Functional and logic programming lab 9th report

Yoshiki Fujiwara, 05-191023

1 Q1: 血縁関係

1.1 動作例

Code 1 動作例

```
?- bloodrelative(sanae,X).
X = iwao ;
X = mine ;
X = miho ;
X = kobo ;
X = sanae ;
X = miho ;
X = kobo ;
X = sanae ;
X = miho ;
X = kobo ;

?- bloodrelative(iwao,X).
X = sanae ;
X = miho ;
X = kobo ;
false.

?- bloodrelative(iwao,mine).
false.
```

1.2 考察

bloodrelative を定義した。まず、共通の祖先を Z として、bloodrelative(X,Y) :- ancestor(X,Z), ancestor(Y,Z). とする。こうすることで共通の祖先を持つ者が list up される。ただ、このままだと、iwao と sanae が血縁関係にならないので、bloodrelative(X,Y) :- ancestor(X,Y).bloodrelative(X,Y) :- ancestor(Y,X). とすることで、祖先の情報がない親子関係についても定義することができるようになる。

2 Q2: 参考述語 mult の実装

2.1 動作例

Code 2 動作例

```
?- mult(s(z),X,s(s(z))).
X = s(s(z)) ;
false.

?- mult(X,s(z),s(s(z))).
X = s(s(z)) ;
false.

?- mult(s(s(z)),s(z),X).
X = s(s(z)).
```

2.2 考察

2.2.1 実装について

mult の三つ目の項に W をという変数を置き、ここに計算結果を代入するようにした。mult の定義は add と同じように二つからなる。

一つ目は、mult(z,,*z*).という定義である。この定義によって、一引数目に *zero* がきたら、*zero* を返すようにしている。

二つ目は、mult(s(X),Y,W) :- add(Y,Z,W), mult(X,Y,Z). という定義である。一つ目の引数をかける回数と見ているのがポイントである。mult で変数 Z に一回少ない回数分 mult したものが返ってきて、それに Y を足すということをしている。add の定義は、スライドの定義通りに実装した。

2.2.2 問い合わせ

どれを変数にして問い合わせができるかについて考察する。結果としては今回の実装ではどの部分を変数にしても問い合わせが可能である。

まず、add がどの部分を変数にして問い合わせができるかについて考察する。スライドに乗っている add の定義は、変数一つの場合はどの部分に変数を持ってきても問い合わせができ、変数二つの場合は、第二引数が変数であれば問い合わせができる定義である。これは add の定義について、add(z,Y,Y). が入っているからである。

例えば、第一引数と第二引数が変数の場合、初めのルールが先に適用されるため、無限ループに陥ることなく、第三引数がゼロになるまで続く。

第二引数と第三引数が変数の場合も同様で、第一引数が z になるまで 2 個目のルールが適用されて、最後に一つ目のルールが適用される。

この定義に着目すれば、mult(s(X),Y,W) :- add(Y,Z,W), mult(X,Y,Z). という定義をすれば、add についてに変数の探索が行われる時、第二引数は必ず変数である。よって、上述の説明より、必ず止まる。mult については add で Z が探索済みなので、一変数の探索を行うことになる。こうすることで、mult のどの部分を変数としても答えを返して止まるようになる。

他の定義をすると、止まらないことがある。例えば、mult(s(X),Y,W) :- add(Z,Y,W), mult(X,Y,Z). と定義する。mult の第三引数が変数の場合、add の第一引数を第三引数が変数となるため、add について解が無限に存在して、mult(s(s(z)),s(z),X). が無限ループに入ってしまう。

もう一つの mult の定義例として、mult(s(X),Y,W) :- mult(X,Y,Z), add(Z,Y,W). がある。これは mult(X,s(z),s(s(z))). が無限ループに入ってしまう。mult について変数が二つの問い合わせをしていて、永遠に問い合わせをしてしまうからである。具体的には、二つめのルールの適用が永遠に可能となってしまう。

よって、一番初めに述べた実装方法で実装した。

3 Q3: reverse append の実装

3.1 動作例

Code 3　動作例

```
?- reverse([1,2],X).
X = [2, 1].

?- reverse([1,2,3],X).
X = [3, 2, 1].

?- reverse([1],X).
X = [1].

?- reverse([],X).
X = [].

?- concat([[1], [2,3]], X).
X = [1, 2, 3].

?- concat([[1], [2,3], [4,5]], X).
X = [1, 2, 3, 4, 5].
```

3.2　考察

まず、スライド通りに apeend を実装した。

3.2.1　reverse

まず、reverse の第二引数に reverse した結果が返るように、すなわち、第二引数を変数とした問い合わせができるように実装する。

評価の際は、第一引数を減らしていって、空リストに帰着させるという方針で実装する。よって、まずは reverse([],[]). とする。そして、cons を用いて、要素を一つずつ取って、後ろにつければ良いと考え、reverse([X | Y], Z) : $-reverse(Y, W), append(W, [X], Z)$. とする。こうすることで、*list* から一つ要素 X を取って、Y を *reverse* したものと X を *append* するコードとしてかける。W は Y を *reverse* した結果を入れる変数である。

3.2.2　concat

これも reverse と同じ方針で実装する。concat の第二引数には concat を行なった結果が格納される。新たな変数 W を導入し、concat([X | Y], Z) :

$-concat(Y,W), append(X,W,Z)$.とかく。$W$ には Y という $list$ の $list$ について $concat$ を行なった結果が格納される。$append$ そしてそれに対して、X を $append$ すれば、$concat$ が実装できたことになる。

4 Q4: Hamilton 路

4.1 動作例

Code 4 動作例

```
?- hamilton([1,2,3,4],[[1,2],[2,3],[3,4]]).
true ;
false.

?- hamilton([1,2,3,4,5,6],[[1,2],[2,3],[3,4],[4,5],[5,4],[5,6]]).
true ;
false.

?- hamilton([],[]).
true ;
false.

?- hamilton([1,2,3,4,5,6],[[1,2],[2,3],[3,4],[4,5],[5,2],[2,6]]).
false.
```

4.2 考察

hamilton 閉路を実装するために、様々な fact を実装したので、その説明をまずする。

まず、append 関数を実装した。append 関数は Ocaml でいう@にあたる関数である。スライドにあった実装をそのまま行なった。

次に、vertexin という fact を実装した。これは第一引数が第二引数の list に含まれているか judge する fact である。これは list を順番に見ていき、同一のものがあれば、第一式、vertexin(X,[X|_]).で $true$ が返される。X の部分が一致していなければ残りの $list$ をみる第二式で判断する。

次に、edges という fact について説明する。この fact は頂点 A と辺集合 E が与えられた時に、E の中で A を始点とする辺の終点を探索する時に用いる。第三引数を変数にして用いるイメージである。

さらに、pick という fact を定義した。この fact は頂点 X を頂点集合 V から抜いたものを用いたいときに用いる。第一引数に取り除きたい vertex を、第二引数に vertex set を、第三引数に変数を持ってきて、第二引数から第一引数を取り除いた veretx set を第 3 引数に入れる。

最後の補助 fact として、search を定義した。この fact は第一引数に頂点 A、第二引数に辺集合 E、第三引数に変数 V を持ってきて使う。(V,E) について A を始点として hamilton path があれば、true を返す。まず、第一引数から次に移る頂点 B を選ぶ。これは edges を用いることで実現できる。edges で得た次に移る候補を頂点集合 V から除いたものを R として、B を始点とする、残りの頂点が R の hamilton path を考える。この動作において、pick を用いて V から R を求めているが、この作業時に、B が V になければ、false が返ってくるので、hamilton path の同一の頂点を二回通らないという条件を満たしている。

最後に、hamilton を定義する。空がきた場合は、true を返すようにしている。まず、vertexin で、任意の頂点 Z を頂点集合から選ぶ。そのを頂点集合から除き、search で hamilton path が存在するか否かを判定する。

このようにして実装ができる。

5 Q5: チューリング完全であることの証明

5.1 方針

チューリング完全であることの定義にはたくさんの同値関係がある。例えば、構造化プログラミング定理と呼ばれるものや、ラムダ計算を記述できることや、再帰関数を記述できることなどがある。

今回は再帰関数を表現できることを示すことで、チューリング完全であることを示す。

再帰関数を表現できることを示すには、Base case と、Composition, Primitive recursion, Minimization をそれぞれ示せばよい。

5.2 Base case について

Basecase について示すべきことは、zero と succ と、proj を表現できることである。

5.2.1 zero と succ

zero については今回の Q2 で使った z がそれにあたり、succ については今回の Q2 で使った、s() がそれにあたる。

5.2.2 proj

proj を表現するものは、list のデータ構造である。list は list のどの要素についてもアクセスが可能であることから、これは proj そのものを表現していると言える。

実際にアクセスするコードは、以下のようになる。

Code 5 proj

```
P(z,[A,_],A).
P(s(X),[_,B],Z) :- P(X,B,Z).
```

こうした上で、Z を変数として問い合わせると、list の X 番目にアクセスできる。

5.3 Composition について

$g_0 ... g_{m-1}$ を recursive function として、

$\lambda(x_0, \ldots, x_{n-1}) \cdot g(g_0(x_0, \ldots, x_{n-1}), \ldots, g_{m-1}(x_0, \ldots, x_{n-1}))$: $N^n \to N$

を表現できることを示せばよい。

以下のように表現できる。

Code 6 Composition

```
P(X1,X2,.....,Xm,Z) :- P1(X1,X2,...Xm,Z1), P2(X1,X2,....,Xm,Z2), ...
Pm(X1,X2,...,Xm,Zm), Q(Z1,Z2,.....,Zm,Z)
```

5.4 Primitice rucursion について

g と h を recursive function として、

$f(\vec{x}, 0) := g(\vec{x})$

$f(\vec{x}, y+1) := h(\vec{x}, y, f(\vec{x}, y))$

を表現できることを示せばよい。

以下のように表現できる。

Code 7 Primitive recursion

```
P(X,z,Z) ;- Q(X,Z)
```

```
P(X,S(Y),Z) :- P(X,Y,W), R(X,Y,W,Z)
```

h の中に f が入っていることは上の W のようにすると、表現できる。

5.5 Minimization について

f を recursive fucntion として、

$\lambda\vec{x}. \quad (\mu_y.f(\vec{x},y)=0) \qquad : \quad N^n \to N$

を表現できることを示せばよい。

これは、$f(\vec{x},y)$ に対応する述語をまず作り、P(X,Y) として、この Y の部分を変数として、prolog で実行すればよい。こうすることで、prolog は $f(\vec{x},y)=0$ を満たす最小の y を返す。よって、Minimization は表現できている。

ここで、Prolog の処理系は最小のものからアクセスするのかという疑問が上がるかもしれないが、実際、prolog の処理系では minimum の値からアクセスされている。

Code 8 Minimization

```
?- add(X,Y,s(s(s(z)))).
X = z,
Y = s(s(s(z))) ;
X = s(z),
Y = s(s(z))
```

以上より、recursive function を表現できたことになり、ty ーリング完全であることが示された。