# lib/basic/tableaux.ath

```
## An implementation of semantic tableaux for first-order logic
## using infinite streams instead of unification. The method
## uses weaving of infinite streams to get a fair and seemingly
## relatively efficient strategy for choosing terms with which
## to instantiate the universal generalizations.

(define (consistent? literals)
  (match literals
    ((split _ (list-of A (split _ (list-of (not A) _)))) false)
    ((split _ (list-of (not A) (split _ (list-of A _)))) false)
    ((split _ (list-of false _)) false)
    ((split _ (list-of (not true) _)) false)
    (_ true)))

(define (dual L)
  (match L
    ((some-atom A) (not A))
    ((not (some-atom A)) A)))


(define empty-stream [])
(define stream-head head)
(define (stream-tail e) ((head (tail e))))
(define empty-stream? null?)

(define (stream-cons x s)
  [x (lambda () s)])

(define (list->stream L)
  (match L
    ([] empty-stream)
    ((list-of x rest) [x (lambda () (list->stream rest))])))

(define (stream-nth stream i)
  (check ((less? i 2) (stream-head stream))
         (else (stream-nth (stream-tail stream) (minus i 1)))))

(define (weave-streams s1 s2)
  (check ((empty-stream? s1) s2)
         (else [(stream-head s1) (lambda () (weave-streams s2 (stream-tail s1)))])))

(define (map-stream f s)
  (check ((empty-stream? s) s)
         (else [(f (stream-head s))
                (lambda () (map-stream f (stream-tail s)))])))

(define (append-streams s1 s2)
  (check ((empty-stream? s1) s2)
         (else [(stream-head s1)
                (lambda () (append-streams (stream-tail s1) s2))])))

(define (all-from i) [i (lambda () (all-from (plus i 1)))])

(define all-numbers (all-from 0))

(define (make-var n) (string->var (join "a" (symbol->string n))))

(define all-variables (map-stream make-var all-numbers))

(define (all-instances P v)
  (map-stream (lambda (a) (replace-var v a P)) all-variables))

(define (specialize-props props x M)
  (dletrec ((loop (method (L specs)
                    (dmatch L
                      ([] (!M specs))
                      ((list-of (some-sent P) rest)
                          (dlet ((spec (!uspec P x)))
```

```
                              (!loop rest [spec (lambda () specs)])))))))
    (!loop props empty-stream)))

(define (refute props terms)
  (dlet ((i (cell 0))
         (limit 10000)
         (all-terms (weave-streams (list->stream terms) all-variables)))
    (dletrec
      ((M (method (props literals evars ugens)
            (dlet ((_ (check ((greater? (inc i) limit) (halt))
                             (else ())))
                   (first (stream-head props))
                   (dm (method (premise) (!force (app-dm premise))))
                   (rest  (stream-tail props)))
              (dmatch first
                ((some-method thunk) (!unary-conj-case (!thunk) rest literals evars ugens))
                ((bind P (and P1 P2)) (!bin-conj-case (!left-and P) (!right-and P) rest literals evars ugens))
                ((not (not P)) (!unary-conj-case (!dn (not (not P))) rest literals evars ugens))
                ((bind P (not (or p1 p2))) (dlet ((_ (!dm P))
                                                  (left (!left-and (and (not p1) (not p2))))
                                                  (right (!right-and (and (not p1) (not p2)))))
                                             (!bin-conj-case left right rest literals evars ugens)))
                ((bind P (not (if _ _))) (!bin-conj-case (!neg-cond1 P) (!neg-cond2 P) rest literals evars ugens))
                ((bind P (iff _ _)) (!bin-conj-case (!left-iff P) (!right-iff P) rest literals evars ugens))
                ((bind P (or _ _)) (!disj-case P rest literals evars ugens))
                ((bind P (not (and p1 p2))) (!disj-case (!dm P) rest literals evars ugens))
                ((bind P (if _ _)) (!disj-case (!cond-def P) rest literals evars ugens))
                ((bind P (not (iff _ _))) (!disj-case (!neg-bicond P) rest literals evars ugens))
                ((bind P (forall (list-of _ _) _))
                   (!M (weave-streams rest
                         (map-stream (lambda (v) (method () (dtry (!uspec P v) (!true-intro))))
                                     (append-streams evars all-terms)))
                       literals evars (add P ugens)))
                 ((bind P (exists x Q))
                    (dlet ((w (fresh-var (sort-of x))))
                      (with-witness w P
                        (!specialize-props ugens w
                           (method (specs)
                             (!M (append-streams specs
                                   [(replace-var x w Q) (lambda () rest)]) literals [w (lambda () evars)] ugens))))))
                 ((bind P (not (forall (list-of _ _) _))) (!unary-conj-case (!qn P) rest literals evars ugens))
                 ((bind P  (not (exists (list-of _ _) _))) (!unary-conj-case (!qn P) rest literals evars ugens))
                 (L (dlet ((L' (dual L)))
                      (dcheck ((member? L' literals) (!comm-absurd L L'))
                              ((equal? L false) (!from-false false))
                              ((equal? L (not true)) (!absurd (!true-intro) L))
                              (else (!M rest (add L literals) evars ugens)))))))))
       (bin-conj-case (method (P1 P2 stream literals evars ugens)
                        (!M [P1 (lambda () [P2 (lambda () stream)])] literals evars ugens)))
       (unary-conj-case (method (P stream literals evars ugens)
                          (!M [P (lambda () stream)] literals evars ugens)))
       (disj-case (method (P s literals evars ugens)
                    (dmatch P
                      ((or P1 P2) (!cases P (assume P1 (!M (stream-cons P1 s) literals evars ugens))
                                            (assume P2 (!M (stream-cons P2 s) literals evars ugens)))))))
       (!M props [] empty-stream [])))))




(define (taut p)
 (!by-contradiction' p
    (assume (not p)
      (dlet ((th (!refute (stream-cons (not p) empty-stream)
                          (choice-prop-subterms p))))
        (!claim th)))))
```