知識工学と人工知能 60 － 6
（1988. 9. 21）

Prologにおけるリフレクションとその意味論

菅野 博靖

富士通(株)国際情報社会科学研究所

リフレクションとは、システムが自らの状態を認識しそれに基づいて自分自身の状態を変更していくという概念である。学習や非単調な推論を実現するという視点からも非常に興味深いものであるが、現在ではまだプログラミング技術の段階であり、理論的な考察はあまりなされていない。本稿では、論理型言語であるPrologにリフレクションを導入することを通して、特にリフレクションの持つ意味論の定式化を試みる。そして、それによりPrologにおけるメタロジカルな言語要素の意味を明確に捉えられることを示す。

Reflection in Prolog and Its Semantics

Hiroyasu SUGANO

International Institute for Advanced Study of Social
Information Science, FUJITSU LIMITED

1-17-25 Shinkamata, Ohta-ku, Tokyo 144, Japan

Reflection is one of the most promising architecture for intelligent computer systems. It enables systems to recognize and change their own computational states dynamically. Therefore, systems with reflective capability can yield the flexible behaviour such as learning, non-monotonic reasoning, and so on. In this paper we propose reflective Prolog (R-Prolog) and try to make a theoretical investigation on it, especially to formalize the semantics of reflection. We also show that R-Prolog makes us able to describe the semantics of meta-logical constituents of Prolog.

1 はじめに

リフレクティブ、すなわち内省的であることは、知的なシステムに対する基本的でかつ本質的な要求だと思われる。例えばプランニングやスケジューリング、さらに問題解決等の複雑な行為を行う場合を考えてみても、自らの(心理的あるいは肉体的)状態や置かれている環境を認識し、それに基づいた意思決定を行うことが極めて重要であることはいうまでもない。AIや認知科学の分野においても、内省と言うことの重要性は以前から認識されており、自己言及の問題やメタ認知の問題と絡めて議論されてきた(例えば[4,6]を参照)。また、計算機科学に話を限ってみても、計算機にリフレクティブな能力を持たせる試みは数多くなされてきており、身近なものでもデバッガやトレース機能、エラー処理機能などがすぐ思い浮かべられる。さらには高次のインタフェースや学習などを実現する上でも、リフレクティブな能力を持つことは不可欠である。

このような理論的にも応用的にも興味深い概念としての「リフレクティブな能力」を具体化するものとして、B.C.Smithによって提案されたリフレクション(あるいは計算的リフレクションともいう)という概念がある[5]。リフレクションとは、計算システムが自分の状態を認識しそれに基づいて自分自身の状態に変更を加えることのできる機構を、プログラミング言語レベルで実現させうるものである。Smithは、実際に3-lispという言語を実現することによって、それが可能であることを示した。3-lispのように、リフレクションを組み入れられた言語をリフレクティブな言語と呼ぶが、リフレクティブな言語が持つ利点をまとめてみると、次のようになる。1.記述力を落とすことなしに言語をコンパクトにできる。2.モジュール性を高めることができる。3.拡張性を高めることができる。

3-lisp以降、いくつかの言語にリフレクションを組み入れようとする試みが行われている[1,3,8]。しかし、それらにおいてもベースとなる言語の意味論が明確でないこともあって、アドホックな印象を否めない。リフレクションを単なるプログラミング技術から、理論的にも耐えうるようにするためには、意味論を明確にしておく必要がある。

本稿では、3-lisp等の言語で実現されてきたリフレクションの概念を論理型言語であるPrologの上に導入し、理論的な考察を加えることによってそれが持つ意味論を定式化することを試みる。そして、それによってこれまでその意味が不透明になりやすかったメタロジカルな述語を統一的な枠組の下に捉えられることを示す。まず次節では、一般的な視点からリフレクションとは何であるのかについて論じる。3節では、まずPrologとその意味論を形式化する。4節では、Prologにおいてリフレクションがどのように意味を持つのかを考察しながら、リフレクションを組み入れたReflective Prolog(R-Prolog)を提案する。そして、それによってPrologのプリミティブがR-Prologの中で定義できることを示す。5節では、3節の議論に基づいてPrologにおけるリフレクションの意味論を定式化することを試みる。最後に、6節でまとめと今後の課題を述べる。

2 リフレクションとは?

この節では以下の節の準備として、まず3-lisp等で実現されているリフレクションの本質は何であるのかを考察し、それによってリフレクションに関する一般的な枠組を設定することを試みる。

リフレクションとは、自らの状態を認識しそれに基づいて自分の状態を動的に変更して行く能力である、と述べた。プログラミング言語レベルでそれが可能であるためには、次の三つの要件が満たされなければならない。

1. メタレベルによる対象レベルの表現。
2. レベル間の推移メカニズム。

3. 因果的結合。

これらを、計算システムというモデルに基づいて説明することにしよう[1,3]。

2.1 計算システム

計算システムとは、実際の計算機を抽象化したものであり、図1のように表現することができる。

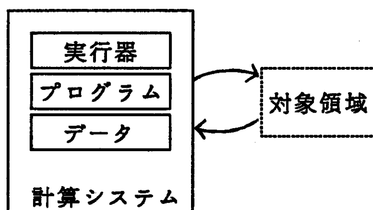

図 1: 計算システム

ここで、対象領域とは計算システムがその中にプログラムやデータとして表現している「世界の一部」であり、計算システムはそれについての計算あるいは推論を行っている。

2.2 メタレベルによる対象レベルの記述

一方、世界の一部としての対象領域には計算機のハードウェアやソフトウェアそれ自身も当然入りうるわけであり、その意味で計算システムについての計算システムというものも存在する。その典型的な例が、インタプリタ、コンパイラ、デバッガ等であろう。あるいは、Bowen とKowalskiによって提案され[2]、論理型言語のプログラミング技法として注目されているメタプログラミングもこの中に入る。このタイプの計算システムを特にメタ計算システムと呼ぶ。(図 2)

プログラミング言語にリフレクションの能力を持たせるためには、少なくともそれ自身のプログラムとその計算の状態をその言語の中で表現し操作できることが要求される。例

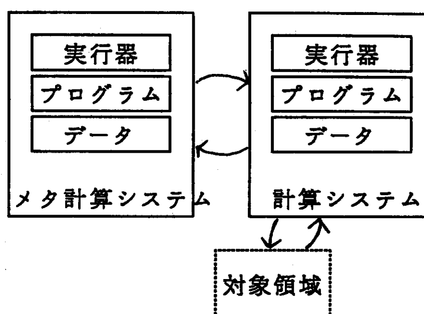

図 2: メタ計算システム

えば、lispの万能インタプリタのイメージである。実際、リフレクティブな言語におけるメタシステムはメタインタプリタとして実現されるのである。

2.3 レベル間の推移メカニズム

さらに、リフレクティブな言語は、メタレベルと対象レベルの間を行き来できなければならない。この要件が満たされなければ、単に複数のレベルで勝手な計算をしていることになるからである。しかし、ただレベルの推移ができれば良いと言うわけでもない。外見的にリフレクティブな動作をしているように見える学習システムやデバッガなどもこのレベル間の推移を行っているが、それらにおいてはすでにシステムに組み込まれた仕方で推移が行われる。たとえば、矛盾やエラーが発生したときに自動的にメタレベルに移動するように組み込まれている。リフレクティブな言語は、言語の中でそのレベル移動のタイミングを記述できることが要求される。これによって、さまざまな動作を柔軟に記述できることになる。

2.4 因果的結合

因果的結合の要件はリフレクションにおいて本質的である。因果的結合と言う概念は、対象領域とそれをモデル化している計算システテ

ム内のデータとの間に、相互に影響をおよぼし合う密接な関係があることを要求している。典型的な例として、ロボットアームの例が良く引き合いに出されるが、アームの位置と計算システム内の位置データは一方が変化すれば他方にもその影響が及ぶと言う意味で、因果的に結合されている。

リフレクティブなシステムとは、メタレベルと対象レベルとの間が因果的に結合されている計算システムである(図3)。すなわち、対象レベルにおける計算の状態はメタレベルにおけるデータに忠実に反映されなければならない。さらに、何らかの理由でメタレベルにおけるデータに変化があったら、対応する対象レベルの状態が変更されなければならない、ということである。この因果的結合の要件によって、リフレクティブなシステムは自己の状態を変更することができるのである。

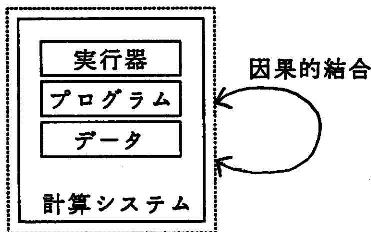

図3: リフレクティブな計算システム

2.5 メタインタプリタの無限階層

対象システムの状態をメタシステムで操作することによって変更を加えると述べたが、メタシステム自身の状態を変更するには同様にメタシステムのメタシステムが必要となる。概念的にこれをおし進めて行くとメタシステムの無限の階層を考えることができる。実際、リフレクティブなシステムの概念的モデルを考えると、このような無限のモデルが必要になる。これはすなわち、メタインタプリタの無限階層というモデルである。

しかし、現実に計算機上でリフレクティブなシステムを実現することを考えれば、このような無限のモデルは単なるモデルに過ぎないことは明らかである。このモデルを有限の世界で実現するためのテクニックがリイフィケーション(reification)であり[7]、3-lispをはじめとしてABCL/Rなどもこの技法によって実現されている。リイフィケーションとは、メタへのアクセスが必要になった時に、メタインタプリタのコピーを積み上げてゆくことである。このテクニックのために有限で実現することが可能なのである。

2.6 リフレクションの意義

リフレクションを用いることによって享受することができる利点をまとめてみると次のようになるであろう。

1. 記述力を落とすことなしに言語をコンパクトにできる。
2. モジュール性を高めることができる。
3. 拡張性を高めることができる。

3 Prologの形式化

Prologにリフレクションを取り入れる上で、いくつかの点をあきらかにしなければならない。例えば、ユニフィケーションやバックトラックといったProlog特有の機構をどのように扱って行くのかを考察することが必要である。そこでまず、Prologを形式化することによって問題点を明確にしよう。この形式化されたPrologをF-Prologを呼ぶことにする。

3.1 F-Prologの構文

定義 3.1.1 F-Prologは次の記号から構成される。

1. 変数。
2. 有限個の定数。
3. 有限個の関数。

4. 有限個の述語。
5. リスト記号[、|、]。
6. 区切り記号 ,(カンマ)、.(ピリオド)、:-(含意)。

以下では、これらの記号に対応した次のようなメタ記号を用いることにする。変数は、添字付き英大文字、例えば$X_1, \ldots, X_n, \ldots$、によって表される。定数、関数、述語記号は、添字付き(あるいは無し)の英小文字、例えばそれぞれ$c_1, \ldots, c_l$、$f_1, \ldots, f_m$、$p_1, \ldots, p_n$、と表される。その他の記号は、メタ記号としても同じ記号が用いられる。述語記号と関数記号にはそれぞれ唯一の非負整数が対応付けられており、その整数を対応する述語(または関数)のアリティと呼ぶ。アリティnの述語(または関数)記号をn項述語(関数)記号と呼ぶ。アリティを明示的に表すため、n項述語記号pをp/nと書く場合がある。また、Vを変数の集合、C, F, Pをそれぞれ定数記号、関数記号、述語記号の有限集合とする。この節では以降、V, C, F, Pを固定して考える。

定義 3.1.2 F-Prologの項、リスト、アトム、節は以下のように帰納的に定義される。

1. 項
   (a) 定数記号、及び変数記号は項である。
   (b) リストは項である。
   (c) fをn項関数記号、$t_1 \ldots t_n$を項とする時、$f(t_1, \ldots, t_n)$も項である。
2. リスト
   (a) []はリストである。
   (b) tを項とし、lをリストとする時、$[t|l]$はリストである。
3. アトム
   (a) pをn項述語記号とし、$t_1 \ldots t_n$を項とする時、$p(t_1, \ldots, t_n)$はアトムである。
4. 節
   (a) aをアトムとするとき、$a.$は節である。この形の節を特に単位節と呼ぶ。
   (b) $a, a_1, \ldots, a_k$をアトムとするとき、$a: -a_1, \ldots, a_k.$は節である。ただし、kは1以上の整数である。

Tを項の集合、ATをアトムの集合、LSをリストの集合、CLを節の集合とする。変数を持たない項を特に、閉項と呼ぶ。また、節$a.$あるいは$a: -a_1, \ldots, a_k.$において、aの述語をその節のキー述語と呼ぶ。

定義 3.1.3 変数の集合Vの有限部分集合から項の集合Tへの関数を代入と呼ぶ。

代入を、$\sigma, \sigma_1, \sigma_2, \ldots$の記号で表す。特に、恒等関数である代入を$\epsilon$で表す。$\{X_1, \ldots, X_n\}$を定義域とする代入$\sigma$に対して、$\sigma(X_i) = t_i (1 \leq i \leq n)$であるとき、$\sigma = < X_1/t_1, \ldots, X_n/t_n >$と書く。$\sigma = < X_1/t_1, \ldots, X_n/t_n >$であるとき、$t_1, \ldots, t_n$がすべて閉項であるとき、$\sigma$を閉代入と呼ぶ。さらに、代入の合成やユニフィケーションの概念は、通常の論理プログラムの理論に従うことにする。

次にデータベースとゴールの定義を与える。

定義 3.1.4 節の有限集合で、その中に含まれる任意の2つの節のキー述語が等しく、かつその上に線形順序関係が定義されている時、その集合を単一キー節集合と呼ぶ。単一キー節集合は、その順序にしたがってリスト形式で表される。

定義 3.1.5 *データベース*、あるいは*プログラム*とは、単一キー節集合の有限集合である。

定義 3.1.6 アトムの(空列も許す)有限列を*ゴール*と呼ぶ。

$a_1, \ldots, a_n$をアトムとするとき、これらからなるゴールをリスト形式で$[a_1, \ldots, a_n]$と書く。空列は、[]と書く。

3.2 F-Prologの意味論

次に、F-Prologにおいて計算の状態を形式化することにしよう。Prologの重要な特徴としてのバックトラックを形式化するためには、計算の各時点の代替案を記録しておく必要がある。そこで、次の定義を与える。

定義 3.2.1

1. ENV'を代入の集合、CLSを単一キー節集合の集合、GLをゴールの集合、Nを非負整数の集合とするとき、$ENV' \times CLS \times GL \times N$の元を<u>継続</u>と呼ぶ。
2. 継続を要素とするリストを継続スタックと呼ぶ。

定義 3.2.2 $ENV = ENV' \cup \{\perp\}$、$DB$をデータベースの集合、$CT$を継続スタックの集合、$N$を非負整数の集合とするとき、計算の状態の集合$ST$は次のように定義される。
$ST = DB \times ENV \times CT \times N$

すなわち、計算の状態は次の4つによって表される。

1. 各時点でのデータベース。
2. 変数への代入リストとしての環境。
3. 残りの仕事を記述しているスタック。
4. 探索の深さを記述している非負整数。

F-Prologの意味を関数によって定義して行こう。

定義 3.2.3 aをアトム、dbをデータベースとする。このとき、関数$\mathcal{U} : AT \times DB \rightarrow ENV \times CLS$を次のように定義する。
$\mathcal{U}(a, db) =$

$$\begin{cases} <\perp, []> & \cup db \text{の中に} a \text{とユニファイ可能な節がないとき} \\ <\sigma, [cl_1, \ldots, cl_n]> & a \text{とユニファイ可能な節のリストが} [cl_1, \ldots, cl_n] \text{であり、} a \text{と} cl_1 \text{の} mgu \text{が} \sigma \text{であるとき} \end{cases}$$

定義 3.2.4 関数$\mathcal{F} : AT \times ST \rightarrow ST$は次のように定義される。ただし、$s =< db, env, ct, n >$を$ST$の元、$a$をアトムとする。
$\mathcal{F}(a, s) =$

$$\begin{cases} < db, \perp, ct, n > \\ \qquad \mathcal{U}(a, db) =< \perp, [] > \text{のとき} \\ \mathcal{E}([b_1, \ldots, b_n], \\ \quad < db, env \cdot \sigma, [[env, [cl_2, \ldots, cl_n], a, n]|ct], n+1 >) \\ \qquad \mathcal{U}(a, db) =< \sigma, [cl_1, \ldots, cl_n] > \text{であり} cl_1 \text{のボディ部が} [b_1, \ldots, b_n] \text{のとき} \end{cases}$$

定義 3.2.5 関数$\mathcal{E} : GL \times ST \rightarrow ST$は次のように定義される。ただし、$s =< db, env, ct, n >$を$ST$の元、$a$をアトム、$g$をゴールとする。

$$\mathcal{E}(g, s) = \begin{cases} s \qquad g = [] \text{のとき} \\ \mathcal{F}(b, < db, env', ct', m >) \\ \qquad g = [a|g'] \text{で、かつ} \mathcal{F}(a, s) =< db^+, \perp, ct^+, n^+ >\text{、} ct = [[env', [cl_2, \ldots, cl_n], b, n]|ct'] \text{のとき} \\ \mathcal{E}(g', \mathcal{F}(a, s)) \\ \qquad \text{それ以外のとき} \end{cases}$$

関数$\mathcal{E}$と$\mathcal{F}$が相互帰納的に定義されていることに注意する必要がある。関数$\mathcal{E}$がF-Prologのトップレベルを構成している。

定理 3.2.1 dbをデータベース、gをゴールとする。このとき、

$$\mathcal{E}(g, < db, [], [], 0 >) = \sigma$$

でかつ、$\sigma \neq \perp$ならば、$g\sigma$はdbの論理的帰結である。

この結果は、ここで示した意味論が論理学的に見たPrologの意味論（宣言的意味論）に対して健全であることを示している。この証明については、それほど困難無く示すことができるが、別稿に譲ることにする。

4 Prologにおけるリフレクション

この節では、前節での形式的な議論に基づいて、Prologにリフレクションを組み入れたR-Prologを提案する。

4.1 なぜPrologにリフレクションか？

なぜPrologにリフレクションを組み入れようとするのか、すなわち、それにどのような意義があるのだろうか。この回答として、少なくとも次の二つを与えることができるだろう。一つは、Prologが非常に多くの言語要素を用意しているため、その宣言的意味はもちろんのこと手続き的な意味も不明瞭になりがちだということである。前節で述べたように、リフレクションの機構を導入することによって少ないプリミティブで言語を構成することができる。これは言語の意味論を明確にするためにも重要なことである。

もう一つは、メタプログラミングとの関係である。メタプログラミングは、Prologを用いて知識の獲得や無矛盾性の管理といった高次の処理を行おうとするときに不可欠な技術である。しかし、その重要性にも関わらず、理論的な考察はほとんどなされていない。リフレクションは、メタプログラミングに対してより広い視点からの枠組を提供している。リフレクションが理論的に深く論じられるようになれば、メタプログラミングに対する指針にもなる。

この二つは非常に重要な点である。これらについては、後で論じることにする。

4.2 R-Prolog

前節で形式化したように、Prologの計算の状態は1.データベース 2.環境 3.継続4. 深さによって表される。リフレクションを取り入れるためには、この4つを明示的に扱うメタインタプリタを用意し、その無限階層モデルを考える必要がある。

図4に、R-Prologのメタインタプリタの一部を示す。述語solveがメタインタプリタのトップレベルである。述語tryは、与えられたサブゴールが単純なものの(リフレクトされてない)場合、collect_clsによってそのサブゴールとユニファイ可能な節のリストを見つけて、expand_clsでそれらを展開する。しかし、サブゴールがリフレクトされていると、一つ上のレベルのメタインタプリタに処理を任せる。makemetagoalは、メタレベルでのゴールを構成するものである。makenewenvとmakenewctによって新しい環境と継続スタックを作る。現在、このメタインタプリタを動かすためのベースレベルの実現をQuintus Prolog上で行っているところである。

R-Prologを用いたプログラミングの例を次に示す。まず、カットの例を考えてみよう。カットはPrologにおいて非常に重要な役割を果たしているが、それがPrologの持つ論理的な意味を壊してしまうために厄介ものとして扱われている。リフレクションを用いることによって、カットを言語の中で定義できる。

```
!:-reflect([cut]).
cut(Db,Db,Env,Env,[First|Rest],NewBts,Lev):-
        complevel(First,Lev),
        cut(Db,Db,Env,Env,Rest,NewBts,Lev).
cut(Db,Db,Env,Env,Bts,Bts,Lev).
```

ここで、complevelはスタックの先頭の要素のレベルと現在のレベルを比較するものであり、現在のレベルから一引いた値より深いか等しければポップする。さらにこのような形で、Waterloo Prolog等で提供されている先祖(ancester)カットもR-Prologの中で実現できる。

次に、assertやretract等のデータベース管理のための述語もリフレクションを用いて定義できることを示す。メタインタプリタはシステムの状態としてデータベースを持って

いるので、それを操作してやれば良い。ここで、retract1(A,Db,NewDb)はDbからAとマッチする節を取り除く述語である。

```
asserta(A):-reflect(asserta(A)).
asserta(A,Db,[A|Db],Env,Env,Bts,Bts,Lev).

retract(A):-reflect(retract(A)).
retract(A,Db,NewDb,Env,Env,Bts,Bts,Lev):-
          retract1(A,Db,NewDb).
```

5 R-Prologの形式化

この節では、F-Prologの意味論に基づいて、R-Prologの意味論を形式化することを試みる。それによって、リフレクションを用いた高次推論メカニズムを理論的に考察する基礎を与える。

5.1 R-Prologの構文

まず、R-Prologの構文を形式化しよう。

定義 5.1.1 R-Prologの記号は定義3.1.1で述べたF-Prologの記号に次のものが加わる。

1. 述語の中に特別なものとして、reflect。
2. 特殊記号、′ (quote)。

定義 5.1.2 R-Prologの項、リスト、アトム、節は定義3.1.2のF-Prologと同様に定義されるが、新たに加わった記号に対して次の項目が加わる。

1. sを項、リスト、アトム、節のいずれかとするとき、$'s$も項である。この形式の項を、特にquote形式という。
2. aをアトムとするとき、reflect('a)はアトムである。

′はメタ記号としても同じものを用いる。ここで、quoteには、次のような性質を課す。
$'s_1 \equiv 's_2$ならば、$s_1 \equiv s_2$。

定義 5.1.3 閉項とは、$quote$形式の項であるか、変数を持たない項である。

単一キー節集合とデータベース、ゴールの定義はF-Prologに従う。

5.2 意味論

F-Prologでは、計算の状態として、$ST = DB \times ENV \times CT \times N$を持ってきた。リフレクションを扱うためには、無限を扱えるようなモデルを構成しなければならない。そのために、次のように状態を考える。

定義 5.2.1 $ST^* = ST \times ST \times ST \times \cdots$

3.2.で定義した関数、$\mathcal{E}$、$\mathcal{F}$をここでも同様に用いる。

定義 5.2.2 関数$\mathcal{F}^* : AT \times ST^* \rightarrow ST^*$は次のように定義される。ただし、$s^* =< s_1, s_2, s_3, \ldots >$を$ST^*$の元、$s_1 = < db_1, env_1, ct_1, n_1 >$、$s_2 = < db_2, env_2, ct_2, n_2 >$、$a$をアトムとする。

$\mathcal{F}^*(a, s^*) =$

$$\begin{cases} extract(\mathcal{F}^*(a', < s_2, s_3, \ldots >)) & a\text{=reflect}('p(t_1, \ldots, t_n))\text{のとき、ただし} a' = p(t_1, \ldots, t_n, 'db_1, Newdb, 'env_1, Newenv, 'ct_1, Newct, 'n_1) \\ < \mathcal{F}(a, s_1), s_2, \ldots > & \text{それ以外のとき} \end{cases}$$

ここで、extractはs^*からs^*への関数で、次のように定義される。ただし、$s^* =< s_1 =< db, env, ct, n >, s_2, \ldots >$とする。

$extract(s^*) =$

$$<< env(Newdb), env(Newenv), env(Newct), n_1 >, s_1, \ldots >$$

定義 5.2.3 関数$\mathcal{E}^* : GL \times ST^* \rightarrow ST^*$は次のように定義される。ただし、$s^* =< s_1, s_2, s_3, \ldots >$を$ST^*$の元、$s_1 =< db_1, env_1, ct_1, n_1 >$、$s_2 =< db_2, env_2, ct_2, n_2 >$、$a$をアトムとする。

$\mathcal{E}^*(g, s^*) =$

$$
\begin{cases}
s^* & g = [\,] \text{のとき} \\
\mathcal{F}(b, < db, env', ct', m >) & g = [a|g'] \text{で、かつ} \mathcal{F}^*(a, s^*) = \\
 & << db^+, \perp, ct^+, n^+ >, s_2, \ldots >\text{、} ct^+ = \\
 & [[env', [cl_1, \ldots, cl_n], b, n]|ct'] \text{のとき} \\
\mathcal{E}(g', \mathcal{F}^*(a, s^*)) & \\
 & \text{それ以外のとき}
\end{cases}
$$

ここでも、$\mathcal{F}^*$と$\mathcal{E}^*$は相互帰納的に定義されている。

リフレクティブな言語の意味は、まさにこの$\mathcal{E}^*$によって表現されるのである。たとえば、カットの例を考えてみよう。

4.2.節で示したリフレクションによるカットの定義からその意味は次のように表される。

$$
\begin{aligned}
&\mathcal{E}^*(!, < s_1, s_2, \ldots >) \\
&= extract(\mathcal{E}^*(cut('db, Newdb, 'env, Newenv, \\
&\qquad 'ct, Newct, 'n), s_2))
\end{aligned}
$$

6 おわりに

本稿では、論理型言語であるPrologにリフレクションの機構を持たせたR-Prologを提案し、その意味論を形式化することを試みた。そして、それに基づいてPrologの意味論を統一的に表現できることを示した。

今後の課題としては、第一に理論的考察を深めることがあげられる。それによってリフレクティブなプログラムの正当性や同値性などの議論を形式的に行うことができる。第二にあげられることは、R-Prologの実現を行うことによって、リフレクションやメタプログラミングの応用分野を開拓することである。さらに、知識獲得や学習、非単調推論に対するリフレクションからの、実現と形式化という興味深い課題も残されている。また、実現に関しては、パフォーマンスの問題をどう解決するかが問題である。

謝 辞

本研究を進めるにあたり、貴重な時間を割いて議論に付き合って下さり、有益な助言を下さった国藤室長、田中(二)、神田研究員に感謝致します。

References

[1] 田中二郎. メタプログラミングとリフレクション. *Bit*, 20(5):41–50, 1988.

[2] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In *Logic Programming*, pages 153–172, Academic Press, 1982.

[3] P. Maes. Reflection in an object-oriented language (draft). In *Meta-Level Architecture and Reflection*, page , 1986.

[4] D. Perlis. Language with self-reference i: foundations. *Artificial Intelligence*, 25:301–322, 1985.

[5] B. C. Smith. Reflection and semantics in lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[6] B. C. Smith. Varieties of self-reference. In *Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 19–43, 1986.

[7] M. Wand and D. P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *ACM Symposium on Lisp and Functional Programming*, pages 298–307, 1986.

[8] T. Watanabe. *Reflection in Object-Oriented Concurrent Systems.* Master's thesis, Tokyo Institute of Technology, 1988.

```
solve([],Db,Db,Env,Env,CT,CT,Lev).
solve(Goals,Db,NewDb,fail,fail,CT,NewCT).
solve([Goal|Goals],Db,NewDb,Env,NewEnv,CT,NewCT,Lev):-
        try(Goal,Db,IntDb,Env,IntEnv,CT,IntCT,Lev),
        solve(Goals,IntDb,NewDb,IntEnv,NewEnv,IntCT,NewCT,Lev).
try(reflect(Goal),Db,NewDb,Env,NewEnv,CT,NewCT,Lev):-
        makemetagoal(Goal,Db,NewDb,Env,NewEnv,CT,NewCT,Metagoal),
        Metagoal.
try(Goal,Db,NewDb,Env,NewEnv,CT,IntCT,Lev):-
        collect_cls(Goal,Clauses,Db),
        expand_cls(Goal,Clauses,Db,NewDb,Env,NewEnv,CT,NewCT,Lev).
expand_cls(Goal,[],Db,NewDb,Env,NewEnv,CT,NewCT,Lev):-
        cutct(CT,IntCT,Env,Lev,NewLev,IntDb,NewGoal),
        try(NewGoal,IntDb,NewDb,Env,NewEnv,IntCT,NewCT,NewLev).
expand_cls(Goal,[Cl|Cls],Db,NewDb,Env,NewEnv,CT,NewCT,Lev):-
        getbody(Cl,Body),
        makenewenv(Goal,Cl,Env,NewEnv).
        makenewct(Goal,Cls,Env,CT,IntCT,Lev)
        solve(Body,Db,NewDb,Env,NewEnv,IntCT,NewCT,[0|Lev]).
makenewenv(Goal,Cl,Env,NewEnv):-
        unify_head(Goal,Cl,Sub),
        compose(Env,Sub,NewEnv).
makenewct(Goal,Cls,Env,CT,[[Goal,Cls,Env,Lev]|CT],Lev).
```

図 4: R-Prologのメタインタプリタ(一部)