論理型言語における ユニフィケーションの拡張と その応用

柴山悦哉
（東京工業大学　理学部）

概要

Prologのプログラムはinput resolution（unification + backtrack 付き三段論法）により実行される。したがって、Prologの拡張としては次の３つの可能性が考えられる。

(1) ユニフィケーションを拡張する。

(2) バックトラック付の三段論法を拡張する。

(3) input resolution以外のメカニズムを新しく取り入れる。

Prologを自然に拡張するためには、(3) より(1) または(2) の方法をとる方が望ましい。本稿では、(1) の方法により、Prologの記述力を高める試みについて述べる。

はじめに

Prologは二つの意味論（宣言的意味と操作的意味）を持っており、我々はPrologのプログラムに対して二通りの解釈を与えることができる。プログラムを宣言的に解釈できる場合には、Prologは比較的わかりやすい言語である。これは、Prologの基礎をなす一階術語論理が、コンピュータで実行することを意図して作られた言語体系ではなく、人間の考えを簡潔に表現するために作られた言語体系だからである。ところが、操作的にPrologのプログラムを解釈しようとすると大変なことになってしまう。Prologの実行原理であるresolution principleは本来、定理自動証明のための手法であり、その動作は人間より機械にとってわかりやすくできている。人間の短期記憶の容量は非常に小さく、backtrack point を何箇所も持つプログラムの実行を追うことはできない。Prologのプログラムは、静的になかめて意味をくみとれる時には比較的読みやすいが、動的に解釈するのは大変な代物であるということができる。

ところが、プログラムを静的に解釈する場合にも、全く問題がないわけではない。人間は思考の途中でよく同一視を行なう。最も単純な場合、人間は 2+3と 5を同一視するし、時には猿と人間を同じ霊長類に属するという理由で同一視する。この同一視という現象はある種のユニフィケーションと考えることができるであろう。ところが、Prologの世界では 2+3と 5は決してユニファイされないし、猿と人間がユニファイされることもない。Prologは等号やis_a関係のない logicに基づいているのであるから、これは当然といえる。無意識的に同一視を行なう我々人間の世界観と、機械的なユニフィケーションに依存するPrologの世界観には大きな隔たりがある。

本稿では、Prologのパターンマッチの限界について考察し、この限界がユニフィケーションを拡張することによって克服できることを示す。さらに、拡張されたユニフィケーションにより、関数風の表記法、抽象データ型、デーモンもどき、lazy execution、インヘリタンス等が実現できることを示す。尚、以下のプログラム例はDec-10 Prolog [10]のシンタックスに準拠する。

1. Prologのパターンの記述力

ユニフィケーションの機能により、Prologは双方向性のパターンマッチを実現している。Prologでは任意の項をパターンとみなすことができ、２つの項はユニファイ可能な時マッチする。以下、Prologの項のことをパターンと呼ぶことがある。

パターンマッチの利点は記述の簡潔さにある。例１としてPrologによる普通のmemberの定義とパターンを全く使わないmemberの定義を掲げる。これらを比較すると、パターンマッチの威力がよくわかるであろう。

ただし、例１(b) においてcar(L,X)はL のcar 部がX であれば成功し、cdr(L,L1) はL のcdr 部がL1なら成功するものとする。

例１(a)　通常のmemberの定義

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

(b) パターンを使わないmemberの定義

```
member(X,L) :- car(L,X).
member(X,L) :- cdr(L,L1),member(X,L1).
```

術語の引数としてパターンが書けることがPrologの最大の特徴の一つであり、この機能によりプログラムを簡潔に記述することができる。例1(b)のように術語の引数がすべて変数であるプログラムは最悪であると言えよう。

例１(a) からもわかるように、Prologでリスト処理を行なう場合、lispのcar, cdr, consに相当する働きはパターンマッチによって実現できる。ところが、lispのmember, assoc, append 等に相当する働きはパターンマッチだけでは実現できない。例えば、" 先頭がfoo である任意のリストとマッチするパターン "は[foo|_] のように記述できるが、" foo という要素を持つ任意のリストとマッチするパターン "はPrologでは記述できない(*) 。また、パターンで記述できないわけではないが、現実的に記述することが困難な場合もある。例えば、" 長さ1000の任意のリストとマッチするパターン "は、[_1,_2,_3,...,_1000]のように記述できるが、実際に書くのは骨の折れる仕事である。

例２

(a)　Prologのパターンで容易に記述できるもの

先頭がfoo であるリスト	[foo|_]
先頭がHeadで残りがRestであるリスト	[Head|Rest]
長さ３のリスト	[_,_,_]
長さ３以上のリスト	[_,_,_|_]

(b)　Prologのパターンで記述しづらいもの

1000番目の要素がfoo であるリスト	[_1,_2,...,_999,foo|_]
1000番目と2000番目の要素が等しいリスト	[_1,_2,...,_999,X,_1001,...,_1999,X|_]

(c)　Prologのパターンで記述できないもの

長さ10以下のリスト
先頭がfoo またはbar であるリスト

例２からもわかるように、Prologのパターンの記述力は十分に強力であるとはいえない。例3(a)はmemberを用いた術語 assocの定義である。もし、mem(X)が"Xを含む任意のリスト" とマッチするようなパターンであるなら、assoc の定義は例3(b)のように書けるはずである。一般には例3(a)より例3(b)の方が簡潔な表記なのだが、通常のPrologではこのように記述することはできない。

(*)　項記述[8] のテクニックを用いるとL:member(foo,L) というパターンが fooという要素を持つ任意のリストとマッチする。同様のテクニックは[4] でも論じられている。

例3(a)

```
assoc(Alist,Key,Value) :-
    member([Key|Value],Alist).
```

例3(b)

```
assoc(mem([Key|Value]),Key,Value).
```

定義

パターンPに対して、Match(P) = { t | t はPとマッチするground term } と定義する。

ここで問題点を少し形式的に述べてみよう。まず functorからなる適当な有限集合をFとし、Fから構成される Herbrand universeをHとする。この時、帰納的可算なHの任意の部分集合Sに対して、Match(P) = Sとなるパターンが存在すればパターンの記述力は完全であるといえるであろう。ところが、Prologでは、Hの部分集合Sが{ t | t は fooを要素とするリスト }のように明らかに帰納的な場合でも、Match(P) = Sとなるパターンが存在しないことがある。ここにPrologの問題点がある。

2. Prologのパターンの拡張

1章の例1,3 からもわかるように、パターンを使うとプログラムの記述が簡潔になる場合が多い。ところが、Prologのパターンの記述力が弱いために、その使用には制限がつけられる。この問題は、Robinsonのアルゴリズム[9] を用いている限り解消できない。そこで、この章ではPrologのユニフィケーション機能を拡張することによってパターンの記述力を高める方法について述べる。

ここでは、プログラム中にパターンの意味を記述するclause(以下、equality clause と呼ぶ) を書けるようにし、ユニファイアは常にequality clause を参照しながらユニフィケーションを行なうものとする。以下、ユニフィケーションを拡張したPrologを通常のPrologと区別する必要がある時には、前者をE-Prologと呼ぶことにする。例4(a)は、" X を要素に持つ任意のリスト" とマッチするパターンmem(X)を定義するequality clause である。

例4(a)

```
mem(X) = [X|_].
mem(X) = [_|mem(X)].
```

一番目のequality clause は、先頭の要素がX であるリストはmem(X)とマッチすると読む。また、二番目のequality clause は、cdr 部がmem(X)とマッチするリストはmem(X)とマッチすると読む。形式的には、mem(X)の意味はMatch(mem(X)) = Match([X|_]) U Match([_|mem(X)])という等式で定義できる。equality clause に対する操作的意味は3章で述べる。

パターンmem(X)を用いて先程は術語 assocを定義したが、例4(b)のように assoc自体もパターンとして定義できる。

例4(b)

```
assoc(key,Value) = mem([Key|Value]).
```

この定義は、assoc(Key,Value)というパターンはmem([Key|Value])というパターンとマッチするような任意のリストとマッチすると読む。直感的には、assoc(Key,Value)はキー Keyを持ちその値が Valueである連想リストとマッチする。また形式的には、Match(assoc(Key,Value)) = Match(mem([Key,Value])) という等式で定義できる。

例4(c)はorとand に相当するパターンの定義である。

例4(c)

```
X::Y = X.                    (X & Y) = Z :- X = Z, Y = Z.
X::Y = Y.
```

まず左の２つのequality clause により、”X または Yとマッチする任意の項”とマッチするパターンとして、X::Yが定義される。例えば、[a|_]::[_,a|_]というパターンは一番目または二番目の要素が aであるリストとマッチする。また、右のequality clause は、X が Zとマッチし Yが Zとマッチする時、X & Y というパターンは Zとマッチすると読む。すなわち、X & Y というパターンは Xとも Yともマッチする項とマッチする。

今までに定義したパターンを使ったプログラムの例を示そう。例5(a)は連想リストで表現された個人データベースである。また例5(b),(c)は、この個人データベースに対する質問を、通常のPrologとE-Prologで記述し対比したものである。

例５(a)　個人データベース

```
person([[name,seiko],[height,156],[job,singer],[sex,female]]).
person([[namae,etsuya],[job,koumuin],[sex,male],[height,173]]).
person([[height,166],[name,yasuhiko],[sex,male],[job,student]]).
```

例5(b)　質問１：　データベースに登録された人名は？

```
?- person(P), (assoc(P,name,N) ; assoc(P,namae,N)).        /* Prolog */

?- person(assoc(name::namae,N)).                           /* E-Prolog */
```

例5(c)　質問２：　人名と職業の組を知りたい。

```
?- person(P), (assoc(P,name,N) ; assoc(P,namae,N)), assoc(P,job,J).        /* Prolog */

?- person(assoc(name::namae,N)&assoc(job,J)).                              /* E-Prolog */
```

このようにユニフィケーションを拡張することによりPrologのプログラムをより簡潔に記述することができる。

3.　拡張ユニフィケーションの実現

本稿の最初でも述べたようにPrologの実行はユニフィケーションとバックトラック付き三段論法によって行なわれる。ユニフィケーションのアルゴリズムはequality clause の導入によって変更を受けるが、バックトラック付三段論法の方を変更する必要はない。そこで、この章では拡張ユニフィケーションのアルゴリズムについてのみ言及する。E-Prologでは、term1 = term2 という形のequality clause を等式とはみずに、term1 -> term2という形の項書き換えシステムと考える。例えば、mem(X)と[a,b,c] のユニフィケーションの際には、mem(X) = [X|_]というequality clause を使ってmem(X)を[X|_] に置き換えてから[a,b,c] とのユニフィケーションを行なう。このequality clause を逆方向に用いて[a,b,c] をmem(a)で置き換えたりはしない。

定義

(1) T1 = T2 :- Gという形のequality clause に対し、T1 = T2 の部分を頭部(head)と呼び、 Gをゴールと呼ぶ。

(2) あるfunctor Fが、あるequality clause の頭部の左辺のトップレベルに現われる時、Fのことをactiveな functorと呼ぶ。また、activeでない functorをpassive な functorと呼ぶ。同様に、activeな functorをトップレベルに持つ項をactiveな項と呼び、そうでないものをpassive な項と呼ぶ。

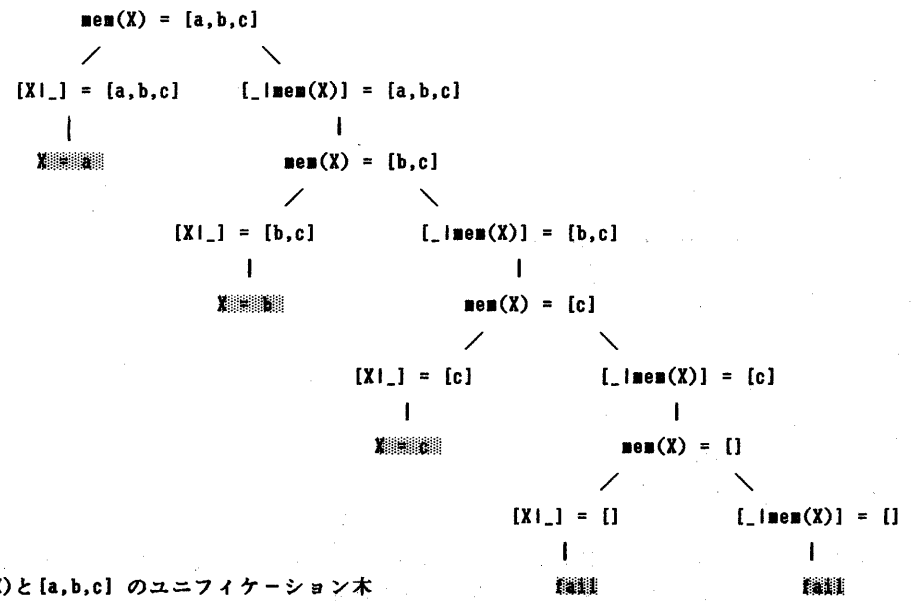
以下に、拡張ユニフィケーションのアルゴリズムをPrologのプログラム風に与える。

アルゴリズム

```
(1) e_unify(T1,T2) :- (var(T1) ; var(T2)), !, T1 = T2.
(2) e_unify(T1,T2) :- passive(T1), passive(T2), !, e_unify_args(T1,T2).
(3) e_unify(T1,T2) :- active(T1), !, reduct(T1,T3), e_unify(T3,T2).
(4) e_unify(T1,T2) :- reduct(T2,T3), e_unify(T3,T1).
    reduct(T1,T2) :- get_equality_clause((T=T2 :- G)), e_unify_args(T1,T), call(G).
    e_unify_args(T1,T2) :-  T1=..[F|Args1], T2=..[F|Args2], e_unify_list(Args1,Args2).
    e_unify_list([],[]).
    e_unify_list([X|X1],[Y|Y1]) :- e_unify(X,Y), e_unify_list(X1,Y1).
```

すなわち、(1) ２つの項のうちどちらかが変数なら、その変数は相手の値で束縛される。(2) ２つの項が共にpassive なら、トップレベルの functorが等しことを調べ、対応する引数どうしをユニファイする。(3) T1がactiveなら、equality clause を用いてT1を置き換え、これとT2をユニファイする。もし、T1を置き換えるのに使えるequality clause が２つ以上存在すれば、バックトラックを繰り返しながら、すべての可能性を確かめる。(4) それ以外の場合、T1がpassive でT2がactiveになるから、(3) とほぼ同様にすればよい。

次の図は、二つの項mem(X)と[a,b,c] に対する拡張ユニフィケーションの探索木である。この場合 Xに a,b,cがバックトラックしながら次々に束縛されていく。



mem(X)と[a,b,c] のユニフィケーション木

4. 関数風の表記法

拡張ユイフィケーションを用いると、[4],[11]でも議論されているように関数風の表記法が使えるようになる。例えば、リスプのappend, reverse に相当するものは例6(a)のように定義できる。

例6(a)

```
append([],X).                          reverse([],[]).
append([A|X],Y) = [A|append(X,Y)].     reverse([A|X]) = append(reverse(X),[A]).
```

この定義によって、append(append([1],[2,3]),reverse([4,5]))のような項をプログラム中に書くことができるようになる。また例6(b)の質問を発すると、結合すると[1,2,3,4] になるリストの組を非決定的にすべて求めることができる。

例6(b)

```
?- append(X,Y) = [1,2,3,4].
```

ここで定義したappend, reverse は関数風ではあるが、あくまでパターンであると筆者は考える。Prologに関数風の表記法を導入するのはいいが、関数自体を導入してしまうと、primitive の数が増えてしまい形式的な議論を困難にするだけである。

5. パターン抽象

パターンを用いて抽象データ型を定義することもできる。抽象データ型の値を操作するために、Iota[7], Clu[6]ではfunctionやprocedure を用いている。また、Prologで抽象データ型を実現しているHimiko[2] では、このために術語を用いる。ここではまず、Himikoによるスタックの定義例を掲げ、次に同じことを術語ではなくパターンを用いて行なってみる。

Himikoで抽象データ型を定義する時には、その型の外側で使える術語の宣言部分(interface part)と各術語の実現部分(realization part)が必要になる。

例7(a)

```
module stack
  interface
    type stack
    rel  empty(<stack>)
         push(<element>,<stack>,<stack>)
         pop(<stack>,<element>,<stack>)
  realization
    repr = <list>
    clause  empty([]).
            push(X,S,[X|S]).
            pop([X|S],X,S).
end-of-module
```

モジュール定義の前半では、スタックを外部から操作するためにempty, push, popという三つの術語用意されていることが記され、さらにこれらの術語の定義域も指定される。また後半では、スタックをリストで表

現することと、三つの術語の実現部分が記述されている。　ところが、このような方法で抽象データ型を定義するとPrologのパターンマッチの機能が有効に使えなくなる。すなわち、push,pop等の術語を呼び出す方ではスタックの内部表現は全く見えないわけだから、これらの術語の引数にパターンを書くことができなくなる(*)。この問題はパターンで抽象データ型を定義する時には解消される。次にパターンを用いたスタックの定義を記述する(構文はHimikoを少し変更したものである)。

例7(b)

```
module stack
  interface
    type stack
    pattern  empty : <stack>
             push(<element>,<stack>) : <stack>
             pop(<stack>,<element>) : <stack>
  realization
    repr = <list>
    clause  empty = [].
            push(X,S) = [X|S].
            pop([X|S],S) = X.
end-of-module
```

こうすれば、呼び出し側でもpush(1,push(2,empty))のようなパターンを書くことができるようになる。また、抽象データ型の特徴が発揮されるので、スタックの実現部分が変更されてもスタックを使う側を変更する必要はない。尚、スタックの場合パターンに対してモード指定[1] ができた方が現実的であろう。

スタック同様にキューなら次の３つのパターンで実現できる。

例8(a)

```
empty = [].
enq(El,Que) = Que1 :- append(Q,[El],Q1).
deq([El|Que]) = Q.
```

キューを実現するパターンの選択には任意性がある。パターンを、第一引数にメッセージを受け取り第二引数に結果を返すオブジェクトと考えると次の様に書くこともできる。

例8(b)

```
queue_super(create) = [].
queue_object(enq(X),Y) = Z :- append(Z,[X],Y).
queue_object(deq(X),[X|Y]) = Y.
```

Prologやリスプはtypelessだから、抽象データ型との相性が悪いという説がある。しかし実際には、これらの言語には型が全くないのではなく、唯一つしか型がないのである。新しい型が定義されると型が一つ増

(*)　実際のHimikoでは、項を一つの抽象データ型と考えることができるので、一部の引数にはパターンを書くことができる。

えるだけのことである。

6. デーモン風プログラム

2章の個人データベースでは、assoc(weight,W) というパターンはweightというキーを持つ連想リストとのみマッチした。もし、assoc の定義を例9のように書き換えれば、assoc(weight,W) というパターンは、相手の連想リストがweightというキーを持たなくてもheightというキーを持てばマッチし、 Wはheightの値から 110を引いたもので束縛されることになる。ここで、"！"はカットオペレータであり最初のequlaity clause が失敗した時のみ二番目のclauseが使われる。

例9

```
assoc(Key,Value) = mem([Key,Value]) :- !.
assoc(weight,W) = assoc(height,H) :- plus(W,110,H).
```

これはデフォルト推論の簡単な場合であるり、デーモンを用いると自然に書ける。デーモンを実現するためには、あるデータが読み書きされる時に特定のルーチンを起動するメカニズムがあればよい。したがってPrologでは、ユニフィケーションの途中である特定のルーチンが起動できればデーモンを実現できるわけだが、通常のPrologではこのようなことは不可能である。E-Prologではequality clause 中のゴールやパターンがユニフィケーションの途中に起動されるので、デーモンもどきのプログラムを書くことができる。ただし、E-Prologはデーモンを支援する特別の構文を持たないから、例9の二行目は、意味的にはデーモンであっても構文上はデーモンに見えない。

7. lazy execution

3章で述べた拡張ユニフィケーションのアルゴリズムに従うと、不完全なかたちながらlazy executionの機能を実現できる。例えば、リスプのappendに相当するパターンを4章のように定義した場合の実行例を以下に示す。

例10

```
?- append([1,2],[3,4]) = X.
  X = append([1,2],[3,4])

?- append([1,2],[3,4]) = [X|Y].
  X = 1
  Y = append([2],[3,4])

?- append([1,2],[3,4]) = [X,Y|Z].
  X = 1
  Y = 2
  Z = append([],[3,4]).
```

3章のアルゴリズムによると、変数以外の項とユニファイされない限りパターンがequality clause による置き換えをうけることはないので、lazy executionに似たメカニズムが実現できる。この機能をうまく使うと無限長の項を扱える場合がある。以下に、自然数の列を表わす項の定義とその実行例を示す。

例11

```
int(N) = [N|int(add1(N))].          /* equality clause */

?- int(0) = [X|Y].
  X = 0
  Y = int(add1(0))

?- int(0) = [X,Y|Z].
  X = 0
  Y = add1(0)
  Z = int(add1(add1(0)))
```

8. インヘリタンス

拡張ユニフィケーションをうまく使うとインヘリタンスに近い機能を実現できる場合がある。例12(a) は、penguin に対してcan_fly という術語を定義し、birdに対してname, age, can_flyという術語を定義している。

例12(a)

```
name(bird(Name,_),Name).          can_fly(penguin(_,_),no).
age(bird(_,Age),Age).
can_fly(bird(_,_),yes).
```

これだけでは、penguin(piyo,10)という項に対してcan_fly は適用できるが、name, age は適用できない。そこで次のequality clause を定義すると、penguin(piyo,10)にたいして、name, age が適用可能となる。

```
bird(Name,Age) = penguin(Name,Age).
```

このequality clause はis_a関係を定義していると考えることができる。ただし、この方法を無制限に用いると大変なことになる。例えば、例12(b) のようなequality clause が1000個もあれば、birdとpenguin のユニフィケーションは非常に効率の悪いものとなろう。また、例12(c) のようにpenguin 自体がactiveなfunctor になっている場合もユニフィケーションの効率は低下する。

例12(b)

```
bird(Name,Age) = pigeon(Name,Age).
bird(Name,Age) = ostrich(Name,Age).
        :
        :
bird(Name,Age) = penguin(Name,Age).
```

例12(c)

```
penguin(Name,Age) =
  emperor_penguin(Name,Age).
      :
      :
      :
```

この問題を解決するために、equality clause の特別なものとしてis_a clause を導入する。ユニファイヤはis_a clause をequality clause とほぼ同様に扱う。ただし、is_a clause はゴール中の項を置き換える時にのみ用い、clauseの頭部に表われる項を置き換えるためには使えないものとする。次のものはis_a clause の例である。

例12(d)

```
pigeon(Name,Age) is_a bird(Name,Age).
ostrich(Name,Age) is_a bird(Name,Age).
penguin(Name,Age) is_a bird(Name,Age).

emperor_penguin(Name,Age) is_a penguin(Name,Age).
```

ここで注意すべきは、equality clause で左辺にあったものがis_a clause では右辺にあり、equality clause で右辺にあったものがis_a clause では左辺にある点である。すなわち、equality clause では頭部の左辺の方が右辺より generalな項であったが、is_a clause ではよりspecificな項になっている。したがって、is_a clause によるインヘリタンスは、ゴール中にある項の方がclauseの頭部にある項よりspecificな場合だけ利用できる(*)。そのかわり、むだな探索に要する手間が大幅にカットされる。

9. 問題点

拡張ユニフィケーションを導入すると、多様な機能をProlog上で実現することができ、またプログラムが簡潔に記述できるようになる。その反面、ただでさえ人間に理解しずらいPrologの実行過程が今まで以上に複雑になる。適切なデバッグ手法の確立が今後の重要な研究課題となるであろう。

参考文献

1. Bowen, D., Byrd, L., Pereira, F., Pereira, L., and Warren, D.: "DECsystem-10 Prolog User's Manual (version 3.47)", Dept. of Artificial Intelligence, University of Edinburgh (1982)
2. Furukawa, K., Nakajima, R., and Yonezawa, A.: Modularization and Abstraction in Logic Programming, New Generation Computing, Vol. 1, No. 2, pp 169-177 (1983)
3. Goldberg, A. and Robson, D.: "Smalltalk-80, the Language and its Implementation", Addison-Wesley (1983)
4. Kornfeld, W.: "Equality for Prolog", Proc. of IJCAI-VIII, pp. 514-519 (1983)
5. Kahn, K.: "Uniform -- A Language Based upon Unification which Unifies (Much of) Lisp, Prolog and Act 1", Proc. of IJCAI-VII, pp. 933-939 (1981)
6. Liskov, B., Russell, A., Bloom, T., Moss, E., Schaffert, C., Scheifler, B., and Snyder, A.: "CLU reference manual", Lecture Notes in Computer Science 114, Springer-Verlag (1980)
7. Nakajima, R. and Yuasa, T., Hagino, T., Honda, M., Koga, A., and Shibayama, E.: "The IOTA Programming System", Lecture Note in Computer Sciences 160, Springer-Verlag (1983)
8. 中島秀之: "項記述", Proc. of the Logic Programming Conference '84, 2-3, (1984)
9. Robinson, J.: "A Machine-Oriented Logic Based on the Resolution Principle", JACM, Vol. 12, No. 1, pp. 23-41 (1965)
10. Sato, M. and Sakurai, T.: "Qute: A Prolog/Lisp Type Language for Logic Programming", Proc. of IJCAI-VIII, pp. 507-513 (1983)
11. 渡辺治: "Prolog における functorの役割について", 情報処理学会ソフトウェア基礎論研究会資料 7-1 (1984)

(*) Smalltalk[3] 等で実現されているインヘリタンスもこのようなものだけである。