Prolog in Ocaml

He Xiao, Shijiao Yuwen

December 16, 2014

Abstract

This report presents the design and implementation of the application 'Prolog-Ocaml'. Basically, this project used Prolog [7] as a template, and designed a variant of Prolog in Ocaml; it covers the most basic phases of implementing a new programming paradigm: lexing, parsing, interpreting. Given the source code in predefined grammar, the delivered product can parse it to an internal AST and execute it. After testing it with some well-known Prolog programs and comparing the results with the official Prolog implementation, some strengths and weaknesses of our implementation have been found.

Grammar

There are several editions of prolog currently available online: SWI-prolog[6], GNU-prolog[1], Amzi-prolog[2], and so on. According to our research: they are not identical, including the grammar. The BNF grammar describing the acceptable input to our Prolog simulator is a core edition[4] of the most basic prolog, so it is safe to start here.

Lexing and Parsing

Our implementation of prolog is more close to SWI-prolog due to the better availability of reference. Because we are implementing a subset of prolog, only a subset of tokens are defined. As an resource, we found an implementation of Amzi-prolog by Ocaml online[3], but Amzi-prolog is more of an old fashion and its documentation are no longer available online. We just used a some of its regular expressions and token explanations, which are shared between different versions of prolog. The overall layout and structure of lexer and parser in this project is the same as our mp homework. In lexer, we firstly define useful regular expressions, and then match symbols or regular expressions with corresponding tokens: there are independent entries for multi-line comments, single and double quoted string contents. In parser, we firstly define language tokens used by lexer and parser, and then the 'goal' nonterminal of our grammar: we allow program entry parsing (with query) and rules entry parsing (without query), and finally we construct the stratification needed for unambiguous parsing with reference to the precedence table we found on SWI-prolog manual (Fig 1).

Interpreting

There are two preliminary functions involved in developing interpretation module. The first one is evaluation function for terms and the second one is for predicates.

In order to process the terms universally, a general value type is defined which covers all the primitive data types, namely 'int', 'float', 'bool','string','list'. A constant term will be evaluated to its primitive value; A compound term which represents arithmetic operations or boolean logical operations will be simplified to a single value. A thing needs to be mentioned is that in standard Prolog, arithmetic operations like $+, -, *, /$ are overloaded, i.e. for both integer operations and floating point operations, the same operator symbol is used for each kind of operation. In order to handle this, both operands are converted to floating point numbers before doing computation if the two operands in a binary arithmetic operation have different precision levels. For

1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	**dynamic**, **discontiguous**, **initialization**, **meta_predicate**, **module_transparent**, **multifile**, **public**, **thread_local**, **thread_initialization**, **volatile**
1100	xfy	;, |
1050	xfy	->, *->
1000	xfy	,
990	xfx	:=
900	fy	\+
700	xfx	<, =, =.., =@=, \=@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, **as**, **is**, >:<, :<
600	xfy	:
500	yfx	+, -, /\, \/, **xor**
500	fx	?
400	yfx	*, /, //, **div**, **rdiv**, <<, >>, **mod**, **rem**
200	xfx	**
200	xfy	^
200	fy	+, -, \
100	yfx	.
1	fx	$

Table 5 : System operators

Figure 1: Precedence table [5]

ListTerm, each term inside the list will be evaluated and the values will be used to compose a value list. For variables terms, there is no way of evaluating an uninstantiated variable: there is no memory concept, variables are eliminated through substitution; as a result, if a variable is still a variable when doing the evaluation, it will never get any value and an exception will be thrown. The other fundamental function is created for evaluating a

predicate to either true or false. Again, the evaluation is performed through structural induction. At first, the predicate name will be checked, if it is one of the supported built-in function, then it will be interpreted in the 'built-in' way. Interestingly, there exists an overlap between predicates and some proportion of compound terms: the binary boolean operators which return boolean values. For these things, their identities depend on the context in which they are evaluated. If the name of the predicate cannot be found in the built-in functions list, then the predicate will be passed to another function which will traverse the rules list to seek a match. In the case that some rule's head can be unified with the query predicate, then the body of the rule will be evaluated, and the final boolean result depends on the result of evaluating that body predicates.

Algorithms For the Interpretation

Big picture: Given a Prolog program consisting of a bunch of clauses and a query, our program is expected to output the first result for the query, and its corresponding substitution.

Our application will first decompose the program to two parts: clause (either fact or rule) list and one query. Then decompose the Query to a list of predicates, where each individual predicate can be evaluated via the above algorithm.

The list of predicates in side the body part of a rule will be evaluated from left to right, and the list of predicates is left-associative. To help the computation of the final boolean result, a list of boolean connectives (, or ; representing $\wedge$, $\vee$ respectively) is also provided by parser. Each time a predicate in the list obtains its result $(curBool, curSig)$, it will compute the new boolean value by applying the boolean operator (look up the top item in connective list) on the $lastBool$(the boolean result of previous predicate, which was computed by the last iteration) and $curBool$. After getting the $newBool$, it will pass it to the next iteration. The substitution $curSig$ will be used to update the remaining predicate list.

After getting the new boolean and new predicate list, the recursion can begin.

Backtracking

Rigorously speaking, the backtracking algorithm implemented in this project is not the same as the classical ones. It will backtrack silently and try to gather all the results, but it will not print the result immediately when find some; instead, it will output the results to the terminal after collecting all the results it obtains. In oder to simulate the behavior of standard Prolog, when multiple results are available, after presenting the first result, it will wait for the user's instruction and then respond accordingly.

Currently, in the application which uses backtracking algorithm (**play_all.exe**), only Horn clause is supported: i.e. the body of the rules as well as the query must be predicates connected via logical and operators (in Prolog, ',').
Furthermore, in order to avoid entering infinite loops, a list is maintained which records, for each rule, all the queries that have matched the head predicate of that rule in the past. It has both advantages and disadvantages after adding this feature: the good thing is it will not get lost in one branch while some other results can be found easily in other branch; the bad thing

Algorithm 1 Algorithm for evaluating a single predicate

Input: single predicate, rules
Output: (true / false, substitution)
function EVAL_PREDICATE(*rules*,*predicate*)
    if *predicate* is built-in **then**
        evaluate *predicate* accordingly
        return (evaluatedResult,[])
    else
        return EVAL_PRED_WITH_RULES (*rules*, *predicate*)
    end if
end function
function EVAL_PRED_WITH_RULES(*rules*, *predicate*)
    for each clause i in the rule list **do**
        if *predicate* unifiable with rule i's head **then**
            get substitution *sigma* from the unification
            if rule i is a fact **then**
                return (true, *sigma*)
            else
                Apply *sigma* to the body of the rule (rename free vars if needed)
                Get the new query q
                Evaluate q and get result $(b, sig2)$
                if b is true **then**
                    get final substitution $finalSig$ by composing $sig2$ and $sigma$
                    return (true, $finalSig$)
                else
                  continue, apply the next rule if possible
                end if
            end if
        end if
    end for
    return (false,[])
end function

is that it may have false alarm so that it will miss some true results. In the evaluation part, we will demonstrate that in certain area, the advantage of our approach outweigh its drawbacks.

Method for collecting all the results

Each time a predicate is evaluated, even if a true result can be obtained in some path, it does not return immediately. Instead, it will continue using remaining rules to try to obtain other results. All these results are combined to form the final result list for evaluating the predicate.
To get all the results for a query with multiple predicates, the first predicate is first evaluated and we can get a list of results. For each result (b, sig), we can use it to update the remaining predicate list and get a new query; then the result list for that branch can be obtained by consulting the updated query. The final result set is simply combining all the result lists in different branches.

Testing Results and Evaluation

In order to pass some certain tests, several built-in functions (such as 'write', 'nl' and most arithmetic and comparing functions) in Prolog are implemented, other than that, no built-in functions in Prolog are supported.

The output format is simulating Prolog's. If no result is found, then a *false* will be given. If multiple results are available, then the substitutions will be printed one by one if the user requires. If there is no substitution exists and the query succeeds, then a true will be given.

Factorial

```
1    factorial(0,1).
2    factorial(N,F) :-
3    N>0, N1 is N-1, factorial(N1,F1),
4    F is N * F1.
5
6    ?- factorial(5,W).
```

Output:
$W = 120.$

Hanoi

```
1    move(1,X,Y,_) :-
2    write('Move top disk from '),
```

```
3       write(X),
4       write(' to '),
5       write(Y),
6       nl.
7       move(N,X,Y,Z) :-
8       N>1,
9       M is N-1,
10      move(M,X,Z,Y),
11      move(1,X,Y,_),
12      move(M,Z,Y,X).
13      ?-   move(3,left,right,center).
```

Output:

```
1    'Move top disk from 'left ' to 'right
2    'Move top disk from 'left ' to 'center
3    'Move top disk from 'right ' to 'center
4    'Move top disk from 'left ' to 'right
5    'Move top disk from 'center ' to 'left
6    'Move top disk from 'center ' to 'right
7    'Move top disk from 'left ' to 'right
8
9    true.
```

List Size

```
1    size([],0).
2    size([H|T],N) :- size(T,N1), N is N1+1.
3
4    ?- size([1,2,3,4],N).
5
6    Output:
7    N=4.
```

Transitive relation

```
1 sis(joyce,niu).
2 sis(keke,joyce).
3 sis(joyce,ker).
4 sis(X,Y) :- sis(X,Z), sis(Z,Y).
5 ?-sis(X,Y).
```

```
1    Output:
2    X=joyce.
3    Y=niu.
4    ;
5
6    X=keke.
7    Y=joyce.
```

```
 8    ;
 9
10    X=joyce.
11    Y=ker.
12    ;
13
14    X=keke.
15    Y=niu.
16    ;
17
18    X=keke.
19    Y=ker.
```

This transitive example is where our implementation achieves a better result than the official SWI-Prolog: the official Prolog only gets the results from the facts, but it runs out of its stack before getting the other two results via applying the last rule.

Limitations

- In the implementation of the 'play.exe' which retrieves a single result, a true result may not be found in the situation where a rule's head can unify with the predicate but the rule does not fire due to the body part does not return true.

- If the input argument is too large, then 'play_all.exe' may not compute the results correctly. Some incorrect result has been found when compute the factorial with big number.

- Some feature like *Cuts* are not supported in the current implementation of our tool.

Due to the time constraints, the application is not tested thoroughly; but to the best of our knowledge, it runs as expected on the provided classic examples.

Possible future work

- Currently, not many built-in functions are supported due to the limited implementation time. However, an idea is that, given the path to official Prolog's library, if the built-in functions are also written in normal syntax of Prolog program, then our implementation should be able to make use of those built-in rules and obtain the results as if all the rules are defined locally in one file.

- Currently, due to the limitation of algorithms used, some results may not be detected even if they are true. An idea to solve this problem is by adding the results obtained from last query as facts to the knowledge base, so in the future queries, those previously missing results will not be missed again. This design is also reflecting the process of children's continuously learning via remembering facts. Another possible solution is: before returning the computed results to the user, the application can analyze the results and get some new facts and place them into the knowledge base. After that, the application re-run the query to check whether the new results obtained is the same as the previous one (if multiple results available, the order maybe different). This process is repeated until a fixed point is reached, when the final complete result will be returned to the user.

Bibliography

[1] GNU. Prolog built-in predicates. `http://www.gprolog.org/manual/html_node/gprolog024.html`, December 2014. [Online; accessed 4-December-2014].

[2] Amzi inc. Amzi prolog. url = http://www.amzi.com, December 2014. [Online; accessed 2-December-2014].

[3] Martin Keegan. ocaml-prolog. url = https://github.com/mk270/ocaml-prolog, 2012. [Online; accessed 9-December-2014].

[4] Ivan Sukin. Prolog bnf. url = https://github.com/simonkrenger/ch.bfh.bti7064.w2013.PrologParser/blob/master/doc/prolog-bnf-grammar.txt, June 2012. [Online; accessed 9-December-2014].

[5] SWI-Prolog. Predicate op/3. url = http://www.swi-prolog.org/pldoc/man?predicate=op/3, 2014. [Online; accessed 9-December-2014].

[6] SWI-Prolog. Swi-prolog documentation. url = http://www.swi-prolog.org, December 2014. [Online; accessed 4-December-2014].

[7] Wikipedia. Prolog. `http://en.wikipedia.org/wiki/Prolog`, December 2014. [Online; accessed 1-December-2014].

Appendix

For the complete source code and test cases, please visit our Github's page at `https://github.com/xiaohe27/PrologInOcaml`

Listing 1 : ProjCommon.ml: Commonly used functions in the project

```ocaml
1  (* ═══════════════ Parsing ═══════════════ *)
2  (* Types *)
3  type const =
4  BoolConst of bool
5  | IntConst of int
6  | FloatConst of float
7  | StringConst of string;;
8
9  type term = Var of string | ConstTerm of const |
10     CompoundTerm of string * (term list) |
11              ListTerm of term list |
12        PredAsTerm of predicate
13
14
15 and predicate = Identifier of string | Predicate of string * (
       term list)
16     | VarAsPred of string ;;
17
18 (* predicates can either be separated by comma or by semi-colon
       *)
19 type clause = Fact of predicate | Rule of predicate *
20          (predicate list * string list);;
21
22 type query = Query of (predicate list * string list);;
23
24 type rules = RuleList of clause list;;
25
26 type program = Prog of rules * query | ProgFromQuery of query;;
27
28 (* ═══════════════ Interpreting ═══════════════ *)
29 (* Values *)
30 type value =
31    BoolVal of bool
32   | IntVal of int
33   | FloatVal of float
34   | StringVal of string
35   | ListVal of value list
36
37
38 type blacklist = (int * string list) list;;
39
40
41
42 (*value output*)
43 let rec print_value v =
44    match v with
45
46   | IntVal n          -> if n < 0 then (print_string "~";
      print_int (abs n)) else print_int n
47   | FloatVal r        -> print_float r
48   | BoolVal true      -> print_string "true"
```

```
49    | BoolVal false       -> print_string "false"
50    | StringVal s         -> print_string ("\"" ^ s ^ "\"")
51
52    | ListVal l           -> print_string "[";
53                             (let rec pl = function
54                                 []       -> print_string "]"
55                               | v::vl  -> print_value v;
56                                           if vl <> []
57                                           then
58                                               print_string "; ";
59                                           pl vl
60                              in pl l)
61
62
63  (*substitution*)
64  type subst = (string * term) list;;
65
66  (* result *)
67  type result = bool * subst ;;
68
69  (*indexed rules*)
70  type indexedRules = (int * clause) list;;
71
72
73
74  (*print term*)
75  let string_of_const c =
76      match c
77      with IntConst n     -> string_of_int n
78         | BoolConst b    -> if b then "true" else "false"
79         | FloatConst f   -> string_of_float f
80         | StringConst s  -> s;;
81
82
83  let isInfix op = match op with
84          "+" -> (true) |
85          "-" -> (true) |
86          "*" -> (true) |
87          "/" -> (true) |
88          "**" -> (true) |
89          _ -> false ;;
90
91  let rec stringOfTermList tl = match tl with
92          [] -> "" |
93          [h] -> string_of_term h |
94          h::t -> string_of_term h ^ ", "^stringOfTermList t
95
96  and string_of_term term=
97    match term with
98      (Var v) -> (v) |
99    (ConstTerm const) -> (string_of_const const) |
100   (CompoundTerm(f,tl)) -> ( if ((isInfix f) && (List.length tl)
      = 2) then (string_of_term (List.hd tl) ^ f ^ string_of_term (
      List.nth tl 1)) else (
```

11

```ocaml
101       f ^ "(" ^ (stringOfTermList tl) ^ ")"   ) )
102     |
103    (ListTerm tl) -> ("[" ^ (stringOfTermList tl) ^ "]") |
104
105       PredAsTerm pred -> (string_of_predicate pred)
106
107 and string_of_subst subst = match subst with
108          [] -> "" |
109          (v,t)::tail -> (v ^ "=" ^ (string_of_term t)) ^
110           ".\n" ^ (string_of_subst tail)
111
112 and string_of_predicate pred=match pred with
113          Identifier(id) -> id |
114          Predicate(f,tl) -> (f ^ "(" ^
115          (stringOfTermList tl) ^ ")" ) |
116          VarAsPred(v) -> v ;;
117
118 let rec stringOfPredList predList connList= match predList with
119            [] -> "" |
120            [pred] -> (string_of_predicate pred) |
121            pred::tail -> ((string_of_predicate (pred)) ^ (List.hd
        connList) ^ (stringOfPredList tail (List.tl connList)));;
122
123 let string_of_clause clause = match clause with
124         Fact fp -> ("Fact "^(string_of_predicate fp) ^ "\t") |
125         Rule (hp,(body,connList)) -> ("Rule: "^(
        string_of_predicate hp) ^ " :- " ^ (stringOfPredList body
        connList));;
126
127 let rec stringOfRuleList rules = match rules with
128   RuleList clst -> (match clst with
129      [] -> "" |
130      h::t -> (string_of_clause h)^"\n"^
131        (stringOfRuleList (RuleList t)));;
132
133 let rec stringOfIndexedRules indexRules =
134   match indexRules with
135    [] -> "" |
136    (i,clause)::tail -> ("Rule " ^ (string_of_int i) ^ ":"
137         ^ (string_of_clause clause) ^
138         ("\n" ^ stringOfIndexedRules tail)) ;;
139
140
141 let rec string_of_stringList strList =
142  match strList with
143    [] -> "" |
144    str::tail -> str ^ ";" ^(string_of_stringList tail);;
145
146 let rec stringOfBlackList blist =
147      match blist with
148    [] -> "" |
149    (n, sl)::(tail) ->
150    ("(" ^ (string_of_int n) ^ (",") ^(string_of_stringList sl)^")
        " )
```

```ocaml
151    ^";\n"^(stringOfBlackList tail)
152 ;;
153
154
155
156 (* Fresh Name stuff *)
157
158 let int_to_string n =
159     let int_to_int_26_list n =
160         let rec aux n l =
161             if n <= 0 then l else let c = ((n-1) mod 26) in aux
    ((n -(c+1))/26) (c::l)
162         in aux n []
163     in
164         let rec aux l = match l with [] -> ""
165                             | n::ns -> (String.make 1 (Char.chr
    (n + 97))) ^ aux ns
166         in aux (int_to_int_26_list n);;
167
168 let freshFor lst =
169     let rec fresh_ n =
170         if List.mem (int_to_string n) lst
171             then fresh_ (n+1)
172         else int_to_string n
173     in fresh_ 1 ;;
174
175
176 let rec get_n_freshVars n lst =
177   if(n <= 0) then (raise (Failure "Cannot get 0 fresh vars"))
178   else (let fstFresh= freshFor lst in (if n=1 then ([fstFresh])
179     else (fstFresh::(get_n_freshVars (n-1) (fstFresh::lst)))) )
    ;;
180
181
182 (* End Fresh name stuff *)
183
184 (*Get free vars in a term*)
185 let rec freeVarsInTerm term =
186         match term with
187         Var v -> [v] |
188         ConstTerm _ -> [] |
189         CompoundTerm(f, tl) -> (
190           toSingleStrArr (List.map (freeVarsInTerm) tl) ) |
191
192         ListTerm(tl) -> (
193           toSingleStrArr (List.map (freeVarsInTerm) tl)) |
194
195         PredAsTerm (pred) -> (freeVarsInPredicate pred)
196 and
197
198 toSingleStrArr listList =
199   match listList with
200   [] -> [] |
201   [singleList] -> (
```

```
202          match singleList with
203        [] -> [] |
204        str::tail -> (str::(toSingleStrArr [tail])) ) |
205
206    fstList::tailListList -> (
207        fstList @ (toSingleStrArr (tailListList)))
208
209  and
210
211  (*Get free vars in other structures*)
212  rmXInList x lst = match lst with []->[] |
213          h::t->if h=x then rmXInList x t
214              else h::(rmXInList x t)
215
216  and rmDup lst = match lst with []->[] |
217        h::t-> h::(rmDup (rmXInList h t))
218
219
220  and freeVarsInPredicate pred =
221    match pred with
222    Identifier id -> ([]) |
223    Predicate (f,tl) -> (toSingleStrArr (List.map (freeVarsInTerm)
          tl)) |
224    VarAsPred v -> [v] ;;
225
226  let rec listSubtract list1 list2 =
227    match list2 with
228    [] -> list1 |
229    h::t -> let newList1 = (rmXInList h list1)
230      in(listSubtract newList1 t) ;;
231
232  let freeVarsInClause clause =
233    match clause with
234    Fact pred -> (freeVarsInPredicate pred) |
235    Rule (head, (body,conn)) -> (let binders= freeVarsInPredicate
        head in
236        let freeVarInBody = (toSingleStrArr (List.map (
        freeVarsInPredicate) body) ) in rmDup(listSubtract
        freeVarInBody binders) ) ;;
237
238  let freeVarsInQuery query=
239    match query with
240    Query(predList,connList) ->
241      (toSingleStrArr (List.map (freeVarsInPredicate) predList) )
        ;;
242
243  let freeVarsInRuleList rules =
244    match rules with
245    RuleList(clauseList) ->
246      (toSingleStrArr (List.map (freeVarsInClause) clauseList));;
247
248  let freeVarsInProgram pgm =
249    match pgm with
250    Prog(rules,query) -> (
```

14

```ocaml
251        let  binders= freeVarsInQuery  query  in
252        let  freeVarsInRules= freeVarsInRuleList  rules  in
253       rmDup ( listSubtract  freeVarsInRules  binders) )  |  (∗not
        precise ∗)
254
255    ProgFromQuery( query )  −>  ( freeVarsInQuery  query )  ;;
256
257
258  (∗Type  testing ∗)
259  let  isTypeTesting  op = match  op  with
260         ” var ”  −>  true  |
261         ” nonvar ”  −>  true  |
262         ” atom ”  −>  true  |
263         ” integer ”  −>  true  |
264         ” float ”  −>  true  |
265         ” number ”  −>  true  |
266         ” atomic ”  −>  true  |
267         ” compound ”  −>  true  |
268         ” callable ”  −>  true  |
269         ” list ”  −>  true  |
270         ” is_list ”  −>  true  |
271         _  −>  false ;;
272
273
274  let  retBool  op = match  op  with
275         ”>”  −>  true  |
276         ”<”  −>  true  |
277         ”=:=”  −>  true  |
278         ”>=”  −>  true  |
279         ”<=”  −>  true  |
280         ”=\\=”  −>  true  |
281         ” ,”  −>  true  |
282         ” ;”  −>  true  |
283         _  −>  false ;;
284
285  (∗  Test  whether  a  function  is  built−in  function  ∗)
286  let  isBuiltInOp  op =  if  ( isTypeTesting  op)  then  true
287                         else  if  (op =  ”=”  ||  op =  ” is ”  ||  op =  ”
       write ”
288                       ||  op =  ” nl ” )  then  true  else  ( retBool  op) ;;
289
290
291  (∗  Test  whether  a  predicate  only  contains  vars  in  the  term  list
        ∗)
292  let  rec  onlyVarsInPred  pred =
293     match  pred  with
294     Identifier  id  −>  ( false )  |
295     Predicate  ( f , termL)  −>  ( onlyVarsInTermList  termL)  |
296     VarAsPred  v  −>  ( true )
297
298  and  onlyVarsInTermList  tl =
299     match  tl  with
300     [ ]  −>  true  |
301     t1 :: tail  −>  (
```

15

```
302        match t1 with
303        Var v -> (onlyVarsInTermList tail) |
304        ConstTerm _ -> false |
305        CompoundTerm _ -> false |
306                  ListTerm _ -> false |
307            PredAsTerm pred0 -> onlyVarsInPred pred0);;
308
309 let rec onlyConstInPred pred =
310    match pred with
311    Identifier id -> (true) |
312    Predicate (f,termL) -> (onlyConstInTermList termL) |
313    VarAsPred v -> (false)
314
315 and onlyConstInTermList tl =
316    match tl with
317    [] -> true |
318    t1::tail -> (
319      match t1 with
320      Var v -> (false) |
321      ConstTerm _ -> onlyConstInTermList tail |
322      CompoundTerm _ -> onlyConstInTermList tail |
323                  ListTerm _ -> onlyConstInTermList tail |
324            PredAsTerm pred0 -> onlyConstInPred pred0);;
325
326
327
328 (* Test whether a string str1 contains another string str2 *)
329 let rec strContains str1 str2 =
330 if str1=str2 then true
331 else (
332 let len1=String.length str1 in
333 let len2=String.length str2 in
334 if len1 <= len2 then false
335 else
336 let sub1 = String.sub str1 0 (len2) in
337 if sub1 = str2 then true
338 else
339 let remain1= String.sub str1 1 (len1 - 1) in
340 strContains remain1 str2
341 );;
342
343
344 let rec isAllTrueBoolList boolList=
345 match boolList with
346 [] -> true |
347 curBool::tail ->
348 (if curBool = false then false
349 else (isAllTrueBoolList tail))
350 ;;
351
352
353 let rec getAllConstInTermList tl =
354 match tl with
355 [] -> [] |
```

```
356
357 curTerm::tail ->
358 (match curTerm with
359     Var v -> getAllConstInTermList tail |
360     PredAsTerm pred0 -> getAllConstInTermList tail |
361     _ -> curTerm::(getAllConstInTermList tail)
362
363 )
364 ;;
365
366 let getAllConstInPred pred=
367 match pred with
368    Identifier _ -> [] |
369    Predicate (_,termL) -> (getAllConstInTermList termL) |
370    VarAsPred _ -> []
371
372
373 (*Given a string list of items in blacklist,
374 test whether all the constant terms in constTL
375 occur in one entry of the list*)
376 let rec isContainedInOneStrInTheList strList constTL =
377 match strList with
378 [] -> false |
379 curStr::tail -> (
380 if (areFoundInOneStr constTL curStr)
381 then (true)
382 else (isContainedInOneStrInTheList tail constTL)
383 )
384
385 and areFoundInOneStr constTL str=
386 let listOfTermStr= List.map (string_of_term) constTL in
387 let occursCheckList= List.map (strContains str) listOfTermStr in
388 isAllTrueBoolList occursCheckList
389 ;;
390
391
392 let addSigToResult sigma result =
393 match result with
394 (b,subst) -> (b, sigma @ subst);;
395
396 let addSigToResultList sigma rl =
397 List.map (addSigToResult sigma) rl;;
```

Listing 2 :   Lexer.mll: Lexer for the input source code

```
1 {
2 open ProjCommon;;
3 open Parser;;
4
5 exception EndInput;;
6 }
7
```

```
 8 (* definitions section *)
 9 let capital = ['A'-'Z'] (* capital letters *)
10 let small = ['a'-'z'] (* small letters *)
11 let digit = ['0'-'9']
12 let underline = ['_'] (* underline character *)
13 let alpha = capital | small | digit | underline (* any
       alphanumeric character *)
14 let word = small alpha* (* prolog words *)
15 let quoted_name = '\'' [^ '\''] '\'' (* quoted names *)
16 let symbol = ['+' '-' '*' '/' '\\' '^' '<' '>' '=' '~' ':' '?' '
       @' '#' '$' '&']
17 let solo_char = ['!' ';' '.' '[' ']' '(' ')' ',' '|']
18 let name = quoted_name | word | symbol+ | solo_char (* valid
       prolog names *)
19 let variable = (capital | underline) alpha* (* prolog variables
       *)
20
21
22 let int = digit+
23 let frac = '.' digit+
24 let exp = ['e' 'E'] ['-' '+']? digit+
25 let float = int frac? exp?
26
27 let whitespace = [' ' '\t' '\n']
28 let open_comment = "(*"
29 let close_comment = "*)"
30
31 rule token = parse
32 | eof                    { EOF }
33 | whitespace             { token lexbuf }
34 |"not"                               { NOT } (* boolean negation *)
35 |"=:="                               { ARITH_EQ } (* arithmetical
       equality *)
36 |"=\\="                              { ARITH_INEQ } (* arithmetical
       inequality *)
37 |"->"                                { ARROW } (* if then [else] *)
38 |"\\="                               { TERM_NOTUNIFY } (* terms do
       not unify *)
39 |"=.."                               { TERM_DECOMP } (* term
       composition/decomposition *)
40 |"=="                                { TERM_EQ } (* term equality *)
41 |"\\=="                              { TERM_INEQ } (* term inequality
        *)
42 |"@=<"                               { TERM_ORDER_LEQ } (* term less
       or equal to (order of terms) *)
43 |"@>="                               { TERM_ORDER_GEQ } (* term
       greater or equal to (order of terms) *)
44 |"=@="                               { TERM_ORDER_EQ } (* term
       equality (order of terms) *)
45 |"\\=@="                             { TERM_ORDER_INEQ } (* term
       inequality (order of terms) *)
46 |"@<"                                { TERM_ORDER_LESS } (* term less
       than (order of terms) *)
```

```
47  |"@>"                              { TERM_ORDER_GREATER } (* term
        greater than (order of terms) *)
48  |">="                              { ARITH_GEQ } (* arithmetical
        greater or equal to *)
49  |"=<"                              { ARITH_LEQ } (* arithmetical
        less or equal to *)
50  |"is"                              { IS } (* variable instantiation
        *)
51  |"::"                              { DOUBLECOLON } (* module(
        database) specifier *)
52  |"\\/"                             { BITWISE_AND } (* bitwise and
        *)
53  |"/\\"                             { BITWISE_OR } (* bitwise or *)
54  |"\\"                              { BITWISE_NOT } (* bitwise not
        *)
55  |"^"                               { VAR_INSTANTIATED } (* is
        variable instantiated? *)
56  |"+"                               { PLUS } (* arithmetical plus *)
57  |"-"                               { MINUS } (* arithmetical minus
        *)
58  |"*"                               { MULT } (* arithmetical
        multiplication *)
59  |"/"                               { DIV } (* arithmetical division
        *)
60  |"("                               { LPAREN } (* left parenthesis
        *)
61  |")"                               { RPAREN } (* right parenthesis
        *)
62  |":"                               { COLON } (* else *)
63  |","                               { COMMA } (* logical and *)
64  |";"                               { SEMICOLON } (* logical or *)
65  |"="                               { TERM_UNIFY } (* unify terms *)
66  |"<"                               { ARITH_LESS } (* arithmetical
        less than *)
67  |">"                               { ARITH_GREATER } (*
        arithmetical greater than *)
68  |"!"                               { CUT } (* cut operator *)
69  |":-"                              { COLONHYPHEN } (* logical
        implication *)
70  |"?-"                              { QUESTIONHYPHEN }
71  |"["                               { LBRACKET } (* left bracket for
        lists *)
72  |"]"                               { RBRACKET } (* right bracket
        for lists *)
73  |"|"                               { PIPE } (* head-tail delimiter
        for lists *)
74  |  ".."          { DOUBLEDOT }
75  |  "."           { DOT }
76  |"%"[^'\n']*      { token lexbuf }
77  |  open_comment       { comment 1 lexbuf }
78  |  close_comment      { raise (Failure "unmatched closed comment")
        }
79  |  '"'          {stringToken "" lexbuf}
80  |  "'"          {singleStringToken "'" lexbuf}
```

19

```
81 | name as id        { NAME (id) }
82 | int               { INT (int_of_string (Lexing.lexeme lexbuf)) }
83 | float       { FLOAT (float_of_string (Lexing.lexeme lexbuf)) }
84 | variable         { VARIABLE (Lexing.lexeme lexbuf) }
85
86 and comment depth = parse
87 open_comment     { comment (depth+1) lexbuf }
88 | close_comment    { if depth = 1 then token lexbuf else comment
        (depth − 1) lexbuf }
89 | eof         { raise (Failure "unmatched open comment") }
90 | _            { comment depth lexbuf }
91
92 and stringToken content = parse
93   | eof {raise (Failure "unexpected end of file")}
94
95   | '"' {STRING content}
96
97   | "\\\\" {let newContent= content ^ "\\" in stringToken
      newContent lexbuf}
98
99   | "\\'" {let newContent= content ^ "'" in stringToken
      newContent lexbuf}
100
101   | "\\\"" {let newContent= content ^ "\"" in stringToken
      newContent lexbuf}
102
103   | "\\t" {let newContent= content ^ "\t" in stringToken
      newContent lexbuf}
104
105   | ("\\n")[' ' '\t']* {let newContent= content ^ "\n " in
      stringToken newContent lexbuf}
106
107   | "\\r" {let newContent= content ^ "\r" in stringToken
      newContent lexbuf}
108
109   | "\\b" {let newContent= content ^ "\b" in stringToken
      newContent lexbuf}
110
111   | "\\ " {let newContent= content ^ "\ " in stringToken
      newContent lexbuf}
112
113   | "\\" (digit as x) (digit as y) (digit as z) {let n=
114      let x=String.make 1 x in let y=String.make 1 y in let z=
      String.make 1 z in
115       (int_of_string z) + 10*(int_of_string y)
116       + 100*(int_of_string x) in if n > 255 then
117          raise (Failure "unknown char") else
118       let newContent= content ^ (String.make 1 (char_of_int n))
119          in stringToken newContent lexbuf }
120
121   | [^'"']+ as pstr {let newContent= content ^ pstr in
      stringToken newContent lexbuf}
122
123 and singleStringToken content = parse
```

```ocaml
124    | eof {raise (Failure "unexpected end of file")}

126    | "'" {STRING (content ^ "'") }

128    | "\\\\" {let newContent= content ^ "\\" in singleStringToken
         newContent lexbuf}

130    | "\\'" {let newContent= content ^ "'" in singleStringToken
         newContent lexbuf}

132    | "\\\"" {let newContent= content ^ "\"" in singleStringToken
         newContent lexbuf}

134    | "\\t" {let newContent= content ^ "\t" in singleStringToken
         newContent lexbuf}

136    | ("\\n")[' ' '\t']* {let newContent= content ^ "\n " in
         singleStringToken newContent lexbuf}

138    | "\\r" {let newContent= content ^ "\r" in singleStringToken
         newContent lexbuf}

140    | "\\b" {let newContent= content ^ "\b" in singleStringToken
         newContent lexbuf}

142    | "\\ " {let newContent= content ^ "\ " in singleStringToken
         newContent lexbuf}

144    | "\\" (digit as x) (digit as y) (digit as z) {let n=
145      let x=String.make 1 x in let y=String.make 1 y in let z=
         String.make 1 z in
146        (int_of_string z) + 10*(int_of_string y)
147        + 100*(int_of_string x) in if n > 255 then
148            raise (Failure "unknown char") else
149          let newContent= content ^ (String.make 1 (char_of_int n))
150              in singleStringToken newContent lexbuf }

152    | [^'''']+ as pstr {let newContent= content ^ pstr in
         singleStringToken newContent lexbuf}


155 {(* do not modify this function: *)
156  let lextest s = token (Lexing.from_string s)

158  let get_all_tokens s =
159      let b = Lexing.from_string (s^"\n") in
160      let rec g () =
161      match token b with EOF -> []
162      | t -> t :: g () in
163      g ()

165 let try_get_all_tokens s =
166     try (Some (get_all_tokens s), true)
167     with Failure "unmatched open comment" -> (None, true)
```

21

```
168          | Failure "unmatched closed comment" -> (None, false)
169    }
```

```
1  %{
2          open ProjCommon
3  %}
4
5  /* Define the tokens of the language: */
6
7  %token <string> STRING
8  %token <string> VARIABLE
9  %token <string> NAME
10 %token <float> FLOAT /*unsigned*/
11 %token <int> INT /*unsigned*/
12 %token DOT DOUBLEDOT
13 %token COLONHYPHEN QUESTIONHYPHEN
14 %token ARROW
15 %token NOT
16 %token TERM_EQ TERM_INEQ IS AS TERM_DECOMP TERM_UNIFY
       TERM_NOTUNIFY
17         ARITH_EQ ARITH_INEQ ARITH_LESS ARITH_GREATER ARITH_GEQ
18         ARITH_LEQ TERM_ORDER_EQ TERM_ORDER_INEQ
       TERM_ORDER_GREATER
19         TERM_ORDER_LESS TERM_ORDER_GEQ TERM_ORDER_LEQ
20 %token DOUBLECOLON
21 %token PLUS MINUS
22 %token MULT DIV
23         BITWISE_AND
24         BITWISE_OR BITWISE_NOT VAR_INSTANTIATED
25 %token SEMICOLON COMMA COLON
26 %token UMINUS UPLUS
27 %token CUT
28 %token LPAREN RPAREN LBRACKET RBRACKET PIPE
29 %token EOF
30
31 %token <bool> BOOL
32 %token DOLLAR
33
34 /* Define the "goal" nonterminal of the grammar: */
35 %type <ProjCommon.rules> rules
36 %start rules
37 %type <ProjCommon.program> program
38 %start program
39
40 %%
41 var:
42 | VARIABLE         { Var $1 }
43 name:
44 | NAME             { ConstTerm(StringConst $1) }
45 atomic_term:
```

```
46    |  var            {  $1  }
47    |  name           {  $1  }
48    |  INT               {  ConstTerm ( IntConst  $1 )  }
49    |  MINUS INT         {  ConstTerm ( IntConst (−$2 ) )  }
50    |  FLOAT             {  ConstTerm ( FloatConst  $1 )  }
51    |  MINUS FLOAT        {  ConstTerm ( FloatConst  (−.$2 ) )  }
52    |  STRING             {  ConstTerm ( StringConst  $1 )  }
53    |  BOOL             {  ConstTerm ( BoolConst  $1 )  }
54    |  LPAREN  term  RPAREN     {  $2  }
55    |  list_term            {  $1  }
56
57  compound_term_1 :
58     | atomic_term         {  $1  }
59     | DOLLAR atomic_term    {  CompoundTerm  ( "$" , [ $2 ] ) }
60
61  compound_term_200 :
62     | compound_term_1  { $1 }
63     | compound_term_1 VAR_INSTANTIATED compound_term_200 {
        CompoundTerm  ( "^" , [ $1 ; $3 ] ) }
64
65  compound_term_400 :
66     | compound_term_200  { $1 }
67     | compound_term_400 MULT compound_term_200 { CompoundTerm ( "∗"
        , [ $1 ; $3 ] ) }
68     | compound_term_400 DIV compound_term_200   { CompoundTerm ( "/"
        , [ $1 ; $3 ] ) }
69
70  compound_term_500 :
71     | compound_term_400  { $1 }
72     | compound_term_500 PLUS compound_term_400 { CompoundTerm ( "+"
        , [ $1 ; $3 ] ) }
73     | compound_term_500 MINUS compound_term_400 { CompoundTerm ( "−"
        , [ $1 ; $3 ] ) }
74     | compound_term_500 BITWISE_AND compound_term_400   {
        CompoundTerm  ( "\\/" , [ $1 ; $3 ] ) }
75     | compound_term_500 BITWISE_OR compound_term_400     {
        CompoundTerm  ( "/\\" , [ $1 ; $3 ] ) }
76
77  compound_term_600 :
78     | compound_term_500  { $1 }
79     | compound_term_500 COLON compound_term_600 { CompoundTerm ( " :"
        , [ $1 ; $3 ] ) }
80
81  compound_term_700 :
82     | compound_term_600 ARITH_EQ compound_term_600          {
        CompoundTerm  ( "=:=" , [ $1 ; $3 ] ) }
83     | compound_term_600 ARITH_INEQ compound_term_600          {
        CompoundTerm  ( "=\\=" , [ $1 ; $3 ] ) }
84     | compound_term_600 ARITH_GEQ compound_term_600         {
        CompoundTerm  ( ">=" , [ $1 ; $3 ] ) }
85     | compound_term_600 ARITH_LEQ compound_term_600         {
        CompoundTerm  ( "=<" , [ $1 ; $3 ] ) }
86     | compound_term_600 ARITH_GREATER compound_term_600      {
        CompoundTerm  ( ">" , [ $1 ; $3 ] ) }
```

```
87    |compound_term_600 ARITH_LESS compound_term_600        {
        CompoundTerm ("<",[$1;$3])}
88
89    |compound_term_600 TERM_UNIFY compound_term_600        {
        CompoundTerm ("=",[$1;$3])}
90    |compound_term_600 TERM_NOTUNIFY compound_term_600    {
        CompoundTerm ("\\=",[$1;$3])}
91    |compound_term_600 TERM_EQ compound_term_600         {
        CompoundTerm ("==",[$1;$3])}
92    |compound_term_600 TERM_INEQ compound_term_600       {
        CompoundTerm ("\\==",[$1;$3])}
93    |compound_term_600 IS compound_term_600              {
        CompoundTerm ("is",[$1;$3])}
94    |compound_term_600 AS compound_term_600              {
        CompoundTerm ("as",[$1;$3])}
95    |compound_term_600 TERM_DECOMP compound_term_600      {
        CompoundTerm ("=..",[$1;$3])}
96
97    |compound_term_600 TERM_ORDER_GEQ compound_term_600   {
        CompoundTerm ("@>=",[$1;$3])}
98    |compound_term_600 TERM_ORDER_LEQ compound_term_600      {
        CompoundTerm ("@=<",[$1;$3])}
99    |compound_term_600 TERM_ORDER_GREATER compound_term_600   {
        CompoundTerm ("@>",[$1;$3])}
100   |compound_term_600 TERM_ORDER_LESS compound_term_600     {
        CompoundTerm ("@<",[$1;$3])}
101   |compound_term_600 TERM_ORDER_EQ compound_term_600      {
        CompoundTerm ("=@=",[$1;$3])}
102   |compound_term_600 TERM_ORDER_INEQ compound_term_600     {
        CompoundTerm ("\\=@=",[$1;$3])}
103
104 compound_term_1050:
105    |compound_term_600 {$1}
106    |compound_term_700 {$1}
107    |compound_term_700 ARROW compound_term_1050 { CompoundTerm ("
        ->",[$1;$3])}
108
109
110   compound_term:
111    |compound_term_1050 {$1}
112
113 list_term:
114 | LBRACKET RBRACKET           { ListTerm [] }
115 | LBRACKET term_list RBRACKET    { ListTerm $2 }
116
117 term:
118 | compound_term        { $1 }
119 | NAME LPAREN term_list RPAREN  { PredAsTerm (Predicate ($1,$3))
        }
120 | NOT LPAREN term RPAREN     { PredAsTerm (Predicate ("not",[$3])
        ) }
121
122 predicate_list:
123 | predicate                { ([$1],[]) }
```

24

```
124 | | predicate COMMA predicate_list     { ($1::(fst $3), ","::snd($3
    |     )) }
125 | | predicate SEMICOLON predicate_list  { ($1::(fst $3), ";"::snd(
    |     $3)) }
126 |
127 | term_list:
128 | | term              { [$1] }
129 | | term COMMA term_list     { $1::$3 }
130 | | term PIPE term_list      { $1::$3 }
131 |
132 | predicate:
133 | | NAME LPAREN term_list RPAREN      { Predicate ($1,$3) }
134 | | NAME                 { Identifier $1 }
135 | | VARIABLE                { VarAsPred $1}
136 | | NOT LPAREN term RPAREN        { Predicate ("not",[$3]) }
137 |
138 |   |compound_term_600 ARITH_EQ compound_term_600        {
    |     Predicate ("=:=",[$1;$3])}
139 |   |compound_term_600 ARITH_INEQ compound_term_600        {
    |     Predicate ("=\\=",[$1;$3])}
140 |   |compound_term_600 ARITH_GEQ compound_term_600        {
    |     Predicate (">=",[$1;$3])}
141 |   |compound_term_600 ARITH_LEQ compound_term_600        {
    |     Predicate ("=<",[$1;$3])}
142 |   |compound_term_600 ARITH_GREATER compound_term_600     {
    |     Predicate (">",[$1;$3])}
143 |   |compound_term_600 ARITH_LESS compound_term_600        {
    |     Predicate ("<",[$1;$3])}
144 |   |compound_term_600 TERM_UNIFY compound_term_600        {
    |     Predicate ("=",[$1;$3])}
145 |   |compound_term_600 TERM_NOTUNIFY compound_term_600     {
    |     Predicate ("\\=",[$1;$3])}
146 |   |compound_term_600 TERM_EQ compound_term_600          {
    |     Predicate ("==",[$1;$3])}
147 |   |compound_term_600 TERM_INEQ compound_term_600         {
    |     Predicate ("\\==",[$1;$3])}
148 |   |compound_term_600 TERM_ORDER_GEQ compound_term_600    {
    |     Predicate ("@>=",[$1;$3])}
149 |   |compound_term_600 TERM_ORDER_LEQ compound_term_600      {
    |     Predicate ("@=<",[$1;$3])}
150 |   |compound_term_600 TERM_ORDER_GREATER compound_term_600   {
    |     Predicate ("@>",[$1;$3])}
151 |   |compound_term_600 TERM_ORDER_LESS compound_term_600     {
    |     Predicate ("@<",[$1;$3])}
152 |   |compound_term_600 TERM_ORDER_EQ compound_term_600      {
    |     Predicate ("=@=",[$1;$3])}
153 |   |compound_term_600 TERM_ORDER_INEQ compound_term_600     {
    |     Predicate ("\\=@=",[$1;$3])}
154 |   |compound_term_600 IS compound_term_600          { Predicate
    |     ("is",[$1;$3])}
155 |
156 | clause:
157 | | predicate DOT               { Fact $1 }
158 | | predicate COLONHYPHEN predicate_list DOT  { Rule ($1,$3) }
```

25

```
159
160 clause_list :
161 | clause          { [$1] }
162 | clause clause_list    { $1::$2 }
163
164 query :
165 | QUESTIONHYPHEN predicate_list DOT  { Query $2 }
166
167 rules :
168 | clause_list      { RuleList $1 }
169
170 program :
171 | rules query      { Prog ($1,$2) }
172 | query            { ProgFromQuery $1 }
```

Listing 4 : Unify.ml: Code for unification

```
1  open ProjCommon
2
3
4  (*Find the value corresponding to the key in the subst if found
         *)
5  let rec getTermFromSubst subst key =
6   match subst with
7     [] -> None |
8    (var,term)::tail ->
9      (if var = key then (Some term)
10       else (getTermFromSubst tail key));;
11
12 (* Given a substitution and a variable, the replacement term
        will be returned.  *)
13 let rec subst_fun subst = fun strVar -> match subst with
14         [] -> Var strVar  |
15         (var, term)::tail -> if strVar = var then term
16                 else subst_fun tail strVar;;
17
18 (*Substitute all the matching vars inside a term*)
19 let rec term_lift_subst subst term = match term with
20                                      Var v -> (subst_fun subst
        v)
21                          |          ConstTerm c -> (term)
22          |           CompoundTerm(f,tl) -> CompoundTerm(f, List.
        map (term_lift_subst subst) tl)
23          |           ListTerm(tl) -> ListTerm(List.map (
        term_lift_subst subst) tl)
24          | (PredAsTerm pred) -> (PredAsTerm (substInPredicate
        subst pred))
25
26
27 and substInPredicate subst predicate = match predicate with
28            Identifier _ -> predicate |
```

26

```ocaml
29            Predicate(f,tl) -> Predicate(f, List.map (
     term_lift_subst subst) tl) |
30            VarAsPred v -> (match (getTermFromSubst subst v)
     with
31              None -> (VarAsPred v) |
32              Some term -> (match term with
33                Var x -> (VarAsPred x) |
34                ConstTerm c -> (match c with
35                  BoolConst b -> (Identifier (
     string_of_const c)) |
36                  _ -> (raise (Failure ("Callable is
     expected, found "^ (string_of_term term)))) ) |
37
38                CompoundTerm(f,tl) -> (if (ProjCommon.
     retBool f) then (Predicate(f,tl))
39                  else (raise (Failure ("Callable is
     expected, found "^ (string_of_term term)))) ) |
40
41                ListTerm(tl) -> (raise (Failure ("Callable
     is expected, found "^ (string_of_term term)))) |
42                PredAsTerm(pred) -> (pred) ));;
43
44
45 (*Occurs check*)
46 let rec occurs x term = match term with
47      Var y -> (x = y) |
48      ConstTerm c -> false |
49      CompoundTerm(f,tl) -> occursInList x tl |
50      ListTerm(tl) -> occursInList x tl   |
51      PredAsTerm pred ->
52        (match pred with
53         Identifier _ -> false |
54         Predicate(f,tl) -> occursInList x tl |
55         VarAsPred v -> (if x = v then true else false))
56
57      and occursInList x termList = match termList with
58        [] -> false |
59        h::t -> if (occurs x h) then true else
60          occursInList x t;;
61
62
63
64 let rec occursInSndPartOfSubst x sigma = match sigma with
65             [] -> false |
66             (k,v)::tail -> if occurs x v then true
67                 else occursInSndPartOfSubst x tail;;
68
69 let rec occursInSubstList x lst = match lst with
70                                [] -> false |
71             (k,v)::tail -> (if x=k then true else
     occursInSubstList x tail);;
72
73
74 (*substitute the constraints list*)
```

27

```
75  let rec eqlist_subst subst constraintList= match constraintList
       with
76                  [] -> [] |
77                  (term1, term2) :: tail -> (term_lift_subst subst
     term1,
78                         term_lift_subst subst term2)
79                  :: eqlist_subst subst tail;;
80
81  let rec updateVarInSubst subst sigma =
82    match sigma with
83    [] -> [] |
84    (k,v)::tail -> ((subst_fun subst k), v)::(updateVarInSubst
     subst tail) ;;
85
86  let rec updateSubst sigma substList = match substList with
87                  [] -> Some [] |
88                  (x, term)::tail -> if occursInSndPartOfSubst x sigma
        then None
89                      else
90                      match (updateSubst sigma tail) with
91                        None -> None |
92                        Some tailResultList -> Some ((x,
     term_lift_subst sigma term) :: tailResultList);;
93
94  let rec updateListWithElement lst (key, value)= match lst with
95                  [] -> [(key, value)] |
96                  (k, v)::tail ->
97                     if k=key then (updateListWithElement tail (key,
     value))
98                     else (k, v):: (updateListWithElement tail (key,
     value));;
99
100 let rec updateListWithList newList oldList = match newList with
101                  [] -> oldList |
102                  keyValPair :: tail -> updateListWithList tail (
103                  updateListWithElement oldList keyValPair);;
104
105 let rec composeSubst sigma substList = match substList with
106                  [] -> Some sigma |
107                  (x, mt)::tail ->
108                     match (updateSubst sigma substList) with
109                       None -> None |
110                       Some sigma2 ->
111                     Some (updateListWithList sigma2 sigma);;
112
113
114 let rec genPairList list1 list2 = if (List.length list1 != List.
     length list2)
115            then None
116             else match (list1, list2) with
117            ([], _) -> Some [] |
118            (_, []) -> Some [] |
119            (h1::t1, h2::t2) -> let tailR=genPairList t1 t2 in
120              match tailR with None -> None |
```

```ocaml
121                    Some tailResultList ->
122                       Some ((h1,h2)::tailResultList);;
123
124
125
126 let getHead termList = match termList with
127           [] -> None |
128                 h::_ -> Some h;;
129
130
131 let getTail tl =
132                        match tl with
133           [] -> (ListTerm []) |
134           [t] -> (ListTerm []) |
135           [_;term] -> (match term with
136                     Var _ -> (term) |
137                 _ -> (ListTerm [term])
138               ) |
139           _::tail -> (ListTerm tail) ;;
140
141
142
143 (*Unify the body part of a rule.*)
144 let rec unify eqlst = match eqlst with
145        [] -> Some [] |
146        (s,t)::eqlst' -> (
147          if s = t then unify eqlst'          (*delete rule*)
148          else (match (s,t) with
149            (ConstTerm _, Var _) -> (unify ((t,s)::eqlst')) |
150
151            (CompoundTerm(_,_), Var _) -> (unify ((t,s)::eqlst'))
      |
152
153            (ListTerm _, Var _) -> (unify ((t,s)::eqlst')) |   (*
      orient rule*)
154
155            (CompoundTerm(f1, termList1), CompoundTerm(f2,
      termList2) ) -> (
156               if f1 = f2 then (
157                let pairList= (genPairList termList1 termList2)
158                in (match pairList with
159                  None -> None |
160                  Some newConstraints -> unify (eqlst' @
      newConstraints)   )
161
162
163 ) else (None) )                    (*Decompose rule*)
164
165                | (ListTerm tl1, ListTerm tl2) -> (
166              if (tl1 = [] && tl2 = []) then unify eqlst'
167              else(
168
169            let tl1Head= getHead tl1 in
170            let tl1Tail= getTail tl1 in
```

```
171                let tl2Head= getHead tl2 in
172                let tl2Tail= getTail tl2 in
173
174              match (tl1Head, tl2Head) with
175                (None, _) -> (None) |
176                (_, None) -> (None) |
177                (Some head1, Some head2) -> (
178                  let newConstraints= [(head1,head2);(tl1Tail,
     tl2Tail)] in
179                    unify (eqlst' @ newConstraints)
180                  )
181
182                ) )
183
184
185 | (Var x, _) -> (
186                          if (x = "_") then (Some [])
187                          else if (occurs x t) then (None)
188        else (
189                              let eqlst''= eqlist_subst ([(x,t
     )]) eqlst' in
190          let sigmaResult= unify eqlst'' in
191            match sigmaResult with
192              None -> None |
193              Some sigma ->
194
195          let sigma2= [(x, term_lift_subst sigma t)] in
196
197          composeSubst sigma2 sigma) )    (*eliminate rule*)
198
199 | _ -> (None)
200
201 )) ;;
202
203
204 (* Used to unify the query and head *)
205 let rec unifyHead eqlst = match eqlst with
206        [] -> Some [] |
207        (s,t)::eqlst' -> (
208          if s = t then unifyHead eqlst'       (*delete rule*)
209          else (match (s,t) with
210            (ConstTerm _, Var _) -> (unifyHead ((t,s)::eqlst')) |
211
212            (CompoundTerm(_,_), Var _) -> (unifyHead ((t,s)::eqlst
     ')) |
213
214            (ListTerm _, Var _) -> (unifyHead ((t,s)::eqlst')) |
     (*orient rule*)
215
216            (CompoundTerm(f1, termList1), CompoundTerm(f2,
     termList2) ) -> (
217              if f1 = f2 then (
218              let pairList= (genPairList termList1 termList2)
219               in (match pairList with
```

30

```ocaml
220                 None -> None |
221                 Some newConstraints -> unifyHead (eqlst ' @
       newConstraints) )
222
223
224 ) else (None) )                    (*Decompose rule*)
225
226                 | (ListTerm tl1, ListTerm tl2) -> (
227              if (tl1 = [] && tl2 = []) then unify eqlst '
228               else (
229
230             let tl1Head= getHead tl1 in
231             let tl1Tail= getTail tl1 in
232             let tl2Head= getHead tl2 in
233             let tl2Tail= getTail tl2 in
234
235             match (tl1Head, tl2Head) with
236               (None, _) -> (None) |
237               (_, None) -> (None) |
238               (Some head1, Some head2) -> (
239                 let newConstraints= [(head1,head2);(tl1Tail,
       tl2Tail)] in
240                  unify (eqlst ' @ newConstraints)
241                 )
242
243                 )
244
245             )
246
247
248 | (Var x, _) ->    (
249                        if (x = "_") then (Some [])
250                        else (if (occurs x t) then (None)
251        else ( match (unifyHead eqlst ') with
252          None -> None |
253          Some tailResult -> if (occursInSubstList x tailResult)
       then
254                          (match t with
255               Var _ -> Some tailResult |
256                _ -> Some (updateListWithElement tailResult (x,t))
        )
257
258                        else (Some ((x,t)::tailResult) ) ) ))
259
260         (*eliminate rule*)
261
262 | _ -> (None)
263
264 ));;
265
266
267 let rec unifyPredicates (pred1,pred2) =
268   match (pred1, pred2) with
```

31

```
269        ( Identifier id1 , Identifier id2 ) −> ( if id1=id2 then Some [ ]
            else None ) |
270        ( Identifier id1 , Predicate ( f , t l ) ) −> ( None ) |
271        ( Identifier _, VarAsPred _) −> ( raise ( Failure "Do not waste
            your time ! Maybe after 1000 years we can get the answer ?" ) )
            |
272        ( Predicate ( f , t l ) , Identifier id2 ) −> ( None ) |
273        ( Predicate ( f1 , tl1 ) , Predicate ( f2 , tl2 ) ) −> (
274          if ( not ( f1 = f2 ) ) then None
275          else ( match ( genPairList tl1 tl2 ) with
276            None −> None |
277            Some eqlst −> ( unifyHead eqlst ) ) ) |
278        ( Predicate _, VarAsPred _) −> ( raise ( Failure "Do not waste
          your time ! Maybe after 1000 years we can get the answer ?" ) ) |
279        ( VarAsPred _, _) −> ( raise ( Failure "Do not waste your time !
          Maybe after 1000 years we can get the answer ?" ) ) ; ;
```

Listing 5 :   Evaluator.ml: Code for evaluating terms

```
 1  open ProjCommon
 2  open Unify
 3
 4  let rec updateMemory m x v = match m with
 5            [ ] −> [ ( x , v ) ] |
 6
 7            ( x0 , v0 ) : : t a i l −> if x0=x then ( x , v ) : : t a i l else ( x0 , v0 )
        : : ( updateMemory t a i l x v )
 8   ; ;
 9
10  let const_to_val c = match c with
11          BoolConst b −> BoolVal b |
12          IntConst i −> IntVal i |
13          FloatConst f −> FloatVal f |
14          StringConst s −> StringVal s ; ;
15
16  let rec val2Term value = match value with
17          BoolVal b −> ConstTerm ( BoolConst b ) |
18          IntVal i −> ConstTerm ( IntConst i ) |
19          FloatVal f −> ConstTerm ( FloatConst f ) |
20          StringVal s −> ConstTerm ( StringConst s ) |
21          ListVal vl −> ListTerm ( List . map ( val2Term ) vl ) ; ;
22
23  let monOpApply op v = match op with
24                  "not" −> ( match v with BoolVal true −> ( BoolVal
        false ) |
25            BoolVal false −> ( BoolVal true ) |
26            _ −> raise ( Failure "only support negation of boolean
        values at current stage ." ) ) |
27
28        "−" −> ( match v with IntVal i −> IntVal (−i ) |
29                  _ −> raise ( Failure "IntNegOp can only operate on
        integers !" ) ) |
```

```
30
31        _ -> (raise (Failure "Not supported this op at present"));;
32
33
34
35
36  let nlOpApply () = print_newline (); true;;
37
38  let writeOpApply contentsTerm =
39                    match contentsTerm with
40         ConstTerm(StringConst contents) -> (
41                    print_string (contents); true) |
42                    _ -> raise (Failure "Write is a built-in
     operation which takes only one string as argument!");;
43
44  let rec binOpApply binop (v1,v2) = match binop with
45         "+" -> (match (v1, v2) with
46              (IntVal i1, IntVal i2) -> (IntVal (i1 + i2)) |
47
48              (FloatVal f1, FloatVal f2) -> (FloatVal (f1 +. f2))
     |
49
50     (IntVal i1, FloatVal i2) -> (FloatVal (float_of_int i1 +. i2
     )) |
51     (FloatVal i1, IntVal i2) -> (FloatVal (float_of_int i2 +. i1
     )) |
52
53     (StringVal s1, StringVal s2)->(StringVal (s1 ^ s2)) |
54
55              _ -> raise (Failure "Unsupported operands!") ) |
56
57         "-" -> (match (v1, v2) with
58              (IntVal i1, IntVal i2) -> (IntVal (i1 - i2)) |
59
60              (FloatVal f1, FloatVal f2) -> (FloatVal (f1 -. f2))
     |
61
62     (IntVal i1, FloatVal i2) -> (FloatVal (float_of_int i1 -. i2
     )) |
63     (FloatVal i1, IntVal i2) -> (FloatVal (i1 -. float_of_int i2
     )) |
64
65              _ -> raise (Failure "Unsupported operands!") ) |
66
67         "*" -> (match (v1, v2) with
68              (IntVal i1, IntVal i2) -> (IntVal (i1 * i2)) |
69
70              (FloatVal f1, FloatVal f2) -> (FloatVal (f1 *. f2))
     |
71
72     (IntVal i1, FloatVal i2) -> (FloatVal (float_of_int i1 *. i2
     )) |
73     (FloatVal i1, IntVal i2) -> (FloatVal (float_of_int i2 *. i1
     )) |
```

```ocaml
74
75                   _ -> raise (Failure "Unsupported operands!") ) |
76
77          "/" -> (match (v1, v2) with
78              (IntVal i1, IntVal i2) -> (if i2 = 0 then raise (
       Failure "ERROR: divide by 0") else (IntVal (i1 / i2))) |
79
80              (FloatVal f1, FloatVal f2) -> (if f2 = 0.0 then
       raise (Failure "ERROR: divide by 0") else (FloatVal (f1 /. f2
       ))) |
81
82              (IntVal i1, FloatVal f2) -> (if f2 = 0.0 then raise
       (Failure "ERROR: divide by 0") else (FloatVal (float_of_int
       i1 /. f2))) |
83
84              (FloatVal f1, IntVal i2) -> (if i2 = 0 then raise (
       Failure "ERROR: divide by 0") else (FloatVal (f1 /.
       float_of_int i2))) |
85
86              _ -> raise (Failure "Both operands should be
       integers!") ) |
87
88
89
90
91
92          "::" -> (match (v1, v2) with
93              (_, ListVal vl) -> (ListVal (v1::vl)) |
94              _ -> raise (Failure "second argument should be a
       list!") ) |
95
96          "," -> (match v1 with BoolVal false -> BoolVal false |
97            BoolVal true -> (
98              match v2 with BoolVal false -> BoolVal false |
99            BoolVal true -> (
100               BoolVal true
101             ) |
102             _-> raise (Failure ", can only connect boolean values!
       ")
103             ) |
104             _-> raise (Failure ", can only connect boolean values!
       ") ) |
105
106
107         ";" -> (match v1 with BoolVal true -> BoolVal true |
108           BoolVal false -> (
109             match v2 with BoolVal false -> BoolVal false |
110           BoolVal true -> (
111             BoolVal true
112           ) |
113           _-> raise (Failure "; can only connect boolean values!
       ")
114           ) |
```

```
115          _-> raise (Failure ";  can only connect boolean values!
         ") ) |
116
117
118       "=:=" -> (match (v1, v2) with
119            (IntVal i1,  IntVal i2) -> (BoolVal (i1 = i2)) |
120
121            (FloatVal f1,  FloatVal f2) -> (BoolVal (f1 = f2)) |
122
123     (IntVal i1,  FloatVal i2) -> (BoolVal (float_of_int i1 = i2))
             |
124     (FloatVal i1,  IntVal i2) -> (BoolVal (float_of_int i2 = i1))
             |
125
126     (StringVal s1,  StringVal s2) -> (BoolVal (s1 = s2)) |
127
128     (BoolVal b1,  BoolVal b2) -> (BoolVal (b1 = b2)) |
129
130            _ -> raise (Failure "Unsupported operands!") ) |
131
132
133
134       ">" ->   (match (v1, v2) with
135            (IntVal i1,  IntVal i2) -> (BoolVal (i1 > i2)) |
136
137            (FloatVal f1,  FloatVal f2) -> (BoolVal (f1 > f2)) |
138
139     (IntVal i1,  FloatVal i2) -> (BoolVal (float_of_int i1 > i2))
             |
140     (FloatVal i1,  IntVal i2) -> (BoolVal (float_of_int i2 < i1))
             |
141
142     (StringVal s1,  StringVal s2)->(BoolVal (s1 > s2)) |
143
144            _ -> raise (Failure "Unsupported operands!") ) |
145
146       ">=" -> (binOpApply ";" ((binOpApply ">" (v1,v2)), (
        binOpApply "=" (v1,v2) ) )) |
147
148       "<" -> (binOpApply ">" (v2,v1)) |
149
150       "<=" -> (binOpApply ";" ((binOpApply "<" (v1,v2)), (
        binOpApply "=" (v1,v2) ) )) |
151
152       "=\\=" -> (binOpApply ";" (binOpApply ">" (v1,v2),
        binOpApply "<" (v1,v2) ) ) |
153
154       "mod" -> (match (v1, v2) with
155            (IntVal i1,  IntVal i2) -> (IntVal (i1 mod i2)) |
156            _ -> raise (Failure "Both operands should be
        integers!") ) |
157
158       "**" ->   (match (v1, v2) with
```

```ocaml
159                (IntVal i1, IntVal i2) -> (FloatVal (float_of_int i1
        ** float_of_int i2)) |
160
161                (FloatVal f1, FloatVal f2) -> (FloatVal (f1 ** f2))
        |
162
163      (IntVal i1, FloatVal i2) -> (FloatVal (float_of_int i1 ** i2
        )) |
164      (FloatVal i1, IntVal i2) -> (FloatVal (i1 ** float_of_int i2
        )) |
165        _ -> raise (Failure "Unsupported operands!")) |
166
167          _ -> raise (Failure "Unsupported operations!")
168
169
170        ;;
171
172
173 let rec typeTest ty term = match ty with
174                            "var" -> (match term with
175                Var _ -> true |
176                _ -> false)
177
178      | "nonvar" -> (match term with
179                Var _ -> false |
180                _ -> true)
181
182      | "atom" -> (match term with
183        ConstTerm(StringConst _) -> true |
184        _ -> false)
185
186      | "integer" -> (match term with
187        ConstTerm(IntConst _) -> true |
188        _ -> false)
189
190      | "float" -> (match term with
191        ConstTerm(FloatConst _) -> true |
192        _ -> false)
193
194      | "number" ->  (typeTest "integer" term) ||
195          (typeTest "float" term)
196
197      | "atomic" -> (typeTest "atom" term) ||
198              (typeTest "number" term)
199
200      | "compound" -> (match term with
201        CompoundTerm(_,_) -> true |
202        _ -> false)
203
204      | "callable" -> (typeTest "atom" term) ||
205            (typeTest "compound" term)
206
207      | "list" -> (match term with
208        ListTerm _ -> true
```

```
209          | _ -> false)
210
211        | "is_list" -> (match term with
212          ListTerm _ -> true
213          | _ -> false)
214
215        | _ -> (print_string "This operation is not supported yet"
     ;
216        false);;
217
218
219
220 let rec eval_term term = match term with
221        ConstTerm(t) -> (const_to_val t)
222      | Var(x) -> (raise (Failure "Var Not Instantiated yet"))
223
224      |        CompoundTerm(f,tl) -> (if isBuiltInOp f then
225          (if (isTypeTesting f) then
226                        (if (List.length tl)!=1  then (BoolVal
     false) else BoolVal(typeTest f (List.hd tl)) )
227              else (
228                        if (List.length tl) != 2 then raise (
     Failure "number of args to the function is wrong")
229                    else (let eq= (List.hd tl, List.nth tl 1) in
230
231                        (match f with
232                      "=" -> (match (Unify.unify [eq]) with
233                        None -> BoolVal(false) |
234                        (Some sig0) -> BoolVal(true) ) |
235
236                      "is" -> (let lhs= fst eq in
237                          let rhs= snd eq in
238                          let rhsVal = eval_term rhs in
239
240                          match lhs with
241                          ConstTerm _ -> (binOpApply "=:=" (
     eval_term lhs,rhsVal) ) |
242                            Var x -> BoolVal(true) |
243                            _ -> BoolVal(false)) |
244                        _ -> (raise (Failure "Not supported yet."))
     ) )
245
246                                        ) )
247
248        else ( if (List.length tl) == 2 then (binOpApply f (
     eval_term (List.hd tl), eval_term (List.nth tl 1))) else (
     raise (Failure "Not supported yet")) ) )
249
250      |        ListTerm(tl) -> (ListVal (List.map (eval_term) tl)
     )
251
252      |        PredAsTerm _ -> (raise (Failure "should not
     evaluate a predicate term here!"))
253
```

```
254        ;;
```

Listing 6 :   Interpreter.ml: Code for interpreting queries

```
 1  open ProjCommon
 2  open Evaluator
 3  open Unify
 4
 5  let rec filter preserveList subst = match subst with
 6                                        [] -> [] |
 7         (v,term)::tail -> (
 8              if occursInList v preserveList then
 9               ((v,term)::(filter preserveList tail))
10              else (filter preserveList tail) )
11  and occursInList item list = match list with
12                                        [] -> false |
13                                       h::tail -> if item = h then true
     else
14                                                      occursInList item
     tail;;
15
16  let rec rmLastItem list = match list with [] -> [] |
17                                            [h] -> [] |
18              h::t -> h::(rmLastItem t);;
19
20  let getBool boolVal = match boolVal with
21                        BoolVal true -> true |
22           BoolVal false -> false |
23           _ -> (raise (Failure "Expect a bool val!"));;
24
25  (*rename the free vars in the body of the rule before applying
     the subst gen from the unification of head and query.*)
26  let rec renameFreeVarsInClause avoidList clause =
27              match clause with
28                Fact fp -> [fp] |
29                Rule (headPred, (body,connList)) -> (
30          let freeVarsInBody= ProjCommon.freeVarsInClause clause
     in
31          let numOfFreeVars= List.length freeVarsInBody in
32          if numOfFreeVars = 0 then (body) else(
33
34          let binders= ProjCommon.freeVarsInPredicate headPred in
35          let genFreshVars= ProjCommon.get_n_freshVars
     numOfFreeVars (avoidList @ binders) in
36          let subst4Fresh= genSubst4Fresh freeVarsInBody
     genFreshVars in
37          (
38
39  (*
40     print_string ("Before renaming, clause is "^(ProjCommon.
     string_of_clause clause));
```

38

```ocaml
41        print_string ("After renaming, pred list is "^(ProjCommon.
       stringOfPredList
42        (List.map (Unify.substInPredicate subst4Fresh) body)
       connList)^"\n");    *)
43
44        List.map (Unify.substInPredicate subst4Fresh) body)
45                )
46      )
47
48 and genSubst4Fresh oldVarList newNameList =
49    if (List.length oldVarList) != (List.length newNameList) then
       (raise (Failure "cannot gen subst 4 fresh because two list
        have diff len."))
50   else (match (oldVarList, newNameList) with
51        ([] ,_) -> [] |
52        (_,[]) -> [] |
53        (v1::tail1 , n1::tail2) -> ( (v1,Var n1)::genSubst4Fresh
       tail1 tail2 ) )
54       ;;
55
56
57 let rec getAVList subst =
58    let termList= getTermList subst in
59     (getVarStrList termList)
60
61 and getTermList sigma=
62  match sigma with
63  []-> [] |
64  (v,t)::tail0 -> (t::(getTermList tail0) )
65
66 and getVarStrList termList =
67    match termList with
68     [] -> [] |
69     term::tail -> (match term with
70       Var x -> (x::(getVarStrList tail)) |
71       _ -> (getVarStrList tail) );;
72
73 (* consult the predicate in user-defined rules *)
74
75 let rec consultSinglePred_debug (rules, usedRules) predicate
       avlist   debug = match rules with
76                                            RuleList ([]) -> (
       false, []) |
77                                            RuleList (clause::tail
       ) -> (match clause with
78                       Fact fp -> (
79                           let _=(
80                           if debug then(
81                           print_string ((string_of_predicate
       predicate) ^ " is trying to match fact "
82                             ^ (string_of_predicate fp) ^ "\n");)
       else ()) in
83                           match(Unify.unifyPredicates (fp,predicate
       ) ) with
```

39

```ocaml
                         None -> (consultSinglePred_debug (RuleList
     tail, (usedRules @ [clause])) predicate avlist debug ) |
                         Some sig0 -> (true, sig0) ) |

                         Rule (headPred, (body, connList)) -> (
                          let _=( if debug then(
                            print_string ("\n" ^ (
     string_of_predicate predicate) ^ " is trying to match the "
                            ^ (string_of_clause (clause)) ^ "\n");)
     else ()) in

                            match (Unify.unifyPredicates (headPred,
     predicate)) with
                               None -> (consultSinglePred_debug (
     RuleList tail, (usedRules @ [clause]))
                               predicate avlist debug) |
                               Some sig0 -> (

let _= ( if debug then
(print_string ("\n after unifying head and query, the following
     subst function is gen:\n" ^
       (ProjCommon.string_of_subst sig0)^"\n");) else ()) in

                                             let avoidList= getAVList
     sig0 @ avlist in
                            let renamedBody= renameFreeVarsInClause
      avoidList clause in

let _= ( if debug then (
print_string ("\nAfter renaming, the body of the rule becomes:\n
     " ^
       (ProjCommon.stringOfPredList renamedBody connList) ^"\n"
     );) else ()) in
                            let newBody= List.map (substInPredicate
      sig0) renamedBody in

let _= ( if debug then (
print_string ("\nAfter applying the subst function gen from
     unification, new body is :\n"
       ^ (ProjCommon.stringOfPredList newBody connList) ^"\n" )
     ;) else ()) in

                               match (consult_debug (RuleList(
     usedRules @ (clause::tail)))
                                    (Query (newBody, connList)) true
     avoidList debug ) with
                                    (false,_) -> (false,[]) |
                                    (true, tailSig) -> (match (Unify.
     composeSubst tailSig sig0) with
                                          None -> (true, []) |
                                          Some finalSig0 ->

let finalSig = (let updatedEQList= Unify.updateVarInSubst sig0
     finalSig0 in
```

40

```ocaml
120
121      match (Unify.unify updatedEQList) with
122        None -> [] |
123        Some fs -> fs ) in
124  (true, finalSig) ) ) ) )
125
126     (*Consult a list of predicates*)
127     (*returns a result which is of the form bool * subst*)
128     (*predicates are left-assoc! But it needs to eval from left
         to right!*)
129 and consult_debug rules query lastBool avlist debug =
130      match query with Query(predList, connList) -> (
131
132      match predList with
133    [] -> (raise (Failure "Query is empty!")) |
134
135    [singlePred] -> (let (singleBool, singleSig) =(
       eval_predicate_debug rules singlePred avlist debug) in
136        match connList with
137          ","::connTail -> ((singleBool && lastBool), singleSig) |
138          ";"::connTail -> ((singleBool || lastBool), singleSig) |
139          _ -> (raise (Failure "Not enough connectives or unknown
       connective."))) |
140
141        fstPred::tailPredList -> (let (fstBool, fstSig) = (
       eval_predicate_debug rules fstPred avlist debug) in
142                                 (*we should only add things to
       the subst list when absolutely needed*)
143                                 let freeVarsInFstPred=
       ProjCommon.freeVarsInPredicate fstPred in
144         let refinedSig= filter freeVarsInFstPred fstSig in
145
146                                 let newTailPredList= List.map
       (substInPredicate refinedSig) tailPredList in
147
148 let _= (if debug then (
149
150           print_string ((stringOfPredList predList (connList)) ^
       " is the old pred list\n");
151           print_string ("after eval first predicate, subst
       derived is "^(string_of_subst fstSig) ^ "\n");
152           print_string ((stringOfPredList newTailPredList (List.
       tl connList)) ^ " is the updated tail pred list\n");
153 ) else ()
154 ) in
155
156                                 let newLastBool= (match
       connList with
157            ","::_ -> (fstBool && lastBool) |
158            ";"::_ -> (fstBool || lastBool) |
159            _ -> (raise (Failure "unknown connective."))) in
160          (consult_debug rules (Query(newTailPredList, List.tl
       connList)) newLastBool avlist debug)
161                                        ) )
```

```
162
163
164
165  (* Consult a predicate which is either user−defined or built−in
        function *)
166  and eval_predicate_debug rules predicate avlist debug = match
        predicate with
167             VarAsPred v −> (raise (Failure "Do not waste your
        time! Maybe after 1000 years we can get the answer?")) |
168
169                               Identifier fact −> (
170
171  let _=(if debug then (
172  print_string ((fact)^" is a fact!");
173  ) else ()) in
174
175                               match fact with
176                               "true" −> (true ,[]) |
177             "false" −> (false ,[]) |
178             "nl" −> (Evaluator.nlOpApply () ,[]) |
179             _ −> consultSinglePred_debug (rules ,[]) predicate
        avlist debug) |
180
181
182            Predicate (f, tl) −> (if (ProjCommon.isBuiltInOp f)
        (* It is built in operation *)
183
184                               then (
185  let _=( if debug then (
186   print_string (f ^ " is a built−in op.\n");
187  ) else () ) in
188                               if (List.length tl) == 1
        then (
189                   let singleTerm = (List.hd tl) in
190
191                   if (ProjCommon.isTypeTesting f) then  (* it is
        type testing *)
192                       (Evaluator.typeTest f singleTerm , [])
193
194                   else (if f = "not" then (getBool (Evaluator.
        monOpApply f (Evaluator.eval_term singleTerm)), [])
195                   else if (f = "write") then ((Evaluator.
        writeOpApply singleTerm) ,[])
196                   else (raise (Failure "cannot generate goal
        from this unary op.")) )
197                               )
198
199                   else (       (* other built−in ops *)
200                               if (List.length tl) != 2 then raise (
        Failure "At the moment, this function is not supported.")
201                   else (let eq= (List.hd tl, List.nth tl 1) in
202
203                       (match f with
204                   "=" −> (match (Unify.unify [eq]) with
```

```
205                            None −> (false, []) |
206                            (Some sig0) −> (true , sig0) ) |
207
208                       "is" −> (let lhs= fst eq in
209                            let rhs= snd eq in
210                            let rhsVal = Evaluator.eval_term rhs in
211
212
213
214                            match lhs with
215                            ConstTerm _ −> (
216
217 let _=( if debug then (
218                       print_string ("\nlhs is "^ (string_of_term
      lhs)
219                                    ^ " and it is constant,  eq op be
      applied on "
220                                    ^ (string_of_term lhs) ^" and "^ (
      string_of_term rhs) ^ "\n" );
221
222 ) else () ) in
223
224
225                       match(Evaluator.binOpApply "=:=" (Evaluator.
      eval_term lhs,rhsVal)) with
226                                            BoolVal false −> (false
      ,[]) |
227                            BoolVal true −> (true,[]) |
228                            _ −> (raise (Failure "should ret
      BoolVal")) ) |
229                            Var x −>
230
231 let _=( if debug then (
232         print_string ("\nlhs is "^(string_of_term lhs) ^", and
      it is a var\n");
233 ) else () ) in
234
235                       (true, [(x,Evaluator.val2Term rhsVal)]) |
236                            _ −> (false, [])) |
237
238                       _ −> (if (ProjCommon.retBool f) then (match
      (binOpApply f (eval_term (fst eq), eval_term (snd eq)))
239                            with BoolVal true −> (true,[]) |
240                            BoolVal false −> (false,[]) |
241                            _ −> (raise (Failure "Unknown
      exception."))  )
242
243                       else (raise (Failure "predicate should
      return boolean!")) ) )) )  )
244
245                                      else  (
246 let _=( if debug then (
247 print_string ((ProjCommon.string_of_predicate predicate)^" is
      user−defined!\n");
```

43

```
248 ) else ( ) ) in
249
250                     consultSinglePred_debug ( rules ,[ ] )  predicate
        avlist  debug) )   (* User defined functions *)
251
252
253 and consultSinglePred ( rules ,unusedRules )  predicate  avlist =
        consultSinglePred_debug ( rules ,unusedRules )  predicate  avlist
        false
254
255 and eval_predicate rules predicate avlist = eval_predicate_debug
         rules  predicate  avlist  false
256
257 and consult rules query lastBool avlist = consult_debug rules
        query  lastBool  avlist  false ;;
```

```
 1 open ProjCommon
 2 open Lexer
 3 open Parser
 4 open Unify
 5 open Evaluator
 6 open Interpreter
 7
 8
 9 (* Debugger *)
10 let rec printDebug indexedRules usedRules sigma pred avlist
       blacklist =
11 print_string " \nCur rules :\n";
12 print_string (ProjCommon.stringOfIndexedRules indexedRules );
13
14 print_string " \nUsed rules :\n";
15 print_string (ProjCommon.stringOfIndexedRules usedRules );
16
17 print_string " \nSigma:\n";
18 print_string (ProjCommon.string_of_subst sigma );
19
20 print_string " \nCur pred is :\n";
21 print_string (ProjCommon.string_of_predicate pred );
22
23 print_string " \nAvoid list is :\n";
24 print_string (ProjCommon.string_of_stringList avlist );
25
26 print_string " \nBlacklist is :\n";
27 print_string (ProjCommon.stringOfBlackList blacklist );
28
29 print_string " \n\n";
30 let _=(read_line ()) in ()
31 ;;
32
33
```

```ocaml
34  (* print result *)
35  let rec printResultList resultList =
36    match resultList with
37    [] -> () |
38    curResult::tail -> (printResult curResult; printResultList
         tail)
39
40  and printResult result =
41  match result with
42  (b,sigma) -> (
43  if b=false then ()
44  else
45  (print_string ((ProjCommon.string_of_subst sigma) ^ "\n");) ) ;;
46
47
48
49  let rec getIndexedRulesHelper rules curIndex = (
50    match rules with
51    RuleList cl -> (match cl with
52         [] -> [] |
53         curClause::tail ->
54         ((curIndex, curClause)::
55         (getIndexedRulesHelper (RuleList tail) (curIndex+1) ))));;
56
57  let getIndexedRules rules = getIndexedRulesHelper rules 0;;
58
59  let rec getClauseWithIndex indexedRules i =
60      match indexedRules with
61    [] -> None |
62    (index,clause)::tail -> (if index=i then Some clause
63           else getClauseWithIndex tail i )
64  ;;
65
66
67  (*check whether a given str is in the black list of a rule*)
68  let rec isInBlackList indexedRules blacklist i pred =
69    let predStr= string_of_predicate pred in
70          let isPredAllVars= (ProjCommon.onlyVarsInPred pred) in
71
72    match (getItemWithIndexI blacklist i) with
73    None -> false |
74    Some listI ->
75     let isHeadPredAllVars =
76    (
77        match (getClauseWithIndex indexedRules i)
78    with      None -> (raise (Failure ("Rule " ^ (string_of_int i)
        ^ " is not in the knowledge base.")))
79
80    | Some ruleI -> (
81             let headI = (match ruleI with
82           Fact hp -> hp |
83           Rule (headPred,_) -> headPred ) in
84
85      ProjCommon.onlyVarsInPred headI)
```

```
86
87
88    ) in (
89       if isHeadPredAllVars && isPredAllVars
90       then ( true )
91       else
92       ( if occursIn predStr listI then true else (
93
94       if (ProjCommon.onlyConstInPred pred) then (false)
95       else (
96       let constPart=(ProjCommon.getAllConstInPred pred) in
97
98       let foundInBlist=(ProjCommon.isContainedInOneStrInTheList
      listI constPart) in
99
100      foundInBlist )
101      )
102
103      )
104    )
105
106 and getItemWithIndexI blacklist i =
107 match blacklist with
108 [] -> None |
109 (j, sl)::tail -> (if i=j then Some sl else (getItemWithIndexI
      tail i) )
110
111 and occursIn predStr listI =
112   match listI with
113   [] -> false |
114   h::t -> (if predStr = h then true
115      else occursIn predStr t);;
116
117 let rec addToBlackList blacklist i predStr =
118   match blacklist with
119   [] -> [(i,[predStr])] |
120   (n, sl)::(tail) ->
121   (if n=i then ((n, predStr::sl)::tail)
122   else ((n, sl)::(addToBlackList tail i predStr)));;
123
124
125 (*Get all the solutions for a singlePred*)
126 (*Get a results list. Will assume conn list for combining bools
      of predicate list is complete*)
127 let rec getAllSol4Pred indexedRules usedRules pred avlist
      blacklist =
128
129
130   match indexedRules with
131      [] -> ([Interpreter.eval_predicate (RuleList([])) pred
      avlist]) |
132
133      (i,curRule)::remainingRuleList ->
134        (
```

46

```
135        if (isInBlackList indexedRules blacklist i pred) then
136 ([ (false , []) ]) else
137
138           match curRule with
139      Fact fp -> (
140
141         match(Unify.unifyPredicates (fp,pred) ) with
142             None -> (getAllSol4Pred remainingRuleList (usedRules
       @ [(i ,curRule)]) pred avlist blacklist) |
143             Some sig0 ->
144
145
146
147      (true , sig0)::(getAllSol4Pred remainingRuleList (usedRules @
       [(i ,curRule)]) pred avlist blacklist) ) |
148
149      Rule (headPred, (body,connList))
150         -> (
151
152
153         match (Unify.unifyPredicates (headPred, pred)) with
154         None -> (getAllSol4Pred remainingRuleList (usedRules @ [(i
       ,curRule)]) pred avlist blacklist)
155           |
156         Some sig0 -> (
157
158
159
160 let newBlackList= addToBlackList blacklist i (ProjCommon.
       string_of_predicate pred) in
161
162
163                 let avoidList= Interpreter.getAVList sig0 @
       avlist in
164                 let renamedBody= Interpreter.
       renameFreeVarsInClause avoidList curRule in
165                 let newBody= List.map (Unify.substInPredicate
       sig0) renamedBody in
166                 let bodyQuery=(Query (newBody, connList)) in
167
168
169
170                 let resList4CurRule =
171            getResultListByApplyingSig (getAllSol (usedRules @
       indexedRules) bodyQuery true avoidList
172            newBlackList) sig0 in
173
174
175
176                 resList4CurRule @ (getAllSol4Pred
       remainingRuleList (usedRules @ [(i ,curRule)]) pred avlist
       newBlackList)
177
178
```

```
179
180 ) )
181        )
182
183
184 and getResultListByApplyingSig resList sigma =
185 match resList with
186    [] -> [] |
187
188    fstResult :: tail -> (
189
190
191            match fstResult with
192            (false, _) -> (getResultListByApplyingSig tail sigma) |
193            (true, fstSig) -> (match (Unify.composeSubst fstSig
     sigma) with
194                    None -> (true, []) :: (
     getResultListByApplyingSig tail sigma) |
195                    Some finalSig0 ->
196
197 let finalSig = (let updatedEQList= Unify.updateVarInSubst sigma
     finalSig0 in
198
199      match (Unify.unify updatedEQList) with
200        None -> [] |
201        Some fs -> fs ) in
202
203  (true, finalSig)::(getResultListByApplyingSig tail sigma)) )
204
205
206 and getAllSol indexedRules query lastBool avlist blacklist = (
207 if lastBool = false then ([])
208 else
209 match query with
210 Query(predList, connList) ->
211   ( match predList with
212     [] -> ((raise (Failure "Query is empty!"))) |
213     [singlePred] -> (
214
215   getAllSol4Pred indexedRules [] singlePred avlist blacklist) |
216
217      fstPred :: predTailList ->  (
218
219     let fstPredResultList = ( getAllSol4Pred indexedRules []
     fstPred avlist blacklist) in
220
221
222           (applyFirstResultToPredList fstPred fstPredResultList
     predTailList connList indexedRules lastBool avlist blacklist)
223 )
224
225 )
226 )
227
```

48

```
228
229  and evalPureQuery query =
230     (Interpreter.consult (RuleList([])) (query) true [])
231
232
233  and applyFirstResultToPredList fstPred fstPredResultList
        tailPredList connList indexedRules lastBool avlist blacklist
        =
234
235    match fstPredResultList with
236           [] -> (raise (Failure "Should return a result, either
        true or false")) |
237           (bool1, sig1)::remainingFstResultList ->
238        (let freeVarsInFstPred= ProjCommon.freeVarsInPredicate
        fstPred in
239         let refinedSig= filter freeVarsInFstPred sig1 in
240
241         let newTailPredList= List.map (substInPredicate refinedSig)
         tailPredList in
242
243
244
245         let newLastBool= (match connList with
246        ","::_ -> (bool1 && lastBool) |
247        ";"::_ -> (bool1 || lastBool) |
248        _ -> (raise (Failure "unknown connective."))) in
249
250
251
252      let allResults4FirstResult =
253     ProjCommon.addSigToResultList refinedSig (getAllSol
        indexedRules (Query(newTailPredList, List.tl connList))
        newLastBool avlist blacklist)
254
255      in
256
257
258    match remainingFstResultList with
259        [] -> allResults4FirstResult |
260
261        _ -> (allResults4FirstResult @
262          (applyFirstResultToPredList fstPred remainingFstResultList
         tailPredList connList indexedRules lastBool avlist blacklist
        ))
263     )
264
265
266
267  ;;
```

```ocaml
1  open ProjCommon
2  open Lexer
3  open Parser
4  open Unify
5  open Evaluator
6  open Interpreter
7  open Backtrack
8
9
10
11 (*Given a prolog program AST in ocaml, execute its semantics and
        return a result*)
12 let addAComma q = match q with
13                       Query (pl,connList) -> (Query (pl, (","::
    connList)) );;
14
15 let rec addCommToCL clauseList =
16   match clauseList with
17   [] -> [] |
18   clause::tail -> (match clause with
19        Rule(hp, (body, connList)) -> (Rule(hp, (body, ","::
    connList)))::(addCommToCL tail) |
20        _ -> clause :: (addCommToCL tail) ) ;;
21
22 let addACommaToRuleList rules =
23   match rules with
24   RuleList(cl) -> (
25     RuleList(addCommToCL cl)
26   ) ;;
27
28
29 let addCommaToPgm pgm =
30   match pgm with
31   Prog(rules, query)-> Prog(addACommaToRuleList rules, addAComma
        query) |
32   ProgFromQuery(query) -> ProgFromQuery(addAComma query);;
33
34
35 let getQueryFromPgm pgm =
36   match pgm with
37   Prog(_,query) -> (query) |
38   ProgFromQuery(query) -> (query);;
39
40 (*Given a prolog string, return a rule list*)
41 let parseRules s = addACommaToRuleList (Parser.rules Lexer.token
        (Lexing.from_string s));;
42
43 (*Given a prolog string, return a prolog program AST in ocaml*)
44 let parseProgram pgmStr = addCommaToPgm(Parser.program Lexer.
        token (Lexing.from_string pgmStr));;
```

50

```
45
46 let execProgram pgm = match (pgm) with
47                         Prog(rules, query) -> (Interpreter.
      consult rules (query) true []) |
48       ProgFromQuery(query) -> (Interpreter.consult (RuleList([])
      ) (query) true []) ;;
49
50 let debugProgram pgm =  match (pgm) with
51                         Prog(rules, query) -> (Interpreter.
      consult_debug rules (query) true [] true) |
52       ProgFromQuery(query) -> (Interpreter.consult_debug (
      RuleList([])) (query) true [] true) ;;
53
54
55 (* get pair of indexed rules and query from as string *)
56 let getIndexedRulesAndQueryFromStr pgmStr =
57   let parsedPgm= (parseProgram pgmStr) in
58   (match parsedPgm with
59     Prog(rules, query)-> (getIndexedRules rules, query) |
60     ProgFromQuery(query) -> ([], query) ) ;;
61
62
63 let getIndexedRulesAndQueryFromPgm pgm =
64
65   (match pgm with
66     Prog(rules, query)-> (getIndexedRules rules, query) |
67     ProgFromQuery(query) -> ([], query) ) ;;
68
69
70 (* print result *)
71
72 let printResult result = match result with
73                         (b,sigma) -> (
74           if sigma = [] then
75             (print_string ("\n"^ (string_of_bool b) ^ ".\n");)
76           else if (b) then(
77             print_string ((ProjCommon.string_of_subst sigma) ^ ".\
     n");)
78           else (print_string "false\n";)
79           ) ;;
80
81 let rec printResultList resultList =
82   match (rmFalseResult resultList) with
83   [] -> print_string "false\n" |
84   [singleResult] -> printResult singleResult |
85   curResult::tail -> (printResult curResult; print_string " ; \n
     "; printResultList tail)
86
87 and rmFalseResult resultList =
88 match resultList with
89 [] -> [] |
90 (b,sigma)::tail -> if b=false then rmFalseResult tail else (b,
     sigma)::(rmFalseResult tail);;
91
```

51

```ocaml
92
93 (* refine the result so that only the assignment to the free vars
        in the query get printed *)
94 let refineResult query result = let outputSig= (let freeVarsInQ=
        ProjCommon.freeVarsInQuery query in
95             Interpreter.filter freeVarsInQ (snd result) )
96         in (fst result, outputSig);;
97
98
99 (* Print the result of the result list one by one, simulating
        prolog's behavior. But not get the result on the fly *)
100 let rec printResultOneByOne resList =
101 let newList= rmFalseResult resList in
102 match newList with
103 [] -> (print_string "false\n";) |
104
105 _ -> (printResOneByOneHelper newList)
106
107 and printResOneByOneHelper resList=
108 match resList with
109 [] -> () |
110 [single] -> (printResult single) |
111 curResult::tail -> ((printResult curResult);
112
113 let decision= (flush stdout ; read_line ()) in
114 if decision = ";" || decision = " "
115 then
116 (print_string "\n";flush stdout; printResOneByOneHelper tail)
117 else ()
118
119 )
120 ;;
121
122 (* A user-friendly way of simulating prolog program: pretty
        print the result. *)
123 let simulateProgram pgmStr =
124                                         let pgm = parseProgram pgmStr
        in
125                                         let result= execProgram pgm
        in
126
127             let updatedResult= refineResult (getQueryFromPgm pgm)
        result in
128             printResult updatedResult;;
129
130
131 (* invoke the backtrack algorithm *)
132 let findAllResultsFromPgmStr pgmStr = let (indexedRules, query)
        = getIndexedRulesAndQueryFromStr pgmStr in
133         let resultList= Backtrack.getAllSol indexedRules query
        true [] [] in
134         let refinedResults = List.map (refineResult query)
        resultList in
135         printResultList refinedResults ;;
```

```
136
137 let findAllResults pgm = let (indexedRules, query) =
        getIndexedRulesAndQueryFromPgm pgm in
138          let resultList= Backtrack.getAllSol indexedRules query
          true [] [] in
139
140          List.map (refineResult query) resultList
141          ;;
```

Listing 9 : Play.ml: Code for interactive interpreter

```
 1 (*
 2    interactive-prolog program interpreter
 3 *)
 4 open ProjCommon
 5 open Lexer
 6 open Parser
 7 open Unify
 8 open Evaluator
 9 open Interpreter
10 open Backtrack
11 open Glue
12
13
14 (* Try to detect if something is getting piped in *)
15 let is_interactive = 0 = (Sys.command "[ -t 0 ]")
16
17 let _ =
18    (if is_interactive
19        then print_endline "\nWelcome to the prolog simulator \n"
20        else ());
21    let rec loop rules =
22    try
23    print_endline "input query or program with query please.";
24      let lexbuf = Lexing.from_channel stdin
25      in (if is_interactive
26            then (print_string "> "; flush stdout)
27            else ());
28
29    (try
30
31    let parsedPgm= Glue.addCommaToPgm(Parser.program Lexer.token
        lexbuf) in
32    let RuleList(existingRules)= rules in
33    let newPgm=(match parsedPgm with
34    Prog(RuleList(curRules), query) ->
35    (Prog(RuleList(curRules @ existingRules),query)) |
36
37    ProgFromQuery(query) -> (Prog(rules,query))   ) in
38
39    let result= Glue.refineResult (Glue.getQueryFromPgm newPgm) (
        Glue.execProgram newPgm) in
```

```
40    let _=  printResult  result  in
41    match newPgm with
42    Prog(newRules, _) -> (loop  newRules) |
43    ProgFromQuery(_) -> (loop  rules)
44
45
46        with  Failure  s -> (print_newline();
47          print_endline  s;
48                            print_newline();
49                            loop  rules)
50            | Parsing.Parse_error ->
51              (print_string "\ndoes  not  parse\n";
52               loop  rules))
53    with  Lexer.EndInput -> exit  0
54  in  loop  (RuleList [])
```

```
1  (*
2     interactive-prolog  simulator  debugger
3  *)
4  open  ProjCommon
5  open  Lexer
6  open  Parser
7  open  Unify
8  open  Evaluator
9  open  Interpreter
10 open  Backtrack
11 open  Glue
12
13
14 (* Try  to  detect  if  something  is  getting  piped  in *)
15 let  is_interactive = 0 = (Sys.command "[ -t  0 ]")
16
17 let  _ =
18   (if  is_interactive
19       then  print_endline "\nWelcome  to  the  prolog  debugger \n"
20       else  ());
21   let  rec  loop  rules =
22   try
23   print_endline "input  query  or  program  with  query  please.";
24     let  lexbuf = Lexing.from_channel  stdin
25     in (if  is_interactive
26           then (print_string "> "; flush  stdout)
27           else  ());
28
29   (try
30
31   let  parsedPgm= Glue.addCommaToPgm(Parser.program  Lexer.token
     lexbuf) in
32   let  RuleList(existingRules)= rules  in
```

54

```
33    let newPgm=(match parsedPgm with
34    Prog(RuleList(curRules), query) ->
35    (Prog(RuleList(curRules @ existingRules),query)) |
36
37    ProgFromQuery(query) -> (Prog(rules,query))   ) in
38
39    let result= Glue.refineResult (Glue.getQueryFromPgm newPgm) (
        Glue.debugProgram newPgm) in
40    let _=  printResult result in
41    match newPgm with
42    Prog(newRules, _) -> (loop newRules) |
43    ProgFromQuery(_) -> (loop rules)
44
45
46        with Failure s -> (print_newline();
47          print_endline s;
48                          print_newline();
49                          loop rules)
50            | Parsing.Parse_error ->
51              (print_string "\ndoes not parse\n";
52              loop rules))
53    with Lexer.EndInput -> exit 0
54  in loop (RuleList [])
```

Listing 11 :   ReadFile.ml: Code for the interpreter which directly reads input from file system

```
 1  open ProjCommon
 2  open Lexer
 3  open Parser
 4  open Unify
 5  open Evaluator
 6  open Interpreter
 7  open Backtrack
 8  open Glue
 9
10  (*Get the path of the prolog program*)
11  let file= Sys.argv.(1);;
12
13  let lexbuf = Lexing.from_channel (open_in file) ;;
14
15  let rec loop rules =
16
17  try (
18    let parsedPgm= Parser.program Lexer.token lexbuf in
19    let RuleList(existingRules)= rules in
20    let newPgm=(match parsedPgm with
21    Prog(RuleList(curRules), query) ->
22    (Prog(RuleList(curRules @ existingRules),query)) |
23
24    ProgFromQuery(query) -> (Prog(rules,query))   ) in
25
```

```
26    let resultList= List.map (Glue.refineResult (Glue.
       getQueryFromPgm newPgm)) (Glue.findAllResults newPgm) in
27    let _= Glue.printResultOneByOne resultList in
28    match newPgm with
29    Prog(newRules, _) -> (loop newRules) |
30    _ -> (loop rules)
31 )
32
33 with Lexer.EndInput -> (exit 0)
34    | _ -> (exit 1)
35
36 in loop (RuleList [])
```

Listing 12 :   *Play_all.ml*: **Code for interactive interpreter which shows every result it finds**

```
 1 (*
 2    interactive-prolog program interpreter
 3 *)
 4 open ProjCommon
 5 open Lexer
 6 open Parser
 7 open Unify
 8 open Evaluator
 9 open Interpreter
10 open Backtrack
11 open Glue
12
13
14 (* Try to detect if something is getting piped in *)
15 let is_interactive = 0 = (Sys.command "[ -t 0 ]")
16
17 let _ =
18    (if is_interactive
19        then print_endline "\nWelcome to the prolog simulator \n"
20        else ());
21    let rec loop rules =
22    try
23    print_endline "input query or program with query please.";
24      let lexbuf = Lexing.from_channel stdin
25      in (if is_interactive
26            then (print_string "> "; flush stdout)
27            else ());
28
29    (try
30
31    let parsedPgm= addCommaToPgm (Parser.program Lexer.token
       lexbuf) in
32    let RuleList(existingRules)= rules in
33    let newPgm=(match parsedPgm with
34    Prog(RuleList(curRules), query) ->
35    (Prog(RuleList(curRules @ existingRules),query)) |
```

56

```
36
37   ProgFromQuery(query) -> (Prog(rules,query))  ) in
38
39   let resultList= (Glue.findAllResults newPgm) in
40   let _=  Glue.printResultOneByOne resultList in
41   match newPgm with
42   Prog(newRules, _) -> (loop newRules) |
43   ProgFromQuery(_) -> (loop rules)
44
45
46       with Failure s -> (print_newline();
47          print_endline s;
48                         print_newline();
49                         loop rules)
50            | Parsing.Parse_error ->
51              (print_string "\ndoes not parse\n";
52               loop rules))
53   with Lexer.EndInput -> exit 0
54  in loop (RuleList [])
```

Listing 13 : make.sh: Code for building the project

```
1  ocamlc -c projCommon.ml
2
3  ocamlyacc parser.mly
4  ocamlc -c parser.mli
5  ocamlc -c parser.ml
6
7  ocamllex lexer.mll
8  ocamlc -c lexer.ml
9
10 ocamlc -c unify.ml
11 ocamlc -c evaluator.ml
12 ocamlc -c interpreter.ml
13 ocamlc -c backtrack.ml
14
15 ocamlc -c glue.ml
16
17
18
19 sh genExecutable.sh play
20 sh genExecutable.sh readFile
21 sh genExecutable.sh debugger
22 sh genExecutable.sh play_all
```

Listing 14 : clean.sh: Code for cleaning the project's binaries

```
1  rm *.mli
2  rm *.cmi
3  rm *.cmo
```

```
4  rm  lexer.ml
5  rm  parser.ml
6  rm  *.exe
7  rm  *~
```

```
1  ocamlc −o "$1".exe projCommon.cmo lexer.cmo parser.cmo unify.cmo
        evaluator.cmo interpreter.cmo backtrack.cmo glue.cmo "$1".ml
```