Vol. 30 No. 6　　情　報　処　理　　June 1989

解　説

メタ推論とリフレクション†

菅野博靖††　田中二郎††

1. はじめに

推論システムやプログラミング言語に柔軟な推論(計算)メカニズムを組み込むために，メタ推論とかリフレクションと呼ばれる概念がここ数年注目されている．これらの概念の基本的な考え方は，「対象レベルとメタレベルを分離し，対象レベルの推論や計算に関するメタな情報を，メタレベルで明示的に記述し操作する」というものである．これらのアイディアに基づいて，推論の戦略についての知識をメタ知識として記述しておいて，状況に応じてダイナミックに戦略を変更したりすることのできる推論システムや，計算の制御に関するさまざまなメタ手続きをユーザ自身によって定義できる言語などが提案されている．本解説の目的は，これらの概念を論理という統一された視点から見渡すことによって考察することにある．

「メタ」とか「リフレクション」という概念は，もともと論理や計算の世界と密接な関係をもっている．たとえば論理学や計算理論の重要な結果のいくつかは，論理や計算の体系について記述するメタな体系を積極的に利用することによって産み出されたものである．また，知識工学や人工知能の分野で関心をもたれている信念や知識の論理においても，メタの概念は重要な役割を果たしている．後者については，本特集号の「知識と信念の論理」でも解説されているように様相論理を用いた形式化がよく知られているところであるが，メタの概念はそれに対するもう一つ別のアプローチを提出するものといえる．論理の枠組みの中にメタの概念を導入することによって，標準論理の拡張に対する独自の方向性を与えることができる．

本稿では，論理の枠組みにこだわりながら，メタ，リフレクションといったキーワードを中心に解説する．次章では，論理の枠組みにおけるメタとリフレクションの概念について概観する．3.では，さまざまなかたちで提案されているメタレベルアーキテクチャの中で，その後の研究に少なからず影響を与えた代表的なシステム，特に，FOL[24]，Bowen & Kowalski のメタプログラミングアプローチ[1),2)]，3-lisp[12),13)] について簡単にまとめる．4.では，論理プログラミングの枠組みの中でメタ推論とリフレクションがどのように捉えられるのかを解説する．5.では，リフレクションを計算モデルとして組み込んだ計算的リフレクションについて最近の発展を紹介する．

2. 論理学におけるメタレベル

論理の体系，あるいは形式的体系は，純粋には単なる記号の体系だが，しばしばそれらはなんらかの対象領域を意図して与えられることがある．たとえば，自然数の体系を扱う数論や，集合の概念を形式化する集合論などである．いま，対象領域をある形式的体系自体に置き，その体系のもつ性質，たとえば「この理論は有限の公理をもつ」とか「この論理式は公理から証明できる」などの性質を記述することを考える．すると，この記述に用いられる言語（ここでは日本語である）は形式的体系に対するメタ言語になる．さらに，メタ言語として形式的体系の言語を用いる場合，記述に用いる形式的体系は記述される形式的体系（これを対象系と呼ぶ）に対するメタ体系（メタ系）と呼ばれる(図-1)．この章では，メタ系による対象系の記述についてもう少し詳しく述べることにしよう．

2.1 形式的体系と表現可能性

形式的体系 T は，言語 L_T と論理式の構成規則 F_T，公理 A_T，および推論規則 R_T からなる．言語 L_T は記号の集合であり，L_T の要素に有限列（これを記号列と呼ぶ）のうち，L_T から F_T に従って構成される記号列を L_T の論理式と呼ぶ．A_T は L_T の論理式の集合である．L_T の論理式 ϕ が A_T の論理式から R_T を用いて構成できるとき ϕ は T で証明可能

† Meta Reasoning and Reflection by Hiroyasu SUGANO and Jiro TANAKA (International Institute for Advanced Study of Social Information Science, FUJITSU LIMITED).

†† 富士通(株)国際情報社会科学研究所

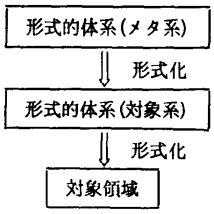

図-1 形式化の階層

であるといい，$\vdash_T \phi$ と書く．一般に形式的体系 T には，命題論理や一階述語論理，ホーン論理などの種類がある．この章では以降，一階述語論理の体系に絞って議論を進めていく．

いま，ある対象領域 A を形式的体系 T で記述することを考えよう．そのためには，A の個々の対象に言及できるだけの十分な強力さが T の言語 L_T に備わっていることが望ましい．たとえば自然数についての理論（数論）は，任意の自然数に対してそれに対応する名前としての基底項（変数を含まない項）をもつべきである．本稿では，形式的体系が領域上のすべての元に対して，一意に定まる名前をもつことを仮定しよう．領域 A の元 a に対する T における名前を $\ulcorner a \urcorner$ と表すことにする．しかし，領域の個々の対象に言及できるだけでは，領域について記述する形式的体系として不十分である．ここで，形式的体系における領域上の関係の表現可能性という概念が必要になる．領域 A 上の n 項関係 R が形式的体系 T で（強い意味で）表現可能であるとは，T の言語で構成される n 変数の論理式 ϕ が存在し，任意の A の元 $a_1, \cdots, a_n$ に対して次が成り立つことである．

$$R(a_1, \cdots, a_n) \Longrightarrow \vdash_T \phi(\ulcorner a_1 \urcorner, \cdots, \ulcorner a_n \urcorner)$$

$$R(a_1, \cdots, a_n) \text{ でない} \Longrightarrow \vdash_T \neg\phi(\ulcorner a_1 \urcorner, \cdots, \ulcorner a_n \urcorner)$$

さらに，表現可能性は T が無矛盾であることを仮定すれば，

$$R(a_1, \cdots, a_n) \Longleftrightarrow \vdash_T \phi(\ulcorner a_1 \urcorner, \cdots, \ulcorner a_n \urcorner)$$

を含意する．この条件が成り立つとき，R は T で弱い意味で表現可能，あるいは T において論理式 ϕ によって弱い意味で表現されるという．

ここで，対象領域として形式的体系 O を取り O 上の証明可能性をメタ系 M で表現したい．つまり O を，L_T から F_T によって構成される O の記号列（構文的対象）を要素とし，「項である」，「論理式である」，「公理である」，「〜は〜の証明である」，などを関係としてもつ領域と考える．O の公理が帰納的（recursive）であり，かつ M ですべての原始帰納的関数が表現可能な場合（たとえば適切な算術の公理系を含んでいる場合），「O 上で論理式 F は証明 P（P は論理式の有限列で表せる）をもつ」という関係が（上記の意味で）表現可能であることを示すことができる（これを表現する論理式を has-proofo(F, P) と書く）．しかし「論理式 F は O で証明可能である」という関係は，論理式 provableo$(F) \equiv (\exists x)$has-proofo(F, x) によって弱い意味では表現可能だが，（強い意味で）表現可能ではないことに注意する必要がある．この事実は，1930 年に Gödel, K. が示した不完全性定理（算術を含む形式的体系における決定不能な命題の存在）の証明において明らかにされたものである．Gödel は特殊な名前付けを行うことによって，上記の M を M 自身のメタ系（すなわち $O=M$）として用いた．これによって，M の中で M の論理式についてのメタな言明を表す論理式（直観的意味は「この論理式は証明不可能である」というもの）を構成し，その論理式とその否定がともに証明不可能であることを示したものである．ここで用いられた名前付けの手法はその後 Gödel 数化（符号化）と呼ばれ，論理学や計算理論のいくつもの重要な結果（特に決定不能性の結果）を導くのに利用されているのは周知のとおりである．

ところで，M の論理式 provable$_M(X)$ は次のような興味深い性質をもつ．

1. $\vdash_M \phi \Rightarrow \vdash_M$ provable$_M(\ulcorner \phi \urcorner)$
2. $\vdash_M$ provable$_M(\ulcorner \phi \urcorner) \wedge$ provable$_M(\ulcorner \phi \rightarrow \psi \urcorner) \rightarrow$ provable$_M(\ulcorner \psi \urcorner)$
3. $\vdash_M$ provable$_M(\ulcorner \phi \urcorner) \rightarrow$ provable$_M(\ulcorner$ provable$_M(\ulcorner \phi \urcorner) \urcorner)$

この三つの性質は Löb の derivability condition と呼ばれているが，ここでの provable$(\ulcorner \phi \urcorner)$ を $\Box\phi$ と読み替えれば様相論理に類似した公理系を得ることができる．たとえば，第一の性質は必然規則の変形であり，第二の性質は厳密帰結公理と呼ばれるものに対応する．この体系を基にして，証明可能性を様相論理として形式化しようとする研究もあり，それは証明可能性論理（Provability Logic）と呼ばれている（たとえば，文献[14]を参照されたい）．ところで，上の性質を様相記号で読み替えた体系は様相論理 S_4 の体系から必然性の公理（$\Box\phi \rightarrow \phi$）を除いたものである．$S_4$ の体系は，一方で，知識の論理の枠組みとしても研究されているのであるが（そのときには□は知識を表す様相記号 K で置き換えられている），上述の体系 M との関係で次のような事実が明らかになっている．$K\phi$ を

M の中で「 ¬ 記号を用いて $K(\ulcorner\phi\urcorner)$ と表し，必然性の公理に対応する公理を導入すると矛盾が発生するのである (Montague の定理)．この事実は，様相論理とメタを用いた拡張との関係を考えるうえで興味深い．これらの問題についての詳細はたとえば[3] を参照のこと．

2.2 Feferman のリフレクション原理

Gödel の結果は「一階述語論理＋算術」という体系の不完全性，すなわち決定不能命題の存在を示すものであった．この結果は，体系に有限個の新しい公理を付け加えていっても依然として成り立つことが証明されている．しかしながら，この体系にある（適切な）メタな言明を超限的に加えていくことによって Gödel の結果を弱められることが，Turing, A. や Feferman, S. によって証明されている．ある形式的体系 A を出発点とした次のような理論の超限列を考えよう．

（a） $A_0 = A$

（b） $A_{\alpha+1} = A_\alpha \cup S$ ただし，S はある条件を満たす論理式の集合

（c） α が極限順序数ならば，$A_\alpha = \bigcup_{\gamma<\alpha} A_\gamma$

ただし，A_α の添え字 α は構成的順序数である[4]．ここで，（b）のある条件を満たす論理式 ϕ としてたとえば，$S = \{\mathrm{Con}_{A_\alpha}\}$ (Con_{A_α} は公理集合 A の無矛盾性を表す論理式) や $S = \{\mathrm{provable}_M(\ulcorner\phi\urcorner) \rightarrow \phi \mid \phi$ は閉論理式$\}$ などの集合が考えられている．Feferman は，この条件を満たす拡張手続きを総称してリフレクション（反射）原理 (reflection principle) と呼んだ．すなわち，Feferman のいうリフレクション原理とは，元の論理式集合から証明できないメタな言明を付け加えることによる理論の拡張を意味している．

さて，具体化されたリフレクション原理を用いることによって，いくつかの興味深い結果を示すことができる．Feferman は S として$\{\forall x \mathrm{provable}_M(\uparrow\phi(x)) \rightarrow \forall x\phi(x) \mid \phi(x)$ は自由変数を唯一つもつ論理式$\}$（ただし $\uparrow\phi(x)$ は x を quote せずに全体を quote することを表す．詳しくは **4.3** を参照のこと）をとると，すべての真なる算術の命題をある順序数のレベルで証明できることを示した．つまり，Gödel の不完全性の結果を超限的な手法を用いることによって弱められることが示されたわけである．ここでの議論は当然のことながら構成的なものではなく，計算機の世界を対象にする以下の議論とは直接結びつくわけではない．しかし，メタの概念を利用した理論の拡張が興味深い結果を与えることを，その後の多くの研究者に印象付けている．

3. 代表的なメタレベルアーキテクチャ

メタレベルを明示的に扱うシステムが計算機科学の分野で注目され始めたのは，Weyhrauch, R. の FOL[24] が提案された 1970 年代の後半からである．80 年代に入ってからは，「メタ」とか「リフレクション」という概念を利用したさまざまなアーキテクチャが現れてきている．この章では，歴史的にみて重要な Weyhrauch の FOL, Bowen, K. A. と Kowalski, R. A. のメタプログラミング，Smith, B. C. の 3-lisp について解説し，メタ推論とリフレクションの研究の流れを紹介する．

3.1 Weyhrauch の FOL

Weyhrauch, R. の提案した FOL[24]は，一階述語論理を対象にしたインタラクティブな推論システムである．FOL にはいくつかの独創的なアイディアが取り入れられており，計算機科学や人工知能のその後の研究に大きな刺激を与えた．

FOL は，一階述語論理の論理式やモデルなどに対応するデータ構造を操作するが，その基本単位となるのは LS pair と呼ばれる三つ組 〈L, SS, F〉 である (図-2)．ここで L は言語定義部，SS はモデルに対応するデータ構造 (simulation structure と呼ばれる)，F は公理と定理のデータベースである．L で与えられた個々の記号は，SS 上の関数や関係に対応付けられており（意味論的結合 semantic attachment と呼ばれる），形式的な項や論理式の評価を SS 上で行うことができる．FOL は，与えられた論理式の証明を，F を用いた構文的な書き換えと意味論的結合を用いた評価を利用しながら，自然演繹法に基づく推論によって実行する．

FOL の革新的なアイディアは，他の LS pair（あるいは理論）に対するメタ推論として *META* という特別な LS pair を用意したことにある．*META* は他の理論（これを T とする）を simulation structure としてもち，T の構文的および意味論的対象に言及するための言語 (*ML*) と T に対する意味論的結合，および T についての公理と定理 (*MF*) をもつ理

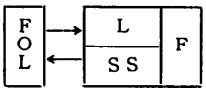

図-2 FOL と LS pair

論である．そして意味論的結合によって，*ML* の記号は *T* の対象や関係に対応づけられている．しかし，メタ系と対象系との間にはその表現の忠実度を保証する条件が必要である．Weyhrauch はそれを Feferman を引用しつつリフレクション（反射）原理と名付けているが，Feferman の用法と異なり，実質それは **2.1** で述べた弱表現可能性に対応する概念である．Weyhrauch のリフレクション原理は，メタレベルの重要性を意識する AI 研究者の間では標準的な用語として用いられている．Weyhrauch のリフレクション原理は非常に一般的な条件であるが，中でも重要な証明可能性について次のように表すことができる．ここでは，対象系 *O* での証明可能性をメタ系 *M* での述語 *Pr* で表すとする．

$$\frac{\vdash_M Pr(\ulcorner A\urcorner)}{\vdash_O A} \qquad \frac{\vdash_O A}{\vdash_M Pr(\ulcorner A\urcorner)}$$

すなわち，メタレベルで $Pr(\ulcorner A\urcorner)$ が証明できれば対象レベルで *A* が証明でき，逆もまた成り立つというものである．FOL はこのリフレクション原理を用いることによって，証明の中でメタ定理を有効に利用することができる．

ところで，すべての LS-pair のメタ理論である *META* は，自分自身のメタ推論でもある．この事実を用いて FOL は，セルフリフレクション（自己反射 self reflection）という概念を提案している．セルフリフレクションによって FOL では，メタ定理を用いて新たなメタ定理を得ることができる．しかし，FOL においてはメタレベルの利用によって得られる定理は，本質的に対象系で証明可能なものに限られているので（これはリフレクション原理が要請している），この意味で FOL でのメタの利用は保守的拡張にすぎない．

3.2 Bowen と Kowalski のメタプログラミング

FOL においては，対象系とメタ系はあくまでも個別の体系であり，それらの間はただリフレクション原理によって関係付けられていた．Bowen, K. A. と Kowalski, R. A. のメタプログラミングは，論理プログラムにおいて対象言語とメタ言語を融合する (amalgamate) ことにより，対象レベルとメタレベルの混在するプログラムを許している[1]．これによって彼らは，平板といわれる論理プログラムにおいて柔軟なプログラミングの可能性を示している．

彼らは，ある体系の中でそれ自身の証明可能性を弱表現する述語，いわゆる demo 述語を提案し，それを用いた興味深いプログラムの例をいくつか示した．特に彼らは，ホーン論理における demo 述語の定義の枠組みを示している．証明可能性を弱表現する demo 述語に対して，

$$A\vdash_L B \Longleftrightarrow \vdash_M \mathrm{demo}(\ulcorner A\urcorner, \ulcorner B\urcorner)$$

が成り立つ．ここで，*A* はホーン節の有限集合，*B* はアトムの連言であり，$\vdash_L$ と $\vdash_M$ はそれぞれ対象系とメタ系における証明可能性を表す．この関係は，Weyhrauch の反射原理に対応するもので，融合された言語の正当性を保証している．またこの demo 述語は，プログラミング言語の言葉でいえばホーン論理のメタインタプリタを記述していることに注意しよう．

demo 述語の新しい点は，公理集合 *A* を明示的に指定した証明可能性を表していることにある．これは，論理プログラムが有限であるという制約で可能になったことであるが，これによって非常に柔軟性に富んだプログラムを書くことができる．たとえば，融合された言語で書かれた次のようなプログラムをみてみよう．

```
innocent(X)←person(X),
            relevant(Facts),
            ¬demo(Facts, ⌜guily(X)⌝).
```

このプログラムは，無罪であることは (innocent(*X*)) 関連する事実から有罪であること (guilty(*X*)) が証明できない場合に導かれることを記述している．いくつもの公理からの証明可能性を一つのプログラムの中で扱うことができるので，多世界モデルを実現しているとも言える．demo 述語を利用したデータベース管理（たとえば知識の同化や調節[7]）や，demo 述語を拡張して制御命令や入出力情報を引数に取ることも提案されている．さらに，demo 述語によって非単調推論をモデル化することもできる．たとえば，デフォルト規則 ($\alpha : M\beta/\omega$) などに現れる「β の否定が証明できないとき」という条件は，demo 述語が利用できる格好の例である．

しかしながら，彼らの言語には変数の扱いに不明瞭な部分が残っている．たとえば，先の innocent(*X*) の例においても「guilty(*X*)」はアトム guilty(*X*) の名前であり，変数 *X* は quote 記号の外からは参照できないはずである．このようなレベルをまたがる変数の参照の問題（しばしば quantifying -in の問題と呼ばれる）は慎重に扱わなければならない．

3.3 Smith, B. C. の 3-lisp

ここまでの議論は主に論理という静的な体系におけ

るものであった．これは，前節の Bowen らのアプローチにおいても基本的に同様である．しかし，プログラムの実行とか挙動といった本質的に動的な対象の中でメタの概念を扱うには，また一つ異なった視点が必要になる．Smith, B.C. の提案した 3-lisp[12),13)] とそこから生まれてきた計算的リフレクションの概念は，ダイナミックな計算の中での自己変更の可能性を秘める興味深いアイディアを含んでいる．

計算的リフレクションは，計算の状態を明示化することで，その状態にアクセスし，変更することを可能にするアーキテクチャである．3-lisp では計算の状態として，変数の束縛状況を表す環境 (environment) とこれから成すべき仕事を表す継続 (continuation) を扱っていて，リフレクティブ手続き (reflective procedure) と呼ばれる関数を呼び出すときに現在の環境と継続を引数として受け取り，変更を行う．しかしながら，計算の途中においてその時点の計算の状態をデータとして扱うには，ある意味で現在の計算を見下ろせる位置，すなわちメタなレベルに立たなければならない．3-lisp では，リフレクティブ手続きが呼ばれた際に（概念的には）一つメタなレベルのインタプリタに処理を任せる．さらに，計算の任意の時点でリフレクティブ手続きを呼べるためには，(概念的には)任意のレベルのメタインタプリタ，すなわち対象-メタの無限の階層があることになる．これをリフレクティブタワーと呼ぶ．しかも，このタワーの各階層でのメタレベルと対象レベルは因果的に結合 (causally connected) されている．因果的結合とは，対象レベルの状態とそれに対するメタレベルのデータが互いに他を忠実に反映しているという条件である．たとえば，変数 x に束縛されている値が a であるとき，その事実はメタレベルでデータとしてアクセスできる．さらに，メタレベルのデータとして x の値が a から a' に変わった場合，対象レベルでも同様に x の値が a から a' へ変更されていなければならない．すなわち，因果的結合とは対象レベルとメタレベルの間に忠実な表現関係があることを要求しており，Weyhrauch のリフレクション原理（あるいは弱表現可能性）に対応している．ただし因果的結合があくまでもダイナミックな概念，つまり時間的な変化を意識した概念であるのに対し，リフレクション原理はあくまでも論理という静的な世界のものである．しかし，この因果的に結合された無限のタワーというのはあくまでも説明のモデルであって，現実には常に有限の階層しか存在しないということに注意すべきである．

次の例は，変数が束縛されているかどうかを調べる関数 BOUND を定義する 3-lisp のプログラムである．

```
(define BOUND
    (lambda reflect [[var] env cont]
      (if (bound-in-env var env)
          (cont '$T)
          (cont '$F))))
```

lambda の後の reflect が BOUND がリフレクティブ手続きであることを宣言しており，呼び出された時点の環境 (env) と継続 (cont) が引数として取り出される．(bound X) を呼び出すことにより，変数 X がその時点での環境において束縛されているかどうかが分かる．これらの情報を操作することによって，catch/throw をはじめとしてこれまでの言語でプリミティブとして与えられていた多くの関数を言語の中で定義することができる．

計算的リフレクションのメカニズムを説明する概念として，Wand と Friedman はライフィケーション (reification) と (狭義の) リフレクション (reflection) という二つの概念をもち出している[21)]．彼らは lisp の方言である scheme にリフレクションを付加した Brown という言語を用いてライフィケーションとリフレクションを説明している．ライフィケーションとは対象レベルの状態をメタレベルのデータに変換する操作（具体化）であり，リフレクションとはその逆の操作である．しかもそれらの変換は因果的結合の要件を満たすように行われる．3-lisp のリフレクションメカニズムは Brown とは詳細では異なっているものの，有限の階層でリフレクティブタワーを実現する手法は同様のテクニックを用いている．

4. 論理プログラミングとリフレクション

Prolog をはじめとする多くの論理型言語は，効率や記述力を向上させるために本来備わっているべき宣言的な性質を損ねている．実際的なプログラミングをしようとすれば，カット，assert, retract などのデータベース管理述語，var, term などの構文述語，read, write などの入出力述語といった多くの非論理的述語を用いることになる．**3.2** で触れたdemo述語を用いたメタプログラミングにしても，Prolog上でのインプリメントは難しくないが，それが論理的にみてどのように意味付けられるのかは今一つ明らかにされていなかった．この章では，メタ推論とリフレクションについて

の論理的枠組みに基づくいくつかの研究を紹介する.

4.1 メタインタプリタ

Bowen らの提案した demo 述語はホーン論理のメタインタプリタにほかならない. ここでは, Hill と Lloyd のアプローチに従って, 論理プログラミング言語のメタインタプリタについて考察する.

メタインタプリタとはある言語のインタプリタをその言語自身で書いたものであり, 起源的には lisp の万能関数に始まるものである. しかしながらその本質は, すべてのチューリング機械を模倣する万能チューリング機械にあり, これはまさに 2.1 で述べた Gödel の符号化 (名前付け) によるメタレベルの対象レベルへの埋込みを基本とするものである. lisp 1.5 においては, このようなレベル間の分離がある程度忠実に行われていた (これを厳密に行ったのが Smith の 2-lisp であり, さらにこれにリフレクション機能を入れたのが前述の 3-lisp であった). しかしながら論理プログラミング言語においては, 本来レベルを区別する言語要素が備わっていないこと, 効率が重視されていることなどからこれまで対象・メタのレベルが明確にされてこなかった.

論理プログラミング言語のメタインタプリタについて触れる上で, Prolog の3行のメタインタプリタ, いわゆるバニラインタプリタについて述べなくてはならないだろう. バニラインタプリタとは次の Prolog プログラムである[6].

```
solve(true).
solve((A, B))←solve(A), solve(B).
solve(A)←clause(A, B), solve(B).
```

このメタインタプリタは, Prolog の実行を非常に表層的にではあるが正確にシミュレートしている. それがこれほど簡単に記述できる理由は, Prolog の最左深さ優先探索, 単一化, バックトラックなどの機構をメタインタプリタで明示的に記述しているのではなく, メタインタプリタを走らせる Prolog インタプリタの機構を暗黙的に利用しているからである. 特に単一化を暗黙的に利用できるのは, 先に述べたように対象レベルの変数とメタレベルの変数を混在して用いているからである. たとえば, よく引き合いに出される append プログラムをこのメタインタプリタ用に書くと次のようになる.

```
clause(append([ ], X, X), true).
clause(append([Y|Y1], X, [Y|Y2]), append
  (Y1, X, Y2)).
```

ここで, 変数 X, Y はメタインタプリタの変数 A, B と明らかに異なった対象領域を想定している. 前者はリストの世界, 後者はアトムや節を対象とした変数である. また, 上の例においての append は述語記号ではなく関数記号として扱われていると考えられる.

バニラインタプリタのこのような論理的問題を分析するために Hill と Lloyd は多ソート論理を用いた形式化を行った[6]. その体系では, 対象レベルの変数をメタレベルの変数として, また対象レベルの述語記号をメタレベルの関数記号として扱っている. 対象レベルのメタレベルによるこのような表現を, 彼らは型付き表現 (typed representation) と呼んでいる. 否定 (negation as failure) を含んだ型付き表現をもつ論理プログラムによって, 彼らはバニラインタプリタを次のように形式化している. (ただし, x, y はそれぞれメタレベルの変数で, x & y, not(x), x if y はそれぞれ $x, y, \neg x, x \leftarrow y$ のメタレベルでの表現である).

```
solve([ ]).
solve(x & y)←solve(x), solve(y).
solve(not(x))←¬solve(x).
    solve(x)←clause(x if y), solve(y).
```

バニラインタプリタをこのように形式化することによって, Hill と Lloyd はバニラインタプリタの正当性, すなわち対象レベルの証明可能性のバニラインタプリタによる弱表現可能性を示している.

4.2 変数の基底表現

しかしながら, 対象レベルの変数をメタレベルでも変数として扱うことによって, 対象レベルの構文に十分に言及できない場合が出てくる. たとえば, Prolog における非論理的な構文述語の var とか term に対する宣言的な意味を与えることができなくなる. この節以降では, これらの問題を回避し, 論理的意味を明らかにするために, 対象レベルの変数をメタレベルの定数として扱うことにする. この立場は Hill らによって, 前述の型付き表現に対して基底表現 (ground representation) と呼ばれている[6),8]. 変数の基底表現を用いることによって, 対象レベルの構文要素がすべてメタレベルの閉項として扱われることになる. 以下では, 対象レベルの構文要素 s に対して「s」を対応するメタレベルの項とし, s の名前と呼ぶことにする (「 」を quote 記号と呼ぶ).

Hill と Lloyd は基底表現を用いることによって, var や term といった構文述語の宣言的意味を与えている. 彼らは, 基底表現を取り込むための型付きの言

語を与えて，その上で否定を含む論理プログラムを形式化した．そして，型付き表現と同様にして，対象レベルでの証明可能性のメタレベルでの弱表現可能性を示している．var や term などの構文述語は実際的プログラミングにおいても重要な役割をもつものであり，これらに論理的に明確な意味論を与えたことは重要な結果であるといえる．

Hill らのアプローチは基本的にメタインタプリタの形式化を主眼においたものであるが，一方で階層的なデータベースにおけるメタレベル推論の基礎として Genesereth と Nilsson[5] の研究がある．彼らのアプローチは，節形式を対象にしたレゾリューション論理の中に quote 記号を用いた変数の基底表現を付加したものである．2レベルのデータベースにおいて，彼らは次のような形式化を行っている．

2レベルデータベース $Db=(Dbo, Db_M)$ を用意しよう．ここで，Dbo は対象レベルデータベース，Db_M はメタレベルデータベースであり，Db_M は特殊な関数記号として next をもつものとする．Db に対する推論規則は極めて一般的なもので，Db を次のステップへと変形していく手続き NEXT である．関数記号 next が意図されているのは，推論手続き NEXT をメタレベルで記述することであるが，一般にはその二つが食い違うことがある．たとえば，あるデータベース Dbo に対して NEXT によって Dbo' が推論され，一方でメタレベルにおいて next(⌜Dbo⌝)=⌜Dbo''⌝ が含意されるとき，Dbo' と Dbo'' が異なればデータベース全体はある意味で矛盾に陥ることになる．Genesereth らは，これらの関係についていくつかの概念を提案しながら興味深い考察をしている．メタレベルを用いた推論によって対象レベルの論理式 ϕ が証明できるとき，ϕ は内省的に含意 (introspectively imply) されるというが，一般には論理的に含意される論理式がすべて内省的に含意されるとは限らない．これはすなわち，メタレベルで対象レベルの推論を制限できることを示している．逆に，Db が内省的に含意する論理式が論理的帰結であることも保証されない．これは，メタレベルで対象レベルの推論を規定しておくことによって，たとえば非単調推論や類推などへの拡張の可能性を示すものである．

4.3 レベルの融合と計算的リフレクション

まず，Bowen らのアプローチを参考に，対象レベルとメタレベルの融合を考えよう．Bowen らのアプローチには，Hill と Lloyd が慎重に行っている変数の扱いに対して，さほど注意を払っていない点に問題がある．筆者の一人は，この問題に対するもう一つ別のアプローチを採りながら，計算的リフレクションの概念を論理プログラミング言語に取り入れた R-Prolog を提案している[15),16)]．

R-Prolog の言語 L_R はこれまで用いていた quote 記号に加えて，特殊な記号として ↑(up 記号) をもつ．そして項の定義に次の項目を付け加える．

x を変数とすると，$\uparrow x$ は項である．

up 記号は quote 記号と同様，項，アトム，節などについて適用できるよう用法を拡張できる．up 記号は項の名前を動的に生成するために用いられる．$\uparrow X$ を含むアトム $P(\uparrow X)$ を呼び出すことは，その時点で X に束縛されている項 t に対して，$P(⌜t⌝)$ を呼び出すことになる．すなわち，X に束縛されている項の名前が $\uparrow X$ に束縛されると考えることができる．また項に適用した場合の up 記号の意味は，変数を変数として残したまま全体を quote することにあるといえる．たとえば，先に述べた guilty(X) の例を L_R で書くとすると，

```
innocent(X)←person(X),
                relevant(Facts),
              ¬demo(Facts, ↑guilty(X)).
```

また，構文述語 var を R-Prolog の中で扱うとすれば，

$p(X)$←var($\uparrow X$), $q(X)$

のように変数に up 記号を付けて用いればよい．データベースには，任意の変数 X に対して var(⌜X⌝) が宣言されているので，X に変数が束縛されていれば var($\uparrow X$) は成功する．up 記号を用いることによって，R-Prolog ではレベル間にまたがる変数の扱いを問題なく行うことができる．

さて，論理型言語における計算的リフレクションについて考えよう．R-Prolog では計算の途中でリフレクティブな述語を呼び出すことによって，計算の状態にアクセスする．たとえば，計算の状態を変数の束縛情報を記録する環境と外部との通信を行う入出力のストリームからなるとする．このとき，リフレクティブ述語として入力述語 read を次のように定義することができる．

```
read(Var)←read 1(↑Var)
read 1(Var1, Env Env1, Is, Is1)
          ←Is=[T|Is1]
            bind(Env, Var1, T, Env1).
```

このとき，read(X) を呼ぶことによりリフレクティ

ブ述語 read 1(↑X) が呼び出され，さらに5引数の read 1 によって現在の環境 Env と入力ストリーム Is を受け取り，Is のトップにある項と「X」をバインドして新たな環境 Env 1 を作る．ただし，このようなストリームを実現するためには適切な同期の機構をシステムがもたなければならない．

計算の状態としてデータベースを明示化することによって，assert や retract などのデータベース管理述語をリフレクティブ述語として定義したり，ゴールの履歴を明示化することによってループが現れていないかどうかの調査を実行中に行うことができる．このように，メタ推論やリフレクションの概念によって，純 Prolog のような平板な論理プログラミング言語にさまざまな実際的な機能を付加することができる．一般にプログラミング言語の機能を拡張しようとする場合，その意味論が明確なある一定の枠組みに従って行うことによって，意味論の明瞭さを保ちながら拡張を行うことが可能である．計算的リフレクションの概念は，そのような枠組みの有力な候補となる概念である．計算的リフレクションの機構を付加した論理型言語 R-Prolog では，リフレクティブな挙動に明確な意味論を与えることが試みられている[16]．

5. 計算的リフレクションの最近の動向

最近 2〜3 年，リフレクションの機構をより本格的に組み込み，ある程度アプリケーションを意識した言語が提案されるようになってきた．本章では，それらの中から最近の動向として，オブジェクト指向言語 3-KRS[9]，並列オブジェクト指向言語 ABCL/R[22),23)]，並列論理型言語 RGHC[17),18)] の紹介をする．

5.1 オブジェクト指向言語 3-KRS

オブジェクト指向言語においてはすべての計算がオブジェクトを単位として進行する．KRS[20] は Common Lisp 上に構築された，特に知識表現を意識したオブジェクト指向言語で，コンパイルによる効率的実行を考慮せずに，インタプリタ・ベースの動的な特徴を強調している．KRS では，クラス，インスタンス，スロットの値，メッセージ，コード定義，などのすべてがオブジェクトとして統一的に扱われる．インスタンス変数とメソッドの区別がなく，オブジェクトは単にいくつかの名前付けされたスロットをもっていて，その中には他の参照するオブジェクトが入る（インスタンス変数に対応）場合もあれば，スロットを定義するコードが入る（メソッドに対応）場合もある．すなわち，KRS の世界は，多数のオブジェクトがさまざまなラベルで名前付けされたリンクで互いに繋がれた世界として理解することができる．

この KRS にリフレクションの機構を組み込んだのが，3-KRS である．3-KRS では，すべてのオブジェクトの背後にメタオブジェクトがあると考える．この考え方は Smalltalk のクラスに対するメタクラス を拡張したものである．Smalltalk のメタクラスはクラスの生成などの限られた情報しか扱えなかったが，メタオブジェクトはオブジェクトの生成，操作，行動など，オブジェクトの解釈やインプリメントに関するすべての情報を扱うことができる．3-KRS では，それぞれのオブジェクトに対して，システムのもつデフォルトのメタオブジェクトが仮定されるが，ユーザが定義できる特定のメタオブジェクトを陽に指定することもできる．さらに，メタオブジェクトのメタ（すなわちメタメタオブジェクト）も考えることができ，この階層は概念的には無限に続く．なおメタオブジェクトはオブジェクトのメタインタプリタであると考えることができ，デフォルトはオブジェクトの直接実行に対応し，特定のメタオブジェクトが指定された場合は，メタインタプリタによる実行に対応する．

3-KRS におけるリフレクションの単位はオブジェクトであり，オブジェクトごとにリフレクションを行うことができる．リフレクションはデフォルトのメタオブジェクトをユーザ指定のメタオブジェクトに置き換えることにより自動的に行われる（インプリシット・リフレクション）．3-KRS におけるリフレクションの例を以下に示す．

```
FOO
  meta_objct : meta_of_foo_object
...

META_OF_FOO_OBJECT
  type : default_meta_object
  referent : foo
  make_an_instance : a modified function that
          makes an instance of the referent

DEFAULT_META_OBJECT
  make_an_instance : a function to make an
          instance of the object
  respond_to_message(X) : a function to let the
          object respond to a message X
...
```

オブジェクト foo についてリフレクションをおこそうとするとき，まず foo のメタオブジェクトをデフォルトの default_meta_object から，ユーザ定義の meta_of_foo_object に変える．この場合，foo ではメタオブジェクトを，meta_of_foo_object ではオブジェクトを，それぞれ陽に指定する．meta_of_foo_object では，default_meta_object を継承（インヘリタンス）させ，とくに変更したい機能を書きかえる（この場合は make_an_instance）．この例では make_an_-instance を書き換え，クラスからインスタンスを作る際に初期値などの特殊な設定ができるようにしているが，この他にもあるオブジェクトに対して，それからデバッグ情報が取り出せるような特殊なメタオブジェクトを指定したり，またあるメッセージ（メッセージもオブジェクトである）に対して，それに特殊なメタオブジェクトを指定することなども可能である．さらにリフレクションの応用として，継承機構の制御を考えることができる．3-KRS では比較的単純な単一継承の機構しか用意していないが，リフレクションを使用することで多重継承などの複雑な継承機構を実現することが可能になる．このようなリフレクションの機能を活かせば，オブジェクト間のさまざまなリンクをたどるさまざまな複雑な操作が可能となり，さまざまな知識表現のメカニズムを実現することができる．

5.2 並列オブジェクト指向言語 ABCL/R

ABCL[25] は Actor 理論に基づき Common Lisp 上に構築されたオブジェクト指向言語である．KRS と ABCL とは互いによく似ているが，ABCL ではオブジェクトの動作の並列性を強く意識しているところが異なっている．すなわち，ABCL では多数のオブジェクトが並列に動きながら互いにメッセージを交換し，全体として計算を進める．しかしオブジェクトの中では計算は逐次的に進行し，それぞれのオブジェクトは到着したメッセージを順に処理するためのメッセージ・キューをもっている．逐次的な手続き呼び出しとは異なり，メッセージを渡してもコントロールが即座に渡るとは限らず，ABCL では過去型，現在型などのいくつかのメッセージパターンが用意されている．たとえば過去型のメッセージは

[〈target〉⇐〈message〉@〈reply〉]

と記述でき，メッセージ送信のあと返事を待たずにオブジェクトは実行を続ける．〈reply〉はメッセージの答えの送り先であるが，@以下は省略することもできる．また現在型のメッセージは

[〈target〉⇐==〈message〉]

で示される．この場合メッセージを送ることで，このオブジェクトの実行は中断され，〈target〉からなにかの返答が返ってくるまで，オブジェクトは次の動作に移らない．これは通常の手続き呼び出しに似ている．また，この記法はメッセージのやりとりを表すだけでなく，返答の内容自身も表す．

この ABCL にリフレクションの機構を組み込んだのが，ABCL/R である．ABCL/R におけるリフレクションの導入の仕方は，基本的には KRS と同じで，それぞれのオブジェクトに，それをモデル化して解釈実行するメタオブジェクト（メタインタプリタ）を考える．オブジェクトは，メッセージキュー，状態，メソッド，評価器からなると考えられるが，メタオブジェクトはそれらの要素をデータとしてモデル化してもっており，オブジェクトAにメッセージmを送ることは，オブジェクトAのメタオブジェクト↑Aのメッセージキューにmのデータとしての表現を送ることに対応する．

KRS ではメタオブジェクトをユーザがいれかえることにより，リフレクションを起こした．ABCL/R でも同様にインプリシットなリフレクションを起動することは可能であるが，その他に，リフレクティブ手続きとリフレクティブ・メッセージの二つの方法が用意されている．

リフレクティブ手続きは 3-lisp のリフレクティブ手続きと同様なもので，たとえば，変数 X の現在の束縛値を聞く (binding X) は以下のように定義できる．

```
(define (binding(X) env cont eval)
  reflect [eval⇐[ :do X env] @ cont])
```

すなわち，ここで reflect はこの手続きがリフレクティブな手続きであることを示すキーワードで，この定義の中では，オブジェクトの環境，継続，評価器を意味する env, cont, eval の三つの変数が自動的に仮引数として加えられる．ここに示したリフレクティブ手続き binding では，まず :do で，X の env の中での値を問い合わせるメッセージが eval に送られ，その答えは cont に返される．cont はオブジェクトの継続を表しているので，この段階でオブジェクト・レベルの計算が再開することになる．

また，リフレクティブ・メッセージは，従来のメッセージ送信の拡張であり，メッセージのターゲットとして直接メタオブジェクトを指定する．たとえば，評

価器を特定の式

〈expression_to_hook〉でフックするような評価器 Hooking_Evaluator に実行中に取り替えるには，以下のようにすればよい．

```
[↑Me⇐[ : change_evaluator
        [Hooking_Evaluator_Class
          ⇐=[ : new 〈expression_to_hook〉]]]]
```

すなわち，Hooking_Evaluator_Class に，:new というメッセージを，フックされる表現〈expression_to_hook〉と一緒に送り，Hooking_Evaluator_Class のインスタンスとして評価器を作る．そして :change_evaluator という評価器を入れ換えるメッセージを Me のメタオブジェクトである↑Me に送る．

こうしたリフレクティブな機能を提供することによって ABCL/R は少ないプリミティブで強力な言語機構の構成を可能にしている．たとえば ABCL/R では，オブジェクトのメソッドを動的に付加，削除，変更したり，オブジェクトを動的にモニタしたりすることが可能である．また ABCL には，オブジェクトを指定されたメッセージしか受け取らないようなモードに変える wait_for という命令があるが，これもリフレクティブ手続きで記述することが可能である．

5.3 並列論理型言語 RGHC

並列論理型言語とは，論理型言語にプロセスや同期の概念を導入した言語で，Parlog, Concurrent Prolog, GHC[19] などがいままでに提案されている．これらは別名コミット・チョイス型言語とも呼ばれ，論理型言語 Prolog にコミット・オペレータと同期のための機構を導入したものである．これらの言語にはそれぞれ微妙な違いはあるが，本質的にはどれも同じものである．ここではその GHC にリフレクションの機構を付加した RGHC について報告する．

リフレクション機構の付加の方法としては，RGHC で RGHC を記述したメタインタプリタを使用する．RGHC の最も簡単なメタインタプリタ（4.1 のバニラインタプリタに相当）は以下のように記述できる．

```
solve(true) :-true|true.
solve(P, Q) :-true|solve(P), solve(Q).
solve(P) :-not_sys(P)|reduce(P, Body),
          solve(Body).
solve(P) :-sys(P)|P.
```

このインタプリタを徐々に富裕化 (enhance) し，実行過程におけるさまざまな情報をインタプリタの中に段階的に顕在化させる．

たとえば，メタインタプリタの中に，「ゴール実行の成功・失敗」と「並列に動きうるプロセスを管理するプロセス・キュー」を顕在化させると以下のようになる．

```
solve(T, T, R) :-true|R=success.
solve([true|H], T, R) :-true|solve(H, T, R).
solve([false|H], T, R) :-true|R=failure.
solve([P|H], T, R) :-not_sys(P)|
  reduce(P, T, NT), solve(H, NT, R).
solve([P|H], T, R) :-sys(P)|sys_exe(P, T, NT),
  solve(H, NT, R).
```

ここでは solve 述語は solve(H, T, R) の三引数となる．この solve でゴール G を実行するには solve([G|T], T, R) を実行すればよい．solve の最初の二つの引数は差分リストの形でプロセス・キューを表し[11]，またゴール実行の成功・失敗は三番目の引数 R に具体化されている．リフレクションとは，インタプリタの中に表れた過程を制御・修正することであるので，富裕化されたインタプリタほどさまざまな制御・修正が可能となる．この例では「ゴール実行の成功・失敗」と「プロセス・キュー」が顕在化されているので，それらを扱うリフレクティブな命令が定義できる．

富裕化されたメタインタプリタを用いることにより，「ゴールの成功・失敗」などゴール実行の際に生ずる（対象レベル）情報を，自動的に solve 述語の引数を通じて外の世界（メタレベル）に流すことができる．また RGHC はリフレクティブ述語として，メタな世界の情報を取り出す命令（get 命令）と対象世界のデータをメタな世界に戻す命令 (put 命令) をもつ．メタな情報としては，「プロセス・キュー」，「変数環境」，「リダクション数」などを考えており，そのそれぞれについて put 命令，get 命令が定義されている．たとえばユーザは，get_q(H, T), put_q(H. T) という命令を実行することにより，メタな世界のプロセス・キューを差分リストの形式で取り出したり，対象世界のデータをメタな世界に送り上げることができる．

RGHC では，リフレクティブな命令の応用として，オペレーティング・システムの例を考察している．オペレーティング・システムにおいては，メモリや CPU タイムなどのシステムのもつ資源をどのように自己管理するかが重要となり，同時に並列性やメタという概念があらわな形で表れてくる．そうしたシステムの記述にリフレクションが有効であると述べている[17]．

6. おわりに

本稿ではメタ推論とリフレクションについて，論理の枠組みにこだわりつつ，いくつかの観点から解説した．表現可能性を中心にした論理におけるメタの位置付け，Feferman のリフレクション原理，FOL，メタプログラミング，3-lisp におけるメタレベルアーキテクチャ，論理プログラミング言語におけるメタやリフレクションの取扱い，そして計算的リフレクションの最近の発展など，メタやリフレクションに関係するトピックを幅広く取りあげた．

メタやリフレクションについての関心は最近とみに高まっており，昨年にはメタレベル・アーキテクチャとリフレクションに関してイタリアの Alghero で開かれたワークショップの成果をまとめた本も出版された[10]．この本にはリフレクションの最近の研究動向がかなり正確に反映されており，リフレクションに関する研究アクティビティが，基礎理論，インプリメント，応用の三つの分野に分けて紹介されている．

メタ推論とリフレクションに関する今後の研究課題として，次のようなものが挙げられる．基礎理論的研究としては，論理的枠組みを重視する立場，計算的リフレクションと二つの方向性があるが，どちらもまだ端緒についたばかりであり，今後の発展が期待されている．特に，計算的リフレクションに対する理論的基礎についての研究に期待が寄せられている．計算的リフレクションをインプリメントの立場からみた場合，現在提案されているシステムが実際的な要求からはほど遠いという問題がある．本稿で紹介した 3-KRS, ABCL/R, RGHC もインプリメントはされているが，それはあくまで実験的なものであり，効率的インプリメントが今後の課題である．一方，リフレクションを応用という立場からみれば，それは自分自身の情報を得て動的に変更する機構だということである．グラフィック出力やマウス入力などのインタラクティブな入出力の扱い，ロボットなどの実時間システムの動的な制御，自己修正能力を利用しての学習など幅広い分野に応用できると考えられる．今後こうした分野における応用が期待されている．

参 考 文 献

1) Bowen, K. and Kowalski, R.: Amalgamating Language and Metalanguage in Logic Programming, in Logic Programming, pp. 153-172, Academic Press (1982).
2) Bowen, K.: Meta-Level Programming and Knowledge Representation, New Generation Computing, Vol. 3, pp. 359-383 (1985).
3) des Rivières, J. and Levesque, H.: The Consistency of Syntactical Treatments of Knowledge, in Theoretical Aspects of Reasoning about Knowledge, pp. 115-130, Morgan Kaufmann (1986).
4) Feferman, S.: Transfinite Recursive Progressions of Axiomatic Theories, J. Symbolic Logic, Vol. 27, No. 3, pp. 259-316 (1962).
5) Genesereth, M. R. and Nilsson, N. J.: Logical Foundations of Artificial Intelligence, Morgan Kaufman (1987).
6) Hill, P. M. and Lloyd, J. W.: Analysis of Meta-Programs, in Proc. of the Workshop on Meta-Programming in Logic Programming (META 88), University of Bristol, pp. 27-42 (1988).
7) 國藤 進，北上 始，宮地泰造，古川康一：知識工学の基礎と応用 (第4回)—Prolog における知識ベースの管理—，計測と制御，Vol. 24, No. 6 (1985).
8) Lloyd, J. W.: Directions for Meta-Programming, in Proc. of FGCS '88, ICOT, pp. 609-617 (1988).
9) Maes, P.: Reflection in an Object-Oriented Language, in Preprints of the Workshop on Meta-Level Architecture and Reflection, Alghro-Sardinia (1986).
10) Maes, P. and Nardi, D. (eds.): Meta-Level Architecture and Reflection North-Holland (1988).
11) Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, ICOT (1983).
12) Smith, B. C.: Reflection and Semantics in a Procedural Language, MIT Laboratory for Computer Science, TR-272 (1982).
13) Smith, B.C.: Reflection and Semantics in Lisp, in Proc. 11 th ACM Symposium on Principles of Programming Languages, pp. 23-35 (1984).
14) Smorynski, C.: Self-Reference and Modal Logic, Springer-Verlag (1985).
15) 菅野博晴：リフレクティブな Prolog の形式化と意味論，信学技報，Vol. 88, No. 312, pp. 41-48 (1988).
16) Sugano, H.: A Formalization of Reflection in Logic Programming, in preparation (1989).
17) Tanaka, J.: A Simple Programming System Written in GHC and Its Reflective Operations, in Proc. of Logic Programming Conference '88, ICOT, pp. 143-149 (1988).
18) Tanaka, J.: Meta-interpreters and Reflective Operations in GHC, in Proc. of FGCS '88,

ICOT, pp. 774-783 (1988).
19) Ueda, K.: Guarded Horn Clauses, ICOT Technical Report, TR-103, ICOT (1985).
20) Van Marcke, K.: The Use and Implementation of the Representation Language KRS, technical report 88-2, Vrije Universiteit Brussels (1988).
21) Wand, M. and Friedman, D. P.: The Mystery of the Tower revealed: A Non-Reflective Description of the Reflective Tower, in.
22) 渡部卓雄, 米澤明憲: Towards Reflection in an Object-Oriented Concurrent Language, 日本ソフトウェア科学会第四回大会論文集, pp. 33-9 342 (1987).
23) Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language, in Proc. of OOPSLA, ACM, pp. 306-315 (1988).
24) Weyhrauch, R. W.: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, Vol. 13, pp. 133-170 (1980).
25) 米澤明憲, 柴山悦哉, J.-P. Briot, 本田康晃, 高田敏弘：オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1, コンピュータ・ソフトウェア, Vol. 3, No. 3, pp. 9-23 (1986).

(平成元年4月27日受付)