```
1 # There are 4 main procedures here: (make-random-term S depth), (make-random-terms S N),
  \# (model-check p), and (model-check-bounded p k). (make-random-term S depth) makes a random
3 # term of sort S and depth d, where "depth" here indicates the maximum-length chain of
4 # reflexive constructor applications. The input sort S must be either a datatype sort or
5 # else Int or Ide. (make-random-terms S N) makes (roughly) N *distinct* terms of sort S, randomly
  # generated using make-random-term. The algorithm will try to distribute evenly the
   # depths of the generated terms as much as possible. (That is why the number of output terms
8 # is usually roughly N, not exactly N.) A procedure call (model-check p) will try to
  # evaluate the truth of p in the standard model (assuming that p contains only function
10 # symbols defined on initial algebras, i.e., on datatypes, and standard domains such as Int;
   # the procedure obviously won't work for loose semantics). If there are any existential
12 # quantifiers in p that are verified, then the values of the corresponding values are
13 # also provided as part of the output. Likewise, if there are any universally quantified
14 # sentences inside p that are falsified, the values of the corresponding bound variables
is # are produced as part of the output. This assumes that all bound variables in p have
   # distinct names, which is easy to ensure (just use rename before passing p to model-check).
  (load-file (file-path [ATHENA_LIB "dt-streams.ath"]))
18
19
  (load-file (file-path [ATHENA_LIB "maps.ath"]))
20
21
   (define (random-range-element low high)
22
      (let ((i (random-int (plus 1 (minus high low)))))
23
24
        (plus (minus i 1) low)))
25
   (define (choose-random-integer)
     (random-range-element 0 100))
27
29 (define (choose-random-real)
    (let ((i1 (choose-random-integer))
30
           (i2 (choose-random-integer))
31
           (str (join (val->string i1) "." (val->string i2))))
32
       (string->num str)))
34
   (define (choose-random-identifier)
35
     (string->id (join "x" (val->string (random-int 10000)))))
36
   (define
     (infinite-depth-sort? S)
39
        (|| (negate (null? (reflexive-constructors-of S)))
            (for-some (irreflexive-constructors-of S) (lambda (c) (infinite-depth-constructor? c S))))
41
     (infinite-depth-constructor? irc S)
42
       (for-some (arg-sorts-unified irc S) infinite-depth-sort?)
43
     (infinite-value-but-finite-depth-constructor? c S)
44
       (&& (negate (infinite-depth-constructor? c S))
45
           (\textit{for-some (arg-sorts-unified c S)} \quad \textbf{(lambda (S) (infinite-value-but-finite-depth-sort? S))))}\\
     (infinite-value-but-finite-depth-sort? S)
47
48
       (&& (negate (infinite-depth-sort? S))
           (|| (member? S ["Int" "Ide"])
49
               (for-some (constructors-of S)
                          (lambda (c) (infinite-value-but-finite-depth-constructor? c S)))))
51
     (infinite-sort? S)
52
       (|| (infinite-depth-sort? S) (infinite-value-but-finite-depth-sort? S)))
53
54
56 (define infinite-depth-sort? (memoize-unary infinite-depth-sort?))
  (define infinite-depth-constructor? (memoize-binary infinite-depth-constructor?))
  (define infinite-value-but-finite-depth-constructor? (memoize-binary infinite-value-but-finite-depth-constructor?))
   (define infinite-value-but-finite-depth-sort? (memoize-unary infinite-value-but-finite-depth-sort?))
   (define infinite-sort? (memoize-unary infinite-sort?))
61
   (define (infinite-at-each-level? S)
     (let ((mem (cell [])))
63
64
       (letrec ((loop (lambda (S)
                         (check ((member? S (ref mem)) false)
65
                                 (else (let ((_ (set! mem (add S (ref mem)))))
66
                                         (&& (infinite-depth-sort? S)
                                             (for-each (constructors-of S)
68
```

2

```
(lambda (c)
                                                             (for-some (arg-sorts-unified c S) (lambda (S') (|| (infinite-val
70
                                                                                                                   (loop S'))))
         (loop S))))
72
73
   (define infinite-at-each-level? (memoize-unary infinite-at-each-level?))
74
75
   (define (make-random-term sort d)
    (let (([irc's rc's] (filter-and-complement (constructors-of sort) (lambda (c) (irreflexive-unif? c sort))))
77
           ([idirc's nidirc's] (filter-and-complement irc's (lambda (c) (infinite-depth-constructor? c sort))))
78
79
           (make-bottom-term (lambda ()
                                (let ((irc (try (choose nidirc's) (choose idirc's)))
80
                                      (arg-terms (map (lambda (S) (make-random-term S 0)) (arg-sorts-unified irc sort))))
81
                                  (make-term irc arg-terms)))))
82
83
      (check ((equal? sort "Int") (choose-random-integer))
              ((equal? sort "Ide") (choose-random-identifier))
84
              ((equal? sort "Real") (choose-random-real))
85
              ((less? d 1) (make-bottom-term))
              (else (match (join rc's idirc's)
87
                      ([] (make-random-term sort 0))
88
                      (cs (let ((c (choose cs))
89
                                 (choose-height (lambda () (random-range-element 0 (minus d 1))))
90
                                 (c-arg-sorts (arg-sorts-unified c sort))
91
                                 (max-height-child (choose (filter (from-to 1 (arity-of c))
92
93
                                                                     (lambda (i) (infinite-depth-sort? (nth i c-arg-sorts))))
                                 ([L1 (list-of x L2)] (split-list c-arg-sorts (minus max-height-child 1)))
94
                                 (arg-terms1 (map (lambda (S) (make-random-term S (choose-height))) L1))
95
                                 (max-term (make-random-term x (minus d 1)))
                                 (arg-terms2 (map (lambda (S) (make-random-term S (choose-height))) L2))
97
                                 (arg-terms (join arg-terms1 (add max-term arg-terms2))))
98
                             (make-term c arg-terms))))))))
99
100
   (define (has-at-least-binary-ref-con S)
101
     (let ((mem (cell [])))
102
        (letrec ((loop (lambda (S)
103
                          (check ((member? S (ref mem)) false)
104
                                 (else (let ((_ (set! mem (add S (ref mem)))))
105
                                          (for-some (constructors-of S)
106
                                                    (lambda (c)
107
                                                      (let ((c-arg-sorts (arg-sorts-unified c S)))
108
                                                        (|| (greater? (length (filter c-arg-sorts (lambda (S') (equal? S' S)
109
                                                             (for-some c-arg-sorts loop))))))))))
111
         (loop S))))
112
113
   (define has-at-least-binary-ref-con (memoize-unary has-at-least-binary-ref-con))
114
115
   (define (has-at-least-one-binary-con? S)
116
117
     (for-some (constructors-of S) (lambda (c) (greater? (arity-of c) 1))))
118
   (define (decide N depth sort)
119
     (let ((d (check ((for-some (irreflexive-constructors-of sort)
120
                                   (lambda (c) (infinite-value-but-finite-depth-constructor? c sort)))
121
122
                           (plus depth 1))
                       (else depth)))
123
            (x (div N d)))
124
        (check ((&& (leq? x 1) (less? depth N))
125
                    (div N depth))
126
               (else x))))
127
128
129
   (define decide (memoize-ternary decide))
130
   (define (make-random-terms sort N)
131
     (let ((ht' (make-term-hash-table 983))
132
            (is-infinite-at-each-level (infinite-at-each-level? sort))
133
134
            (make (lambda (d)
                    (letrec ((loop (lambda ()
135
                                      (let ((t (make-random-term sort d)))
136
                                        (match (term-look-up ht' t)
137
                                         (() (term-enter ht' t true))
138
```

```
(_ (check ((&& (greater? d 0) is-infinite-at-each-level) (loop))
                                                     (else ()))))))))
140
                       (loop)))))
141
        (check ((|| (negate (infinite-depth-sort? sort))
142
                     (negate (infinite-sort? sort)))
143
                    (stream-take (make-all-ground-terms sort) N))
144
               (else (let ((has-at-least-one-binary-ref-con? (has-at-least-binary-ref-con sort))
145
                            (depth (check ((negate (infinite-at-each-level? sort))
                                               (check ((has-at-least-one-binary-con? sort) 10)
147
                                                      (else N)))
148
149
                                            (has-at-least-one-binary-ref-con? 5)
                                            (else 8)))
150
                            ([depth' count] (check ((leq? N depth) [N 1])
151
                                                     (else [depth (decide N depth sort)])))
152
                            (range (from-to 1 count))
153
154
                            (_ (map (lambda (d)
                                              (map (lambda (_) (make d)) range))
155
                                    (from-to 0 depth')))
                            (results (map first (show-table ht')))
157
                            (sorted-results (merge-sort results (lambda (t1 t2) (less? (height t1) (height t2)))))
158
                             (_ ()))
159
160
                         sorted-results)))))
161
   (define (dup-count L)
162
163
      (letrec ((loop (lambda (L res)
                         (match L
164
                           ([] res)
165
                           ((list-of x more) (check ((member? x more) (loop more (plus res 1)))
166
                                                      (else (loop more res)))))))))
167
        (loop L 0)))
168
169
170
   (define (test sort N)
171
     (let ((t1 (time))
            (L (make-random-terms sort N))
172
173
            (t2 (time))
            (dc (dup-count L)))
174
       (print "\nLength: " (length L) "\ndup-count: " dc "\nTotal time: " (minus t2 t1) "\n")))
175
176
   (define
177
     (apply-env-to-term env)
178
        (lambda (t)
179
          (match t
180
181
            ((some-var v) (apply-map env v))
            (((some-symbol f) (some-list args)) (make-term f (map (apply-env-to-term env) args)))))
182
183
     (apply-env-to-sent env)
        (lambda (p)
184
185
          (match p
            ((some-atom t) ((apply-env-to-term env) t))
186
187
            (((some-sent-con pc) (some-list args)) (pc (map (apply-env-to-sent env) args)))
            (((some-quant q) (some-var x) (some-sent body))
188
                 (q x ((apply-env-to-sent (add-binding x x env)) body)))))
189
190
   (define (apply-env env x)
191
192
     ((apply-env-to-sent env) x))
193
   (define (model-check p)
194
195
     (let ((bindings (cell [])))
        (letrec ((V (lambda (p env)
196
                       (match p
197
                         ((some-atom _) (eval (apply-env env p)))
198
199
                         ((not q) (negate (V q env)))
                         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
200
                         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
201
202
                         ((if p1 p2) (V (or (not p1) p2) env))
                         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
203
                         ((forall x body) (let ((terms (make-random-terms (sort-of x) 50)))
                                              (for-each terms
205
                                                         (lambda (t.)
206
                                                            (let ((res (V body (add-binding x t env)))
207
                                                                  ( (match res
208
```

```
(false (set! bindings (add [x --> t] (ref bindings))))
210
                                                                        (_ ())))
                                                              res)))))
                         ((exists x body) (let ((terms (make-random-terms (sort-of x) 50)))
212
                                              (for-some terms
213
                                                         (lambda (t)
214
                                                            (let ((res (V body (add-binding x t env)))
215
                                                                   (_ (match res
                                                                        (true (set! bindings (add [x --> t] (ref bindings))))
217
                                                                        ( ())))
218
219
                                                              res)))))))))
          (let ((res (V p empty-map)))
220
            (match (ref bindings)
221
222
              ([] res)
223
              (L [res L]))))))
224
225
   (define (model-check-bounded p N)
227
      (let ((bindings (cell [])))
228
        (letrec ((V (lambda (p env)
229
230
                       (match p
                         ((some-atom _) (eval (apply-env env p)))
231
                         ((neg q) (negate (V q env)))
232
233
                         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
234
                         ((if p1 p2) (V (or (not p1) p2) env))
235
236
                         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
                         ((forall x body) (let ((terms (make-random-terms (sort-of x) N)))
237
                                              (for-each terms
238
                                                         (lambda (t.)
239
                                                            (let ((res (V body (add-binding x t env)))
241
                                                                   ( (match res
                                                                        (false (set! bindings (add [x --> t] (ref bindings))))
242
243
                                                                        (_ ())))
                                                              res)))))
244
                         ((exists x body) (let ((terms (make-random-terms (sort-of x) N)))
                                              (for-some terms
246
                                                         (lambda (t)
247
                                                            (let ((res (V body (add-binding x t env)))
248
                                                                   (_ (match res
249
250
                                                                        (true (set! bindings (add [x --> t] (ref bindings))))
251
                                                                        (_ ())))
                                                              res)))))))))
252
          (let ((res (V p empty-map)))
253
            (match (ref bindings)
254
255
              ([] res)
              (L [res L]))))))
256
257
258
   (define (model-check-bounded p N)
259
      (let ((bindings (cell [])))
260
        (letrec ((V (lambda (p env)
261
262
                       (match p
                         ((some-atom _) (eval (apply-env env p)))
263
                         ((neg q) (negate (V q env)))
264
265
                         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
266
                         ((if p1 p2) (V (or (not p1) p2) env))
267
                         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
268
                         ((forall x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
270
                                              (for-each terms
271
                                                         (lambda (t)
                                                            (let ((res (V body (add-binding x t env)))
272
                                                                   ( (match res
273
                                                                        (false (set! bindings (add [x --> t] (ref bindings))))
                                                                        (_ ())))
275
276
                                                              res)))))
                         ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
277
                                              (for-some terms
278
```

```
(lambda (t)
                                                            (let ((res (V body (add-binding x t env)))
280
                                                                   (_ (match res
                                                                        (true (set! bindings (add [x --> t] (ref bindings))))
282
                                                                        (_ ())))
283
                                                              res)))))))))
284
          (let ((res (V p empty-map)))
285
            (match (ref bindings)
              ([] res)
287
              (L [res L]))))))
288
289
290
   (define (model-check-bounded p N)
291
      (let ((bindings (table 10)))
292
        (letrec ((V (lambda (p env)
293
294
                       (match p
                         ((some-atom _) (eval (apply-env env p)))
295
                         ((neg q) (negate (V q env)))
                         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
297
                         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
298
                         ((if p1 p2) (V (or (not p1) p2) env))
299
                         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
300
                         ((forall x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
301
                                              (for-each terms
302
303
                                                         (lambda (t)
                                                            (let ((res (V body (add-binding x t env)))
304
                                                                   (_ (table-add bindings [x --> t])))
305
                                                              res)))))
306
                         ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
307
                                              (for-some terms
308
                                                         (lambda (t.)
309
310
                                                            (let ((res (V body (add-binding x t env)))
                                                                   (_ (table-add bindings [x --> t])))
311
                                                              res)))))))))
312
313
          (let ((res (V p empty-map)))
            (match (table->list bindings)
314
              ([] res)
              (L [res L]))))))
316
317
    (define (model-check-bounded p N)
318
      (let ((bindings (table 10)))
319
320
        (letrec ((V (lambda (p env)
                       (match p
321
322
                         ((some-atom _) (let ((q (apply-env env p))
                                                  (_ (print "\nabout to eval q: " q))
323
                                                 (res (eval-silent q))
324
325
                                                  (_ (print "\nresult: " res))
                                                 (_ ()))
326
327
                                             res))
                         ((neg q) (negate (V q env)))
328
                         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
329
                         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
330
                         ((if p1 p2) (V (or (not p1) p2) env))
331
332
                         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
                         ((forall x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
333
                                              (for-each terms
334
335
                                                         (lambda (t)
                                                            (let ((res (V body (add-binding x t env)))
336
                                                                   (_ (table-add bindings [x --> t])))
337
338
                                                              res)))))
                         ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
340
                                              (for-some terms
341
                                                         (lambda (t)
                                                            (let ((res (V body (add-binding x t env)))
342
                                                                  ( (table-add bindings [x --> t])))
343
                                                              res)))))))))
          (let ((res (V p empty-map)))
345
346
             [res bindings]))))
347
   (define (mcb p N)
348
```

```
(check ((poly? p) (model-check-bounded (make-monomorphic-instance p) N))
            (else (model-check-bounded p N))))
350
352
            # (negate (lambda (x)
353
                           (match x
354
                            (true false)
355
                             (false true)
357
                             (_ (not x)))))
358
   (define (falsify p N)
359
      (let ((T (table 10))
360
            (get-bound (lambda (N x)
361
                            (check ((numeral? N) N)
362
363
                                   (else (N x)))))
            (vars-of-interest (table 10)))
364
        (letrec ((apply-env (lambda (t)
365
                                  ((some-var _) (try (table-lookup T t) (let ((_ (print "\nNothing for this: " t))) t)))
367
                                  (((some-symbol f) (some-list args)) (let ((res (make-term f (map apply-env args)))) res)))
368
                  (falsify (lambda (p)
369
370
                               (match p
                                 ((some-atom _) (let ((q (apply-env p))
371
                                                        #(_ (print "\nAbout to evaluate this term: " q))
372
373
                                                        (res (match (eval-silent (apply-env p))
374
                                                               (() (let ((_ ())
                                                                          #(_ (print "\nUnit result on this term: " p))
375
376
                                                                         ) true)) (res res)))
                                                       #(_ (print "\nResult: " res))
377
378
                                                    (negate res)))
379
                                 ((not q) (verify q))
                                 ((and (some-list args)) (for-some args falsify))
381
                                 ((or (some-list args)) (for-each args falsify))
382
383
                                 ((if p1 p2) (falsify (or (not p1) p2)))
                                 ((iff p1 p2) (|| (falsify (if p1 p2))
384
                                                    (falsify (if p2 p1))))
                                  ((\textit{forall x q}) \ (\textit{let} \ ((\textit{terms (st (make-all-ground-terms (sort-of x)) (get-bound N x)))}) \\
386
                                                   (for-some terms
387
388
                                                              (lambda (t)
                                                                 (let ((_ (table-add T [x --> t]))
389
                                                                        (_ (table-add vars-of-interest [x --> t])))
390
391
                                                                     (falsify q))))))
                                 ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
392
393
                                                    (for-each terms
                                                               (lambda (t)
394
395
                                                                 (let ((_ (table-add T [x --> t])))
                                                                   (falsify q)))))))))
396
397
                  (verify (lambda (p)
                             (match p
398
                               ((some-atom _) (let (#(_ (print "\nAbout to evaluate this term: " p))
399
                                                       (res (match (eval-silent (apply-env p))
400
                                                               (() false) (res res)))
401
402
                                                       #(_ (print "\nResult: " res))
403
                                                        )
                                                 res))
404
405
                               ((not q) (falsify q))
                               ((and (some-list args)) (for-each args verify))
406
                               ((or (some-list args)) (for-some args verify))
407
                               ((if p1 p2) (verify (or (not p1) p2)))
408
                               ((iff p1 p2) (&& (verify (if p1 p2))
410
                                                  (verify (if p2 p1))))
                               ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
411
412
                                                 (for-each terms
                                                            (lambda (t)
413
                                                               (let ((_ (table-add T [x --> t])))
                                                                  (verify q))))))
415
416
                               ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
417
                                                  (for-some terms
                                                             (lambda (t)
418
```

```
(let ((_ (table-add T [x --> t]))
                                                                        (_ (table-add vars-of-interest [x --> t])))
420
                                                                    (verify q)))))))))
421
          (let ((p (check ((poly? p) (make-monomorphic-instance p))
422
                             (else p))))
423
             (match (falsify p)
424
               (true ['success (make-map (table->list vars-of-interest))])
425
                      'failure))))))
427
428
    (define (falsify p N)
429
      (let ((T (table 10))
430
             (get-bound (lambda (N x)
431
                             (check ((numeral? N) N)
432
433
                                     (else (N x)))))
434
             (vars-of-interest (table 10)))
        (letrec ((apply-env (lambda (t)
435
                                    ((\textbf{some-var} \_) \ (\textbf{try} \ (\textbf{table-lookup} \ \texttt{T} \ \textbf{t}) \ (\textbf{let} \ ((\_ \ (\textbf{print} \ \texttt{"} \ \textbf{n}) \textbf{n}) \textbf{t}))) \\
437
                                    (((some-symbol f) (some-list args)) (let ((res (make-term f (map apply-env args)))) res)))
438
                  (falsify (lambda (p)
439
                                (match p
440
                                  ((some-atom _) (let ((q (apply-env p))
441
                                                          #(_ (print "\nAbout to evaluate this term: " q))
442
443
                                                         (res (match (eval-silent (apply-env p))
444
                                                                 (() (let ((_ ())
                                                                            #(_ (print "\nUnit result on this term: " p))
445
446
                                                                           ) true)) (res res)))
                                                         #(_ (print "\nResult: " res))
447
448
449
                                                      (negate res)))
                                  ((not q) (verify q))
451
                                  ((and (some-list args)) (for-some args falsify))
                                  ((or (some-list args)) (for-each args falsify))
452
453
                                   ((if p1 p2) (falsify (or (not p1) p2)))
                                  ((iff p1 p2) (|| (falsify (if p1 p2))
454
                                                      (falsify (if p2 p1))))
455
                                  ((forall\ x\ q)\ (\textbf{let}\ ((terms\ (st\ (make-all-ground-terms\ (sort-of\ x))\ (get-bound\ N\ x)))))
456
                                                     (for-some terms
457
458
                                                                (lambda (t)
                                                                    (let ((_ (table-add T [x --> t]))
459
                                                                          (_ (table-add vars-of-interest [x --> t])))
460
461
                                                                       (falsify q))))))
                                   ((exists x q) (check ((member? x (fv q)) false) (else (falsify q)))))))
462
                  (verify (lambda (p)
463
                              (match p
464
465
                                ((some-atom _) (let (#(_ (print "\nAbout to evaluate this term: " p))
                                                         (res (match (eval-silent (apply-env p))
466
467
                                                                 (() false) (res res)))
                                                             (print "\nResult: " res))
468
469
                                                          )
                                                   res))
470
                                ((not q) (falsify q))
471
472
                                ((and (some-list args)) (for-each args verify))
                                ((or (some-list args)) (for-some args verify))
473
                                ((if p1 p2) (verify (or (not p1) p2)))
474
475
                                ((iff p1 p2) (&& (verify (if p1 p2))
                                                    (verify (if p2 p1))))
476
                                ((forall x q) (check ((member? x (fv q)) false)
477
478
                                                        (else false)))
                                ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
480
                                                   (for-some terms
481
                                                               (lambda (t)
482
                                                                 (let ((_ (table-add T [x --> t]))
                                                                        (_ (table-add vars-of-interest [x --> t])))
483
                                                                    (verify q)))))))))
          (let ((p (check ((poly? p) (make-monomorphic-instance p))
485
486
                             (else p))))
487
             (match (falsify p)
               (true ['success (make-map (table->list vars-of-interest))])
488
```

```
'failure))))))
490
   (define (verify p N)
491
492
      (let ((T (table 10))
            (vars-of-interest (table 10)))
493
494
        (letrec ((apply-env (lambda (t)
                               (match t
495
                                  ((some-var _) (try (table-lookup T t) t))
                                  (((some-symbol f) (some-list args)) (make-term f (map apply-env args)))))))
497
                  (falsify (lambda (p)
498
                               (match p
499
                                 ((some-atom _) (negate (eval-silent (apply-env p))))
500
                                 ((not q) (verify q))
501
                                 ((and (some-list args)) (for-some args falsify))
502
                                 ((or (some-list args)) (for-each args falsify))
503
                                 ((if p1 p2) (falsify (or (not p1) p2)))
504
                                 ((iff p1 p2) (|| (falsify (if p1 p2))
505
                                                   (falsify (if p2 p1))))
                                 ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
507
                                                  (for-some terms
508
                                                             (lambda (t)
509
                                                                (let ((_ (table-add T [x --> t]))
510
                                                                       (_ (table-add vars-of-interest [x --> t])))
511
                                                                   (falsify q))))))
512
513
                                 ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
514
                                                   (for-each terms
515
                                                              (lambda (t)
                                                                (let ((_ (table-add T [x --> t])))
516
                                                                  (falsify q)))))))))
517
                  (verify (lambda (p)
518
                            (match p
519
                               ((some-atom _) (eval-silent (apply-env p)))
521
                               ((not q) (falsify q))
                               ((and (some-list args)) (for-each args verify))
522
523
                               ((or (some-list args)) (for-some args verify))
                               ((if p1 p2) (verify (or (not p1) p2)))
524
                               ((iff p1 p2) (&& (verify (if p1 p2))
                                                 (verify (if p2 p1))))
526
                               ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
527
528
                                                (for-each terms
                                                           (lambda (t)
529
                                                              (let ((_ (table-add T [x --> t])))
530
531
                                                                 (verify q))))))
                               ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
532
533
                                                 (for-some terms
                                                            (lambda (t)
534
                                                              (let ((_ (table-add T [x --> t]))
                                                                    (_ (table-add vars-of-interest [x --> t])))
536
537
                                                                (verify q)))))))))
          (let ((p (check ((poly? p) (make-monomorphic-instance p))
538
                           (else p))))
539
            (match (verify p)
540
              (true ['success (make-map (table->list vars-of-interest))])
541
542
                     'failure))))))
543
   (define (ground-bounded0 p N)
544
545
      (match p
        (((some-quant q) (some-var v) body)
546
547
            (let ((qsort (sort-of v))
                  (terms (st (make-all-ground-terms (sort-of v)) (N qsort))))
548
                (forall (and (map (lambda (t) (replace-var v t (ground-bounded0 body N))) terms)))
550
                (exists (or (map (lambda (t) (replace-var v t (ground-bounded0 body N))) terms)))
551
552
        (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded0 arg N)) args)))
553
554
        (_ p)))
555
556
557
        # (_ (let ((dom-sorts (filter-out (map sort-of (subterms p))
558
```

```
#
                                                      (lambda (s) (|| (datatype-sort? s) (member? s ["Int" "Real" "Ide"])))))
                             (_ (map (lambda (s) (make-fresh-constants s (val->string N))) dom-sorts)))
560
               p))))
561
562
563
   (define (make-all-terms S N)
564
      (let ((fvars (filter (fv (ab)) (lambda (v) (equal? (sort-of v) S)))))
565
        (join fvars (st (make-all-ground-terms S) N))))
566
567
   (define (ground-bounded0' p N)
568
569
      (match p
        (((some-quant q) (some-var v) body)
570
571
            (let ((terms (make-all-terms (sort-of v) N)))
572
              (match q
                 (forall (and (map (lambda (t) (replace-var v t (ground-bounded0' body N))) terms)))
573
                 (exists (or (map (lambda (t) (replace-var v t (ground-bounded0' body N))) terms)))
574
                 (((g))))
575
        (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded0' arg N)) args)))
577
        (_ p)))
578
579
580
581
   (define (ground-bounded' p N)
582
583
      (check ((less? N 1) [])
             ((poly? p) (ground-bounded0' (make-monomorphic-instance p) N))
584
             (else (ground-bounded0' p N))))
585
586
   (define (ground-bounded p N)
587
       (let ((how-many (match N
588
                           (\ (\texttt{some-term}\ \_)\ (\texttt{lambda}\ (\_)\ \texttt{N})\ )
589
590
                           (_ N))))
        (check ((&& (term? N) (less? N 1)) [])
591
               ((poly? p) (ground-bounded0 (make-monomorphic-instance p) how-many))
592
593
               (else (ground-bounded0 p how-many)))))
594
   (define gb ground-bounded)
595
596
   (define (quant-depth p)
597
598
      (match p
        (((some-quant q) (some-var v) body)
599
           (plus 1 (quant-depth body)))
600
        (((some-sent-con sc) (some-list args)) (max* (map quant-depth args)))
601
        ((some-sent _) 0)
602
603
        ((some-list L) (max* (map quant-depth L)))
        (_ 0)))
604
   (define qd quant-depth)
606
607
   (define (ground-bounded-2 p sort-elem-table)
608
    (check ((greater? (guant-depth p) 4) p)
609
      (else
610
      (match p
611
612
        (((some-quant q) (some-var v) body)
            (let ((terms (table-lookup sort-elem-table (sort-of v))))
613
614
615
                 (forall (and (map (lambda (t) (replace-var v t (ground-bounded-2 body sort-elem-table))) terms)))
                 (exists (or (map (lambda (t) (replace-var v t (ground-bounded-2 body sort-elem-table))) terms)))
616
617
        (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded-2 arg sort-elem-table)) args)))
618
619
        (_ p)))))
620
621
622
   (define (quant-count p)
623
      (match p
        (((some-quant q) (some-var v) body)
625
           (plus 1 (quant-count body)))
626
627
        (((some-sent-con sc) (some-list args)) (plus* (map quant-count args)))
        ( 0)))
628
```

```
(define (ground-props props N)
630
      (let ((g (lambda (p)
631
                  (check ((poly? p) (ground-bounded (make-monomorphic-instance p) N))
632
                          (else (ground-bounded p N))))))
633
634
        (map q props)))
635
   (define gp ground-props)
636
637
   (define (size* L) (plus* (map size L)))
638
639
   (define (ground-eval2 p sort-elem-table)
640
      (let (#(_ (print "\nGrounding the evaluation of this defining axiom: " p " using this sort-elem-table: " sort-elem-t
641
642
            ([t f mono-args] (match p
                                  ((forall (some-list _) (= (t as ((some-symbol f) (some-list args))) RHS)) [t f args])
643
                                   ((forall (some-list _) (if _ (= (t as ((some-symbol f) (some-list args))) RHS))) [t f args
644
            (term-list-for-each-arg (map (lambda (t)
645
                                               (let ((S (sort-of t)))
                                                 (table-lookup sort-elem-table S)))
647
648
                                            mono-args))
            (ar (arity-of f))
649
            (product (check ((less? (arity-of f) 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
650
                              (else (cprods term-list-for-each-arg))))
651
            (silent-eval-mode-value (ref silent-eval-mode))
652
653
            (_ (set! silent-eval-mode true))
654
            (eval-pair (lambda (p)
                           (try (let ((term (make-term f p))
655
                                       (value (eval term)))
656
                                  (= term value))
657
                                ())))
658
            (res (map-select eval-pair product (unequal-to ())))
659
            (_ (set! silent-eval-mode silent-eval-mode-value)))
660
661
662
    (define (ground-eval-proc f card-map)
663
      (let ((ar (arity-of f))
664
            (t (make-monomorphic-instance (make-term f (map (lambda (_) (fresh-var)) (from-to 1 ar)))))
665
            (term-list-for-each-arg (map (lambda (x)
666
                                               (let ((S
                                                         (sort-of x))
667
668
                                                     (N (try (card-map S) card-map)))
                                                 (st (make-all-ground-terms S) N)))
669
                                             (children t)))
670
            (\texttt{product} \ (\texttt{check} \ ((\texttt{less? ar 2}) \ (\texttt{map} \ (\texttt{lambda} \ (\texttt{x}) \ [\texttt{x}]) \ (\texttt{first term-list-for-each-arg})))
671
                              (else (cprods term-list-for-each-arg))))
672
673
            (silent-eval-mode-value (ref silent-eval-mode))
            (_ (set! silent-eval-mode true))
674
            (eval-pair (lambda (p)
                           (try (let ((term (make-term f p)))
676
677
                                     # (_ (print "\nAbout to evaluate this term: " term))
                                       (value (eval term)))
678
                                  (= term value))
679
                                ())))
680
            (res (map-select eval-pair product (unequal-to ())))
681
682
            (_ (set! silent-eval-mode silent-eval-mode-value)))
683
        res))
684
685
   (define (ground-eval-proc-2 f sort-table)
686
      (let ((ar (arity-of f))
687
            (t (make-monomorphic-instance (make-term f (map (lambda (_) (fresh-var)) (from-to 1 ar)))))
688
689
            (term-list-for-each-arg (map (lambda (t)
690
                                                 (table-lookup sort-table (sort-of t)))
                                            (children t)))
691
692
            (product (check ((less? ar 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
                              (else (cprods term-list-for-each-arg))))
693
            (silent-eval-mode-value (ref silent-eval-mode))
            ( (set! silent-eval-mode true))
695
            (eval-pair (lambda (p)
696
697
                           (try (let ((term (make-term f p)))
                                     # (_ (print "\nAbout to evaluate this term: " term))
698
```

```
(value (eval term)))
                                 (= term value))
700
701
                               ())))
            (res (map-select eval-pair product (unequal-to ())))
702
            (_ (set! silent-eval-mode silent-eval-mode-value)))
703
704
       res))
705
706
   (define (smt-prove goal)
707
     (dlet ((props (map (lambda (p) (ground-bounded p 1)) (ab)))
708
             (ht (table 10))
709
             (_ (table-add ht ['solver --> 'yices]))
710
711
             (p (and (add (not goal) props)))
             (_ (print "\nGOAL: " p))
712
             (res (smt-solve p ht)))
713
       (dmatch res
714
         ('Unsatisfiable (!force goal))
715
          (_ (!proof-error "\nCannot smt-prove the given goal.")))))
716
```