プログラミング-言語・基礎・実践- 21－7
(1995. 3. 9)

自己反映的な値呼びλ計算の操作的意味論

山岡 順一　渡部 卓雄

北陸先端科学技術大学院大学
情報科学研究科

〒923-12 石川県能美郡辰口町旭台 15
TEL: 0761-51-1256, FAX: 0761-51-1349
E-mail: {jun,takuo}@jaist.ac.jp

本稿では、値呼びλ計算にもとづく自己反映的なプログラミング言語の操作的意味論、及びその実例となるプログラミング言語の実装について述べる。ここで示す意味論は、きわめて単純な規則によって表されており、自己反映計算に関する種々の考察の基盤となることが期待される。

キーワード: 自己反映計算, 抽象書換え系, 値呼びλ計算

An Operational Semantics of a Reflective Language based on λ_v-calculus

Junichi YAMAOKA　Takuo WATANABE

School of Information Science,
Japan Advanced Institute of Science and Technology

15 Asahidai, Tatsunokuchi, Ishikawa 923-12, Japan
TEL: +81-761-51-1256, FAX: +81-761-51-1349
E-mail: {jun,takuo}@jaist.ac.jp

In this paper, we propose an operational semantics of an applicable reflective programming language; the language is a reflective extension of the λ_v-calculus. As the semantics is simple and clear, it is expected to be a basis of theoretical/practical aspects of reflection. Moreover, we have implemented the language based on the semantics.

Keywords: Reflection, Abstract Rewriting, λ_v-calculus

1 はじめに

計算システムが自己の構成や計算過程に関する計算を行なうことを**自己反映計算**と呼び、このような機能をもつ計算システムを**自己反映的なシステム**と呼ぶ。

近年、自己反映計算はソフトウェアの構成原理として注目されてきており、自己反映的なプログラミング言語やOSなどの実現が進んでいる。これは、拡張性や動的適応性などといった自己反映的なシステムの優れた特性が認識されているためである[5]。

しかし、現在までのところ、自己反映計算に関する理論的な研究は特定の計算モデルや言語に依存したものが多く、自己反映計算の一般的な定義や意味論は与えられていない。このため、自己反映計算の種々の性質は、今一つ不明確な部分が多い。また、同様の理由により、自己反映的なプログラミング言語において、プログラム変換や正当性の検証などに関する形式的な議論を行なうことができないという問題がある。

本研究では、このような問題の解決へのアプローチとして、一般的な計算システムのモデルである抽象書換え系を用いて自己反映的なシステムをモデル化した。さらに、この抽象書換え系の具体例として、値呼びλ計算を適用した場合に得られるプログラミング言語の操作的意味論を与え、それにもとづくプログラミング言語処理系を実装した。

以下では、抽象書換え系を用いた自己反映的なシステムのモデル化、及び自己反映的な値呼びλ計算の意味論について順に述べた後、最後に、この意味論にもとづいて実装されたプログラミング言語Lambda/Rについて述べる。

2 抽象書換え系によるリフレクティブタワーのモデル化

自己反映計算を実現する枠組としては、B. C. Smithによって提案された手続き的リフレクション[4]が一般的である。手続き的リフレクションにおいて、自己反映的なシステムは、無限に重なったインタプリタの階層としてモデル化される。このインタプリタの階層を**リフレクティブタワー**と呼ぶ。リフレクティブタワーを構成する各インタプリタは、一段上のインタプリタによって解釈/実行されるプログラムであり、ユーザレベルプログラムはタワーの最下層のインタプリタによって解釈/実行されると考えられる。この最下層のインタプリタをベースレベルのシステムと呼び、ベースレベルのシステムを解釈/実行しているインタプリタをメタレベルのシステムと呼ぶ。本稿では、便宜上、下から n 段目のインタプリタをメタ^{n}レベルのシステムと呼ぶことにする。

本節では、リフレクティブタワーを構成する各インタプリタを抽象書換え系としてモデル化する。抽象書換え系は、計算を書換えとして一般化したものであり、計算システムの一般的な性質を考察する際に有用なモデルである[1]。

最初に、以降で用いる関係$\triangleright$の定義を行なう。

定義 2.1 完備半順序集合(cpo) A, B 間に連続関数 $f : A \to B$ 及び $g : B \to A$ が存在し、かつ次式が成立するとき、A は (f, g) によって B に埋め込まれるといい、これを $B \triangleright A$ と記す。

$$g \circ f = id_A$$

$$f \circ g \sqsubseteq id_B$$

□

有限、あるいは可算無限の集合 L、及び L 上の書換え規則 $\xrightarrow{L}$ からなる抽象書換えシステム $\langle L, \xrightarrow{L} \rangle$ において、集合 L は、次のように書換え規則を用いて順序を設定すると完備半順序集合になる。但し、$\xrightarrow{L*}$ は $\xrightarrow{L}$ の反射推移閉包であるとする。

$$\forall t_0, t_1 \in L.\, (t_0 \sqsubseteq t_1) \Leftrightarrow (t_0 \xrightarrow{L*} t_1 \vee t_1 \xrightarrow{L*} t_0)$$

以下では、上記の関係を用いて、抽象書換え系にもとづくリフレクティブタワーのモデル化を行なう。但し、あくまでもモデル化の方針を示すことを目的とし、細かい証明は省略する。

リフレクティブタワーのモデル化に際して、抽象書換え系の拡張という概念を次のように定義する。

定義 2.2 抽象書換え系 $\langle L, \xrightarrow{L} \rangle, \langle \bar{L}, \xrightarrow{\bar{L}} \rangle$ において次の式が満たされる場合、$\langle \bar{L}, \xrightarrow{\bar{L}} \rangle$ は $\langle L, \xrightarrow{L} \rangle$ の拡張であるという。

$$\forall t_0, t_1 \in L.$$
$$(t_0 \in \bar{L}) \wedge (t_1 \in \bar{L}) \wedge (t_0 \xrightarrow{L} t_1 \Rightarrow t_0 \xrightarrow{\bar{L}} t_1)$$

□

ここで、ベースレベルの抽象書換え系を $\langle L_0, \xrightarrow{L_0} \rangle$ と定める。拡張によってベースレベルで利用可能となる項の集合を $\bar{L}_0$ とすると、ベースレベルの厳密なモデルを内包するメタレベルのシステム $\langle L_1, \xrightarrow{L_1} \rangle$ は、次の式を満たすことになる。

$$L_1 \triangleright \bar{L}_0 \times 2^{\bar{L}_0 \times L_0}$$

上式の右辺において、$\bar{L}_0$ はベースレベルの項の集合、$2^{\bar{L}_0 \times L_0}$ はベースレベルの書換え規則の集合を表している。同様に、拡張によってメタ$^{n-1}$レベルで利用可能となる項の集合を $\bar{L}_{n-1}$ とすると、メタ$^{n-1}$レベルのシス

テムの厳密なモデルを内包する メタ^n レベルのシステム $\langle L_n, \overset{L_n}{\rightarrow}\rangle$ は次の式を満たす。

$$L_n \rhd \bar{L}_{n-1} \times 2^{\bar{L}_{n-1}\times\bar{L}_{n-1}} \quad (n=1,2,\ldots)$$

ここで注意しなければならないのは、無限の階層をもつリフレクティブタワーを実現するためには、すべてのレベルのシステムをベースレベルのシステムによって実現する必要があるということである。つまり、自然数の集合を $\mathbf{N}$ とすると、$\forall n \in \mathbf{N}.\ L_n = \bar{L}_n = L_0$ という式が満たされなければならないのである。したがって、自己の厳密なモデルを内包する自己反映的な抽象書換え系においては、次の式が成り立つことになる。

$$L_0 \rhd L_0 \times 2^{L_0\times L_0}$$

しかし、実際には、L_0 と $2^{L_0\times L_0}$ は濃度が異なるため、上式は明らかに成立しない。これにより、可算無限の項集合をもち、自己の厳密なモデルを内包する抽象書換えシステムは存在しないことがわかる。

以上で示したように、無限の階層をもつリフレクティブタワーは、メタ^n レベルのシステムが メタ$^{n-1}$ レベルのシステムの厳密なモデルを内包するように構成することはできない。但し、メタ^n レベルのシステムが メタ$^{n-1}$ レベルのシステムの変更部分のモデルを内包するようなリフレクティブタワーは、次のように構成することができる。

$$L_n \rhd L_0 \times A \quad (n=1,2,\ldots)$$
$$A \subset 2^{L_0\times L_0}$$

A は、各レベルの書換え規則の変更可能範囲を示す集合であり、加算個の要素をもつ。つまり、書換え規則の変更部分を $r \in A$ とした場合、項 t_0, t_1 について $\langle t_0, t_1\rangle \in r$ ならば、$t_0 \overset{L_0}{\rightarrow} t_1$ という書換え規則が存在するものと考える。通常、リフレクションによる計算システムの機能変更は漸進的に行なわれるため、このようにメタ^n レベルのシステムがメタ$^{n-1}$ レベルのシステムの変更部分のモデルをもつことは、合理的であると言える。また、リフレクティブタワー全体において、これらの変更可能範囲をまとめた集合 M を次のように定める。

$$M = \coprod_{n\in\mathbf{N}} A$$

ここで、簡約が実行されるレベルを $n \in \mathbf{N}$、書換え規則の変更部分をリフレクティブタワー全体について集めたものを $R \in M$、簡約対象となる項を $t \in L_0$ とすると、上記のように構成されるリフレクティブタワーの状態は、次のような三つ組によって表すことができる。

$$\langle n, R, t\rangle$$

本節では、このような状態をもつリフレクティブタワーの (構造的) 操作的意味論を図1のよう定める。但し、メタ$^{n-1}$ レベルの書換え規則の変更部分を $r_n \in A$ とし、$\overset{L'_n}{\rightarrow} = \overset{L_n}{\rightarrow} \cup r_n$ とする。また、$L_n \times A_n$ を L_{n+1} に埋め込む関数を (Φ_+, Φ_-) とし、項 $t_0, t_1 \in L$ 間に書換え規則 $\overset{L}{\rightarrow}$ が存在しないことを $t_0 \overset{L}{\not\rightarrow} t_1$ と書くことにする。

また、図1において、$R[r'_n/r_n]$ は、r_n を r'_n で置き換えることを意味する。このようにして構成されるリフレクティブタワーの概念図を図2に示す。

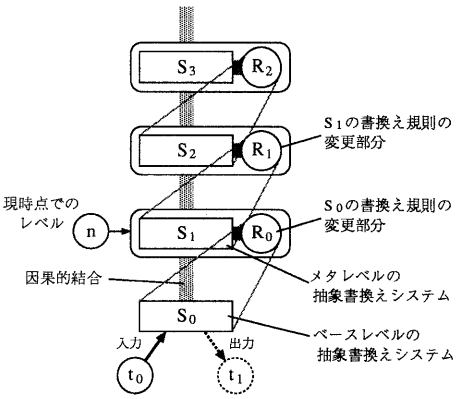

図 2: 自己反映的な抽象書換え系のモデル

3 自己反映的なλ_v計算の意味論

本節では、前節に示したリフレクティブタワーの意味論に沿って、リフレクティブなプログラミング言語を定義する。このプログラミング言語は、図2における各レベルの抽象書換え系に値呼びλ計算を適用したものである。λ計算によって扱われる項 (λ項) は、すべての計算可能な関数を表現できることが証明されているため、ここで与えるプログラミング言語はすべての計算可能な関数を記述できることになる。以下では、その構文と意味論について、順に説明する。

3.1 構文

本節で定義するプログラミング言語の抽象構文を以下に示す。

$x \in X$	変数
$b \in B$	定数 (δ項)
$f \in F$	組み込み関数 (δ項)
$v \in V$	値
$e \in E$	項
$C \in Con$	文脈

$$\frac{t_0 \xrightarrow{L_n} t_1 \quad \forall r'_n, t_2.\ \Phi_+(t_0, r_n) \overset{L'_{n+1}}{\not\rightarrow} \Phi_+(t_2, r'_n)}{\langle n, R, t_0\rangle \rightarrow \langle n, R, t_1\rangle} \tag{1}$$

$$\frac{\Phi_+(t_0, r_n) \overset{L'_{n+1}}{\rightarrow} \Phi_+(t_1, r'_n) \quad \forall r'_n, r''_n, t_2.\ \Phi_+(\Phi_+(t_1, r'_n), r_{n+1}) \overset{L'_{n+2}}{\not\rightarrow} \Phi_+(\Phi_+(t_2, r'_n), r'_{n+1})}{\langle n, R, t_0\rangle \rightarrow \langle n, R[r'_n/r_n], t_1\rangle} \tag{2}$$

$$\frac{\exists r'_n, t_1.\ \Phi_+(t_0, r_n) \overset{L'_{n+1}}{\rightarrow} \Phi_+(t_1, r'_n) \quad \langle n+1, R, t_1\rangle \rightarrow \langle n+1, R', t_2\rangle}{\langle n, R, t_0\rangle \rightarrow \langle n, R', t_2\rangle} \tag{3}$$

図 1: 抽象書換えにもとづくリフレクティブタワーの意味論

$$v ::= x \mid b \mid f \mid \lambda x.e \mid \uparrow_e \mid \downarrow_e \mid \Uparrow \mid \Downarrow$$
$$e ::= v \mid (e\ e)$$
$$C ::= [\,] \mid (v\ C) \mid (C\ e)$$

ここで、$\uparrow_e, \downarrow_e, \Uparrow, \Downarrow$ を除くと、δ簡約の概念を導入した通常のλ計算の構文と一致する。これらのプログラム構成要素の正確な意味は3.2項の意味論によって示されるが、直観的には、次のような意味をもつと考えられる。

$\uparrow_e$	項の表現を作る演算子
$\downarrow_e$	項の表現から実体を取り出す演算子
$\Uparrow$	メタレベル実行をおこす演算子
$\Downarrow$	メタレベル実行から復帰する演算子

次節以降においては、上記の構文要素を添字つきで表す場合がある。例えば、e, e_0, e_1 などは項を表し、C, C_0, C_1 などは文脈を表すものとする。

3.2 意味論

まず、ベースレベルの計算に用いられる簡約規則を次のように定める。但し、β簡約の capture avoidance は暗黙のうちに行なわれるものとする。また、$\Delta : F \times E \rightarrow E_\perp$ は組み込み関数の定義を与える関数であり、$\Phi_{e-}, \Phi_{e+} : E_\perp \rightarrow E_\perp$ は項の表現方法を与える関数であるとする。

$$\forall e_0, e_1.\ C[(\lambda x.e_0)\ e_1] \xrightarrow{\beta} C[e_0\{x := e_1\}] \quad \beta\text{簡約}$$
$$\forall f, e.\ C[f\ e] \xrightarrow{\delta} C[\Delta(f, e)] \quad \delta\text{簡約}$$
$$\forall e.\ C[\uparrow_e\ e] \xrightarrow{\phi} C[\Phi_{e+}(e)]$$
$$\forall e.\ C[\downarrow_e\ e] \xrightarrow{\phi} C[\Phi_{e-}(e)]$$

ここで、項が文脈を用いて表現されているのは、可簡約項を定めるためである。3.1項における文脈の構文定義によると、上記の簡約規則は値呼び簡約になる。しかし、通常のλ計算とは異なり、リフレクティブに拡張したλ 計算においては、文脈の構文を定めただけでは可簡約項を一意に決定することはできない。なぜなら、リフレクティブなプログラミング言語においては、プログラム構成要素の意味が変化する可能性があるため、項の中の値 v も評価しなければならないからである。したがって、文脈の構文とは別に、可簡約項を定めるための戦略を与える必要がある。ここでは、$C[e]$ という記述は単に項を表すものではなく、項の状態$\langle C, e\rangle : Con \times E$ を表すものとし、簡約規則は $(Con \times E) \rightarrow (Con \times E)$ という型をもつものとする。さらに、この項の状態を操作する関数を以下のように与える。但し、これらの関数は、上から順に引数のパターンマッチングを行なうことによって適用される式が決定されるものとする。

$$\begin{array}{lcl}
mc & : & E \rightarrow (Con \times E) \\
mc(v) & = & \langle [\,], v\rangle \\
mc(e_0\ e_1) & = & let\ \langle C, e_2\rangle = mc(e_0)\ in\ \langle (C\ e_1), e_2\rangle \\
rc & : & (Con \times E) \rightarrow (Con \times E) \\
rc((v_0\ [\,]), v_1) & = & \langle [\,], (v_0\ v_1)\rangle \\
rc((v_0\ C_0), v_1) & = & let\ \langle C_1, e\rangle = rc\ \langle C_0, v_1\rangle\ in \\
 & & \langle (v_0\ C_1), e\rangle \\
rc(([\,]\ e), \uparrow_e) & = & \langle [\,], (\uparrow_e\ e)\rangle \\
rc(([\,]\ e), \Uparrow) & = & \langle [\,], (\Uparrow\ e)\rangle \\
rc([\,]\ e_0), v) & = & let\ \langle C, e_1\rangle = mc(e_0)\ in \\
 & & \langle (v\ C), e_1\rangle \\
rc((C_0\ e_0), v) & = & let\ \langle C_1, e_1\rangle = rc(C_0, v)\ in \\
 & & \langle (C_1\ e_0), e_1\rangle \\
rc(C_0, e_0) & = & let\ \langle C_1, e_1\rangle = mc(e_0)\ in \\
 & & \langle rh(C_0, C_1), e_1\rangle \\
rh & : & (Con \times Con) \rightarrow Con \\
rh([\,], C) & = & C \\
rh((v\ C_0), C_1) & = & (v\ rh(C_0, C_1)) \\
rh((C_0\ e), C_1) & = & (rh(C_0, C_1)\ e)
\end{array}$$

上記において、c は項から項の状態を作り出す関数、rc は項の状態の遷移を行なう関数、rh は rc の補助関数である。

これらの関数を用いることにより、λ計算にもとづくリフレクティブなプログラミング言語の操作的意味論は、図3 のように書くことができる。但し、簡単化のために、ベースレベルの簡約規則をまとめて $\xrightarrow{b} = \xrightarrow{\beta} \cup \xrightarrow{\delta} \cup \xrightarrow{\phi}$ とする。また、各レベルの簡約規則の変更点 r_n による簡約を $\xrightarrow{r_n}$ によって表す。

また、入力される項を e_{in} とすると、リフレクティブタワー全体の初期状態 $\langle n_I, R_I, C_I[e_I]\rangle$ は次のようになる。

$$n_I = 0$$
$$R_I = \langle \emptyset, \emptyset, \emptyset, \ldots\rangle$$
$$C_I[e_I] = mc(e_{in})$$

上記の意味論において、リフレクティブタワーの各レベルの状態は、簡約を行なっているレベル n、各レベルの簡約規則の変更部分 r_n、及び項の状態 e からなる三つ組 $\langle n, r_n, e\rangle$ として表されている。ここで、レベルを示す数 n は、プログラムから簡約が行なわれているレベル

$$\frac{C \neq [\,] \quad \forall e_1.\, e_0 \overset{b}{\not\rightarrow} e_1 \quad \forall e_2.\, C[\Phi_{e+}(e_0)] \overset{r_n}{\not\rightarrow} \Phi_{e+}(e_2)}{\langle n, R, C[e_0]\rangle \rightarrow \langle n, R, rc(C[e_0])\rangle} \qquad \text{可簡約項の再設定}$$

$$\frac{e_0 \overset{b}{\rightarrow} e_1 \quad \forall e_2.\, C[\Phi_{e+}(e_0)] \overset{r_n}{\not\rightarrow} \Phi_{e+}(e_2)}{\langle n, R, C[e_0]\rangle \rightarrow \langle n, R, rc(C[e_1])\rangle} \qquad \text{ベースレベル簡約}$$

$$\frac{C[\Phi_{e+}(e_0)] \overset{r_n}{\rightarrow} \Phi_{e+}(e_1) \quad C_1[e_2] = mc(e_1) \quad \forall e_3.\, C_1[\Phi_{e+}(e_2)] \overset{r_{n+1}}{\not\rightarrow} \Phi_{e+}(e_3)}{\langle n, R, C[e_0]\rangle \rightarrow \langle n, R, mc(e_1)\rangle} \qquad \text{メタレベル簡約 1}$$

$$\frac{C[\Phi_{e+}(e_0)] \overset{r_n}{\rightarrow} \Phi_{e+}(e_1) \quad \langle n+1, R, mc(e_1)\rangle \rightarrow \langle n+1, R, e_2\rangle}{\langle n, R, C[e_0]\rangle \rightarrow \langle n, R, mc(e_2)\rangle} \qquad \text{メタレベル簡約 2}$$

$$\frac{e_1 = \Phi_{s+}(n, r_n, C[e_0])}{\langle n, R, C[\Uparrow\ e_0]\rangle \rightarrow \langle n+1, R, C[e_0\ e_1]\rangle} \qquad \text{レイフィケーション}$$

$$\frac{\exists m.\, e_0 = \Phi_{s+}(m, r'_{n-1}, mc(e_1))}{\langle n, R, C[\Downarrow\ e_0]\rangle \rightarrow \langle n-1, R[r'_{n-1}/r_{n-1}], mc(e_1)\rangle} \qquad \text{リフレクション}$$

図 3: 自己反映的な値呼びλ計算の操作的意味論

を知るために、便宜上、つけ加えたものである。このメタ$^{n-1}$レベルの状態は、(Φ_{s+}, Φ_{s-}) という二つの関数によって、メタ^{n}レベルの項に埋め込まれていると考えられる。すなわち、(Φ_{s+}, Φ_{s-}) によって次の式が成り立つということである。

$$E \triangleright N \times A \times E$$
$$A \subset 2^{(Con \times E) \times (Con \times E)}$$

以上において、$\Phi_{s+}, \Phi_{s-}, \Phi_{e-}, \Phi_{e+}$については具体的な意味を与えていないが、これらの関数については様々な実現方法が考えられる。次節では、これらの具体的な定義を与えたプログラミング言語について述べる。

4 プログラミング言語 Lambda/R

本節では、前節で示した意味論の実例となるプログラミング言語 Lambda/R について述べる。Lambda/R は、3節の意味論の実現であり、そのインタプリタは関数型プログラミング言語 Gofer によって実装されている。以下では、3.1項に示した構文と Lambda/R の構文の対応、Lambda/R における関数$\Phi_{e+}, \Phi_{e-}, \Phi_{s+}, \Phi_{s-}$などの定義、及びリフレクションを用いた簡単なプログラム例を示す。

4.1 Lambda/R の構文

Lambda/R の抽象構文は、以下の通りである。但し、変数、シンボル、組み込み関数の具体的な構文定義については、プログラミング言語の本質とは関係がないので省略する。

3.1項の記号	Lambda/R の記号
λ	!
$\uparrow_e$	!>
$\downarrow_e$	!<
$\Uparrow$	!+
$\Downarrow$	!-

表 1: Lambda/R の記号

$x \in X$ 変数
$s \in Sym$ シンボル
$n \in Int$ 整数
$b \in B$ 定数
$f \in F$ 組み込み関数
$v \in V$ 値
$e \in E$ 項
$C \in Con$ 文脈

$b ::= s \mid n \mid ?$
$v ::= x \mid b \mid f \mid !x.e \mid !> \mid !< \mid !+ \mid !-$
$e ::= v \mid (e\ e)$
$C ::= [\,] \mid (v\ C) \mid (C\ e)$

上記において、シンボル s は関数記号などに用いる定数である。また、定数 ? は簡約規則の記述に用いられる特殊記号である。その他の記号については、3.1項において使用した表記と Lambda/R における表記の対応表を表1に示す。

4.2 項の表現

本節では、Lambda/R における関数Φ_{e+}, Φ_{e-}の定義について述べる。

先に述べたように、Φ_{e+}はメタ$^{n-1}$レベルの項の表現となるメタ^{n}レベルの項を作り出す役割をもち、Φ_{e-}はその逆の働きをする。Lambda/R において、Φ_{e+}は !> に対応するので、項 e の表現は (!> e) という式によって作り出すことができる。ここで注意すべきことは、(!> e) の結果は正規形の項でなければならないということである。なぜなら、項 e の表現は、プログラムではなくデータを意味しており、プログラムに対する簡約規則が適用されてはならないからである。もし、(!> e) の結果が正規形でないとすると、e の表現が簡約されてしまうため、タイミングによっては元の e を正確に復元することができなくなる。

Lambda/R では、(!> e) は、必ず正規形の項になるように実装されている。すなわち、$\forall e_0, e_1.\ \Phi_{e+}(e_0) \overset{b}{\not\rightarrow} e_1$ という式が成り立つということである。具体的なΦ_{e+}, Φ_{e-}の定義は、次の通りである。

$$
\begin{array}{lcl}
\Phi_{e+} & : & E_\perp \to E_\perp \\
\Phi_{e+}(\perp) & = & \perp \\
\Phi_{e+}(v) & = & \lambda xy.(x\ v) \\
\Phi_{e+}(e_0\ e_1) & = & \lambda xy.(y\ (\Phi_{e+}(e_0)\ \Phi_{e+}(e_1)))
\end{array}
$$

$$
\begin{array}{lcl}
\Phi_{e-} & : & E_\perp \to E_\perp \\
\Phi_{e-}(\lambda\ xy.(x\ v)) & = & v \\
\Phi_{e-}(\lambda xy.(y\ (e_0\ e_1))) & = & (\Phi_{e-}(e_0)\ \Phi_{e-}(e_1)) \\
\Phi_{e-}(e) & = & \perp
\end{array}
$$

4.3 ベースレベルの表現

本節では、関数Φ_{s+}, Φ_{s-}の定義について述べる。

Φ_{s+}, Φ_{s-}は、メタ$^{n-1}$レベルの状態をメタ^{n}レベルの項に埋め込む関数である。つまり、(Φ_{s+}, Φ_{s-}) によって次の式が満たされる必要がある。

$$E \rhd N \times M \times (Con \times E)$$

Lambda/R において、Φ_{s+}, Φ_{s-}は、補助関数Φ_{r+}, Φ_{r-}, Φ_{C+}, Φ_{C-}を用いて次のように定義されている。

$$
\begin{array}{l}
\Phi_{s+} : (N \times M \times (Con \times E)) \to E \\
\Phi_{s+}(n, R, C[e]) \\
\quad = pair(pair(n, \Phi_{r+}(R)), pair(\Phi_{c+}(C), \Phi_{e+}(e))) \\
\Phi_{s-} : E \to (N \times M \times (Con \times E))_\perp \\
\Phi_{s-}(\lambda x.(x\ \lambda y.(y\ n\ e_1)\ \lambda z.(z\ e_2\ e_3))) \\
\quad = state(n, \Phi_{R-}(e_1), epair(\Phi_{C-}(e_2), \Phi_{e-}(e_3))) \\
\Phi_{s-}(e) = \perp
\end{array}
$$

ここで、(Φ_{r+}, Φ_{r-}) は規則を項に埋め込む関数の対であり、(Φ_{C+}, Φ_{C-}) は文脈を項に埋め込む関数の対である。また、関数 $pair, epair, state$ は、次のように定義される。

$$
\begin{array}{lcl}
pair & : & E \times E \to E \\
pair(e_0, e_1) & = & \lambda x.(x\ e_0\ e_1) \\
epair & : & (Con_\perp \times E_\perp) \to (Con \times E)_\perp \\
epair(\perp, e) & = & \perp \\
epair(C, \perp) & = & \perp \\
epair(C, e) & = & \langle C, e \rangle
\end{array}
$$

$$
\begin{array}{l}
state : (N \times A_\perp \times (C \times E)_\perp) \to (N \times A \times (C \times E))_\perp \\
state(n, \perp, C[e]) = \perp \\
state(n, r_m, \perp) = \perp \\
state(n, r_m, C[e]) = \langle n, r_m, C[e] \rangle
\end{array}
$$

以下では、関数$\Phi_{r+}, \Phi_{r-}, \Phi_{C+}, \Phi_{C-}$の定義について、それぞれ解説を行なう。

4.4 簡約規則の表現

Lambda/R において、メタ^{n}レベルの簡約規則の変更部分 r_nは$2^{(Con \times E) \times (Con \times E)}$の真部分集合であった。Lambda/R において、この簡約規則の実体は、`[(E, E)]` という型をもつ Gofer のリストによって表現されている。但し、Gofer において、`[(E, E)]` という型は、$E \times E$ の要素からなるリストを表す。このリストを `rlist` とすると、Lambda/R の簡約規則は次のように定められる。

$$
\begin{array}{l}
\forall C_0[e_0], C_1[e_1]. \\
\quad C_0[e_0] \overset{r_n}{\rightarrow} C_1[e_1] \Leftrightarrow \\
\qquad \exists e_2, e_3.\ (e_2,\ e_3) \in \texttt{rlist} \wedge \\
\qquad\quad eq(\Phi_{e+}(e_0), e_2) \wedge \\
\qquad\quad mc(e_3\ pair(\Phi_{C+}(C_0), \Phi_{e+}(e_0))) = C_1[e_1]
\end{array}
$$

$$
\begin{array}{lcl}
eq & : & (E \times E) \to Bool \\
eq(e, e) & = & True \\
eq(e, ?) & = & True \\
eq((e_0\ e_1), (e_2\ e_3)) & = & eq(e_0, e_2) \wedge eq(e_1, e_3) \\
eq(e_0, e_1) & = & False
\end{array}
$$

但し、$Bool = \{False, True\}$ とする。

上記より、簡約規則の記述に関して、定数 ? がいわゆるワイルドカードとして働くことがわかるであろう。例えば、任意の値 v に対して (:100! v) $\overset{r_n}{\rightarrow}$ 100 となるような簡約規則を設定したい場合は、((:100! ?), !x.100) という項の組が `rlist` に追加されるように簡約規則の表現を操作すればよい。

ここで、r_nから `rlist` への変換を行なう関数を $rtol$、`rlist` から r_n への変換を行なう関数を $ltor$ とすると、簡約規則の表現を作り出す関数 Φ_{r+}、及びその逆の働きをする関数 Φ_{r-} は次のように与えられる。

$$
\begin{array}{l}
\Phi_{r+} : A \to E \\
\Phi_{r+} = \Phi'_{r+} \circ rtol \\
\Phi_{r-} : E \to A_\perp \\
\Phi_{r-} = ltor \circ \Phi'_{r-} \\
\Phi'_{r+} : \texttt{[(}E,\ E\texttt{)]} \to E \\
\Phi'_{r+}(\texttt{[]}) = \lambda xy.x \\
\Phi'_{r+}(\texttt{(}e_0,\ e_1\texttt{):xs}) \\
\quad = \lambda xy.(y\ pair(pair(e_0, e_1), \Phi'_{r+}(\texttt{xs})))
\end{array}
$$

$$\Phi'_{r-} : E \to [(E,\ E)]_\perp$$
$$\Phi'_{r-}(\lambda xy.x) = []$$
$$\Phi'_{r-}(\lambda xy.(y\ \lambda z.(z\ \lambda x.(x\ e_0\ e_1)\ e_3)))$$
$$= cons((e_0,\ e_1), \Phi'_{r-}(e_3))$$
$$\Phi'_{r-}(e) = \perp$$
$$cons : ((E,\ E) \times [(E,\ E)]_\perp) \to [(E,\ E)]_\perp$$
$$cons(\mathtt{x}, \perp) = \perp$$
$$cons(\mathtt{x}, \mathtt{xs}) = \mathtt{x{:}xs}$$

4.5 文脈の表現

文脈の表現を作り出す関数Φ_{C+}, Φ_{C-}は、次のように定義される。

$$\Phi_{C+} : Con \to E$$
$$\Phi_{C+}([]) = \lambda xy.x$$
$$\Phi_{C+}(v\ C)$$
$$= \Phi'_{C+}(\lambda xy.(y\ \lambda xy.(x\ \Phi_{e+}(v))\ \lambda xy.x), C)$$
$$\Phi_{C+}(C\ e) \quad = \Phi'_{C+}(\lambda xy.(y\ \lambda xy.(y\ \Phi_{e+}(e))\ \lambda xy.x), C)$$
$$\Phi_{C-} : E \to Con_\perp$$
$$\Phi_{C-}(e) = \Phi'_{C-}([], e)$$

$$\Phi'_{C+} : (E \times Con) \to E$$
$$\Phi'_{C+}(e\ [\,]) = e$$
$$\Phi'_{C+}(e(v\ C)) = \Phi'_{C+}(\lambda xy.(y\ \lambda xy.(x\ \Phi_{e+}(v))\ e), C)$$
$$\Phi'_{C+}(e_0\ (C\ e_1))$$
$$= \Phi'_{C+}(\lambda xy.(y\ \lambda xy.(y\ \Phi_{e+}(e_1))\ e_0), C)$$
$$\Phi'_{C-} : (Cont \times E) \to Con_\perp$$
$$\Phi'_{C-}(C, \lambda xy.x) = C$$
$$\Phi'_{C-}(C, \lambda xy.(z\ \lambda u.(u\ v\ w))) = \Phi'_{C-}(\Phi''_{C-}(C, v), w)$$
$$\Phi'_{C-}(C, e) = \perp$$
$$\Phi''_{C-} \quad : \quad (Cont \times E) \to Con_\perp$$
$$\Phi''_{C-}(C, \lambda xy.(x\ e)) \quad = \quad capp(\Phi_{e-}(e), C)$$
$$\Phi''_{C-}(C, \lambda xy.(y\ e)) \quad = \quad cpass(C, \Phi_{e-}(e))$$
$$\Phi''_{C-}(C, e) \quad = \quad \perp$$
$$capp \quad : \quad (E_\perp \times Con) \to Con_\perp$$
$$capp(\perp, C) \quad = \quad \perp$$
$$capp(v, C) \quad = \quad (v\ C)$$
$$capp(e, C) \quad = \quad \perp$$
$$cpass \quad : \quad (Con \times E_\perp) \to Con_\perp$$
$$cpass(C, \perp) \quad = \quad \perp$$
$$cpass(C, e) \quad = \quad (C\ e)$$

4.6 プログラム例

本節では、Lambda/R による簡単なプログラム例を示す。

実装したLambda/R処理系には、演算子 `!>` によって得られたベースレベルの状態の表現から必要な必要な情報を取り出すための組み込み関数 `lv`, `rule`, `cont`, `term` が用意されている。同じく、`/lv`, `/rule`, `/cont`, `/term` は、ベースレベルの状態の表現の特定部分を入れ換える関数である。また、`nil`, `cons`, `head`, `tail` は、λ項によるリストの表現を与える関数であり、規則や文脈の表現の操作に利用できる。これは、4.4項、4.5項からわかるように、規則や文脈の表現が `nil`, `cons` によって構成されるリストと同じ構造をもつためである。

以下の例において、行の先頭の記号 ">" はLambda/Rのプロンプトである。

例 4.1 図4に、$C[(\mathtt{:100!}\ v)] \xrightarrow{r_n} \mathtt{100}$ という簡約規則を追加するプログラムを示す。`:100!` はシンボルであり、vは任意の値である。定数 `?` を用いることによって、任意の値に対して適用される簡約規則を定義している。

例 4.2 図5に、不動点演算子を用いて3の階乗を計算するプログラムを示す。まず、条件分岐関数 `:lif` (lazy if) を定義し、値呼び不動点演算子 `fix`、及び `:lif` を用いて3の階乗を計算している。`:lif` は、第1引数を第2引数と第3引数に適用する単純な関数であり、通常、第1引数として$\lambda xy.x$か$\lambda xy.y$をとる。但し、通常の関数とは異なり、`:lif` は第2引数と第3引数を簡約せずに第1引数の適用を行なう。Lambda/R は値呼び簡約を行なうので、ここで `:lif` でなくベースレベルの分岐関数を用いると、引数が先に簡約されてしまい、プログラムの実行は停止しない。

5 関連研究

本稿で示した自己反映的な値呼びλ計算の意味論は、文献[2]に示されているものと似ている。しかし、文献[2]におけるλ計算の拡張は、リフレクティブタワーの状態という概念がなく、実際的なプログラミング言語としてはまったく不十分である (但し、意味論が比較的簡単であり、プログラムの等式論理が与えられている点は評価できる)。

抽象書換え系にもとづくリフレクティブタワーのモデル化については、本研究とは異なる検討の方針が文献[6]に述べられている。また、任意のλ項を正規形のλ項で表現する手法については、文献[3]を参考にした。

6 おわりに

本稿では、自己反映的なプログラミング言語の意味論、及びそれにもとづくプログラミング言語の実装例を示した。これらの研究によって得られた成果は、次の通りである。

- リフレクティブタワーの状態をもったプログラミング言語において、自己反映計算を含む計算の意味をきわめて単純な規則によって表現できたため、今後の種々の検討に関する見通しが非常に良くなった。
- 構築した意味論にもとづいてプログラミング言語処理系を作成したことにより、実際のプログラミング言語に対する意味論の適用性を確認することができた。

```
> (!+ !x.
      (!- (/rule x (cons (pair (!> (:100! ?))
                               !x.(/snd x (!> 100)))
                         (rule x)))))
> (:100! 15)
100
> (+ 1 (:100! 24))
101
```

図 4: $C[(\texttt{:100!}\ v)] \xrightarrow{r_n} \texttt{100}$ という簡約規則の追加

```
> (!+ !x.(!- (/rule x
    (cons (pair (!> :lif)
          !y.(/snd y
               (!> (!+ !x.
                   (!- (!cx.(!tc.(!ttc.
                         (/term (/cont x (tail ttc))
                                (((head cx) !x.(!> 0) !x.(!< x))
                                ((head tc)  !x.(!> 0) !x.x)
                                ((head ttc) !x.(!> 0) !x.x)))
                           (tail tc)) (tail cx)) (cont x)))))))
          (rule x)))))
> (fix !f.!n.(:lif (= n 0) 1 (* n (f (- n 1)))) 3)
6
```

図 5: 不動点演算子を用いた 3 の階乗の計算

また、今後の課題としては、プログラム変換や正当性の検証などに関する検討が挙げられる。この点において、本稿で示した操作的意味論は、これらの検討の基盤となることが期待される。但し、自己反映計算を含む計算は、プログラム変換において極めて有利な特性である参照透明性をもたないため、一般的なプログラム変換の手法を確立することが困難であることが予想される。

また、もうひとつの興味深い研究の方向としては、プログラミング言語 Lambda/R の機能強化が挙げられる。Lambda/R は、文脈に関するリフレクションが可能という点でユニークであり、かつ強力な記述力をもっている。しかし、現段階では、Gofer インタプリタ上に実装されているため、実行速度が非常に遅いという欠点がある。より実用的なプログラミング環境を構築するためには、実行速度の向上、及び組み込み関数の整備などが必要である。

謝辞

本研究を進めるに当たって、北陸先端科学技術大学院大学言語設計学講座の二木厚吉教授、菅原太郎氏に有益な議論と助言を戴きました。記して感謝致します。

参考文献

[1] 井田昌之. 計算モデルの基礎理論. 岩波講座ソフトウェア科学 12. 岩波書店, March 1991.

[2] Anurag Mendhekar and Daniel P. Friedman. Towards a Theory of Reflective Programming Languages (Extended Abstract). In *OOPSLA '93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1993.

[3] Torben Æ. Mogensen. Efficient Self-Interpretation in Lambda Calculus. (unpublished), URL: http://www.diku.dk/research-groups/topps/bibliography/1992.html#D-128, 1992.

[4] Brian Cantwell Smith. Reflection and Semantics in a Procedural Language (Ph. D. Thesis). Technical Report TR-272, Laboratory for Computer Science, MIT, 1982.

[5] 渡部卓雄. リフレクション. コンピュータソフトウェア, Vol. 3, No. 11, pp. 5–14, 1994.

[6] 渡部卓雄. 抽象書換えにもとづく自己反映計算の定式化手法. 日本ソフトウェア科学会第 11 回大会論文集, pp. 309–312, November 1994.