プログラミング 6－11
（1996. 3. 27）

条件付き項書換え系に基づく言語におけるメタ計算

沼澤 政信　栗原 正仁　大内 東
numasawa@huie.hokudai.ac.jp

北海道大学工学部
〒060 札幌市北区北13条西8丁目

本稿では，条件付き項書換え系に基づく言語へのメタ計算機能の導入を提案し，その処理系の実現と応用について述べる．まず，書換え規則の条件評価中のメタ情報を格納する4つのスタックを持った仮想簡約マシンを導入し，このマシンの状態推移を形式化した簡約関係により，言語の操作的意味を定義する．次に，このマシンにメタ計算機能を付加することにより，言語を拡張する．メタ計算機能はメタ変換およびベース変換と呼ばれる2つの基本機能により定義されている．4つのスタックとプログラムをメタレベル・オブジェクト，構成子項をベースレベル・オブジェクトと定義しており，メタ変換は前者から後者への変換，ベース変換はその逆操作である．応用例として，メンバーシップ条件付き項書換え系の処理とエラーの際のデバッガとのインタフェースの例を示す．また，遅延評価をメタ変換に導入することによる効率改善について述べる．

Meta-computation in a Conditional Term Rewriting System-based Language

Masanobu NUMAZAWA　Masahito KURIHARA　Azuma OHUCHI

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

We introduce meta-computation mechanisms into a conditional term rewriting system-based language and discuss about their implementation and application. First, we introduce a virtual reduction machine equipped with four stacks for storing meta-information during condition evaluation. This machine is used to define the operational semantics of our language in terms of a reduction relation reflecting its state transition. Secondly, we extend the language by adding to our machine two basic meta-computation mechanisms called meta- and base-transformations. The meta-transformation transforms meta-level objects (stacks and programs) into base-level objects (constructor terms), while the base-transformation is just its inverse. The mechanisms are applied in two simple examples: specification of a membership-conditional term rewriting system and an interface with a debagger. Finally, we improve the efficiency of our system by employing a delayed evaluation technique for the meta-transformation.

1 はじめに

書換え系 [1] は，関数・論理型言語 [2]，等式論理の自動証明 [3] や抽象データ型に基づく仕様記述 [4] など，記号計算の研究分野における基礎理論として重要な役割を担っている．これは，このような分野において，計算の意味や性質の解析，プログラムの変換・検証といった技術を数学的に捉えられることから，より厳密な議論を展開することを可能としている．書換え系の中でも，広範囲にわたり応用され，しかも性質がよく研究されているものの一つが項書換え系 [5][6][7] である．これは，更に，様々な拡張も試みられており，条件付き項書換え系 [8]，メンバーシップ条件付き項書換え系 [9] などが提案されている．また，書換え系に基づくプログラミング言語やその処理系も製作されている [2][10].

さて，近年の急速な計算機の処理能力の向上に伴い，ソフトウェアに対する要求が多様化してきている．その要求に柔軟な対応を行うため，より高度で複雑なプログラムを簡潔に記述できるプログラミング言語の開発は必要不可欠であり，その有用性は高い．このための一つのアプローチとして，拡張性および動的適応性をもった柔軟な計算メカニズムを提供するメタプログラミング機能をもったプログラミング言語の提案が数多く行われている．中でも特に，計算システムが「自分自身」について感知したり自分自身を変更したりすることを含むような計算は自己反映計算（リフレクション）[11] と呼ばれ，関数型言語をベースとするもの [12]，論理型言語をベースとするもの [13]，オブジェクト指向言語をベースとするもの [14] などが提案されている．

以上のような状況を踏まえて，最近，書換え系をベースとした自己反映システムが提案された．渡部 [15] は抽象書換え系を用いて，自己反映計算に対する形式的な定義を与え，山岡ら [16] は自己反映的な抽象書換え系の具体例として，値呼びλ計算を取り上げ，その場合に得られるプログラミング言語の操作的意味論および処理系の実装を行っている．また，栗原ら [17] は項書換え系に自己反映計算機能を付加した言語的枠組みとして REPS（Reflective Equational Programming System）を提案している．

本稿では，REPS の基礎となる計算モデルを通常の項書換え系から条件付き項書換え系に拡張した言語とその処理系について述べる．ここで生じた問題点とその解決策の概略を以下に述べる．

REPS においては，自己反映機能により書換え規則の集合（プログラム）が計算中に動的に変化し得るので，リダクションマシンの状態を書換え対象である項 s とプログラム $\mathcal{R}$ の順序対 $(s, \mathcal{R})$ で定義し，その列 $(s_1, \mathcal{R}_1) \rightarrow (s_2, \mathcal{R}_2) \rightarrow \ldots$ で計算を表現している．しかし，この定義を条件付き書換え規則を扱う場合にもそのまま用いると，条件部の評価過程はマシンの状態に適切に反映されなくなる．これは条件評価中の自己反映機能の応用を制限してしまう．例えば，この機能を用いて計算中にエラーが生じた場合，デバッガを呼び出したいとしよう．しかし，条件評価中にエラーが生起した場合，その時のマシンの適切な状態記述をデバッガに渡すことはできない．

この問題点を解決するためには，マシンの状態を，条件部の評価中の情報を持ち得るように改める必要がある．そこで我々は，初めに，書換え規則の条件評価中のメタ情報を格納する 4 つのスタックを持った仮想簡約マシンを導入し，このマシンの状態推移を形式化した簡約関係により言語の操作的意味を定義した．これにより，条件部の評価過程がマシンの状態に適切に反映できるようになる．次に，このマシンにメタ計算機能を付加することにより問題点を解決し，REPS の機能を含んだより高機能の処理系を構築した．また，REPS における効率問題を遅延評価の技術により解決した．

2 CTRS 仮想簡約マシン

CTRS 仮想簡約マシンの計算状態を 4 つのスタック（**対象項 (Subject term) スタック** S，**候補規則 (canDidate rules) スタック** D，**文脈 (conteXt) スタック** X，**相 (Phase) スタック** P）とプログラム $\mathcal{R}$ の組 $(S, D, X, P, \mathcal{R})$ で定義する．ただし，本章の範囲ではメタ計算機能を扱わないので計算中にプログラム $\mathcal{R}$ が変化することはない．したがって，計算状態を (S, D, X, P) と略記する．任意の計算状態において 4 つのスタックの深さは等しい．表 1 に各スタックの要素のデータ型を示す．計算の初期状態においては各スタックの深さは 1 であり，表 1 に示す唯一の要素が積まれていると仮定する．ただし，$top(T)$ は任意のスタック T のトップ要素，$\{\}$ は空集合，$\Box$ は空文脈を表す．

表 1: 各スタックの要素の型と初期状態

スタック	S	D	X	P
データ型	項	ルールの集合	文脈	相
初期状態	初期項	$\{\}$	$\Box$	$fetch$

対象項スタック S には（再帰的な）条件付き書換えの各レベルでのリデックスの候補が積まれる．すなわち，底には初期項 s_0 の簡約項のある部分項 s（$s_0 \rightarrow^*_{\mathcal{R}} C[s]$ を満たす）が積まれ，更にその上には s を簡約するためのルール $\ell \rightarrow r \quad if \quad t_1 = t'_1, \ldots, t_n = t'_n \in \mathcal{R}$ の 1 つの条件式から作られる項 $u_0 = eq(t_i\theta, t'_i\theta)$ の簡約項の部分項 u（$u_0 \rightarrow^*_{\mathcal{R}} C[u]$ を満たす）が積まれる．ただし，θ は代入，eq は項数 2 の予め決められている特定の関数記号である．$top(S)$ 要素を**現対象項**と呼ぶ．

候補規則スタック D は，現対象項 s とプログラム $\mathcal{R}$ 間の（不完全な）パターン照合で得られるルール（以後，**候補規則**と呼ぶ）の集合 $fetch(s, \mathcal{R})$ を要素とする．ただし，次の制約を満たすものと仮定する．

$$\mathcal{R}_{mat}(s) \subseteq fetch(s, \mathcal{R}) \subseteq \mathcal{R}$$

$$\mathcal{R}_{\mathrm{mat}}(\mathrm{s}) = \{\ell \rightarrow \mathrm{r} \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \in \mathcal{R} \mid \exists\theta, \mathrm{s} \equiv \ell\theta\}$$

$\mathcal{R}_{mat}$ は，現対象項 s と照合する左辺を持つルールの集合である．$fetch(s, \mathcal{R})$ の具体的な定義は（処理系に依存して）$\mathcal{R}_{mat}$ を包含する任意の集合でよいが，効率を損なわない範囲でできるだけ小さな集合になるように設計する．例えば，何らかの効率の良い方法（例えばインデキシング）で判断して，明らかに照合しないルールを $\mathcal{R}$ から除去したものを $fetch(s, \mathcal{R})$ とする．すなわち，関数記号のみの照

合，関数記号と第一引数のみの照合などの部分的な照合で全体的照合の可能性を予め判断してルールを選択するということである．便宜上，$top(D) = \{\ \rho\ \ldots\}$に全順序を付け，その第一要素$\rho$を現候補規則と呼ぶ．現候補規則が$\rho$で，$\mathcal{R}' = top(D) - \{\ \rho\ \}$のとき，$top(D)$を$\{\ \rho\ \} \uplus \mathcal{R}'$と表示する．$\uplus$は互いに素な2つの集合の結び（直和）を表す．各要素の順序付けにより，$top(D)$のρと$\mathcal{R}'$への分解は一意であることに注意．

文脈スタックXの任意の深さの要素は，対象項スタックSの同じ深さにある要素（項）の周りの文脈$C[\,]$である．

相スタックPは，計算状態を制御するために，相（*phase*）という情報を持つ．相は*fetch*, *match*, *move*, *eq?*, *eval*, *replace*(θ), *halt*の7つがある．また，現対象項と候補規則の左辺との照合の結果得られる代入θを相*replace*の付加情報として持たせている．相スタックのトップの要素$top(P)$を現在相と呼ぶ．誤解の恐れがなければ単に相と呼ぶ．

仮想簡約マシンにおける計算は計算状態の集合上の簡約関係$\Rightarrow$によって定義される．計算状態は15種類の操作ステップにより推移する．

相*halt*は計算の停止時にのみ用いられ，他の6つの相は各操作ステップにより適当な相へ移行する．現在相が*move*の場合の操作ステップ（文脈移動（右へ）と文脈移動（上へ））は書換え戦略を最左最外戦略として定義している．

以下，順に各操作ステップによる状態推移を$\Rightarrow$に関しての公理として形式的に表示する．便宜上，$s \equiv top(S)$とする．また，リストを表す項$cons(x,\ y)$の略記表現として Lisp 言語の記法 $(x\ \ .\ \ y)$ を用いる．また，$(x\ \ y\ \ .\ \ z)$ は $(x\ \ .\ \ (y\ \ .\ \ z))$，$(x\ \ y\ \ \ldots\ \ z)$ は $(x\ .(y\ .(\ldots\ .(z\ .nil)\ldots)))$ の略記である．更に，スタックTに要素eをプッシュした結果得られるスタックをProlog言語のリスト記法を借りて$[e \mid T]$と表す．$[e',\ e \mid T]$は$[e' \mid\ [e \mid T]]$の略記である．また，ルールρの左辺および右辺をそれぞれ$lhs(\rho)$，$rhs(\rho)$で表す．＿はDon't Careを表す．

規則取り込み:

$(\ [s \mid S'],\ [\_ \mid D'],\ X,\ [fetch \mid P']\ )$

$\Rightarrow\ (\ [s \mid S'],\ [fetch(s, \mathcal{R}) \mid D'],\ X,\ [match \mid P']\ )$

不照合:

$(\ [s \mid S'],\ [\{\ \rho\ \} \uplus \mathcal{R}' \mid D'],\ X,\ [match \mid P']\ )$

$\Rightarrow\ (\ [s \mid S'],\ [\mathcal{R}' \mid D'],\ X,\ [match \mid P']\ )$

ただし，$\ell \equiv lhs(\rho) \in \mathcal{T},\ \not\exists \theta.s \equiv \ell\theta$.

照合成功（条件なし）:

$(\ [s \mid S'],\ [\{\ \ell \rightarrow r\ \} \uplus \mathcal{R}' \mid D'],\ X,\ [match \mid P']\ )$

$\Rightarrow\ (\ [s \mid S'],\ [\{\ \ell \rightarrow r\ \} \uplus \mathcal{R}' \mid D'],\ X,\ [replace(\theta) \mid P']\ )$

ただし，$s \equiv \ell\theta,\ \ell \in \mathcal{T}$.

照合成功（条件付き）:

$(\ [s \mid S'],\ [\{\ \rho\ \} \uplus \mathcal{R}' \mid D'],\ X,\ [match \mid P']\ )$

$\Rightarrow\ (\ [s',\ s \mid S'],\ [\{\ \},\ \{\ \rho\ \} \uplus \mathcal{R}' \mid D'],\ [C[\,] \mid X],$
$[fetch,\ replace(\theta) \mid P']\ )$

ただし，$s \equiv \ell\theta,\ \rho = \ell \rightarrow r\ if\ t_1 = t_1', \ldots, t_n = t_n',\ n > 0,\ \ell \in \mathcal{T},$
$s' \equiv eq(t_1\theta,\ t_1'\theta),\ C[\,] \equiv (\square\ eq(t_2\theta,\ t_2'\theta)\ \ldots\ eq(t_n\theta,\ t_n'\theta))$.

照合失敗（複合項）:

$(\ [F(s_1,\ s_2,\ \ldots,\ s_m) \mid S'],\ [\{\ \} \mid D'],\ [C[\,] \mid X'],\ [match \mid P']\ )$

$\Rightarrow\ (\ [s_1 \mid S'],\ [\{\ \} \mid D'],\ [C[F(\square,\ s_2,\ \ldots,\ s_m)] \mid X'],\ [fetch \mid P']\ )$

ただし，$m > 0$.

照合失敗（定数，変数）:

$(\ [s \mid S'],\ [\{\ \} \mid D'],\ X,\ [match \mid P']\ )$

$\Rightarrow\ (\ [s \mid S'],\ [\{\ \} \mid D'],\ X,\ [move \mid P']\ )$

ただし，$s \in \mathcal{C} \cup \mathcal{V}$.

置換:

$(\ [s \mid S'],\ [\{\ \rho\ \} \uplus \mathcal{R}' \mid D'],\ [C[\,] \mid X'],\ [replace(\theta) \mid P']\ )$

$\Rightarrow\ (\ [C[r\theta] \mid S'],\ [\{\ \} \mid D'],\ [\square \mid X'],\ [fetch \mid P']\ )$

ただし，$r \equiv rhs(\rho) \in \mathcal{T}$.

文脈移動（右へ）:

$(\ [s \mid S'],\ D,\ [C[F(\ldots,\ \square,\ t,\ \ldots)] \mid X'],\ [move \mid P']\ )$

$\Rightarrow\ (\ [t \mid S'],\ D,\ [C[F[\ldots,\ s,\ \square,\ldots]] \mid X'],\ [fetch \mid P']\ )$

文脈移動（上へ）:

$(\ [s \mid S'],\ D,\ [C[F(\ldots,\ \square)] \mid X'],\ [move \mid P']\ )$

$\Rightarrow\ (\ [F(\ldots,\ s) \mid S'],\ D,\ [C[\,] \mid X'],\ [eq? \mid P']\ )$

停止:

$(\ [s \mid S'],\ D,\ [\square \mid X'],\ [move \mid P']\ )$

$\Rightarrow\ (\ [s \mid S'],\ D,\ [\square \mid X'],\ [halt \mid P']\ )$

ただし，$s \not\equiv H(\ldots)$.

eq:

$(\ [eq(u,\ u') \mid S'],\ D,\ X,\ [eq? \mid P']\ )$

$\Rightarrow\ (\ [eq(u,\ u') \mid S'],\ D,\ X,\ [eval \mid P']\ )$

非eq:

$(\ [F(\ldots) \mid S'],\ D,\ X,\ [eq? \mid P']\ )$

$\Rightarrow\ (\ [F(\ldots) \mid S'],\ D,\ X,\ [move \mid P']\ )$

ただし，$F \neq eq$.

条件評価継続:

$(\ [eq(u,\ u) \mid S'],\ D,\ [(\ldots\ \square\ t\ \ldots) \mid X'],\ [eval \mid P']\ )$

$\Rightarrow\ (\ [t \mid S'],\ D,\ [(\ldots\ eq(u,u)\ \square\ \ldots) \mid X'],\ [fetch \mid P']\ )$

条件評価成功:

$(\ [eq(u,\ u) \mid S'],\ [\_ \mid D'],\ [(\ldots\ \square) \mid X'],\ [eval \mid P']\ )$

$\Rightarrow\ (\ S',\ D',\ X',\ P'\ )$

条件評価失敗:

$(\ [eq(u,\ u') \mid S'],\ [\_,\ \{\ \rho\ \} \uplus \mathcal{R}' \mid D''],\ [\_ \mid X'],\ [eval,\ \_ \mid P'']\ )$

$\Rightarrow\ (\ S',\ [\mathcal{R}' \mid D''],\ X',\ [match \mid P'']\ )$

ただし，$u \not\equiv u'$.

3 メタ計算機能の導入

3.1 メタ変換とベース変換

我々はメタ計算機能を実現するために，項，プログラムおよび4つのスタックをメタレベル・オブジェクトとして扱い，一方，構成子項をユーザプログラムがデータとしてア

クセスできるベースレベル・オブジェクトとして扱う．そして，前者から後者への変換およびその逆変換を基本機能として用意する．前者からの変換操作をメタ変換と呼ぶ．メタ変換で得られたデータを対応するメタレベル・オブジェクトのメタ表現と呼ぶ．REPS[17] における関数の定義にしたがって，$term^{\wedge}$および$rules^{\wedge}$をそれぞれ項およびプログラムをメタ変換する関数とし，$Sstack^{\wedge}$，$Dstack^{\wedge}$，$Xstack^{\wedge}$，$Pstack^{\wedge}$をそれぞれS，D，X，Pの各スタックをメタ変換する関数として新たに用意する．

メタ変換の逆変換をベース変換と呼ぶ．ベース変換は，項，プログラム，スタックのメタ表現が適切な場合に，それぞれを対応するメタレベル・オブジェクトに変換する．これもまたREPSの定義にしたがって，$term^{\vee}$および$rules^{\vee}$をそれぞれ項およびプログラムのメタ表現をベース変換する関数とし，$Sstack^{\vee}$，$Dstack^{\vee}$，$Xstack^{\vee}$，$Pstack^{\vee}$をそれぞれS，D，X，Pの各スタックのメタ表現をベース変換する関数として新たに用意する．

3.2 構文

一般のCTRSでは，ルールの左辺と右辺は共に項であるが，我々の言語では，項ではないメタ計算式を導入する．これは書換えの対象にならず，ルールの左辺では$t : L$ (ただし，$t \in \mathcal{T}$，$L \notin \mathcal{T}$)，右辺ではL (ただし，$L \notin \mathcal{T}$) としてのみ現れる表現であり，システムのメタ計算機能とのインタフェースの役割を果たす．項とメタ計算式を総称して式と呼ぶ．

メタ計算式におけるLは$R(s)$，$S(s)$，$D(s)$，$X(s)$，$P(s)$ (ただし，$s \in \mathcal{T}$) の形の項のみをそれぞれ高々1つ含む{ }で囲まれたリストである．ただし，R，S，D，X，Pはそれぞれプログラム$\mathcal{R}$とスタックS，D，X，Pの名前を表す組込みの関数記号である．以下では，リストLを集合とみて要素の所属判定に記号$\in$を用いる．例えば，$L = \{R(r)\ S(a)\}$のとき，$S(a) \in L$である．

ルールの形は，左辺と右辺がそれぞれ項であるかメタ計算式であるかによって，4つに分類できる．左辺がメタ計算式であるルール$t : L \rightarrow$ (式) if (条件) はメタ変換，右辺がメタ計算式であるルール (式) $\rightarrow L$ if (条件) はベース変換を引き起こす．

3.3 メタ計算機能を付加したCTRS仮想簡約マシン

この項では第2章で述べた仮想簡約マシンにメタ変換とベース変換の機能を付加し，簡約関係$\Rightarrow$を拡張する．

$\Rightarrow$は前述の15ステップに新たに5つの操作ステップを追加して定義される．以下，便宜上$s \equiv top(S)$とし，各操作ステップを示す．

以下の3つのどのステップも，現候補規則の左辺がメタ計算式$t : L$であるときに限り適用可能である．どのステップを実行するかは，対象項$s \equiv F(s_1, \ldots, s_m)$と左辺$t : L$のメタ照合の結果に依存する．メタ照合では，以下の条件を満たす代入θが存在するかどうかを確かめる．この条件をメタ照合条件と呼び，述語$Meta\text{-}Match(s, t, L, \theta)$で表す．ただし，この述語は，暗黙に，照合の際のマシンの状態$(S, D, X, P, \mathcal{R})$にも依存することに注意する．

$$
\begin{cases}
F(term^{\wedge}(s_1), \ldots, term^{\wedge}(s_m)) \equiv t\theta, \\
\text{もし } \mathrm{R}(u_1) \in L \text{ ならば, } rules^{\wedge}(\mathcal{R}) \equiv u_1\theta, \\
\text{もし } \mathrm{S}(u_2) \in L \text{ ならば, } Sstack^{\wedge}(S) \equiv u_2\theta, \\
\text{もし } \mathrm{D}(u_3) \in L \text{ ならば, } Dstack^{\wedge}(D) \equiv u_3\theta, \\
\text{もし } \mathrm{X}(u_4) \in L \text{ ならば, } Xstack^{\wedge}(X) \equiv u_4\theta, \\
\text{もし } \mathrm{P}(u_5) \in L \text{ ならば, } Pstack^{\wedge}(P) \equiv u_5\theta.
\end{cases}
$$

メタ不照合:

$$
\begin{aligned}
&([s \mid S'],\ [\{\rho\} \uplus \mathcal{R}' \mid D'],\ X,\ [match \mid P']) \\
&\quad \Rightarrow\ ([s \mid S'],\ [\mathcal{R}' \mid D'],\ X,\ [match \mid P'])
\end{aligned}
$$

ただし，$\nexists\theta.Meta\text{-}Match(s, t, L, \theta),\ t : L = lhs(\rho)$.

メタ照合成功 (条件なし)：

$$
\begin{aligned}
&([s \mid S'],\ [\{\rho\} \uplus \mathcal{R}' \mid D'],\ X,\ [match \mid P']) \\
&\quad \Rightarrow\ ([s \mid S'],\ [\{\rho\} \uplus \mathcal{R}' \mid D'],\ X,\ [replace(\theta) \mid P'])
\end{aligned}
$$

ただし，$\exists\theta.Meta\text{-}Match(s, t, L, \theta),\ t : L = lhs(\rho)$.

メタ照合成功 (条件付き)：

$$
\begin{aligned}
&([s \mid S'],\ [\{\rho\} \uplus \mathcal{R}' \mid D'],\ X,\ [match \mid P']) \\
&\quad \Rightarrow\ ([eq(t_1\theta, t_1'\theta),\ s \mid S'],\ [\{\},\ \{\rho\} \uplus \mathcal{R}' \mid D'],\ [(\Box\ eq(t_2\theta, t_2'\theta) \\
&\qquad \ldots\ eq(t_n\theta, t_n'\theta)) \mid X],\ [fetch,\ replace(\theta) \mid P'])
\end{aligned}
$$

ただし，$\rho = t : L \rightarrow e'\ if\ t_1 = t_1', \ldots, t_n = t_n',\ n > 0,$
$\exists\theta.Meta\text{-}Match(s, t, L, \theta)$.

次のステップは，現候補規則の右辺がメタ計算式L'のとき実行される．$L'\theta$はメタ計算式$L' = \{s_1 \ldots s_m\}$の各要素のインスタンスのリストを表す項$(s_1\theta\ \ldots\ s_m\theta)$を意味する．REPSの仕様と同様に，現在の文脈$top(X)$の情報は失われることに注意されたい．

メタ置換:

$$
\begin{aligned}
&([\_ \mid S'],\ [\{\rho\} \uplus \mathcal{R}' \mid D'],\ [\_ \mid X'],\ [replace(\theta) \mid P']) \\
&\quad \Rightarrow\ ([H(L'\theta) \mid S'],\ [\{\} \mid D'],\ [\Box \mid X'],\ [fetch \mid P'])
\end{aligned}
$$

ただし，$L' = rhs(\rho) \notin \mathcal{T}$.

次のステップは，関数記号Hを持つ項$H(L!)$が正規形である時に実行される．このステップのみがプログラム$\mathcal{R}$を変えることができるので，推論規則中の状態記述に$\mathcal{R}$を陽に加えた．R，S，D，X，Pはそれぞれプログラム$\mathcal{R}$とスタックS，D，X，Pの名前を表す組込みの関数記号である．

$$
\begin{cases}
\hat{\mathcal{R}} = & \text{もし } R(u_1) \in L! \text{ならば, } rules^{\vee}(u_1), \text{ それ以外は} \mathcal{R}. \\
\hat{S} = & \text{もし } S(u_2) \in L! \text{ならば, } Sstack^{\vee}(u_2), \text{ それ以外は } S. \\
\hat{D} = & \text{もし } D(u_3) \in L! \text{ならば, } Dstack^{\vee}(u_3), \text{ それ以外は } D. \\
\hat{X} = & \text{もし } X(u_4) \in L! \text{ならば, } Xstack^{\vee}(u_4), \text{ それ以外は } X. \\
\hat{P} = & \text{もし } P(u_5) \in L! \text{ならば, } Pstack^{\vee}(u_5), \text{ それ以外は } P.
\end{cases}
$$

ベース変換:

$$
(S,\ D,\ X,\ P,\ \mathcal{R}) \Rightarrow (\hat{S},\ \hat{D},\ \hat{X},\ \hat{P},\ \hat{\mathcal{R}})
$$

ただし，$top(S) \equiv H(L!),\ top(X) \equiv \Box,\ top(P) = move$.

3.4 メタ表現

これまでの議論は我々の言語の抽象的な枠組みのみを述べている．実際の処理系では構文とメタ表現を具体的に規定する必要がある．

基本的にはREPS[17]と同様である．変数は?マークを変数名の前につけて表記する．ルールの左辺と右辺を区切るには$\rightarrow$ではなく=を用いる．

3.4.1 項，プログラム，および文脈のメタ表現

関数 $term^{\wedge}$, $context^{\wedge}$及びその逆変換 $term^{\vee}$, $context^{\vee}$ は REPS における定義に従う．プログラム$\mathcal{R}$のメタ表現 $rules^{\wedge}(\mathcal{R})$ は，条件なしルールのための REPS の定義を基にして，適当な補助関数と構成子を用いて条件付きルールのための定義へと拡張する．

3.4.2 4つのスタックのメタ表現

各スタックのメタ変換の定義を以下に示す．

$Sstack^{\wedge}(S) \equiv (term^{\wedge}(s_n)\ \ldots\ term^{\wedge}(s_1))$
$Dstack^{\wedge}(D) \equiv (rules^{\wedge}(d_n)\ \ldots\ rules^{\wedge}(d_1))$
$Xstack^{\wedge}(X) \equiv (context^{\wedge}(c_n)\ \ldots\ context^{\wedge}(c_1))$
$Pstack^{\wedge}(P) \equiv (phase^{\wedge}(p_n)\ \ldots\ phase^{\wedge}(p_1))$

ここで，nは各々のスタックの深さであり，スタックのトップ要素（添字 n の要素）からボトム要素（添字 1 の要素）までをメタ変換したものがリストとして並べられる．関数 $phase^{\wedge}$は，$replace(\theta)$ 以外の相に対しては関数 $term^{\wedge}$における定数の変換と同様（変換後も不変）の変換を定義し，$replace(\theta)$ に対しては代入θのメタ表現をもつように定義する．ただし，代入θのメタ表現については，ここでは説明を省略する．

4 議論

4.1 応用

項書換え系に基づく言語におけるメタ計算の応用事例は文献 [17] で議論されている．本節では，本稿で導入した機能によって可能となった応用例を示す．メンバーシップ条件付き項書換え系（MCTRS）[9] においては，各ルール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，この条件は一般にメタレベルの情報を必要とするためルールでは仕様を記述できず，処理系では条件を評価するプログラムをユーザがシステム記述言語（例えば，Lisp）で記述するようになっていた．それに対して，メタレベルの情報を直接扱える我々の処理系においては，メンバーシップ条件を（我々の仮想簡約マシンでモデル化されているメタレベル情報のみを用いているならば）ルールで記述できる．

例として，$F = \{d,\ +,\ s,\ 0\}$, $F' = \{+,\ s,\ 0\}$ を関数記号の集合とし，次の MCTRS を考える [9].

$$\mathcal{R} = \begin{cases} x + 0 \to x \\ x + s(y) \to s(x + y) \\ d(x) \to x + x \qquad \textbf{if} \quad x \in \mathcal{T}(F') \end{cases}$$

この例を我々の言語で記述すると以下のようになる．

```
plus[?x,0] = ?x.
plus[?x,s[?y]] = s[plus[?x,?y]].
d[?x] = plus[?x,?x] if plusexpr[?x] = true.
plusexpr[?x]:{} = pexpr[?x].
pexpr[0] = true.
pexpr[(s ?x)] = pexpr[?x].
pexpr[(plus ?x ?y)] = true
     if pexpr[?x] = true, pexpr[?y] = true.
```

このプログラムでは，関数plusexpr においてメタ変換を行うことで，引数の項の構造を（メタレベルで）調べている．$T(F')$ の要素，つまりプログラムでは0, s, plus からなる項に限り，関数plusexpr によりtrue に簡約されるのである．

さて，別の応用例として，デバッガとのインタフェースを考える．我々の処理系は仮想簡約マシンによって計算状態を定義しているので，条件評価中であっても，適切な計算状態をメタ情報として得ることができる．プログラムにエラーが生じ，デバッガに現在のプログラムとスタック情報を渡すと仮定しよう．例えば，自然数上での除算を定義したプログラムに次の 2 本のルールを加える．

```
(1)  div[?n, 0]:{ S[?s] D[?d] X[?x] P[?p] R[?r] }
       = debug[?s, ?d, ?x, ?p, ?r] .
(2)  continue[?s, ?d, ?x, ?p, ?r]
       = { S[?s] D[?d] X[?x] P[?p] R[?r] } .
```

例えば，項div[s[s[0]], 0] がリデックスの候補として選ばれると，(1) によりメタ変換が行われる．これにより，?s, ?d, ?x, ?p, ?r にはそれぞれスタック S, D, X, Pおよびプログラム$\mathcal{R}$のメタ表現が代入される．関数debug は引数としてこれらの情報を受け取り，適当なデバッグ動作を行う．その後，ユーザとの対話等により，これらの内容が更新され，計算を再開するためにそれらを引数とする関数continue が呼び出されるものと仮定する．そうすると，ルール (2) により，メタ表現がベース変換され，ここで得られた新しい計算状態から計算を再開することができる．

4.2 効率化

メタレベル・オブジェクトとベースレベル・オブジェクトとの間の変換を単純に実現すると，それが主要なオーバーヘッドとなって効率を悪くしてしまう．我々の処理系ではその問題点を解決するために，メタ計算式の L の構文の定義の工夫と，遅延評価（*delayed evaluation*）を導入して効率化を図った．

メタ変換は，メタオブジェクトを構成子項に変換する．このとき，メタオブジェクトのサイズに比例してその変換時間は長くなり，後にそのデータにアクセスが全くない，もしくはその大部分にアクセスがないときなどには，その変換は無駄になってしまう．そこで前節では，そのデータにアクセスが全くない場合を考慮して，REPS のように全メタレベル・オブジェクトを必ずメタ変換するのではなく，メタ計算式の L の部分に変換したいメタレベル・オブジェクトを選択的に指定できるように構文を定義した．例えば，単に現対象項のみのメタ表現を得るためには，プログラムやスタックのメタ表現は必要ないので，メタ計算式を $t:\{\}$ と書けばよい．更に，プログラムおよび対象項スタックのメタ変換を行いたい時は，メタ計算式を $t:\{R(r)\ \ S(a)\}$ の形で記述すればよい．

次に，メタ変換後のデータの大部分にアクセスがない場合の無駄を防ぐために，変換要求があっても即座に変換せず，変換を遅延させる実現手法をとった．すなわち，メタレベル・オブジェクトから遅延オブジェクトと呼ばれるオブジェクトへの変換にとどめておき，その後，遅延オブジェ

クトの内部へのアクセス要求があった場合にのみ，それに応じて一部分をメタ表現に変換するという方法で無駄な変換を避ける．

本研究では，処理系をCLOS（Common Lisp Object System）を用いて構築しており，この方法はCLOSのクラス定義とメソッド定義によって実現した．

項f[a,b]をメタ変換することを考えよう．項f[a,b]はクラスApplicationのインスタンスとして実現されている．初めに，遅延評価をしない$term$^の実行結果は図1のようになる．$term$^によって，3つのApplicationオブジェクトで実現されるリストに（副作用的に）変更される．

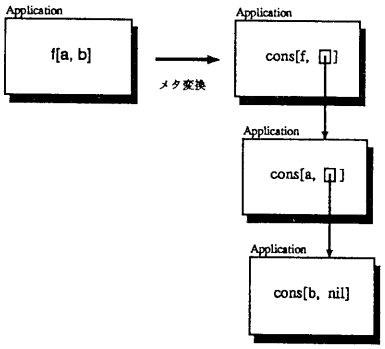

図 1: 遅延評価をしないメタ変換

これに対して，遅延評価を行う$term$^関数の実行結果を図2に示す．その結果，クラスはApplicationからDelayedに（CLOSのメソッド change-class により副作用的に）変更される．クラスDelayedのインスタンスはメタ表現の遅延オブジェクトである．この変換は項のサイズによらず一定時間でなされる．

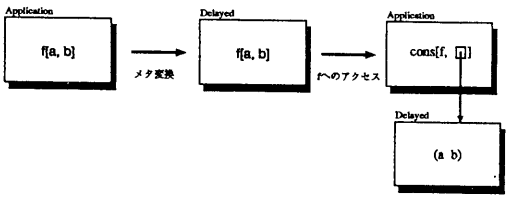

図 2: 遅延評価をするメタ変換

次に，この遅延オブジェクトの第1要素（関数記号f）へのアクセスを行うと，図2に示すような副作用的な変化が生ずる．これは構成子fへのアクセスと同時に，部分的なメタ変換が行われたことを意味する．引数部分の(a b)においては，依然，アクセス要求がないのでメタ変換は遅延されている．

5 おわりに

本稿では，条件付き項書換え系（CTRS）に基づく言語にメタ計算機能を導入し，その処理系を実現した．CTRSの条件評価を考慮して，4つのスタックをもつ仮想簡約マシンに基づいて計算状態を定義し，条件評価中の計算状態のメタ情報を得ることを可能とした．応用例として，メンバーシップ条件付き項書換え系の処理，およびデバッガとのインタフェースを示した．また，遅延評価の導入による効率向上について述べた．

最後に今後の課題を述べる．本稿の仮想簡約マシンは，簡約戦略を最左最外戦略に固定している．対象項に対してどの部分項を簡約するかということは，正規化戦略（正規形が存在するならば必ず求める）あるいはより効率のよい処理系を実現するためには不可欠な問題である．したがって，メタ計算機能により，簡約戦略を動的に変更し得るような仮想簡約マシンの構築が興味深い課題の一つとなる．

参考文献

[1] Dershowitz, N. and Jouannaud, J. P.: 書換え系, コンピュータ基礎理論ハンドブック II, 丸善, pp. 243–321 (1994).

[2] 佐々木重雄, 井田哲雄: 制約解消系を備えた関数・論理型言語の処理系とその実装, 情報処理学会論文誌, Vol. 36, No. 9, pp. 2152–2160 (1995).

[3] Plaisted, D. A.: Equational Reasoning and Term Rewriting Systems, *Handbook of Logic in Artificial Intelligence and Logic Programming* (Gabbay, D.(ed.)), Vol. 1, Oxford Science Publications, pp. 274–367 (1993).

[4] 稲垣康善, 坂部俊樹: 抽象データタイプの代数的仕様記述の基礎 (1), 情報処理, Vol. 25, No. 1, pp. 47–53 (1984).

[5] Klop, J. W.: Term Rewriting Systems, *Handbook of Logic in Computer Science* (Abramsky, S.(ed.)), Oxford University Press, pp. 1–116 (1992).

[6] 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. 24, No. 2, pp. 133–146 (1983).

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

[8] Bergstra, J. A. and Klop, J. W.: Conditional Rewrite Rules: Confluence and Termination, *Journal of Computer and System Sciences*, Vol. 32, No. 3, pp. 323–362 (1986).

[9] Toyama, Y.: Confluent Term Rewriting Systems with Membership Conditions, *Proc. of Intern. Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science, Vol. 308, pp. 228–241 (1987).

[10] O'Donnell, M. J.: *Equational Logic as a Programming Language*, The MIT Press (1985).

[11] Maes, P.: Issues in Computational Reflection, *Meta-Level Architectures and Reflection* (Maes, P. and Nardi, D.(eds.)), North-Holland, pp. 21–35 (1988).

[12] Smith, B. C.: Reflection and Semantics in Lisp, *Proc. of ACM Symp. on Principle of Programming Languages*, pp. 23–35 (1984).

[13] Tanaka, J.: An Experimental Reflective Programming System written in GHC, *Journal of Information Processing*, Vol. 14, No. 1, pp. 75–84 (1991).

[14] Yonezawa, A. and Watanabe, T.: An Introduction to Object-based Reflective Concurrent Computations, *Proc. of ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, ACM SIGPLAN Notices, Vol. 24, pp. 50–54 (1989).

[15] 渡部貞雄: 書換えに基づく自己反映計算の定式化, 信学技法, SS94-12, pp. 1–7 (1994).

[16] 山岡順一, 渡部貞雄: 自己反映的な値呼びλ計算の操作的意味論, 信学技法, SS94-52, pp. 49–56 (1995).

[17] 栗原正仁, 佐藤崇昭, 大内東: 項書換えシステムにおける自己反映計算, コンピュータソフトウェア, Vol. 12, No. 4, pp. 3–14 (1995).