lib/basic/msr-tests.ath

```
(load-file "msr.ath")
(load-file "rewriting.ath")

(define ars rewrite-search)
(define gr get-all-rewrites)

(define R1 (commutative +))
(define R2 (associative +))
(define R3 (commutative *))
(define R4 (associative *))

(define rules [R1 R2 R3 R4])
(assert rules)

(define (test-df s t)
  (let ((t1 (time))
        (_ (!drs-df s t rules rewrite-one-redex))
        (t2 (time))
        (time-spent (minus t2 t1)))
    (println (join "Time: " (val->string time-spent) " seconds"))))

(define (test-rf s t)
  (let ((t1 (time))
        (_ (!drs-rf s t rules rewrite-one-redex))
        (t2 (time))
        (time-spent (minus t2 t1)))
    (println (join "Time: " (val->string time-spent) " seconds"))))

(define (test-bf s t)
  (let ((t1 (time))
        (_ (!drs-bf s t rules rewrite-one-redex))
        (t2 (time))
        (time-spent (minus t2 t1)))
    (println (join "Time: " (val->string time-spent) " seconds"))))


(define (test-pairs L)
  (let ((failures (cell 0))
        (sum (cell 0))
        (pair-num (length L))
        (_ (print "\nHave " pair-num " equations to test...\n"))
        (index (cell 0))
        (test-bf (lambda (s t)
                   (let ((t1 (time))
                         (_ (set! index (plus (ref index) 1)))
                         (res (dtry (!drs-bf s t rules rewrite-one-redex) (!true-intro)))
                         (t2 (time))
                         (time-spent (minus t2 t1)))
                    (match res
                      (true (seq (print "\nFAILED test #" (ref index) "\n")
                                 (set! failures (plus (ref failures) 1))))
                      (_ (seq (print "\nFinished test #" (ref index) " in " time-spent " seconds...\n")
                              (set! sum (plus (ref sum) time-spent))))))))
        (_ (map-proc (lambda (pair) (match pair ([s t] (test-bf s t)))) L))
        (successes (minus pair-num (ref failures)))
        (av-time (div (ref sum) successes)))
    (print "\nManaged all except for " (ref failures) " tests.\nAverage proof time: " av-time "\n")))


(define (dtest-pairs L)
  (let ((failures (cell 0))
        (sum (cell 0))
        (pair-num (length L))
        (_ (print "\nHave " pair-num " equations to test...\n"))
        (index (cell 0))
        (test-bf (lambda (s t)
                   (let ((t1 (time))
                         (_ (set! index (plus (ref index) 1)))
                         (res (dtry (!drs-df s t rules rewrite-one-redex) (!true-intro)))
                         (t2 (time))
                         (time-spent (minus t2 t1)))
                    (match res
                      (true (seq (print "\nFAILED test #" (ref index) "\n")
                                 (set! failures (plus (ref failures) 1))))
                      (_ (seq (print "\nFinished test #" (ref index) " in " time-spent " seconds...\n")
                              (set! sum (plus (ref sum) time-spent)))))))
        (_ (map-proc (lambda (pair) (match pair ([s t] (test-bf s t)))) L))
        (successes (minus pair-num (ref failures)))
        (av-time (div (ref sum) successes)))
    (print "\nManaged all except for " (ref failures) " tests.\nAverage proof time: " av-time "\n")))

(define (btest-pairs L)
  (let ((failures (cell 0))
        (sum (cell 0))
        (pair-num (length L))
        (_ (print "\nHave " pair-num " equations to test...\n"))
        (index (cell 0))
        (test-bf (lambda (s t)
                   (let ((t1 (time))
                         (_ (set! index (plus (ref index) 1)))
                         (res (dtry (!drs-rf s t rules rewrite-one-redex) (!true-intro)))
                         (t2 (time))
                         (time-spent (minus t2 t1)))
                    (match res
                      (true (seq (print "\nFAILED test #" (ref index) "\n")
                                 (set! failures (plus (ref failures) 1))))
                      (_ (seq (print "\nFinished test #" (ref index) " in " time-spent " seconds...\n")
                              (set! sum (plus (ref sum) time-spent)))))))
        (_ (map-proc (lambda (pair) (match pair ([s t] (test-bf s t)))) L))
        (successes (minus pair-num (ref failures)))
        (av-time (div (ref sum) successes)))
    (print "\nManaged all except for " (ref failures) " tests.\nAverage proof time: " av-time "\n")))


(define (test s t)
  (let ((res (ars s t [R1 R2] "depth-first" 50000 "silent")))
     res))

##########################################################################################
#                                    Some tests:
##########################################################################################

(define s1 (+ ?a (+ ?b ?c)))
(define t1 (+ ?a (+ ?c ?b)))
(define t2 (+ (+ ?b ?c) ?a))
(define t3 (+ (+ ?a ?b) ?c))
(define t4 (+ (+ ?b ?a) ?c))
(define t5 (+ ?c (+ ?b ?a)))

(rewrite-search s1 t1 rules 'best-first)

(define t6 (+ (+ ?c ?a) ?b))


# (gr s1 rules)

# (load-file "msr-tests.ath")



(test-bf s1 t1)
(test-bf s1 t1)

(test-bf s1 t2)


(test-bf s1 t3)

```

```

(test-bf s1 t4)

(test-bf s1 t5)

(test-bf s1 t5)


(define A (+ ?g (+ ?e (+ (+ ?a ?b) (+ ?d ?c)))))
(define B (+ (+ ?c (+ ?d (+ ?e (+ ?b ?a)))) ?g))
(define C (+ ?a (+ ?b (+ ?c (+ ?d (+ ?e ?g))))))
(define D
  (+ ?g
     (+ ?a
        (+ ?e
           (+ (+ ?c ?b)
              ?d)))))

(define E (+ (+ (+ ?g (+ ?c ?e)) (+ ?b ?a)) ?d))


(define (test-bf a b) ())

(test-bf A B)
(test-bf B A)

(test-bf B C)
(test-bf C B)

(test-bf A C)

(test-bf C A)

(test-bf A D)
(test-bf D A)

(test-bf B D)
(test-bf D B)

(test-bf C D)
(test-bf D C)


(define A1 (+ ?e (+ (+ ?a ?b) (+ ?d ?c))))
(define B1 (+ ?c (+ ?d (+ ?e (+ ?b ?a)))))

(test-bf A1 B1)
(test-bf B1 A1)


# Best-first seach with some randomness succeeds in interconverting any two of
# the terms A, B, C, and D below (on the basis of the comm/assoc properties of + and *).
# Depth-first search fails on all of them with a max depth of 4000:

(test-bf B C)
(test-bf C B)

(test-bf A C)
(test-bf A C)

(test-bf A D)
(test-bf D A)

(test-bf B D)
(test-bf D B)

(test-bf C D)
(test-bf D C)

(test-bf D A)
```

```
(test-bf D A)

(test-bf B D)
(test-bf D B)

(test-bf D C)

(test-bf A E)
(test-bf E A)

(test-bf B E)
(test-bf E B)
(test-bf C E)
(test-bf E C)


(define E1 (+ ?c (+ ?b (+ ?a (+ ?d (+ ?g ?e))))))

(test-bf E E1)
(test-bf E1 E)
(test-bf E1 A)
(test-bf E1 C)
(test-bf D E1)


(distance D C)
(distance E1 C)

(define a (+ (+ ?a ?b) (+ ?c ?d)))
(define b1 (+ ?d (+ ?c (+ ?b ?a))))
(define b2 (+ ?c (+ ?d (+ ?b ?a))))
(define b3 (+ ?b (+ ?d (+ ?c ?a))))
(define b4 (+ (+ (+ ?d ?c) ?b) ?a))
(define b5 (+ (+ (+ ?d ?c) ?a) ?a))
(define start1 (+ (+ (+ ?a ?b) ?c) (+ ?d ?e)))
(define finish1 (+ ?e (+ ?d (+ ?c (+ ?b ?a)))))
(define start2 (+ ?foo (+ (+ (+ ?a ?b) ?c) (+ ?d ?e))))
(define finish2 (+ (+ ?e (+ ?d (+ ?c (+ ?b ?a)))) ?foo))
(define finish2a (+ ?e (+ ?d (+ ?c (+ ?b (+ ?foo ?a))))))

(define start3 (+ (+ ?foo ?goo) (+ (+ (+ ?a ?b) ?c) (+ ?d ?e))))
(define finish3 (+ (+ ?e (+ ?d (+ ?c (+ ?b ?a)))) (+ ?goo ?foo)))
(define finish4 (+ ?e (+ ?d (+ ?c (+ ?b (+ ?goo (+ ?a ?foo)))))))

(test-bf a b1)
(test-bf a b1)
(test-bf a b2)
(test-bf a b3)
(test-bf a b4)

(test-bf start1 finish1)
(test-bf finish1  start1)

(test-bf start2 finish2)
(test-bf finish2 start2)

(test-bf start2 finish2a)
(test-bf finish2a start2)

(test-bf start3 finish3)
(test-bf finish3 start3)

(test-bf start3 finish4)

(test-bf finish4 start3)

(define X1 (+ finish4 start3))
(define Y1 (+ start3 finish3))
(define X2 (+ start2 start3))
(define Y2 (+ finish3 finish2))
```

```

(test-bf X1 Y1)

(test-bf X2 Y2)



(define s (+ (+ finish4 start3) (+ start2 start3)))
(define f (+ (+ start3 finish3) (+ finish3 finish2)))
(test-bf s f)

(define S1 (+ (* ?a ?b)
              (+ (+ ?A (* ?B ?C))
                 (* ?e (* ?d ?f)))))

(define S2 (+ (+ (* (* ?f ?d) ?e) (+ ?A (* ?C ?B)))
              (* ?b ?a)))

(define S3 (+ (+ (+ (* ?C ?B) ?A)
                 (* (* ?d ?e) ?f))
              (* ?b ?a)))


(define pairs [[A B] [B A] [B C] [C B] [A C]  [C A] [A D] [D A] [B D] [D B] [C D] [D C] [A E] [E A] [B E] [E B] [C E]
               [E E1] [E1 E] [E1 A] [E1 C] [D E1]
               [a b1] [a b2] [a b3] [a b4] [start1 finish1] [finish1 start1]  [start2 finish2] [finish2 start2]
               [start2 finish2a] [finish2a start2] [start3 finish3] [finish3 start3] [start3 finish4] [finish4 start3]
               [X1 Y1] [X2 Y2] [S1 S2] [S2 S1] [S1 S3] [S3 S1] [S2 S3] [S3 S2]])

# Can't do [s f]



(define (t) (test-pairs pairs))
EOF
(load-file "msr-tests.ath")

(test-pairs pairs)

(dtest-pairs pairs)
(btest-pairs pairs)

(test-bf S1 S2)

(test-bf S2 S1)
(test-bf S2 S1)

(test-bf S1 S3)
(test-bf S3 S1)

(test-bf S2 S3)
(test-bf S3 S2)

(define S4 (+ (+ (+ (+ ?A1 (+ ?A2 (* ?C ?B))) ?A)
                 (* (* ?d ?e) ?f))
              (* ?b ?a)))


(define S5 (+ (+ (* (* ?f ?d) ?e) (+ (+ ?A1 (+ ?A2 (* ?C ?B))) ?A))
              (* ?b ?a)))

(test-bf S4 S5)
(test-bf S5 S4)

(define S6 (+ S4 S5))
(define S7 (+ S5 S4))

(test-bf S6 S7)
(test-bf S7 S6)

```

```
(define S8 (+ S6 (+ S5 S4)))
(define S9 (+ (+ S4 S5) S6))

(test-bf S8 S9)

################################################################################

(load-file "rewriting.ath")

(domain Z)
(datatype Nat zero (succ Nat))
(declare Minus (-> (Nat Nat) Nat))
(declare Zzero Z)
(declare neg (-> (Nat) Z))

(define p1 (forall ?x:Nat (= (Minus ?x ?x) zero)))
(define p2 (= Zzero (neg zero)))
(assert p1 p2)

(define a ?a:Nat)
(define s (neg (Minus a a)))
(define t Zzero)

(define rules [p1 p2])

(!drs-bf s t rules rewrite-one-redex)

################################################################################
## D: Added 1/21/2009 to document an apparent bug:

# Uncomment the following if starting here:
#

(load-file "rewriting.ath")
(datatype Nat zero (succ Nat))

(domain E)

(declare T (-> (E) Nat))

(declare (s1 s2) E)

(declare t Nat)

(define p1 ((T s1) = t))

(define p2 ((T s2) = (succ t)))

(assert p1 p2)

# This fails

(!drs-bf (T s2) (succ (T s1)) [p1 p2] rewrite-one-redex)

# This works
(!drs-bf (succ (T s1)) (T s2) [p1 p2] rewrite-one-redex)

# This works:
(!chain [(T s2)
         = (succ t)            [p2]
         = (succ (T s1))       [p1]])

# So does this, since chain tries both calls of drs-bf:

(!chain [(T s2)
         = (succ (T s1))       [p1 p2]])

# But if we do essentially the same thing with a binary symbol T1
# instead of T, it fails:

```

```
(domain C)

(declare T1 (-> (C E) Nat))

(declare C1 C)

(declare (x1 x2) E)

(declare t1 Nat)

(define p1 ((T1 C1 x1) = t1))

(define p2 ((T1 C1 x2) = (succ t1)))

(assert p1 p2)

# This fails:
(!drs-bf (T1 C1 x2) (succ (T1 C1 x1)) [p1 p2] rewrite-one-redex)

# and so does this:
(!drs-bf (succ (T1 C1 x1)) (T1 C1 x2) [p1 p2] rewrite-one-redex)

# So, although this works:
(!chain [(T1 C1 x2)
         = (succ t1)          [p2]
         = (succ (T1 C1 x1)) [p1]])

# the following fails (calls external theorem prover)
# since both drs-bf calls fail:

(!chain [(T1 C1 x2)
         = (succ (T1 C1 x1)) [p1 p2]])

# This works:

(define p3 (!sym p1))

(!drs-bf (T1 C1 x2) (succ (T1 C1 x1)) [p3 p2] rewrite-one-redex)

# hence this does too:
(!chain [(T1 C1 x2)
         = (succ (T1 C1 x1)) [p3 p2]])
```