『論理と形式化』資料 No.8
ホーン節と論理プログラミング

亀山幸義 (kam@cs.tsukuba.ac.jp)

一階述語論理は強力な表現力を持ち、Computer Science に現れる様々な論理的言明を、正確かつ精密に表現することができる。しかし、表現力がありすぎて、その論理式を「取り扱う」のは簡単ではないときがある。(「取り扱い」とは、自動証明などを指す。)

そこで、一階述語論理の論理式を、特定の形に制限して処理しやすくすることが考えられる。命題論理に対する論理積標準形等は、そのような特定の形の 1 つである。

制限するので一般性に欠ける話になってしまう、と思う人がいるかもしれないが、実際にはそうではなく、ここで紹介するホーン節によるプログラミング言語 (Prolog) は、計算能力に関して十分 (Turing 完全である) ことが知られている。

1 ホーン節

ホーン節 (Horn clause) は、以下の特別な形をした一階述語論理式のことである。

$$\forall x_1 \cdots \forall x_n((A_1 \wedge A_2 \wedge \cdots \wedge A_m) \supset B)$$

ただし、$m, n \geq 0$ であり、$A_1, \cdots, A_m, B$ は、すべて、原子論理式 (つまり、k 引数の述語記号 p と、項 $t_1, \cdots, t_k$ を使って $p(t_1, \cdots, t_k)$ という形になっている論理式) である。

上記の式で、$x_1, \cdots, x_n$ は、$A_1, A_2, \cdots, A_m, B$ に出現する全ての変数を適当な順番で並べたものである。

$A_1, \cdots, A_m, B$ に変数が 1 つも含まれなければ $n = 0$ となる。$m = 0$ の場合も許されており、その場合、本体部分は B のみとなる。

ホーン節は以下の点で制限が加わっている。

- 全称記号が一番外側のみに現れ、存在記号は一切現れない。
- 全称記号の内側の論理式が $(A_1 \wedge \cdots \wedge A_m) \supset B$ の形に限定されている。
- $A_1, \cdots, A_m, B$ はすべて原子論理式である。

例 1 (ホーン節の例)**.** この例における言語は以下のものとする。

- 定数: alice, bob, chris, emma, fritz
- 関数記号: なし
- 述語記号: isParent, isGrandP(いずれも引数が 2 個、つまり、アリティ 2 とする)

以下のものは、上記言語におけるホーン節の例である。

- isParent(alice,bob) (論理式の意味: aliceは bobの親である。)
- isParent(bob,chris)
- isParent(bob,emma)
- isGrandP(bob,fritz)

- $\forall X \forall Y \forall Z (\text{isParent}(X, Y) \wedge \text{isParent}(Y, Z) \supset \text{isGrandP}(X, Z))$ (論理式の意味: X が Y の親で、Y が Z の親なら、X は Z の祖父母である。)

都合上、変数の名前は大文字で始まり、定数、関数記号、述語記号の名前は小文字で始まるものとする。つまり X と書けば変数であり、x と書けば定数・関数記号・述語記号のいずれかである。

これらのホーン節を仮定して、一階述語論理の推論規則を使うと、isGrandP(alice,chris) が証明できる。また、

$\forall X$ (isParent(chris, X) $\supset$ isGrandP(chris, X)) が証明できる。

例 2 (ホーン節の例)**.** 言語は以下のものとする。

- 定数: 東京, 埼玉, 栃木, 茨城, 福島.
- 関数記号: なし。
- 述語記号: adj, reach(いずれもアリティ 2)。

以下のものは、上記言語におけるホーン節の例である。

- adj(東京, 茨城). (意味: 東京は茨城と隣接している)
- adj(東京, 埼玉).
- adj(埼玉, 栃木).
- adj(茨城, 埼玉).
- adj(茨城, 栃木).
- adj(茨城, 福島).
- adj(栃木, 福島).
- $\forall X \forall Y$ $(\text{adj}(X, Y) \supset \text{reach}(X, Y))$. (意味: 隣接していれば、到達可能である。)
- $\forall X \forall Y \forall Z$ $(\text{adj}(X, Y) \wedge \text{reach}(Y, Z) \supset \text{reach}(X, Z))$.

これらから、たとえば、reach(東京, 福島) を導くことができる。

例 3 (ホーン節の例)**.** 言語は以下のものとする。

- 定数: 0.
- 関数記号: s.
- 述語記号: add(アリティ 3)。

以下のものは、上記言語におけるホーン節の例である。

- $\forall Y$ $\text{add}(0, Y, Y)$.
- $\forall X \forall Y \forall Z$ $(\text{add}(X, Y, Z) \supset \text{add}(s(X), Y, s(Z)))$.

これらから、たとえば、$\text{add}(s(s(0)), s(s(s(0))), s(s(s(s(s(0))))))$ を導くことができる。これは、$2 + 3 = 5$ を意味している。

例 4 (ホーン節の例)**.** 言語は以下のものとする。

- 定数: 0.

- 関数記号: s (アリティ 1).
- 述語記号: add, times(以上、アリティ 3).

以下のものは、上記言語におけるホーン節の例である。

- $\forall Y\ \mathrm{times}(0, Y, 0)$.
- $\forall X \forall Y \forall W \forall Z\ (\mathrm{times}(X, Y, W) \wedge \mathrm{add}(W, Y, Z) \supset \mathrm{times}(s(X), Y, Z))$.

例 3 のホーン節と合わせると、たとえば、$\mathrm{times}(s(s(0)), s(s(s(0))), s(s(s(s(s(0))))))$ を導くことができる。これは、$2 * 3 = 6$ を意味している。

2 ホーン節に対する推論

ホーン節に限定したことによるメリットは、効率的に証明できることである。
準備: 一階述語論理の推論体系で、$(A_1 \wedge A_2) \supset B$ と A_1 と A_2 から B を導ける。

$$\dfrac{(A_1 \wedge A_2) \supset B \quad \dfrac{A_1 \quad A_2}{A_1 \wedge A_2}}{B}$$

これを一般化すると、$(A_1 \wedge \cdots \wedge A_m) \supset B$ と $A_1, \cdots, A_m$ から B を導けることがわかる。そこで、以下では、この導出を 1 回の推論でできると仮定する。

例 1:

$$\dfrac{\mathrm{isParent}(\mathrm{alice}, \mathrm{bob}) \quad \mathrm{isParent}(\mathrm{bob}, \mathrm{chris}) \quad \dfrac{\forall x \forall y \forall z (\mathrm{isParent}(x, y) \wedge \mathrm{isParent}(y, z) \supset \mathrm{isGrandP}(x, z))}{\mathrm{isParent}(\mathrm{alice}, \mathrm{bob}) \wedge \mathrm{isParent}(\mathrm{bob}, \mathrm{chris}) \supset \mathrm{isGrandP}(\mathrm{alice}, \mathrm{chris})}}{\mathrm{isGrandP}(\mathrm{alice}, \mathrm{chris})}$$

例 2. (「東京」を「東」とする等の省略を用いる。)

$$\dfrac{\mathrm{adj}(\text{東}, \text{茨}) \quad \dfrac{\mathrm{adj}(\text{茨}, \text{福}) \quad \dfrac{\forall X \forall Y (\mathrm{adj}(X, Y) \supset \mathrm{reach}(X, Y))}{\mathrm{adj}(\text{茨}, \text{福}) \supset \mathrm{reach}(\text{茨}, \text{福})}}{\mathrm{reach}(\text{茨}, \text{福})} \quad \dfrac{\forall X \forall Y \forall Z\ (\mathrm{adj}(X, Y) \wedge \mathrm{reach}(Y, Z) \supset \mathrm{reach}(X, Z))}{\mathrm{adj}(\text{東}, \text{茨}) \wedge \mathrm{reach}(\text{茨}, \text{福}) \supset \mathrm{reach}(\text{東}, \text{福})}}{\mathrm{reach}(\text{東}, \text{福})}$$

例 3. ($s(s(0))$ を「2」と省略した。)

$$\dfrac{\dfrac{}{\forall Y\ \mathrm{times}(0, Y, 0)}}{\mathrm{times}(0, 2, 0)}$$

例 4. ($s(s(0))$ を「2」と省略した。)

$$\dfrac{\dfrac{\dfrac{\forall Y\ \mathrm{add}(0, Y, Y)}{\mathrm{add}(0, 3, 3)} \quad \dfrac{\forall X \forall Y \forall Z\ (\mathrm{add}(X, Y, Z) \supset \mathrm{add}(s(X), Y, s(Z)))}{\mathrm{add}(0, 3, 3) \supset \mathrm{add}(1, 3, 4)}}{\mathrm{add}(1, 3, 4)} \quad \dfrac{\forall X \forall Y \forall Z\ (\mathrm{add}(X, Y, Z) \supset \mathrm{add}(s(X), Y, s(Z)))}{\mathrm{add}(1, 3, 4) \supset \mathrm{add}(2, 3, 5)}}{\mathrm{add}(2, 3, 5)}$$

これらの形式推論を見ていると、共通するパターンが見えてこないだろうか？

- ホーン節の集合から原子論理式 A を導くためには、2 つのパターンがあり、(1) A 自身が、与えられたホーン節の中にあるか、ホーン節の変数を具体化したものである。例: $\mathrm{isParent}(\mathrm{alice}, \mathrm{bob})$ や $\mathrm{times}(0,2,0)$
- (2) $B_1 \wedge \cdots \wedge B_n \supset A$ という形の論理式と、原子論理式 $B_1, \cdots, B_n$ から、A を導いている。
- この場合、$B_1 \wedge \cdots \wedge B_n \supset A$ は、ホーン節として与えられた $\forall X_1 \cdots \forall X_m (C_1 \wedge \cdots \wedge C_n \supset D)$ において、$X_1, \cdots, X_m$ を具体的な項にして得られたものである。
- 残る $B_1, \cdots, B_n$ は、原子論理式なので、もう 1 度上記パターンを使って導く。

事実: ホーン節から導ける原子論理式は、必ず、上記の形の推論の繰返しで導くことができる。

3 論理プログラミング

「計算の実行=推論過程」と考えるのが、論理プログラミング (論理型プログラミング, logic programming) である。このうち、特に、ホーン節に対する推論をそのままプログラミング言語として実現したのが、Prolog [Kowalski 1973] である。(「論理プログラミング」というのは、本来は、論理を使ったプログラミング形態の総称であるので、非常に幅広い言葉であるはずだが、今日、「論理プログラミング言語」といえば、Prolog や Prolog の拡張のみを指すことが多い。)

ここでは、Prolog の処理系を使って、論理プログラミングを楽しむことにする。

3.1 Prolog のプログラムとゴール

Prolog では、ホーン節を見やすくするため、以下のように表す。

$$B :- A_1, A_2, \cdots, A_m.$$

これは、$\forall x_1 \cdots \forall x_n (A_1 \wedge A_2 \wedge \cdots \wedge A_m \supset B)$ という ホーン節を表す。ここで、$x_1, \cdots, x_n$ は、$B, A_1, \cdots, A_m$ に出現する、すべての変数を適当な順番で並べたものである。

表記の上での変更をまとめると以下の通りである。

- :- という、「ならば」の左右をひっくりかえした論理記号を使う。(つまり、$A \supset B$ を B:-A とあらわす。)
- $\wedge$ のかわりに「コンマ」で区切って書く。
- 全称記号を書かない。(A_i や B に出現する変数をすべて $\forall$ で束縛して並べるだけなので、書かなくても復元できる。)
- 最後に「ピリオド」を書く。
- $m = 0$ の場合は、 :- も書かない。($B.$ と書く。)

例 5 (Prolog におけるホーン節の表記)**.** 前述の例の ホーン節を Prolog の表記法であらわしてみると以下のようになる。

- isParent(alice,bob).
- isGrandP(X,Z) :- isParent(X,Y), isParent(Y,Z).
- reach(X,Y) :- adj(X,Y).

- reach(X,Z) :- adj(X,Y), reach(Y,Z).

一階述語論理の論理式としての表記より、だいぶすっきりとした事がわかるであろう。

さて、Prolog のプログラムは、いくつかのホーン節から構成される。これらは、Prolog における推論において「仮定する事実および推論規則」(それらが成立するという仮定のもとで、何らかの事実を導きだしたい) を表す。これらを用いて、何らかの有意義な結論を推論によって導きたいのであるが、それを、Prolog ではゴールと呼ばれ、以下の形とする。

$$?- G_1, G_2, \cdots, G_k.$$

ここで ?– は意味のある記号ではなく、これがゴールであることを表しているだけである。

上記における各 G_i は、原子論理式であり、意味としては、「与えられた Prolog プログラムから、$G_1 \wedge G_2 \wedge \cdots \wedge G_k$ が導けるか」ということを聞いている。導けるとき、YES で、導けないとき NO である。

上記は、単純な例を書いたが、一般には、G_i は変数を含む。その場合は、G_i たちにあらわれる変数全部をならべて $y_1, \cdots, y_l$ とすると、

「与えられた Prolog プログラムから、$\exists y_1 \cdots \exists y_l$ $(G_1 \wedge G_2 \wedge \cdots \wedge G_k)$ が導けるか」

という質問となる。そのような $y_1, \cdots, y_l$ が存在すれば答えは YES であり、存在しなければ答えは NO である。

4 Prolog 処理系： SWI-Prolog

SWI-Prolog 処理系では、プログラムは xxx.pl という名前のファイルにいれるとよい。上記の add, times のプログラムは、以下のような内容として、test.pl というファイルにいれることにする。

```
add(0,Y,Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
times(0,Y,0).
times(s(X),Y,Z) :- times(X,Y,W), add(W,Y,Z).
```

3 行目の times の定義で、変数 Y は 1 回しか出てこないため、実質的には使われない変数である。親切な SWI-Prolog 処理系は、そのことで警告をだしてくるが、無視してよいが、警告が気になる人は、3 行目を、

```
times(0,_,0).
```

というように、下線 (アンダースコア) を使うとよい。これは、「使わない変数」を意味する。

処理系の起動は、swipl とするだけである。

```
% swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.4)
...
?- [test].
true.
```

ここで [test]. とタイプしたのは、ファイル test.pl からプログラムを読み込め、という意味である。(最後のピリオドを忘れないこと。)

プログラムを読みこんだだけでは、Prolog の処理系は動かない。以下の形のものをクエリ (質問; Query) として入力すると動きだす。

```
?- add(0,s(0),Y).
Y = s(0).
```

このように入力すると、「$\exists Y\ add(0, s(0), Y)$ となるか」という質問をしたことになり、答えは、yes である。ただし、Prolog 処理系は、単に yes/no ではなく、yes の場合は、実際に存在する Y の値まで求めてくれる。この場合、$Y = s(0)$ という返事がかえってきている。なお、クエリでの限量子が、$\forall$ でなく、$\exists$ であることに注意されたい。

クエリは、Prolog 処理系にとっては「それを証明する目標」となるものであり、「ゴール (goal)」とも呼ばれる。

上記のクエリの場合は、Y となるものが 1 つしかなかったのでそれで終了したが、解が複数ある場合は、次の解を要求するかどうかをユーザが入力する必要がある。たとえば、

```
human(alice).
human(bob).
human(chris).
```

というプログラムをファイル test.pl にいれて

```
?- [test].
% test compiled 0.00 sec, 1,664 bytes
true.

?- human(Y).
Y = alice yes

?-  human(Y).
Y = alice no
Y = bob. yes

?-
```

というように、1 つ目の解 aliceのあとに、ユーザが yes といれると、そこで終わるが、no といれると、次の解を表示してくれる。yes といれると、ユーザが望む解がでたと認識し、終了する。(この場合は、解が 3 個しかないので、no をいれると chrisを表示した後、終了する。)

5 論理プログラミングの特徴

論理プログラミングには、論理における推論 (自動証明) 手続きを「計算」と見なす、という発想そのものが当時としては極めて画期的であったが、さらに、以下のような特徴がある。

- 宣言型プログラミング; 事実や規則を書くだけで (対象領域を正確に記述するだけで)、実際の証明手順を一切記述せずに、解を求めることができる。
- 双方向性; 既に見てきたように、「入力から引数を求める」という流れが固定されておらず、どの引数からどの引数への計算とでも見なせる。

もちろん、デメリットもあり、実行速度が必ずしも高速ではない (高速化しようとすると、上記のメリットをある程度削ってしまう) ということはある。現在、汎用のプログラミング言語として Prolog を使うことはあまりないが、ルールベースのプログラミングや知識発見の手段としての帰納論理プログラミングなど、特定用途では、様々な形で Prolog の発展形を見ることができる。

6 まとめ

- ホーン論理 (ホーン節に限定した論理)
- 論理プログラミング
- Prolog による実際の処理

7 演習問題

演習問題 1:

上記の add プログラムを add.pl というファイルにいれ、[add]. として読みこんだあと、以下のクエリを順番に入力して、どのような解が返ってくるか観察しなさい。(なお、?- の部分は、システムが出力したプロンプトであるので、ユーザは、そのあとの部分を入力すればよい。最後にピリオドが必要なことに注意せよ。)

複数の解がある場合は、そのすべての解を観察しなさい。

```
?- add(0,s(0),Y).
?- add(s(s(0)),s(s(s(0))),Y).
?- add(s(s(0)),Y,s(s(s(0)))).
?- add(Y,s(s(0)),s(s(s(0)))).
?- add(X,Y,s(s(s(s(s(0)))))).
```

演習問題 2:

同様に times プログラムに対して、以下のクエリを試しなさい。

```
?- times(s(s(0)), s(s(s(0))), X).
?- times(s(s(0)), X, s(s(s(s(s(0)))))).
?- times(X, Y, s(s(s(s(s(0)))))).
```

演習問題 3:

余力がある人は、「X の Y 乗が Z になる」や「X と Y の最小公倍数が Z になる」を計算する Prolog プログラムを書きなさい。