lib/basic/msr.ath

```
## D, 1-21-2010: Made a small change in the distance procedure to make it work
## with Ide's. arity-of fails on an Ide, so there is now special case code
## that assigns 0 to equal Ide's, 1.0 to unequal ones, without calling arity-of.

## K, 12-30-2009: To avoid having msr.ath load up rewriting.ath, an extra parameter
## "rewrite*" has been added to the deductive versions of the searches: drs-bf,
## drs-df, and drs-rf. To get the previous versions, simply pass the rewrite*
## procedure defined in rewriting.ath as an argument.

## K, 12-28-2009: Added a breadth-first version: drs-rf. In general this is hopelessly
## inefficient, but in small cases it may be a better choice than best-first.

## K, 12-27-2009: Speeded up best-first search quite significantly by introducing
## a bit of randomness in the search. Lowered the maximum depths to 2000 for best-first
## and 3000 for depth-first search. Typically it takes about 40-50 seconds to reach these
## depths, so this amounts to setting a maximum search time of less than a minute.
## Best-first search is now overwhelmingly more efficient than depth-first search,
## and can handle all the examples in this file within the allowed search limit.

## K, 12-26-2009: A simple implementation of multiple-step rewriting by depth-first
## and best-first search. The functionality is implemented both as procedures and as
## methods. The procedures are called rewrite-search-df and rewrite-search-bf (for
## depth- and best-first, respectively). The methods are drs-df and drs-bf, respectively.
## A procedure call (rewrite-search-df s t equations) tries to transform s into
## t by applying various identities from the 'equations' list (each of which
## must be a universally quantified identity, possibly w/ zero quantifiers).
## It works by searching the state space of all derivations in a depth-first way.
## A call (rewrite-search-bf s t equations) works in the same way, except that
## best-first search is used. The score of a term is its distance from the target
## term t. The distance between two terms is computed by a simple linear-time
## metric measure from 0 to 1.

## If either procedure finds a derivation, it outputs one. A derivation consists
## of a list of pairs, each pair comprising a term s_i and an equation by virtue of
## which  s_i follows from s_{i-1}. To get silent output, redefine the procedure
## mprint. Both procedures only search up to a certain depth d, quitting when that
## depth is exceeded. By default, d is set to 1000 for best-first and 5000 for
## depth-first search, which seems to be enough to get most simple rewriting steps
## (e.g., by AC) in a fairly efficient way.
## Searches beyond these depths are too slow for proof checking. (The implementations
## are quite unoptimized, however, and there should be much room for
## performance improvement.)

## The method drs-df works simply by running rewrite-search-df to discover a
## derivation, and then transitively threading rewrite* through that derivation
## to get a proof of the desired identity. The method drs-bf works likewise,
## except that it uses rewrite-search-bf for the search.

## Best-first search almost always finds dramatically shorter derivations.
## However, it usually takes a bit longer than depth-first to reach the same
## depth because of the need to sort the fringe with every expansion.
## This sorting can be avoided by just sorting the expanded nodes and then
## simply merging with the existing fringe, but even merging can slow down
## the search. Overall, best-first appears to be the better choice, but
## more experience is needed.

## File msr-tests.ath defines a small test suite

(define mprint print)

(define max-depth-df 3000)
(define max-depth-bf 3000)
(define max-depth-bf 80000)
(define no-progress-limit (cell 30))
(define no-progress-limit (cell 80))

## Redefine mprint as follows to get silent output:

```

```
(define (mprint x) (print x))

(define (mprintln s)
  (mprint (join "\n" s "\n")))

(define (make-state term parent equation score sub)
  [term parent equation score sub])

(define state-term first)
(define state-parent second)
(define state-equation third)
(define state-score fourth)
(define state-sub fifth)

(define (state->string s)
  (val->string (state-term s)))

(define (show-state state msg)
  (seq (mprint (join msg "\n"))
       (mprint (join (state->string state) "\n"))
       (mprint (join "\nand its distance: " (val->string (state-score state)) "\n"))
       (mprint (join "\nand its equation:\n" (val->string (state-equation state)) "\n"))))


(define (has-parent? s)
  (match (state-parent s)
    ((some-list _) true)
    (_ false)))

(define (distance t1 t2)
  (match [t1 t2]
    ([((some-symbol f1) (some-list args1))
      ((some-symbol f2) (some-list args2))]
       (check ((equal? f1 f2)
                  (let ((arity (match t1 (x:Ide 0) (_ (arity-of f1)))))
                    (check ((equal? arity 0) 0)
                           (else (times (div 1.0 (times 2.0 arity))
                                        (distance* args1 args2 0.0))))))
              (else 1.0)))
    ([(some-var x) x] 0.0)
    (_ 1.0))
  (distance* terms1 terms2 sum)
    (match [terms1 terms2]
      ([[] []] sum)
      ([(list-of t1 rest1) (list-of t2 rest2)]
        (distance* rest1 rest2 (plus sum (distance t1 t2))))))


(define (dom t)
  (letrec ((loop (lambda (children i results)
                   (match children
                     ([] results)
                     ((list-of ith-child rest)
                       (let ((S (map (lambda (pos) (add i pos))
                                     (dom ith-child))))
                         (loop rest (plus i 1) (join S results))))))))
    (add [] (loop (children t) 1 []))))

(define (subterm t pos)
   (match pos
     ([] t)
     ((list-of i rest) (subterm (ith (children t) i) rest))))

(define (replace s pos t)
  (letrec ((loop (lambda (terms current i rest-pos results)
                   (match terms
                     ([] (rev results))
                     ((list-of arg more) (let ((res (check ((equal? current i) (replace arg rest-pos t))
                                                           (else arg))))
                                           (loop more (plus current 1) i rest-pos (add res results))))))))
```

```
  (match [pos s]
    ([[] _] t)
    ([(list-of i rest) ((some-symbol f) (some-list args))]
      (let ((results (loop args 1 i rest [])))
        (make-term f results))))))

(define (map-select f L pred)
  (match L
    ([] [])
    ((list-of x rest) (let ((res (f x)))
                        (check ((pred res) (add res (map-select f rest pred)))
                               (else (map-select f rest pred)))))))

(define (get-rewrite s equation)
  (match equation
    ((forall (some-list vars) body)
      (match (match-terms s (lhs body))
        ((some-sub sub) [(sub (rhs body)) equation])
        (_ ())))
    (_ ())))

(define (ugen-vars s uvars)
  (filter (vars s) (lambda (v) (member? v uvars))))

(define (var-condition uvars left right)
  (let ((left-uvars   (ugen-vars left uvars))
        (right-uvars  (ugen-vars right uvars))
        (uvar? (lambda (v) (member? v uvars)))
        (cond1 (subset? left-uvars right-uvars))
        (cond2 (negate (uvar? right)))
        (cond3 (|| (leq? (term-size left) (term-size right)))))
    (&& cond1 cond2 cond3)))

(define (match-with-antecedent s equation uvars antecedent left right)
  (match (match-terms s left uvars)
    ((some-sub sub) (check ((all-components-hold (sub antecedent)) [(sub right) equation])
                           (else ())))
    (_ (match (match-terms s right uvars)
        ((some-sub sub) (check ((&& (var-condition uvars left right) (all-components-hold antecedent))
                                [(sub left) equation])
                               (else ())))
            (_ ())))))

(define (match-with-antecedent s equation uvars antecedent left right)
  (match (match-terms s left uvars)
    ((some-sub sub) (let ((ant-uvars (filter (vars antecedent) (lambda (v) (member? v uvars))))
                          (antecedent' (sub antecedent))
                          (var-eq (lambda (v1 v2)
                                    (equal? (var->string v1) (var->string v2))))
                          ([sub' new?] (check ((subset-eq? ant-uvars (supp sub) var-eq) (let ((_ (mprint "\nsubset, no
                                                       (else (find-list-element (ab)
                                                               (lambda (p) (negate (equal? (match-props-3 p antecedent' uvars)
                                                               (lambda (p) [(match-props-3 p  antecedent' uvars) true])
                                                               (lambda ()  [() false])))))))
                      (match sub'
                        ((some-sub _) (let ((sub'' (check (new? (compose-subs sub' sub)) (else sub'))))
                                        (check ((all-components-hold (sub'' antecedent)) [(sub'' right) equation sub''
                                               (else ())))))
                        (_ ()))))
    (_ (match (match-terms s right uvars)
        ((some-sub sub) (check ((&& (var-condition uvars left right) (all-components-hold antecedent))
                                [(sub left) equation sub])
                               (else ())))
            (_ ())))))



(define (get-rewrite s equation)
  (match equation
    ((forall (some-list uvars) (= (some-term left) (some-term right)))
```

```
209       (match (match-terms s left uvars)
210         ((some-sub sub) [(sub right) equation sub])
211         (_ (match (match-terms s right uvars)
212              ((some-sub sub) (check ((var-condition uvars left right)
213                                      [(sub left) equation sub])
214                                    (else ())))
215              (_ ()))))))
216     ((forall (some-list uvars) (if antecedent (= (some-term left) (some-term right))))
217        (match-with-antecedent s equation uvars antecedent left right))
218
219     ((forall (some-list uvars) (iff antecedent (= (some-term left) (some-term right))))
220        (match-with-antecedent s equation uvars antecedent left right))
221     ((forall (some-list uvars) (iff (= (some-term left) (some-term right)) antecedent))
222        (match-with-antecedent s equation uvars antecedent left right))
223     (_ ())))
224
225 (define (get-all-rewrites s equations)
226   (let ((positions (dom s))
227         (do-pos (lambda (pos)
228                   (let ((s/pos (subterm s pos))
229                         (results (map-select (lambda (eqn) (get-rewrite s/pos eqn)) equations (lambda (x) (negate (equa
230                     (map (lambda (result)
231                            (match result
232                              ([(some-term t) (some-sentence eqn) (some-sub sub)] [(replace s pos t) eqn sub])))
233                          results)))))
234    (fold join (map do-pos positions) [])))
235
236 (define (multiple? n k) (equal? (mod n k) 0))
237
238 (define (no-progress? np-count)
239   (greater? np-count (ref no-progress-limit)))
240
241 (define (make-search-procedure pre-process expand goal-state? equal-states? already-seen? state-less? max style)
242   (let (
243         (random-int (lambda (x) (let ((lim (check ((greater? x 10) 10) (else 0)))) (random-int (minus x lim) x))))
244         (fringe-table (make-term-hash-table 16007))
245 #        (fringe-table (make-term-hash-table))
246         (closed-table (make-term-hash-table 16007))
247 #        (closed-table (make-term-hash-table))
248         (enter-state (lambda (state table)
249                        (term-enter table (state-term state) state)))
250         (enter-states (lambda (states table)
251                         (map (lambda (state) (enter-state state table))
252                              states))))
253    (letrec ((search (lambda (fringe fringe-table closed closed-table count last-score np-count)
254                       (match fringe
255                         ([] 'failed)
256                         ((list-of first-state more-states)
257                           (seq (term-table-remove fringe-table (state-term first-state))
258                                (pre-process first-state more-states count)
259                                (check
260                                  ((goal-state? first-state) (seq (mprintln (join "Success. Count: " (val->string count
261                                                                  [first-state count]))
262                                  ((less? max count) (seq (mprintln "Exceeded max iterations") 'failed))
263                                  ((no-progress? np-count) (seq (mprintln "Search is getting stuck, aborting...") 'faile
264                                  (else (let ((new-score (state-score first-state))
265                                              (np-count' (check ((less? new-score last-score) 0)
266                                                                (else (plus np-count 1))))
267                                              (new-states (expand first-state))
268                                              (new-states' (filter new-states
269                                                                   (lambda (s)
270                                                                     (&& (negate (already-seen? s closed-table))
271                                                                         (negate (already-seen? s fringe-table))))))
272                                              (ns-count (length new-states'))
273                                              (_ (enter-states new-states' fringe-table))
274                                              (_ (enter-state first-state closed-table))
275                                              (_ (mprint (join "\nFiltered out "
276                                                               (val->string (minus (length new-states) ns-count))
277                                                               " states.\n Formed " (val->string ns-count) " new states
278                                              ([depth-first? breadth-first? best-first?]
```

```
279                                                  [(equal? style 'depth-first) (equal? style 'breadth-first) (equal? style
280                                                ([sort merge] (check (depth-first?  [(lambda (states) states)
281                                                                                    (lambda (x y z) (join x y))])
282                                                                     (best-first? [(lambda (states) (sort states state-le
283                                                                     (else [(lambda (states) states) ()]))))
284                                                (sorted-new-states (sort new-states'))
285                                                (fringe' (check (breadth-first? (join more-states sorted-new-states))
286                                                                (else (check ((multiple? count 10)
287                                                                              (let ((L (join sorted-new-states more-sta
288                                                                                     (L-size (length L))
289                                                                                     (mid (check ((greater? L-size 1) (d
290                                                                                                  (else 1)))
291                                                                                     ([x rest] (decompose-nth L (random-
292                                                                                (add x rest)))
293                                                                              (else (merge sorted-new-states more-states
294                                                  (search fringe' fringe-table
295                                                          (add first-state closed) closed-table (plus count 1) new-score np-cou
296      (lambda (init-state)
297        (let ((_ (enter-state init-state fringe-table)))
298          (search [init-state] fringe-table [] closed-table 1 (state-score init-state) 0))))))
299
300 (define (get-chain final-state)
301   (letrec ((loop (lambda (s results)
302                    (check ((has-parent? s) (loop (state-parent s) (add [(state-term s) (state-equation s) (state-sub s
303                           (else (add (state-term s) results))))))
304     (let ((results (loop final-state []))
305           (show-triple (lambda (r)
306                          (match r
307                            ((some-term _) (mprintln (val->string r)))
308                            ([s eqn sub] (mprint (join "-------->\n" (val->string s) "\nby:\n" (val->string eqn) "\n"
309                                                       " and sub: " (val->string sub) "\n"))))))
310           (_ (map show-triple results))
311           (_ (mprintln (join "\nNumber of steps in this derivation: " (val->string (minus (length results) 1))))))
312       results)))
313
314
315 (define (rewrite-search s t equations df?)
316   (let     (
317            (_ (mprintln (join "\nNumber of equations: " (val->string (length equations)))))
318            (term-table (make-term-hash-table))
319            (expand (lambda (state)
320                      (let ((term (state-term state))
321                            (new-terms-equations-subs (get-all-rewrites term equations))
322                            (ms (lambda (t-e-sub)
323                                  (match t-e-sub
324                                    ([term eqn sub] (make-state term state eqn (distance term t) sub))))))
325                        (map ms new-terms-equations-subs))))
326            (pre-process (lambda (first-state more-states count)
327                           (seq (mprintln (join "\nCount: " (val->string count)))
328                                (mprint (join "\nCurrent fringe size: " (val->string (plus 1 (length more-states)))))
329                                (show-state first-state (join "\nCurrent state (state #" (val->string count) "):")))))
330            (goal-state? (lambda (state)
331                           (equal? (state-term state) t)))
332            (equal-states? (lambda (s1 s2)
333                             (equal? (state-term s1) (state-term s2))))
334            (state-less? (lambda (s1 s2)
335                           (let ((score1 (state-score s1))
336                                 (score2 (state-score s2)))
337                             (check ((&& (equal? score1 1.0) (equal? score2 1.0))
338                                     (less? (term-size (state-term s1)) (term-size (state-term s2))))
339                                    (else (less? score1 score2))))))
340            (already-seen? (lambda (s term-table)
341                             (negate (equal? (term-look-up term-table (state-term s)) ()))))
342            (limit (check (df?  max-depth-df)
343                          (else max-depth-bf)))
344            (search (make-search-procedure pre-process expand goal-state? equal-states? already-seen? state-less? limit
345            (init-state (make-state s () () (distance s t) empty-sub))
346            (search-result (search init-state)))
347     (match search-result
348       ([some-state count] [(get-chain some-state) count])
```

```
      (_ 'failed))))


(define (rewrite-search s t equations df?)
  (let    ((term-table (make-term-hash-table))
           (expand (lambda (state)
                     (let ((term (state-term state))
                           (new-terms-equations-subs (get-all-rewrites term equations))
                           (ms (lambda (t-e-sub)
                                 (match t-e-sub
                                   ([term eqn sub] (make-state term state eqn (distance term t) sub))))))
                       (map ms new-terms-equations-subs))))
           (pre-process (lambda (first-state more-states count) ()))
           (goal-state? (lambda (state)
                          (equal? (state-term state) t)))
           (equal-states? (lambda (s1 s2)
                            (equal? (state-term s1) (state-term s2))))
           (state-less? (lambda (s1 s2)
                          (let ((score1 (state-score s1))
                                (score2 (state-score s2)))
                            (check ((&& (equal? score1 1.0) (equal? score2 1.0))
                                    (less? (term-size (state-term s1)) (term-size (state-term s2))))
                                   (else (less? score1 score2))))))
           (already-seen? (lambda (s term-table)
                            (negate (equal? (term-look-up term-table (state-term s)) ()))))
           (limit (check (df?  max-depth-df)
                         (else max-depth-bf)))
           (search (make-search-procedure pre-process expand goal-state? equal-states? already-seen? state-less? limit
           (init-state (make-state s () () (distance s t) empty-sub))
           (search-result (search init-state)))
    (match search-result
      ([some-state count] [(get-chain some-state) count])
      (_ 'failed))))

(define (rewrite-search-df s t equations)
   (rewrite-search s t equations 'depth-first))

(define (rewrite-search-rf s t equations)
   (rewrite-search s t equations 'breadth-first))

(define (rewrite-search-bf s t equations)
   (rewrite-search s t equations 'best-first))

(define rs rewrite-search)

## New Additions in terms of SML:

(define (rewrite-search s t eqns style)
  (sml-rewrite-search s t eqns (id->string style) (plus max-depth-bf 18000) "silent"))

(define rs rewrite-search)

(define (drs s t equations rewrite* style)
    (dletrec ((loop (method (current-identity results)
                      (dmatch results
                        ((list-of [(some-term s) (some-sent eqn) (some-sub sub)] (some-list rest))
                           (dlet ((theorem (!rewrite* (rhs current-identity) s eqn sub))
                                  (new-identity (!tran current-identity theorem)))
                             (!loop new-identity rest)))
                        ([] (!claim current-identity))))))
  (dmatch (rewrite-search s t equations style)
    ((derivation as [(list-of (some-term first-term) rest) (some-term count)])
      (dlet (#(_ (print "\nDERIVATION FOUND: " derivation))
             (seed  (!reflex first-term))
             #(_ (print "\nSeed 1: " seed))
              (theorem (!loop seed rest))
             #(_ (print "\nTHEOREM: " theorem))
             #(_ (mprintln (join "Depth reached: " (val->string count))))
            )
       (!claim theorem)))
```

```
419     ((as res (list-of (list-of first-term _) rest))
420       (dlet (#(_ (print "\nDERIVATION 2 FOUND: " res))
421              (seed  (!reflex first-term))
422              #(_ (print "\nSeed 2: " seed))
423               (theorem (!loop seed rest))
424              #(_ (print "\nTHEOREM: " theorem))
425              )
426        (!claim theorem))))))
427
428 (define (drs-df s t equations rewrite*)
429   (!drs s t equations rewrite* 'depth-first))
430
431 (define (drs-bf s t equations rewrite*)
432   (!drs s t equations rewrite* 'best-first))
433
434 (define (drs-rf s t equations rewrite*)
435   (!drs s t equations rewrite* 'breadth-first))
436
437
438 (define (find-eqn-proof s t eqns)
439    (rewrite-search s t eqns 'best-first))
```