし得る部分木をテーブル化したものである。例えば、$\langle a(b,v),v\rangle$に対しては、図3中の$\langle 4,2\rangle$をたどり、$\langle 3,2\rangle$すなわち$a(a(v,v),b)$がマッチすることがわかる。

このアルゴリズムに要するスペースを節約するにはテーブルを作らず、図3に示す関係を作ったところで前処理を終り、マッチング時にこの関係から情報を引き出すようにすればよい。しかし、単に図3の関係を引き継ぐのでは、アルゴリズムのステップ数をマッチング時に持ち込むだけであるから、これを少しでも減らす工夫が必要である。

そのために、包含グラフを到達可能なすべての関係を示すグラフにし、さらに、各ノードが部分木とマッチし得る条件、例えば、何番目のアーギュメントであるか等の情報をもたせる。ただし、変数についてはすべてとマッチし得るので何もしない。この考え方で図3を書き直すと図4のようになる。

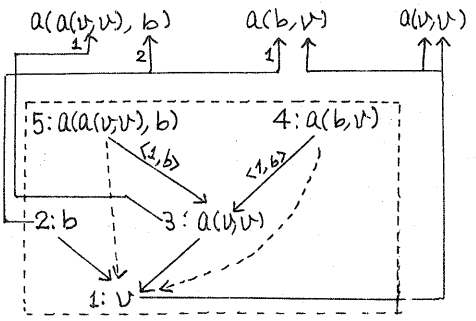


図4. グラフと部分木の関係(2)

現実のプログラミング環境ではパターンすなわち等式が更新されることが少なくない。このような状況に対応し得るには、保持している情報の修正が容易であること、前処理の時間があまり大きくないことなどが要求される。ここで述べた改良法はこれらの条件を満たしている。

6. インタプリタの試作

等式言語をパターンマッチングによって処理するインタプリタを試作した。そこでは、5に述べた上昇型マッチングではなく、下降型マッチングを用いている。この方法は、前処理には時間を要しないが、マッチング時に時間を要する。

4.に掲げた等式によるプログラムはこのインタプリタによって処理されたものである。その処理時間を他の言語との比較で表2に示す。

表2 処理時間の比較

言語 \ Prime(n)	10	20	30	40
HLISP	14	24	34	45
LISPもどき (SECDマシン)	577	796	999	1226
等式言語	544	1973	3797	6147

単位 msec. HITAC M-180

表2からわかるように、その速度は遅い。処理時間を短縮させるために、現在、パターンマッチングをもご述べた方法に変更中なので、その効果については改めて報告したい。

7. おわりに

等式言語によるプログラムがきわめて理解し易く、他の実数型言語に比べても充分な表現力のあることを例示した。また、等式を書き換規則とみなし、計算の過程を実行するインタプリタについて述べた。特に、リダクションに必要なパターンマッチングについて、前処理に要する時間とスペースを改良する方法を検討した。パターンが更新されることを考慮すると有効な方法であると思われる。

等式言語は多くの望ましい性質をもっているが、この種の言語が現実の問題を処理し得る実用的言語になるためにはいくつかの解決しなければならない問題がある。そのうち、次の二つは特に重要であると思われる。

1) データ型や実用上、便利なsyntactic sugarを導入すること
2) 高速の処理系の作成

2)の有望な方向の一つはTurner[4]になるような組合せ子の応用であろう。

参考文献

[1] O'Donnell M: Computing in Systems Described by Equations, Lecture Notes in Comput. Sci. 58, Springer-Verlag (1977).

[2] Hoffmann, C.M. and O'Donnell, M.J: Programming with equations ACM TOPLAS 4-1, pp.83-112 (Jan. 1982)

[3] Hoffmann, C.M. and O'Donnell, M.J: Pattern matching in Trees Journal of ACM 29-1, pp.68-95 (Jan. 1982)

[4] Turner, D. A.: A new implementation technique for applicative languages. Softw. Prac. Exper. 9, pp 31-49 (1979)