HOKKAIDO UNIVERSITY

Title	項書換えシステムにおける自己反映計算
Author(s)	栗原, 正仁; 佐藤, 崇昭; 大内, 東
Citation	北海道大學工學部研究報告, 167, 57-66
Issue Date	1994-01-14
Doc URL	http://hdl.handle.net/2115/42410
Type	bulletin (article)
File Information	167_57-66.pdf

Instructions for use

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

北海道大学工学部研究報告
第167号（平成6年）

Bulletin of the Faculty of Engineering
Hokkaido University, No. 167 (1994)

項書換えシステムにおける自己反映計算

栗原　正仁　　佐藤　崇昭　　大内　東

（平成5年8月31日受理）

Reflective Computation in Term Rewriting Systems

Masahito, KURIHARA, Taka-aki SATO and Azuma OHUCHI

(Received August 31, 1993)

Abstract

Reflection is a mechanism that makes computational systems highly flexible by allowing them to manipulate meta objects that represent the current state of its own computation. We propose Reflective Equational Programming System (REPS), which is a framework for programming languages based on term rewriting systems equipped with reflective facilities called reification and deification. The arguments of a redex, the context around the redex and the current set of rewrite rules are considered as meta objects in REPS, and by the reification operation they are transformed into constructor terms considered as base-level objects. The deification operation transforms base-level constructor terms into meta objects.

1．まえがき

自己反映計算（reflection）とは，計算システムが「自分自身」に関するいわゆるメタ的な情報を参照したり変更したりすることを含むような計算である。自己反映計算をうまく使えば複雑で高度なプログラムを統一的な思想に基づいて簡潔に記述できる可能性があることから，最近この研究が活発に行われている。その基本的な考え方については，菅野ら[15]，田中[17]，Maes[12]などの解説を参照していただきたい。

このような計算システムは，必然的に，プログラムをデータとして扱ったり，逆にデータをプログラムとして解釈実行する機能を持つことが必要である。もちろん，現在の計算機の大多数を占めるノイマン型計算機においては，アセンブリ言語を使えばこのようなことが可能なのは明らかである。しかし，高級言語レベルで如何にしてそれを実現するかが問題である。高級言語はそれぞれの思想に従って計算機を抽象化したものと考えられるから，自己反映計算を如何に抽象化するかが研究の一つのポイントとなる。

歴史的にはこの試みは関数型言語をベースとして始められた。Smith[14]の3-Lispを始めとし，Wandら[21]，Danvyら[2]，Bawden[1]，Jeffersonら[7]などの研究がある。論理型言語をベースとするものとしては，菅野[16]，田中[18]など，オブジェクト指向言語をベースとするものとしては，松

情報工学科　システム工学講座

岡ら[13]などがある。しかし，書換え型の計算モデルをベースとする言語である項書換えシステムに自己反映計算を導入する試みは今のところ見られない。本論文ではそれを試み，REPS(Reflective Equational Programming System) という言語の枠組みを提案する。

自己反映計算を実現するアプローチには少なくとも2つのものが知られている。第一のもの(これを直接法と呼ぼう)は，言語 L のインタプリタを言語 L で記述するメタ循環インタプリタという技法を応用して，インタプリタの(無限の)階層(これをリフレクティブタワー(reflective tower)と呼ぶ) を実現するものである。この言語における自己反映計算の基本機能はインタプリタの階層中を上下に移動して計算を実行することである。関数型言語における直接法の例としては，Smith[14]の3-Lisp や Jefferson ら[7]の Simple Reflective Interpreter がある。

それに対し，第二のアプローチ（これを間接法と呼ぼう）は，プログラムやその実行環境などの「メタオブジェクト」からユーザプログラムがアクセスすることのできる「ベースレベルオブジェクト(データ)」への変換およびその逆変換を基本機能とするものである。前者の機能を reification，後者を deification という。この枠組みによってリフレクティブタワーという概念の説明がなされることもあるが，本来このアプローチはその概念を必ずしも必要としない。関数型言語における間接法の例としては，Wand ら[21]の Brown や Danvy ら[2]の Blond がある。2つのアプローチの違いを明確に理解するには，直接法については Jefferson ら[7]，間接法については栗原ら[10]を参照し，実現手法を比較されたい。よく似た2つの言語に対して各アプローチによって簡潔なインタプリタが具体的に実現されている。

本論文で提案する REPS は間接法に基づいて，項書換えシステムに自己反映計算機能を取り入れている。項書換えシステムの一般的な解説は二木ら[5]，井田[6]，Dershowitz ら[3]，Klop[9]などを参照していただきたい。項書換えシステム R における計算は，項の集合上の二項関係$\rightarrow_R$により定義されている。ここで，$s \rightarrow_R t$ が成り立つのは，ある文脈 $C[]$，代入 θ，および書換え規則 $\ell \rightarrow r \in R$ が存在し，$s \equiv C[\ell\theta]$ かつ $t \equiv C[r\theta]$ が成り立つときに限る。この定義中の文脈 $C[]$ およびシステム R 自体は，項書換えシステムの計算対象ではなく，インタプリタ中に存在するメタオブジェクトである。また，代入 θ における変数x に関する値$x\theta$ は項ではあるが，これは一般に，計算式のようなプログラムの一部と解釈され，ユーザプログラムにとっては必ずしもデータとして扱われないことから，これもメタオブジェクトと考える。それに対し，REPS では構成子と呼ばれる関数記号のみからなる項（構成子項）をデータとみなす。REPS は上記のメタオブジェクト（$x\theta$，$C[]$，R）を構成子項に変換する reification，および構成子項をメタオブジェクトに変換して自らの実行環境を変更する deification を基本機能として提供している。

本論文は以下のように構成されている。2章では REPS の仕様を述べている。まず，構文を定義し，次に，自己反映計算の基本操作である reification と deification を定義し，最後に REPS の操作的意味を簡約関係として定義している。3章では著者らの作成した REPS 処理系の実現について述べている。ただし，詳細なプログラミング技法ではなく，メタオブジェクトと構成子項相互の変換の方法が説明の中心である。4章は本論文の内容をまとめている。項書換えシステムの研究分野における自己反映計算の応用と今後の課題を展望している。

2．仕　　様

2.1 構文

関数記号の集合を F，変数記号の集合を V とする。各関数記号 $\mathrm{F} \in F$ には項数 (arity) と呼ばれる非負整数 *arity*(F)が付随している。項数0の関数記号を定数記号という。F と V から作ら

れる項の集合を $T(F, V)$ で表す。変数記号の集合や関数記号の集合を明示する必要のない場合は，項の集合を T で表す。

次に，自己反映計算を引き起こすための構文を導入する。t，c，$r \in T$ が項のとき，その3組 $t.c.r$ を自己反映式という。項および自己反映式を総称して式という。自己反映式は項ではない。項が書換えの対象になるのに対し，自己反映式はそうならない。自己反映式は書換え規則の左辺または右辺，すなわちプログラムの一部としてのみ現われる表現であり，システムの自己反映機能とのインタフェースの役割を果たす。一方，項も通常の項書換えシステムと同様に，書換え規則の左辺または右辺を構成できる。そのため，便宜上，項と自己反映式を総称して式と呼ぶことにした。2つの式 e，e' が（構文的に）同じであるとき，$e \equiv e'$ と書く。

式の順序対 $e \rightarrow e'$ を書換え規則，式 e，e' をそれぞれその規則の左辺，右辺という。

書換え規則の集合を項書換えシステムという。本論文では，項書換えシステムを計算機プログラムとみなしているので，これを単にプログラムと呼ぶこともある。

書換え規則の形は，左辺，右辺がそれぞれ項であるか自己反映式であるかによって，4つに分類できる。後に簡約の定義で明らかになるが，左辺が自己反映式である書換え規則は reification，右辺が自己反映式である書換え規則は deification という自己反映計算特有の操作を引き起こすために用いられる。

関数記号の集合 F は，便宜上，共通部分をもたない2つの集合 F_D と F_C に分割される。F_D，F_C に属する関数記号をそれぞれ演算子（operator），構成子（constructor）と呼ぶ。構成子のみから作られる（変数も含まない）項を構成子項と呼び，その集合を $T(F_C)$ で表す。

書換え規則の左辺の最も左の位置に現われる記号は必ず演算子であることとする。すなわち，左辺が項 $F(\ldots)$ あるいは自己反映式 $F(\ldots).c.r$ ならば，$F \in F_D$ である。通常のプログラミング言語で説明すれば，演算子は手続き，構成子はデータ構造（レコード構造）を表現していると解釈される。特に，構成子項は通常の意味での「データ」を表現している。一方，構成子項とは限らない一般の「項」は，Lisp 言語においてフォームと呼ばれるもの（評価可能なオブジェクト）に相当する。そのため，本論文では「項」と「フォーム」，「構成子項」と「データ」という言葉をそれぞれ同じ意味で用いる。

項 s，およびその中に含まれる部分項の生起 t を考えよう。非形式的には，t のまわりの s に関する情報を文脈（context）と呼ぶ。この文脈を $C[]$ と表すとき，$C[t]$ は項 s と構文的に同じ項を表すものと定義する。また，$C[u]$ は $C[t]$ 中の生起 t を項 u で置き換えて得られる項を表す。

形式的には，文脈は s 中の生起 t を特別な定数記号 □ に置き換えた項として表現されることが多い。たとえば，項 f(a, g(b), x) における g(b) のまわりの文脈は f(a, □, x) である。しかし，本論文では，文脈は「項」で表現される必要はないことを強調するため，あえて，形式的な定義をしないでおく。実際，項書換えシステムの処理系では，ポインタやスタックを用いて暗黙に文脈が表現されていることが多いであろう。

2.2 reification と deification

REPS では，次の3つの関数の存在を仮定している。

$term^{\wedge}$: $T \mapsto T(Fc)$
$context^{\wedge}$: $C \mapsto T(Fc)$
$rules^{\wedge}$: $R \mapsto T(Fc)$

定義域 T，C，R はそれぞれ，項の集合，文脈の集合，プログラムの集合である。各関数の値域はいずれも構成子項の集合 $T(Fc)$ である。

さらに，REPS は次の3つの部分関数の存在を仮定している。

$term^{\vee}$: $T(Fc) \mapsto T$
$context^{\vee}$: $T(Fc) \mapsto C$
$rules^{\vee}$: $T(Fc) \mapsto R$

以上の 6 つの(部分)関数は以下の条件を満たさなくてはいけない。各 $f \in \{term, context, rules\}$ について，定義域中のすべての x に対して，

- $f^{\vee}(f^{\wedge}(x)) = x$
- $f^{\vee}(x)$ が定義されているならば，$f^{\wedge}(f^{\vee}(x)) = x$

最初の 3 つの関数は reification という操作の際に，メタオブジェクトをベースレベル・オブジェクトに変換するために用いられる。項，文脈，プログラムは計算の実行環境の一部(すなわち，メタオブジェクト）であり，通常はユーザプログラムからデータとしてアクセスすることはできない。(ここでは項をデータではなくフォームと見ている。）各関数により，メタオブジェクトは構成子項，すなわちユーザプログラムがアクセスできるデータに変換される。こうして得られた構成子項を，それぞれ，項，文脈，プログラムのメタ表現という。

最後の 3 つの部分関数は reification の逆操作 deification の際に用いられるもので，項，文脈，プログラムの（適切な）メタ表現をそれぞれメタオブジェクトである項，文脈，プログラムに変換する。メタ表現が「適切な」構成子項でないときは，この変換は定義されない（でエラーになる）のでこれらは部分関数である。

要するに，reification は計算の実行環境をベースレベルからメタレベルへ移行させる操作である。あるいは，相対的には同じことであるが，別な見方をすれば，メタレベルの実行環境をベースレベルでアクセスできるデータに変換する操作である。deification は reification の逆操作である。なお，reification, deification の辞書的な意味は，後者の見方に基づいて，それぞれ「具象化」，「神格化」だが，適切な和訳が定着していないので，本論文では英単語のまま用いる。

著者らが作成した処理系における各（部分）関数の定義は3章で述べる。

2.3 簡約

項 s とプログラム R の順序対 (s, R) をリダクションマシンの状態と呼ぶ。REPS における計算は，状態の集合上の二項関係⇒でモデル化される。(⇒はこの段落のすぐ後で定義される。)$\Rightarrow^*$ は⇒の反射推移閉包を表す。すなわち，$(s, R) \Rightarrow \cdots \Rightarrow (s', R')$ なる長さ 0 以上の列が存在するとき，$(s, R) \Rightarrow^* (s', R')$。また，$(s, R) \Rightarrow^* (s', R')$ が成り立ち，かつ，$(s', R') \Rightarrow (s'', R'')$ なる状態 (s'', R'') が存在しないとき，$(s, R) \Rightarrow^! (s', R')$ と書く。このとき，(s', R') は (s, R) の正規形であるという。また，s' は s の（R に関する）正規形であるともいう。⇒を 1 ステップ簡約関係と呼ぶ。(以下の定義で断りなく使われている記号は，存在限量子（∃）で限

量されているものとして，読んでいただきたい。)

定義2.1（1ステップ簡約関係）（s，R）⇒（s'，R'）が成り立つのは，$s \equiv C[u]$ を満たす文脈 $C[]$ と部分項 u，および代入 θ，書換え規則 $e \to e' \in R$ が存在し，以下の(1)，(2)が成り立つときである。

(1) **照合条件**

normal case：左辺 e が項で，かつ，$u \equiv e\theta$；または

reification：左辺 e が自己反映式 $t.c.r$ で，かつ，

$$u \equiv f(s_1, \ldots, s_n)$$
$$t\theta \equiv f(term^{\wedge}(s_1), \ldots, term^{\wedge}(s_n))$$
$$c\theta \equiv context^{\wedge}(C[])$$
$$r\theta \equiv rules^{\wedge}(R)$$

(2) **置換条件**

normal case：右辺 e′ が項で，かつ，

$$s' \equiv C[e'\theta]$$
$$R' = R \quad ；または$$

deification：右辺 e' が自己反映式 $t'.c'.r'$ で，かつ，

$$(H(t'\theta,\ c'\theta,\ r'\theta),\ R) \Rightarrow^{!} (H(t!,\ c!,\ r!),\ R_1)$$
$$s' \equiv context^{\vee}(c!)\ [term^{\vee}(t!)]$$
$$R' \equiv rules^{\vee}(r!)$$

ただし，H はあらかじめ決められている特定の構成子である。

この定義は簡約を（静的な）二項関係として見ているが，以下では，計算という動的な観点からこれを操作的に説明する。リダクションマシンの状態が（s，R）であるとする。簡約による計算は，照合および置換の2ステップからなる。

照合ステップでは，マシンは何らかの戦略を用いて，(1)の条件を満たす部分項 u，文脈 $C[]$，書換え規則（$e \to e'$）$\in R$ および代入 θ を求める。u をリデックスという。(u が定まれば $C[]$ は $s \equiv C[u]$ より一意に定まる。) 書換え規則の左辺が項であるものに対しては，この処理は通常の項書換えシステムと同じで，リデックス（の候補）u と書換え規則（の候補）の左辺 e の照合（パターンマッチング）により θ が（もし存在すれば）求められる。一方，左辺が自己反映式 $t.c.r$ の場合は，reification 操作が実行される。まず，リデックス（の候補）の各引数，リデックスのまわりの文脈，および現在のプログラムがメタ表現に変換される。これらと t,c,r を照合することにより θ が求められる。

置換ステップではマシンの状態を更新する。書換え規則の右辺 e' が項のときは普通の項書換えシステムと同じで，リデックス u が右辺のインスタンス $e'\theta$ に置き換えられる。右辺が自己反映式 $t'.c'.r'$ の場合は，3つのインスタンス $t'\theta, c'\theta, c'\theta$ から作られた項 $H(t'\theta,\ c'\theta,\ r'\theta)$ が簡約される。その正規形を $H(t!,\ c!,\ r!)$ としよう。次に，deification 操作が実行される。$t!$，$c!$，$r!$ が，それぞれ項，文脈，およびプログラムに変換され，マシンの新たな状態が構成される。文脈 $C[]$ とプログラム R，および途中で生成されたプログラム R_1 は放棄される。これはユーザプログラムによってマシンの状態が完全に決定されるためである。

3．実　　現

3.1 REPS 処理系の構文

前節で定義したREPSの構文は，REPSの理論的基礎を論じるための抽象的なものである。REPS処理系では少し異なる構文を用いているので，まずそれを述べる。

変数記号は疑問符'?'で始まる名前により関数記号と区別される。たとえば，?x は変数記号である。

項の引数を囲むには丸括弧でなく角括弧を用いる。すなわち，$F(t_1, \ldots, t_n)$ではなく，$F[t_1, \ldots, t_n]$ である。ただし，F が定数記号（すなわち，$n=0$）のときは，F が演算子か構成子により構文が異なる。F が演算子ならば F[]，構成子ならば F と書く。

項の特別な場合として，「リスト」を表現するための特別な構文がある。これは Lisp 言語におけるリストの表記法と同じものであり，それらは cons および nil という構成子を用いて構成された項を表す。

$() \equiv \mathrm{nil}$
$(X.\ Y) \equiv \mathrm{cons}[X,\ Y]$
$(X\ Y\ \ldots\ Z) \equiv (X\ .(\ldots\ .(Z\ .\mathrm{nil})\ldots)))$

3.2 項のメタ表現

項をメタ表現に変換する $term^{\wedge} : T \mapsto T(F_C)$ およびその逆変換 $term^{\vee} : T(F_C) \mapsto T$ を定義しよう。この目的のため，各関数記号 $F \in F$ に対し，構成子でかつ定数記号である $F' \in F_C$ が1対1に対応して存在していると仮定する。これは，正の項数をもつ関数記号 F はそれ自体では項ではないことによる。メタ表現では F 自体も項すなわちデータとして扱いたいので，F と同一視できる定数記号 F'を導入するのである。前節と同様の意味で，F'を F のメタ表現という。F'が構成子であるのは，メタ表現に対する要請による。

特別な場合として，F が構成子でかつ定数記号であるときは F'≡F とする。すなわち，この場合，両者は「同一視」されるというよりはむしろ，さらに強く，「同一」である。この定義と，任意の関数記号 F のメタ表現 F'が構成子でかつ定数記号であることから，F のメタ表現（さらに，メタメタ表現，一般にメタ^{n}表現）はいずれも F'となる。

REPS 処理系では，F と F'を構文的に区別せず，どちらも F で表している。F の項数が正なら，その使われ方 F[...] から構文的に明らかに定数記号 F'を表す F と区別できる。また，F が演算子である定数記号なら F[]という表記法により構成子の F から区別される。最後に，F が構成子である定数記号なら，約束により F'≡F であるから問題を生じない。

次に，変数記号のメタ表現を定義しよう。変数記号は?x のように疑問符から始まる名前をもっている。REPS 処理系ではそのメタ表現を構成子項 var[x]と定義している。ここで，var はこの目的のために用いられる特別な構成子，x は変数名の先頭の疑問符を取り除いた名前をもつ定数記号（構成子）である。（任意の変数記号?x に対し，このような x が存在すること，およびその逆を仮定する。）

最後に，関数記号を含む項 $F[t_1, \ldots, t_n]$のメタ表現を定義する。これはリスト（F' $term^{\wedge}(t_1) \ldots term^{\wedge}(t_n)$）として再帰的に定義する。ただし，$F'$は実際には F が用いられる。以上をまとめたのが次式である。たとえば，$term^{\wedge}$(F[g[], h[c]])≡(F (g) (h c)) となる。

$term^{\wedge}(?\mathrm{x}) \equiv \mathrm{var}[\mathrm{x}]$,　when ?x $\in V$
$term^{\wedge}(\mathrm{c}) \equiv \mathrm{c}$,　when $c \in F_C$, $arity(\mathrm{c}) = 0$

$$
\begin{aligned}
&\mathit{term}^{\wedge}(\mathrm{F}[t_1, \ldots, t_n]) \equiv (\mathrm{F}'\ \mathit{term}^{\wedge}(t_1) \ldots \mathit{term}^{\wedge}(t_n)), \\
&\qquad\qquad \text{when } \mathit{arity}(\mathrm{F}) > 0 \text{ or} \\
&\qquad\qquad\qquad \mathit{arity}(\mathrm{F}) = 0 \wedge \mathrm{F} \in F_D \\
&\mathit{term}^{\vee}(\mathrm{vam}[\mathrm{x}]) \equiv ?\mathrm{x}, \quad \text{when } \mathrm{x} \in F_C \\
&\mathit{term}^{\vee}(\mathrm{c}) \equiv \mathrm{c}, \qquad \text{when } \mathrm{c} \in F_C \\
&\mathit{term}^{\vee}(\mathrm{F}'\ t_1 \ldots t_n) \equiv \mathrm{F}[\mathit{term}^{\vee}(t_1), \ldots, \mathit{term}^{\vee}(t_n)], \\
&\qquad\qquad \text{when } \mathrm{F}' \in F_C
\end{aligned}
$$

3.3 文脈のメタ表現

メタ表現の最適な設計は，処理系自体の簡約の処理方式に依存するほか，応用プログラムがそのメタ表現に対してどのような操作を頻繁に行うかに依存する。文脈の場合にもこのことがあてはまる。文脈は形式的には定数記号□を1個もつ項として定義されるが，その他にも多くの代替案が考えられる。現在のところ，項書換えシステムの自己反映計算は本研究が最初の例であるため，その応用範囲が明確に定まっておらず，この選択は難しい。したがって，現在のREPS処理系は著者らがたてた1つの仮定に基づいて設計されている。それは，「文脈においては，□により近い位置の情報ほどアクセス頻度が高いであろう」というものである。すなわち，□のすぐ上の関数記号，あるいはすぐ左右の項ほどアクセスが多いと仮定する。□というのは何らかの意味で現在注目されている部分項が存在する位置であり，その位置に近い位置にある情報ほど重要性があるのは自然な考え方であろう。このため，REPS処理系では文脈のメタ表記を下記のように定める。

$$
\begin{aligned}
&\mathit{context}^{\wedge}(\square) \equiv \mathrm{hole} \\
&\mathit{context}^{\wedge}(C[\mathrm{F}[t_1, \ldots, t_{i-1}, \square, t_{i+1}, \ldots, t_n]]) \\
&\quad \equiv \mathrm{context}[\mathrm{F}', (\mathit{term}^{\wedge}(t_{i-1}) \ldots \mathit{term}^{\wedge}(t_1)), \\
&\qquad\qquad (\mathit{term}^{\wedge}(t_{i+1}) \ldots \mathit{term}^{\wedge}(t_n)) \\
&\qquad\qquad \mathit{context}^{\wedge}(C[])]
\end{aligned}
$$

ただし，hole，context は構成子，F'はFのメタ表現である。たとえば，文脈G(F(A, □, ?x))のメタ表現はcontext[F, (A), (var[x]), context[G, nil, nil, hole]] である。

context[*f, lc, rc, uc*] の引数 *lc, rc, uc* をそれぞれ，左文脈，右文脈，上文脈と呼ぶ。左文脈の項の順序が逆順になっているのは，リストにおいては左側の要素ほどアクセス時間が短いからである。(左，右，上とは，文脈の項による表現をよく知られた木構造で2次元に図式化したときの方向を指す。) 最後に，文脈の deification 手続きを定義する。

$$
\begin{aligned}
&\mathit{context}^{\vee}(\mathrm{hole}) = \mathrm{hole} \\
&\mathit{context}^{\vee}(\mathrm{context}[F', (t_{i-1} \ldots t_1), \\
&\qquad\qquad (t_{i+1} \ldots t_n), \\
&\qquad\qquad uc]) \\
&\quad = C[F[\mathit{term}^{\vee}(t_1), \ldots, \mathit{term}^{\vee}(t_{i-1}), \square, \\
&\qquad \mathit{term}^{\vee}(t_{i+1}), \ldots, \mathit{term}^{\vee}(t_n)]]
\end{aligned}
$$

ただし，$C[] = \mathit{context}^{\vee}(uc)$ である。

3.4 プログラムのメタ表現

プログラム，すなわち，書換え規則の集合 $R=\{r_1, \ldots, r_n\}$ のメタ表現 $rules(R)$ を以下のように定義する。補助関数 $rule^{\wedge}$，$expr^{\wedge}$ は，それぞれ，書換え規則および式のメタ表現を求めるものである。また，rule，reflect は構成子である。

$rules^{\wedge}(\{r_1, \ldots, r_n\}) \equiv (rule(r_1) \ldots rule(r_n))$
$rule^{\wedge}(e \rightarrow e') \equiv \mathrm{rule}[expr^{\wedge}(e),\ expr^{\wedge}(e')]$
$expr^{\wedge}(t) \equiv term^{\wedge}(t),\quad \text{when } t \in T$
$expr^{\wedge}(t.c.r) \equiv \mathrm{reflect}[term^{\wedge}(t),\ term^{\wedge}(c),\ term^{\wedge}(r)]$
$rules^{\vee}(\mathrm{nil}) = \{\}$
$rules^{\vee}((r\ .\ rs)) = \{rule^{\vee}(r)\} \cup rules^{\vee}(rs)$
$rule^{\vee}(\mathrm{rule}[e, e']) = expr^{\vee}(e) \rightarrow expr^{\vee}(e')$
$expr^{\vee}(t) = term^{\vee}(t),\quad \text{when } t \not\equiv \mathrm{reflect}[\ldots, \ldots, \ldots]$
$expr^{\vee}(\mathrm{reflect}[t,c,r]) = term^{\vee}(t).\ term^{\vee}(c).\ term^{\vee}(r)$

4．応用および展望

自己反映計算の応用としては，一般によく議論されているように，OS やプログラミング環境あるいはプログラミング言語のユーザによるカスタマイズなどが典型的なものである。従来のアーキテクチャでは，これらの応用のためにはたとえば C 言語のような低レベルな言語を使用しなくてはならず，また，個々の既存システム毎にそのソースコードを読んでそれを改変する必要があった。自己反映機能は，これらの応用に必要なメタレベルへのアクセスを，高級言語による統一的かつ抽象度の高いインタフェースで実現し得る。

項書換えシステムの自己反映機能の応用についても同様なことがいえる。項書換えシステムは基本的には一階の等式論理に基づく論理体系である。しかし，すべてのプログラムが書換え規則の集合として記述され，それらがリダクションマシンで実行されている環境を想定しよう。このような状況では項書換えシステムは実用的なプログラミング言語としてその役割を果たさなければならない。すると当然，副作用などの「非論理的」な処理など一階の等式論理の枠外の計算を行なう必要があろう。そのような処理の一つ一つに対して処理系に例外的な機能を逐一付加するかわりに，自己反映機能はそのための統一的なインタフェースを提供しうる。

次に，項書換え分野特有の応用について考察しよう。Knuth と Bendix により考案された完備化手続きは，与えられた等式の集合から停止性と合流性を満たす項書換えシステムを生成しようとするものである。ここでは，書換え規則が動的に作られ，それらがその後の計算に使用される。すなわち，規則は生成される時点では計算対象であるデータとみなされるが，その後はプログラムの一部として用いられる。これは明らかに自己反映的な操作を含んでいる。これまでの通常の実現手法では，書換え規則はあくまでもデータとして扱われ，それを解釈するためのインタプリタが別に実現されている。しかし，先ほど述べたように，完備化手続き自体が項書換えシステムで記述され，それがリダクションマシンで実行されている環境を考えてみよう。この場合，インタプリタで書換えを実行するのは明らかに効率が悪い。生成された書換え規則が直接リダクションマシンで実行されるようにできれば効率が良い。これは自己反映機能により可能となる。これは書換え規則の動的なコンパイルともいえる。その点で，書換え規則を動的に Lisp にコンパイルする外山[20] のプログラムの成功は，この方法の有用性を示している。

項書換えシステムを拡張した計算モデルがいろいろ提案されている。外山[19]，山田[22]のメンバーシップ条件付き項書換えシステムにおいては，各書換え規則にメンバーシップ条件と呼ばれる条件を付けることができる。ただし，メンバーシップ条件は一般にメタレベルの情報を必要とするため書換え規則では記述できない。現在の実現では，メンバーシップ条件を判定するプログラムをユーザがLispで記述するようになっている。REPSにおいても条件付き書換え規則を記述できるように拡張し，自己反映機能を適切に設計すれば，メンバーシップ条件を書換え規則で記述できるようになるであろう。

それ以外の拡張モデルでメタ的な要素を含むものを幾つか挙げてみよう。馮ら[4]は，定理自動証明への応用を念頭に動的書換え計算モデルを提案している。Leler[11]はaugmented term rewriting system(ATRS)を提案し，それに基づいて制約プログラミング言語処理系を作成している。ATRSの特徴の一つは，大域的なスコープをもつ「変数」(項書換えシステムの構文規則の観点からすれば「定数記号」)に値を束縛できることである。aにbを束縛することはプログラムに動的に書換え規則a→bを付加することと等価であるとしてその意味が説明されている。Kirchnerら[8]は完備化手続きの発散を避ける一手法として，動的にメタ規則を生成することを提案している。以上のような種々の拡張のために処理系を一つ一つ作るかわりに，十分強力な自己反映機能をもつ処理系のカスタマイズによりそれを実現するアプローチが考えられる。

最後に今後の研究課題を展望する。まず，本論文のREPSの仕様は通常の項書換えシステムの簡約の定義をできるだけ自然に拡張して自己反映機能を実現しようとするアプローチで，いわば，ボトムアップ的なものである。明らかにそれとは逆のトップダウン的なアプローチによる研究が必要である。それには，すでに述べた様々な応用をうまく実現するためにはどのような自己反映機能があればよいかを考察すればよい。

また，REPSの効率良い処理系の作成も実用的な観点から重要である。メタオブジェクトとそれに対応する項（メタ表現）との間の変換が主要なオーバヘッドとなっている。この変換を高速化したり，遅延させたりして，オーバヘッドを小さくする必要がある。

REPSとは異なるアプローチによって項書換えシステムへ自己反映機能の導入を試みることも興味があろう。第1章に述べたように，「自己反映計算」という一般的な枠組みのもとでは，REPSのアプローチは間接的である。それに対し，直接的なアプローチにおいては処理系自体が書換え規則の集合として記述されるだろう。たとえば，矢野ら[23]はこのような記述を実際に作成しユーザに解放した処理系を実現している。このような自己記述と処理系が因果的結合を保ちながら計算が進行するようにすれば自己反映機能が得られる。ただし，このアプローチにおいては自己記述の詳細に依存したインタフェースを設計してしまうと処理系の抽象度を相当犠牲にしてしまう。しかし，このアプローチで成功すれば，簡約戦略などもユーザがカスタマイズできるなど，自己反映機能の有用性がさらに高まるであろう。

参考文献

1) Bawden, A.: Reification without evaluation, *Proc. of ACM Symp. on LISP and Functional Programming,* 1988, pp. 342-351.

2) Danvy, O. and Malmkjær, K.: Intensions and extensions in the reflective tower, *Proc. of ACM Symp. on LISP and Functional Programming,* 1988, pp. 327-341.

3) Dershowitz, N. and Jouannaud, J.-P.: Rewrite systems, In van Leeuwen J. (ed.): *Handbook of Theoretical Computer Science,* MIT Press, 1990, pp. 243-320.

4）馮　速，坂部俊樹，稲垣康善：動的項書換え計算モデルとその応用，信学技研，COMP91-47，1991，pp. 31-40.
5）二木厚吉，外山芳人：項書き換え型計算モデルとその応用，情報処理，Vol.31，No. 2，1983，pp. 133-146.
6）井田哲雄：書換えモデル，計算モデルの基礎理論，岩波書店，1991，pp. 224-296.
7）Jefferson, S. and Friedman, D. P.: A simple reflective interpreter, *Proc. of Intern. Workshop on New Models for Software Architecture : Reflection and Meta-Level Architecture,* 1992, pp. 48-58.
8）Kirchner, H. and Hermann, M.: Meta-rule synthesis from crossed rewrite systems, *Proc. of Intern. Workshop on Conditional and Typed Rewriting Systems, LNCS,* Vol. 516, 1990, pp. 143-154.
9）Klop, J. W.: Term rewriting systems, In Abramsky, S. et al. (ed.): *Handbook of Logic in Computer Science,* Oxford University Press, 1992, pp. 1-116.
10) Kurihara, M. and Ohuchi, A.: An algebraic specification and an object-oriented implementation of a reflective language, *Proc. of Intern. Workshop on New Models for Software Architecture : Reflection and Meta-Level Architecture,* 1992, pp. 137-142.
11) Leler, W.: *Constraint Programming Languages : Their Specification and Generation,* Addison-Wesley, 1988.
12) Maes, P.: Issues in computational reflection, In Maes, P. and Nardi, D. (ed.): *Meta-Level Architectures and Reflection,* North-Holland, 1988, pp. 21-35.
13) Matsuoka, S., Watanabe, T., Ichisugi, Y. and Yonezawa, A.: Object-oriented concurrent reflective architectures, *Proc, of ECOOP Workshop on Object-Based Concurrent Programming, LNCS,* Vol. 612, 1992.
14) Smith, B. C.: Reflection and semantics in Lisp, *Proc. of ACM Symp. on Principle of Programming Languages,* 1984, pp. 23-35.
15) 菅野博靖，田中二郎：メタ推論とリフレクション，情報処理，Vol. 30，No. 6，1989，pp. 694-705.
16) Sugano, H.: Concurrent logic language and reflective computation in distributed environments, *Proc. of Intern. Workshop on New Models for Software Architecture : Reflection and Meta-Level Architecture,* 1992, pp. 69-74.
17) 田中二郎：メタプログラミングとリフレクション，In 井田哲雄，田中二郎(編)：続 新しいプログラミング・パラダイム，共立出版，1990，pp. 215-234.
18) Tanaka, J.: An experimental reflective programming system written in GHC, *Journal of Information Processing,* Vol. 14, No. 1, 1991, pp. 75-84.
19) Toyama, Y.: Confluent term rewriting systems with membership conditions, *Proc. of Intern. Workshop on Conditional Term Rewriting Systems, LNCS,* Vol. 308, 1987, pp. 228-241.
20) Toyama, Y.: Fast Knuth-Bendix completion with a term rewriting system compiler, *Information Processing Letters,* Vol. 32, No. 6, 1989, pp. 325-328.
21) 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, pp. 111-134.
22) Yamada, J.: Confluence of terminating membership conditional TRS, *Proc. of Intern. Workshop on Conditional Term Rewriting Systems, LNCS,* Vol. 656, 1992, pp. 378-392.
23) 矢野博之，布川博士，富樫　敦，野口正一："EVAL"をユーザに解放した項書き換え系の実現，情報処理学会第34回全国大会講演論文集，1987，pp. 731-732.