lib/basic/dt-streams.ath

```
#(load-file (file-path [ATHENA_LIB "streams.ath"]))


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


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

(load-file "streams.ath")

#(load-file "rsarray.ath")

(load-file "maps.ath")

(define breadth-factor (cell 20))

# The above is the factor by which broader (bushier) terms are preferred to deeper terms.
# The minimum possible value for this factor is 1, which will tend to strongly
# favor deep terms in the generation of the infinite stream of all ground terms.
# A factor of 20-30 strikes a decent balance. A factor of 100 or 200 (or more)
# strongly favors broad rather than deep terms. Of course no matter what the
# value, all terms will be generated in the limit since this is an infinite
# stream. The issue is how the infinite stream is enumerated, i.e. which terms
# are more likely to be listed sooner rather than later.

# The procedure make-all-bounded-ground-terms takes a (datatype sort S and
# a non-negative integer depth d and generates a (possibly infinite) stream
# of all ground terms of sort S of depth d. The procedure make-all-ground-terms
# takes a sort S and returns (a probably infinite) stream of all and only
# the ground terms of S. It does this first by arranging by stream-mapping
# make-all-bounded-ground-terms to the infinite stream of all non-negative
# integers. This gives us an infinite stream of infinite streams. It then
# traverses this infinite matrix of terms in Cantor's way, but in a more
# flexible way: instead of always consuming one element of the matrix on
# any given move, it consumes (ref breadth-factor) elements from the first row,
# (ref breadth-factor) - 1 from the second row, (ref breadth-factor) - 2 from
# the third row, etc. It then goes back to the first row to bite another
# chunk of (ref breadth-factor) elements, and so on. Since lower rows have
# deeper terms, a high breadth-factor will ensure that the enumeration
# moves to deeper terms more slowly.

# Note that making all ground terms of depth d+1 first needs to make
# all ground terms of depth d. Dynamic programming is used to memoize
# the streams of each depth i = 0, 1, 2, ... from the bottom up so
# that they don't need to be repeatedly recomputed. The procedures
# share access to a hash table from datatype names to resizable vectors.
# Each element i of the vector corresponding to a datatype D holds
# the infinite stream of all ground terms of D of depth i. Without
# this memoization scheme the algorithm would be hopelessly inefficient.

#(load "./lib/basic/dt-model-checker.ath")

(define sort "(List-Of Boolean)")

(define ht (table 23))

(define sort-stream-table (table 10))

(define (register-sort-stream sort stream)
   (table-add sort-stream-table [sort --> stream]))

(define (unregister-sort-stream sort)
  (try (seq (table-remove sort-stream-table sort) ()) ()))

# (define f (all-bounded-ground-terms sort))
# (f 0) fails
# (process-irc Pair sort) fails...
# (all-ground-terms "(List-Of Boolean)") fails on its first inv
```

```
# (process-rc Cons "(List-Of Boolean)") fails
# (takes-args-of-sort true sort) fails
# (define [c S] [true "(List-Of Boolean)"])

(define [make-all-ground-terms
         make-all-bounded-ground-terms
         all-bounded-ground-terms
         all-ground-terms
         process-irc
         process-rc]
  (let ((ht1 (make-hash-table 23)))
    (letrec ((all-bounded-ground-terms
                (lambda (sort)
                  (lambda (d)
                    (let ((mem-table (match (try-looking-up sort ht)
                                       ([t] t)
                                       (_ (let ((A (make-rs-array 10000 () 2000))
                                                (_ (table-add ht [sort --> A])))
                                            A))))
                          (entry (rs-array-sub mem-table (plus d 1))))
                      (check ((unequal? entry ()) entry)
                             ((equal? d 0) (let ((res (fair-weave (map (lambda (c)
                                                                         (process-irc c sort))
                                                                       (irreflexive-constructors-of sort))))
                                                  (_ (rs-array-set mem-table (plus d 1) res)))
                                               res))
                             (else (let ((res (fair-weave (map (lambda (rc)
                                                                 ((process-rc d) rc sort))
                                                               (random-shuffle (reflexive-constructors-of sort)))))
                                                  (_ (rs-array-set mem-table (plus d 1) res)))
                                               res)))))))
             (process-irc (lambda (irc sort)
                            (check ((less? (arity-of irc) 1) (list->stream [irc]))
                                   (else (let ((arg-streams (map all-ground-terms (arg-sorts-unified irc sort))))
                                           (match arg-streams
                                             ([stream] (stream-map irc stream))
                                             (_ (stream-map (lambda (cprod)
                                                              (make-term irc cprod))
                                                            (stream-zip* arg-streams)))))))))
             (process-rc (lambda (d)
                           (lambda (rc sort)
                             (let ((sig (get-signature-unified rc sort))
                                   (arg-sorts (all-but-last sig))
                                   (range-sort (last sig))
                                   (arg-streams (map (lambda (arg-sort)
                                                       (check ((|| (equal? arg-sort range-sort)
                                                                   (for-some (constructors-of arg-sort)
                                                                             (lambda (c) (takes-args-of-sort c range-s
                                                              ((all-bounded-ground-terms arg-sort) (minus d 1)))
                                                              (else (all-ground-terms arg-sort))))
                                                     arg-sorts)))
                               (match arg-streams
                                 ([stream] (stream-map rc stream))
                                 (_ (stream-map (lambda (cprod)
                                                  (make-term rc cprod))
                                                (stream-zip* arg-streams)))))))))
             (all-ground-terms
                (lambda (sort)
                  (letrec ((getNext (lambda (front-streams back-streams i-streams how-many)
                                      (match front-streams
                                        ([] (check ((empty-stream? (stream-head i-streams))
                                                    (match back-streams
                                                      ([] empty-stream)
                                                      (_ (getNext (rev back-streams) [] i-streams
                                                   (else (let ((new-stream (stream-head i-streams)))
                                                           (getNext (rev (add new-stream back-streams
                                                   ((list-of stream more-streams)
                                                      (check ((empty-stream? stream)
                                                                (getNext more-streams back-streams i-streams how-
                                                              (else (let ((how-many' (check ((equal? how-many (ref
```

```
                                                                                                (else how-many)))
                                                                                    (chunk (list->stream (stream-take stream
                                                                                [(stream-head chunk)
                                                                                 (lambda ()
                                                                                    (stream-append
                                                                                      (stream-tail chunk)
                                                                                      (getNext more-streams
                                                                                               (add (stream-tail-k stream ho
                                                                                               i-streams (max 1 (minus how-m
                                                (check ((datatype-sort? sort)
                                                          (getNext [] [] (stream-map (all-bounded-ground-terms sort) non-negati
                                                       ((equal? sort "Int") all-integers)
                                                       ((equal? sort "Real") all-reals)
#                                                       ((equal? sort "Ide") all-identifiers)
                                                       (else   (let ((new-symbols (make-fresh-constants sort (get-flag "dom-dt-
                                                                 (list->stream new-symbols))))))))
       [(lambda (sort)
          (try (table-lookup sort-stream-table sort)
               (all-ground-terms sort)))

        (lambda (sort d)
          ((all-bounded-ground-terms sort) d))

        all-bounded-ground-terms

        all-ground-terms

        process-irc

        process-rc
       ]))))


# Needed in rewriting.ath:

(set! make-all-terms-thunk-cell
   (lambda (datatype-domain)
     (stream-take (make-all-ground-terms datatype-domain) 100000)))

EOF


(define [make-all-ground-terms
         process-rc]
  (let ((ht1 (make-hash-table 23)))
    (let ((A1 (lambda (x) x))
          (A6 (lambda (sort)
                    (check ((datatype-sort? sort)
                                                 1)
                                               ((equal? sort "Int") all-positive-integers)
                                               ((equal? sort "Real") all-reals)
                                               ((equal? sort "Ide") all-identifiers)
                                               (else   (let ((limit (string->num (get-flag "dom-dt-default-size")))
                                                             (new-symbols (map (lambda (_) (make-fresh-constant sort))
                                                                               (from-to 1 limit))))
                                                         new-symbols))))))
       [A1 A6])))


(define make-all-ground-terms
  (let ((A6 (lambda (sort)
                (check ((datatype-sort? sort)
                                                 1)
                                               ((equal? sort "Int") all-positive-integers)
                                               ((equal? sort "Real") all-reals)
                                               ((equal? sort "Ide") all-identifiers)
                                               (else   (let ((limit (string->num (get-flag "dom-dt-default-size")))
                                                             (new-symbols (map (lambda (_) (make-fresh-constant sort))
                                                                               (from-to 1 limit))))
                                                         new-symbols))))))
```

```
209     A6))
210
211 (define make-all-ground-terms
212   (let ((A6
213    (lambda (sort)
214                 (check ((datatype-sort? sort)
215                                                    1)
216                                                ((equal? sort "Int") all-positive-integers)
217                                                ((equal? sort "Real") all-reals)
218                                                ((equal? sort "Ide") all-identifiers)
219                                                (else   (let ((limit (string->num (get-flag "dom-dt-default-size")))
220                                                              (new-symbols (map (lambda (_) (make-fresh-constant sort))
221                                                                                (from-to 1 limit))))
222                                                         new-symbols))))))
223     A6))
224
225 (define (make-all-ground-terms  sort)
226                 (check ((datatype-sort? sort)
227                                                    1)
228                                                ((equal? sort "Int") all-positive-integers)
229                                                ((equal? sort "Real") all-reals)
230                                                ((equal? sort "Ide") all-identifiers)
231                                                (else   (let ((limit (string->num (get-flag "dom-dt-default-size")))
232                                                              (new-symbols (map (lambda (_) (make-fresh-constant sort))
233                                                                                (from-to 1 limit))))
234                                                         new-symbols))))
235     A6))
```