Functional and logic programming lab 11th report

Yoshiki Fujiwara, 05-191023

1　Q1: 述語がうまく動作しない理由

以下の述語がうまく動作しない理由を述べる。

Code 1　動作例

```
ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z).
ancestor(X,Y) :- parent(X,Y).
```

例えば、ancestor(kobo,iwao) を問い合わせると帰ってこない。

まず、ancestor(kobo,iwao) について一番めのルール ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z). を適用する。

適用すると、ancestor(Z,iwao),parent(kobo,Z). となり、ancestor(Z,iwao) について一番目のルールを適用する。

適用すると、ancestor(Z_1, $iwao$), $parent(Z, Z_1)$.となり、$ancestor(Z_1, iwao)$ について一番目のルールを適用することになる。

このように、ancestor(変数,iwao) を永遠に問い合わせることになる。

以上から、この場合では無限ループに陥ってしまうことが言える。

2　Q2: 述語がうまく動作しない理由

Code 2　動作例

```
nat(z).
nat(s(N)) :- nat(N).
nat_list([]).
nat_list([N|X]) :- nat(N), nat_list(X).
```

このプログラムは自然数の list を返すことを想定していると判断した。

まず、nat_list(X) の動きを見る。以下のような動きをする。

Code 3　動作例

```
?- nat_list(X).
X = [] ;
X = [z] ;
X = [z, z] ;
X = [z, z, z] ;
X = [z, z, z, z] ;
X = [z, z, z, z, z] ;
X = [z, z, z, z, z, z]
```

このように z の list が返る。

この理由について記述する。nat_list について、まず、nat_list([]). を見る。これで [] が返る。

次に、nat_list([N | X]) : $-nat(N), nat\_list(X)$.の初めのルールを適用する。nat を適用すると、[] が返され、二番目のルールが適用される。

これが繰り返される。

このルール適用は永遠に終わらないため、nat_list([N | X]) の二番目の $rule$ の中の $nat(N)$ の二番目に入ることはない。

これはなぜなら、二番目の探索をするのは、一番目の探索が終わってからであるからである。

よって、上のような出力になる。

3　Q3: 三目並べ

3.1　動作例

Code 4　動作例

```
?- win(1,[0,0,0,0,0,0,0,0,0]).
false.

?- win(2,[0,0,0,0,0,0,0,0,0]).
false.

?- win(1,[1,0,0,2,0,0,0,0,0]).
true ;
```

```
true ;
true ;
false.

?- win(1,[0,0,0,2,1,0,0,0,0]).
true ;
true ;
true ;
true ;
true ;
true ;
false.

?- win(1,[0,1,0,2,0,0,0,0,0]).
true ;
true ;
false.
```

3.2 考察

この win を実装するために様々な述語を定義したのでその説明を順にする。

盤面については長さ 9 の list で表し、まだ置かれていない場所を 0、置かれた場所を 1 と 2 で表すということにする。

3.2.1 three_line

この述語は、P について、盤面で三つ並んでいるか否かを判定する関数である。三つ並んでいれば true、並んでいなければ false になる。

3.2.2 end

この述語は盤面が最終盤面か否かを判定する。最終盤面であれば true、最終盤面でなければ false を返す。

3.2.3 change

相手の勝ちを予測する場面があるが、その時に盤面の 1 と 2 を反対にして評価することにした。この述語は、list のうち、1 である場所を 2 に、2 である場所を 1 にする述語である。

3.2.4 exist_step

この述語は、exist_step(P,B,NEW) のように表し、P には 1 か 2 が入り、0 である場所のどこかに P を入れる。これは盤面を一つ進めることに対応している。この実装には前回実装した append 関数を用いた。

3.2.5 win

これまでの述語を使って、win という述語を定義することができる。win(P,B) は盤面が B で次に打つのが P なら、P の必勝であることを表す。

これには二種類ある。まず一つ目は、P のマークが三つすでに並んでいる場合である。これは、three_line(P,B) で調べることができる。そして二つ目は、相手のマークがすでに三つ並んでいなくて、一つ進めると lose(相手, 進めた後の盤面) となる場合である。相手のマークがすでに三つ並んでいないことは、change(B,C), $\backslash + three\_line(P, C)$ で表現できる。また、それ以降の部分は、$exist\_step(P, B, B1), change(B1, B2), lose(P, B2).$ で表現できる。$exist\_step$ で打てる手を調べて、$change(B1, B2), lose(P, B2)$ で P の相手の負けを判定している。この負けの判定の $lose$ については以下で説明する。

3.2.6 lose

lose(P,B) は盤面が B で次に打つのが P なら、P の相手が必ず勝つことを表す述語である。

lose についても、二つの場合がある。まず一つ目は、すでに相手が三つ揃えている場合である。その場合は、lose(P,B) :- change(B,B1), three_line(P,B1). で調べることができる。

二つめの場合は、最終局面ではなく、P がどんな手を打っても相手が勝つ場合である。

最終局面ではないことは $\backslash + end(B)$ で書くことができる。

任意のを表現するために、否定を用いて表現した。先ほどの「P がどんな手を打っても相手が勝つ場合を否定する」と、「P がどこかに置くと相手は勝つことができない」になる。

それを exist_and_not_win(P,B) :- exist_step(P,B,B1), change(B1,B2), $\backslash + win(P, B2).$ という述語で表現した。

以上から、lose の二番目の場合は、lose(P,B) :- $\backslash + end(B), \backslash + exist\_and\_not\_win(P, B).$ と表すことができる。

3.2.7　全体として

上の動作例では、スライドに乗っていた必勝法三通りと、初期盤面がどちらの必勝でもないことを示した。

4　Q4: 処理系の実装

この実装について、時間的制約のもと、終わらせることができなかったのですが、部分点だけでもいただけると幸いです。

4.1　実装

まず、型の実装をした。述語の型と、expression の型を定義した。

expression の型については述語に対応する TyFun と、変数に対応する TyVar とそれ以外を表す TySym とした。

unifier を作る必要がある。導出において、最汎単一化子を求めることが必要になるからである。このコードは第八回の課題で作成したものに基づいて作成した。

次に、eval_command を実装した。eval.ml 参照。今回はプログラムの部分は初めから env に保存しておく実装にしたため、とい合わせに対応する Query の部分のみを実装すれば良い。

Query の部分で受け取った問い合わせる式はまずは、Queue に入る。そして、search_solution 関数に渡される。

search_solution 関数では、関数の中で search 関数を呼ぶ。主な探索は search 関数で行う。この関数では何をするかというと、答えの出力を行う。search 関数で返ってくるのは制約集合であるため、その制約を実際の変数に代入して、答えとする。また、探索に失敗していた場合は false を返す関数である。この中の補助関数として、gen_solution という関数が定義されているが、この関数は、制約を実際の変数に代入する関数である。

search 関数の説明をする。探索は幅優先探索を行うことにした。まず、探索を始める (goal の)predicate について、construct_node という関数を用いて、goal の predicate を書き換える。queue の中身を見るために、一つ一つ pop していく作業を行いたいので、construct_node の前に、queue をコピーしておいた。

そして新しく得た goal のもと、eval_goal 関数を用いて、評価する。goal から一つ取り

出して、rule を用いて unify を try する全ての goal に対して unify ができなかった場合は fail となる。もし、unify できたら、unify を行なって、代入後の goal と代入の pair を queue に追加する

よって、今回実装したのは、unifier の部分と、eval する部分である。eval する部分のデバッグが行えておらず、error が出てしまいます。型については type.ml に、eval については eval.ml に実装しています。