

## lib/basic/dt-model-checker.ath

```

1  # There are 4 main procedures here: (make-random-term S depth), (make-random-terms S N),
2  # (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
6  # generated using make-random-term. The algorithm will try to distribute evenly the
7  # 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
9  # 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;
11 # 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
15 # are produced as part of the output. This assumes that all bound variables in p have
16 # distinct names, which is easy to ensure (just use rename before passing p to model-check).
17
18 (load-file (file-path [ATHENA_LIB "dt-streams.ath"]))
19
20 (load-file (file-path [ATHENA_LIB "maps.ath"]))
21
22 (define (random-range-element low high)
23   (let ((i (random-int (plus 1 (minus high low)))))
24     (plus (minus i 1) low)))
25
26 (define (choose-random-integer)
27   (random-range-element 0 100))
28
29 (define (choose-random-real)
30   (let ((i1 (choose-random-integer))
31         (i2 (choose-random-integer))
32         (str (join (val->string i1) "." (val->string i2))))
33     (string->num str)))
34
35 (define (choose-random-identifier)
36   (string->id (join "x" (val->string (random-int 10000)))))
37
38 (define
39   (infinite-depth-sort? S)
40   (|| (negate (null? (reflexive-constructors-of S)))
41       (for-some (irreflexive-constructors-of S) (lambda (c) (infinite-depth-constructor? c S))))
42   (infinite-depth-constructor? irc S)
43   (for-some (arg-sorts-unified irc S) infinite-depth-sort?)
44   (infinite-value-but-finite-depth-constructor? c S)
45   (&& (negate (infinite-depth-constructor? c S))
46       (for-some (arg-sorts-unified c S) (lambda (S) (infinite-value-but-finite-depth-sort? S))))
47   (infinite-value-but-finite-depth-sort? S)
48   (&& (negate (infinite-depth-sort? S))
49       (|| (member? S ["Int" "Ide"])
50           (for-some (constructors-of S)
51                     (lambda (c) (infinite-value-but-finite-depth-constructor? c S)))))
52   (infinite-sort? S)
53   (|| (infinite-depth-sort? S) (infinite-value-but-finite-depth-sort? S)))
54
55
56 (define infinite-depth-sort? (memoize-unary infinite-depth-sort?))
57 (define infinite-depth-constructor? (memoize-binary infinite-depth-constructor?))
58 (define infinite-value-but-finite-depth-constructor? (memoize-binary infinite-value-but-finite-depth-constructor?))
59 (define infinite-value-but-finite-depth-sort? (memoize-unary infinite-value-but-finite-depth-sort?))
60 (define infinite-sort? (memoize-unary infinite-sort?))
61
62 (define (infinite-at-each-level? S)
63   (let ((mem (cell [])))
64     (letrec ((loop (lambda (S)
65                      (check ((member? S (ref mem)) false)
66                              (else (let ((_ (set! mem (add S (ref mem)))))
67                                      (&& (infinite-depth-sort? S)
68                                          (for-each (constructors-of S)
69                                                        (lambda (c) (loop (c S)))))))))))
67     (loop S)
68     (ref mem)))

```

```

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

```



```

209         (false (set! bindings (add [x --> t] (ref bindings))))
210         (_ ())))
211         res))))
212     ((exists x body) (let ((terms (make-random-terms (sort-of x) 50)))
213       (for-some terms
214         (lambda (t)
215           (let ((res (V body (add-binding x t env)))
216             (_ (match res
217               (true (set! bindings (add [x --> t] (ref bindings))))
218               (_ ())))
219             res)))))))))
220   (let ((res (V p empty-map)))
221     (match (ref bindings)
222       ([[] res)
223        (L [res L])))))
224
225
226
227 (define (model-check-bounded p N)
228   (let ((bindings (cell [])))
229     (letrec ((V (lambda (p env)
230       (match p
231         ((some-atom _) (eval (apply-env env p)))
232         ((neg q) (negate (V q env)))
233         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
234         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
235         ((if p1 p2) (V (or (not p1) p2) env))
236         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
237         ((forall x body) (let ((terms (make-random-terms (sort-of x) N)))
238           (for-each terms
239             (lambda (t)
240               (let ((res (V body (add-binding x t env)))
241                 (_ (match res
242                   (false (set! bindings (add [x --> t] (ref bindings))))
243                   (_ ())))
244                 res)))))))))
245         ((exists x body) (let ((terms (make-random-terms (sort-of x) N)))
246           (for-some terms
247             (lambda (t)
248               (let ((res (V body (add-binding x t env)))
249                 (_ (match res
250                   (true (set! bindings (add [x --> t] (ref bindings))))
251                   (_ ())))
252                 res)))))))))
253         (let ((res (V p empty-map)))
254           (match (ref bindings)
255             ([[] res)
256              (L [res L])))))
257
258
259 (define (model-check-bounded p N)
260   (let ((bindings (cell [])))
261     (letrec ((V (lambda (p env)
262       (match p
263         ((some-atom _) (eval (apply-env env p)))
264         ((neg q) (negate (V q env)))
265         ((and (some-list args)) (for-each args (lambda (p) (V p env))))
266         ((or (some-list args)) (for-some args (lambda (p) (V p env))))
267         ((if p1 p2) (V (or (not p1) p2) env))
268         ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
269         ((forall x body) (let ((terms (st (make-all-ground-terms (sort-of x) N)))
270           (for-each terms
271             (lambda (t)
272               (let ((res (V body (add-binding x t env)))
273                 (_ (match res
274                   (false (set! bindings (add [x --> t] (ref bindings))))
275                   (_ ())))
276                 res)))))))))
277         ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x) N)))
278           (for-some terms

```

```

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

```



```

419                                     (let ((_ (table-add T [x --> t])))
420                                         (_ (table-add vars-of-interest [x --> t])))
421                                     (verify q)))))))))
422 (let ((p (check ((poly? p) (make-monomorphic-instance p))
423                 (else p))))
424   (match (falsify p)
425     (true ['success (make-map (table->list vars-of-interest))])
426     (_ 'failure))))))
427
428
429 (define (falsify p N)
430   (let ((T (table 10))
431         (get-bound (lambda (N x)
432                       (check ((numeral? N) N)
433                             (else (N x))))))
434     (vars-of-interest (table 10)))
435   (letrec ((apply-env (lambda (t)
436                         (match t
437                           ((some-var _) (try (table-lookup T t) (let ((_ (print "\nNothing for this: " t))) t))
438                           ((some-symbol f) (some-list args)) (let ((res (make-term f (map apply-env args)))) res)))
439     (falsify (lambda (p)
440               (match p
441                 ((some-atom _) (let ((q (apply-env p))
442                                     #(_ (print "\nAbout to evaluate this term: " q))
443                                     (res (match (eval-silent (apply-env p))
444                                           (()) (let ((_ ()))
445                                                  #(_ (print "\nUnit result on this term: " p))
446                                                  ) true)) (res res)))
447                                     #(_ (print "\nResult: " res))
448                                     )
449                                     (negate res)))
450                 ((not q) (verify q))
451                 ((and (some-list args)) (for-some args falsify))
452                 ((or (some-list args)) (for-each args falsify))
453                 ((if p1 p2) (falsify (or (not p1) p2)))
454                 ((iff p1 p2) (|| (falsify (if p1 p2))
455                                   (falsify (if p2 p1))))
456                 ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x)))
457                                       (for-some terms
458                                         (lambda (t)
459                                           (let ((_ (table-add T [x --> t]))
460                                                 (_ (table-add vars-of-interest [x --> t])))
461                                               (falsify q))))))
462                 ((exists x q) (check ((member? x (fv q)) false) (else (falsify q))))))
463     (verify (lambda (p)
464               (match p
465                 ((some-atom _) (let ((#(_ (print "\nAbout to evaluate this term: " p))
466                                     (res (match (eval-silent (apply-env p))
467                                           (()) false) (res res)))
468                                     #(_ (print "\nResult: " res))
469                                     )
470                                     res))
471                 ((not q) (falsify q))
472                 ((and (some-list args)) (for-each args verify))
473                 ((or (some-list args)) (for-some args verify))
474                 ((if p1 p2) (verify (or (not p1) p2)))
475                 ((iff p1 p2) (& (verify (if p1 p2))
476                                   (verify (if p2 p1))))
477                 ((forall x q) (check ((member? x (fv q)) false)
478                                       (else false))
479                 ((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]))
483                                                 (_ (table-add vars-of-interest [x --> t])))
484                                               (verify q))))))
485     (let ((p (check ((poly? p) (make-monomorphic-instance p))
486                     (else p))))
487       (match (falsify p)
488         (true ['success (make-map (table->list vars-of-interest))])

```

```

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

```



```

559      #                                     (lambda (s) (|| (datatype-sort? s) (member? s ["Int" "Real" "Ide"]))))))
560      #                                     (_ (map (lambda (s) (make-fresh-constants s (val->string N)) dom-sorts)))
561      #                                     p))))
562
563
564 (define (make-all-terms S N)
565   (let ((fvars (filter (fv (ab)) (lambda (v) (equal? (sort-of v) S))))))
566     (join fvars (st (make-all-ground-terms S) N))))
567
568 (define (ground-bounded0' p N)
569   (match p
570     ((some-quant q) (some-var v) body)
571     (let ((terms (make-all-terms (sort-of v) N)))
572       (match q
573         (forall (and (map (lambda (t) (replace-var v t (ground-bounded0' body N))) terms)))
574         (exists (or (map (lambda (t) (replace-var v t (ground-bounded0' body N))) terms)))
575         (_ p))))
576     (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded0' arg N)) args)))
577     (_ p)))
578
579
580
581
582 (define (ground-bounded' p N)
583   (check ((less? N 1) []))
584   ((poly? p) (ground-bounded0' (make-monomorphic-instance p) N))
585   (else (ground-bounded0' p N)))
586
587 (define (ground-bounded p N)
588   (let ((how-many (match N
589                     ((some-term _) (lambda (N)
590                                         (_ N))))
591         (check ((&& (term? N) (less? N 1)) []))
592         ((poly? p) (ground-bounded0 (make-monomorphic-instance p) how-many))
593         (else (ground-bounded0 p how-many)))))
594
595 (define gb ground-bounded)
596
597 (define (quant-depth p)
598   (match p
599     ((some-quant q) (some-var v) body)
600     (plus 1 (quant-depth body)))
601     (((some-sent-con sc) (some-list args)) (max* (map quant-depth args)))
602     ((some-sent _) 0)
603     ((some-list L) (max* (map quant-depth L)))
604     (_ 0)))
605
606 (define qd quant-depth)
607
608 (define (ground-bounded-2 p sort-elem-table)
609   (check ((greater? (quant-depth p) 4) p)
610   (else
611   (match p
612     ((some-quant q) (some-var v) body)
613     (let ((terms (table-lookup sort-elem-table (sort-of v))))
614       (match q
615         (forall (and (map (lambda (t) (replace-var v t (ground-bounded-2 body sort-elem-table))) terms)))
616         (exists (or (map (lambda (t) (replace-var v t (ground-bounded-2 body sort-elem-table))) terms)))
617         (_ p))))
618     (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded-2 arg sort-elem-table)) args)))
619     (_ p))))
620
621
622
623 (define (quant-count p)
624   (match p
625     ((some-quant q) (some-var v) body)
626     (plus 1 (quant-count body)))
627     (((some-sent-con sc) (some-list args)) (plus* (map quant-count args)))
628     (_ 0)))

```

```

629
630 (define (ground-props props N)
631   (let ((g (lambda (p)
632             (check ((poly? p) (ground-bounded (make-monomorphic-instance p) N))
633                    (else (ground-bounded p N))))))
634     (map g props)))
635
636 (define gp ground-props)
637
638 (define (size* L) (plus* (map size L)))
639
640 (define (ground-eval2 p sort-elem-table)
641   (let (#_ (print "\nGrounding the evaluation of this defining axiom: " p " using this sort-elem-table: " sort-elem-table)
642         ([t f mono-args] (match p
643                             ((forall (some-list _) (= (t as ((some-symbol f) (some-list args))) RHS)) [t f args])
644                             ((forall (some-list _) (if _ (= (t as ((some-symbol f) (some-list args))) RHS)) [t f args]
645                             (term-list-for-each-arg (map (lambda (t)
646                                                           (let ((S (sort-of t)))
647                                                             (table-lookup sort-elem-table S)))
648                                                           mono-args))
649                             (ar (arity-of f))
650                             (product (check ((less? (arity-of f) 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
651                                       (else (cprods term-list-for-each-arg))))
652                             (silent-eval-mode-value (ref silent-eval-mode))
653                             (_ (set! silent-eval-mode true))
654                             (eval-pair (lambda (p)
655                                           (try (let ((term (make-term f p))
656                                                         (value (eval term)))
657                                                 (= term value))
658                                           ())))
659                             (res (map-select eval-pair product (unequal-to ())))
660                             (_ (set! silent-eval-mode silent-eval-mode-value))
661                             res)))
662
663 (define (ground-eval-proc f card-map)
664   (let ((ar (arity-of f))
665         (t (make-monomorphic-instance (make-term f (map (lambda (_) (fresh-var)) (from-to 1 ar))))))
666     (term-list-for-each-arg (map (lambda (x)
667                                   (let ((S (sort-of x))
668                                         (N (try (card-map S) card-map)))
669                                     (st (make-all-ground-terms S) N)))
670                                   (children t)))
671     (product (check ((less? ar 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
672             (else (cprods term-list-for-each-arg))))
673     (silent-eval-mode-value (ref silent-eval-mode))
674     (_ (set! silent-eval-mode true))
675     (eval-pair (lambda (p)
676                 (try (let ((term (make-term f p))
677                           #_ (print "\nAbout to evaluate this term: " term))
678                     (value (eval term))
679                     (= term value))
680                 ())))
681     (res (map-select eval-pair product (unequal-to ())))
682     (_ (set! silent-eval-mode silent-eval-mode-value))
683     res))
684
685
686 (define (ground-eval-proc-2 f sort-table)
687   (let ((ar (arity-of f))
688         (t (make-monomorphic-instance (make-term f (map (lambda (_) (fresh-var)) (from-to 1 ar))))))
689     (term-list-for-each-arg (map (lambda (t)
690                                   (table-lookup sort-table (sort-of t)))
691                                   (children t)))
692     (product (check ((less? ar 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
693             (else (cprods term-list-for-each-arg))))
694     (silent-eval-mode-value (ref silent-eval-mode))
695     (_ (set! silent-eval-mode true))
696     (eval-pair (lambda (p)
697                 (try (let ((term (make-term f p))
698                           #_ (print "\nAbout to evaluate this term: " term))
699                     (value (eval term))
700                     (= term value))
701                 ())))

```

```

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

```