lib/basic/eval.ath

```
## This file defines two normal-form evaluators for
## ground datatype terms. One of them, eval, is computational,
## i.e., a procedure. The other, deval, is a method.
## Both are rather slow, but the deductive version is
## even slower.

(load-file "util.ath")

(define rule-table (make-symbol-hash-table))

(define (equational-rule? p)
  (match p
    ((forall (some-list _) (= _ _)) true)
    (_ false)))

(define (cond-rule? p)
  (match p
    ((forall (some-list vars) (if _ _)) true)
    (_ false)))

(define (orient-bc p f)
  (match p
    ((forall (some-list vars)
       (iff ant (bind conclusion (= ((val-of f) (some-list args)) right))))
         (match ant
           ((= ((val-of f) (some-list args')) right')
             (let ((syms  (get-term-syms* args))
                   (syms' (get-term-syms* args')))
               (check ((&& (for-each syms constructor?)
                           (|| (negate (null? syms)) (negate (for-each syms' constructor?))))
                       (forall* vars (= ant conclusion)))
                      ((for-each syms' constructor?)
                        (forall* vars (if conclusion ant)))
                      (else ())))))
           (_ (forall* vars (if ant conclusion)))))
       (_ ())))


(define (get-rule f p)
 (match p
  ((forall (some-list vars) (= ((val-of f) (some-list args)) right))
    (let ((left-syms (get-term-syms* args)))
      (check ((&& (for-each left-syms constructor?)
                  (|| (negate (null? left-syms)) (negate (member? f (get-term-syms right))))) [p])
             (else []))))
  ((forall (some-list vars) (if ant (bind conclusion (= ((val-of f) (some-list args)) right))))
    (let ((left-syms (get-term-syms* args)))
      (check ((for-each left-syms constructor?) [(forall* vars (if ant conclusion))])
             (else []))))
  ((forall (some-list vars) (iff ant (= ((val-of f) (some-list args)) right)))
     [(orient-bc p f)])
  ((forall (some-list vars) (iff (bind conclusion (= ((val-of f) (some-list args)) right)) cond))
     [(orient-bc (forall* vars (iff cond conclusion)) f)])
  ((forall (some-list vars) ((val-of f) (some-list args)))
    [(forall* vars (= (make-term f args) true))])
  ((forall (some-list vars) (not ((val-of f) (some-list args))))
    [(forall* vars (= (make-term f args) false))])
  ((forall (some-list vars) (if ant ((val-of f) (some-list args))))
    (get-rule f (forall* vars (if ant (= (make-term f args) true)))))
  ((forall (some-list vars) (iff (bind left ((val-of f) (some-list _)))
                                 (bind right ((val-of f) (some-list _)))))
    (get-rule f (forall* vars (= left right))))
  ((forall (some-list vars) (iff ant ((val-of f) (some-list args))))
    (join (get-rule f (forall* vars (if ant (= (make-term f args) true))))
          (get-rule f (forall* vars (if (not ant) (= (make-term f args) false))))))
  ((forall (some-list vars) (iff ((val-of f) (some-list args)) conclusion))
    (let ((p1 (forall* vars (iff (= (make-term f args) true) conclusion)))
          (p2 (forall* vars (if (not conclusion) (= (make-term f args) false)))))
```

```
69        (join (get-rule f p1) (get-rule f p2))))
70    ((forall (some-list vars) (and (some-list props)))
71          (flatten (map (lambda (p) (get-rule f (forall* vars p))) props)))
72    ((and (some-list props)) (flatten (map (lambda (p) (get-rule f p)) props)))
73    (_ [])))
74
75
76  (define (get-rules f)
77    (let ((wv (lambda (x y) ()))
78          (all-rules (fold join (map (lambda (p) (check ((assertion? p) (let ((_ (wv "p: " p))
79                                                                              (res (get-rule f p))
80                                                                              (_ (wv "res: " res)))
81                                                                          res))
82                                                       (else [])))  (ab)) [])))
83          ([equational-rules rest] (filter-and-complement all-rules equational-rule?))
84          (cond-rules (filter rest cond-rule?)))
85      [equational-rules cond-rules]))
86
87
88  (define (ugen-vars s uvars)
89    (filter (vars s) (lambda (v) (member? v uvars))))
90
91  (define (var-condition uvars left right)
92    (let ((left-uvars   (ugen-vars left uvars))
93          (right-uvars  (ugen-vars right uvars))
94          (cond1 (subset? right-uvars left-uvars))
95          (uvar? (lambda (v) (member? v uvars)))
96          (cond2 (negate (uvar? left))))
97      (&& cond1 cond2)))
98
99  (define (find-matching-rule rules t)
100   (match rules
101     ([] ())
102     ((list-of (bind R (forall (some-list _vars) (= left right))) rest)
103       (match [(match-terms t left) (var-condition _vars left right)]
104         ([(some-sub sub) true] [R sub rest])
105         (_ (find-matching-rule rest t))))
106     ((list-of (bind R (forall (some-list _vars) (if _ (= left right)))) rest)
107       (match [(match-terms t left) (var-condition _vars left right)]
108         ([(some-sub sub) true] [R sub rest])
109         (_ (find-matching-rule rest t))))))
110
111 (define (lookup-rules f)
112   (match (look-up-symbol rule-table f)
113     ((some-list L) L)
114     (_ (let ((L (get-rules f))
115              (_ (enter-symbol rule-table f L)))
116          L))))
117
118
119 (define (update-rules f L)
120   (match L
121     ([eq-rules cond-rules]
122       (match (look-up-symbol rule-table f)
123         ([old-eq-rules old-cond-rules]
124           (let ((eq-test (lambda (R) (negate (member? R old-eq-rules))))
125                 (cond-test (lambda (R) (negate (member? R old-cond-rules))))
126                 (new-eq-rules (join old-eq-rules (filter eq-rules eq-test)))
127                 (new-cond-rules (join old-cond-rules (filter cond-rules cond-test))))
128             (enter-symbol rule-table f [new-eq-rules new-cond-rules])))
129         (_ (enter-symbol rule-table f L))))
130     (_ ())))
131
132
133 (define (built-in-constant? f args)
134   (&& (null? args) (|| (equal? (sort-of f) "Int")
135                        (equal? (sort-of f) "Real")
136                        (equal? (sort-of f) "Ide"))))
137
138
```

```
(define (numeric? f)
   (member? f [+ - * /]))

(define (apply-numeric f args)
  (match [f args]
    ([+ [t1 t2]] (plus t1 t2))
    ([+ [t]] t)
    ([- [t1 t2]] (minus t1 t2))
    ([* [t1 t2]] (times t1 t2))
    ([/ [t1 t2]] (div t1 t2))
    ([- [t]] (- t))))

(define (eval1 t)
    (match t
      (((some-symbol f) (some-list args))
        (check ((constructor? f) (make-term f (map eval1 args)))
               ((equal? f =) (equal? (eval1 (first args)) (eval1 (second args))))
               ((numeric? f) (apply-numeric f (map eval1 args)))
               (else (let ((args' (map eval1 args))
                           (t' (make-term f args'))
                           ([eq-rules cond-rules] (lookup-rules f))
                           (error-msg (join "Error: No matching rule found for the term\n" (val->string t))))
                       (try-rules t' (join eq-rules cond-rules)
                                (lambda () (check ((built-in-constant? f args) t)
                                                  (else (halt)))))))))
     ((not arg) (negate (eval1 arg)))
     ((and (some-list args)) (fold (lambda (b1 b2) (&& b1 b2)) (map eval1 args) true))
     ((or (some-list args)) (fold (lambda (b1 b2) (|| b1 b2)) (map eval1 args) false))
     ((if arg1 arg2) (|| (negate (eval1 arg1)) (eval1 arg2)))
     ((some-var _) t))
  (try-rules s rules K)
                        (match (find-matching-rule rules s)
                          ([(forall (some-list _) (= _ right)) sub rest] (try (eval1 (sub right))
                                                                              (try-rules s rest K)))
                          ([(forall (some-list _) (if p (= _ right))) sub rest]
                            (try (let (([res sub'] (eval-cond p sub)))
                                   (check (res (eval1 (sub' (sub right))))
                                          (else (try-rules s rest K))))
                                 (try-rules s rest K)))
                          (_ (K)))
 (loop conditions sub type)
   (match conditions
     ([] (match type
           ('conjunction [true sub])
           (_ [false sub])))
     ((list-of cond more) (let (([res sub'] (eval-cond cond sub))
                                (sub''  (compose-subs sub' sub)))
                            (match [type res]
                              (['conjunction true]  (loop more sub'' type))
                              (['disjunction false] (loop more sub'' type))
                              (['conjunction false] [false sub''])
                              (['disjunction true]  [true sub'']))))))
 (eval-cond cond sub)
   (match cond
     ((= left right) (let ((left'  (eval1 (sub left)))
                           (right' (eval1 (sub right))))
                       (match (match-terms left' right')
                         ((some-sub sub') [true (compose-subs sub' sub)])
                         (_ [false sub])))))
     (((some-symbol R) (some-list terms)) [(eval1 (sub cond)) sub])
     ((not p) (match (eval-cond p sub)
                ([b sub'] [(negate b) sub'])))
     ((and (some-list args)) (loop args sub      'conjunction))
     ((or (some-list args))  (loop args sub      'disjunction))
     ((if arg1 arg2) (eval-cond (or (not arg1) arg2) sub))))


(define (eval2 t)
  (let ((t' (eval1 t)))
    (match t
```

```
209       ((some-term _) (rhs (= t t')))
210       (_ t'))))
211
212 (define (decimal n)
213   (match n
214     (zero 0)
215     ((succ k) (plus (decimal k) 1))))
216
217 (define (make-unary k)
218   (check ((equal? k 0) zero)
219          (else (succ (make-unary (minus k 1))))))
220
221 (define (eval t)
222   (try (eval2 t)
223        (println (join "Unable to reduce the term:\n" (val->string t) "\nto a normal form."))))
224
225
226 (define (eval t)
227   (try (eval2 t)
228        (println (join "Unable to reduce the term:\n" (val->string t) "\nto a normal form."))))
229
230
231 (set-precedence eval 100)
```