lib/basic/rewriting.ath

```
(load-file "lib/basic/prop-tab.ath")

(load-file "lib/basic/tableaux.ath")

(load-file "lib/basic/msr.ath")

## (standard-reduce-proc-name-suffix)

(define (dm-rec premise)
  (!prop-taut premise (app-dm premise)))

## NEW DEFS, JULY 03 2015:
# (define (get-eval-proc-name' f)
#   (let ((N (arity-of f))
#         (t (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 N)))))
#     (get-eval-proc-name (root t))))

# (define (get-eval-proc-name-1' f)
#   (let ((N (arity-of f))
#         (t (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 N)))))
#     (get-eval-proc-name-1 (root t))))
#(define get-eval-proc-name get-eval-proc-name')
#(define get-eval-proc-name-1 get-eval-proc-name-1')

(define (get-reduce-proc-name f)
  (join (get-eval-proc-name f) (standard-reduce-proc-name-suffix)))

(define (get-reduce-proc-name-1 f)
  (join (get-eval-proc-name-1 f) (standard-reduce-proc-name-suffix)))

(define derive-theorem
  (method (goal premises)
       (!vprove-from goal premises [['poly true] ['subsorting false] ['max-time 60000]])))

(define mderive-theorem
  (method (goal premises)
       (!vprove-from goal premises [['poly false] ['subsorting false] ['max-time 600000]])))

(define sderive-theorem
  (method (goal premises)
       (!sprove-from goal premises [['poly true] ['subsorting false] ['max-time 60]])))


(define (existence p q)
    (dlet ((q-body (quant-body q)))
      (dmatch (match-props-modulo-CD p q-body)
        ((some-sub sub) (dseq (!mp (!taut (if p (sub q-body))) p)
                              (!egen* q (sub (qvars-of q))))))))

(define (restrict-right-hand-side-sorts left right)
  (let ((left-fvars  (vars left))
        (left-fvar-names (map var->string left-fvars))
        (right-fvars (filter (vars right)
                             (lambda (v) (member? (var->string v) left-fvar-names)))))
    (match (match-terms left-fvars right-fvars)
      ((some-sub sub) (sub right))
      (_ right))))


(define (sorted-tran eq1 eq2)
  (dtry (!tran eq1 eq2)
        (dlet (([eq1' eq2'] (match (and eq1 eq2)
                              ((and (some-sent p1) (some-sent p2)) [p1 p2])))
               (eq1'' (!sort-instance eq1 eq1'))
               (eq2'' (!sort-instance eq2 eq2')))
           (!tran eq1'' eq2''))))
```

```
(define (restrict-right-hand-side-sorts l r) r)

(define deval-cell (cell ()))

(primitive-method (leibniz t1 t2 newP fv)
  (iff (replace-var fv t1 newP)
       (replace-var fv t2 newP)))

(define (leibniz t1 t2 newP fv)
  (!force (iff (replace-var fv t1 newP)
               (replace-var fv t2 newP))))


(define vpf' vpf)

(define (vpf goal props)
  (dlet ((_ (print "\nCalling external theorem prover...\n")))
    (!vpf' goal props)))

(define (vpf goal props)
 (dlet ((_ ()))
  (!fail "About to call external ATP, failing instead...\n")))

(declare unary ((T) -> (T) T))

#(declare (--> <-- <--> <== ==> <==>) Boolean)

#(declare (--> <-- <-->) Boolean)
#(define ==> ===>)
#(define <== <===)
#(define <==> <===>)

conclude neg-id-lemma :=
   (forall ?x . false <==> forall ?y . ?x =/= ?y)
  pick-any x
     (!equiv
         assume false
            (!from-false (forall ?y . x =/= ?y))
         assume  hyp := (forall ?y . x =/= ?y)
           (!absurd conclude (x = x)
                      (!reflex x)
                     conclude (x =/= x)
                      (!uspec hyp x)))

##############################################################################
# (define (direction? x)
#   (member? x [= --> <-- <--> <== ==> <==>]))

# (define (equational? x)
#   (member? x [= --> <--]))

# (define (chain-symbol? x)
#   (member? x [--> <-- <--> <== ==> <==> =]))

(define (direction? x)
   (|| (member? x [= --> <-- <--> ==> <==>])
       (&& (binary-proc? x) (let ((res (x true true)))
                              (|| (equal? res (if true true)) (equal? res (iff true true)))))))

(define (equational? x)
   (member? x [= --> <--]))

(define chain-symbol? direction?)

(define (direction->string x)
  (match x
    (if "==>")
    (iff "<==>")
    (_ (symbol->string x))))
```

```
#########################################################################################

(define (in-rewriting-trace-mode)
  (member? (get-debug-mode) ["rewriting" "simple" "detailed"]))

(define level (cell 0))

(define (indent level)
  (check ((less? level 1)
          ())
         (else (seq
                 (print "    ")
                 (indent (minus level 1))))))

(define (rename' p)
  (match p
    ((some-sent _) (rename p))
    (_ p)))

### DRM: my attempt to define this (needed by pos-substitute-equals)
(define (prop-pos-replace P pos v)
  (letrec ((prop-pos-replace-args
            (lambda (n rest args v)
                      (match args
                        ((list-of x rest-args)
                         (check
                          ((equal? n 1)
                           (add (prop-pos-replace x rest v) rest-args))
                          (else
                           (add x (prop-pos-replace-args (minus n 1) rest rest-args v)))))
                        ([] [])))))
    (match pos
      ([] v)
      ((list-of n rest)
       (match P
         (((some-symbol f) (some-list args))
          (make-term f (prop-pos-replace-args n rest args v)))
         ((not arg)
          (check
           ((equal? n 1)
            (not (prop-pos-replace arg rest v)))
           (else P)))
         (((some-sent-con pc) P1 P2)
           (check
            ((equal? n 1)
             (pc (prop-pos-replace P1 rest v)
                 P2))
            ((equal? n 2)
             (pc P1
                 (prop-pos-replace P2 rest v)))
            (else P)))
         (((some-quant quant) _v B)
          (check
           ((equal? n 3)
            (quant _v (prop-pos-replace B rest v)))
           (else P))))))))

#############################################################
(define (three-cases case1 case2 case3)
  (dmatch [case1 case2 case3]
    ([(if P1 Q) (if P2 Q) (if (and (not P1) (not P2)) Q)]
     (dseq
      (!two-cases (if P1 Q)
                  ((if (not P1) Q)
                   BY (assume (not P1)
                        (!two-cases (if P2 Q)
                                    ((if (not P2) Q)
                                     BY (assume (not P2)
                                          (!mp (if (and (not P1) (not P2)) Q)
                                               (!both (not P1) (not P2))))))))))))
```

```

##########################################################
# If a proposition of the form (and Q R), (or Q R), or (iff Q R) is in
# the assumption base, prove the same proposition with the arguments
# exchanged.  E.g.,
#   (assume (and A B) (!reorder (and A B)))
# proves (if (and A B) (and B A))

(define (reorder P)
  (dmatch P
    ((and Q R)
     (!both (!right-and (and Q R))
            (!left-and (and Q R))))
    ((or Q R)
     (!cases (or Q R)
          (assume Q
            (!either R Q))
          (assume R
            (!either R Q))))
    ((iff Q R)
     (!equiv
      (assume R
        (!mp (!right-iff (iff Q R)) R))
      (assume Q
        (!mp (!left-iff (iff Q R)) Q))))))

## The following method takes two terms t1 and t2 where
## the equality (= t1 t2) is in the assumption base, and
## derives the theorem (= t2 t1).

## This method takes three terms t1, t2, and t3 such that
## t1 = t2 and t2 = t3 hold, and derives the equality t1 = t3.

## The method below takes a term t1, a theorem P, and a term t2,
## where the equality (= t1 t2) holds, and returns the proposition
## obtained from P by replacing every occurrence of t1 by t2.

(define (replace-term-in-term t1 t t2)
  (match t
     ((val-of t1) t2)
     (((some-symbol f) (some-list args))
         (make-term f (map (lambda (t) (replace-term-in-term t1 t t2)) args)))
     (s s)))

(define (replace-term-in-prop t1 P t2)
  (match P
    ((some-atom t) (replace-term-in-term t1 t t2))
    ((not _Q) (not (replace-term-in-prop t1 _Q t2)))
    (((some-sent-con pc) (some-list props))
         (pc (map (lambda (p)
                    (replace-term-in-prop t1 p t2)) props)))
    (((some-quant quant) v B) (quant v (replace-term-in-prop t1 B t2)))))

define replace-term-in-sentence := replace-term-in-prop

(define (substitute-equals t1 P t2)
  (dlet ((fv (fresh-var (sort-of t1)))
         (newP (replace-term-in-prop t1 (rename P) fv))
         (biconditional (dseq (dcheck ((holds? (= t1 t2)) (!sym (= t1 t2)))
                                      (else (!sym (= t2 t1))))
                              (!leibniz t1 t2 newP fv))))
    (!mp (!left-iff biconditional) P)))

(define (equality-to-equivalence p)
  (dmatch p
    ((= s t) (!equiv (assume s
                       (!substitute-equals s s t))
                     (assume t
                       (!substitute-equals t t s))))))

```

```
## The following is a more selective, positional version of
## substitute-equals. It takes a term t1, a theorem P, a position
## pos (represented as a list of numeric terms, say [2 1 4]) and
## a term t2, where the equality (= t1 t2) must hold. It returns
## the proposition obtained from P by replacing the occurrence of
## t1 in P at position pos by t2.

(define (pos-substitute-equals t1 P pos t2)
  (dlet ((t1=t2 (!claim (= t1 t2)))
         (v  (fresh-var (sort-of t1)))
         (newP (prop-pos-replace P pos v))
         (biconditional (!leibniz t1 t2 newP v)))
    (!mp (!left-iff biconditional) P)))

## The method eq-congruence takes two terms t1 and t2, where
## the equality (= t1 t2) must hold, a term t, and a variable v
## and returns the equality (= t1' t2'), where t1' is obtained from
## t by replacing every occurrence of v by t1, and t2' is obtained
## from t by replacing every occurrence of v by t2.

# (define eq-congruence
#   (method (t1 t2 t v)
#     (dlet ((t1=t2 (!claim (= t1 t2)))
#            (v' (fresh-var))
#            (newt (replace-var v v' t))
#            (newt{t2/v'} (replace-var v' t2 newt))
#            (prop (= newt newt{t2/v'}))
#            (newt{t1/v'}=newt{t2/v'}<==>newt{t2/v'}=newt{t2/v'} (!leibniz t1 t2 prop v'))
#            (newt{t2/v'}=newt{t2/v'}==>newt{t1/v'}=newt{t2/v'}
#                 (!right-iff newt{t1/v'}=newt{t2/v'}<==>newt{t2/v'}=newt{t2/v'}))
#            (newt{t2/v'}=newt{t2/v'} (!reflex newt{t2/v'})))
#       (!mp newt{t2/v'}=newt{t2/v'}==>newt{t1/v'}=newt{t2/v'}
#            newt{t2/v'}=newt{t2/v'}))))

## The following method, positional congruence, works with positions
## instead of variables. It takes again two terms t1 and t2 such that
## (= t1 t2) is a theorem, a term t, and a position pos, and returns
## the equality (= t1' t2'), where t1' is obtained from t by replacing
## plugging t1 at position pos, and t2' is obtained from t by plugging
## t2 at position pos.

# (define (pos-congruence t1 t2 t pos)
#   (dlet ((v (fresh-var [t1 t2 t]))
#          (newt (term-replace t pos v)))
#     (!eq-congruence t1 t2 newt v)))

##==============================================================================
##                             FUNCTION CONGRUENCE
##==============================================================================

## The method fun-cong below takes a function symbol f (of arbitrary arity),
## and two lists of terms, s-terms = [s1 ... sn] and t-terms = [t1 ... tn],
## such that s_i = t_i is in the assumption base for every i = 1,...,n,
## and derives the equality f(s1,...sn) = f(t1,...,tn).

(define (fun-cong f s-terms t-terms)
  (dletrec ((v (fresh-var (join (vars* s-terms) (vars* t-terms))))
            (do-args (method (first-s_i first-t_i rem-s_j rem-t_j eq)
                        (dmatch [rem-s_j rem-t_j]
                          ([[] []] (!claim eq))
                          ([(list-of s_j more-s_j) (list-of t_j more-t_j)]
                              (dlet ((F (= (make-term f s-terms)
                                           (make-term f (join first-t_i (join [v] more-s_j)))))
                                     (bi-cond (!leibniz s_j t_j F v))
                                     (new-eq (!mp (!left-iff bi-cond) eq)))
                                (!do-args (join first-s_i [s_j])
                                          (join first-t_i [t_j])
                                          more-s_j more-t_j new-eq)))))))
     (!do-args [] [] s-terms t-terms (!reflex (make-term f s-terms)))))
```

```
(define (fun-cong f s-terms t-terms)
  (!fcong (= (make-term f s-terms) (make-term f t-terms))))

##=========================================================================
##                            RELATION CONGRUENCE
##=========================================================================

## The method rel-cong below takes an atomic theorem P of the form (R s1 ... sn),
## the terms s-terms [s1...sn], and the terms t-terms [t1...tn], where si = ti
## must be in the assumption base for all i, and returns the theorem (R t1 ... tn).

(define (rel-cong P s-terms t-terms)
  (dletrec ((do-args (method (s-terms t-terms theorem)
                       (dmatch [s-terms t-terms]
                         ([[] []] (!claim theorem))
                         ([(list-of s more-s) (list-of t more-t)]
                             (dlet ((new-theorem (!substitute-equals s theorem t)))
                               (!do-args more-s more-t new-theorem)))))))
     (!do-args s-terms t-terms P)))

## DRM: the above definition applies each substitution to the whole
## theorem P, which is incorrect (according to the comment preceding
## it).  The following is a modified version that uses
## pos-substitute-equals instead of substitute-equals, to restrict
## application of individual substitution to the corresponding
## argument position only.

(define (rel-cong P s-terms t-terms)
  (dletrec ((do-args (method (s-terms t-terms theorem n)
                       (dmatch [s-terms t-terms]
                         ([[] []] (!claim theorem))
                         ([(list-of s more-s) (list-of t more-t)]
                          (dlet ((new-theorem
                                  (!pos-substitute-equals s theorem [n] t)))
                            (!do-args more-s more-t new-theorem (plus n 1))))))))
    (!do-args s-terms t-terms P 1)))

## The argument list s-terms in the method rel-cong above is superfluous, since
## it can be extracted from the atomic theorem P. Hence rel-cong-2 below simply
## takes a theorem P, which again must be of the form (R s1 ... sn), and a list
## of terms t-terms [t1 ... tn], where si = ti must be in the asm. base, and
## derives the theorem (R t1 ... tn)

(define (rel-cong-2 P t-terms)
  (dcheck ((atom? P) (!rel-cong P (children P) t-terms))))

##===========================================================================
##                              RECURSIVE CONGRUENCE
##===========================================================================

## This is a powerful recursive congruence method. If any
## subterms of t1 and t2 in corresponding positions are equal
## (with the equality in the assumption base), everything else
## being the same, then the theorem (= t1 t2) is returned.

(define (rec-cong t1 t2)
    (dmatch (equal? t1 t2)
      (true (!equality t1 t1))
      (_ (dmatch (fetch (lambda (P)
                          (|| (equal? P (= t1 t2))
                              (equal? P (= t2 t1)))))
           (() (dlet ((root1 (root t1))
                      (root2 (root t2))
                      (args1 (children t1))
                      (args2 (children t2)))
                 (dmatch (equal? root1 root2)
                   (true (dletrec ((do-args
                                    (method (s-terms t-terms)
                                      (dmatch [s-terms t-terms]
                                        ([[] []] (!fun-cong root1 args1 args2))
```

```
                                        ([(list-of s1 more) (list-of t1 rest)]
                                          (dseq
                                            (!rec-cong s1 t1)
                                            (!do-args more rest)))))))
                           (!do-args args1 args2))))))
           (P (dmatch P
                ((= (val-of t1) (val-of t2)) (!claim P))
                (_ (!sym (= t2 t1))))))))))




(define (rec-cong t1 t2)
        (dcheck
           ((equal? t1 t2) (!reflex t1))
           ((holds? (= t1 t2)) (!claim (= t1 t2)))
           ((holds? (= t2 t1)) (!sym (= t2 t1)))
           (else (dlet ((root1 (root t1))
                        (root2 (root t2))
                        (args1 (children t1))
                        (args2 (children t2)))
                   (dmatch (equal? root1 root2)
                     (true (dletrec ((do-args
                                       (method (s-terms t-terms)
                                         (dmatch [s-terms t-terms]
                                           ([[] []] (!fun-cong root1 args1 args2))
                                           ([(list-of s1 more) (list-of t1 rest)]
                                             (dseq
                                               (!rec-cong s1 t1)
                                               (!do-args more rest)))))))
                             (!do-args args1 args2))))))))


(define (rec-rel-cong p q)
  (dlet ((p-terms (children p))
         (q-terms (children q)))
    (!map-method
       (method (term-pair)
         (dmatch term-pair
           ([(some-term s) (some-term t)] (!rec-cong s t))))
       (list-zip p-terms q-terms)
       (method (results)
         (!rcong p q)))))



## The procedure try-rewrite determines whether a term s rewrites into a term t
## on the basis of a given rewrite rule left -> right, and if so, returns the
## corresponding substitution. Specifically, a call (try-rewrite s t left right K)
## will return a substitution theta whenever (a) s matches left under theta
## and (b) applying theta to right yields t. If either (a) or (b) is false, then
## the failure continuation K is invoked.


#(define match-terms-core match-terms)

# (define (match-terms s t uvars)
#   (match [s t]
#     ([_ (- (- (val-of s)))] true)
#     ([(- (- (some-term x))) x] true)
#     (_ (match-terms s t uvars))))

# (define (equal-up-to-double-negation s t)
#    (|| (equal? s t)
#        (&& (numeral? s)
#            (|| (equal? (- (- s)) t) (equal? s (- (- t)))))))

(define (try-rewrite s t left right uvars K)
 (let ((eqn (= s t)))
   (match (match-terms eqn (= left right) uvars)
```

```
     ((some-sub sub) (check ((|| (equal? t (sub right)) (sort-instance? (rhs eqn) (sub right))) sub)
                            (else (K))))
     (_ (K)))))

## A call (rewrites? s t rule direction), where rule is a universally quantified
## identity (forall v1 ... vk (= t1 t2)) or (forall v1 ... vk (if p (= t1 t2))),
## possibly with zero quantifiers, will return the relevant substitution
## if s rewrites into t on the basis of rule, or if t rewrites into s on
## the basis of rule, depending on the given direction (or if either holds,
## if the direction is =). If neither holds, then the constant 'false' is returned.

(define (get-identity p)
  (match p
    ((= _ _) p)
    ((if _ (bind consequent (= _ _))) consequent)
    ((iff _ (bind consequent (= _ _))) consequent)
    (_ ())))

## UQM

(define (rewrites? s t rule direction)
  (match rule
    ((forall (some-list uvars) (= (some-term L) (ite _ (some-term R1) (some-term R2))))
        (match (rewrites? s t (forall* uvars (= L R1)) direction)
          ((some-sub sub) sub)
          (_ (rewrites? s t (forall* uvars (= L R2)) direction))))
    ((forall (some-list uvars) body)
      (match (get-identity body)
        (((= left right) where (negate (&& (var? left) (var? right))))
           (let ((failure-cont (lambda () false)))
             (match direction
                (--> (try-rewrite s t left right uvars failure-cont))
                (<-- (try-rewrite t s left right uvars failure-cont))
                (= (try-rewrite s t left right uvars
                      (lambda () (try-rewrite t s left right uvars failure-cont)))))))
         (_ false)))
    (_ false)))

(define (show t1 t2 direction show-left-term?)
  (let ((f (lambda ()
             (seq (indent (plus (get-trace-level) 1))
                  (check (show-left-term? (seq (indent-print (plus (times 4 (get-trace-level)) 2) t1)
                                               (print newline)
                                               (indent (plus (get-trace-level) 1))))
                         (else (indent (minus (get-trace-level) 1))))
                  (print (join (direction->string direction) newline))
                  (indent (plus (get-trace-level) 1))
                  (indent-print (plus (times 4 (get-trace-level)) 2) t2)
                  (print newline)))))
  (check ((equal? t1 t2) ())
         (else (match (get-debug-mode)
                 ("rewriting" (f))
                 ("detailed" (f))
                 (_ ()))))))

(define (prove-condition p methods)
  (!find-some methods (method (M) (!M p)) fail))

(define (negateR x)
  (match x
    (true false)
    (false true)
    (_ (not x))))


(define (orient rule)
  (dmatch rule
    ((forall (some-list var-list) (if ant (= s t)))
      (dcheck ((|| (subset? (vars t) (vars s))
                   (negate (subset? (vars s) (vars t)))
```

```
                   (negate (null? (fv rule))))
               (!claim rule))
              (else (!generalize var-list (method (eigen-vars)
                                            (dlet ((rule' (!uspec* rule eigen-vars))
                                                   (ant' (antecedent rule')))
                                              (assume ant'
                                                   (!sym (!mp rule' ant'))))))))
    ((forall (some-list var-list) (iff ant (= s t)))
      (dcheck ((|| (subset? (vars t) (vars s))
                   (negate (subset? (vars s) (vars t)))
                   (negate (null? (fv rule))))
               (!claim rule))
              (else (!generalize var-list (method (eigen-vars)
                                            (dlet ((rule' (!uspec* rule eigen-vars))
                                                   (rule'' (!left-iff rule'))
                                                   (ant' (antecedent rule'')))
                                              (assume ant'
                                                   (!sym (!mp rule'' ant'))))))))
    ((forall (some-list var-list) (= s t))
      (dcheck ((|| (subset? (vars t) (vars s))
                   (negate (subset? (vars s) (vars t)))
                   (negate (null? (fv rule))))
                (!claim rule))
              (else (!generalize var-list (method (eigen-vars)
                                            (dlet ((rule' (!uspec* rule eigen-vars)))
                                              (!sym rule'))))))))

#(define rule (forall ?x ?y (= ?x (Plus ?y ?y))))

#(assume rule (!orient rule))

#(define orient claim)

## A call (!rewrite* t1 t2 rules direction) attempts to derive the identity (= t1 t2)
## by rewriting t1 into t2 or vice versa (depending on the 'direction' argument)
## on the basis of the given rules, each of which is of the form
##         (forall v1 ... vk (= s t))
##      or (forall v1 ... vk (if _ (= s t))),
##      or (forall v1 ... vk (iff _ (= s t))),
## where we might have zero universal quantifiers. The rules must be in the a.b.
## Any number of subterms of each term may be rewritten, by any of the given rules.
## However, if the given direction is other than =, i.e., if it's either --> or
## <--, then the rewriting can only proceed in one direction, e.g., in the case of -->,
## the rules are applied only to subterms of t1, and in the case of <--, only
## to subterms of t2. This can limit the usefulness of the method, since in the
## general case there is no reason to explicitly restrict the direction of the
## rewriting (from a cognitive perspective, equalities are generally perceived
## as inherently symmetric). So the default direction should be =.
## Note that if t1 and t2 are identical then no rules need to be supplied,
## i.e., (!rewrite* t t []) will always derive (= t t). Obviously, if the
## rule is conditional then the condition(s) must obtain for the rule
## to be applied successfully.

(define (rewrite-one-redex t1 t2 equation sub)
  (dlet (([uvars left right] (decompose-equation equation)))
    (dletrec ((loop (method (s t)
                      (dcheck ((&& (equal? (sub left)  s)
                                   (equal? (sub right) t))
                                (dmatch (!uspec* equation (sub uvars))
                                  ((bind p (= (val-of s) _)) (!claim p))
                                  ((bind p (if ant (= (val-of s) _))) (!mp p (!prove-components-harder ant)))
                                  ((bind p (if ant (= _ _))) (!sym (!mp p (!prove-components-harder ant))))
                                  ((bind p (iff _ (= (val-of s) _)))
                                    (dlet ((p' (!left-iff p)))
                                     (!mp p' (!prove-components-harder (antecedent p')))))
                                  ((bind p (iff _ (= _ _)))
                                    (dlet ((p' (!left-iff p)))
                                      (!sym (!mp p' (!prove-components-harder (antecedent p'))))))))
                              ((equal? s t) (!equality s t))
                              (else (!map-method
```

```
                                       (method (term-pair)
                                         (dmatch term-pair
                                           ([s' t'] (!rewrite-one-redex s' t' equation sub))))
                                       (zip (children s) (children t))
                                       (method (results)
                                        (!fcong (= s t)))))))))
      (!loop t1 t2))))

(define (rewrite-one-redex t1 t2 equation sub)
  (dlet (([uvars left right] (decompose-equation equation))
        (body (!uspec* equation (sub uvars))))
    (dletrec ((loop (method (s t)
                      (dcheck ((|| (equal? (sub left)  s) (equal? (sub right)  s))
                                (dmatch body
                                  ((bind p (= (val-of s) _)) (!claim p))
                                  ((bind p (= _ _)) (!sym p))
                                  ((bind p (if ant (= (val-of s) _))) (!mp p (!prove-components-harder ant)))
                                  ((bind p (if ant (= _ _))) (!sym (!mp p (!prove-components-harder ant))))
                                  ((bind p (iff _ (= (val-of s) _)))
                                    (dlet ((p' (!left-iff p)))
                                     (!mp p' (!prove-components-harder (antecedent p')))))
                                  ((bind p (iff _ (= _ _)))
                                    (dlet ((p' (!left-iff p)))
                                      (!sym (!mp p' (!prove-components-harder (antecedent p'))))))))
                              ((equal? s t) (!equality s t))
                              (else (!map-method
                                       (method (term-pair)
                                         (dmatch term-pair
                                           ([s' t'] (!loop s' t'))))
                                       (zip (children s) (children t))
                                       (method (results)
                                        (!fcong (= s t)))))))))
      (!loop t1 t2))))

## ite-ir takes an equality of the form (= s (ite C R1 R2)) and produces
## (= s R1) if C holds, or else (= s R2) if (~ C) holds.

(primitive-method (ite-ir ite-equality)
   (match ite-equality
     ((= (some-term s) (ite  condition res1 res2))
       (check ((holds? condition) (= s res1))
              ((holds? (complement condition)) (= s res2))))))

(primitive-method (ite-ir* ite-equality)
  (letrec ((loop (lambda (ite-equality)
                   (match ite-equality
                     ((= (some-term s) (ite  condition res1 (nested-ite-term as (ite _ _ _))))
                       (check ((holds? condition) (= s res1))
                              (else (loop (= s nested-ite-term)))))
                     ((= (some-term s) (ite _ _ _)) (let ((res (!ite-ir ite-equality))) res))))))
    (loop ite-equality)))

(define (ite-ir' C R1 R2)
   (dlet ((ite-term (ite C R1 R2)))
     (!ite-ir (!reflex ite-term))))

(define (ite-term? t)
  (match t
    ((ite _ _ _) true)
    (_ false)))


(define (search t1 t2 rules direction)
  (dtry (!drs-bf t1 t2 rules rewrite-one-redex)
        (dcheck ((equal? direction =)
                 (!sym (!drs-bf t2 t1 rules rewrite-one-redex))))))


(define (find-ite-match-0 t1 t2 lhs C rhs1 rhs2)
  (let ((eqn (= t1 t2)))
```

```
     (match (match-terms eqn (= lhs rhs1))
        ((some-sub _) C)
        (_ (match (match-terms eqn (= lhs rhs2))
             ((some-sub _) (complement C))
             (_ (match rhs1
                  ((ite (some-term C') (some-term rhs1') (some-term rhs2'))
                     (find-ite-match-0 t1 t2 lhs C' rhs1' rhs2'))
                  (_ ())))))))))

(define (find-ite-match t1 t2 lhs C rhs1 rhs2)
 (let (
       #(_ (print "\nAbout to call find-ite-match on t1: " t1 ", t2: " t2 ", lhs: " lhs ", C: " C
       #           ", rhs1: " rhs1 ", and rhs2: " rhs2)
      )
  (match (find-ite-match-0 t1 t2 lhs C rhs1 rhs2)
     (() (match (find-ite-match-0 t2 t1 lhs C rhs1 rhs2)
           ((some-sent cond) [cond 'reversed])
           (_ (error "No match..."))))
    ((some-sent cond) [cond 'normal]))))

(define (rewrite** t1 t2 rules direction)
  (dletrec ((loop (method (terms1 terms2)
                    (dmatch [terms1 terms2]
                      ([[] []] (!fcong (= t1 t2)))
                      ([(list-of s rest1) (list-of t rest2)]
                        (dseq (!rewrite** s t rules direction)
                              (!loop rest1 rest2))))))
            (rules' (match rules
                      ((some-list _) rules)
                      ((some-sent P) [P]))))
   (dlet ((methods' (filter rules' method?)))
    (dcheck ((equal? t1 t2) (!equality t1 t2))
            (else (!find-some rules'
                     (method (rule)
                       (dlet ((rule' (!orient rule)))
                         (dmatch [rule' (rewrites? t1 t2 rule' direction)]
                           ([(forall (some-list vars) _) (some-sub sub)]
                              (dlet (# (_ (print "\nMATCHING SUB for rewriting: "
                                     # t1 " into " t2 ":\n" sub "\nand rule':\n" rule' "\nand vars:\n" vars))
                                     (res
                               (dmatch (!uspec* rule' (sub vars))
                                 ((bind p (= (some-term lhs) (ite (some-term C)
                                                                  (some-term rhs1)
                                                                  (some-term rhs2))))
                                    (dlet ((res (dmatch (find-ite-match t1 t2 lhs C rhs1 rhs2)
                                                  ([(some-sent cond) 'reversed]
                                                    (dlet ((_ (!prove-condition cond
                                                                (add prove-components-of methods'))))
                                                      (!sym (!ite-ir* p))))
                                                  ([(some-sent cond) _]
                                                    (dlet ((_ (!prove-condition cond
                                                                 (add prove-components-of methods'))))
                                                      (!ite-ir* p))))))
                                      (dcheck ((equal? res (= t1 t2)) (!claim res))
                                              (else (!sort-instance res (= t1 t2))))))
                                 ((bind p (= _ _)) (dmatch (match-terms (= t1 t2) p vars)
                                                     ((some-sub _) (!claim p))
                                                     (_ (dmatch (match-terms (= t2 t1) p vars)
                                                          ((some-sub _) (!sym p))))))
                                 ((bind p (if _ (= (val-of t1) _))) (dlet ((_ ())
                                                                          ## (_ (print "\nAntecedent: " (antecedent p))
                                                                             (th (!prove-condition (antecedent p)
                                                                                                   (add prove-component
                                                                            ## (_ (print "\nPROVED ANTECEDENT!\n"))
                                                                            (_ ()))
                                                                        (!mp p (!prove-condition (antecedent p)
                                                                                                 (add prove-components-
                                 ((bind p (if _ (= _ _))) (dlet (## (_ (print "\nTrying antecedent: " p))
                                                                 (th  (!prove-condition (antecedent p)
                                                                                        (add prove-components-of meth
```

```
                                                                  (_ ()))
                                                            (!sym (!mp p th))))
                                 ((bind p (iff _ (= (val-of t1) _)))
                                  (dlet ((p' (!left-iff p)))
                                    (!mp p' (!prove-condition (antecedent p') (add prove-components-of methods')))))

                                 ((bind p (iff _ (= _ _)))
                                  (dlet ((p' (!left-iff p)))
                                    (!sym (!mp p' (!prove-condition (antecedent p') (add prove-components-of methods')
                                 (dtry (conclude (= t1 t2)
                                         (!claim res))
                                       (conclude (= t1 t2)
                                         (!sort-instance res (= t1 t2)))
                                       (dmatch t1
                                         ((ite C (some-term R1) (some-term R2))
                                            (dcheck ((equal? t2 R1) (dlet ((_ (!prove-condition C (add prove-component
                                                                       (!ite-ir' C R1 R2)))
                                                    ((equal? t2 R2) (dlet ((_ (!prove-condition (complement C) (add pr
                                                                       (!ite-ir' C R1 R2))))))
                                        ))))))
                     (method ()
                       (dmatch [t1 t2]
                         ([((some-symbol f) (some-list args1)) (f (some-list args2))]
                           (dtry (!loop args1 args2)
                                 (!search t1 t2 rules direction)))
                         (_ (!search t1 t2 rules direction))))))))))


(define (rewrite*** t1 t2 rules direction)
  (dtry (!rewrite** t1 t2 rules direction)
        (!fail "\nAbout to call ATPs, failing instead... \n")))

(define (rewrite*** t1 t2 rules direction)
    (dcheck ((get-boolean-flag "atps-with-chain")
              (!vprove-from (= t1 t2) rules [['poly true] ['subsorting false] ['max-time 160]]))
            (else  (!rewrite** t1 t2 rules direction))))

#(define (rewrite*** t1 t2 rules direction)
#  (!thread-methods [(method () (!rewrite** t1 t2 rules direction))
#                    (method () (!derive-theorem (= t1 t2) rules))]))


## This method takes two equational theorems eq1 and eq2, where eq1 is
## t=u and eq2 is v=u, and derives the equational theorem t=v.
## ****> Superceded by combine (which is used with reduce and expand)

(define combine-equations
  (method (eq1 eq2)
    (dmatch eq1
      ((= t11 t12)
       (dmatch eq2
         ((= t21 t22)
          (dcheck
           ((equal? t12 t22)
            (dseq
             (!sym (= t21 t22))
             (!tran (= t11 t22) (= t22 t21)))))))))))

#################################################################################
##
## Rewriting methods: setup, reduce, expand, combine

(define (universal-quantifiers P)
  (match P
    ((forall _x _Q) (add _x (universal-quantifiers _Q)))
    (_ [])))

(define (universal-quantifierless P)
  (match P
    ((forall x _Q) (universal-quantifierless _Q))
```

```
    (_ P)))

(define (positions&subterms t k)
  (add [[] t]
       (fold join (map
                   (lambda (child)
                             (let ((n (cell 0))
                                   (p&s (positions&subterms child n)))
                               (seq
                                 (set! k (plus (ref k) 1))
                                 (map
                                  (lambda (position&subterm)
                                            (match position&subterm
                                              ([position subterm]
                                               [(add (ref k) position) subterm])))
                                  p&s))))
                   (children t))
             []))))


(define (positions-and-subterms t)
  (positions&subterms t (cell 0)))

(define (attempt-rewrite current-equation new-term
                         proposition position subterm direction)
    (dlet ((term0
            (match (universal-quantifierless proposition)
              ((= _lhs _rhs) _lhs)
              ((if condition (= _lhs _rhs))
               _lhs)
              (_p (let ((dummy
                         (!proof-error (join "Left-hand-side of a proposition used "
                                             "in rewriting must be\n (with quantifiers removed) "
                                             "an equality or a conditional equality.\n"
                                             "Instead it was\n" (val->string _p) "\n"))))
                   dummy))))
           (subst (unify term0 subterm)))
      (dcheck
       ((negate (equal? subst false))
        (dlet ((proposition1
                (!uspec* proposition
                         (subst (universal-quantifiers proposition))))
               (result
                (dmatch proposition1
                  ((= _lhs _rhs)
                   (dtry
                    (dmatch direction
                      (--> (!pos-substitute-equals
                            _lhs (ref current-equation) position _rhs))
                      (<-- (dseq
                            (!sym proposition1)
                            (!pos-substitute-equals
                             _rhs (ref current-equation) position _lhs))))
                    (!true-intro)))
                  ((if condition (= _lhs _rhs))
                   (dtry
                    (dseq
                     (dcheck ((holds? condition)
                              (!true-intro))
                             (else (!claim false)))
                     (dmatch direction
                       (--> (dseq
                             (!mp proposition1 condition)
                             (!pos-substitute-equals
                              _lhs (ref current-equation) position _rhs)))
                       (<-- (dseq
                             (!sym (!mp proposition1 condition))
                             (!pos-substitute-equals
                              _rhs (ref current-equation) position _lhs)))))
                    (!true-intro)))))
```

```
               (hit-target? (match result
                              ((= _lhs _rhs)
                               (equal? _rhs new-term))
                              (_ false))))
          (dcheck (hit-target?
                   (dlet ((dummy (set! current-equation result)))
                     (!claim result)))
                  (else (!true-intro)))))
        (else (!true-intro)))))

(define (try-all-terms current-equation new-term equation
                       positions&subterms direction)
  (dmatch positions&subterms
    ((list-of [position subterm] more)
     (dlet ((attempt
             (!attempt-rewrite current-equation new-term
                               equation position subterm direction)))
       (dcheck ((equal? attempt true)
                (!try-all-terms current-equation new-term equation
                                more direction))
               (else (!claim attempt))))))

# For debugging uncomment the following redefinition and change
# the position being checked to a position where you think the error is
# occurring.
# (define (try-all-terms current-equation new-term equation
#                        positions&subterms direction)
#   (dmatch positions&subterms
#     ((list-of [position subterm] more)
#      (dlet ((dummy (seq
#                     (print "\nposition: ") (write position)
#                     (write subterm))))
#        (dcheck ((equal? position [2 1 2])
#                (!attempt-rewrite current-equation new-term equation
#                                  position subterm direction))
#               (else
#        (dlet ((success (cell true))
#              (ce (cell (ref current-equation))))
#        (dseq
#         (dtry (!attempt-rewrite ce new-term equation position
#                                  subterm direction)
#               (dlet ((dummy (set! success false)))
 #                (!true-intro)))
#         (dcheck ((equal? (ref success) true)
#                  (!attempt-rewrite current-equation new-term equation
#                                    position subterm direction))
#                 (else (!try-all-terms current-equation new-term equation
#                                       more direction))))))))))

(define previous-equation (cell (cell true)))

(define (in-rewriting-trace-mode)
  (member? (get-debug-mode) ["rewriting" "simple" "detailed"]))

(define (do-rewrite current-equation new-term equation direction)
  (dlet ((old-term (match (ref current-equation)
                     ((= _lhs _rhs) _rhs)))
         (goal-eqn (= old-term new-term))
         ([old-term new-term] (match goal-eqn
                                ((= (some-term L) (some-term R)) [L R])))
         (dummy (check ((in-rewriting-trace-mode)
                        (check ((negate (equal? current-equation
                                                (ref previous-equation)))
                                (seq
                                  (indent (plus (get-trace-level) 1))
                                  (print "Rewriting\n")
                                  (indent (plus (get-trace-level) 1))
                                  (indent-print (plus (times 4 (get-trace-level)) 2) old-term)
                                  (print "\n")
                                  ))
```

```
                              (else ()))))
                         (else ()))))
         (result (dmatch direction
                   (--> (!try-all-terms current-equation
                                         new-term
                                         (rename equation)
                                         (positions&subterms (unary old-term) (cell 1))
                                         direction))
                   (<-- (!try-all-terms current-equation
                                         new-term
                                         (rename equation)
                                         (positions&subterms (unary new-term) (cell 1))
                                         direction))
                   (= (dseq
                       (!derive (= old-term new-term) equation)
                       (dlet ((new (!tran (ref current-equation) (= old-term new-term)))
                              (dummy (set! current-equation new)))
                         (!claim new))))))
         (dummy1
          (check ((in-rewriting-trace-mode)
                  (seq
                    (indent (plus (get-trace-level) 1))
                    (match direction
                      (--> (print "-->\n"))
                      (<-- (print "<--\n"))
                      (= (print " = \n")))
                    (indent (plus (get-trace-level) 1))
                    (indent-print (plus (times 4 (get-trace-level)) 2) new-term)
                    (print "\n")
                    (set! previous-equation current-equation)))
                 (else ()))))
    (!claim result)))

(define (reduce current-equation new-term equation)
  (!do-rewrite current-equation new-term equation -->))

(define (expand current-equation new-term equation)
  (!do-rewrite current-equation new-term equation <--))

(define neither-left-nor-right (cell (cell true)))

(define (setup current-equation term)
  (dlet ((equation (!equality term term))
         (dummy (seq
                  (set! current-equation equation)
                  (set! previous-equation neither-left-nor-right))))
    (!claim equation)))

(define (combine left right)
  (!combine-equations (ref left) (ref right)))


###########################################################################
# More powerful rewriting method: chain
#
# (!chain [t_0 = t_1 [P_1] = t_2 [P_2] = ... = t_n [P_n]])
# proves and returns the equation (= t_0 t_n), provided each
# equation (= t_{i-1} t_i) can be obtained with reduce or expand
# using P_i.  Instead of =, one can use --> to restrict to
# the use of reduce or <-- to restrict to the use of expand.

(define (chain L)
  (dletrec ((c (cell true))
            (unbracket (lambda (P) (match P ([_P] (unbracket _P)) (_ P))))
            (bracket (lambda (P) (match P ((list-of _x _more) P) (_ [P]))))
            (chain-help
             (method (L)
               (dmatch L
                 ((list-of --> (list-of _y (list-of _P _rest)))
                  (dseq
```

```
                   (!reduce c _y (unbracket _P))
                   (!chain-help _rest)))
                 ((list-of <-- (list-of _y (list-of _P _rest)))
                  (dseq
                   (!expand c _y (unbracket _P))
                   (!chain-help _rest)))
                 ((list-of = (list-of _y (list-of _P _rest)))
                  (dseq
                   (!do-rewrite c _y (bracket _P) =)
                   (!chain-help _rest)))
                 ([] (!claim (ref c)))))))
    (dmatch L
      ((list-of t rest)
       (dseq
        (!setup c t)
        (!chain-help rest))))))


## The current version of chain* supports directional rewriting, so this should work:
## (!chain* [(Plus (succ ?foo) zero) --> (succ ?foo) [Plus-zero-axiom]])
## and so should this:
## (!chain* [(succ ?foo) <-- (Plus (succ ?foo) zero) [Plus-zero-axiom]])
## but this should not:
## (!chain* [(Plus (succ ?foo) zero) <-- (succ ?foo) [Plus-zero-axiom]])
## and nor should this:
## (!chain* [(succ ?foo) --> (Plus (succ ?foo) zero) [Plus-zero-axiom]])
## But this of course should work:
## (!chain* [(succ ?foo) = (Plus (succ ?foo) zero) [Plus-zero-axiom]])
## as should this:
## (!chain* [(Plus (succ ?foo) zero) = (succ ?foo) [Plus-zero-axiom]])

(define (show-equiv p1 p2 direction show-left-prop?)
  (let ((f (lambda ()
              (seq (indent (plus (get-trace-level) 1))
                   (check (show-left-prop? (seq (indent-print (plus (times 4 (get-trace-level)) 2) p1)
                                                (print newline)
                                                (indent (plus (get-trace-level) 1))))
                          (else (indent (minus (get-trace-level) 1))))
                   (print (join (direction->string direction) newline))
                   (indent (plus (get-trace-level) 1))
                   (indent-print (plus (times 4 (get-trace-level)) 2) p2)
                   (print newline)))))
  (check ((equal? p1 p2) ())
         (else (match (get-debug-mode)
                 ("rewriting" (f))
                 ("detailed" (f))
                 (_ ()))))))

(define (equate-subterms atom1 atom2 rules K)
  (dlet ((left-subterms  (children atom1))
         (right-subterms (children atom2))
         (term-pairs     (zip left-subterms right-subterms)))
    (!map-method
       (method (term-pair)
         (dmatch term-pair
           ([s t] (!rewrite*** s t rules =))))
       term-pairs K)))

(define (equate-atoms s t rules)
  (dlet ((identity (!rewrite*** s t rules =)))
    (!force t)))

(define (score atom)
  (lambda (atom')
     (check ((equal? (root atom) (root atom'))
             (plus 1 (check ((subset? (leaves atom') (leaves atom)) 1)
                            (else 0))))
            (else 0))))
```

```
(define (align atoms-1 atoms-2)
  (letrec ((loop (lambda (atoms-1 atoms-2 res)
                   (match atoms-1
                     ([] (rev res))
                     ((list-of atom-1 rest-1)
                       (find-max atoms-2 (score atom-1)
                         (lambda (A) (loop rest-1 (remove A atoms-2) (add A res)))
                         (lambda ()  (loop rest-1 atoms-2 (add atom-1 res)))))))))
    (loop atoms-1 atoms-2 [])))

(define (apply-tran premise-1 premise-2)
  (dmatch [premise-1 premise-2]
    ([(iff p1 p2) (if p2 p3)] (assume p1
                                (!mp premise-2 (!mp (!left-iff premise-1) p1))))
    ([(iff p1 p2) (if p3 p2)] (assume p3
                                (!mp (!right-iff premise-1)
                                     (!mp premise-2 p3))))
    ([(iff p1 p2) (iff p3 p2)] (assume p3
                                (!mp (!right-iff premise-1)
                                     (!mp (!left-iff premise-2) p3))))

    ([(iff p1 p2) (iff p2 p3)] (!equiv-tran premise-1 premise-2))
    ([(if p1 p2) (if p2 p3)] (assume p1
                               (!mp premise-2 (!mp premise-1 p1))))
    ([(if p1 p2) (if p3 p1)] (!apply-tran premise-2 premise-1))
    ([(if p1 p2) (iff p2 p3)] (assume p1
                                 (!mp (!left-iff premise-2)
                                      (!mp premise-1 p1))))
    ([(if p2 p1) (iff p2 p3)] (assume p3
                                 (!mp premise-1
                                      (!mp (!right-iff premise-2) p3))))
    ([p1 (if p1 p2)]  (!claim premise-2))
    ([p1 (iff p1 p2)] (!claim premise-2))
    ([p2 (iff p1 p2)] (!claim premise-2))))

(define (commute p)
  (match p
    ((and p1 p2) (and p2 p1))
    (_ p)))

(define (match-props-modulo-conj p q)
  (match (match-props p q)
    ((some-sub sub) sub)
    (_ (match [p q]
        ([(if (and p1 p2) body) (if (and q1 q2) body')]
            [(match-props p (if (and q2 q1) body'))])
        ([(iff (and p1 p2) body) (iff (and q1 q2) body')]
            [(match-props p (iff (and q2 q1) body'))])
        (_ ())))))


(define (get-components p kind)
  (match [p kind]
    ([(and (some-list args)) 'conj] args)
    ([(or (some-list args))  'disj] args)
    (_ ())))

(define (match-props-AC p1 p2 uvars)
  (match [(match-props-3 p1 p2 uvars) p2]
    ([(some-sub sub) _] [sub p2 'not-reversed])
    # ([_ (= (some-term s) (some-term t))]
    #         (match (match-props-3 p1 (= t s) uvars)
    #           ((some-sub sub) [sub p2 'reversed])
    #           (_ false)))
    ([_ (and q1 q2)] (let ((p2' (and q2 q1)))
                       (match (match-props-3 p1 p2' uvars)
                         ((some-sub sub) [sub p2' 'reversed])
                         (_ false))))
```

```
    ([_ (or q1 q2)]  (let ((p2' (or q2 q1)))
                       (match (match-props-3 p1 p2' uvars)
                         ((some-sub sub) [sub p2' 'reversed])
                         (_ false))))
    ([_ (not p2')]
            (match p1 ((not p1') (match-props-AC p1' p2' uvars))
                       (_ false)))
    (_ false)))

(define (is-id? dual-proc)
  (equal? (not true) (dual-proc (not true))))

(define (match-some-component p C uvars kind dual?)
    (match (match-props-AC p (dual? C) uvars)
      ([(some-sub sub) C' rev-flag] [[C' sub rev-flag]])
      (_ (match (check ((is-id? dual?) [])
                       (else (match-props-AC (dual? p) C uvars)))
           ([(some-sub sub) C' rev-flag] [[C' sub rev-flag]])
           (_ (match (get-components C kind)
                ((some-list args) (match-some-component* p args uvars kind dual? []))
                (_ [])))))))
    (match-some-component* p Cs uvars kind dual? results)
     (match Cs
       ([] results)
       ((list-of (some-sent C) rest)
          (match-some-component* p rest uvars kind dual?
                                      (join (match-some-component p C uvars kind dual?) results)))))

(define (match-some-conjunct p C uvars)
   (match (match-some-component p C uvars 'conj (lambda (x) x))
      ([] (let ((p-conjuncts (get-conjuncts-recursive p)))
            (find-element p-conjuncts (lambda (p')
                                         (negate (null? (match-some-component p' C uvars 'conj (lambda (x) x)))))
                (lambda (p')
                  (match-some-component p' C uvars 'conj (lambda (x) x)))
                (lambda () []))))
      (res res)))


(define (match-some-conjunct p C uvars)
    (find-element'
      (get-all-conjuncts p)
      (lambda (L)
        (negate (null? L)))
      (lambda (q)
        (match-some-component q C uvars 'conj id))
      id
      (lambda () [])))

(define (match-some-dual-conjunct p C uvars)
   (match-some-component p C uvars 'conj complement))

(define (match-some-disjunct p C uvars)
   (match-some-component p C uvars 'disj (lambda (x) x)))

(define (match-some-dual-disjunct p C uvars)
   (match-some-component p C uvars 'disj complement))

(define (label-list L label)
  (let ((f (lambda (x) (add label x))))
    (map f L)))

(define (match-antecedent p left q right uvars)
  (let ((L1 (match-some-conjunct q right uvars))
        (Lb (label-list (match-some-conjunct p left uvars) 'conj))
        (Lc (label-list (match-some-disjunct p left uvars) 'disj))
        (L2 (join Lb Lc))
        #(mprint (lambda (x y) (print (join (val->string x) "  " (val->string y)))))
        (mprint (lambda (x y) ())))
    (find-first' (cprod L1 L2)
```

```
      (lambda (pair-res)
        (match pair-res
          ([[right-conjunct _ q-flag]
            [label left-component left-sub  p-flag]]
             (let ((_ ())
                   #(_ (print "\nAbout to try matching " (if (left-sub left-component) q) " against: " (if left-compon
                   (_ ()))
              (match (match-props-3 (if (left-sub left-component) q) (if left-component right-conjunct) uvars)
                ((some-sub sub) (seq (mprint "\nleft-component: " left-component)
                                     (mprint "\nright-conjunct: " right-conjunct)
                                     [sub label [left-component p-flag] [right-conjunct q-flag]]))
                (_ false))))
          (_ false)))
      (lambda () ())))))

(define (apply-dm p q)
  (match p
    ((not (and (some-list _))) (app-dm p))
    ((not (or (some-list _)))  (app-dm p))
    ((and (some-list _))       (app-dm p))
    ((or (some-list _))        (app-dm p))
    (_ (not q))))


(define (subset-matching L1 L2 uvars)
  (letrec ((loop (lambda (remaining sub)
                   (match remaining
                     ([] sub)
                     ((list-of p rest) (find-first' L2
                                          (lambda (q)
                                            (match (match-props-3 (sub p) (sub q) uvars)
                                              ((some-sub sub') (loop rest (compose-subs sub' sub)))
                                              (_ false)))
                                          (lambda () false)))))))
     (loop L1 empty-sub)))


(define (match-consequent p right q left uvars)
   (let ((L1 (match-some-dual-conjunct p right uvars))
         (Lb (label-list (match-some-dual-disjunct q left uvars) 'no-dm))
         (Lc (label-list (match (match-props-AC q (apply-dm (complement left) q) uvars)
                           (false [])
                           ([sub y z] [[y sub z]])) 'dm))
         (L2 (join Lb Lc))
         (mprint (lambda (x y) ())))
    (find-first' (cprod L1 L2)
      (lambda (pair-res)
        (match pair-res
          ([[dual-right-conjunct _ p-flag]
            [dm-flag dual-left-disjunct _ q-flag]]
              (match [dm-flag (match-props-3 (if p q) (if dual-right-conjunct dual-left-disjunct) uvars)]
                (['dm (some-sub sub)] (seq (mprint "\ndual-right-conjunct: " dual-right-conjunct)
                                           (mprint "\ndual-left-disjunct: " dual-left-disjunct)
                                           [sub [(complement dual-right-conjunct) p-flag]
                                                [(complement dual-left-disjunct) q-flag] 'dm]))
                (['no-dm (some-sub sub)] [sub [(complement dual-right-conjunct) p-flag]
                                                [(complement dual-left-disjunct) q-flag]])
                (_ false)))
          (_ false)))
      (lambda ()
        (find-first' L1
          (lambda (res)
             (match res
               ([dual-right-conjunct (some-sub right-sub) p-flag]
                   (let ((La  (right-sub (get-conjuncts-recursive q)))
                         (Lb  (right-sub (get-conjuncts-recursive (apply-dm (complement left) q))))
                         (res-sub (subset-matching La Lb uvars)))
                     (match res-sub
                        ((some-sub _) [(compose-subs res-sub right-sub)
                                       [(complement dual-right-conjunct) p-flag]
```

```
                                      [(right-sub left) 'not-reversed] 'dm])
                        (_ false))))
               (_ false)))
          (lambda () ())))))))

(define (match-consequent p right q left uvars)
   (let ((L1 (match-some-dual-conjunct p right uvars))
         (Lb (label-list (match-some-dual-disjunct q left uvars) 'no-dm))
         (Lc (label-list (match (match-props-AC q (apply-dm (complement left) q) uvars)
                           (false [])
                           ([sub y z] [[y sub z]])) 'dm))
         (L2 (join Lb Lc))
         (mprint (lambda (x y) ())))
    (find-first' (cprod L1 L2)
      (lambda (pair-res)
        (match pair-res
          ([[dual-right-conjunct _ p-flag]
            [dm-flag dual-left-disjunct _ q-flag]]
              (match [dm-flag (match-props-3 (if p q) (if dual-right-conjunct dual-left-disjunct) uvars)]
                (['dm (some-sub sub)] (seq (mprint "\ndual-right-conjunct: " dual-right-conjunct)
                                          (mprint "\ndual-left-disjunct: " dual-left-disjunct)
                                          [sub [(complement dual-right-conjunct) p-flag]
                                               [(complement dual-left-disjunct) q-flag] 'dm]))
                (['no-dm (some-sub sub)] [sub [(complement dual-right-conjunct) p-flag]
                                               [(complement dual-left-disjunct) q-flag]])
                (_ false)))
          (_ false)))
      (lambda ()
        (find-first' L1
          (lambda (res)
             (match res
               ([dual-right-conjunct (some-sub right-sub) p-flag]
                   (let ((La  (right-sub (get-conjuncts-recursive q)))
                         (Lb  (right-sub (get-conjuncts-recursive (apply-dm (complement left) q))))
                         (res-sub (subset-matching La Lb uvars)))
                     (match res-sub
                        ((some-sub _) [(compose-subs res-sub right-sub)
                                      [(complement dual-right-conjunct) p-flag]
                                      [(right-sub left) 'not-reversed] 'dm])
                        (_ false))))
               (_ false)))
          (lambda () ())))))))

(define (mc p q rule)
   (match rule
    ((forall (some-list uvars) (bind conditional (if left right)))
       (match-consequent p right q left uvars))))

(define (ma p q rule)
  (match rule
    ((forall (some-list uvars) (bind conditional (if left right)))
       (match-antecedent p left q right uvars))))


(define (commute? p flag)
  (dmatch flag
    ('reversed (!comm-opt p))
    (_ (!claim p))))

(define (derive-nested-rule R p q)
  (dmatch R
    ((forall (some-list uvars) ((some-sent-con sc) left right))
       (dmatch right
         ((forall (some-list uvars') (if left' right'))
            (dmatch (match-props-3 (if p q) (if left' right') (join uvars' uvars))
              ((some-sub sub) (dlet ((M (match sc
                                          (if claim)
                                          (iff left-iff))))
                                 (!mp (!M (!uspec* R (sub uvars)))
                                      (sub left)))))))))))
```

```

(define (conj-elim' p C)
  (dmatch C
    ((and (some-list args)) (!decompose C (method (_) (!claim p))))
    (_ (dtry (!claim p)
             (!sort-instance C p)))))

(define (make-cond-method p q rule uvars left right M)
  (match (match-antecedent p left q right uvars)
     ([(some-sub sub) 'conj [p-pat rev-flag-p] [q-pat rev-flag-q]]
         (method (_ _) (dlet (
                              (R (method ()
                                   (dseq (!comm-opt p)
                                         (!decompose p (method (_)
                                                          (dlet ((left-side (!conj-intro (sub left))))
                                                             (!mp (!M (!uspec* rule (sub uvars))) left-side))))))))
                         (dmatch rev-flag-q
                           ('reversed (!comm (!conj-elim' (commute q) (!R))))
                           (_ (!conj-elim' q (!R)))))))
     ([(some-sub sub) 'disj rev-flag-left rev-flag-right]
        (method (_ _) (dlet (#(_ (print "p: " p ", q: " q ", left: " left ", right: " right ", uvars: " uvars "\nsub:
                             (inst-rule (!M (!uspec* rule (sub uvars))))
                             (conclusion (!mp inst-rule (!disj-intro (sub left))))
                             #(_ (mprint "\ninst-rule: " inst-rule " conclusion: " conclusion))
                              (_ ()))
                          (dseq (!comm-opt p)
                                (!conj-elim' q conclusion)))))
     (_ (match (match-consequent p right q left uvars)
          ([(some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q]]
             (method (_ _) (dlet ((irule (!M (!uspec* rule (sub uvars))))
                                  ([ileft iright] [(antecedent irule) (consequent irule)])
                                  (_ (conclude (complement iright)
                                       (!complement-conjunction iright (!commute? p rev-flag-p))))
                                  (not-ileft (conclude (complement ileft)
                                               (!mt irule (complement iright)))))
                             (!negate-disjunct not-ileft q))))
            ([(some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q] 'dm]
              (method (_ _) (dlet ((mprint (lambda (x y) (print x y)))
                                   (irule (!M (!uspec* rule (sub uvars))))
                                   ([ileft iright] [(antecedent irule) (consequent irule)])
                                   (iright' (conclude (complement iright)
                                               (!complement-conjunction iright (!commute? p rev-flag-p))))
                                   (ileft' (conclude (complement ileft)
                                              (!mt irule iright')))
                                   (res (!dm-rec ileft')))
                              (dmatch res
                                ((and (some-list _)) (!prove-components-harder q))
                                (_ (!commute? res rev-flag-q))))))
          (_ ())))))

(define make-cond-method-old make-cond-method)

(define (make-cond-method p q rule uvars left right M rule-body)
  (match (match-antecedent p left q right uvars)
     ([(some-sub sub) 'conj [p-pat rev-flag-p] [q-pat rev-flag-q]]
         (method (_ _) (dlet ((R (method ()
                                   (dseq (!comm-opt p)
                                         (!decompose p (method (_)
                                                          (dlet ((left-side (!conj-intro (sub left)))
                                                                 (rule-instance (!uspec* rule (sub uvars)))
                                                                 (rule-instance' (!sort-instance rule-instance (sub rul
                                                             (!mp (!M rule-instance') left-side))))))))
                         (dmatch rev-flag-q
                           ('reversed (!comm (!conj-elim' (commute q) (!R))))
                           (_ (!conj-elim' q (!R)))))))
     ([(some-sub sub) 'disj rev-flag-left rev-flag-right]
        (method (_ _) (dlet (#(_ (print "p: " p ", q: " q ", left: " left ", right: " right ", uvars: " uvars "\nsub:
                             (rule-instance (!uspec* rule (sub uvars)))
                             (rule-instance (!sort-instance rule-instance (sub rule-body)))
                             (inst-rule (!M rule-instance))
```

```
                             (conclusion (!mp inst-rule (!disj-intro (sub left))))
                             #(_ (mprint "\ninst-rule: " inst-rule " conclusion: " conclusion))
                              (_ ()))
                          (dseq (!comm-opt p)
                                (!conj-elim' q conclusion)))))
     (_ (match (match-consequent p right q left uvars)
          ([(some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q]]
             (method (_ _) (dlet ((rule-instance (!uspec* rule (sub uvars)))
                                  (rule-instance (!sort-instance rule-instance (sub rule-body)))
                                  (irule (!M rule-instance))
                                  ([ileft iright] [(antecedent irule) (consequent irule)])
                                  (_ (conclude (complement iright)
                                       (!complement-conjunction iright (!commute? p rev-flag-p))))
                                  (not-ileft (conclude (complement ileft)
                                                (!mt irule (complement iright)))))
                             (!negate-disjunct not-ileft q))))
            ([(some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q] 'dm]
              (method (_ _) (dlet ((mprint (lambda (x y) (print x y)))
                                   (rule-instance (!uspec* rule (sub uvars)))
                                   (rule-instance (!sort-instance rule-instance (sub rule-body)))
                                   (irule (!M rule-instance))
                                   ([ileft iright] [(antecedent irule) (consequent irule)])
                                   (iright' (conclude (complement iright)
                                              (!complement-conjunction iright (!commute? p rev-flag-p))))
                                   (ileft' (conclude (complement ileft)
                                             (!mt irule iright')))
                                   (res (!dm-rec ileft')))
                              (dmatch res
                                ((and (some-list _)) (!decompose res (method (_) (!prove-components-harder q))))
                                (_ (!commute? res rev-flag-q))))))
          (_ ()))))


(define (make-method p q rule)
  (match rule
    ((forall (some-list uvars) (rule-body as (if left right)))
         (make-cond-method p q rule uvars left right claim rule-body))
    ((forall (some-list uvars) (rule-body as (iff left right)))
       (check ((equal? left true)
                  (match (match-props-3 q right uvars)
                    ((some-sub sub) (method (p q)
                                        (!mp (!left-iff (!uspec* rule (sub uvars)))
                                             (!true-intro))))
                    (_ ())))
              (else (let ((p->q (match (make-cond-method p q rule uvars left right left-iff rule-body)
                                 ((some-method M) M)
                                 (_ (make-cond-method p q rule uvars right left right-iff rule-body))))
                          (q->p (match (make-cond-method q p rule uvars left right left-iff rule-body)
                                 ((some-method M) M)
                                 (_ (make-cond-method q p rule uvars right left right-iff rule-body)))))
                      (match [(method? p->q) (method? q->p)]
                        ([false false] ())
                        (_ (method (premise goal)
                             (dcheck ((equal? premise p) (!p->q premise goal))
                                     (else (!q->p premise goal))))))))))
    ((some-atom A)  (match p ## p is the identity here, so it can be ignored.
                       ((= _ _) (method (_ q)
                                  (!rec-rel-cong A q)))
                       (_ ())))
    ((not (some-atom A))
       (match p
         ((= _ _)
           (method (_ q)
             (dmatch q
               ((not (some-atom B))
                  (!by-contradiction q
                    (assume B
                       (dlet ((res (!rec-rel-cong B A)))
                          (!absurd res rule))))))))
         (_ ()))))
```

```
    (_ ())))

(define (augment-methods methods rules p q)
  (let ((f (lambda (rule) (make-method p q rule))))
    (join (filter (map f rules) method?) methods)))

(define (try-implication-method M p q)
  (conclude (if p q)
    (dtry (conclude (if p q)
            (assume p
              (!M p)))
          (conclude (if p q)
            (!M (if p q)))
          (conclude (if p q)
            (assume p
              (!M p q))))))

(define (try-equivalence-method M p q)
  (dtry (conclude (iff p q)
          (!M (iff p q)))
        (conclude (iff p q)
          (!equiv (assume p (!M p))
                  (assume q (!M q))))
        (conclude (iff p q)
          (!equiv (assume p (!M p q))
                  (assume q (!M q p))))))

(define (get-implication-structurally p q rules methods)
  (!find-some (augment-methods methods rules p q)
    (method (M) (!try-implication-method M p q))
    (method ()
      (dmatch [p q]
        ([_ (val-of p)] (assume p (!claim q)))
        ([(= (some-term s) (some-term t))
          (= ((some-symbol f) s) (f t))]
            (assume p
              (!fcong q)))
        ([(some-atom _) (some-atom _)]
          (assume p
            (dtry (!equate-subterms p q rules
                     (method (identities)
                       (!map-method sym identities
                          (method (_)
                           (dtry (!rcong p q)
                                 (!mp (!left-iff (!equality-to-equivalence (= p q))) p))))))
                  (!equate-atoms p q rules))))
        ([(and p1 p2) (and q1 q2)]
          (!uni-and-cong (!get-implication-structurally p1 q1 rules methods)
                         (!get-implication-structurally p2 q2 rules methods)))
        ([(and (some-list props1)) (and (some-list props2))]
           (!map-method (method (pair)
                          (dmatch pair
                            ([p1 q1] (!get-implication-structurally p1 q1 rules methods))))
                        (list-zip props1 props2)
                        (method (results)
                           (!uni-and-cong* results))))
        ([(forall (some-var v1) (some-sentence p1)) (forall (some-var v2) (some-sentence p2))]
            (assume p
              (pick-any x2
                 (dlet ((step-1 (!uspec p x2))
                        (step-2 (!get-implication-structurally step-1 (replace-var v2 x2 p2) rules methods)))
                   (!mp step-2 step-1)))))
        ([(exists (some-var v1) (some-sentence p1)) (exists (some-var v2) (some-sentence p2))]
            (assume hyp := p
               (pick-witness w hyp witness-premise
                 (dlet ((step-1 (!get-implication-structurally witness-premise (replace-var v2 w p2) rules methods))
                        (body (!mp step-1 witness-premise)))
                   (!egen q w)))))
        ([(or p1 p2) (or q1 q2)]
          (!uni-or-cong (!get-implication-structurally p1 q1 rules methods)
```

```
                        (!get-implication-structurally p2 q2 rules methods)))
        ([(or (some-list props1)) (or (some-list props2))]
           (!map-method (method (pair)
                          (dmatch pair
                            ([p1 q1] (!get-implication-structurally p1 q1 rules methods))))
                        (list-zip props1 props2)
                        (method (results)
                           (!uni-or-cong* results)))))))

(define (get-equivalence-structurally p q rules methods)
  (dletrec ((loop (method (p q)
                    (!find-some (augment-methods methods rules p q)
                      (method (M) (!try-equivalence-method M p q))
                       (method ()
                         (dmatch [p q]
                           ([_ (val-of p)] (!ref-equiv p))
                           ([(some-atom _) (some-atom _)]
                             (!equate-subterms p q rules
                               (method (identities)
                                 (!map-method sym identities
                                   (method (_)
                                     (dtry (!equiv (assume p
                                                      (!rcong p q))
                                                   (assume q
                                                      (!rcong q p)))
                                           (!equality-to-equivalence (= p q))))))))
                           ([(not p1) (not q1)] (!not-cong (!loop p1 q1)))
                           ([((some-sent-con pc) p1 p2) (pc q1 q2)]
                             (!(choose-cong-method pc)
                               (!loop p1 q1) (!loop p2 q2)))
                           ([(forall (some-var v1) body1) (forall (some-var v2) body2)]
                             (!ugen-cong p q loop))
                           ([(exists (some-var v1) body1) (exists (some-var v2) body2)]
                             (!egen-cong p q loop))))))))
    (!loop p q)))

(define (get-implication-through-structure p q rules methods)
  (dtry (!get-implication-structurally p q rules methods)
        (dlet ((p<==>q (!get-equivalence-structurally p q rules methods)))
          (assume p
            (!mp (!left-iff p<==>q) p)))))


# Blocking this out so as to move get-equivalence lower, after get-implication,
# since this meth-trans is hardly ever used, it seems.

#(define (meth-trans p1 p2 methods rules-or-methods max)
#  (!breadth-first p1 p2 methods max
#    (method () (!left-iff (!get-equivalence p1 p2 rules-or-methods)))))

(define (meth-trans p1 p2 methods rules-or-methods max)
  (dlet (#(_ (mark 'H))
          (_ ()))
    (!fail "")))

(define (sentential-components-of p)
  (match p
    ((and (some-list _)) (get-conjuncts p))
    ((or (some-list _)) (get-disjuncts p))
    ((not q) [q])
    (_ [p])))


(define (derive-nested-rule' R p q)
  (dmatch R
    ((forall (some-list uvars) ((sc as (|| if iff)) left (forall (some-list uvars') ((|| if iff) left' right'))))
      (dtry (dmatch (match-props-3 (if p q) (if left' right') (join uvars' uvars))
               ((some-sub sub) (dlet ((M (match sc
                                            (if claim)
                                            (iff left-iff))))
```

```
                                   (!mp (!M (!uspec* R (sub uvars)))
                                       (sub left)))))
             (dlet ((all-vars (join uvars' uvars))
                    (V (join (constants&vars p) (constants&vars q))))
               (!find-some
                   (ab)
                   (method (fact)
                      (dmatch (match-props-3 fact left all-vars)
                         (((some-sub sub) where (non-null (fast-intersection (constants&vars fact) V)))
                              (dlet ((M (match sc
                                          (if claim)
                                          (iff left-iff))))
                                (!mp (!M (!uspec* R (sub uvars)))
                                     (sub left))))))
                    (method () (!proof-error ""))))))))


(define (derive-nested-rule-new R p q)
  (dmatch R
    ((forall (some-list uvars) ((sc as (|| if iff)) left (forall (some-list uvars') ((|| if iff) left' right'))))
        (dtry (dmatch (match-props-3 (if p q) (if left' right') (join uvars' uvars))
                 ((some-sub sub) (dlet ((M (match sc
                                              (if claim)
                                              (iff left-iff))))
                                     (!mp (!M (!uspec* R (sub uvars)))
                                         (sub left)))))
               (dlet ((all-vars (join uvars' uvars))
                      (V (join (constants&vars p) (constants&vars q))))
                 (!map-method-non-strictly
                     (method (fact)
                        (dmatch (match-props-3 fact left all-vars)
                           (((some-sub sub) where (non-null (fast-intersection (constants&vars fact) V)))
                              (dlet ((M (match sc
                                            (if claim)
                                            (iff left-iff))))
                                  (!mp (!M (!uspec* R (sub uvars)))
                                      (sub left))))))
                      (ab)
                      (method (rules)
                        (dmatch rules
                          ([]  (!proof-error ""))
                          ([(some-sent res)] (!claim res))
                          ((list-of _ (list-of _ _)) (!conj-intro rules)))))))))))

(define (get-implication-new p q rules-or-methods)
  (dlet (([methods rules] (filter-and-complement rules-or-methods method?)))
     (!map-method-non-strictly
        fact->bicond
        rules
        (method (bc-rules)
          (!map-methods-non-strictly [(method (R) (!derive-nested-rule-new R p q))] rules
             (method (new-rules)
                (!decompose* new-rules
                    (method (new-rules')
                      (dlet ((rules' (join new-rules' bc-rules rules))
                             (methods' (augment-methods methods rules' p q)))
                       (!find-some methods'
                          (method (M) (!try-implication-method M p q))
                          (method ()
                            (dtry (conclude (if p q)
                                    (!get-implication-through-structure p q rules' methods'))
                                  (!meth-trans p q methods rules-or-methods 7)))))))))))))

(define (get-implication p q rules-or-methods)
  (dlet (([methods rules] (filter-and-complement rules-or-methods method?))
        )
     (!map-methods-non-strictly [fact->bicond (method (R) (!derive-nested-rule' R p q))] rules
      (method (new-rules)
        (dlet ((rules' (join new-rules rules))
               (methods' (augment-methods methods rules' p q)))
```

```
          (!find-some methods'
            (method (M) (!try-implication-method M p q))
            (method ()
              (dtry (conclude (if p q)
                       (!get-implication-through-structure p q rules' methods'))
                    (!meth-trans p q methods rules-or-methods 7)))))))))

(define get-implication get-implication-new)

(define (get-equivalence p q rules-or-methods)
  (dlet (([methods rules] (filter-and-complement rules-or-methods method?)))
    (!map-methods-non-strictly [fact->bicond (method (R) (!derive-nested-rule R p q))] rules
       (method (new-rules)
         (dtry (dlet ((rules' (join new-rules rules))
                      (methods' (augment-methods methods rules' p q)))
                  (!get-equivalence-structurally p q rules' methods'))
               (!equiv (!get-implication p q rules-or-methods)
                       (!get-implication q p rules-or-methods)))))))

(define (get-equivalence* p q rules-or-methods)
  (dlet ((writeln-val (lambda (x y) ()))
         (_ (writeln-val "p: " p))
         (_ (writeln-val "q: " q)))
    (dtry (!get-equivalence p q rules-or-methods)
          (dlet ((_ (print "\nCalling external ATP for equivalence...\n")))
            (!vpf (iff p q) (filter rules-or-methods (lambda (x) (negate (method? x)))))))))


(define (get-equivalence* p q rules-or-methods)
  (dlet ((writeln-val (lambda (x y) ()))
         (_ (writeln-val "p: " p))
         (_ (writeln-val "q: " q)))
    (dcheck ((get-boolean-flag "atps-with-chain")
              (!vprove-from (iff p q) (filter rules-or-methods (lambda (x) (negate (method? x))))
                                      [['poly true] ['subsorting false] ['max-time 160]]))
            (else (!get-equivalence p q rules-or-methods)))))

#   (!thread-methods [(method () (!get-equivalence p q rules-or-methods))
#                     (method () (!derive-theorem (iff p q) (filter rules-or-methods (lambda (x) (negate (method? x)))
#          (!fail "About to call external ATP, failing instead...\n"))))

(define (get-implication* p q rules-or-methods)
  (dlet ((writeln-val (lambda (x y) ()))
         (_ (writeln-val "p: " p))
         (_ (writeln-val "q: " q)))
    (dtry (!get-implication p q rules-or-methods)
          (dlet ((_ (print "\nCalling external ATP for implication...\n")))
            (!vpf (if p q) (filter rules-or-methods (lambda (x) (negate (method? x)))))))))




(define (get-implication* p q rules-or-methods)
  (dlet ((writeln-val (lambda (x y) ())))
    (dcheck ((get-boolean-flag "atps-with-chain")
              (dtry
                  (!vprove-from (if p q) (filter rules-or-methods (lambda (x) (negate (method? x))))
                                     [['poly true] ['subsorting false] ['max-time 160]])
                  (!get-implication p q rules-or-methods)))
            (else (!get-implication p q rules-or-methods)))))

#    (!thread-methods [(method () (!get-implication p q rules-or-methods))
#                      (method () (!derive-theorem (if p q) (filter rules-or-methods (lambda (x) (negate (method? x)))
#           (!fail "About to call external ATP, failing instead...\n"))))

(define gi get-implication)
(define gi* get-implication*)

(define (get-list L)
  (letrec ((loop (lambda (L results)
```

```
                   (match L
                     ([] (rev results))
                     ((list-of (some-list L1) more) (loop more (join L1 results)))
                     ((list-of x more)  (loop more (add x results)))))))
    (loop L [])))

(define (make-list x)
  (match x
    ((some-list L) (get-list L))
    (_ [x])))

(define (rcong-rec premise goal)
  (dmatch [premise goal]
    ([((some-symbol R) (some-list args1)) (R (some-list args2))]
      (dletrec ((loop (method (args1 args2)
                        (dmatch [args1 args2]
                          ([[] []] (!rcong premise goal))
                          ([(list-of s rest1) (list-of t rest2)]
                            (dseq (!rec-cong s t)
                                  (!loop rest1 rest2)))))))
        (!loop args1 args2)))))

(define (equational-step current-theorem left right L direction new-object rest
                         get-rules no-rules-given? iteration show-info)
  (dlet (#(_ (print "\nCurrent theorem: " current-theorem " and new-object: " new-object))
         ([old-term old-term' diff?] (match [right (atom? right) (identity? current-theorem)]
                                       ([(= _ (some-term t)) _ _]
                                           [t right true])
                                       ([((some-symbol _) (bind L (list-of _ _))) true false]
                                          [(first (rev L)) right true])
                                       ([((some-symbol _) (bind L (list-of _ _))) true true]
                                          [right right false])
                                       (_ [right right false])))
         ([old-term new-object] (match (= old-term new-object)
                                  ((= (some-term L) (some-term R)) [L R])))
         (rules (check (no-rules-given? (ab)) (else (make-list (head rest)))))
         (rule-cell (cell rules))
         (_ (dcheck (no-rules-given? (!true-intro)) (else (!get-rules rules rule-cell))))
         (rules' (ref rule-cell))
         (rules'' (map rename' rules'))
         (_ (show-info current-theorem old-term new-object direction iteration))
         (tran' (check ((equal? iteration 0) sorted-tran) (else tran)))
         (new-equation (!map-multi-method decompose rules''
                         (method (results)
                          (dlet (#(_ (print "\nAbout to rewrite*** with the following rules: " results))
                                  (_ ()))
                           (dtry (!rewrite*** old-term new-object results direction)
                                 (dcheck (diff? (!rewrite*** old-term' new-object results direction))
                                         (else (!fail)))
                                 (dlet ((eval-res (!(ref deval-cell) old-term))
                                        # (eval-res (!reflex old-term))
                                        #(_ (!fail))
                                        )
                                   (dcheck ((equal? (rhs eval-res) new-object) (!claim eval-res))
                                           (else (!fail))))))))))
    (dmatch current-theorem
      ((= _ _) (!tran' current-theorem new-equation))
      ((some-atom A) (dlet ((terms (children A))
                            (terms' (join (all-but-last terms) [new-object]))
                            (new-th (make-term (root A) terms')))
                       (!rcong-rec current-theorem new-th)))
      ((if ant (= _ _)) (assume ant
                          (!tran' (!mp current-theorem ant) new-equation)))
      ((if ant (bind A ((some-symbol f) (list-of _ _))))
        (dlet ((terms' (join (all-but-last (children A)) [new-object]))
               (new-th (make-term (root A) terms')))
          (assume ant
            (!rcong (!mp current-theorem ant) new-th))))))
```

```
(define (first-iteration? n)
  (equal? n 0))

(define (show-info writer)
  (lambda (ct ot no di it)
    (seq (writer "current-theorem: " ct)
         (writer "old-term: " ot)
         (writer "new-object " no)
         (show ot no di (first-iteration? it)))))

# (define (infer p1 p2 rules direction)
#   (dmatch direction
#     (<==> (!get-equivalence* p1 p2 rules))
#     (==>  (!get-implication* p1 p2 rules))
#     (<==  (!get-implication* p2 p1 rules))
#     (_    (!fail (join "Error: " (val->string direction)
#                        " found where <==>, ==>, or <== was expected.")))))

(define (infer p1 p2 rules direction)
  (dcheck ((binary-proc? direction) (dmatch (direction true false)
                                      ((iff true false) (!get-equivalence* p1 p2 rules))
                                      ((if true false)  (!get-implication* p1 p2 rules))
                                      ((if false true)  (!get-implication* p2 p1 rules))
                                      (_    (!fail (join "Error: " (val->string direction)
                                                         " found where <==>, ==>, or <== was expected.")))))
          (else (!fail (join "Error: " (val->string direction)
                                   " found where <==>, ==>, or <== was expected.")))))


(define (infer p1 p2 rules direction)
  (dmatch direction
    (if (!get-implication* p1 p2 rules))
    (iff  (!get-equivalence* p1 p2 rules))
    (_ (dcheck ((binary-proc? direction) (dmatch (direction true false)
                                           ((iff true false) (!get-equivalence* p1 p2 rules))
                                           ((if true false)  (!get-implication* p1 p2 rules))
                                           ((if false true)  (!get-implication* p2 p1 rules))
                                           (_    (!fail (join "Error: " (val->string direction)
                                                              " found where <==>, ==>, or <== was expected.")))))
                (else (!fail (join "Error: " (val->string direction)
                                                         " found where <==>, ==>, or <== was expected.")))))))

(define (implication-step current-theorem side new-object direction
                          no-rules-given? rest get-rules iteration wv)
  (dlet ((rules (check (no-rules-given? (ab))
                       (else (make-list (head rest)))))
         (rule-cell (cell rules))
         (_ (dcheck (no-rules-given? (!true-intro))
                    (else (!get-rules rules rule-cell))))
         (rules' (ref rule-cell))
         (rules'' (map rename' rules'))
         ([side new-object] (match (if side new-object)
                              ((if (some-sent l) (some-sent r)) [l r])))
         (new-theorem (!map-multi-method decompose rules''
                        (method (results)
                          # Add the following if you want prop-taut to be included by default when no rules are specif
                          (dlet ((results' (check (no-rules-given? (add prop-taut results))
                                                  (else results))))
                             (!infer side new-object results' direction)))))
         (_ (wv "current-theorem: " current-theorem))
         (_ (wv "side: " side))
         (_ (wv "new-object " new-object))
         (_ (wv "direction: " direction))
         (_ (wv "new-theorem " new-theorem))
         ([current-theorem' new-theorem'] (match (and current-theorem new-theorem)
                                            ((and (some-sent p1) (some-sent p2)) [p1 p2])))
         (_ (show-equiv side new-object direction (first-iteration? iteration)))
         (_ (!sort-instance current-theorem current-theorem')))
    (!apply-tran current-theorem' new-theorem')))
```

```
(define (do-end current-theorem end-choice wv)
  (dmatch current-theorem
   ((bind th (iff p1 p2)) (dmatch end-choice
                           ('last (!mp (!left-iff th) (!prove-components-harder p1)))
                           ('first (!mp (!right-iff th) (!prove-components-harder p2)))
                           (_ (!claim th))))
   ((bind th (if p1 _))  (dmatch end-choice
                          ('last (dlet ((_ (wv "th: " th))
                                        (_ (wv "" (join "\nWill try to prove components of: "
                                                        (val->string p1)))))
                                  (!mp th (!prove-components-harder p1))))
                          ('first (!mp th (!prove-components-harder p1)))
                          (_ (!claim th))))
   (_ (!claim current-theorem))))

(define (get-all-rules generic-chain)
   (method (L list-cell)
     (dcheck ((for-some L chain-symbol?)
               (dlet ((th (!generic-chain L 'last))
                     (_  (set! list-cell [th])))
                 (!claim th)))
             (else (dlet ((_ (set! list-cell L)))
                     (!true-intro))))))

(define (backward-implication-direction? dir)
  (&& (binary-proc? dir)
      (equal? (dir true false) (if false true))))


(define (ordinal-string i)
  (let ((suffix (match (mod i 10)
                 (1 (check ((equal? i 11) "th")
                           (else "st")))
                 (2 "nd")
                 (3 "rd")
                 (_ "th"))))
    (join (val->string i) suffix)))


(define (dcatch' M1 M2)
  (dlet ((failed (cell false))
         (res (dtry (!M1)
                    (dlet ((_ (set! failed true)))
                      (!true-intro)))))
    (dmatch (ref failed)
      (true (!M2))
      (_ (!claim res)))))


(define (generic-chain given-list end-choice)
  (dletrec ((writeln-val (lambda (x y) (print x  "\n" (val->string y) "\n")))
            (writeln-val (lambda (x y) ()))
            #(mprint (lambda (x) (print x)))
            (mprint (lambda (x) ()))
            (continue (lambda () ()))
            (_ (check ((in-rewriting-trace-mode) (seq (indent (plus (get-trace-level) 1))
                                                      (print "Chaining:\n")))
                      (else ())))
            (get-rules (get-all-rules generic-chain))
            (loop (method (current-theorem L backwards? iteration)
                    (dmatch [current-theorem L]
                      ([(_ left right) (split [direction new-object] rest)]
                        (dcheck ((equational? direction)
                                  (dlet ((no-rules-given? (|| (null? rest) (direction? (head rest))))
                                         (rest' (check (no-rules-given? rest) (else (tail rest))))
                                         (current-theorem' (!dcatch'
                                                              (method () (!equational-step current-theorem left right 
                                                                           (restrict-right-hand-side-sorts rig
                                                                          get-rules no-rules-given? iteration
                                                              (method () (!proof-error (join "Equational chaining erro
```

```
                                                                                       " step of the chain, in going fro
                                                                                       "\nto:\n" (val->string new-object
                                     (!loop current-theorem' rest' backwards? (plus iteration 1))))
                                 (else (dlet ((side (check ((identity? current-theorem) current-theorem)
                                                           (backwards? left)
                                                           (else right)))
                                              (finished? (null? rest))
                                              (no-rules-given? (|| finished? (direction? (head rest))))
                                              (rest' (check (no-rules-given? rest)
                                                            (else (tail rest))))
                                              (_ (mprint (join "\ncurrent-theorem: " (val->string current-theorem)
                                                               "\nside: " (val->string side) "\nnew-object: " (val->str
                                                               "\nno-rules-given?: " (val->string no-rules-given?)
                                                               "\nrest: " (val->string rest) "\niteration: " (val->stri
                                              (new-current-theorem (!dcatch'
                                                                      (method ()
                                                                         (!implication-step current-theorem side
                                                                            (restrict-right-hand-side-sorts side new-obj
                                                                         no-rules-given? rest get-rules iteration writel
                                                                      (method () (dlet ((previous (match (&& (term? righ
                                                                                                   (true right)
                                                                                                   (_ (match current-theorem
                                                                                                        ((iff LBS RBS) RBS)
                                                                                                        (_ current-theorem))))))
                                                                         (!proof-error (join "Implicational chaining err
                                                                                             " step of the chain, in goi
                                                                                             "\nto:\n" (val->string new-
                                              (backwards (|| backwards? (backward-implication-direction? direction))))
                                         (!loop new-current-theorem rest' backwards (plus iteration 1))))))
                       (_ (!do-end current-theorem end-choice writeln-val))))))
      (dmatch given-list
        ((list-of first-object (bind rest (list-of dir _)))
          (dcheck ((equational? dir) (!loop (!reflex first-object) rest false 0))
                  (else (dlet ((_ ())) (!loop (!ref-equiv first-object) rest false 0))))))))

(define (chain-last L)
   (!generic-chain L 'last))

(define (chain-first L)
   (!generic-chain L 'first))

(define (chain L)
   (!generic-chain L 'none))

(define [chain-> chain<-] [chain-last chain-first])
(define [c-> c<-] [chain-last chain-first])

define chain-transformer :=
  method (sentence-transformer L modifier)
    letrec {insert-given := lambda (x)
                              match x {
                                (some-sent _) => try {(sentence-transformer x) | x}
                              | (some-method _) => x
                              | (some-list L) => (map insert-given L)
                              | _ => (sentence-transformer x)
                              };
            insert-givens :=
              lambda (L)
                match L {
                  (list-of direction (list-of y (list-of x rest))) =>
                    (join [direction y (insert-given x)]
                          (insert-givens rest))
                | [] => []
                }
            }
    match L {
      (list-of h rest) => let {L' := (add h (insert-givens rest))}
                             (!generic-chain L' modifier)
    }
```

```
(define (left-side equivalence)
  (dmatch equivalence
    ((bind th (iff _ p2)) (!mp (!right-iff th) (!prove-components-of p2)))))

(define (right-side equivalence)
  (dmatch equivalence
    ((bind th (iff p1 _)) (!mp (!left-iff th) (!prove-components-of p1)))
    ((bind th (if p1 _))  (!mp th (!prove-components-of p1)))))

#### Some more examples of new uses of chain:
#
#### Example 1: Here p2 is obtainable from p1 via double
#### negation, as well as rewriting with Plus-zero-axiom
#### and Times-Commutativity:
#
# (define p1 (not (not (< (Plus ?x zero) (Times ?z1 ?z2)))))
#
# (define p2 (< ?x (Times ?z2 ?z1)))
#
# (!chain [p1 <==> p2 [bdn Plus-zero-axiom Times-Commutativity]])
#
#### Example 2: Here p2 is obtainable from p1 by commuting
#### the conjuncts of p1, and also rewriting various subterms
#### in accordance with the commutativity of times and the
#### plus-zero axiom:
#
# (define p1 (or (not true) (and (< (Plus ?x zero) ?y)
#                                (pos (Times ?z1 ?z2)))))
#
# (define p2 (or (not true) (and (pos (Times ?z2 ?z1))
#                                (< ?x ?y))))
#
# (!chain [p1 <==> p2 [comm Plus-zero-axiom Times-Commutativity]])
#
#### Example 3: Similar to example 1, but uni-directional:
#
# (define p1 (and (= ?foo ?goo)
#                 (not (not (< (Plus ?x zero)
#                              (Times ?z1 ?z2))))))
#
# (define p2 (< ?x (Times ?z2 ?z1)))
#
# (!chain [p1 ==> p2 [bdn right-and Plus-zero-axiom Times-Commutativity]])
#
#### Example 4: Backward goal reduction:
##
## (!chain [?A <== (and ?A ?B)           [left-and]
##             <== (and ?C (and ?A ?B))  [right-and]])
##
#### Example 5: Quantified propositions:
##
## (define p (forall ?x (and (= ?x (Plus ?foo ?goo))
##                           (= (Plus ?x zero) ?y))))
##
## (define q (forall ?w (and (= ?w (Plus ?goo ?foo))
##                           (= ?w ?y))))
##
## (!chain [p <==> q [Plus-Commutativity Plus-zero-axiom]])
## (!chain [p ==> q  [Plus-Commutativity Plus-zero-axiom]])
## (!chain [p <== q  [Plus-Commutativity Plus-zero-axiom]])
##
## These examples would work with 'exists' in place of 'forall'
## as well.
##################################################################

(define (test-proof p)
  (dlet ((dummy (write p)))
     (!true-intro)))
```

```
(define (test-proof p)
  (seq
    (write p)
    (print "Proved!")))

(define (spec proposition terms)
  (dlet ((spec-proposition (!uspec* proposition terms)))
    (dmatch spec-proposition
      ((if _P _Q)
       (!mp spec-proposition
            _P))
      ((iff _P _Q)
       (!mp (!left-iff spec-proposition)
            _P))
      (_ (!claim spec-proposition)))))

(define (spec-right proposition terms)
  (dlet ((spec-proposition (!uspec* proposition terms)))
    (dmatch spec-proposition
      ((iff _P _Q)
       (!mp (!right-iff spec-proposition)
            _Q)))))

# New names for same methods (old spec names are kept temporarily
# for backward compatibility):

## It turns out that the method instance was similar to fire, in util.ath.
## I think the name 'instance' is more appropriate than 'fire', so I have
## been using 'instance' instead of 'fire', having defined 'instance' as
## 'fire' in util.ath (since 'fire' is a slight generalization of 'instance').

# (define (instance proposition terms)
#   (dlet ((instance-proposition (!uspec* proposition terms)))
#     (dmatch instance-proposition
#       ((if _P _Q)
#        (!mp instance-proposition
#           _P))
#       ((iff _P _Q)
#        (!mp (!left-iff instance-proposition)
#           _P))
#       (_ (!claim instance-proposition)))))

(define (right-instance proposition terms)
  (dlet ((instance-proposition (!uspec* proposition terms)))
    (dmatch instance-proposition
      ((iff _P _Q)
       (!mp (!right-iff instance-proposition)
            _Q)))))

(define (left-instance proposition terms)
  (dlet ((instance-proposition (!uspec* proposition terms)))
    (dmatch instance-proposition
      ((iff _P _Q)
       (!mp (!left-iff instance-proposition)
            _P)))))

# A similar method, for use where you want to be explicit
# about the assumption used to discharge the antecedent of the
# specialized implication or equivalence.

(define (mp-instance proposition terms assumption)
  (dlet ((instance-proposition (!uspec* proposition terms)))
    (dmatch instance-proposition
      ((if _P _Q)
       (!mp instance-proposition
            (dcheck ((equal? _P assumption) (!claim _P))
                    (else (dlet ((dummy
                                  (write "Error: Failed application of mp-instance, due to:")))
                             (!claim assumption))))))
      ((iff _P _Q)
```

```
      (!mp (!left-iff instance-proposition)
           (dcheck ((equal? _P assumption) (!claim _P))
                   (else (dlet ((dummy
                                 (write "Error: Failed application of mp-instance, due to:")))
                           (!claim assumption))))))
     (_ (!claim instance-proposition)))))


# A method that subsumes uspec and uspec*

(define (special-case P terms)
  (dmatch terms
    ([] (!claim P))
    ((list-of t more-terms) (!special-case (!uspec P t) more-terms))
    (_ (!uspec P terms))))

# Methods for which the first argument is the desired special case,
# or an antecendent or consequent of the desired special case,
# and the second argument is the property to be specialized.  The
# instantiation is deduced by unification.

(define (left-extract P1 P)
  (dmatch (universal-quantifierless P)
    ((iff _A _B) (dlet ((subst (unify-props _B P1))
                        (term-list (universal-quantifiers P)))
                   (!mp (!left-iff (!special-case P (subst term-list)))
                        (!prove-components-of (subst _A)))))))

(define (right-extract P1 P)
  (dmatch (universal-quantifierless P)
    ((iff _A _B) (dlet ((subst (unify-props _A P1))
                        (term-list (universal-quantifiers P)))
                   (!mp (!right-iff (!special-case P (subst term-list)))
                        (!prove-components-of (subst _B)))))))

(define (extract P1 P)
  (conclude P1
    (dmatch (universal-quantifierless P)
      ((if _A _B)
       (dlet ((subst (match-props P1 _B)))
         (dcheck ((negate (equal? subst false))
                  (dlet ((term-list (universal-quantifiers P)))
                    (!mp (!special-case P (subst term-list))
                         (!prove-components-of (subst _A))))))))
      (_P
       (dtry
        (!left-extract P1 P)
        (!right-extract P1 P)
        (dlet ((subst (match-props P1 _P)))
          (dcheck ((negate (equal? subst false))
                   (!special-case P (subst (universal-quantifiers P))))))))))

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

(define tracing (cell false))

(define (conclude-old prop)
  (seq
    (check ((ref tracing)
            (seq
              (set! level (plus (ref level) 1))
              (indent (ref level))
              (print "Proving at level ")
              (write-val (ref level))
              (print ":\n")
              (indent (ref level))
              (write-val prop)
              (print "\n")))
           (else ()))
    (method (proof)
```

```
      (dseq
       (prop BY (!claim proof))
       (dlet ((dummy
               (check ((ref tracing)
                       (seq
                         (indent (ref level))
                         (print "Done at level ")
                         (write-val (ref level))
                         (print "\n")
                         (set! level (minus (ref level) 1))))
                      (else ()))))
         (!claim prop))))))

# To turn on tracing of concludes, enter
#   (set-debug-mode "conclude")
# To turn it off, enter
#   (set-debug-mode "off")

# To turn on tracing of rewrite rule applications (and concludes):
#   (set-debug-mode "rewriting")
# To turn it off, enter
#   (set-debug-mode "off")
# or, if you still want concludes traced:
#   (set-debug-mode "conclude")

# For legacy code that uses deduce:
(define (trace)
  (seq
   (set! tracing true)
   (set! level 0)))
# Also at top-level do
#   (set-debug-mode "rewriting")

# The following redefines rel-cong so that it
# doesn't need to have equations of the form (= x x) in the
# assumption base (to use the built-in rel-cong you must
# do (!equality x x) for terms x that appear unchanged in
# corresponding positions of s-terms and t-terms)

(define (rel-cong P s-terms t-terms)
  (dletrec ((do-args (method (s-terms t-terms theorem n)
                       (dmatch [s-terms t-terms]
                         ([[] []] (!claim theorem))
                         ([(list-of s more-s) (list-of t more-t)]
                          (dlet ((new-theorem
                                  (!pos-substitute-equals s theorem [n] t)))
                            (!do-args more-s more-t new-theorem (plus n 1))))))))
    (!do-args s-terms t-terms P 1)))

# Using this rel-cong, we define (rewrite P1 P2) where P1 and P2
# are predicates with the same predicate symbols and same number of
# arguments, and P1 is in the assumption base.  If for each argument
# position with corresponding terms s and t the equation (= s t) is in the
# assumption base, then P2 is entered as a theorem in the assumption base.

(define (rewrite P1 P2)
  (dletrec
      ((loop (method (L1 L2 theorem n)
               (dmatch [L1 L2]
                 ([(list-of x1 rest1) (list-of x2 rest2)]
                  (dcheck ((negate (equal? x1 x2))
                           (dlet ((new-theorem
                                   (!pos-substitute-equals x1 theorem [n] x2)))
                             (!loop rest1 rest2 new-theorem (plus n 1))))
                          (else (!loop rest1 rest2 theorem (plus n 1)))))
                 (_ (!claim theorem))))))
    (!loop (children P1) (children P2) P1 1)))

(define cl chain-last)
```

```
(define (dt=? s t)
  (dmatch [s t]
     ([((some-symbol f1) (some-list terms1))
       ((some-symbol f2) (some-list terms2))]
         (dcheck ((for-each [f1 f2] constructor?)
                   (dcheck ((equal? f1 f2)
                            (dletrec ((loop
                                       (method (L)
                                         (dmatch L
                                           ([] (!fcong (= s t)))
                                           ((list-of term-pair rest)
                                            (dmatch (!dt=? (first term-pair) (second term-pair))
                                              ((= _ _)  (!loop rest))
                                              (res (!chain-last [res ==> (s =/= t) [(datatype-axioms (sort-of s))]]))))))))
                               (!loop (zip terms1 terms2))))
                            (else (dlet ((s-sort (sort-of s))
                                         (dt-axioms (datatype-axioms s-sort))
                                         (_ (check ((hold? dt-axioms) ())
                                                   (else (print "\nWarning: the datatype axioms for " s-sort
                                                                 "are not in the assumption base.\n")))))
                                    (dtry (!chain-last [true ==> (s =/= t) [dt-axioms]])
                                          (!chain-last [true ==> (t =/= s) [dt-axioms]
                                                                   ==> (s =/= t) [unequal-sym]]))))))
                  ((&& (real-numeral? s) (real-numeral? t)) (!real-comp s t))
                  ((&& (meta-id? s) (meta-id? t)) (!id-comp s t))))))

# The dt-comp-method-cell is a cell defined in util.ath right before the
# definition of prove-components-of and uses whatever method is inside that
# cell to prove equalities and inequalities. By setting the contents of
# that cell to dt=?, we make dt=? the default method for proving equalities
# and inequalities in antecedents of conditional equations.

(define (dt-comp p)
  (dmatch p
    ((= (some-term s) (some-term t))
       (dcheck ((&& (super-canonical? s) (super-canonical? t)) (!dt=? s t))
               ((all-ground? [s t]) (!chain [s = t]))
               (else (!dt=? s t))))
    ((not (= (some-term s) (some-term t))) (!dt=? s t))
    ((not (some-term s)) (dcheck ((ground? s) (!identity->atom (!chain [s = false])))))))


(set! dt-comp-method-cell
        (method (p)
           (dmatch p
             ((= (some-term l) (some-term r)) (!dt=? l r)))))

(set! dt-comp-method-cell dt-comp)


(define (dt-comp' _ p)
  (!dt-comp p))

(define dt-comps [dt-comp dt-comp'])

#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#                                    TERM EVALUATION CODE
#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define decompose-equation'
  (lambda (eqn)
    (match (rename eqn)
      ((forall (some-list uvars) (if guard (= pattern res))) [pattern guard res])
      ((forall (some-list uvars) (= pattern res)) [pattern () res]))))

(define pat-of first)
(define guard-of second)
(define res-of third)
```

```
(define (unifiable x y)
  (try (match (unify* x y)
         ((some-sub _) true)
         (_ false))
       false))

(define (make-equivalence-classes triples)
  (letrec ((loop (lambda (remaining-triples classes-so-far)
                   (match remaining-triples
                     ([] (rev classes-so-far))
                     ((list-of (as triple [pat guard res]) more)
                        (match (for-some' classes-so-far
                                 (lambda (triple-list)
                                    (unifiable pat (pat-of (first triple-list)))))
                          ([classes-1 triple-list classes-2]
                             (let ((classes' (join classes-1 [(join triple-list [triple])] classes-2)))
                               (loop more classes')))
                          (_ (loop more (add [triple] classes-so-far)))))))))
      (loop triples [])))

(define (simplify-guard triple previous-triples)
  (match triple
    ([_ () _] triple)
    ([pat guard res]
       (let ((guard-conjuncts (get-conjuncts-recursive guard))
             (structural-inequalities (conjuncts-of (diff* pat (map pat-of previous-triples))))
             (guard-conjuncts' (filter guard-conjuncts (lambda (c) (negate (member? c structural-inequalities)))))
             ([pat fgc res] (letrec ((loop (lambda (prev-triples final-triple)
                                                 (match [prev-triples final-triple]
                                                   ([[] _] final-triple)
                                                   ([(list-of [_ () _] more) _] (loop more final-triple))
                                                   ([(list-of [pat0 guard0 res0] more) [pat conjuncts res]]
                                                     (match (alpha-variants? pat0 pat)
                                                       ((some-sub sub)
                                                         (let ((guard0-conjuncts' (sub (conjuncts-of guard0)))
                                                               ([pat' conjuncts' res'] [(sub pat) (sub conjuncts) (sub
                                                               (conjuncts'' (filter-out conjuncts'
                                                                                        (lambda (c)
                                                                                          (member? (complement c) guard
                                                           (loop more [pat' conjuncts'' res'])))
                                                       (_ (loop more final-triple)))))))
                                     (loop previous-triples [pat guard-conjuncts' res]))))
         [pat (and* fgc) res]))))


# (define (glean-all-sure-negations previous-triples pat guard res)
#   (letrec ((loop (lambda (previous-triples results)
#                    (match previous-triples
#                       ([] results)
#                       ((list-of [pat0 guard0 res0] more)
#                          (match (non-linear-instance? pat0 pat)
#                            ((some-sub sub) (match (get-non-linear-constraints pat0)
#                                                   ([c] (sub c)

(define (linearize-pattern pat)
  (let ((T (table 10))
        (T' (table 10))
        (already-seen? (lambda (v) (try (seq (table-lookup T v) true) false))))
    (letrec ((loop (lambda (t)
                      (match t
                         ((some-var _) (let ((new-var (lhs (= (fresh-var) t)))
                                             (_ (table-add T' [t --> new-var]))
                                             (constraints (check ((already-seen? t)
                                                                     (let ((old-var (table-lookup T t))
                                                                           (_ (table-add T [t --> new-var])))
                                                                        [(= old-var new-var)]))
                                                                  (else (let ((_ (table-add T [t --> new-var])))
                                                                          []))))))
```

```
                                         [new-var constraints]))
                         (((some-symbol f) (some-list args)) (let (([args' constraints] (loop* args [[] []])))
                                                               [(make-term f args') constraints])))))
             (loop* (lambda (terms res)
                      (match [terms res]
                        ([[] [terms' constraints]] [(rev terms') (rev constraints)])
                        ([(list-of s more) [terms' constraints]]
                            (let (([s' constraints'] (loop s)))
                              (loop* more [(add s' terms') (join constraints' constraints)]))))))))
       (let ((res (loop pat))
             (sub (make-sub (table->list T))))
         [res sub]))))

(define (make-and props)
  (match props
    ([p] p)
    ([] true)
    (_ (and props))))

(define (augment-guards guard equalities)
  (match [equalities guard]
    ([[] _] guard)
    ([_ ()] (make-and equalities))
    (_ (and (join (get-conjuncts guard) equalities)))))

(define (apply-sub sub guard)
  (check ((equal? guard ()) guard)
         (else (sub guard))))

(define (linearize triple previous-triples)
  (match triple
    ([pat guard res] (let (([[pat' equalities] sub] (linearize-pattern pat))
#                           (_ (print "\npat: " pat "\nguard: " guard "\nres: " res "\npat': " pat' "\nequalities: " e
                           (_ ()))
                       [pat' (apply-sub sub (augment-guards guard equalities)) (sub res)]))))

#(define (linearize triple previous-triples)
#  triple)

(define (process-equivalence-class class)
  (letrec ((loop (lambda (remaining-triples processed-triples results)
                   (match remaining-triples
                     ([] (rev results))
                     ((list-of triple more-triples)
                        (let ((triple' (linearize triple processed-triples)))
                          (loop more-triples (add triple processed-triples)
                                             (add triple' results))))))))
    (loop class [] [])))


(define size' (lambda (x)
                 (match x
                   (() 0)
                   (_ (size x)))))


(define sort-class (lambda (class)
                       (merge-sort class
                           (lambda (triple1 triple2)
                             (less? (size' (guard-of triple1)) (size' (guard-of triple2)))))))

(define (analyze equations)
  (let ((all-triples (map decompose-equation' equations))
        (list-of-classes (make-equivalence-classes all-triples))
        (list-of-classes (map sort-class list-of-classes)))
    (map sort-class (map process-equivalence-class list-of-classes))))

(define triple->equation (lambda (triple)
                            (match triple
                              ([pat guard res]
```

```
                                    (let ((eqn (check ((member? guard [() true]) (= pat res))
                                                      (else (if guard (= pat res))))))
                                      (close eqn))))))

(define (all-vars? L) (for-each L var?))



(define (all-constructors? t)
  (for-each (syms t) (lambda (f) (|| (constructor? f) (numeral? f) (meta-id? f)))))

(define (all-constructors* L)
  (&&* (map all-constructors? L)))

(define (guards-should-be-matched? guard-res-list uvars)
 (try
  (let ((all-identical (lambda (L) (equal? (length (rd L)) 1)))
        (guards  (filter-out (map first guard-res-list) (lambda (x) (member? x [true ()]))))
        (guard-lhsides (map lhs guards))
        (guard-rhsides (map rhs guards))
        (results (map second guard-res-list))
        (cond1 (all-identical guard-lhsides))
        (cond2 (all-constructors* guard-rhsides))
        (cond3 (for-some guard-rhsides
                  (lambda (rhs)
                     (for-some (vars rhs) (lambda (v) (negate (member? v uvars))))))))
   (&& cond1 cond2 cond3))
   (seq # (print "\nFailed try...\n")
        false)))

(define (rhs-vars guard)
  (match guard
    ((= _ (some-term t)) (get-vars-manual t))
    (_ [])))

(define (get-extra-vars guard-res-list uvars)
  (let ((guards  (map first guard-res-list))
        (guard-rhside-vars (rd (flatten (map rhs-vars guards)))))
    (list-diff guard-rhside-vars uvars)))


(define (split-point L1 L2)
  (match [L1 L2]
    ([(list-of m rest1) (list-of m rest2)] (split-point rest1 rest2))
    ([(list-of m rest) _] L1)
    (_ [])))

(define (get-eval-proc-name-generic f mod-path)
  (let ((long-name (get-eval-proc-name-1 f))
        (toks  (tokenize long-name "."))
        ([mods epc] [(all-but-last toks) (last toks)])
        (mods' (split-point mods mod-path)))
     (match mods'
       ([] epc)
       (_ (join (separate mods' ".") "." epc)))))

(define (get-reduce-proc-name-generic f mod-path)
   (join (get-eval-proc-name-generic f mod-path) (standard-reduce-proc-name-suffix)))

(define [translate-symbol compile-term compile-guard compile-terms]
  (letrec
        ((translate-symbol (lambda (g)
                             (let ((long-name (get-eval-proc-name-1 g))
                                   #(_ (print "\nMOD PATH: " mod-path))
                                   (_ ()))
                               (check ((constructor? g)
                                         (symbol->string g))
                                      ((prefix?  (all-but-last (split-string long-name `.)) mod-path)
                                         (get-eval-proc-name g))
                                      (else (get-eval-proc-name-generic g mod-path))))))
```

```
        (compile-term (lambda (term mapping translate-symbol)
                       (let (#(_ (print "\nAbout to translate this term: " term))
                             (_ ()))
                        (letrec ((loop (lambda (t)
                                         (match t
                                           ((some-var x) (apply-map mapping x))
                                           (((some-symbol g) []) (val->string g))
                                           (((some-symbol g) (some-list args))
                                               (check ((equal? g =)
                                                         (join lp (translate-symbol g) blank (separate (map loop args) l
                                                       (else (join lp (translate-symbol g) blank (separate (map loop ar
                          (loop term)))))
        (compile-guard (lambda (guard mapping translate-symbol)
                         (letrec ((loop (lambda (g)
                                          (match g
                                            ((some-term _) (compile-term g mapping translate-symbol))
                                            ((not g') (join lp "negate " (loop g') rp))
                                            ((and g1 g2) (join lp "&& " (loop g1) blank (loop g2) rp))
                                            ((or g1 g2) (join lp "|| " (loop g1) blank (loop g2) rp))))))
                           (loop guard))))
        (compile-terms (lambda (terms mapping translate-symbol)
                         (map (lambda (t) (compile-term t mapping translate-symbol)) terms))))
 [translate-symbol compile-term compile-guard compile-terms]))

(define (constant? t)
  (&& (symbol? t) (equal? (arity-of t) 0)))

(define (poly-constant? t)
  (&& (constant? t) (poly? t)))

(define (mono? t) (negate (poly? t)))

(define (compile-entry equation mod-path)
  (let (
        #(_ (print "\nCalling compile-entry on equation: " equation))
        (translate-symbol (lambda (g)
                            (let ((long-name (get-eval-proc-name-1 g))
                                  #(_ (print "\nMOD PATH: " mod-path))
                                  (_ ()))
                              (check ((constructor? g)
                                        (symbol->string g))
                                     ((prefix?  (all-but-last (split-string long-name `.)) mod-path)
                                        (get-eval-proc-name g))
                                     (else (get-eval-proc-name-generic g mod-path))))))
        (compile-term (lambda (term mapping translate-symbol)
                       (let (#(_ (print "\nAbout to translate this term: " term))
                             (_ ()))
                        (letrec ((loop (lambda (t)
                                         (match t
                                           ((some-var x) (apply-map mapping x))
                                           (((some-symbol g) []) (val->string g))
                                           (((some-symbol g) (some-list args))
                                               (check ((equal? g =)
                                                         (join lp (translate-symbol g) blank (separate (map loop args) l
                                                       (else (join lp (translate-symbol g) blank (separate (map loop ar
                          (loop term)))))
        (compile-guard (lambda (guard mapping translate-symbol)
                         (letrec ((loop (lambda (g)
                                          (match g
                                            ((some-term _) (compile-term g mapping translate-symbol))
                                            ((not g') (join lp "negate " (loop g') rp))
                                            ((and g1 g2) (join lp "&& " (loop g1) blank (loop g2) rp))
                                            ((or g1 g2) (join lp "|| " (loop g1) blank (loop g2) rp))))))
                           (loop guard))))
        (compile-terms (lambda (terms mapping translate-symbol)
                         (map (lambda (t) (compile-term t mapping translate-symbol)) terms))))
    (match equation
      ([uvars (as pattern (f (some-list args))) guard-res-list]
       (let ((f-name (symbol->string f))
             (count (length guard-res-list))
```

```
            (new-vars (map (lambda (i)
                             (join "x" (val->string i)))
                           (from-to 1 (length uvars))))
            (input-vars (map (lambda (i)
                               (join "t" (val->string i)))
                             (from-to 1 (length uvars))))
            (mapping (extend empty-map (list-zip uvars new-vars)))
            (input-mapping (extend empty-map (list-zip uvars input-vars)))
            (are-all-vars (all-vars? args))
            (mapping (check (are-all-vars input-mapping)
                            (else mapping)))
            (pattern-translation (check (are-all-vars "_")
                                        (else (join lb (separate (compile-terms args mapping symbol->string) blank) rb
            (translate-clause (lambda (guard-res-pair i)
                                (match guard-res-pair
                                  ([guard res] (let ((cg (check ((&& (equal? guard true) (equal? i count)) "else")
                                                                (else (compile-guard guard mapping translate-symbol))
                                                     #(_ (print "\nCalling compile-term on this arg: " res " and this
                                                     (fourth-result (compile-term res mapping translate-symbol))
                                                     #(_ (print "\nFourth result: " fourth-result))
                                                      (_ ()))
                                                 (join lp cg blank (compile-term res mapping translate-symbol) rp))))
        (check
          ((negate (guards-should-be-matched? guard-res-list uvars))
               (let ((check-clauses (separate (map-with-index translate-clause guard-res-list) newline))
                     (check-translation (join lp "check " check-clauses rp)))
                 (join lp pattern-translation blank check-translation rp)))
           (else (let ((extra-vars (get-extra-vars guard-res-list uvars))
                       (compile-pattern (lambda (p mapping ts)
                                          (compile-term p mapping ts)))
                       (mapping' (extend mapping (list-zip extra-vars
                                                           (map (lambda (i)
                                                                  (join "p" (val->string i)))
                                                                (from-to 1 (length extra-vars))))))
                       (translate-match-clause
                           (lambda (guard-res-pair i)
                             (match guard-res-pair
                               ([guard res] (let ((cg (check ((&& (equal? guard true) (equal? i count)) "_")
                                                             (else (compile-pattern (rhs guard) mapping' translate-sy
                                              (join lp cg blank (compile-term res mapping' translate-symbol) rp)))))
                       (discriminant  (compile-term (lhs (first (first guard-res-list))) mapping' translate-symbol))
                       (match-clauses (separate (map-with-index translate-match-clause guard-res-list) newline))
                       (match-translation (join lp "match " discriminant "\n"
                                                match-clauses rp)))
                   (join lp pattern-translation blank match-translation rp))))))
      ((forall (some-list uvars)
               (= (f (some-list args)) rhs))
         (let ((f-name (symbol->string f))
               (new-vars (map (lambda (i)
                                (join "x" (val->string i)))
                              (from-to 1 (length uvars))))
               (mapping (extend empty-map (zip uvars new-vars)))
               (lhs' (join lb (separate (compile-terms args mapping symbol->string) blank) rb))
               (rhs' (compile-term rhs mapping translate-symbol)))
           (join lp lhs' blank rhs' rp)))
      ((forall (some-list uvars)
               (if antecedent (= (f (some-list args)) rhs)))
         (let ((f-name (symbol->string f))
               (new-vars (map (lambda (i)
                                (join "x" (val->string i)))
                              (from-to 1 (length uvars))))
               (mapping (extend empty-map (zip uvars new-vars)))
               (input-vars (map (lambda (i)
                                  (join "t" (val->string i)))
                                (from-to 1 (length uvars))))
               (input-mapping (extend empty-map (zip uvars input-vars)))
               (are-all-vars (all-vars? args))
               (mapping (check (are-all-vars input-mapping)
                               (else mapping)))
               (lhs' (check (are-all-vars "_")
```

```
                            (else (join lb (separate (compile-terms args mapping symbol->string) blank) rb))))
               (guard (compile-guard antecedent mapping translate-symbol))
               (lhs'' (join lp lhs' " where " guard rp))
               (rhs' (compile-term rhs mapping translate-symbol)))
          (join lp lhs'' blank rhs' rp))))))

(primitive-method (d+' s t)
  (let ((sym  (string->symbol "+"))
        (rator (evaluate (get-eval-proc-name sym)))
        (res (rator s t)))
   (= (sym s t) res)))

(define (d+' s t)
  (!force (let ((sym  (string->symbol "+"))
                (rator (evaluate (get-eval-proc-name sym)))
                (res (rator s t)))
      (= (sym s t) res))))

(primitive-method (d-' s t)
  (let ((sym  (string->symbol "-"))
        (rator (evaluate (get-eval-proc-name sym)))
        (res (rator s t)))
   (= (sym s t) res)))

(define (d-' s t)
  (!force
     (let ((sym  (string->symbol "-"))
           (rator (evaluate (get-eval-proc-name sym)))
           (res (rator s t)))
       (= (sym s t) res))))

(define (ded-unary-minus s)
  (!force
     (let ((sym  (string->symbol "-"))
           (rator (evaluate (get-eval-proc-name sym))))
       (= (sym s) (rator s)))))

(primitive-method (d*' s t)
  (let ((sym  (string->symbol "*"))
        (rator (evaluate (get-eval-proc-name sym)))
        (res (rator s t)))
   (= (sym s t) res)))

(define (d*' s t)
  (!force (let ((sym  (string->symbol "*"))
        (rator (evaluate (get-eval-proc-name sym)))
        (res (rator s t)))
   (= (sym s t) res))))

(primitive-method (d/' s t)
  (let ((sym  (string->symbol "/"))
        (rator (evaluate (get-eval-proc-name sym)))
        (res (rator s t)))
   (= (sym s t) res)))

(define (d/' s t)
  (!force (let ((sym  (string->symbol "/"))
        (rator (evaluate (get-eval-proc-name sym)))
        (res (rator s t)))
   (= (sym s t) res))))

(primitive-method (d=' s t)
  (match ((evaluate (get-eval-proc-name (string->symbol "="))) s t)
    (true (= (= s t) true))
    (_ (= (= s t) false))))

(define (d=' s t)
  (!force (match ((evaluate (get-eval-proc-name (string->symbol "="))) s t)
    (true (= (= s t) true))
    (_ (= (= s t) false)))))
```

```
(primitive-method (d<' s t)
  (let ((sym (string->symbol "<")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false)))))

(define (d<' s t)
  (!force (let ((sym (string->symbol "<")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false))))))

(primitive-method (d<=' s t)
  (let ((sym (string->symbol "<=")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false)))))

(define (d<=' s t)
  (!force (let ((sym (string->symbol "<=")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false))))))

(primitive-method (d>=' s t)
  (let ((sym (string->symbol ">=")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false)))))

(define (d>=' s t)
  (!force (let ((sym (string->symbol ">=")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false))))))

(primitive-method (d>' s t)
  (let ((sym (string->symbol ">")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false)))))

(define (d>' s t)
  (!force (let ((sym (string->symbol ">")))
    (match ((evaluate (get-eval-proc-name sym)) s t)
      (true (= (sym s t) true))
      (_ (= (sym s t) false))))))

## stran:

(define (stran eq1 eq2)
  (dtry (!tran eq1 eq2)
        (dmatch [eq1 eq2]
          ([(= l1 r1) (= l2 r2)]
            (dcheck ((sort-instance? r1 l2)
                         (dlet ((th (!sort-instance eq2 (= r1 r2))))
                           (!tran eq1 th))))))))


(define (deductive-version str)
  (match (rev str)
     ((split L1 (as L2 (list-of `. rest))) (join (rev L2) "d" (rev L1)))
     (_ (join "d" str))))

(define (unary-minus? f arity)
  (&& (equal? (symbol->string f) "-")
      (equal? arity 1)))

(define (deval0 t)
```

```
  (dmatch t
    ((some-var _) (!reflex t))
    (((some-symbol f) []) (!reflex t))
    ((bind term ((some-symbol f) (as args (list-of t more))))
        (dcheck ((&& (negate (free-constructor? f)) (unequal? (fsd f) ()))
                  (dtry
                    (!map-method deval0 args
                       (method (eqns)
                         (dlet ((rhs-list (map rhs eqns))
                                (rhs' (make-term f rhs-list))
                                (th1 (= term rhs'))
                                (_ (!fcong th1))
                                (f-method (check ((unary-minus? f (length args)) ded-unary-minus)
                                                 (else (try (evaluate (deductive-version (get-eval-proc-name-1 f)))
                                                            (evaluate (join "d" (get-eval-proc-name f)))))))
                                ((as th2 (= l r)) (!app-method f-method rhs-list))
                                (th (!stran th1 th2)))
                          (dcheck ((equal? rhs' r) (!claim th2))
                                  (else (dlet ((th3 (!deval0 r)))
                                          (!stran th th3)))))))
                    (!map-method deval0 args
                       (method (eqns)
                         (dlet ((rhs-list (map rhs eqns))
                                (term' (make-term f rhs-list)))
                           (!fcong (= term term')))))))
                (else (!map-method deval0 args
                         (method (eqns)
                           (dlet ((rhs-list (map rhs eqns))
                                  (rhs' (make-term f rhs-list))
                                  (th (= term rhs')))
                              (!fcong th)))))))))


(define (deval t)
  (dlet ((res (dtry (!deval0 t) (!true-intro))))
    (dmatch res
      (true (!proof-error (join "\nUnable to reduce the term:\n" (val->string t) "\nto a normal form.\n")))
      (_ (!claim res)))))

(set! deval-cell deval)

(define (evaluate-guard guard)
  (dmatch guard
    (((some-symbol pred) (some-list args))
        (dmatch (!deval guard)
           ((as res (= _ true)) (!identity->atom res))
           ((as res (= _ false)) (!identity->atom res))))
    ((not p) (dmatch (!evaluate-guard p)
               ((val-of p) (!by-contradiction (not (not p))
                                (assume (not p)
                                   (!absurd p (not p)))))
               ((as res (not (val-of p))) (!claim res))))
    ((and p1 p2) (dmatch (!evaluate-guard p1)
                   ((val-of p1) (dmatch (!evaluate-guard p2)
                                  ((val-of p2) (!both p1 p2))
                                  (not-p2 (!by-contradiction (not guard)
                                             (assume guard
                                               (!absurd (!right-and guard) not-p2))))))
                   (not-p1 (!by-contradiction (not guard)
                              (assume guard
                                (!absurd (!left-and guard) not-p1))))))
    ((or p1 p2) (dmatch (!evaluate-guard p1)
                  ((val-of p1) (!either p1 p2))
                  (not-p1 (dmatch (!evaluate-guard p2)
                            ((val-of p2) (!either p1 p2))
                            (not-p2 (!by-contradiction (not guard)
                                       (assume guard
                                          (!cases guard
                                                  (assume p1 (!absurd p1 not-p1))
                                                  (assume p2 (!absurd p2 not-p2)))))))))))
```

```
(define (do-clauses lhs guard-rhs-list justification)
  (dletrec ((loop (method (guard-res-list)
                    (dmatch guard-res-list
                      ((list-of [guard res] more)
                         (!evaluate-guard guard
                            (method ()
                              (!chain [lhs = res justification]))
                            (method ()
                              (!loop more))))))))
     (!loop guard-rhs-list)))


(define (prove-clause-step lhs rhs guard axioms)
  (dlet ((justification' (match axioms
                           ((some-list _) (add guard axioms))
                           ((some-sent _) [guard axioms])
                           (_ axioms))))
     (!chain [lhs = rhs justification'])))

(define (prove-clause-step lhs rhs guard axioms)
 (dlet ((original-axioms axioms)
        (given (= lhs rhs))
        (rename-first (lambda (L)
                        (match L
                          ((list-of (some-sent p) more) (add (rename p) more))
                          (_ L)))))
  (dletrec ((loop (method (axioms)
                    (dmatch (rename-first axioms)
                      ((list-of (|| (first-axiom as (forall (some-list uvars) (axiom-body as (if _ con))))
                                    (first-axiom as (forall (some-list uvars) (axiom-body as con))))
                                rest)
                        (dlet (#(_ (print "\nLooking at this axiom: " first-axiom "\n"))
                              )
                           (dmatch (match-sentences given con)
                              ((some-sub sub)
                                 (dtry (dmatch axiom-body
                                         ((if _ _)
                                            (dlet ((body' (!uspec* first-axiom (sub uvars)))
                                              #(_ (print "\nGot this body': " (val->string body') "\n"))
                                              ([ant con] (match body' ((if (some-sent A) (some-sent B)) [A B])))
                                              # (_ (print "\nWill now try to prove components of this guard: "
                                              #            (val->string ant)
                                              #                  "\ngiven this guard: " (val->string guard) "\n"))
                                              (_ (!prove-components-of ant)))
                                             (!mp body' ant)))
                                         ((= _ _) (!uspec* first-axiom (sub uvars))))
                                       (!loop rest)))
                              (_ (!loop rest)))))
                      (_ (dlet ( # (_ (print "\nFailed! prove-clause loop on this lhs:\n" (val->string lhs)
                                 #             "\nand this rhs:\n" (val->string rhs)
                                 #             "\nand this guard:\n" (val->string guard)
                                 #             "\nand these axioms:\n" original-axioms))
                                   (_ ())
                                )
                           (!fail)))
       ))))
     (!loop axioms))))

(define (prove-clause-step lhs rhs guard axioms)
  (!force (= lhs rhs)))

(define (do-clauses lhs guard-rhs-list justification)
  (dletrec ((loop (method (guard-res-list)
                    (dmatch guard-res-list
                      ((list-of [guard res] more)
                         (dlet (#(_ (print "\nAbout to evaluate this guard: " (val->string guard) "\n"))
                                (th (!evaluate-guard guard))
                                #(_ (print "\nFinished guard evaluation...\n"))
                                )
```

```
                              (dmatch th
                                ((val-of guard)
                                   (dlet (# (_ (print "\nThis guard was true: " (val->string guard)
                                          #            "\and we'll now chain from this lhs:\n" (val->string lhs)
                                          #                 " to this rhs side:\n" (val->string res) "\n"))
                                          (justification' (match justification
                                                              ((some-list _) (add guard justification))
                                                              ((some-sent _) [guard justification])
                                                              (_ justification)))
                                         )
                                      #(!chain [lhs = res justification'])
                                       (!prove-clause-step lhs res guard justification)

                                   ))
                                (_ (!loop more))))))))
     (!loop guard-rhs-list)))

(define [success failure] [(method () (!dmark 'S)) (method () (!dmark 'F))])

(define (dcompile-entry equation def-eqn-id mod-path)
  (let (#(_ (print "\nGive equation: " equation))
        (translate-symbol (lambda (g) (symbol->string g)))

                              # (let ((long-name (get-eval-proc-name-1 g)))
                              #   (check ((constructor? g)
                              #             (symbol->string g))
                              #          ((equal? mod-path (all-but-last (split-string long-name '.)))
                              #             (get-eval-proc-name g))
                              #          (else long-name)))))

#        (_ (print "\nAbout to dcompile-entry the following entry:\n" equation "\nwith the following def-eqn-id: " def
        (compile-term (lambda (term mapping)
                        (letrec ((loop (lambda (t)
                                         (match t
                                           ((some-var x) (apply-map mapping x))
                                           ((g []) (val->string g))
                                           ((g (some-list args))
                                                 (join lp (translate-symbol g)
                                                       blank (separate (map loop args) blank) rp))))))
                          (loop term))))
        (compile-term' (lambda (term mapping)
                         (letrec ((loop (lambda (t)
                                          (match t
                                            ((some-var x) (apply-map mapping x))
                                            ((g []) (val->string t))
                                            ((g (some-list args))
                                               (let ((_ ())
#                                                     (_ (print "\nTerm being translated: " t))
#                                                     (_ (print "\nRoot symbol is translated to this: " (translate-sym
                                                    )
                                                  (join lp
                                                        (translate-symbol g)
                                                        blank
                                                        (separate (map loop args) blank)
                                                        rp)))))))
                           (loop term))))
        (compile-guard (lambda (guard mapping)
                         (letrec ((loop (lambda (g)
                                          (match g
                                            ((some-term _) (compile-term' g mapping))
                                            ((not g') (join lp "not " (loop g') rp))
                                            ((and g1 g2) (join lp "and " (loop g1) blank (loop g2) rp))
                                            ((or g1 g2) (join lp "or " (loop g1) blank (loop g2) rp))))))
                           (loop guard))))
        (compile-terms (lambda (terms mapping)
                         (map (lambda (t) (compile-term t mapping)) terms)))
        (compile-terms' (lambda (terms mapping)
                          (map (lambda (t) (compile-term' t mapping)) terms))))
    (match equation
      ([uvars (as pattern (f (some-list args))) guard-res-list]
```

```
           (let ((f-name (symbol->string f))
                 (count (length guard-res-list))
                 (new-vars (map (lambda (i)
                                   (join "x" (val->string i)))
                               (from-to 1 (length uvars))))
                 (input-vars (map (lambda (i)
                                     (join "t" (val->string i)))
                                   (from-to 1 (length uvars))))
                 (mapping (extend empty-map (list-zip uvars new-vars)))
                 (input-mapping (extend empty-map (list-zip uvars input-vars)))
                 (are-all-vars (all-vars? args))
                 (mapping (check (are-all-vars input-mapping)
                                  (else mapping)))
#                (_ (print "\nmapping: " mapping))
                 (pattern-translation (check (are-all-vars "_")
                                              (else (join lb (separate (compile-terms' args mapping) blank) rb))))
                 (translate-clause (lambda (guard-res-pair i)
                                      (match guard-res-pair
                                        ([guard res] (let ((cg (compile-guard guard mapping))
                                                           (RHS (compile-term' res mapping)))
                                                       (join lb cg blank RHS rb)))))))
         (check
           ((negate (guards-should-be-matched? guard-res-list uvars))
                (let ((LHS (compile-term' pattern mapping))
                      (check-clauses (join lb (separate (map-with-index translate-clause guard-res-list) blank) rb))
                      (check-translation (join lp "!do-clauses " LHS blank check-clauses blank def-eqn-id rp)))
                  (join lp pattern-translation blank check-translation rp)))
            (else (let ((extra-vars (get-extra-vars guard-res-list uvars))
                         (compile-pattern (lambda (p mapping ts)
                                            (compile-term' p mapping)))
                         (mapping' (extend mapping (list-zip extra-vars
                                                         (map (lambda (i)
                                                                (join "p" (val->string i)))
                                                              (from-to 1 (length extra-vars))))))
                         (translate-match-clause
                             (lambda (guard-res-pair i)
                                (match guard-res-pair
                                  ([guard res] (let ((cg (check ((&& (equal? guard true) (equal? i count)) "_")
                                                                (else (join "(res as (= _ " (compile-pattern (rhs guard
                                                     (result-term (compile-term' res mapping'))
                                                     (RHS (join "(!chain [" (compile-term pattern mapping') " = " resul
                                                 (join lp cg blank RHS rp))))))
                         (discriminant  (join lp "!deval " (compile-term' (lhs (first (first guard-res-list))) mapping'
                         (match-clauses (separate (map-with-index translate-match-clause guard-res-list) newline))
                         (match-translation (join lp "dmatch " discriminant "\n"
                                                     match-clauses rp)))
                    (join lp pattern-translation blank match-translation rp))))))
      ((forall (some-list uvars)
                 (= (as pattern (f (some-list args))) rhs))
         (let ((f-name (symbol->string f))
               (new-vars (map (lambda (i)
                                 (join "x" (val->string i)))
                               (from-to 1 (length uvars))))
               (mapping (extend empty-map (zip uvars new-vars)))
               (LHS (compile-term' pattern mapping))
               (lhs' (join lb (separate (compile-terms' args mapping) blank) rb))
               (rhs' (compile-term rhs mapping))
               (RHS (compile-term' rhs mapping))
               (dbody (join "(!chain " lb LHS blank " = " RHS blank lb def-eqn-id rb rb rp)))
          (join lp lhs' blank dbody rp)))
      ((forall (some-list uvars)
                 (if antecedent (= (as pattern (f (some-list args))) rhs)))
         (let ((f-name (symbol->string f))
               (new-vars (map (lambda (i)
                                 (join "x" (val->string i)))
                               (from-to 1 (length uvars))))
               (mapping (extend empty-map (zip uvars new-vars)))
               (input-vars (map (lambda (i)
                                   (join "t" (val->string i)))
                                 (from-to 1 (length uvars))))
```

```
                (input-mapping (extend empty-map (zip uvars input-vars)))
                (are-all-vars (all-vars? args))
                (mapping (check (are-all-vars input-mapping)
                                (else mapping)))
                (LHS (compile-term' pattern mapping))
                (lhs' (check (are-all-vars "_")
                             (else (join lb (separate (compile-terms' args mapping) blank) rb))))
                (guard (compile-guard antecedent mapping))
                (lhs'' lhs')
                (rhs' (compile-term rhs mapping))
                (RHS (compile-term' rhs mapping))
                (dbody (join "(!chain " lb LHS blank " = " RHS blank lb def-eqn-id rb rb rp))
                (dbody (join "(!do-clauses " LHS blank lb lb guard blank RHS rb rb blank def-eqn-id rp)))
          (join lp lhs'' blank dbody rp))))))

(define (all-alpha-variants? ec)
 (let ((are-alpha-variants (lambda (s t)
                             (match (alpha-variants? s t)
                               ((some-sub _) true)
                               (_ false)))))
    (letrec ((loop (lambda (current-pat remaining-triples)
                     (match remaining-triples
                       ([] true)
                       ((list-of [pat guard res] more) (&& (are-alpha-variants pat current-pat)
                                                           (loop pat more)))
                       (_ false)))))
       (match ec
         ([] true)
         ((list-of [pat _ _] more) (loop pat more))))))

(define (get-guard g)
  (match g
    (() true)
    (_ g)))

(define (simple-simplify gr-list pat-vars)
  (match gr-list
    ([] gr-list)
    ((list-of [guard1 res1] more)
      (let ((conjuncts1 (get-conjuncts guard1)))
        (check ((negate (equal? (length conjuncts1) 1)) gr-list)
              (else (let ((sole-first-conjunct (first conjuncts1)))
                      (letrec ((loop (lambda (rem-pairs sure-so-far accum)
                                       (match rem-pairs
                                         ([] (add [guard1 res1] (rev accum)))
                                         ((list-of [guard res] more)
                                           (let ((conjuncts (get-conjuncts guard))
                                                 (conjuncts' (filter-out conjuncts (lambda (c) (member? (complement c
                                                 (guard' (make-and conjuncts'))
                                                 (sure-so-far' (match conjuncts'
                                                                 ([_] (add (first conjuncts') sure-so-far))
                                                                 (_ sure-so-far))))
                                             (loop more sure-so-far' (add [guard' res] accum))))))))
                        (loop more [sole-first-conjunct] []))))))))))

(define (make-single-entry ec)
  (match ec
    ([first-and-only-triple] (triple->equation first-and-only-triple))
    ((list-of (as first-triple [pat1 guard1 res1]) more-triples)
       (letrec ((get-pgr-list (lambda (remaining-triples results sub)
                                (match [remaining-triples results]
                                  ([[] _] [(rev results) sub])
                                  ([(list-of [pat guard res] more) []]
                                      (let ((guard' (get-guard guard)))
                                        (get-pgr-list more (add [pat guard' res] results) sub)))
                                  ([(list-of [pat guard res] more) (list-of [previous-pat previous-guard previous-res]
                                      (let ((guard' (get-guard guard))
                                            (sub' (unify previous-pat pat)))
                                        (get-pgr-list more (add [pat guard' res] results) (compose-subs sub' sub))))))
         (let ((get-vars (lambda (x)
```

```
                           (match x
                             (() [])
                             (_ (vars x)))))
#                (uvars (rev (rd (join (get-vars-manual pat1) (get-vars guard1)))))
               (uvars (rd (get-vars-manual pat1)))
               (pat pat1)
               ([pat-guard-res-list sub] (get-pgr-list ec [] empty-sub))
               ((answer as [uvars' pat' gr-list'])
                      [(sub uvars) (sub pat) (map (lambda (triple)
                                                     (match triple
                                                       ([p g r] [(sub g) (sub r)])))
                                                   pat-guard-res-list)])
               (pvars (sub (vars pat1)))
##               (_ (print "\nanswer: " answer "\nsub: " sub "\npvars: " pvars "\nand gr-list': " gr-list'))
               (gr-list'' (simple-simplify gr-list' pvars)))
           [uvars' pat' gr-list''])))))

(define reprocess-ec (lambda (ec)
                       (check ((all-alpha-variants? ec) [(make-single-entry ec)])
                              (else (map triple->equation ec)))))

(define (compile-symbol f mod-path)
 (let ((arity (arity-of f))
        (f-name (get-eval-proc-name f))
        (params (map (lambda (i) (join "t" (val->string i)))
                     (from-to 1 arity)))
        (params'  (separate params blank))
        (all-equations (defining-axioms f))
        (processsed-equivalence-classes (analyze all-equations))
        (reprocessed-equivalence-classes (map reprocess-ec processsed-equivalence-classes))
        (all-entries (flatten reprocessed-equivalence-classes))
        (clauses (map (lambda (e) (compile-entry e mod-path)) all-entries))
        (body (match clauses
                ([(split "(_" rest)] (all-but-last rest))
                (_ (join lp "match " lb params' rb newline tab tab (separate clauses (join newline tab tab)) rp))))
        (res (join lp "letrec " lp lp f-name blank lp "lambda " lp params' rp newline tab body rp rp rp newline tab tal
    res))


(define (escape str)
  (escape-string str))

(define (dcompile-symbol f mod-path)
 (let ((arity (arity-of f))
        (f-name (get-eval-proc-name f))
        (symbol-name (symbol->string f))
        (safe-symbol-name (join lp "string->symbol " quote (escape symbol-name) quote rp))
        (params (map (lambda (i) (join "t" (val->string i)))
                     (from-to 1 arity)))
        (params'  (separate params blank))
        (all-equations (defining-axioms f))
        (defining-axioms-id "defining-axioms")
        (processsed-equivalence-classes (analyze all-equations))
        (reprocessed-equivalence-classes (map reprocess-ec processsed-equivalence-classes))
        (all-entries (flatten reprocessed-equivalence-classes))
        (clauses (map (lambda (e) (dcompile-entry e defining-axioms-id mod-path)) all-entries))
        (body (match clauses
                ([(split "(_" rest)] (all-but-last rest))
                (_ (join lp "dmatch " lb params' rb newline tab tab (separate clauses (join newline tab tab)) rp))))
        (res (join lp "let " lp lp defining-axioms-id blank lp "defining-axioms" blank safe-symbol-name rp rp rp newli
                   lp "method " lp params' rp newline tab body rp rp)))
    res))

(define (compile-symbols fsyms mod-path)
 (let ((arities (map arity-of fsyms))
       (names (map get-eval-proc-name fsyms))
       (param-lists (map (lambda (fsym)
                           (map (lambda (i) (join "t" (val->string i)))
                                (from-to 1 (arity-of fsym))))
                         fsyms))
```

```
       (param-lists' (map (lambda (param-list) (separate param-list blank)) param-lists))
       (equation-lists (map defining-axioms fsyms))
       (clause-lists (map (lambda (equation-list)
                            (let ((processsed-equivalence-classes (analyze equation-list))
                                  (reprocessed-equivalence-classes (map reprocess-ec processsed-equivalence-classes))
                                  (all-entries (flatten reprocessed-equivalence-classes)))
                              (map (lambda (e) (compile-entry e mod-path)) all-entries)))
                          equation-lists))
       (bodies (map (lambda (pcl)
                      (match pcl
                        ([param-list clause-list]
                           (match clause-list
                             ([(split "(_" rest)] (all-but-last rest))
                             (_ (join lp "match " lb param-list rb newline tab tab (separate clause-list (join newline
                    (zip param-lists' clause-lists)))
       (lams (map (lambda (pbl)
                     (match pbl
                       ([param-list body] (join lp "lambda " lp param-list rp newline tab body rp))))
                  (zip param-lists' bodies)))
       (bindings (map (lambda (name-and-lam)
                        (match name-and-lam
                           ([name lam] (join lp name blank lam rp))))
                      (zip names lams)))
       (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb r
    res))

(define (compile-symbols-with-default fsyms mod-path)
 (let ((arities (map arity-of fsyms))
       (names (map get-eval-proc-name fsyms))
       (param-lists (map (lambda (fsym)
                            (map (lambda (i) (join "t" (val->string i)))
                                 (from-to 1 (arity-of fsym))))
                         fsyms))
       (param-lists' (map (lambda (param-list) (separate param-list blank)) param-lists))
       (equation-lists (map defining-axioms fsyms))
       (clause-lists (map (lambda (equation-list-fsym-params)
                            (let (([equation-list [fsym params]] equation-list-fsym-params)
                                  (processsed-equivalence-classes (analyze equation-list))
                                  (reprocessed-equivalence-classes (map reprocess-ec processsed-equivalence-classes))
                                  (all-entries (flatten reprocessed-equivalence-classes))
                                  (last-entry (join "(_ (" (val->string fsym) " " params "))")))
                              (join (map (lambda (e) (compile-entry e mod-path)) all-entries)
                                    [last-entry])))
                          (list-zip equation-lists (list-zip fsyms param-lists'))))
       (bodies (map (lambda (pcl)
                      (match pcl
                        ([param-list clause-list]
                           (match clause-list
                             ([(split "(_" rest)] (all-but-last rest))
                             (_ (join lp "match " lb param-list rb newline tab tab (separate clause-list (join newline
                    (zip param-lists' clause-lists)))
       (lams (map (lambda (pbl)
                     (match pbl
                       ([param-list body] (join lp "lambda " lp param-list rp newline tab body rp))))
                  (zip param-lists' bodies)))
       (bindings (map (lambda (name-and-lam)
                        (match name-and-lam
                           ([name lam] (join lp name blank lam rp))))
                      (zip names lams)))
       (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb r
    res))

#(assert* ite-axioms :=
#   (fun-def [(ite true ?x _) = ?x
#           | (ite false _ ?y) = ?y]))

#(define (ite' x y z)
#  (match x
#     (true y)
#     (_ z)))
```

```
(define [+' -' *' /' %'] [plus minus times div mod])
(define [+` -` *` /` %`]   [+' -' *' /' %'])

(define [=' <' >' <=' >='] [struc-equal? less? greater? less-or-equal? greater-or-equal?])

(define [=` <` >` <=` >=`]  [=' <' >' <=' >='])

(define [+'R -'R *'R /'R %'R] [+' -' *' /' %'])

(define (+'R x y)
  (match [x y]
    ([0 _] y)
    ([_ 0] x)
    (_ (plus x y))))

(define (-'R x y)
  (match [x y]
    ([_ 0] x)
    ([(some-term z) z] 0)
    (_ (minus x y))))

(define (*'R x y)
  (match [x y]
    ((|| [0 _] [_ 0]) 0)
    ([1 _] y)
    ([_ 1] x)
    (_ (times x y))))

(define [='R <'R >'R <='R >='R] [=' <' >' <=' >='])

(define (='-basic s t)
  (let ((res (struc-equal? s t)))
    (check ((equal? res true) res)
           ((&& (super-canonical? s) (super-canonical? t)) res)
           (else (let ((f (root s))
                       (g (root t)))
                   (check ((&& (free-constructor? f) (free-constructor? g) (unequal? f g)) false)
                          (else (let ((eq (= s t)))
                                  (check ((holds? eq) true)
                                         ((holds? (not eq)) false)
                                         (else (equal? s t)))))))))))

(define (=' s t)
  (check ((|| (poly? s) (poly? t))
            (match (= s t)
              ((= (some-term l) (some-term r)) (='-basic l r))))
         (else (='-basic s t))))

(define =` =')

(define (='R s t)
  (let ((res (struc-equal? s t)))
    (check ((equal? res true) res)
           ((&& (super-canonical? s) (super-canonical? t)) res)
           (else (let ((f (root s))
                       (g (root t)))
                   (check ((&& (free-constructor? f) (free-constructor? g) (unequal? f g)) false)
                          (else (let ((eq (= s t)))
                                  (check ((holds? eq) true)
                                         ((holds? (= t s)) true)
                                         ((holds? (not eq)) false)
                                         ((equal? s t) true)
                                         (else (= s t)))))))))))

(define silent-eval-mode (cell false))

(define
 (eval1 t)
```

```
    (match t
      (((some-symbol f) (some-list args))
         (check ((null? args) t)
                ((constructor? f)
                   (match (defining-axioms f)
                     ([]  (make-term f (map eval args)))
                     (_ (let ((f' (try (evaluate (get-eval-proc-name-generic f (mod-path)))
                                       (evaluate (get-eval-proc-name f))))
                              (args' (map eval args)))
                          (try (app-proc f' args') (make-term f args'))))))
               (else (let ((f' (try (evaluate (get-eval-proc-name-generic f (mod-path)))
                                     (evaluate (get-eval-proc-name f)))))
                       (app-proc f' (map eval args))))))
    ((not p) (negate (eval p)))
    ((and (some-list args)) (&&* (map eval args)))
    ((or (some-list args)) (||* (map eval args)))
    ((if p1 p2) (eval (or (not p1) p2)))
    ((iff p1 p2) (eval (and (if p1 p2) (if p2 p1))))
    (_ t))
  (eval t)
    (try (let ((res (eval1 t)))
           (try (rhs (= t res))
                res))
         (check ((ref silent-eval-mode) ())
                (else (print "\nUnable to reduce the term:\n" t "\nto a normal form.\n")))))


(define
 (eval1 t)
    (match t
      (((some-symbol f) (some-list args))
        (check ((|| (canon? t) (meta-id? t) (&& (null? args) (null? (defining-axioms f)))) t)
               ((constructor? f)
                   (match (defining-axioms f)
                     ([]  (make-term f (map eval1 args)))
                     (_ (let ((f' (try (evaluate (get-eval-proc-name-generic f (mod-path)))
                                       (evaluate (get-eval-proc-name f))))
                              (args' (map eval1 args))
                              (res (try (app-proc f' args') (make-term f args'))))
                          (check ((poly? res) (rhs (= t res))) (else res))))))
               (else (let ((f' (try (let ((name (get-eval-proc-name-generic f (mod-path)))
                                          )
                                      (evaluate name))
                                    (evaluate (get-eval-proc-name f))))
                           (res (app-proc f' (map eval1 args))))
                       (check ((poly? res) (rhs (= t res))) (else res))))))
    ((not p) (negate (eval1 p)))
    ((and (some-list args)) (&&* (map eval1 args)))
    ((or (some-list args)) (||* (map eval1 args)))
    ((if p1 p2) (eval1 (or (not p1) p2)))
    ((iff p1 p2) (eval1 (and (if p1 p2) (if p2 p1))))
    (_ t))
  (eval t)
   (check ((has-vars? t) (print "\nNon-ground term given as input, unable to reduce the term:\n" t "\nto a normal form
    (else (try (eval1 t)
               (check ((ref silent-eval-mode) ())
                      (else (print "\nUnable to reduce the term:\n" t "\nto a normal form.\n")))))))


(make-private "eval1")

(define (eval-silent t)
  (let ((x (ref silent-eval-mode))
        (_ (set! silent-eval-mode true))
        (res (eval t))
        (_ (set! silent-eval-mode x)))
     res))


(define
```

```
 (reduce1 t)
    (match t
      (((some-symbol f) (some-list args))
        (check ((|| (canon? t) (meta-id? t) (&& (null? args) (null? (defining-axioms f)))) t)
               ((&& (constructor? f) (null? (defining-axioms f))) (make-term f (map reduce1 args)))
               (else (let ((f' (try (evaluate (get-reduce-proc-name-generic f (mod-path)))
                                    (evaluate (get-reduce-proc-name f))
                                    ())))
                       (match f'
                         ((|| () (some-symbol _)) (make-term f (map reduce1 args)))
                         (_ (app-proc f' (map reduce1 args))))))))
    ((not p) (negateR (reduce1 p)))
    ((and (some-list args)) (&&R (map reduce1 args)))
    ((or (some-list args)) (||R (map reduce1 args)))
    ((if p1 p2) (reduce1 (or (not p1) p2)))
    ((iff p1 p2) (reduce1 (and (if p1 p2) (if p2 p1))))
    ((forall (some-list vars) (some-sentence body)) (forall* vars (reduce1 body)))
    ((exists (some-list vars) (some-sentence body)) (exists* vars (reduce1 body)))
    (_ t))
  (reduce t)
    (match t
      ((some-sentence p) (check ((holds? p) true)
                                ((|| (holds? (complement p)) (holds? (not p))) false)
                                (else (let ((res (reduce1 t)))
                                        (try (rhs (= t res))
                                             res)))))
       (_ (let ((res (reduce1 t)))
                     (try (rhs (= t res))
                       res)))))


(define (eval2 t)
  (match t
    (((some-symbol f) (some-list args))
      (check ((|| (canon? t) (meta-id? t) (&& (null? args) (null? (defining-axioms f)))) t)
             ((constructor? f) (make-term f (map eval2 args)))
             (else (let ((error? (cell false))
                         (f' (try (evaluate (get-eval-proc-name-1 f))
                                  (evaluate (get-eval-proc-name f))
                                  (set! error? true))))
                     (check ((ref error?) t)
                            (else (let ((res (try (app-proc f' args)
                                                  (set! error? true))))
                                     (check ((ref error?) t)
                                            (else (eval2 res))))))))))
    ((not p) (negate (eval2 p)))
    ((and (some-list args)) (&&* (map eval2 args)))
    ((or (some-list args)) (||* (map eval2 args)))
    ((if p1 p2) (eval2 (or (not p1) p2)))
    ((iff p1 p2) (eval2 (and (if p1 p2) (if p2 p1))))
    (_ t)))

(define (eval' t)
  (try (eval2 t)
        t))

(set-precedence eval 5)
(set-precedence reduce 5)

(define (matches-with l r theta-cell)
 (let (#(_ (print "\nInside matches-with, with left: " l " and right: " r))
       (_ ())
      )
  (match [l r]
     ([(some-term left) (some-term right)] (match (match-terms left ((ref theta-cell) right))
                                              ((some-sub sub) (seq (set! theta-cell (compose-subs sub (ref theta-cell)
                                              (_ false)))
     ([(some-sentence left) (some-sentence right)]
          (match (match-sentences left ((ref theta-cell) right))
            ((some-sub sub) (seq (set! theta-cell (compose-subs sub (ref theta-cell))) true))
```

```
            (_ false)))
     (_ false))))

(define (matches-with left right theta-cell)
  (let ((theta (ref theta-cell)))
    (match (match-terms (theta left) (theta right))
       ((some-sub sub) (seq (set! theta-cell (compose-subs sub theta)) true))
       (_ false))))

(define (translate-symbol-new g mod-path evaluator-name)
  (let ((reducing? (check ((equal? evaluator-name "eval1") false) (else true)))
        (long-name (check (reducing? (get-reduce-proc-name-1 g))
                          (else (get-eval-proc-name-1 g))))
        (_ ()))
        (check ((constructor? g)
                  (symbol->string g))
               ((prefix?  (all-but-last (split-string long-name `.)) mod-path)
                  (check (reducing? (get-reduce-proc-name g))
                         (else (get-eval-proc-name g)))
                )
               (else (check (reducing?  (get-reduce-proc-name-generic g mod-path))
                            (else (get-eval-proc-name-generic g mod-path)))))))


(define (translate-arg-symbol-new g mod-path evaluator-name)
  (symbol->string g))

(define (compile-constant-term t mod-path evaluator-name)
   (check ((null? (defining-axioms t)) (val->string t))
          (else (join lp (translate-symbol-new (root t) mod-path evaluator-name) rp))))

(define (compile-constant-term-arg t mod-path evaluator-name)
  (check ((poly? t) (let ((v (var->string (fresh-var))))
                      (join "(" v " as " (val->string t) ")")))
         (else (val->string t))))

 (define (compile-term-new term mapping mod-path evaluator-name)
    (letrec ((loop (lambda (t)
                (match t
                   ((some-var x) (try (match (apply-map mapping x)
                                        (()  (val->string t))
                                        (res res))
                                      (val->string t)))
                   ((g []) (compile-constant-term t mod-path evaluator-name))
                   ((g (some-list args))
                     (check ((&& false (equal? evaluator-name "reduce")
                                 (equal? (fsd0 g) ()))
                               (join lp (symbol->string g) " "  (separate (map loop args) blank) rp))
                            (else (join lp (translate-symbol-new g mod-path evaluator-name) blank (separate (map loop a
       (loop term)))


 (define (compile-arg-new term mapping mod-path evaluator-name)
    (letrec ((loop (lambda (t)
                (match t
                   ((some-var x) (try (match (apply-map mapping x)
                                        (()  (val->string t))
                                        (res res))
                                      (val->string t)))
                   ((g []) (compile-constant-term-arg t mod-path evaluator-name))
                   ((g (some-list args))
                     (check ((&& (equal? evaluator-name "reduce")
                                 (equal? (fsd0 g) ()))
                               (join lp (symbol->string g) " " (separate (map loop args) blank) rp))
                            (else (join lp (translate-arg-symbol-new g mod-path evaluator-name) blank (separate (map lo
       (loop term)))

 (define (compile-term-and-boolean-combination term mapping mod-path evaluator-name)
    (letrec ((loop (lambda (t)
                (match t
```

```
                   ((some-var x) (try (match (apply-map mapping x)
                                        (()  (val->string t))
                                        (res res))
                                      (val->string t)))
                   ((not (some-sentence arg)) (join "(negate " (loop arg) ")"))
                   ((and (some-list args)) (let ((results (map loop args)))
                                             (join "(&& " (separate results "  ") ")")))
                   ((or (some-list args)) (let ((results (map loop args)))
                                             (join "(|| " (separate results "  ") ")")))
                   ((g []) (compile-constant-term t mod-path evaluator-name))
                   ((g (some-list args))
                     (join lp (translate-symbol-new g mod-path evaluator-name) blank (separate (map loop args) blank)
       (loop term)))

 (define (compile-term-with-no-eval term mapping mod-path evaluator-name)
    (letrec ((loop (lambda (t)
                (match t
                   ((some-var x) (try (match (apply-map mapping x)
                                        (()  (val->string t))
                                        (res res))
                                      (val->string t)))
                   ((g []) (compile-constant-term t mod-path evaluator-name))
                   ((g (some-list args))
                     (join lp (symbol->string g) blank (separate (map loop args) blank) rp))))))
       (loop term)))

(define (guard->where-condition guard mapping left-uvars fsym-being-defined needs-sub? mod-path evaluator-name)
  (let ((orient (lambda (s t t-vars)
                   (check ((negate (subset? t-vars left-uvars)) (= s t))
                          ((negate (subset? (vars s) left-uvars)) (= t s))
                          ((constructor? (root s)) (= t s))
                          ((constructor? (root t)) (= s t))
                          ((member? fsym-being-defined (syms s)) (= s t))
                          ((member? fsym-being-defined (syms t)) (= t s))
                          (else (= s t)))))
        (_ (set! needs-sub? false))
        (_ ()))
    (letrec ((loop (lambda (guard)
                      (match guard
                         ((= s t) (let ((t-vars (vars t))
                                        #(_ (print "\nDoing this guard: "  guard ", with left-uvars: " left-uvars))
                                        (oriented-guard (orient s t t-vars))
                                        #(_ (print "\nOriented guard: " oriented-guard))
                                        (_ ()))
                                   (match oriented-guard
                                      ((= s t) (check ((subset? (vars t) left-uvars)
                                                          (compile-term-new (= s t) mapping mod-path evaluator-name))
                                                      (else (seq (set! needs-sub? true)
                                                                 (join "(matches-with " (compile-term-new s mapping mo
                                                                                        (compile-term-new t mapping mo
                         ((some-atom A)
                            (let (#(_ (print "\nDoing this guard: "  guard ", with left-uvars: " left-uvars))
                                   (_ ()))
                               (check ((subset? (vars A) left-uvars) (compile-term-new A mapping mod-path evaluator-nam
                                      (else (seq (set! needs-sub? true)
                                                 (join "(eval1 ((ref theta-cell) "
                                                        (val->string A) "))"))))))
                         ((and (some-list guards)) (let ((results (map loop guards)))
                                                     (check ((equal? evaluator-name "eval1")
                                                               (join "(&& " (separate results "  ") ")"))
                                                            (else (join "(&&R [" (separate results "
") "])")))))
                         ((or (some-list guards)) (let ((results (map loop guards)))
                                                    (check ((equal? evaluator-name "eval1")
                                                              (join "(|| " (separate results "  ") ")"))
                                                           (else (join "(||R [" (separate results "
") "])")))))
                         ((not guard) (check ((equal? evaluator-name "eval1")
                                                 (join "(negate " (loop guard) ")"))
                                             (else (join "(negateR " (loop guard) ")"))))))))
```

```
      (match guard
        (true "_")
        (_ (loop guard))))))


(define (compile-equation-into-single-clause eqn mod-path evaluator-name)
 (let ((needs-sub? (cell false))
  (res (match eqn
    ((forall (some-list uvars) (=  (as left ((some-symbol f) (some-list args))) (some-term right)))
       (let ((proper-uvars (vars left))
             #(nums-and-uvars (list-zip (from-to 1 (length proper-uvars)) proper-uvars))
             (params (map (lambda (i)
                                (join "x" (val->string i)))
                          (from-to 1 (length proper-uvars))))
             (mapping (extend empty-map (list-zip proper-uvars params)))
             (arg-translations (separate (map (lambda (a) (compile-arg-new a mapping mod-path evaluator-name)) args) "
             (pattern (join "[" arg-translations "]"))
             (result (compile-term-new right mapping mod-path evaluator-name)))
         (join "(" pattern "  " result ")\n")))
    ((forall (some-list uvars) (if (some-sentence guard)
                                   (= (as left ((some-symbol f) (some-list args)))
                                      (some-term right))))
       (let ((proper-uvars (vars left))
             (params (map (lambda (i)
                                (join "x" (val->string i)))
                          (from-to 1 (length proper-uvars))))
             (mapping (extend empty-map (list-zip proper-uvars params)))
             (where-condition (guard->where-condition guard mapping proper-uvars f needs-sub? mod-path evaluator-name)
             (arg-translations (separate (map (lambda (a) (compile-term-new a mapping mod-path evaluator-name)) args)
             (pattern (check ((equal? where-condition "_")
                                (join "[" arg-translations "]"))
                             (else  (join "([" arg-translations "] where " where-condition ")"))))
             (result (check ((ref needs-sub?) (join "(" evaluator-name " ((ref theta-cell) " (compile-term-with-no-eva
                            (else (compile-term-new right mapping mod-path evaluator-name)))))
         (join "(" pattern "  " result ")\n")))
    ((forall (some-list uvars) (iff (as left ((some-symbol f) (some-list args))) (some-sentence right)))
       (let ((proper-uvars (vars left))
             (params (map (lambda (i)
                                (join "x" (val->string i)))
                          (from-to 1 (length proper-uvars))))
             (mapping (extend empty-map (list-zip proper-uvars params)))
             (arg-translations (separate (map (lambda (a) (compile-term-new a mapping mod-path evaluator-name)) args)
             (pattern (join "[" arg-translations "]"))
             (result (compile-term-and-boolean-combination right mapping mod-path evaluator-name)))
         (join "(" pattern "  " result ")\n"))))))
   [res (ref needs-sub?)]))

(define (compile-equation-into-single-clause' eqn mod-path evaluator-name)
  (match eqn
    ((forall (some-list uvars) (and (some-list args)))
        (map (lambda (arg) (compile-equation-into-single-clause (forall* uvars arg) mod-path evaluator-name)) args))
    (_ (let ((res (compile-equation-into-single-clause eqn mod-path evaluator-name))
#             (_ (print "\nResult: " res))
             )
          [res]))))

(define (compilable-bicond-axiom p)
  (match p
    ((forall (some-list uvars) (iff (some-atom A) (some-sentence RHS)))
       (match A
         (((some-symbol f) (some-list _)) (unequal? f =))
         (_ false)))
    (_ false)))

(define (eqn-guard e)
  (match e
    ((forall (some-list _) (if g _)) g)
    ((forall (some-list _) _) ())))

(define (follows-from-0 c lhs eqns)
```

```
  (let ((entails? (lambda (p1 p2)
                    (equal? (complement p1) (complement p2))))
        (apply-dm (lambda (p)
                    (match p
                      ((not (or (some-list props))) (and* (map (lambda (q) (complement q)) props)))
                      (_ p)))))
    (for-some eqns
              (lambda (e)
                (match (rename e)
                  ((forall (some-list _) (if g (= lhs' _)))
                     (match (unify lhs lhs')
                        (false false)
                        ((some-sub sub)
                         (let ((c' (sub c))
                               (g' (sub g)))
                          (|| (equal? c' (complement g'))
                              (let ((conjuncts (get-conjuncts (app-dm (not g')))))
                                 (for-some conjuncts
                                           (lambda (c)
                                              (entails? c c')))))))))
                  (_ false))))))

(define (follows-from c lhs eqns)
  (let ((left-sides (map left-rule-side eqns))
        (diff-props (try (get-conjuncts (diff* lhs left-sides)) [])))
    (|| (follows-from-0 c lhs eqns)
        (member? c diff-props))))


# rearrange-and-simplify takes a list of eqns E1 ... En and reorders and simplifies them
# so that redundant guards in the newly produced permutation are eliminated. A guard in
# an equation Ei' in the new list E1' ... En' is considered redundant if it follows from
# the negation of the guard of some preceding equation E1' .. E{i-1}'. What constitutes
# "follows from" is relative, and handled by the internal procedure entails? inside
# follows-from above. Right now it's just syntactic identity, but entails? could be
# redefined to, e.g., take into account datatype axioms.

(define (rearrange-and-simplify eqns)
   (let (#(_ (print "\nAbout to rearrange and simplify the following eqns:\n" eqns))
         (guard-size (lambda (g) (match g (() 0) (_ (size g)))))
         (eqns (merge-sort eqns
                  (lambda (e1 e2)
                    (less? (guard-size (eqn-guard e1))
                           (guard-size (eqn-guard e2)))))))
     (letrec ((loop (lambda (remaining-eqns previous-eqns)
                      (match remaining-eqns
                        ([] (rev previous-eqns))
                        ((list-of e rest)
                          (match e
                            ((forall (some-list uvars) (if g (as body (= (some-term l) (some-term r)))))
                               (let ((conjuncts (get-conjuncts g))
                                     (conjuncts' (filter-out conjuncts (lambda (c) (follows-from c l previous-eqns))))
                                     (new-guard (match conjuncts'
                                                  ([] true)
                                                  (_ (and* conjuncts')))))
                                 (loop rest (add (forall* uvars (if new-guard body)) previous-eqns))))
                            (_ (loop rest (add e previous-eqns)))))))))
        (let ((res (loop eqns []))
              #(_ (print "\nRESULT:\n"  res))
             )
          res))))

# To debug the simplifier rearrange-and-simplify while making sure that compilation works,
# uncomment the following two lines to redefine rearrange-and-simplify as the identity function:

#(define (rearrange-and-simplify eqns)
# eqns)

(define (compile-symbols-simple fsyms mod-path)
 (let ((arities (map arity-of fsyms))
```

```
       (names (map get-eval-proc-name fsyms))
       (param-lists (map (lambda (fsym)
                            (map (lambda (i) (join "t" (val->string i)))
                                 (from-to 1 (arity-of fsym))))
                         fsyms))
       (param-lists' (map (lambda (param-list) (separate param-list blank)) param-lists))
       (get-defining-axioms (lambda (f)
                              (let ((axioms (rev (defining-axioms f)))
                                    (bicond-sources (get-bicond-sources f))
                                    (compilable-elements
                                        (filter
                                          bicond-sources
                                          (lambda (pair)
                                            (match pair
                                             ([eqn bc] (compilable-bicond-axiom bc))))))
                                    (bicond-axioms (rd (map second compilable-elements)))
                                    (equational-axioms (list-diff axioms (rd (map first compilable-elements)))))
                                (join equational-axioms bicond-axioms))))
       (equation-lists (map (lambda (f) (rearrange-and-simplify (get-defining-axioms f))) fsyms))
#       (_ (map-proc (lambda (el) (print "\nFinal axiom list to be compiled: " el)) equation-lists))
       (clause-lists (map (lambda (equation-list)
                            (map (lambda (e) (flatten (compile-equation-into-single-clause' e mod-path "eval1"))) equat
                          equation-lists))
#       (_ (map-proc (lambda (cl) (seq (print "\nClause list: ") (print cl) clause-lists))))

       (bodies (map (lambda (pcl)
                      (match pcl
                        ([param-list clause-list]
                           (let ((clause-list' (map first clause-list))
                                 (needs-sub (for-some (map second clause-list) (lambda (x) x)))
                                 (body1 (join lp "match " lb param-list rb newline tab tab (separate clause-list' (joi
                             (check (needs-sub (join "(let ((theta-cell (cell empty-sub))) " body1 ")"))
                                    (else body1))))))
                    (list-zip param-lists' clause-lists)))
       (lams (map (lambda (pbl)
                    (match pbl
                      ([param-list body] (join lp "lambda " lp param-list rp newline tab body rp))))
                  (list-zip param-lists' bodies)))
       (bindings (map (lambda (name-and-lam)
                        (match name-and-lam
                           ([name lam] (join lp name blank lam rp))))
                      (list-zip names lams)))
       (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb rp
   res))


(define (compile-symbols-simple-with-default fsyms mod-path)
 (let ((arities (map arity-of fsyms))
       (eval-names (map get-eval-proc-name fsyms))
#       (_ (print "\nEval names: " (separate eval-names ", ")))
       (names (map get-reduce-proc-name fsyms))
#       (_ (print "\nAnd reduce names: " (separate names ", ")))
       (param-lists (map (lambda (fsym)
                            [(map (lambda (i) (join "t" (val->string i)))
                                  (from-to 1 (arity-of fsym)))
                             fsym])
                         fsyms))
#       (_( print "\nParam lists: " param-lists))
       (param-lists' (map (lambda (param-list)
                            (match param-list
                              ([plist fsym] [(separate plist blank) fsym])))
                          param-lists))
#       (_( print "\nParam lists': " param-lists'))
       (get-defining-axioms (lambda (f)
                              (let ((axioms (rev (defining-axioms f)))
                                    (bicond-sources (get-bicond-sources f))
                                    (compilable-elements
                                        (filter
                                          bicond-sources
                                          (lambda (pair)
```

```
                                            (match pair
                                             ([eqn bc] (compilable-bicond-axiom bc))))))
                                    (bicond-axioms (rd (map second compilable-elements)))
                                    (equational-axioms (list-diff axioms (rd (map first compilable-elements)))))
                                (join equational-axioms bicond-axioms))))
       (equation-lists (map get-defining-axioms fsyms))
#       (equation-lists (map (lambda (f) (rearrange-and-simplify (get-defining-axioms f))) fsyms))
       (clause-lists (map (lambda (equation-list)
                             (map (lambda (e) (flatten (compile-equation-into-single-clause' e mod-path "reduce"))) equa
                           equation-lists))
       (bodies (map (lambda (pcl)
                       (match pcl
                         ([[param-list fsym] clause-list]
                            (let ((clause-list' (map first clause-list))
#                                 (last-clause (join "(_ (make-term " (val->string fsym) " [" param-list "]))"))
                                  (term-to-be-reduced (join "(" (val->string fsym) " " param-list ")"))
                                  (last-clause (join "(_ " term-to-be-reduced ")"))
                                  (clause-list' (join clause-list' [last-clause]))
                                  (needs-sub (for-some (map second clause-list) (lambda (x) x)))
                                  (body1 (join lp "match " lb param-list rb newline tab tab (separate clause-list' (joi
                                  (body1' (check (needs-sub (join "(let ((theta-cell (cell empty-sub))) " body1 ")"))
                                                 (else body1)))
                                  (body1'' (join "(try (resolve-redex " term-to-be-reduced ")\n
" body1' ")")))
                             body1''))))
                     (list-zip param-lists' clause-lists)))
       (lams (map (lambda (pbl)
                     (match pbl
                       ([[param-list fsym] body] (join lp "lambda " lp param-list rp newline tab body rp))))
                  (list-zip param-lists' bodies)))
       (bindings (map (lambda (name-and-lam)
                         (match name-and-lam
                            ([name lam] (join lp name blank lam rp))))
                      (list-zip names lams)))
       (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb r
    res))

(define (cd f)
  (println (compile-symbols-with-default [f] [])))


(define (vars-to-rid left-args guard-free-vars)
  (letrec ((loop (lambda (rem-args previous-args results)
                   (match rem-args
                      ([] results)
                      ((list-of (some-var v) more)
                        (let ((V (vars* (join previous-args more))))
                           (check ((|| (member? v V) (member? v guard-free-vars))
                                        (loop more (add v previous-args) results))
                                  (else  (loop more (add v previous-args) (add v results))))))
                      ((list-of arg more)
                         (loop more (add arg previous-args) results))))))
    (loop left-args [] [])))


(define (econd f eqns)
  (let ((eqns (map rename eqns))
        (N (arity-of f))
        (g (get-symbol f))
        (freshvars (map (lambda (_) (fresh-var)) (1 to N)))
        (get-cond (lambda (eqn)
                    (match eqn
                       ((forall (some-list uvars) (if (some-sent p) (= (left as ((val-of g) (some-list args))) R)))
                          (let ((pfv (free-vars p))
                                (vars-to-discard (vars-to-rid args pfv))
                                (evars (filter-out (dedup (join (vars left) pfv))
                                                   (lambda (v)
                                                     (member? v vars-to-discard))))
                                (conds1 (map-select
                                            (lambda (var-arg-pair)
```

```
                                               (match var-arg-pair
                                                  (([v a] where (negate (member? a vars-to-discard))) (= v a))
                                                  (_ ())))
                                            (list-zip freshvars args)
                                            (unequal-to ())))
                                 (and-conds (join conds1 [p]))
                                 (body  (match and-conds
                                          ([(some-sent q)] q)
                                          ([] true)
                                          (_ (and and-conds)))))
                             (match evars
                               ([] body)
                               (_ (exists* evars body)))))
                        ((forall (some-list uvars) (p as (= (left as ((val-of g) (some-list args))) R)))
                           (let ((vars-to-discard (vars-to-rid args []))
                                 (evars (filter-out (vars left)
                                                    (lambda (v)
                                                      (member? v vars-to-discard))))
                                 (conds1 (map-select
                                            (lambda (var-arg-pair)
                                               (match var-arg-pair
                                                  (([v a] where (negate (member? a vars-to-discard))) (= v a))
                                                  (_ ())))
                                            (list-zip freshvars args)
                                            (unequal-to ())))
                                 (and-conds (join conds1))
                                 (body  (match and-conds
                                          ([] true)
                                          ([(some-sent q)] q)
                                          (_ (and and-conds)))))
                             (match evars
                               ([] body)
                               (_ (exists* evars body))))))))
           (total-body (or (map get-cond eqns)))
           (body-fv (map var->string (free-vars total-body)))
           (freshvars' (filter freshvars (lambda (v) (member? (var->string v) body-fv)))))
   (forall* freshvars' total-body)))


(define (dcond f eqns)
  (let ((eqns (map rename eqns))
        (N (arity-of f))
        (g (get-symbol f))
        (freshvars (map (lambda (_) (fresh-var)) (1 to N)))
        (make-cond (lambda (freshvars eqn)
                     (match eqn
                        ((forall (some-list uvars) (if (some-sent p) (= (left as ((val-of g) (some-list args))) R)))
                           (let ((pfv (free-vars p))
                                 (vars-to-discard (vars-to-rid args pfv))
                                 (evars (filter-out (dedup (join (vars left) (free-vars p)))
                                                    (lambda (v)
                                                      (member? v vars-to-discard))))
                                 (conds1 (map-select
                                            (lambda (var-arg-pair)
                                               (match var-arg-pair
                                                  (([v a] where (negate (member? a vars-to-discard))) (= v a))
                                                  (_ ())))
                                            (list-zip freshvars args)
                                            (unequal-to ())))
                                 (and-conds (join conds1 [p]))
                                 (body  (match and-conds
                                          ([(some-sent q)] q)
                                          ([] true)
                                          (_ (and and-conds)))))
                             (match evars
                               ([] body)
                               (_ (exists* evars body)))))
                        ((forall (some-list uvars) (p as (= (left as ((val-of g) (some-list args))) R)))
                           (let ((vars-to-discard (vars-to-rid args []))
                                 (evars (filter-out (dedup (join (vars left) (free-vars p)))
```

```
                                                       (lambda (v)
                                                         (member? v vars-to-discard))))
                                  (conds1 (map-select
                                               (lambda (var-arg-pair)
                                                  (match var-arg-pair
                                                     (([v a] where (negate (member? a vars-to-discard))) (= v a))
                                                     (_ ())))
                                               (list-zip freshvars args)
                                               (unequal-to ())))
                                  (and-conds (join conds1))
                                  (body  (match and-conds
                                             ([] true)
                                             ([(some-sent q)] q)
                                             (_ (and and-conds)))))
                              (match evars
                                ([] body)
                                (_ (exists* evars body))))))))
        (make-cond-2 (lambda (freshvars eqn1 eqn2)
                       (forall* freshvars
                                 (if (make-cond freshvars eqn1)
                                     (not (make-cond freshvars eqn2)))))))
    (letrec ((loop (lambda (eqns res)
                      (match eqns
                         ([] (rev res))
                         ((list-of eqn more) (let ((res' (map (lambda (eqn2)
                                                                   (make-cond-2 freshvars eqn eqn2))
                                                                more)))
                                                  (loop more (join res' res))))))))
        (loop eqns [])))))


(define (fun-def-conds-e f)
  (match f
    ((some-symbol _) (econd f (defining-axioms f)))
    ((some-proc _)
      (let ((fsym (root (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 (arity-of f)))))))
        (econd fsym (defining-axioms fsym))))))

(define get-ec fun-def-conds-e)

(define (fun-def-cond-d0 f)
  (let ((res  (match (dcond f (defining-axioms f))
                 ([(some-sent p)] [p])
                 ([] [true])
                 ((some-list L) L)))
        (clean-up (lambda (res)
                    (match res
                       ((forall (some-list uvars) body)
                          (let ((body-fv (free-vars body))
                                (uvars' (filter uvars (lambda (v) (member? v body-fv)))))
                            (forall* uvars' body))))))
    (map clean-up res)))

(define (fun-def-conds-d f)
  (match f
    ((some-symbol _) (fun-def-cond-d0 f))
    ((some-proc _)
      (let ((fsym (root (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 (arity-of f)))))))
         (fun-def-cond-d0 fsym)))))

(define get-dc fun-def-conds-d)

(define (def-obligations f)
   (let ((p (get-ec f))
         ([error at_least_one_conditional_equation] (check-fun-def f))
         (_ (check ((&& (negate error) (negate at_least_one_conditional_equation))
                      (print "\nThe definition of " f " is well-formed; the following conditions hold.\n"))
                   (else ())))
         (qs (get-dc f)))
     (add p qs)))
```

```
(define fun-def-conds def-obligations)

(define (spf s props N)
    (!sprove-from s props [['poly true] ['subsorting false] ['max-time N]]))

(define (vpf s props N)
    (!vprove-from s props [['poly true] ['subsorting false] ['max-time N]]))

(define (prove goal premises)
  (!spf goal premises 100))

define (derive-from goal premises options)  :=
  let {max := try {(options 'max-time) | 100};
       c := try   {(options 'used-premises) | () }}
    check {(equal? (options 'atp) 'vampire) =>
            match c {
              (some-cell _) => (!vprove-from goal premises [c ['poly true] ['subsorting false] ['max-time max]])
           | _ => (!vpf goal premises max)}
         | else => (!spf goal premises max)}

define top-smt-solve := smt-solve

define make-all-terms-thunk-cell := (cell ())

module SMT {

define (in x range) :=
  match range {
     [l h] => ((x <= h) & (l <= x))
  }

define (check-option-key-and-value k v) :=
 let {error := lambda (k v) (error (join "\nWrong key-value entry given as an option: " (val->string k) " := " (val->s
    match k {
      'results => match v { (some-table _) => () | _ => (error k v) }
    | 'solver => check {(member? v ['yices 'cvc]) => () | else => (error k v)}
    | _ => ()
    }

#==================================================================================

define (basic-augment-answer-table answer-table eqn) :=
  match eqn {
    (= l r) => let {prior-terms := try { (HashTable.lookup answer-table l) | [] }}
                 (HashTable.add answer-table [l --> (add r prior-terms)])
  }

define (get-ints-from-code code-string) :=
  let {toks := (tokenize code-string " \t-)(");
       numerals := (dedup (filter toks all-digits?))}
    (join (map string->num numerals)
          (map lambda (n) (- (string->num n)) numerals))

define (get-reals-from-code code-string) :=
  let {toks := (tokenize code-string " \t)(-");
       numerals := (dedup (filter toks lambda (t) (for-each t lambda (c) (|| (digit? c) (equal? c `.))))))}
    (join (map string->num numerals)
          (map lambda (n) (- (string->num n)) numerals))

define (get-metaids-from-code code-string) :=
  let {toks := (tokenize code-string " \t)(");
       ids := (dedup (filter toks lambda (t) try { (seq (id->string (evaluate t)) true) | false }))}
    (map (o id->string evaluate) ids)

define (non-datatype-sort D) := (negate (datatype-sort? D))

define (make-all-equations fsym code-string) :=
  let {input-domains := (all-but-last (get-signature fsym));
       #_ :=  (print "\nAbout to make all equations for fsym: " fsym ", with input-domains: " (separate input-domains
```

```
       make-all-terms := (ref make-all-terms-thunk-cell);
       make-all-terms' := lambda (sort)
                              check {(equal? sort "Int") => (get-ints-from-code code-string)
                                   | (equal? sort "Real") => (get-reals-from-code code-string)
                                   | (equal? sort "Ide") => (get-metaids-from-code code-string)
                                   | else => (make-all-terms sort)}}
    check {(&& false (for-some input-domains non-datatype-sort)) => []
         | else => let {all-inputs := (cprod* (map make-all-terms' input-domains));
                        #_ := (print "\n\tLength of all-inputs for symbol " fsym " is: " (length all-inputs));
                        get-answer := lambda (input)
                                        try {
                                         let {inputs := (separate (map val->string input) " ");
                                              exp := (join "(" code-string inputs ")");
                                              result := (evaluate exp)}
                                           (= (make-term fsym input) result)
                                        | () };
                        results := (map-select get-answer all-inputs (unequal-to ()));
                        #_ := (print "\n\tThere are " (length results) " resulting equations for symbol " fsym);
                        #_ := (print "\nAnd the results:\n" (separate (map val->string results) "\n"));
                        _ := ()
                        }
                     results}

define (cvc-code->equations code-table answer-table) :=
   (map-proc lambda (key-val-pair)
               match key-val-pair {
                 [fsym code] => let {eqns := (make-all-equations fsym code)}
                                  (map-proc lambda (eqn) (basic-augment-answer-table answer-table eqn) eqns)
               }
             (HashTable.table->list code-table))

define augment-cvc-answer-table :=
  lambda (answer-table)
     let {#_ := (print "\nEntering augment-cvc-answer-table, size of incoming answer-table: " (HashTable.size answer-t
          L := (table->list answer-table);
          code-table := (HashTable.table);
          _ := (map-proc lambda (key-val-pair)
                           match key-val-pair {
                             ([(some-list f-str) (some-list code)] where (&& (string? f-str) (string? code))) =>
                               let {_ := (HashTable.remove answer-table f-str);
                                    f := (string->symbol f-str)}
                                 (HashTable.add code-table [f --> code])
                           | _ => ()
                           }
                         L);
         #_ := (print "\nHere's code-table:\n" code-table "\nand here's answer-table:\n" answer-table);
         _ := (cvc-code->equations code-table answer-table);
         #_ := (print "\nExiting augment-cvc-answer-table, size of outgoing answer-table: " (HashTable.size answer-tabl
         _ := ()
         }
       ()

define (extract-cvc-extension-table answer-table) :=
 let {extension-table := (HashTable.table);
      L := (HashTable.table->list answer-table);
      _ := (map lambda (key-val-pair)
                  match key-val-pair {
                     [((some-term l) where (ground? l)) term-list] =>
                         match (dedup term-list) {
                            [(some-term r)] => (HashTable.add extension-table [l --> r])
                          | res => () #(print "\nCould not extract a sole answer for the term " l ", got this instead:
                         }
                  }
                L)}
   extension-table

define (smt-solve C options) :=
  let {ht := (HashTable.table);
       _ := (map-proc lambda (k)
                        let {_ := (check-option-key-and-value k (options k))}
```

```
                           (HashTable.add ht [k --> (options k)])
                      (Map.keys options));
       #_ := (print "\nHere's ht right before the solving: " ht);
       #_ := (print "\nAnd here's options right before the solving: " options);
       res := (top-smt-solve C ht);
       #_ := (print "\nAnd here's ht right AFTER the solving: " ht);
       #_ := (print "\nAnd here's options right after the solving: " options);
       post-process-cvc-answers := lambda (t)
                                     let {#_ := (print "\nHere's (downcase-string (options 'solver)): " (downcase-stri
                                          cond1 := (prefix? "'cvc" (downcase-string (val->string (options 'solver))));
                                          cond := (&& cond1
                                                       (negate (prefix? "'un" (downcase-string (val->string t)))))}
                                      try {check {cond =>
                                                   let {#_ := (print "\nHere's (options 'results): " (options 'results)
                                                        _ := (augment-cvc-answer-table (options 'results))
                                                        }
                                                     t
                                                 | else => t}
                                        | t}}
   match res {
     (some-term t) => let {#_ := (print "\nThe main call to smt-solve produced this TERM: " (val->string t));
                           _ := (post-process-cvc-answers t)}
                        t
   | _ => let {#_ := (print "\nThe main call to smt-solve produced this result: " (val->string res));
               final-res := 'Satisfiable;
               _ := (post-process-cvc-answers final-res)}
            final-res
   }

define (replace-all-subterms t ht) :=
 check {(|| (meta-id? t) (numeral? t)) => t
 | else =>
  try { (first (table-lookup ht t))
      | match t  {
          ((some-symbol f) (some-list args)) =>
            (make-term f (map lambda (s) (replace-all-subterms s ht)
                              args))
        | _ => t
        }
      }}

define (solve p) :=
  let {ht := (table)}
    match (top-smt-solve p (table [['results --> ht] ['solver --> 'yices]])) {
      (some-term t) => t
    | _ => let {pairs := (map lambda (pair)
                                 match pair {
                                    [x vals] => [x (first vals)]
                                   }
                                (filter (table->list ht)
                                       lambda (p) (var? first p)));
                pairs' := (map (lambda (p) [(first p) (replace-all-subterms (second p) ht)]) pairs)}
             (make-sub pairs')
    }

define built-in-symbols :=
  (map string->symbol ["<" ">" "<=" ">=" "=" "+" "-" "*" "/"])

define (built-in? f) := (member? f built-in-symbols)

define (cost-term-leaves cost-term) :=
  match cost-term {
    ((some-symbol f) (some-list args)) =>
         check {(built-in? f) => (flatten (map cost-term-leaves args))
              | else => [cost-term]}
  | _ => [cost-term]
  }

define (apply-solution sub s) :=
  let {res := (sub s)}
```

```
    match (res equal? s) {
      true => ()
    | _ => res
    }

define (apply-solution-new ht s) :=
  let {res := try {(HashTable.lookup ht s) | true}}
    match res {
      true => ()
    | (list-of h _) => h
    | (some-term t) => t
    | v => let {_ := (print "\nGot the following value by applying ht to the term " s ": " v)}
             ()
    }

define (get-cost solution cost-terms) :=
  let {costs := (map lambda (cost-term)
                       (apply-solution solution cost-term)
                     cost-terms)}
   (eval (foldl + 0 costs))

define (get-cost-new ht cost-terms) :=
  let {#_ := (print "\nCost-terms: " cost-terms);
       costs := (map lambda (cost-term)
                       (apply-solution-new ht cost-term)
                     cost-terms);
       #_ := (print "\nAnd their costs: " costs);
       _ := ()}
    (eval (foldl + 0 costs))


define (midpoint l h) := (l plus ((h minus l) div 2))

define (solve-and-minimize constraint cost-term max-cost) :=
 let {counter := (cell 0);
      cost-terms := (cost-term-leaves cost-term)}
   letrec {loop := lambda (lo hi)
                     let {_ := (inc counter)}
                     check {(hi less? lo)  => 'Unsatisfiable
                          | (lo equal? hi) => (solve (and constraint (cost-term = hi)))
                          | else => let {mid := (lo midpoint hi);
                                         cost-constraint := (cost-term in [lo mid])}
                                    match (solve (and constraint cost-constraint)) {
                                      (some-sub sub) => check {(less? lo mid) =>
                                                                 let {total-cost := (get-cost sub cost-terms)}
                                                                  (loop lo total-cost)
                                                             | else => sub}
                                    | _ => (loop (plus mid 1) hi)
                                    }}}
      match (loop 0 max-cost) {
        (some-sub sub) => let {_ := (print "\nTotal cost: " (get-cost sub cost-terms) "\n")}
                            sub
      | res => res
      }

define (range->string a b suffix) :=
 (join "[" (val->string a) "," (val->string b) "]" suffix)


define (solve-and-minimize-new constraint cost-term max-cost options) :=
 let {counter := (cell 0);
      cost-terms := (cost-term-leaves cost-term);
      get-minutes := lambda (s)
                       (div s 60.0);
      solve := lambda (core-constraint cost-constraint)
                 let {#_ := (print "\nSolving with cost constraint: " cost-constraint "\n");
                      _ := (HashTable.clear (options 'results));
                      #_ := (print "\nEntering solver...");
                      res := (smt-solve (and core-constraint cost-constraint) options);
                      #_ := (print "\nExiting solver...");
```

```
                      _ := (augment-cvc-answer-table (options 'results))}
                   res}
   letrec {loop := lambda (lo hi iteration)
                      let {_ := (inc counter);
                           _ := (print (join "\n---------- Iteration #" (val->string iteration) " ----------\n"))}
                      check {(hi less? lo)  => 'Unsatisfiable
                           | (lo equal? hi) => let {_ := (print "Cost constraint specifying that the cost term is in "
                                                 (solve constraint (cost-term = hi))
                           | else => let {mid := (lo midpoint hi);
                                          #_ := (print "\nGiven lo: " lo ", hi: " hi ", and midpoint: " mid ".");
                                          _ := (print (join "Cost constraint specifying that the cost term is in " (ra
                                          cost-constraint := (cost-term in [lo mid]);
                                          t1 := (time);
                                          solver-result := (solve constraint cost-constraint);
                                          t2 := (time);
                                          elapsed := (get-minutes (minus t2 t1))}
                                      match solver-result {
                                        'Unsatisfiable => let {_ := (print "\nFailed in " elapsed
" minutes, will now try again with a properly adjusted range...\n")}
                                                          check {(iteration equals? 1) => 'Unsatisfiable
                                                                | else => (loop (plus mid 1) hi (plus iteration 1))}
                                      | 'Satisfiable => check {(less? lo mid) =>
                                                                 let {total-cost := (get-cost-new (options 'results) co
                                                                      _ := (print (join "\nSuccess in " (val->string e
                                                                                   (val->string total-cost)
                                                                                    ", will now try again with a proper
                                                                     (loop lo total-cost (plus iteration 1))
                                                                | else => 'Satisfiable}
                                      | other =>  let {str := (take (val->string other) 200);
                                                       _ := (print (join "\nIndeterminate result after " (val->string e
" minutes: " str));
                                                       _ := (print ", will now try again with a properly adjusted rang
                                               (loop (plus mid 1) hi (plus iteration 1))
                                      }}}
      match (loop 0 max-cost 1) {
        'Satisfiable => let {_ := (print "\nTotal cost: " (get-cost-new (options 'results) cost-terms) "\n")}
                          'Satisfiable
      | res => res
      }

define (holds? p) :=
  match (solve (not p)) {
    'Unsatisfiable => true
  | _ => false
  }

set-precedence solve 2
set-precedence holds? 2

(define (sc->string sc)
  (match sc
    (and "and")
    (or  "or")))

(define (make-constraint sc strings)
   (let ((sc-string (sc->string sc)))
     (match strings
       ([s] s)
       (_ (join lparen sc-string blank strings)))))

(define (sum-all terms)
  (match terms
    ([x] x)
    ((list-of x (bind rest (list-of _ _))) (+ x (sum-all rest)))))

(define (sum n)
  (check ((less? n 1) 0)
         (else (plus n (sum (minus n 1))))))

(define (make-cost-term t)
```

```
  (match t
   ((some-var x) (string->var (join "cost" (var->string x) ":Int")))
   (((some-symbol f) (some-list _)) (string->var (join "cost" (symbol->string f) ":Int")))))

(define (make-constraint n)
  (let ((span (from-to 1 n))
        (vars (map (lambda (_) (fresh-var "Int")) span))
        (counter (cell 1))
        (cost (cell 1))
        (range-sentences-and-var-values-1
            (map (lambda (v)
                   (let ((low ((inc counter) times 10))
                         (hi  (plus low 5)))
                     [(in v [low hi])
                      (= v (plus low 1))]))
                 vars))
        (range-sentences-and-var-values-2
           (map (lambda (v)
                  (let ((low ((inc counter) times 100))
                        (hi  (plus low 10)))
                    [(in v [low hi])
                     (= v (plus low 2))]))
                vars))
        ([range-sentences-1 var-values-1] (unzip range-sentences-and-var-values-1))
        ([range-sentences-2 var-values-2] (unzip range-sentences-and-var-values-2))
        (constraint (or (and* range-sentences-1) (and* range-sentences-2)))
        (mid (midpoint 1 n))
        (values-1 (take var-values-1 mid))
        (values-2 (second (split-list var-values-2 mid)))
        (values (join values-1 values-2))
        (cost-constraints (map (lambda (var-val)
                                 (match var-val
                                   ((= v val) (let ((v-cost-term (make-cost-term v)))
                                                (ite (= v val) (= v-cost-term 0) (= v-cost-term (inc cost)))))))
                               values))
       (cost-variables (map make-cost-term vars))
       (cost-term (sum-all cost-variables))
       (cost-constraint (and* cost-constraints))
       (max-cost (sum (length vars))))
    [constraint vars cost-constraint cost-term max-cost]))

(define [constraint-30 vars-30 cost-constraint-30 cost-term-30 max-cost-30] (make-constraint 30))

(define [constraint-100 vars-100 cost-constraint-100 cost-term-100 max-cost-100] (make-constraint 100))

# (running-time (lambda () (solve-and-minimize (and constraint-30 cost-constraint-30) cost-term-30 max-cost-30)) 0)

# (solve-and-minimize (and constraint-100 cost-constraint-100) cost-term-100 max-cost-100)

}

EOF
(println (cd [N.Plus] ["N"]))
(define css compile-symbols-simple)
(define csd compile-symbols-with-default)

(println (css [append] []))
(println (cs [append] []))

(println (css [mem] []))

(println (css [I] []))

(define ri (iff (= ?A_1062
        null)
     (forall ?v1080:'S
       (iff (in ?v1080
               ?A_1062)
            (in ?v1080
               null)))))
```

```
(define sri (iff (= ?A_1062
        null)
     (forall ?v1085:'T
       (iff (in ?v1085
                ?A_1062)
            (in ?v1085
                null)))))

(assume ri
  (!sort-instance ri sri))

(load-file "cc")
```