lib/basic/dt-model-checker.ath

```
# 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
# term of sort S and depth d, where "depth" here indicates the maximum-length chain of
# reflexive constructor applications. The input sort S must be either a datatype sort or
# 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
# 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
# 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
# quantifiers in p that are verified, then the values of the corresponding values are
# also provided as part of the output. Likewise, if there are any universally quantified
# sentences inside p that are falsified, the values of the corresponding bound variables
# 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"]))

(load-file (file-path [ATHENA_LIB "maps.ath"]))

(define (random-range-element low high)
   (let ((i (random-int (plus 1 (minus high low)))))
     (plus (minus i 1) low)))

(define (choose-random-integer)
  (random-range-element 0 100))

(define (choose-random-real)
  (let ((i1 (choose-random-integer))
        (i2 (choose-random-integer))
        (str (join (val->string i1) "." (val->string i2))))
    (string->num str)))

(define (choose-random-identifier)
  (string->id (join "x" (val->string (random-int 10000)))))

(define
  (infinite-depth-sort? S)
     (|| (negate (null? (reflexive-constructors-of S)))
         (for-some (irreflexive-constructors-of S) (lambda (c) (infinite-depth-constructor? c S))))
  (infinite-depth-constructor? irc S)
    (for-some (arg-sorts-unified irc S) infinite-depth-sort?)
  (infinite-value-but-finite-depth-constructor? c S)
    (&& (negate (infinite-depth-constructor? c S))
        (for-some (arg-sorts-unified c S) (lambda (S) (infinite-value-but-finite-depth-sort? S))))
  (infinite-value-but-finite-depth-sort? S)
    (&& (negate (infinite-depth-sort? S))
        (|| (member? S ["Int" "Ide"])
            (for-some (constructors-of S)
                      (lambda (c) (infinite-value-but-finite-depth-constructor? c S)))))
  (infinite-sort? S)
    (|| (infinite-depth-sort? S) (infinite-value-but-finite-depth-sort? S)))


(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?))

(define (infinite-at-each-level? S)
  (let ((mem (cell [])))
    (letrec ((loop (lambda (S)
                      (check ((member? S (ref mem)) false)
                             (else (let ((_ (set! mem (add S (ref mem)))))
                                     (&& (infinite-depth-sort? S)
                                         (for-each (constructors-of S)
                                                   (lambda (c)
                                                      (for-some (arg-sorts-unified c S) (lambda (S') (|| (infinite-val
                                                                                                         (loop S')))))
      (loop S))))

(define infinite-at-each-level? (memoize-unary infinite-at-each-level?))

(define (make-random-term sort d)
 (let (([irc's rc's] (filter-and-complement (constructors-of sort) (lambda (c) (irreflexive-unif? c sort))))
       ([idirc's nidirc's] (filter-and-complement irc's (lambda (c) (infinite-depth-constructor? c sort))))
       (make-bottom-term (lambda ()
                           (let ((irc (try (choose nidirc's) (choose idirc's)))
                                 (arg-terms (map (lambda (S) (make-random-term S 0)) (arg-sorts-unified irc sort))))
                             (make-term irc arg-terms))))))
   (check ((equal? sort "Int") (choose-random-integer))
          ((equal? sort "Ide") (choose-random-identifier))
          ((equal? sort "Real") (choose-random-real))
          ((less? d 1) (make-bottom-term))
          (else (match (join rc's idirc's)
                  ([] (make-random-term sort 0))
                  (cs (let ((c (choose cs))
                            (choose-height (lambda () (random-range-element 0 (minus d 1))))
                            (c-arg-sorts (arg-sorts-unified c sort))
                            (max-height-child (choose (filter (from-to 1 (arity-of c))
                                                              (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)))
                            (arg-terms1 (map (lambda (S) (make-random-term S (choose-height))) L1))
                            (max-term (make-random-term x (minus d 1)))
                            (arg-terms2 (map (lambda (S) (make-random-term S (choose-height))) L2))
                            (arg-terms (join arg-terms1 (add max-term arg-terms2))))
                        (make-term c arg-terms)))))))

(define (has-at-least-binary-ref-con S)
  (let ((mem (cell [])))
    (letrec ((loop (lambda (S)
                     (check ((member? S (ref mem)) false)
                            (else (let ((_ (set! mem (add S (ref mem)))))
                                    (for-some (constructors-of S)
                                              (lambda (c)
                                                (let ((c-arg-sorts (arg-sorts-unified c S)))
                                                  (|| (greater? (length (filter c-arg-sorts (lambda (S') (equal? S' S)
                                                      (for-some c-arg-sorts loop))))))))))
      (loop S))))


(define has-at-least-binary-ref-con (memoize-unary has-at-least-binary-ref-con))

(define (has-at-least-one-binary-con? S)
  (for-some (constructors-of S) (lambda (c) (greater? (arity-of c) 1))))

(define (decide N depth sort)
  (let ((d  (check ((for-some (irreflexive-constructors-of sort)
                              (lambda (c) (infinite-value-but-finite-depth-constructor? c sort)))
                      (plus depth 1))
                   (else depth)))
        (x (div N d)))
    (check ((&& (leq? x 1) (less? depth N))
                (div N depth))
           (else x))))

(define decide (memoize-ternary decide))

(define (make-random-terms sort N)
  (let ((ht' (make-term-hash-table 983))
        (is-infinite-at-each-level (infinite-at-each-level? sort))
        (make (lambda (d)
                (letrec ((loop (lambda ()
                                 (let ((t (make-random-term sort d)))
                                   (match (term-look-up ht' t)
                                    (() (term-enter ht' t true))
                                    (_ (check ((&& (greater? d 0) is-infinite-at-each-level) (loop))
                                              (else ())))))))))
                  (loop)))))
    (check ((|| (negate (infinite-depth-sort? sort))
                (negate (infinite-sort? sort)))
               (stream-take (make-all-ground-terms sort) N))
           (else (let ((has-at-least-one-binary-ref-con? (has-at-least-binary-ref-con sort))
                       (depth (check ((negate (infinite-at-each-level? sort))
                                        (check ((has-at-least-one-binary-con? sort) 10)
                                               (else N)))
                                     (has-at-least-one-binary-ref-con? 5)
                                     (else 8)))
                       ([depth' count] (check ((leq? N depth) [N 1])
                                              (else [depth (decide N depth sort)])))
                       (range (from-to 1 count))
                       (_ (map (lambda (d)
                                       (map (lambda (_) (make d)) range))
                               (from-to 0 depth')))
                       (results (map first (show-table ht')))
                       (sorted-results (merge-sort results (lambda (t1 t2) (less? (height t1) (height t2)))))
                        (_ ()))
                    sorted-results)))))

(define (dup-count L)
  (letrec ((loop (lambda (L res)
                    (match L
                      ([] res)
                      ((list-of x more) (check ((member? x more) (loop more (plus res 1)))
                                               (else (loop more res))))))))
    (loop L 0)))

(define (test sort N)
  (let ((t1 (time))
        (L (make-random-terms sort N))
        (t2 (time))
        (dc (dup-count L)))
   (print "\nLength: " (length L) "\ndup-count: " dc "\nTotal time: " (minus t2 t1) "\n")))

(define
  (apply-env-to-term env)
    (lambda (t)
      (match t
        ((some-var v) (apply-map env v))
        (((some-symbol f) (some-list args)) (make-term f (map (apply-env-to-term env) args)))))
  (apply-env-to-sent env)
    (lambda (p)
      (match p
        ((some-atom t) ((apply-env-to-term env) t))
        (((some-sent-con pc) (some-list args)) (pc (map (apply-env-to-sent env) args)))
        (((some-quant q) (some-var x) (some-sent body))
             (q x ((apply-env-to-sent (add-binding x x env)) body))))))

(define (apply-env env x)
  ((apply-env-to-sent env) x))

(define (model-check p)
  (let ((bindings (cell [])))
    (letrec ((V (lambda (p env)
                  (match p
                    ((some-atom _) (eval (apply-env env p)))
                    ((not q) (negate (V q env)))
                    ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                    ((or (some-list args)) (for-some args (lambda (p) (V p env))))
                    ((if p1 p2) (V (or (not p1) p2) env))
                    ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
                    ((forall x body) (let ((terms (make-random-terms (sort-of x) 50)))
                                       (for-each terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (match res
                                                               (false (set! bindings (add [x --> t] (ref bindings))))
                                                               (_ ()))))
                                                      res)))))
                    ((exists x body) (let ((terms (make-random-terms (sort-of x) 50)))
                                       (for-some terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (match res
                                                               (true (set! bindings (add [x --> t] (ref bindings))))
                                                               (_ ()))))
                                                      res)))))))))
      (let ((res (V p empty-map)))
        (match (ref bindings)
          ([] res)
          (L [res L]))))))



(define (model-check-bounded p N)
  (let ((bindings (cell [])))
    (letrec ((V (lambda (p env)
                  (match p
                    ((some-atom _) (eval (apply-env env p)))
                    ((neg q) (negate (V q env)))
                    ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                    ((or (some-list args)) (for-some args (lambda (p) (V p env))))
                    ((if p1 p2) (V (or (not p1) p2) env))
                    ((iff p1 p2) (V (and (if p1 p2) (if p2 p1)) env))
                    ((forall x body) (let ((terms (make-random-terms (sort-of x) N)))
                                       (for-each terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (match res
                                                               (false (set! bindings (add [x --> t] (ref bindings))))
                                                               (_ ()))))
                                                      res)))))
                    ((exists x body) (let ((terms (make-random-terms (sort-of x) N)))
                                       (for-some terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (match res
                                                               (true (set! bindings (add [x --> t] (ref bindings))))
                                                               (_ ()))))
                                                      res)))))))))
      (let ((res (V p empty-map)))
        (match (ref bindings)
          ([] res)
          (L [res L]))))))


(define (model-check-bounded p N)
  (let ((bindings (cell [])))
    (letrec ((V (lambda (p env)
                  (match p
                    ((some-atom _) (eval (apply-env env p)))
                    ((neg q) (negate (V q env)))
                    ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                    ((or (some-list args)) (for-some args (lambda (p) (V p env))))
                    ((if p1 p2) (V (or (not p1) p2) env))
                    ((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)))
                                       (for-each terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (match res
                                                               (false (set! bindings (add [x --> t] (ref bindings))))
                                                               (_ ()))))
                                                      res)))))
                    ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                       (for-some terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (match res
                                                               (true (set! bindings (add [x --> t] (ref bindings))))
                                                               (_ ()))))
                                                      res)))))))))
      (let ((res (V p empty-map)))
        (match (ref bindings)
          ([] res)
          (L [res L]))))))


(define (model-check-bounded p N)
  (let ((bindings (table 10)))
    (letrec ((V (lambda (p env)
                  (match p
                    ((some-atom _) (eval (apply-env env p)))
                    ((neg q) (negate (V q env)))
                    ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                    ((or (some-list args)) (for-some args (lambda (p) (V p env))))
                    ((if p1 p2) (V (or (not p1) p2) env))
                    ((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)))
                                       (for-each terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (table-add bindings [x --> t])))
                                                      res)))))
                    ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                       (for-some terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (table-add bindings [x --> t])))
                                                      res)))))))))
      (let ((res (V p empty-map)))
        (match (table->list bindings)
          ([] res)
          (L [res L]))))))

(define (model-check-bounded p N)
  (let ((bindings (table 10)))
    (letrec ((V (lambda (p env)
                  (match p
                    ((some-atom _) (let  ((q (apply-env env p))
#                                          (_ (print "\nabout to eval q: " q))
                                          (res (eval-silent q))
#                                          (_ (print "\nresult: " res))
                                          (_ ()))
                                      res))
                    ((neg q) (negate (V q env)))
                    ((and (some-list args)) (for-each args (lambda (p) (V p env))))
                    ((or (some-list args)) (for-some args (lambda (p) (V p env))))
                    ((if p1 p2) (V (or (not p1) p2) env))
                    ((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)))
                                       (for-each terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (table-add bindings [x --> t])))
                                                      res)))))
                    ((exists x body) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                       (for-some terms
                                                 (lambda (t)
                                                    (let ((res (V body (add-binding x t env)))
                                                          (_ (table-add bindings [x --> t])))
                                                      res)))))))))
      (let ((res (V p empty-map)))
         [res bindings]))))

(define (mcb p N)
 (check ((poly? p) (model-check-bounded (make-monomorphic-instance p) N))
        (else (model-check-bounded p N))))


        # (negate (lambda (x)
        #            (match x
        #              (true false)
        #              (false true)
        #              (_ (not x)))))

(define (falsify p N)
  (let ((T (table 10))
        (get-bound (lambda (N x)
                      (check ((numeral? N) N)
                             (else (N x)))))
        (vars-of-interest (table 10)))
    (letrec ((apply-env (lambda (t)
                         (match t
                            ((some-var _) (try (table-lookup T t) (let ((_ (print "\nNothing for this: " t))) t)))
                            (((some-symbol f) (some-list args)) (let ((res (make-term f (map apply-env args)))) res)))
             (falsify (lambda (p)
                         (match p
                           ((some-atom _) (let ((q  (apply-env p))
                                                    #(_ (print "\nAbout to evaluate this term: " q))
                                                   (res (match (eval-silent (apply-env p))
                                                          (() (let ((_ ())
                                                                         #(_ (print "\nUnit result on this term: " p))
                                                                        ) true)) (res res)))
                                                   #(_ (print "\nResult: " res))
                                                    )
                                                (negate res)))
                           ((not q) (verify q))
                           ((and (some-list args)) (for-some args falsify))
                           ((or (some-list args)) (for-each args falsify))
                           ((if p1 p2) (falsify (or (not p1) p2)))
                           ((iff p1 p2) (|| (falsify (if p1 p2))
                                            (falsify (if p2 p1))))
                           ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
                                           (for-some terms
                                                     (lambda (t)
                                                        (let ((_ (table-add T [x --> t]))
                                                              (_ (table-add vars-of-interest [x --> t])))
                                                           (falsify q))))))
                           ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
                                            (for-each terms
                                                      (lambda (t)
                                                        (let ((_ (table-add T [x --> t])))
                                                          (falsify q))))))))
             (verify (lambda (p)
                       (match p
                         ((some-atom _) (let  (#(_ (print "\nAbout to evaluate this term: " p))
                                                   (res (match (eval-silent (apply-env p))
                                                          (() false) (res res)))
                                                   #(_ (print "\nResult: " res))
                                                    )
                                          res))
                         ((not q) (falsify q))
                         ((and (some-list args)) (for-each args verify))
                         ((or (some-list args)) (for-some args verify))
                         ((if p1 p2) (verify (or (not p1) p2)))
                         ((iff p1 p2) (&& (verify (if p1 p2))
                                          (verify (if p2 p1))))
                         ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
                                         (for-each terms
                                                   (lambda (t)
                                                      (let ((_ (table-add T [x --> t])))
                                                         (verify q))))))
                         ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
                                          (for-some terms
                                                    (lambda (t)
                                                      (let ((_ (table-add T [x --> t]))
                                                            (_ (table-add vars-of-interest [x --> t])))
                                                        (verify q))))))))))
      (let ((p (check ((poly? p) (make-monomorphic-instance p))
                      (else p))))
        (match (falsify p)
          (true ['success (make-map (table->list vars-of-interest))])
          (_     'failure)))))


(define (falsify p N)
  (let ((T (table 10))
        (get-bound (lambda (N x)
                      (check ((numeral? N) N)
                             (else (N x)))))
        (vars-of-interest (table 10)))
    (letrec ((apply-env (lambda (t)
                         (match t
                            ((some-var _) (try (table-lookup T t) (let ((_ (print "\nNothing for this: " t))) t)))
                            (((some-symbol f) (some-list args)) (let ((res (make-term f (map apply-env args)))) res)))
             (falsify (lambda (p)
                         (match p
                           ((some-atom _) (let ((q  (apply-env p))
                                                #(_ (print "\nAbout to evaluate this term: " q))
                                               (res (match (eval-silent (apply-env p))
                                                      (() (let ((_ ())
                                                                #(_ (print "\nUnit result on this term: " p))
                                                               ) true)) (res res)))
                                               #(_ (print "\nResult: " res))
                                                )
                                            (negate res)))
                           ((not q) (verify q))
                           ((and (some-list args)) (for-some args falsify))
                           ((or (some-list args)) (for-each args falsify))
                           ((if p1 p2) (falsify (or (not p1) p2)))
                           ((iff p1 p2) (|| (falsify (if p1 p2))
                                            (falsify (if p2 p1))))
                           ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
                                           (for-some terms
                                                     (lambda (t)
                                                        (let ((_ (table-add T [x --> t]))
                                                              (_ (table-add vars-of-interest [x --> t])))
                                                           (falsify q))))))
                           ((exists x q) (check ((member? x (fv q)) false) (else (falsify q)))))))
             (verify (lambda (p)
                       (match p
                         ((some-atom _) (let  (#(_ (print "\nAbout to evaluate this term: " p))
                                               (res (match (eval-silent (apply-env p))
                                                      (() false) (res res)))
                                               #(_ (print "\nResult: " res))
                                                )
                                          res))
                         ((not q) (falsify q))
                         ((and (some-list args)) (for-each args verify))
                         ((or (some-list args)) (for-some args verify))
                         ((if p1 p2) (verify (or (not p1) p2)))
                         ((iff p1 p2) (&& (verify (if p1 p2))
                                          (verify (if p2 p1))))
                         ((forall x q) (check ((member? x (fv q)) false)
                                              (else false)))
                         ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) (get-bound N x))))
                                          (for-some terms
                                                    (lambda (t)
                                                      (let ((_ (table-add T [x --> t]))
                                                            (_ (table-add vars-of-interest [x --> t])))
                                                        (verify q))))))))))
      (let ((p (check ((poly? p) (make-monomorphic-instance p))
                      (else p))))
        (match (falsify p)
          (true ['success (make-map (table->list vars-of-interest))])
          (_    'failure))))))

(define (verify p N)
  (let ((T (table 10))
        (vars-of-interest (table 10)))
    (letrec ((apply-env (lambda (t)
                         (match t
                            ((some-var _) (try (table-lookup T t) t))
                            (((some-symbol f) (some-list args)) (make-term f (map apply-env args))))))
             (falsify (lambda (p)
                         (match p
                           ((some-atom _) (negate (eval-silent (apply-env p))))
                           ((not q) (verify q))
                           ((and (some-list args)) (for-some args falsify))
                           ((or (some-list args)) (for-each args falsify))
                           ((if p1 p2) (falsify (or (not p1) p2)))
                           ((iff p1 p2) (|| (falsify (if p1 p2))
                                             (falsify (if p2 p1))))
                           ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                           (for-some terms
                                                     (lambda (t)
                                                        (let ((_ (table-add T [x --> t]))
                                                              (_ (table-add vars-of-interest [x --> t])))
                                                           (falsify q))))))
                           ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                            (for-each terms
                                                      (lambda (t)
                                                        (let ((_ (table-add T [x --> t])))
                                                          (falsify q)))))))))
             (verify (lambda (p)
                       (match p
                         ((some-atom _) (eval-silent (apply-env p)))
                         ((not q) (falsify q))
                         ((and (some-list args)) (for-each args verify))
                         ((or (some-list args)) (for-some args verify))
                         ((if p1 p2) (verify (or (not p1) p2)))
                         ((iff p1 p2) (&& (verify (if p1 p2))
                                          (verify (if p2 p1))))
                         ((forall x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                         (for-each terms
                                                   (lambda (t)
                                                      (let ((_ (table-add T [x --> t])))
                                                         (verify q))))))
                         ((exists x q) (let ((terms (st (make-all-ground-terms (sort-of x)) N)))
                                          (for-some terms
                                                    (lambda (t)
                                                      (let ((_ (table-add T [x --> t]))
                                                            (_ (table-add vars-of-interest [x --> t])))
                                                        (verify q))))))))))
      (let ((p (check ((poly? p) (make-monomorphic-instance p))
                      (else p))))
        (match (verify p)
          (true ['success (make-map (table->list vars-of-interest))])
          (_    'failure))))))

(define (ground-bounded0 p N)
  (match p
    (((some-quant q) (some-var v) body)
        (let ((qsort (sort-of v))
              (terms (st (make-all-ground-terms (sort-of v)) (N qsort))))
          (match q
            (forall (and (map (lambda (t) (replace-var v t (ground-bounded0 body N))) terms)))
            (exists (or (map (lambda (t) (replace-var v t (ground-bounded0 body N))) terms)))
            (_ p))))
    (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded0 arg N)) args)))
    (_ p)))



    # (_ (let ((dom-sorts (filter-out (map sort-of (subterms p))
    #                                          (lambda (s) (|| (datatype-sort? s) (member? s ["Int" "Real" "Ide"])))))
    #                  (_ (map (lambda (s) (make-fresh-constants s (val->string N))) dom-sorts)))
    #      p))))


(define (make-all-terms S N)
  (let ((fvars (filter (fv (ab)) (lambda (v) (equal? (sort-of v) S)))))
    (join fvars (st (make-all-ground-terms S) N))))

(define (ground-bounded0' p N)
  (match p
    (((some-quant q) (some-var v) body)
        (let ((terms (make-all-terms (sort-of v) N)))
          (match q
            (forall (and (map (lambda (t) (replace-var v t (ground-bounded0' body N))) terms)))
            (exists (or (map (lambda (t) (replace-var v t (ground-bounded0' body N))) terms)))
            (_ p))))
    (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded0' arg N)) args)))
    (_ p)))




(define (ground-bounded' p N)
  (check ((less? N 1) [])
         ((poly? p) (ground-bounded0' (make-monomorphic-instance p) N))
         (else (ground-bounded0' p N))))

(define (ground-bounded p N)
   (let ((how-many (match N
                     ((some-term _) (lambda (_) N))
                     (_ N))))
    (check ((&& (term? N) (less? N 1)) [])
           ((poly? p) (ground-bounded0 (make-monomorphic-instance p) how-many))
           (else (ground-bounded0 p how-many)))))

(define gb ground-bounded)

(define (quant-depth p)
  (match p
    (((some-quant q) (some-var v) body)
       (plus 1 (quant-depth body)))
    (((some-sent-con sc) (some-list args)) (max* (map quant-depth args)))
    ((some-sent _) 0)
    ((some-list L) (max* (map quant-depth L)))
    (_ 0)))

(define qd quant-depth)

(define (ground-bounded-2 p sort-elem-table)
 (check ((greater? (quant-depth p) 4) p)
  (else
  (match p
    (((some-quant q) (some-var v) body)
        (let ((terms (table-lookup sort-elem-table (sort-of v))))
          (match q
            (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)))
            (_ p))))
    (((some-sent-con sc) (some-list args)) (sc (map (lambda (arg) (ground-bounded-2 arg sort-elem-table)) args)))
    (_ p)))))



(define (quant-count p)
  (match p
    (((some-quant q) (some-var v) body)
       (plus 1 (quant-count body)))
    (((some-sent-con sc) (some-list args)) (plus* (map quant-count args)))
    (_ 0)))
```

```
(define (ground-props props N)
  (let ((g (lambda (p)
             (check ((poly? p) (ground-bounded (make-monomorphic-instance p) N))
                    (else (ground-bounded p N))))))
    (map g props)))

(define gp ground-props)

(define (size* L) (plus* (map size L)))

(define (ground-eval2 p sort-elem-table)
  (let (#(_ (print "\nGrounding the evaluation of this defining axiom: " p " using this sort-elem-t
        ([t f mono-args] (match p
                            ((forall (some-list _) (= (t as ((some-symbol f) (some-list args))) RHS)) [t f args])
                            ((forall (some-list _) (if _ (= (t as ((some-symbol f) (some-list args))) RHS))) [t f args
        (term-list-for-each-arg (map (lambda (t)
                                       (let ((S (sort-of t)))
                                         (table-lookup sort-elem-table S)))
                                     mono-args))
        (ar (arity-of f))
        (product (check ((less? (arity-of f) 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
                        (else (cprods term-list-for-each-arg))))
        (silent-eval-mode-value (ref silent-eval-mode))
        (_ (set! silent-eval-mode true))
        (eval-pair (lambda (p)
                     (try (let ((term (make-term f p))
                                (value (eval term)))
                            (= term value))
                          ()))))
        (res (map-select eval-pair product (unequal-to ())))
        (_ (set! silent-eval-mode silent-eval-mode-value)))
    res))

(define (ground-eval-proc f card-map)
  (let ((ar (arity-of f))
        (t (make-monomorphic-instance (make-term f (map (lambda (_) (fresh-var)) (from-to 1 ar)))))
        (term-list-for-each-arg (map (lambda (x)
                                       (let ((S  (sort-of x))
                                             (N (try (card-map S) card-map)))
                                         (st (make-all-ground-terms S) N)))
                                     (children t)))
        (product (check ((less? ar 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
                        (else (cprods term-list-for-each-arg))))
        (silent-eval-mode-value (ref silent-eval-mode))
        (_ (set! silent-eval-mode true))
        (eval-pair (lambda (p)
                     (try (let ((term (make-term f p))
                              # (_ (print "\nAbout to evaluate this term: " term))
                                (value (eval term)))
                            (= term value))
                          ()))))
        (res (map-select eval-pair product (unequal-to ())))
        (_ (set! silent-eval-mode silent-eval-mode-value)))
    res))


(define (ground-eval-proc-2 f sort-table)
  (let ((ar (arity-of f))
        (t (make-monomorphic-instance (make-term f (map (lambda (_) (fresh-var)) (from-to 1 ar)))))
        (term-list-for-each-arg (map (lambda (t)
                                         (table-lookup sort-table (sort-of t)))
                                     (children t)))
        (product (check ((less? ar 2) (map (lambda (x) [x]) (first term-list-for-each-arg)))
                        (else (cprods term-list-for-each-arg))))
        (silent-eval-mode-value (ref silent-eval-mode))
        (_ (set! silent-eval-mode true))
        (eval-pair (lambda (p)
                     (try (let ((term (make-term f p))
                              # (_ (print "\nAbout to evaluate this term: " term))
```

```
                                (value (eval term)))
                            (= term value))
                          ()))))
        (res (map-select eval-pair product (unequal-to ())))
        (_ (set! silent-eval-mode silent-eval-mode-value)))
    res))


(define (smt-prove goal)
  (dlet ((props (map (lambda (p) (ground-bounded' p 1)) (ab)))
         (ht (table 10))
         (_ (table-add ht ['solver --> 'yices]))
         (p (and (add (not goal) props)))
         (_ (print "\nGOAL: " p))
         (res (smt-solve p ht)))
    (dmatch res
      ('Unsatisfiable (!force goal))
      (_ (!proof-error "\nCannot smt-prove the given goal.")))))
```