プログラミング-言語・基礎・実践- 11－8
(1993. 3. 10)

自己反映計算機能をもつ等式プログラム処理系の実現

佐藤 崇昭　栗原 正仁　大内　東

北海道大学工学部

札幌市北区北13条西8丁目
北海道大学工学部情報工学科

あらまし

自己反映計算とは，因果結合的に自分自身の計算に関して，計算したり，影響を与えたりする計算システムによって実行されるものである．関数型，論理型，オブジェクト指向など，様々な枠組みで，自己反映的な言語が提案されてきている．しかしながら，等式プログラミングの枠組みでは，提案されていない．

本稿では，自己反映計算機能を等式プログラミングの枠組みに導入することを試み，自己反映計算機能をもつ等式プログラム処理系REPSをCLOS(Common Lisp Object System)によって，実現する．

REPSでは，計算状態をリデックス，文脈，書き換え規則の集合によってモデル化している．リーフィケーションによって，これらのメタレベルのオブジェクトは，プログラム中で計算可能なオブジェクト(項)へ変換される．リフレクションは，リデックス，文脈，書き換え規則の集合をREPS項で指定することによって，リダクションマシンの計算状態に影響を与えることができる．

和文キーワード　自己反映計算　因果結合　等式プログラミング　リーフィケーション　リフレクション

An Implementation of an Equational Programming System with Reflective Facilities

Taka-aki SATOH　Masahito KURIHARA　Azuma OHUCHI

Faculty of Engineering, Hokkaido University

Department of Imformation Engineering,
Faculty of Engineering,
Hokkaido University,
Kita 13, Nishi 8, Kita-ku,
Sapporo 060, Japan.

Abstract

Computational Reflection is the activity performed by a computational system which computes about (and possibly affects) its own computation in a causually connected way. Many reflective languages have been proposed in the frameworks of functional programming, logic programming, and object-oriented programming. However, reflective languages are not proposed in the framework of equational programming.

We try to introduce 'reflective' facilities into the framework of an equational programming, and implement a Reflective Equational Programming System REPS in CLOS (Common Lisp Object System).

In REPS, computation is modeled by a redex, a context, a set of rewriting rules. *Reification* translates these meta-objects into computable objects (terms) in programs. *Reflection* specifies a redex, a context and a set of rewriting rules all of which are expressed in REPS term, and can affect the reduction machine registers.

英文 key words　computational reflection　causually connection　equational programming　reification　reflection

1 はじめに

自己反映計算 (*reflective computation*)[1] とは，計算システムが因果結合 (*causually connection*) 的に自分自身の計算に関して行う計算である．自己反映計算は，自己反映システム (*reflective system*) と呼ばれる特別な計算システムによって実行される．計算システムは一般に，ある外部の問題領域に関する計算を実行したり，その問題領域における操作をサポートすることを目的としていると考えることができる．このため，計算システムは，計算の対象となる問題領域を表現している内部構造をもつことが必要となる．計算システムがもつべき内部構造として，問題領域に存在する実体とその実体間の関係を表現しているデータ及び，このデータをどのように操作するかを記述しているプログラムがある．計算は，計算システムのプロセッサ (CPU またはインタプリタ) が，このプログラムを実行することにより，実行されるのである．このとき，計算システムの内部構造と，それが表現している問題領域とが，両者のうちの一方が変化すれば，他方にも必ず，対応する影響が及ぶような関係で結合している．すなわち，計算システムと問題領域は因果結合している．このような枠組みにおいて，メタシステム (*meta-sysmtem*) とは，その問題領域がベースレベルシステムと呼ばれる計算システムであるような計算システムである．メタシステムとベースレベルシステムが，同一のシステムであるとき，この計算システムは，自己反映システムと呼ばれる．

自己反映計算機能をもつプログラミング言語は，自分自身の挙動に関して感知することを必要とされるプログラムを研究するために，様々な枠組みにおいて提案されてきた．例えば，人工知能の分野では，自分自身の挙動をユーザに説明するプログラムにおいて，このような感知が必要である．プログラミング言語の研究においても，拡張性の優れたプログラミング言語を研究する上で，同様なことが必要とされる．すなわち，必要とされるプログラミング言語とは，ユーザがそのプログラミング言語自体を変更するプログラムを，その言語で記述できるものである．Lisp は，このような言語の一つである．なぜなら，ユーザは Lisp 自体を，新しいスペシャルフォーム (*special form*) を定義することによって，変更することができる．新しいスペシャルフォームを定義することによって，Lisp のインタプリタに変更を加えることができる．しかしながら，等式プログラミング [2, 9] の枠組みにおける自己反映計算機能の導入は，提案されていない．

本稿では，等式プログラミングの枠組みにおける自己反映計算機能を考察し，自己反映計算機能をもつ等式プログラミング処理系 (Reflective Equational Programming System，以下では REPS と略記) を与え，CLOS(*Common Lisp Object System*)[3] によって REPS を実現する．

等式プログラミングとは，項書き換えシステムを計算モデルとし，等式を左辺から右辺への書き換え規則として用いる．したがって，等式プログラムは，書き換え規則の集合である．

REPS では，計算状態を書き換え規則の集合とリデックスおよびそのまわりの文脈によって，モデル化している．これらの計算状態を表すメタ・レベルのオブジェクトはリーフィケーション (*reification*) によって，計算可能なオブジェクトとなる．また，リフレクション (*reflection*) によって，任意の書き換え規則の集合や文脈を指定して，REPS の計算状態を変更することができる．

2 REPS の自己反映アーキテクチャ

ここでは，等式プログラムの枠組みにおいて，ユーザが等式プログラム処理系を等式プログラムによって変更することを可能とする自己反映計算アーキテクチャを与える．そのために，等式プログラム処理系の計算をモデル化することを試みる．

2.1 計算のモデル化

等式プログラム処理系とは，リダクションマシンと書き換え規則の集合 (等式プログラム) から構成され，リダクションマシンにユーザが任意の項を入力すると，その正規形を返すものである．これを図 1に示す．

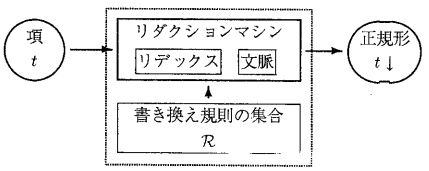

図 1: 等式プログラム処理系

すなわち，等式プログラム処理系における計算とは，書き換え規則の集合 $\mathcal{R}$ のもとで，項 t が与えられると，リダクションマシンが図 2に示すようにリデックスと文脈を変化させながら，t の正規形 $t\downarrow$ を求めることである．

```
1:  begin
2:    while (t にリデックスとなる部分項 t/u が存在) do
3:    begin
4:      t/u の周りを文脈 C とする
5:      t/u と代入θで左辺が照合する書き換え規則
        l = r ∈ R を適用する
6:      t を C[rθ] に書き換え，これを t とする
7:    end
8:    t は正規形 (終了)
9:  end;
```

図 2: リダクションマシンの計算手順

従って，等式プログラムにおいて計算を実行するリダクションマシンの計算状態は，リデックスとその周りの文脈および書き換

え規則の集合の3つの計算上のオブジェクトによってモデル化できると考えられる．このため，REPSでは，この3つオブジェクトによって，リダクションマシンの計算状態をモデル化している．しかしながら，これらはユーザプログラムに対して，メタレベルであるリダクションマシンがもつオブジェクトであるため，プログラムから直接扱うことができない．REPSでは，リーフィケーション (*reification*) によって，これらを項として扱うことを可能とし，リフレクション (*reflection*) によって，これらに対応する項を指定し，計算状態を変更することを可能としている．

2.2 REPSの項

REPSは，項書き換えシステムを計算モデルとする等式プログラミングの枠組みで，自己反映計算機能を付加したものであるから，項書き換えシステムと同様な項を扱うことができる．しかし，REPSでは自己反映計算を実行するために，特別な項であるリーファイア (*reifier*) と呼ばれるものを導入して項の概念を拡張する．ただし，リーファイアは，書き換え規則の左辺全体としてのみ，出現することができるとする．REPSの項は，次のように定義される．

定義 2.1 (REPSの項) 関数シンボルの集合$\mathcal{F}$, 変数シンボルの集合$\mathcal{V}$上で，REPSの項の集合$\mathcal{T}(\mathcal{F},\mathcal{V})$を次の条件を満たす最小集合であるとする．

変数

$x \in \mathcal{V}$ならば，

$x \in \mathcal{T}(\mathcal{F},\mathcal{V})$

複合項

$t_1, t_2, \ldots, t_n \in \mathcal{T}(\mathcal{F},\mathcal{V}), f \in \mathcal{F}$ならば，

$f(t_1, t_2, \ldots, t_n) \in \mathcal{T}(\mathcal{F},\mathcal{V})$

リーファイア

$f(t_1, t_2, \ldots, t_n) \in \mathcal{T}(\mathcal{F},\mathcal{V}), R \in \mathcal{V}, C \in \mathcal{V}$ならば，

$f(t_1, t_2, \ldots, t_n).R.C \in \mathcal{T}(\mathcal{F},\mathcal{V})$

以下では，変数シンボルの集合や関数シンボルの集合を明示する必要のない場合は，項の集合を$\mathcal{T}$で表す．

REPSでは，特別な項であるリーファイアを扱うので，項と書き換え規則の左辺との照合についても，一般的な等式プログラミングとは異なる照合の概念が必要となるので，ここで，項の照合について定義する．

定義 2.2 (項の照合) 項$f(t_1,\cdots,t_n)$が，ある代入θによって，書き換え規則$l = r$の左辺lと照合するのは，次の条件を満たすときである．

- $l \equiv f(S_1,\cdots,S_n).R.C$のとき，

$$f(S_1,\cdots,S_n)\theta \equiv f(term^{\wedge}(t_1),\cdots,term^{\wedge}(t_n))$$

- *otherwise*

$$l\theta \equiv f(t_1,\cdots,t_n)$$

ただし，$term^{\wedge}$は，任意の項$t \in \mathcal{T}$に対して，それと対応するメタ表記の項$T \in \mathcal{T}$へと変換する関数である．

$$term^{\wedge}: \ \mathcal{T} \ \mapsto \ \mathcal{T}$$

2.3 リーフィケーション

リーフィケーションとは，リダクションマシンの計算状態を表すメタレベルのオブジェクトであるリデックス，書き換え規則の集合，文脈をプログラムへ渡すプロセスである．これらのメタレベルのオブジェクトは，プログラム中で計算可能なオブジェクト，すなわち項へと変換されて，プログラムへ渡される．このように，計算状態を表すメタレベルのオブジェクトを，プログラム中で計算可能な項へ変換することをリーファイする (*reify*) という．書き換え規則の集合および文脈は，項ではないので，リーフィケーションの中で，項へと変換する必要がある．

リーフィケーションはリーファイアによって実行される．すなわち，書き換え規則の集合$\mathcal{R}$において，書き換えの対象の項をtとする．ある書き換え規則

$$(f(\cdots).R.C \rightarrow s) \in \mathcal{R}$$

が存在し，tのリデックスとして選ばれた部分項がt/uであるとする．$f(\cdots).R.C$が代入θによって，t/uに照合するならば，この書き換え規則がtに適用される．このとき，リーフィケーションが起こり，$\mathcal{R}$と文脈$\mathcal{C}$が項へと変換され，それぞれリーファイアで指定した変数R, Cへ代入される．そして，t全体が対応する右辺

$$s\theta \bullet \{rules^{\wedge}(\mathcal{R})/R, context^{\wedge}(\mathcal{C})/C\}$$

に書き換えられ，メタレベルのリダクションマシンへ渡される．ただし，$rules^{\wedge}, context^{\wedge}$はそれぞれ，書き換え規則の集合を項へ，文脈を項へ変換する関数である．

$$rules^{\wedge}: \quad \mathcal{R} \ \mapsto \ \mathcal{T}$$
$$context^{\wedge}: \ \mathcal{C} \ \mapsto \ \mathcal{T}$$

リーフィケーションによる書き換えでは，その直前の文脈は，リーファイされて項へ変換されているので，放棄されることになる．

2.4 リフレクション

リフレクションとは，プログラムにおいて項で指定した計算状態 (リデックス，書き換え規則の集合，文脈) へREPSの計算状態を変更するプロセスである．リフレクションは特別な関数シンボル`reduce`が最外部に現れる項が書き換えられるときに起きる．すなわち，REPSが項

$$\texttt{reduce}(T,R,C)$$

を書き換えるとき，次に示すように，項T,R,Cを反映した計算状態のリダクションマシンが起動される．ここで，T,R,Cはそれぞれ，メタ表記の項，項で表現した書き換え規則の集合，項で表現した文脈である．

$$\begin{array}{ll} \text{リデックス} & \mapsto \ term^{\vee}(T) \\ \text{書き換え規則の集合} & \mapsto \ rules^{\vee}(R) \\ \text{文脈} & \mapsto \ context^{\vee}(C) \end{array}$$

ただし，$term^{\vee}$はメタ表記の項をオブジェクト表記へ，$rules^{\vee}$は項を書き換え規則の集合へ，$context^{\vee}$は項を文脈へ，それぞれ変換する関数である．

$$\begin{array}{lrcl} term^{\vee}: & \mathcal{T} & \mapsto & \mathcal{T} \\ rules^{\vee}: & \mathcal{T} & \mapsto & \mathcal{R} \\ context^{\vee}: & \mathcal{T} & \mapsto & \mathcal{C} \end{array}$$

項$\mathtt{reduce}(T,R,C)$を書き換える書き換え規則は，プリミティブな書き換え規則である．

3 REPSの操作的意味論

REPSの操作的意味論として，まず，条件部のない書き換え規則(等式)によって定まる簡約$\twoheadrightarrow$を定義する．ここでは，書き換え規則$l=r$を，書き換えの向きを明示するため，$l \to r$と表す．まず，代入を定義する．

3.1 代入

x_iがすべて異なるような項t_iと変数x_iの対の有限集合

$$\theta = \{t_1/x_1, \cdots, t_n/x_n\}$$

を代入という．このとき，項tに対して，パターン照合によって得られる代入θを作用させた結果$t\theta$を次のように定義する．

定義 3.1 (代入の適用) 任意の項$t \in \mathcal{T}$に対して，照合による代入θを作用させた結果は，次のようになる．

- $t \in \mathcal{V}$ならば，

$$t\theta = \begin{cases} t_i & if\ t \equiv x_i \\ t & otherwise \end{cases}$$

- $t \equiv f(s_1, \cdots, s_m)$ ならば，

$$f(s_1, \cdots, s_m)\theta = f(s_1\theta, \cdots, s_m\theta)$$

- $t \equiv g(u_1, \cdots, u_l).R.C$ならば，

$$(g(u_1, \cdots, u_l).R.C)\theta = g(u_1\theta, \cdots, u_l\theta).R.C$$

リーファイア$g(u_1, \cdots, u_l).R.C$の変数R,Cはそれぞれ，リーフィケーションが起きたときにリーファイされた書き換え規則の集合，文脈を代入するためのものであるので，照合によって得られる代入は適用されない．

定義3.1により，REPSにおける任意の代入θは，$\mathcal{T} \mapsto \mathcal{T}$なる写像である．

3.2 簡約の定義 (I)

ここでは，ユーザが等式プログラムで与える書き換え規則に対応する簡約を定義する．

定義 3.2 (簡約 I) ある書き換え規則$(l \to r) \in \mathcal{R}$，およびパターン照合で得られる代入$\theta$に対して，

- $l \equiv f(\cdots).R.C$ならば(リーフィケーションによる書き換え)，

$$C[f(\cdots)\theta] \to r\tau$$

ただし，τは，

$$\tau = \theta \cdot \{rules^{\wedge}(\mathcal{R})/R, context^{\wedge}(\mathcal{C})/C\}$$

なる代入の合成である．

- *otherwise*

$$C[l\theta] \to C[r\theta]$$

適用された書き換え規則の左辺がリーファイアであれば，リーフィケーションが起こり，文脈が放棄されるが，それ以外の場合では，文脈が保持される．

3.3 簡約の定義 (II)

特別な関数シンボル reduce が最外部に現れる項は，左辺がリーファイアである書き換え規則の右辺全体としてのみ出現することができる．従って，任意の項の真部分項[1]として，出現することはない．

reduce による書き換えが実行されると，リフレクションが起きるため，一般に書き換えの前後では，書き換え規則の集合が異なる．従ってここでは，このような書き換えを明確に定義するために，項tに対して，それを書き換える書き換え規則の集合$\mathcal{R}$を明示して，$\{t\}_{\mathcal{R}}$という表記をする．このような項とそれを簡約する書き換え規則の集合を明示した項を閉包項と呼ぶ．閉包項上の書き換え関係を$\Rightarrow$で表す．

定義 3.3 (準備) 関数シンボル reduce が最外部に現れる項から構成される閉包項

$$\{\mathtt{reduce}(T,R,C)\}_{\mathcal{R}}$$

は，次のように書き換えられる(リフレクション)．

$$\{\mathtt{reduce}(T,R,C)\}_{\mathcal{R}_0} \Rightarrow \{\mathcal{C}[t]\}_{\mathcal{R}}$$

ただし，$t,\mathcal{R},\mathcal{C}$は次のようにして，与えられる．

$$\begin{cases} t = term^{\vee}(T) \\ \mathcal{R} = rules^{\vee}(R) \\ \mathcal{C} = context^{\vee}(C) \end{cases}$$

[1]任意の項tの部分項の集合$S(t)$から，t自身を除いた集合に属する項を真部分項という．

定義 3.4 (閉包項と項における書き換えの関係) 閉包項における書き換えと項における書き換えは，次の条件を満たす．

$$\{s\}_{\mathcal{R}} \Rightarrow \{t\}_{\mathcal{R}'} \text{ならば}, s \to t$$

定義 3.3と定義 3.4から，関数シンボル reduce による簡約が定義される．

定義 3.5 (簡約 II) 関数シンボル reduce による簡約は，定義3.3と定義 3.4から定まる書き換え関係である．

すなわち，$\mathtt{reduce}(T, R, C) \to C[t]$ は，定義 3.3から，項だけを抽出して得られる関係である．

このようにして，定義 3.2と定義 3.5から二項関係$\to$が定義される．条件部をもたない書き換え規則による簡約$\twoheadrightarrow$は，$\to$の反射推移閉包として定義される．条件部をもつ書き換え規則の適用は，その条件部の項を簡約して得られる正規形が true となるときのみに限定される．従って，条件部をもつ書き換え規則によって定まる簡約は，条件部のない書き換え規則の場合よりも制限されたものとなるが，同様に定義される．

4 ベースレベルとメタレベル

REPS におけるベースレベルとメタレベルの関係を図 3に示す．

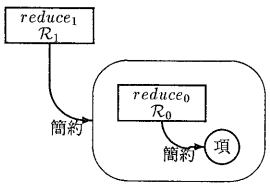

図 3: レベル間の関係

ユーザが与えるプログラム (書き換え規則の集合)$\mathcal{R}_0$において，リーフィケーションを実行することにより，メタレベルのリダクションマシン $reduce_1$が起動する．メタレベルの書き換え規則の集合 $\mathcal{R}_1$には，リフレクションを実行する特別な関数シンボル reduce を定義している書き換え規則を含んでいる．従って，メタレベルへ，reduce が最外部に出現するような項が渡されると，ここでリフレクションが実行される．

5 REPS のシンタクス

ここでは，REPS プログラム (書き換え規則の集合) を記述するためのシンタクスを定義する．

5.1 表記法

REPS では，変数 var，定数 const，複合項 cterm，リーファイア reifier を項として扱う．これらの表記法を BNF 記法で与える．

```
<symbols>::=a|b|c|...|x|y|z|...|f|g|h|...
<var>::='?'<symbols>
<const>::=<symbols>
<cterm>::=<symbols>'('<arg-list>')'
<reifier>::=<cterm>'.'<var>'.'<var>
<term>::=<var>|<const>|<cterm>|<reifier>
<arg-list>::=<term>|<term>','<arg-list>
```

また，REPS プログラム prog の表記法は次のようにして与えられる．

```
<non-cond-rule>::=<const>'='<term>|
                  <cterm>'='<term>|
                  <reifier>'='<term>
<condition>::=' when '<term>
<cond-rule>::=<non-cond-rule><condition>
<rule>::=<non-cond-rule>|<cond-rule>
<rules>::=<rule>|<rule>','<rules>
<prog>::='{'<rules>'}'
```

ただし，すべての書き換え規則は，次の制約を満たすものとする．

制約 5.1 (REPS の書き換え規則)

(1) 書き換え規則 rule の条件部 condition には，簡約によって得られる正規形が true,false となる単項ブール式だけを記述することができる．

(2) 特別な関数シンボル reduce が最外部に出現する項は，左辺がリーファイアである書き換え規則の右辺全体としてのみ記述できる．

5.2 メタ表記

REPS では，メタレベルのオブジェクトである計算状態 (リデックス，書き換え規則の集合，文脈) に対する計算が可能となっている．しかし，ユーザがこれらに対して計算を実行するプログラムを記述するためには，これらがメタレベルにおいて，どのような項で表現されているのかを知らなければならない．項に対して，それと対応するメタレベルでの表記をメタ表記という．すなわち，ユーザが自己反映計算を実行するプログラムを記述するためには，次に示すような，メタ表記を知る必要がある．

5.2.1 項とそのメタ表記

プログラム中に現れる項に対して，そのメタ表記は次のようになる．

変数

```
?x ↦ var(x)
```

定数

```
a ↦ ap(a,())
```

複合項

```
f(g(?y),···)
↦ ap(f,cons(ap(g,cons(var(y),())),···))
```

リーファイア

```
rf(···).?r.?c
↦ reifier(rf,cons(···),var(r),var(c))
```

また，プログラムの記述を簡単にするため，ある項に対して，そのメタ表記を表す記号‘ ^ ’を導入する．項tのメタ表記をTとすると，$T \equiv\hat{}\ t$である．ただし，tが変数を含む項であるとき，その変数への代入が行われた後のインスタンスに対して，上記の関係が成立する．

5.2.2 書き換え規則の集合とそのメタ表記

書き換え規則$l = r\ when\ c$のメタ表記は，

$$\texttt{rule}[L,R,C]$$

である．ただし，L,R,Cはそれぞれ，l,r,cのメタ表記であり，条件部をもたない書き換え規則の場合は，Cは () となる．書き換え規則の集合のメタ表記は，書き換え規則のメタ表記$RULE$のリスト

$$\texttt{cons}[RULE,\ \cdots]$$

である．

5.2.3 文脈とそのメタ表記

出現位置のメタ表記は，空リスト () および自然数

```
s(0),s(s(0)), ···
```

からなるリストで表現される．例えば，出現位置$\epsilon, 1\cdot 2$のメタ表記はそれぞれ，

```
ϵ     ↦  ()
1·2   ↦  cons(s(0),cons(s(s(0)),()))
```

文脈$\mathcal{C}$のメタ表記は，空の場所 □ の出現位置のメタ表記Pと書き換えの対象の項全体のメタ表記Tで表される．

$$\texttt{context}(P,T)$$

6 REPS によるプログラミング

ここでは，REPS の自己反映計算機能を応用したプログラムを幾つか挙げる．

6.1 項の判定

項が変数であるかを判定する関数 varp，定数であるかを判定する関数 constp，複合項であるかを判定する関数 ctermp はそれぞれ，次のように記述できる．

```
{
    varp(var(?x)).?r.?c
    = reduce(^true,?r,?c),
    varp(ap(?f,?args)).?r.?c
    = reduce(^false,?r,?c),

    constp(var(?x)).?r.?c
    = reduce(^false,?r,?c),
    constp(ap(?c,())).?r.?c
    = reduce(^true,?r,?c),
    constp(ap(?f,cons(?arg,?args))).?r.?c
    = reduce(^false,?r,?c),

    ctermp(ap(?f,cons(?arg,?args))).?r.?c
    = reduce(^true,?r,?c),
    ctermp(ap(?f,())).?r.?c
    = reduce(^false,?r,?c),
    ctermp(var(?x)).?r.?c
    = reduce(^false,?r,?c),
}

    REPS:varp(?x)
    ->true
    REPS:varp(f(a))
    ->false
```

与えられた 2 つの項が等しい項であるかを判定する関数 eq-term は，次のように記述できる．ここで，関数 eq-sym はシンボルの等価性を判定するプリミティブな関数であるとする．

```
{
    if(true,?x,?y) = ?x,
    if(false,?x,?y) = ?y,

    eq-term(var(?x),var(?y)).?r.?c
    = reduce(^eq-sym(?x,?y),?r,?c),
    eq-term(var(?x),ap(?f,?args)).?r.?c
    = reduce(^false,?r,?c),
    eq-term(ap(?f,?args),var(?x)).?r.?c
    = reduce(^false,?r,?c),
    eq-term(ap(?c1,()),ap(?c2,())).?r.?c
    = reduce(^eq-sym(?c1,?c2),?r,?c),
    eq-term(ap(?f1,cons(?arg1,?args1)),
```

```
        ap(?f2,cons(?arg2,?args2)))
= reduce(^eq-args(cons(?arg1,?args1),
                  cons(?arg2,?args2)),?r,?c)
   when eq-sym(?f1,?f2),
eq-term(ap(?f1,cons(?arg1,?args1)),
        ap(?f2,cons(?arg2,?args2)))
= reduce(^false,?r,?c)
    when not(eq-sym(?f1,?f2)),
eq-term(ap(?f,cons(?arg,?args)),
        ap(?c,())).?r.?c
= reduce(^false,?r,?c),
eq-term(ap(?c,()),
        ap(?f,cons(?arg,?args))).?r.?c
= reduce(^false,?r.?c),
eq-term(ap(?c1,()),ap(?c2,())).?r.?c
= reduce(^eq-sym(?c1,?c2),?r,?c),

eq-args(cons(var(?x),?args1),
        cons(var(?y),?args2))
= eq-args(?args1,?args2) when eq-sym(?x,?y),
eq-args(cons(var(?x),?args1),
        cons(var(?y),?args2))
= false when not(eq-sym(?x,?y)),
eq-args(cons(var(?x),?args1),
        cons(ap(?f,?args),?args2))
= false,
eq-args(cons(ap(?f,?args),?args1),
        cons(var(?x),?args2))
= false,
eq-args(cons(ap(?f1,?args1),?x),
        cons(ap(?f2,?args2),?y))
= if(eq-args(?args1,?args2),
    eq-args(?x,?y),
    false) when eq-sym(?f1,?f2),
eq-args(cons(ap(?f1,?args1),?x),
        cons(ap(?f2,?args2),?y))
= false when not(eq-sym(?f1,?f2)),
eq-args((),()) = true,
}
```

```
REPS:eq-term(a,b)
->false
REPS:eq-term(f(?x),f(?x))
->true
```

6.2 パターン照合

項tがパターンとして与えられる項pに照合するかを判定する関数pattern-matchは，次のように記述できる．

```
{
    if(true,?x,?y) = ?x,
    if(false,?x,?y) = ?y,

    pattern-match(?term1,?term2).?r.?c
    = reduce(^pm(?term1,?term2),?r,?c),

    pm(var(?x),var(?y)) = true,
    pm(var(?x),ap(?f,?args)) = false,
    pm(ap(?f,?args),var(?x)) = true,
    pm(ap(?c1,()),ap(?c2,()))
    = ture when eq-sym(?c1,?c2),
    pm(ap(?c1,()),ap(?c2,()))
    = false when not(eq-sym(?c1,?c2)),
    pm(ap(?f1,cons(?arg1,?args1)),
       ap(?f2,cons(?arg2,?args2)))
    = pm-args(cons(?arg1,?args1),
              cons(?arg2,?args2))
       when eq-sym(?f1,?f2),
    pm(ap(?f1,cons(?arg1,?args1)),
       ap(?f2,cons(?arg2,?args2)))
    = false when not(eq-sym(?f1,?f2)),

    pm-args((),()) = true,
    pm-args((),cons(?arg,?args)) = false,
    pm-args(cons(?arg,?args),()) = false,
    pm-args(cons(?arg1,?args1),cons(?arg2,?args2))
    = pm-args(?args1,?args2) when pm(?arg1,?arg2),
    pm-args(cons(?arg1,?args1),cons(?arg2,?args2))
    = false when not(pm(?arg1,?arg2))
}
```

```
REPS:pattern-match(f(a),?y)
->true
REPS:pattern-match(g(h(a,b),c),g(h(?x,b),a))
->false
```

6.3 正規形の判定

条件付きの書き換え規則では，ある項が正規形であるかどうかを，書き換え規則の左辺との照合によって，判定することはできない．したがって，項tの正規形の判定を，tとこれを1ステップ書き換えた項t'が等しい項であるかで判定する．この判定にeq-termを利用すると，正規形の判定関数normal-form-pは次

のように記述できる．

```
{
    ... eq-term の定義は省略 ...

    normal-form-p(?term).?r.?c
    = reduce(?term,?r,
             context(cons(s(s(0)),()),
               ap(eq-term,
                  cons(?term,cons(ap([],()),())))))
}
```

このような書き換え規則の集合 $\mathcal{R}_n$ と，正規形を判定の対象の等式プログラム $\mathcal{R}_{prog}$ の書き換え規則の集合として，REPS に与えると，正規形の判定ができる．

6.4 エラー処理

自然数上の割り算プログラムにおけるエラー処理を例として示す．ある数を 0 で割るような場合があれば，これは明らかにエラーである．通常の等式プログラムでは，割り算 div を定義する書き換え規則だけではなく，すべての関数のに対して，エラー処理のための書き換え規則も付け加えなくてはならない．しかし，REPS では，文脈をプログラムによって変更できるので，div を定義する書き換え規則にエラー処理のための書き換え規則を一つ付け加えるだけで，簡約の実行中に項の深い出現位置でエラーが起きた場合でも，直ちに error へ書き換えることができる．

```
{
    minus(?x,0)=?x,
    minus(0,?x)=0,
    minus(s(?x),s(?y))=minus(?x,?y),

    div(0,?x)=0,
    div(?x,^0).?r.?c=reduce(^error,(),[]),
    div(s(?x),s(?y))
    =s(div(minus(?x,?y),s(?y)))
}

REPS:div(s(s(0)),s(0))
->s(0)
REPS:div(s(0),0)
->error
```

7 おわりに

本稿では，等式プログラミングの枠組みにおける自己反映計算を考察し，これを導入した．また，自己反映計算機能をもつ等式プログラム処理系 REPS の仕様を与え，CLOS により実現し，応用例として，REPS プログラムを幾つか示した．

REPS の自己反映計算機能は，すべての計算に関して直接的に，貢献するわけではない．REPS の自己反映計算機能が有効となるのは，項に関する情報，書き換え規則の集合，文脈などの計算状態を表している計算システムの内部構造に関する計算である．自己反映計算は，計算状態を表すメタレベルの情報を，プログラムによって計算可能にするための明確なインタフェースを与えるものと考えることができる．

REPS の応用例は，自己反映計算機能を積極的に利用することによって，本稿で挙げた他にも考えられる．計算の実行中に動的に書き換え規則の集合を変更させる計算システム [7] や，書き換え規則の条件部に正規形であるかを判定する関数のような自己反映計算を利用した単項ブール式を記述すること [8] など，様々な応用例があるものと考えられる．

謝辞

本研究の一部は，文部省科学研究費補助金 No.04650298 の助成による．

参考文献

[1] Maes, P., Issues in computational reflection, in Maes, P. and Nardi, D. ed., Meta-Level Architectures and Reflection, North-Holland, (1988), 21-35.

[2] O'Donnel, M.J., Equational Logic as a Programming Language, MIT Press, (1985).

[3] Winston, P.H. and Horn, B.K.P., LISP, 3rd ed., Addison-Wesley, (1989).

[4] Wand, M. and Friedman, D.P., The mystery of the tower revealed: a non-reflective description of the reflective tower, in Maes, P. and Nardi, D. ed., Meta-Level Architectures and Reflection, North-Holland, (1988),111-134.

[5] Danvy, O. and Makmkjær, K., Intentions and extensions in the reflective tower, Proc. ACM LISP and Functional Programming, (1988), 327-341.

[6] Masahito Kurihara, Azuma Ohuchi, An Algebraic Specification of a Reflective Language, Proc. COMPSAC'91(1991).

[7] 馮 速, 坂部 俊樹, 稲垣 康善, 動的項書き換え計算モデルとその応用, 電子情報通信学会研究報告, COMP 91-47, 31-40, (1991).

[8] 山田 順之介, 停止性をもつメンバシップ条件付き TRS の合流性について, 電子情報通信学会論文誌, D-I, Vol. J74-D-I No.9 666-674, (1991).

[9] 富樫 敦, "等式プログラミングから融合型プログラミングへ", 新しいプログラミングパラダイム, 井田編, 共立出版, (1988), 187-218.

[10] 井田哲雄, 計算モデルの基礎理論, 岩波書店, (1991), 223-296.