Prolog における functor の役割について

渡辺 治 （東京工業大学・理学部・情報科学科）

1. はじめに

Prolog プログラムの実行は、 resolution principle に基づいて行われる*。 ところが、 resolution principle は本来、 定理(論理式)証明のための手法であり、 プログラム実行のための手法ではない。 したがって「・・・ が成り立つことの証明」には十分であっても、 「・・・ を満たす値を求める」には不十分な点がある。 このことが、Prolog における functor の役割を中途半端なものにしている。 そして functor が日の目を見ないことが、Prolog プログラムの書きにくさの一因になっていると思う。

本稿では、 resolution(resolution principle に基づく実行方法)のどの点がプログラム実行に不適当なのか、 それが Prolog プログラムの書きにくさにどう影響するのかを中心に述べる。 また、 どのような解決法があるのかについても簡単に説明する。

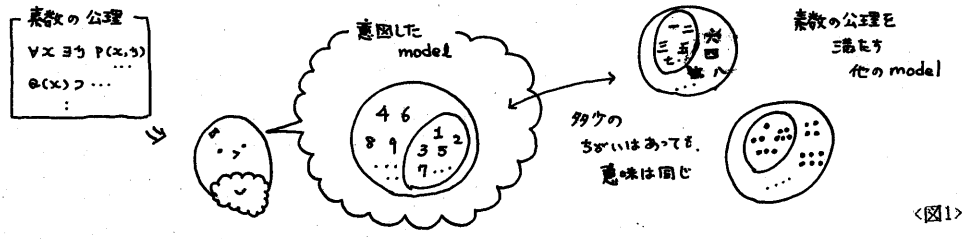
2. Resolution の限界と、 functor が function になれない理由

ここでは、 プログラム実行手段に resolution を用いたとき生じる問題点を、 簡単な例 Love affair を中心に説明する。 そしてそのことが、 Prolog というプログラミング言語の仕様に及ぼす影響を考える。

Logic Programming のイメージ

Love affair の例に入る前に、 Logic Programming における resolution の役割を明確にしておこう。 そもそも、なぜ Logic でプログラムが書けるのだろうか? どうして論理式の集りがプログラムになるのだろう? この問いに対するイメージ的な答えを紹介しよう。 Resolution の役割は、 そのイメージ的な説明の中で明らかになるだろう。

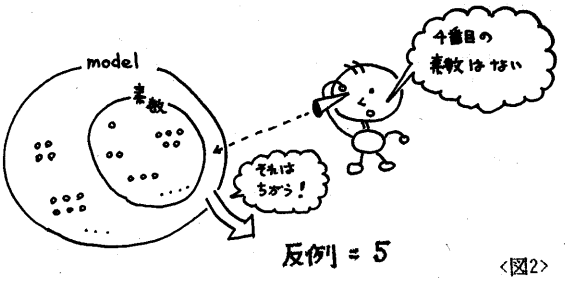
従来のプログラムでは、 欲しい出力(計算結果)の求め方を記述する。 たとえば「n番目の素数の求め方」を書く。 それに対し論理式によって「n番目の素数の性質(公理)」を記述するのが Logic Programming のやり方である。 その公理が正しい(無矛盾)ならば、 その公理が成り立つ世界(model)が存在する。 もちろん、 model は一意には決まらない。しかし公理が十分であれば、対応する model はすべて我々の意図する世界と本質的には異ならない(図1)。



<図1>

(*) 本稿で述べている事とは別の観点から resolution の問題点を指摘し、 他の方法による Prolog の実行を考えている人々もいる([1])。

出力は、 その model を観察することにより得ることができる。 具体的には「条件を満たす出力はない」といった仮定を与えて、それに対する反応をみるのである。 実際には出力があるはずなのだから、 その仮定は反証される。 その反例が、 求めたい出力である(図2)。



<図2>

公理により世界を構成することは、一階の述語論理で行うことができ、 観察は resolution により実現できる。 Resolution は、仮定が公理に矛盾することを示すことにより背理法的に証明を行う手法だ。 しかも、 証明の過程で反例を得ることができる。 したがって、我々は resolution を観察の手段として使えるのである。

Resolution の限界

以上が Logic Programming に対して私の持っているイメージである。もちろん「出力の求め方」を記述したほうが楽な問題も多い。 しかし「求め方」の記述は複雑だが、 「性質」の記述は楽にできる問題には Logic Programming が適していると思う。 私は、 Love affair がそのような問題のよい例であると思った。

Love affair はパズルである(図3)。 これを解くプログラムを作るのは少々面倒だと思う。 しかし問題の状況は、 論理式で簡単に書くことができる(図4)。 「これこそ Logic Programming の威力を示すよい例だ!」と思った。 しかし resolution では、 欲しい答えを得られなかった。

<図3>

Love affair

ゆうこ、みゆき、ともこ の3人の女性と、おさむ、やすお、かずお の3人の男性が無人島に流れ着いた。自然の摂理から、しばらくすると各男性は女性を、各女性は男性を、各々一人づつ好きになった。 ただし、

みゆきが好きな人は、ゆうこが好きで、
やすおが好きな人は、かずおが好きで、
ともこが好きな人が好きな人は、おさむが好きで、
ゆうこはおさむが好きだった。

では、誰がゆうこを好きになっただろう?

<図4>

$\forall x \exists y [Female(x) \supset Male(y) \wedge Love(x, y)].$

$\forall x \exists y [Male(x) \supset Female(y) \wedge Love(x, y)].$

$\forall x [Love(みゆき, x) \supset Love(x, ゆうこ)].$

$\forall x [Love(やすお, x) \supset Love(x, かずお)].$

$\forall x \forall y [Love(ともこ, x) \wedge Love(x, y) \supset Love(y, おさむ)].$

Love(ゆうこ,おさむ).

Male(おさむ) ∧ ・・・・・.

Female(ゆうこ) ∧ ・・・・・.

(注) これらの論理式は、すべて ∧ で結ばれている。ここには、1人が2人以上を好きにならないという事は記述されていないが、 図5では implicit に記述されている。

実際に resolution で答えを求めてみよう。 図4を resolution のやりやすい形になおしたのが図5である。 ここで f は、∃y を表現するために導入された Skolem function である。 f(x) の意味は、 「x が好きな人」と考えることができる。 さて我々は Love(x, ゆうこ) なる x を求めたい。 そこで(2)式、 すなわち「誰もゆうこを好きでない」を図5

<図5>

公理

Male(f(x)) ∧ Love(x, f(x)) ← Female(x).

Female(f(x)) ∧ Love(x, f(x)) ← Male(x).

Love(f(みゆき), ゆうこ). --- (1)

Love(f(やすお), かずお).

Love(f(f(ともこ)), おさむ).

Love(ゆうこ, おさむ).

Male(おさむ) ∧ ・・・・

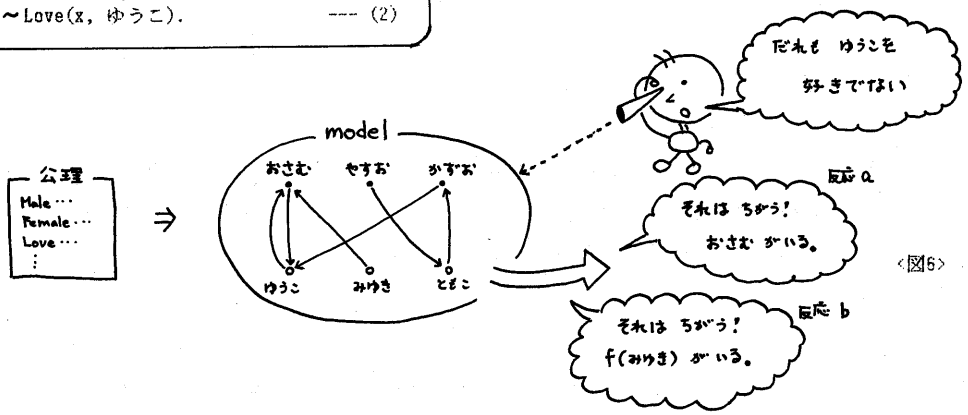
Female(ゆうこ) ∧ ・・・・

ただし、 f は Skolem function。

Resolution の仮定

～Love(x, ゆうこ). --- (2)

の公理系に加えて、 resolution により矛盾を導く。 たしかに、 (1)と(2)の両式より矛盾が導ける。 しかし f(みゆき) という反例しか得られない。何が悪いのか? 実は何も悪くない。 「誰がゆうこを好きか」という問いに対して、 「f(みゆき)、 すなわち、 みゆきが好きな人」と答えても間違いではない。 しかも、 そのような人は存在するのだから、 (2)式は反証されたことになる。 にもかかわらず我々は不満である。 我々は図6-aのような反応を期待していたのだが、 実際には図6-bのような答えしか返ってこなかったからだ。 なぜこのようなことが起きたのだろう。



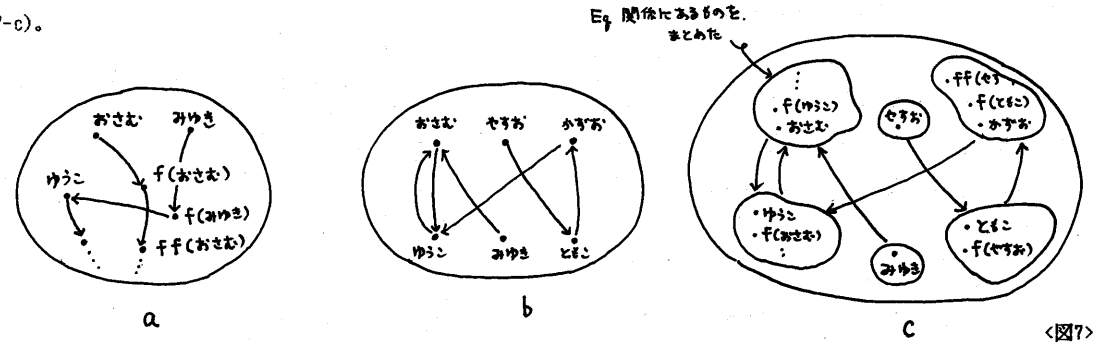
<図6>

Resolution の model は、 Herbrant universe の上に作られる。 Love affair の公理に対する model は図7-aのようになり、 我々の意図した model (図7-b)とは大きくくい違う。 これは「登場人物・・・ しかいない」ということを言い忘れていたからである。 そこで

Eq(x, おさむ) ∧ Eq(x, やすお) ∧ Eq(x, かずお) ∧

Eq(s, ゆうこ) ∧ Eq(x, みゆき) ∧ Eq(x, ともこ). ---(3)

という公理を図5の公理系に加えれば、 Herbrant universe 上の model も我々の意図する model とほぼ同じになる(図7-c)。



<図7>

ところが、たとえ(3)式を公理に加えても、 resolution による結果は同じだ。 Resolution で(2)式から矛盾を導き出すのに、 (3)式は必要ないからである。図7(c)のようなmodelを得るには、 すべての述語のすべてのgrand instanceに対して真偽を決めねばならない。 ところが Love(x, ゆうこ) を反証するには、 Love(f(みゆき), ゆうこ) が真であることがわかればよい。 Resolution では、そのことだけを調べるため Love(f(みゆき), ゆうこ) が真であることは判明するが、 Eq(f(みゆき), おさむ) が成立するかどうかには関知しないのである。

Prolog への影響

結局 resolution では f(みゆき) が何であるかがわからないまま、 証明過程が終了してしまう。 つまり、 resolution では f(みゆき) の値を求められない。 Prolog でも同じだ。 Prolog では functor を使って、 たとえば P(f(b), g(b, c)) と書くことができる(g, f が functor、 P は predicate)。 しかし resolution では f(b) や g(c, d) の値を求めることができないため、 それらは形の上の意味しか持たない。 つまり Prolog では functor は データ構造を表現する道具としてしか使われていない。

3. Functor の値が求められないため生じる不都合

§2で説明したように、 resolution では functor の値を求めることができない。 そのため Prolog では、 functor は通常の function のように使われず、 ただ構造を表現する道具としてしか使われていない。 このことは、 プログラム の書きにくさの原因になっていると思う。 その例をいくつか考えてみよう。

アセンブラ的プログラミングになる

たとえば、 $1+(1/x)^n$ を y に求める問題を考えてみよう(図8)。普通の Prolog では、 プログラムは(1)または(2)になるだろう。 これらでは「1/x を求めて u にしまう」、「 u を求めて v にしまう」、「1+v を求めて y にしまう」といった手順で y を求めている。 これはアセンブラ的プログラミングだと思う。 Function が Prolog で使えないために、このようなプログラミング・スタイルになってしまったのだ。 もし +, /, exp という function が使えるのならば、 (3)のようにスマートに書ける。

「Prolog でも + や / のように、 exp という primitive operator が用意されていれば、 (4)のように書くことができる」と言われるかも知れない。 でも、 必要となる function をすべて primitive として用意することは不可能である。 また、「数値計算を考えるから悪いのだ」と思われるかも知れない。 しかし、 数値を扱う場合以外でも同様な問題を考えることができる(図9)。

図8

y := 1 + (1/x) のプログラム*

```
(1) Comp(x, n, y)
          ←— Div(1, x, u),
             Exp(u, n, v),
             Plus(v, 1, y).

(2) Comp(x, n, y)
          ←— Is(u, 1/x),
             Exp(u, n, v),
             Is(y, 1+v).

(3) Comp(x, n, 1+exp(1/x, n))  ←— .

(4) Comp(x, n, y)
          ←—Is(y, 1+exp(1/x, n)).
```

(*) 本稿では、n, u, v, ・・・ で変数を、a, b, c, ・・・ で定数を表わす ことにする。

z := x ∪ y のプログラム

従来の方法	Functionを使うと
Union(x, y, z) ← Intersection(x, y, u), Delete(x, u, v), Append(v, y, z).	Union(x, y, x - (x ∩ y) + y) ← .

<図9>

文字列を扱いにくい

通常の Prolog を拡張して、 文字列を扱うことを考えよう。 リストを Prolog で扱うことができるのは、たとえば [a, b, c] は、 cons(a, cons(b, cons(c, nil))) の省略形であると考えているからだ。 そのおかげで、 [a, b, c] と cons(a, x) を unify することができるのである。 それと同様に、 文字列を扱うためには、 たとえば 'abcd' は、 concat(a, concat(b, concat(c, d))) の省略形だと考えねばならない。 こうすることにより、 concat(a, x) を 'abcd' と unify することができ、 x に 'bcd' をとってくることができる。 しかしリストとちがって文字列の処理では、 concat('ab', x) も 'abcd' に unify することができ、 その結果 x に 'cd' をとってくることができると考えたい。 ところが、

concat('ab', x) = concat(concat(a, b), x),

'abcd' = concat(a, concat(b, concat(c, d))),

であり、 Prolog ではこれら2つを unify することはできない。

文字列を扱うもう一つの方法は、 システム提供の述語(primitive predicate)を使うことである。 たとえば図10のような文字列処理用の述語が用意されていれば、 それを使って文字列処理を行うことができる。 しかしその場合、 プログラムにおける述語の引数には変数しか登場しなくなり、 unification は単なるパラメータ受け渡しのメカニズムになってしまう(図11)。そのため、 Prolog の良さがかなり失われてしまうと思う。

<図10>

文字列処理用の primitive predicate の例

```
Concat(x, y, z):  xy を z へ求める。
Length(x, n):     x の 長さを n に求める。
Left(x, n, y):    x の n 文字目より左の文字列を y へ求める。
Right(x, n, y):   x の n 文字目から右の文字列を y へ求める。
Mid(x, n, m, y):  x の n 文字目から m 文字ぶんを y へ 求める。
Find(x, y, n):    x 中に y が出てきたら、 その位置を n に求める。 出てこなければ、 fail.
```

<図11>

文字列 x 中のパターン v をすべて w に換えたものを y に求める。

```
Replace(x, v, w, y)
 ←— Find(x, v, n),
      Left(x, n, x1), Concat(x1, w, y1)
      Length(w, m), Right(x, n + m, x2), Replace(x2, v, w, y2),
      Concat(y1, y2, y).

Replace(x, v, w, x) ←— .
```

述語の各引数はすべて変数なのでパターン・マッチングの必要がない。 つまり、 unification は call by reference と同じになる。

新しいデータ型との相性が悪い

文字列を扱いにくいという問題は、もう少し一般的に考えることができる。Prolog で構造をもったデータを作るには、functor を用いる。ただし、functor は function としてではなくデータ構造を表現する道具として使われている。つまり Pascal のレコード型におけるフィールド名やタグと同じような意味で使われているのである。Functor により作られるデータ型を総称して term 型と呼ぶことにしよう。この term 型に対する操作は unification によって行われるが、それが非常にうまく行われている。それが Prolog の魅力の一つだと思う。簡単な例で考えてみよう。

二分木の反転を行う場合を例にとり、Prolog のデータ型操作方法を Pascal と対比させながら書いてみた(図12)。手続き reverse では、x に与えられたデータが realtree であるか atom であるかを判定しなければならない。Prolog では、t(u, v) に unify するかどうか ── 一番目の clause を選ぶか二番目を選ぶか ──によって realtree と atom の区別ができる。また realtree 型のデータに対してその子供をとってくる必要があるが、Prolog では t(u, v) に unify した時点で u と v に子供の木が代入(bind)されている。さらに反転した木を構成しなくてはならない

<図12>

例: 二分木の反転

二分木を x に与えると、それを反転したものを y に返す手続き reverse を考える。

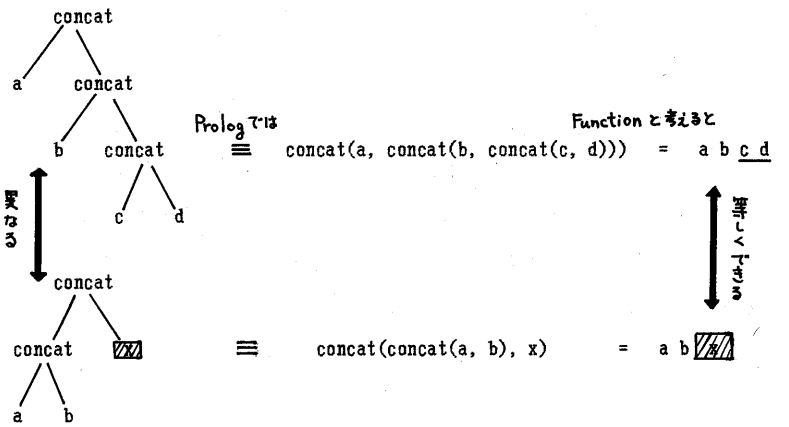
Prolog	Pascal
手続き	
Reverse(t(u, v), t(v', u')) ⟵ Reverse(v, v'), Reverse(u, u'). Reverse(u, u) ⟵.	proc reverse(x: tree, var y: tree); . . . .
データ型	
realtree: t(・, ・) の形をしているもの atom: その他	tree: record case tag: int of 0: (t: realtree); 1: (a: atom) end; realtree: record l: tree; r: tree end; atom: 適当な型
データ型の操作 (Reverse(x, y) という呼び出しに対して)	
1. realtree か atom の判定	
realtree ⇔ x が t(u, v) と unify する	realtree ⇔ x.tag = 1
2. x の子供の木を取り出す	
x が t(u, v) と unify したあとで、 u: 一番目の子供 v: 二番目の子供	u := x.t.l v := x.t.r
3. 新しい木を y に構成する	
y を t(v', u') と unify させる	y.t.l := v' y.t.r := u' y.tag := 0
(注) これら3つのことは、Reverse(x, y) と Reverse(t(u, v), t(v', u')) の 1回の unification で行うことができる。	

が、 Prolog では t(v', u') が y に unify することにより新しい二分木が作られ y へしまわれる(bind される)。 つまりデータ型に対する基本操作、 1.データ型の判別、 2.データ型の構成要素の取り出し、 3.あるデータ型データの構成、 をすべて unification で行うことができるのである。 しかもそれらが 1回の unification でできることが多い(図12: 注)。 Term 型というデータ型を unification によって統一的にしかも簡単に扱うところに、 Prolog の Prolog らしさがあると思う。

ところがこの考え方は文字列型など、 他のデータ型との相性が悪い。 新しいデータ型を Prolog に取り入れる場合には、 (1) そのデータ型を term 型に合わせ unification で操作できるようにする(例: 'abc' を concat(a, concat(b, c)) とみなす), (2) そのデータ型を操作するための述語(primitive predicate)を用意する(例: 図11)、の2通りの方法が考えられる。 しかし文字列の例でみてきたように、 この2つの方法には次のような問題点がある。 (1)の場合には、 新しいデータ型の表現方法や要素の取り出し方が term 型と同じに決められてしまい、 新しいデータ型特有の扱いが不可能になる(例: concat('ab', x) が 'abcd' と unify できない)。 また(2)の場合には、 述語(primitive predicate)がそのデータ型に対する操作を行うため、 unification によりデータ型を操作するという Prolog のよさが失われてしまう(例: 図11)。

以上の問題点は、 functorをデータ構造表現の道具とみなしているから、 つまり functor が形上の意味しか持たないから生じるのである。 Functor はあるデータ構造を持ったデータを作り出す function であると考えると、 これらの問題は生じない。 図13にその違いを示した。 今の Prolog では concat(a, concat(b, concat(c, d))) そのものがデータの表現である。 したがって concat('ab', x) には match しない。 ところが concat(x, y) は文字列 x と y をつなげた文字列を求める function だと考えると、 x = 'cd' のときに 2つが等しくなる —— match する —— ということができる。 これはほんの一例だが、 このように考えると term 型以外のデータ型も term 型と同じような形で(unification と似た方法で)操作できるだろう。

<図13>



共通変数が欲しくなる

Functor が形の上の意味しか持たないことの、 もう一つの影響は ∃ (存在記号)が思ったように使えないということである。 Resolutionにおいて、∃ を表現するために Skolem function を用いた。 たとえば Love game での f(x) がそうである。 しかし functor が形の上の意味しか持たないので、 functor を使って ∃ を表現したとしてもプログラ

ムとして見た場合には、 意味がない。 つまりプログラムでは事実上 ∃ が書けない。

論理式において ∃ が書けないとなると大問題であるが、 プログラミングにおいてはどうだろう? Prolog プログラムを作る上で、 論理式における ∃x に相当するものを使いたくなることがある。 2つの clause で共通な変数が、 あるいは非局所的な変数が欲しいなぁという気持が生じる場合がそうである。 現在の Prolog では、 assert や retract を使って非局所的データを扱っているが、 好ましい方法とはいえない。 この問題も functor を function とみなすことができると、 ある程度解決できると思う。

4. 解決方法について

§3で述べたような Prolog プログラムの書きにくさは多くの人々が感じている点であり、 それに対する対策もいろいろと考えられている。 私の知る範囲では、 Prolog/KR の実行可能パターン、 Himiko における virtual unification、 Uniform の拡張 unification、 Prolog/Lisp 型言語 Qute などのアプローチがおもしろいと思う(順に[2, 3, 4, 5])。 それらの解決法のうち「functor の意味はその形ではなく function のようにその値である」という考え方に基づく方法がもっとも自然であると思う(先にあげたものの中では Uniform、 Qute がその方針を採用している)。 Functor を function と見るという考え方は、 §3において何度か説明してきた。 またそうすることにより、 先に述べた問題点が解決できそうだということも示してきた。 ここではこの考え方をもう少し明確にし、 それを実現するにはどうしたらよいかについて述べる。 またその場合には、 どのような問題点があるかについても簡単にふれたい。

Functor を function とみなすには Prolog をどう拡張したらよいか

Functor を function のように考えるというのは、 正確に言うとどういうことだろうか。 たとえば f(a) の値は 'f(a)' という term 型のデータであり、 f はそのような term 型のデータを求める function と考えてもよい。 すべての functor はこのタイプの function だと思えば、 今の Prolog でも functor を function と見ていることになる。 しかし fact(3) の値は、 'fact(3)' ではなく 6 であると考えたい。 これが、 ここでいう functor をfunction とみなすということである。 つまり fact という functor は 3 に対して 6 を返す function であるということを規定しておくこと——functor の 定義——ができ、 fact(3) が出てきた時はその規定にしたがって fact(3) は 6 だとみなすこと——functorの評価——ができるということが、 functor を function として使うということである。 この考え方は、 今まで述べてきたように resolution では実現できない(resolution では、 functor f はすべて 'f(・・・)' を返す function と定義されているようなものだ)。 したがってこの考え方の実現のためには、 それを賄う機構を現在の Prolog に付け加えなければならない。 すなわち、 次のような拡張が必要である。

(1) Functor の定義

Functor がどのような function であるかを、 どこかで定義しておかなければならない。

(2) Unification の拡張

Unification を行う時に、 term 中の functor の値を評価しなければならない(unification というのは term 型のデータ間の matching であるから、 functor の値同志 の unification という言い方は正しくないかも知れない)。

(2)の表現は、まだあいまいである。たとえば functor の評価を Lisp のように考えると、unification を拡張しても図13のようなことができない。そのため unification のよさ、Prolog らしさがこの拡張システムでは反映できなくなる。ここでは次のような unification の拡張を考えている(これだと図13のようなことができる)。

(2') 拡張unification (semantic unification と呼ばれているらしい)

2つの term の拡張 unification とは、それらを functor の定義にしたがって評価した場合に等しくなるように各変数の値や対応関係を決めることである。

Prolog システムに(1),(2')の拡張を加えることにより functor を function とみなすことができる。また逆に言うと、(1),(2')のことができるということが、functor を function とみなすということの正確な定義である。

Prolog を拡張する時の問題点

Prolog システムを(1),(2')のように拡張する場合に、いくつか考えねばならない点がある。それを簡単に述べてみよう。

Functor を定義する場合に、いつ・どこで・どのように定義すればよいかが問題になる。一つのデータ構造とそれらを扱う functor をひとまとめにして定義するという案が考えられる。こうすると Prolog の module 化ができるだろう(module 化については、[3]に述べられている)。それとは対称的に、Lisp のような dynamic な定義を考えるならば functor を共通変数のようにも使うことができるようになる。

次に拡張 unification について考えよう。(2')で述べたような unification は、従来の unification の使いやすさを保存したまま functor を function とみなすことができるという点で、非常に自然な拡張だと思う。しかし、これにはいくつか問題がある。たとえば(2')の定義通りに考えると、fact(x) と 6 を unify させて x に 3 を得ることができる。しかし一般にこのようなことは不可能である。また、たとえできたとしても効率面で実用的でないことが多いだろう。(2')の考え方をどこまで制限すれば、効率と使いやすさの両面で妥当なものになるだろうか。その境界線を決める必要がある。また unification では most general unifier が一意に決まるが、拡張 unification では必ずしも一つになるとはかぎらない。その場合どれを選択したらよいかの基準を決めねばならない。

プログラミング・スタイルについて

最後に、functor を function として使えた場合のプログラミング・スタイルについて少し述べておきたい。たとえば階乗を求める場合、Fact という述語を使う方法と、fact という functor を使う方法の2つができることになる。だからといって functor を function とみるのは余計なことだ、ということにはならないだろう。それよりむしろ、現在の Prolog のように階乗まで resolution により求めるのはやりすぎだと考えるべきだと思う。Function 的なものに向いている仕事と、Prolog 的な形で処理するのに向いている仕事があり、場合に応じて使いわけるのが好ましいプログラミング・スタイルではないだろうか。私はデータ構造の基本的操作や一般の算術演算には function のほうが向いており、それ以上の仕事、あるいはそれらを組み合わせた仕事には述語のほうが向いているように思う。

5. おわりに

本稿の目的は、多くの人々が感じていると思われる Prolog プログラミングのいくつかの問題点を「resolurion はプログラム実行手段としては不完全だ」という観点から説明し、さらにその立場からそれらの問題点の解決案を示すことである。そのためにまず、プログラム実行の手段として見た場合、resolution は function の値を求められないという問題点をもっていることを示した。そして、それが原因となっている Prolog プログラミングの問題点をいくつかあげた。最後にそれらの解決法として、functor を function とみなす方法を述べ、その実現について考えた。

最後に、本稿を作るにあたりいろいろと貴重な御意見、御批判をして下さった東工大・情報科学科の米澤先生、浅川氏，横内氏に感謝致します。また、本稿を清書するにあたりお世話になった東工大・木村研究室の方々にも謝意を表します。

参考文献

1. T. Sakurai, On foundation of Prolog, Proc. the Logic Programming Conference '83, Tokyo (1983).
2. 中島秀之, Prolog/KR の概要, 情報処理学会記号処理研究会資料 18-5 (1982).
3. A. Yonezawa, K. Furukawa and R. Nakajima, Modularization and abstraction in Logic programming, Research Report C-55, Dept. of Information Science, Tokyo Inst. of Tech. (1983).
4. K. Kahn, The implementation of Uniform: A knowledge-representation/programming language based upon equivalence of description, Research Report of UPMAIL, Dept. of Computer Science, Univ. of Uppsala (1981).
5. M. Sato and T. Sakurai, Qute: A Prolog/Lisp type language for Logic programming, Technical Report 83-10, Dept. of Information Science, Univ. of Tokyo (1983).
6. C. Chang and R. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, NY (1973).
7. W. Clocksin and C. Mellish, Programming in Prolog, Springer-Verlag, NY (1981).