Functional and logic programming lab 12nd report

Yoshiki Fujiwara, 05-191023

1　等式論理

1.1　論理的解釈

この論理的解釈で正しいことを示す。

eq(c,b) が成立しているので、4 つ目のルールを用いて、eq(b,c) も成立する。

よって、eq(a,b) と eq(b,c) より、三つ目のルールを用いて、eq(a,c) も成立する。よって、論理的解釈のもとでは、eq(a,c) は true である。

1.2　prolog 処理系での問い合わせ

prolog では上のルールから順に適用されていくので、まず、eq(a,c) について三つ目のルールが適用される。

そして、eq(a,Y),eq(Y,c) のルールについて、まず左側の eq(a,Y) の探索を行う。

まず、一つ目のルールより、Y=b で true となる。prolog 処理系は深さ優先探索なので、三つ目の eq(X,Y),eq(Y,Z) のルールについてもみて、

eq(a,Y_1)$, eq(Y_1, Y)$ のルールの探索を行う。

このように、eq(a, 変数) の探索を永遠に行い続けるようになる。

よって止まらない。

1.3　処理系の工夫

これを解消するために、処理系を幅優先探索にすれば良い。

幅優先探索にすると、上述の eq(a,Y),eq(Y,c) のルールについて、まず左側の eq(a,Y) の探索を行う段階で、Y=b のみがまず eq(a,Y) を true にする答えとして探索結果が出る。

その次に、eq(b,c) の探索に入るが、一つ目〜三つ目のルールは適用されても、解がな

い。4 つ目のルールで適用されるルールでは、eq(b,c):-eq(c,b) 隣、true が返る。
よって、論理的解釈に近づく。

2 単一化

2.1 論理的解釈

論理的解釈のもとでは、
一つ目の式は、$\forall X.q(X,X) \rightarrow test$ を表していて、
二つ目の式は $\forall X.q(X,f(X))$ を表している。
test. の真偽値を問うことは、$\forall X.q(X,X)$ の真偽値を問うことと等しく、$\forall X.q(X,f(X))$ だけでは導けないので、false が返ってくる。

2.2 test. の問い合わせ

しかし、prolog 処理系で、test. を問い合わせると、true が返る。
これは、q(X,X) が true となる解が存在していることを表す。
実際に q(X,X) を問い合わせしてみると、X=f(X). が返ってくる。
これは prolog 処理系が単一化の出現チェックを行なっていないからである。出現チェックを行なっていない場合、最汎単一化子となる代入 X=t に置いて、t に X が入っている状態になり得るからである。
SLD 導出において、q(X,X) と q(X,f(X)) が単一化可能か否かを調べる。その際に、出現チェックをしていないと、X=f(X) を、単一化子として単一化可能となってしまう。これによって、X=f(X) が答えとして出てしまう。

3 否定

3.1 論理的解釈

プログラムについて論理的解釈を行うと、
一つ目の式については、$\forall X.(p(X) \leftrightarrow X = a)$、
二つ目の式については、$\forall X.(q(X) \leftrightarrow X = b)$ となる。
問い合わせについては、$\exists X.(\neg p(X) \wedge q(X))$ と論理的解釈ができる。
X=b について、これは成立するため、この問い合わせは true となるはずである。

3.2 prolog 処理系での問い合わせ

しかし、prolog 処理系で問い合わせを行うと、false となる。$\backslash + p(X)$ は常に、$false$ となってしまう。これは $p(a)$ で成立するというルールしか宣言していないためである。$prolog$ 処理系では、$\backslash + p(X)$ が論理的意味として、$\neg \exists X.p(X)$ となってしまう。

?- q(X), $\backslash + p(X)$.とすると、論理的解釈と同じ結果が得られる。これはなぜなら、$q(X)$ についての問い合わせを先に行なって、$X = b$ が返ってきて、$\backslash + p(b)$ は $true$ を返すため、$X = b$ が返ってくる。こうして、not の部分を変数でなくすれば、期待した通りの問い合わせができる。

4 発展 1

4.1 論理的帰結であること

このプログラムが論理的帰結であることを示す。

p(a) の真偽は必ず定まるので、p(a) か $\neg p(a)$ のどちらかは真となるので、r(a) は常に true となる。

4.2 Prolog 処理系での問い合わせ

Prolog 処理系でこれを問い合わせると、無限ループに入ってしまう。

r(a) について prolog で問い合わせを行うと、p(a) についての問い合わせを行う。

p(a) についての問い合わせを行うと無限ループする。なぜなら、p(X) :- p(f(X)) の問い合わせが永遠に終わらないからである。

5 発展 2

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。

5.1 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。