

## lib/basic/prolog.ath

```

1  #load "list-of"
2
3  module Prolog {
4
5  define print-info-flag := (cell true)
6
7  define (print-info) := (set! print-info-flag true)
8
9  define (dont-print-info) := (set! print-info-flag false)
10
11 define (mprint str) :=
12   check {(ref print-info-flag) => (print str)}
13   | else => ()}
14
15 # Embedding of SWI Prolog into Athena. First draft
16 # checked in on October 01, 2011, by K.A.
17
18 # Athena-SWI counterparts of some standard Prolog predicates:
19 # is, ==, \==, ==, and \=
20
21 declare is, ==, \==, arith-eq, arith-uneq: (T) [T T] -> Boolean
22
23 # Cut, fail, call, Prolog write, bagof, setof, and findall:
24
25 declare cut, fail: Boolean
26
27 declare write: [Ide] -> Boolean
28
29 declare call: [Boolean] -> Boolean
30
31 declare bagof, setof, findall: (T1,T2,T3) [T1 T2 T3] -> Boolean
32
33 define term-transformers := (cell [])
34
35 define (add-transformer T) := (set! term-transformers (add T (ref term-transformers)))
36
37 define (make-safe T) :=
38   lambda (t)
39     try { (T t) | t }
40
41 define (apply-transformer T) :=
42   let {T' := (make-safe T)}
43   lambda (sub)
44     match sub {
45       (some-sub sub) =>
46         (map lambda (v)
47           [v --> (T' (sub v))])
48         (supp sub))
49       | (some-list L) => (map lambda (triple)
50         match triple {
51           [(some-var v) --> (some-term t)] => [v --> (T' t)]
52           | _ => triple
53         }
54         L)
55     }
56
57 define (transform-sub x) :=
58   match x {
59     (some-sub sub) => letrec {loop := lambda (transformers res)
60       match transformers {
61         [] => res
62         | (list-of T more) => (loop more ((apply-transformer T) res))
63       }
64       }
65     (loop (ref term-transformers) sub)
66   | (some-list _) => (map transform-sub x)
67   | _ => x

```

```

68 }
69
70 declare goal1: (T1) [T1] -> Boolean
71 declare goal2: (T1, T2) [T1 T2] -> Boolean
72 declare goal3: (T1, T2, T3) [T1 T2 T3] -> Boolean
73 declare goal4: (T1, T2, T3, T4) [T1 T2 T3 T4] -> Boolean
74 declare goal5: (T1, T2, T3, T4, T5) [T1 T2 T3 T4 T5] -> Boolean
75 declare goal6: (T1, T2, T3, T4, T5, T6) [T1 T2 T3 T4 T5 T6] -> Boolean
76 declare goal7: (T1, T2, T3, T4, T5, T6, T7) [T1 T2 T3 T4 T5 T6 T7] -> Boolean
77 declare goal8: (T1, T2, T3, T4, T5, T6, T7, T8) [T1 T2 T3 T4 T5 T6 T7 T8] -> Boolean
78 declare goal9: (T1, T2, T3, T4, T5, T6, T7, T8, T9) [T1 T2 T3 T4 T5 T6 T7 T8 T9] -> Boolean
79 declare goal10: (T1, T2, T3, T4, T5, T6, T7, T8, T9, T10) [T1 T2 T3 T4 T5 T6 T7 T8 T9 T10] -> Boolean
80
81 define goal-vec := (make-vector 10 ())
82
83 (vector-set! goal-vec 0 goal1)
84 (vector-set! goal-vec 1 goal2)
85 (vector-set! goal-vec 2 goal3)
86 (vector-set! goal-vec 3 goal4)
87 (vector-set! goal-vec 4 goal5)
88 (vector-set! goal-vec 5 goal6)
89 (vector-set! goal-vec 6 goal7)
90 (vector-set! goal-vec 7 goal8)
91 (vector-set! goal-vec 8 goal9)
92 (vector-set! goal-vec 9 goal10)
93
94 (define (get-goal-predicate i)
95   (vector-sub goal-vec (minus i 1)))
96
97 # Some hash tables necessary for getting around Prolog's
98 # syntactic idiosyncracies:
99
100 define [var-table var-table' constant-table constant-table' fsym-table fsym-table'] :=
101   [(table 97) (table 97) (table 97) (table 97) (table 97) (table 97)];
102
103 define [var-counter constant-counter fsym-counter] :=
104   [(cell 0) (cell 0) (cell 0)];
105
106 define clear-memory :=
107   lambda ()
108     (seq (table-clear var-table)
109          (table-clear var-table')
110          (table-clear constant-table)
111          (table-clear constant-table')
112          (table-clear fsym-table)
113          (table-clear fsym-table')
114          (set! var-counter 0)
115          (set! constant-counter 0)
116          (set! fsym-counter 0));
117
118 define make-fresh-var :=
119   lambda ()
120     ("V" joined-with (val->string inc var-counter))
121
122 (define (make-fresh-constant)
123   (join "a" (val->string (inc constant-counter))))
124
125 (define (make-fresh-fun-sym)
126   (join "h" (val->string (inc fsym-counter))))
127
128 (define (prolog-legal? str)
129   (for-each str (lambda (c) (|| (alpha-numeric-char? c) (equal? c '_)))))
130
131 (define (make-prolog-constant c)
132   (check ((integer-numeral? c) (check ((greater-or-equal? c 0) (val->string c))
133                                         (else (join "-" (val->string (abs c))))))
134          ((equal? c cut) "!")
135          ((equal? c fail) "fail")
136          ((equal? c true) "true")
137          ((equal? c false) "fail"))

```

```

138         (else (let ((str (symbol->string c)))
139             (check ((prolog-legal? str) (add 'c str))
140                 (else (try (add 'k (table-lookup constant-table str))
141                     (let ((str' (make-fresh-constant))
142                         (_ (table-add constant-table [str --> str'])))
143                         (_ (table-add constant-table' [str' --> c]))))
144                     (add 'k str'))))))))
145
146 (define (numeric-op? f)
147     (member? f [(string->symbol "+")
148                 (string->symbol "-")
149                 (string->symbol "*")
150                 (string->symbol "/")
151                 (string->symbol "<")
152                 (string->symbol ">")]))
153
154 (define (make-prolog-functor f)
155     (check ((numeric-op? f) (symbol->string f))
156         ((equal? f (string->symbol "<=")) "<=")
157         ((equal? f (string->symbol ">=")) ">=")
158         ((equal? f =) (symbol->string f))
159         ((equal? f is) "is")
160         ((equal? f ==) "==")
161         ((equal? f \==) "\\==")
162         ((equal? f arith-eq) "=:=")
163         ((equal? f arith-uneq) "=\\=")
164         ((equal? f call) "call")
165         ((equal? f bagof) "bagof")
166         ((equal? f setof) "setof")
167         ((equal? f findall) "findall")
168         (else (let ((str (symbol->string f)))
169             (check ((prolog-legal? str) (add 'f str))
170                 (else (try (add 'g (table-lookup fsym-table str))
171                     (let ((str' (make-fresh-fun-sym))
172                         (_ (table-add fsym-table [str --> str'])))
173                         (_ (table-add fsym-table' [str' --> f]))))
174                     (add 'g str'))))))))
175
176
177 (define (make-prolog-term t)
178     (match t
179         ((some-var v) (let ((str (var->string v)))
180             (check ((prolog-legal? str) (add 'X str))
181                 (else (try (add 'Y (table-lookup var-table str))
182                     (let ((str' (make-fresh-var))
183                         (_ (table-add var-table [str --> str'])))
184                         (_ (table-add var-table' [str' --> v]))))
185                     (add 'Y str'))))))
186         ((write s) (join "write(' " (join "O" (id->string s) "\\n" " ' ")))
187         ((some-symbol f) (some-list args)
188             (match args
189                 ([] (make-prolog-constant f))
190                 (_ (join (make-prolog-functor f)
191                     "("
192                     (separate (map make-prolog-term args) ",")
193                     ")"))))
194
195 (define (make-prolog-prop p)
196     (match p
197         ((some-atom _) (make-prolog-term p))
198         ((not (some-sent q)) (join "not(" (make-prolog-prop q) ")"))
199         ((and (some-list args)) (separate (map make-prolog-prop args) ", "))
200         ((or (some-list args)) (join "(" (separate (map make-prolog-prop args) "; ") ")"))
201         ((if true consequent) (make-prolog-prop consequent))
202         ((if antecedent consequent) (join (make-prolog-prop consequent)
203             " :- "
204             (make-prolog-prop antecedent)))
205         ((forall (list-of _ _) (if (exists (list-of _ _) cond) (some-atom body)))
206             (make-prolog-prop (if cond body)))
207         ((forall (list-of _ _) (iff (some-atom body) (exists (list-of _ _) cond)))

```

```

208     (make-prolog-prop (if cond body)))
209     ((forall (list-of _ _) body) (make-prolog-prop body))
210     ((iff (as left (some-atom _)) right) (make-prolog-prop (if right left)))
211     ((iff left (as right (some-atom _)) (make-prolog-prop (if left right)))
212     (_ "")))
213
214 (define [bar comma lparen rparen lbrack rbrack blank colon scolon quot-mark]
215     [" | " ", " "(" " ")" "[" "]" " " ":" ";" "\""])
216
217 (define [c-comma c-lparen c-rparen c-blank c-newline] [' ' '(' ')' '\blank '\n])
218
219 (define (white-space? c)
220     (member? c [c-blank c-newline]))
221
222 (define (variable? x)
223     (match x
224     ((list-of (some-char c) _) (|| (upper-case-alpha-char? c) (equal? c '_)))
225     (_ false)))
226
227 (define (get-string str res)
228     (match str
229     ([[] [(rev res) []]])
230     ((list-of (some-char c) rest)
231     (check ((member? c [c-blank c-newline c-lparen c-rparen c-comma]) [(rev res) str])
232     (else (get-string rest (add c res)))))))
233
234 (define (get-symbol str)
235     (check ((member? str ["+" "-" "*" "/" "="]) (string->symbol str))
236     ((all-digits? str) (string->symbol str))
237     ((equal? str "[]") nil)
238     ((equal? str "'.') ::)
239     (else (match str
240     ((list-of 'c rest) (string->symbol rest))
241     ((list-of 'k rest) (table-lookup constant-table' rest))
242     ((list-of 'f rest) (string->symbol rest))
243     ((list-of 'g rest) (table-lookup fsym-table' rest))))))
244
245 (define (fresh-variable? str)
246     (equal? str "_"))
247
248 (define [parse-term parse-terms]
249     (letrec ((get-term (lambda (str)
250     (match (get-string str [])
251     ([root (list-of '( rest))]
252     (match (get-terms rest [])
253     ([args (list-of ' ) rest']
254     (let ((fsym (get-symbol root))
255     (res-term (make-term fsym args)))
256     [res-term rest'])))
257     ([root rest] (let ((mprint (join "\nroot: " (val->string root))))
258     (check ((fresh-variable? root) (let ((_) ()) [(fresh-var) rest]))
259     ((variable? root) (match root
260     ((list-of 'Y more) [(table-lookup var-table' more)
261     (check
262     ((null? (tail root))
263     [(string->var root) rest])
264     (else [(string->var (tail root))
265     rest])))))
266     (else (match root
267     ((list-of '- more) (let (([t more'] (get-term (join more rest))
268     [(make-term (string->symbol "-") [t]) more
269     (let ((make-term (get-symbol root) []) rest))))))))
270     (get-terms (lambda (str terms)
271     (match (get-term str)
272     ([term (list-of ' , rest)] (get-terms rest (add term terms)))
273     ([term rest] [(rev (add term terms)) rest])))))
274     [get-term (lambda (str) (get-terms str [])]))))
275
276
277 (define (get-line str)

```

```

276 (letrec ((loop (lambda (str chars)
277   (match str
278     ([[] [(rev chars) []])
279     ((list-of '\n rest) [(rev (add '\n chars)) rest])
280     ((list-of (some-char c) rest) (loop rest (add c chars)))))))
281   (loop str [])))
282
283
284
285
286 (define (process-output-line str v)
287   (let ((_ (mprint (join "\nProcessing this line: " str "\n"))))
288     (match str
289       ((list-of 'X more) (let (([var-name rest] (get-string more []))
290                               (term (first (parse-term (skip-until rest printable?))))
291                               (equality (= (string->var var-name) term))
292                               (equality (= v term)))
293                             [(lhs equality) (rhs equality)]))
294       ((list-of 'Y more) (let (([var-name rest] (get-string more []))
295                               (term (first (parse-term (skip-until rest printable?))))
296                               (equality (= (table-lookup var-table' var-name) term))
297                               (equality (= v term)))
298                             [(lhs equality) (rhs equality)]))
299       ## Output line:
300       ((list-of 'O more) (seq (print more) ()))
301       ((split "yes" _) true)
302       ((split "no" _) false))))
303
304
305 (define (process-output data vars)
306   (letrec ((loop (lambda (data results vars)
307     (match [data vars]
308       ([[] []] [true (rev results)])
309       ([[] _] [false (rev results)])
310       ([_ (list-of v more-vars)]
311        (match (get-line data)
312          ([line rest] (match (process-output-line line v)
313                             ([l r] (loop rest (add [l r] results) more-vars))
314                             (true (loop rest results more-vars))
315                             (false [false []])
316                             (() (loop rest results vars))))))
317          ([_ []]
318           (match (get-line data)
319             ([line rest] (match (process-output-line line ())
320                                ([l r] (loop rest (add [l r] results) []))
321                                (true (loop rest results []))
322                                (false [false []])
323                                (() (loop rest results []))))))
324             (match (loop data [] vars)
325               ([([some-term b] pairs) [b (make-sub pairs)])])))
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137

```

```

348 (vars (rev (vars* query-list)))
349 (var-strings (map make-prolog-term vars))
350 (make-write-var-string (lambda (var-string)
351   (join "write(' " var-string ' '),write(' '),write_canonical(" var-string "), nl)))
352 (write-var-strings (match (separate (map make-write-var-string var-strings) ",")
353   ([] [])
354   (str (join str ","))))
355 (write-var-strings' (match (separate (map make-write-var-string var-strings) ",")
356   ([] [])
357   (str (join "," str))))
358 ( _ (mprint (join "\nGoal string: " (val->string goal-string))))
359 (all-ground (for-each query-list ground?))
360 # (param-file "prologparams.txt")
361 # (param-string (join " -f " input-file " -g \"" goal-string write-var-strings' "\" -t \"halt\" -q > " output-fi
362 # ( _ (write-file param-file param-string))
363 # (command (join "swipl " param-file))
364 (command (join "swipl -f " input-file " -g \"" goal-string write-var-strings' "\" -t \"halt\" -q > " output-fi
365 ( _ (mprint (join "\nTIME UP TILL COMMAND EXECUTION: " (val->string (minus (time) start-time))))
366 ( _ (mprint (join "\nCommand: " command)))
367 ( _ (exec-command command))
368 (output-data (read-file output-file))
369 ( _ (mprint (join "\nOutput: " (val->string output-data)))))
370 (match output-data
371   ([ (check ((null? query-list) [true empty-sub])
372     (else [false empty-sub]))])
373   (str (process-output str vars)))
374
375 (define (solve-with-time-limit query-list program time-limit)
376   (match query-list
377     ((some-list _) (solve-with-time-limit-aux program query-list time-limit))
378     ( _ (solve-with-time-limit-aux program [query-list] time-limit)))
379
380 (define [LB RB COMMA BACKSLASH] ['[ ' ']' ',' '\'])
381
382 (define (accum-until stream pred)
383   (letrec ((loop (lambda (stream res)
384     (match stream
385       ([ (rev res) []])
386       ((list-of (some-char c) more)
387        (check ((pred c) [(rev res) stream])
388         (else (loop more (add c res))))))))))
389   (loop stream []))
390
391 (define (get-list-content stream)
392   (accum-until stream (lambda (c) (equal? c RB))))
393
394 (define (get-one-list stream)
395   (match (skip-until stream printable?)
396     ((list-of (val-of LB) more) (let (([str rest] (get-list-content more)))
397       [(first (parse-terms str)) (tail rest)])))
398
399 (define (process-find-all-output str vars)
400   (letrec ((loop (lambda (stream results)
401     (match (skip-until stream printable?)
402       ([ (rev results) []])
403       ([ (val-of RB) (rev results) ]
404        ((list-of (val-of COMMA) more)
405         (loop more results))
406        ( _ (match (get-one-list stream)
407              ([ (some-list terms) rest']
408               (let ((sub-content (map (lambda (var-term-pair)
409                 (match var-term-pair
410                   ([v t] (let ((equality (= v t)))
411                     [(lhs equality) (rhs equality)]))))
412                 (zip vars terms))))
413               (loop rest' (add sub-content results))))))))))
414   (loop (tail str) []))
415
416 (define (process-find-all-output' str vars)
417   (match (process-find-all-output str vars)

```

```

418 ((some-list sub-lists) (map make-sub sub-lists)))
419
420 (define numeric-term-portray-def
421   (join "\nis_numeric(X) :- functor(X,+,_), !."
422         "\nis_numeric(X) :- functor(X,-,_), !."
423         "\nis_numeric(X) :- functor(X,*,_), !."
424         "\nis_numeric(X) :- functor(X/,_), !."
425         "\nis_numeric(X) :- functor(X,<,_), !."
426         "\nis_numeric(X) :- functor(X,<=,_), !."
427         "\nis_numeric(X) :- functor(X,>,_), !."
428         "\nis_numeric(X) :- functor(X,>=,_), !."
429         "\nportray(X) :- is_numeric(X), write_canonical(X).\n"))
430
431 private define solve-all-with-time-limit-aux :=
432 lambda (program query-list time-limit)
433   (let (([input-file output-file error-file] ["a.pl" "o.pl" "e.pl"]))
434     (_ (delete-files [input-file output-file error-file]))
435     (_ (clear-memory))
436     (prolog-program (separate (map (lambda (p) (join (make-prolog-prop p) ".\n")) program) ""))
437     (_ (mprint (join "\nGiven program:\n" (val->string prolog-program))))
438     (_ (write-file input-file "\nuse_module(library(time)).\n"))
439     (_ (write-file input-file numeric-term-portray-def))
440     (_ (write-file input-file prolog-program))
441     (vars (rev (vars* query-list)))
442     (var-strings (map make-prolog-term vars))
443     (var-string (join "[" (separate var-strings ",") "]" ))
444     (answer-var (add 'X (var->string (fresh-var))))
445     (goal-string (join "("
446                       (separate (map make-prolog-term query-list)
447                                ",")
448                       ")"))
449     (total-goal (check ((|| true (greater? time-limit 0))
450                        (join "call_with_time_limit(" (val->string time-limit)
451                          ",findall(" var-string "," goal-string "," answer-var ")), write_term(" answer-var
452                          (else (join "findall(" var-string "," goal-string "," answer-var ")), write_term(" answer-var
453                        (_ (mprint (join "\nGoal string: " (val->string total-goal))))
454                        (command (join "swipl -f " input-file " -g \" total-goal \" -t \"halt\" -q > " output-file " 2> " error-file
455                        (_ (mprint (join "\nCommand: " command)))
456                        (_ (exec-command command))
457                        (output-data (read-file output-file))
458                        (_ (mprint (join "\nOutput: " output-data)))
459                        (_ ()))
460     (match output-data
461       ([[] []])
462       (str (process-find-all-output' str vars))))
463
464 private define solve-N-with-time-limit-aux :=
465 lambda (prog goal N time-limit)
466   (letrec ((loop (lambda (i subs)
467                     (check ((less? N i) (rev subs))
468                           (else (let ((sub-negations (lambda (sub)
469                                                            (map (lambda (v) (not (= v (sub v)))) (supp sub))))
470                                (all-negations (flatten (map sub-negations subs)))
471                                (goal' (join goal all-negations)))
472                                (match (solve-with-time-limit-aux prog goal' time-limit)
473                                  ([false _] (rev subs))
474                                  ([true (some-sub sub)] (loop (plus i 1) (add sub subs))))))))))
475     (check ((less? N 1) [])
476            (else (loop 1 [])))))
477
478 (define (solve-N-with-time-limit goal prog N time-limit)
479   (match goal
480     ((some-list _) (solve-N-with-time-limit-aux prog goal N time-limit))
481     (_ (solve-N-with-time-limit-aux prog [goal] N time-limit)))
482
483 (define MAX-TIME-LIMIT 10000000)
484
485 define solve-aux :=
486 lambda (prog g)
487   (match g

```

```

488 ((some-list _) (solve-with-time-limit-aux prog g MAX-TIME-LIMIT))
489 ((some-sentence _) (solve-with-time-limit-aux prog [g] MAX-TIME-LIMIT))
490 (_ (error "Prolog.solve error: the goal must be either a sentence or a list of sentences.")))
491
492 (define (solve g prog)
493   (transform-sub (solve-aux prog g)))
494
495 (define (solve-all-with-time-limit query-list program time-limit)
496   (match query-list
497     ((some-list _) (solve-all-with-time-limit-aux program query-list time-limit))
498     (_ (solve-all-with-time-limit-aux program [query-list] time-limit))))
499
500 define solve-all-aux :=
501 lambda (prog g)
502   (match g
503     ((some-list _) (solve-all-with-time-limit-aux prog g MAX-TIME-LIMIT))
504     ((some-sentence _) (solve-all-with-time-limit-aux prog [g] MAX-TIME-LIMIT))
505     (_ (error "Prolog.solve-all error: the goal must be either a sentence or a list of sentences.")))
506
507 (define (solve-all g prog)
508   (transform-sub (solve-all-aux prog g)))
509
510 define solve-N-aux :=
511 lambda (prog goal N)
512   (match goal
513     ((some-list _) (solve-N-with-time-limit-aux prog goal N MAX-TIME-LIMIT))
514     ((some-sentence _) (solve-N-with-time-limit-aux prog [goal] N MAX-TIME-LIMIT))
515     (_ (error "Prolog.solve-N error: the goal must be either a sentence or a list of sentences.")))
516
517 (define (solve-N goal prog N)
518   (solve-N-aux prog goal N))
519
520 (define (make-new-clause g-pred goal-vars goals subs)
521   (let ((negate-sub (lambda (sub)
522                       (or (map (lambda (v)
523                                 (not (= v (sub v))))
524                             goal-vars))))
525         (head (make-term g-pred goal-vars))
526         (body (and (join goals (map negate-sub subs))))))
527     (if body head))
528
529 (define (solve-N-aux goals prog N lim)
530   (let ((goals (match goals ((some-list _) goals) (_ [goals])))
531         (goal-vars (vars* goals))
532         (var-num (length goal-vars))
533         (g-pred (get-goal-predicate var-num))
534         )
535     (letrec ((loop (lambda (i subs)
536                      (check ((less? N i) (rev subs))
537                             (else (let ((new-clause (make-new-clause g-pred goal-vars goals subs))
538                                         (new-goal (make-term g-pred goal-vars))
539                                         (new-prog (join prog [new-clause]))
540                                         (res (check ((less? lim 0) (solve-aux new-prog new-goal))
541                                                         (else (solve-with-time-limit-aux new-prog new-goal))))))
542                                (match res
543                                  ([false _] (rev subs))
544                                  ([_ (some-sub sub)] (loop (plus i 1) (add sub subs))))))))
545         (loop 1 []))))
546
547 (define (solve-N goals prog N)
548   (transform-sub (solve-N-aux goals prog N (- 1))))
549
550 (define (solve-N-with-time-limit goals prog N lim)
551   (transform-sub (solve-N-aux goals prog N lim)))
552
553 } # module Prolog
554
555 module Horn {
556
557 (define pred-table (table 100))

```



```

558
559 (define (make-fresh-pred-name f)
560   (let ((symbol? (lambda (str)
561     (try (seq (string->symbol str) true) false)))
562     (index (cell 0))))
563   (letrec ((loop (lambda (prefix name)
564     (check ((symbol? name) (loop prefix (join prefix (val->string (inc index)))))
565       (else name)))))
566     (let ((first-try (map downcase (join (map (lambda (c) (check ((equal? c \.) \_) (else c))) (val->string f)) "\_P
567       (loop first-try first-try)))))
568
569 (define (boolean-symbol? f)
570   (equal? (last (get-signature f)) "Boolean"))
571
572 (define (get-pred-version f)
573   (let ((f (get-symbol f)))
574     (check ((boolean-symbol? f) f)
575       (else (try (table-lookup pred-table f)
576         (let ((f-pred-name (make-fresh-pred-name f))
577           (sig (get-signature f))
578           (sort-string (separate sig " "))
579           (toks (tokenize-string sort-string ['\']))
580           (toks (filter toks (lambda (t) (equal? (first t) 'T))))
581           (toks (dedup (map (lambda (t) (first (tokenize-string t ['\blank '( ')]))) toks)))
582           (sort-var-string (check ((null? toks) "") (else (join "(" (separate toks ", " " "))))))
583           (sort-string' (filter-out sort-string (lambda (c) (equal? c \'))))
584           (cmd (join "declare " f-pred-name ": " sort-var-string " [" sort-string' "]" -> Boolean))
585           (_ (process-input-from-string cmd true))
586           (pf (string->symbol f-pred-name))
587           (_ (table-add pred-table [f --> pf])))
588         pf))))))
589
590
591 (define (term->horn-clause t)
592   (match t
593     ([[] (some-var _) ((some-symbol _) [])] [[] t])
594     (((some-symbol f) (some-list args))
595       (check ((constructor? f)
596         (let (([arg-clauses arg-vars] (unzip (map term->horn-clause args)))
597           (arg-clauses (flatten (map join arg-clauses)))
598           [arg-clauses (make-term f arg-vars)]))
599         ((boolean-symbol? f)
600           (let (([arg-clauses arg-vars] (unzip (map term->horn-clause args)))
601             (arg-clauses (flatten (map join arg-clauses)))
602             [(join arg-clauses [(make-term f arg-vars)] [])])
603           (else (match args
604             ([[] [[] t]]
605              _ (let (([arg-clauses arg-vars] (unzip (map term->horn-clause args)))
606                (arg-clauses (flatten (map join arg-clauses)))
607                (out-var (fresh-var))
608                (last-clause (make-term (get-pred-version f) (join arg-vars [out-var])))
609                [(join arg-clauses [last-clause]) out-var])))
610
611 (define thc term->horn-clause)
612
613 (define (literal->hc t)
614   (match t
615     ((not (some-term t))
616       (match (term->horn-clause t)
617         ([[(clauses as (list-of _ _)) ()] [(join (all-but-last clauses) [(not (last clauses))]) ()]
618           [(clauses (some-term bool-term)] [(join clauses [(not bool-term)] [])])])
619         ((some-term _) (term->horn-clause t))))
620
621 (define (get-all-clauses bool-terms)
622   (let (([clauses _] (unzip (map literal->hc bool-terms)))
623     (flatten (map join clauses)))
624
625 (define (smart-and L)
626   (match L
627     ([[(some-sent p)] p])

```

```

628     ([ true)
629     (_ (and L))))
630
631 (define (eqn->horn-clause-aux eqn)
632   (match eqn
633     ((forall (some-list _) (= (l as ((some-symbol f) (some-list args)))
634                               (some-term r)))
635      (check ((boolean-symbol? f)
636              (match (term->horn-clause r)
637                ([clauses ()] (if (smart-and clauses) 1))
638                ([clauses bool-term] (if (smart-and (join clauses [bool-term])) 1))))
639      (else (let ((fp (get-pred-version f)))
640              (match (term->horn-clause r)
641                ([clauses out] (if (smart-and clauses) (make-term fp (join args [out])))))))))
642     ((forall (some-list _)
643      ([| (if (ant as ([| (some-term guard) (guard as (not (some-term _)))) (body as (= (l as ((some-symbol f) (some-term
644      (if (ant as (and (some-list guards))) (body as (= (l as ((some-symbol f) (some-list args))) (some-term
645      (let ((guards (try [guard] guards))
646            (conjuncts (get-conjuncts-recursive ant))
647            (clauses (get-all-clauses conjuncts)))
648      (match (eqn->horn-clause-aux body)
649        ((if (some-sent ant) (some-sent con))
650        (let ((clauses' (join clauses (get-conjuncts-recursive ant)))
651              (clauses' (filter-out clauses' (lambda (c) (equal? c true)))))
652        (if (smart-and clauses') con)))))))))
653
654 (define (eqn->horn-clauses eqn)
655   (match eqn
656     ((forall (list-of _ _) (if (exists (list-of _ _) (some-sent ant)) (some-atom body)))
657      (eqn->horn-clauses (if ant body)))
658     ((forall (list-of _ _) (iff (some-atom body) (exists (list-of _ _) cond)))
659      (eqn->horn-clauses (if cond body)))
660     ((forall (some-list _) (if (ant as (or (some-list guards))) con))
661      (let ((D (get-disjuncts-recursive ant))
662            (eqns (map (lambda (d) (if d con)) D))
663            (clauses (map eqn->horn-clause-aux eqns)))
664        clauses))
665     (_ (try [(eqn->horn-clause-aux eqn)] []))))
666
667 (define (post-process clause)
668   (match clause
669     ((if true (some-sent body)) body)
670     ((if (ant as ([| (some-term cond)
671                     (and (some-list conds))]) body)
672      (let ((conds (try [cond] conds))
673            (body-vars (vars body)))
674        (letrec ((loop (lambda (clauses idents non-idents)
675                          (match clauses
676                            ([| [(rev idents) (rev non-idents)]]
677                             ((list-of (c as (= (some-var x) _)) (some-list rest))
678                              (check ((| (member? x (vars* rest)) (negate (member? x body-vars)))
679                                     (loop rest idents (add c non-idents)))
680                              (else (loop rest (add c idents) non-idents))))
681                             ((list-of (some-sent c) (some-list rest))
682                              (loop rest idents (add c non-idents)))))))
683          (let (([identities non-idents] (loop (get-conjuncts-recursive ant) [] []))
684                (bindings (map (lambda (i) (match i ((= (some-term l) (some-term r)) [l r])) identities))
685                (sub (make-sub bindings)))
686            (match non-idents
687              ([| (sub body))
688              (_ (if (smart-and non-idents) (sub body))))))
689      (_ clause)))
690
691 (define pp post-process)
692
693 (define (ehc eqn) (map post-process (eqn->horn-clauses eqn)))
694
695 ### make-horn-clauses is the official procedure for taking a (possibly conditional) equation
696 ### and turning it into a list of horn clauses:
697

```

```

698 (define (make-horn-clauses L)
699   (match L
700     ((some-sent p) (ehc p))
701     ((some-list _) (flatten (map ehc L)))))
702
703 (define mhc make-horn-clauses)
704
705 (define (test-sym f)
706   (let ((eqns (map quant-body (defining-axioms f))))
707     (flatten (map ehc eqns))))
708
709 (define (get-syms x)
710   (match x
711     ((some-term _) (get-term-syms x))
712     ((some-sent _) (get-prop-syms x))
713     ((some-list _) (flatten (map get-syms x)))))
714
715 (define (occurring-syms s)
716   (try
717     (let ((m (fsd s)))
718       (match (m 'occurring-syms)
719         ((some-list sym-names) (map string->symbol sym-names))
720         (_ [])))
721     []))
722
723 (define (guard-syms s)
724   (try
725     (let ((m (fsd s)))
726       (match (m 'guard-syms)
727         ((some-list sym-names) (map string->symbol sym-names))
728         (_ [])))
729     []))
730
731 (define (get-all-syms goal)
732   (let ((syms0 (dedup (get-syms goal)))
733         (syms0 (filter-out syms0 (lambda (s) (null? (defining-axioms s)))))
734         (T (table 100))
735         (_ (map-proc (lambda (s) (table-add T [s --> true])) syms0))
736         (reachable-syms (lambda (s)
737           (let (([osyms gsyms] [(occurring-syms s) (guard-syms s)]))
738             (join osyms gsyms))))
739         (reachable-syms* (lambda (syms)
740           (filter-out (dedup (flatten (map reachable-syms syms)))
741             (lambda (s) (null? (defining-axioms s))))))
742         (existing? (lambda (s) (try (table-lookup T s) false))))
743     (letrec ((loop (lambda (syms)
744       (let ((syms' (filter-out (reachable-syms* syms) existing?))
745             (_ (map-proc (lambda (s) (table-add T [s --> true])) syms'))
746             (check (null? syms') ()
747               (else (loop syms'))))))
748         (let ((_ (loop syms0)))
749           (dedup (map first (table->list T))))))
750
751
752 (define (get-all-syms-sorted goal)
753   (let ((L (get-all-syms goal)))
754     (prim-sort L (lambda (s1 s2)
755       (try (let ((m (fsd s1)))
756         (match (m 'needed-by-syms)
757           ((some-list sym-names) (member? s2 (map string->symbol sym-names)))
758           (_ false)))
759         false))))))
760
761 (define (sorted-defining-axioms s)
762   (prim-sort (defining-axioms s)
763     (lambda (p1 p2) (less? (size p1) (size p2)))))
764
765 (define (get-all-horn-clauses goal)
766   (let ((syms (get-all-syms-sorted goal))
767         (all-clauses (flatten (map (lambda (s)

```

```

768                                     (make-horn-clauses (sorted-defining-axioms s)))
769                                     syms))))
770     (dedup all-clauses)))
771
772 (define (test goal)
773   (get-all-horn-clauses goal))
774
775 (define (make-goal p)
776   (make-horn-clauses p))
777
778 (define (solve goals)
779   (let ((program-clauses (get-all-horn-clauses goals))
780         (goals (match goals ((some-list _) goals) (_ [goals])))
781         (goal-clauses (get-all-clauses (flatten (map get-conjuncts-recursive goals)))))
782     (match (Prolog.solve-aux program-clauses goal-clauses)
783       ([true (some-sub sub)] (let ((variables (vars* goals))
784                                   (bindings (list-zip variables (map sub variables))))
785                               [true (Prolog.transform-sub (make-sub bindings))]))
786       (res res))))
787
788 (define (solve-all goals)
789   (let ((program-clauses (get-all-horn-clauses goals))
790         (goals (match goals ((some-list _) goals) (_ [goals])))
791         (goal-clauses (get-all-clauses (flatten (map get-conjuncts-recursive goals)))))
792     (L (Prolog.solve-all-aux program-clauses goal-clauses))
793     (map (lambda (sub)
794           (let ((variables (vars* goals))
795                 (bindings (list-zip variables (map sub variables))))
796             (Prolog.transform-sub (make-sub bindings))))
797         L)))
798
799 (define (solve-N' goals N)
800   (let ((program-clauses (get-all-horn-clauses goals))
801         (goals (match goals ((some-list _) goals) (_ [goals])))
802         (goal-clauses (get-all-clauses (flatten (map get-conjuncts-recursive goals)))))
803     (L (Prolog.solve-N-aux goal-clauses program-clauses N (- 1)))
804     (map (lambda (sub)
805           (let ((variables (vars* goals))
806                 (bindings (list-zip variables (map sub variables))))
807             (Prolog.transform-sub (make-sub bindings))))
808         L)))
809
810 (Prolog.dont-print-info)
811
812 } # module Horn
813
814 extend-module Prolog {
815   (Prolog.dont-print-info)
816
817   (define (solve-goal g)
818     (let ((_ (Prolog.dont-print-info))
819           (_ ()))
820       (Horn.solve g)))
821
822   (define auto-solve solve-goal)
823
824   (define (solve-goal-all g)
825     (let ((_ (Prolog.dont-print-info)))
826       (Horn.solve-all g)))
827
828   (define (defining-clauses g)
829     (Horn.get-all-horn-clauses g))
830
831   (define (query-clauses g)
832     (let ((g (match g
833               ((some-list _) g)
834               (_ [g]))))
835       (Horn.get-all-clauses (flatten (map get-conjuncts-recursive g)))))
836
837

```

```
838
839 (define auto-solve-all solve-goal-all)
840
841 (define (solve-goal-N g N)
842   (let ((_ (Prolog.dont-print-info)))
843     (Horn.solve-N' g N)))
844
845 (define auto-solve-N solve-goal-N)
846
847 }
848
849 set-precedence Prolog.solve-goal 50
850 set-precedence Prolog.auto-solve 50
851
852 EOF
853 (load "lib/basic/prolog.ath")
```