lib/basic/util.ath

```
define (map f L) :=
  letrec {loop := lambda (L results)
                  match L {
                    [] => (rev results)
                  | (list-of x rest) => (loop rest (add (f x) results))
                  }}
    (loop L [])

(define o (lambda (f g) (lambda (x) (f (g x)))))

(define first head)

(define second (o first tail))

(define third (o second tail))

(define fourth (o third tail))
(define fifth  (o fourth tail))
(define sixth  (o fifth tail))
(define seventh  (o sixth tail))
(define eigth  (o seventh tail))

(define last (o head rev))

(define list-last (o head rev))

module HashTable {
define table := table
define lookup := table-lookup
define remove := table-remove
define clear := table-clear
define size := table-size
define table->string := table->string
define table->list := table->list
define (in x T) := try { let {_ := (table-lookup T x)} true | false }
define (keys T) := (map first (table->list T))
define (map-to-keys T f) :=
   letrec {loop := lambda (pairs res)
                   match pairs {
                     [] => res
                   | (list-of [k _] more) => (loop more (add (f k) more))
                   }}
      (loop (table->list T) [])
define add := table-add
}

module Map {

define make := make-map
define add := map-add
define remove := map-remove
define size := map-size
define (empty? m) := ((size m) equal? 0)
define keys := map-keys
define values := map-values
define key-values := map-key-values
define map-to-values := map-to-values
define map-to-key-values := map-to-key-values
define apply-to-key-values := map-to-values
define apply-or-same := lambda (m x) try { (m x) | x }
define apply-to-both := lambda (m f)
                          (make-map (map lambda (pair)
                                          match pair {
                                            [k v] => [(f k) (f v)]
                                          }
                                        (map-key-values m)))
define foldl := map-foldl
}
```

```
69
70  define [table? map?] :=
71    [lambda (x) match x { (some-table _) => true | _ => false }
72     lambda (x) match x { (some-map _) => true | _ => false }]
73
74  define prim-sort := sort
75
76  define prop? := sentence?
77
78  (define (complement p)
79    (match p
80      ((not q) q)
81      (_ (not p))))
82
83  (set-precedence complement 300)
84
85  (define (complements? p q)
86    (|| (equal? p (complement q))
87        (equal? q (complement p))))
88
89  define equals? := equal?
90
91  (define added-to add)
92  (set-precedence added-to 115)
93
94  (define (resolve-redex s)
95    (let ((res (check ((prop? s)
96                       (check ((holds? s) true)
97                              ((holds? (complement s)) false)
98                              (else ())))
99                      (else ()))))
100     (match res
101       (() (match (fetch (lambda (p)
102                           (match p
103                             ((= (val-of s) _) true)
104                             (_ false))))
105             ((= _ result) result)))
106       (_ res))))
107
108 define debug :=
109   lambda (s v)
110     let {_ := (print (join s " = "))}
111     (write v)
112
113 (define id (lambda (x) x))
114
115 (declare (--> <-- <-->) Boolean)
116
117 (define (try-looking-up x ht)
118   (try [(table-lookup ht x)]
119        ()))
120
121
122
123 (define (dmemoize-unary M)
124   (let ((ht (table 101)))
125     (method (x)
126        (dmatch (try-looking-up x ht)
127          ([y] (!claim y))
128          (_ (dlet ((y (!M x))
129                    (_ (table-add ht [x --> y])))
130               (!claim y)))))))
131
132
133 (define absurd0 absurd)
134
135 (define (absurd p q)
136   (dmatch [p q]
137     ([(some-sentence prem) (not prem)] (!absurd0 p q))
138     ([(not (some-sentence prem)) prem] (!absurd0 q p))
```

```
    (_ (!absurd0 q p))))

(define (ab) (get-ab))

(define (ab') (fetch-all (lambda (_) true)))

(define (flip b) match b {true => false | false => true})

(define (switch sc negated)
  match [negated sc] {
    [true and] => or
  | [true or] => and
  | [true forall] => exists
  | [true exists] => forall
  | _ => sc
  })

(define (map f L)
  (letrec ((loop (lambda (L results)
                   (match L
                     ([] (rev results))
                     ((list-of x rest) (loop rest (add (f x) results)))))))
    (loop L [])))

(define (nnf-linear p negated?)
  match p {
    (some-atom _) => check {negated? => (not p) | else => p}
  | (not q) => (nnf-linear q (flip negated?))
  | (p1 ==> p2) => (nnf-linear (or (not p1) p2) negated?)
  | (p1 <==> p2) => (nnf-linear (or (and p1 p2)
                                    (and (not p1) (not p2))) negated?)
  | ((some-sent-con sc) (some-list args)) => ((switch sc negated?) (map (lambda (p) (nnf-linear p negated?)) args))
  | ((some-quant q) (some-var v) body) =>
        ((switch q negated?) v (nnf-linear body negated?))
  })

##(define (nnf p) (nnf-linear p false))

(define sprove-from0 sprove-from)

(define vprove-from0 vprove-from)

(define spf
  (method (goal premises)
    (!sprove-from goal premises [['poly true] ['subsorting false] ['max-time 10000]])))

(define vpf
  (method (goal premises)
    (!vprove-from goal premises [['poly true] ['subsorting false] ['max-time 1000]])))

(define (prove goal) (!spf goal (ab)))
(make-private "prove")

(define prove ())
(define tspf
  (method (goal premises t)
    (!sprove-from goal premises [['poly true] ['subsorting false] ['max-time t]])))

(define vpf
  (method (goal premises)
    (!vprove-from goal premises [['poly true] ['subsorting false] ['max-time 60]])))

(define mvpf
  (method (goal premises)
    (!vprove-from goal premises [['poly false] ['subsorting false] ['max-time 60]])))

(define mspf
  (method (goal premises)
    (!sprove-from goal premises [['poly false] ['subsorting false] ['max-time 60]])))
```

```
(define vderive vpf)
(define sderive spf)

(define (qvar-of P)
  (match P
    (((some-quant Q) x _) x)))

(make-private "qvar-of")

(define (property-from-induction-goal g)
  (match g
    ((forall (some-var v) body)
       (lambda (x) (replace-var v x (rename body))))))

(make-private "property-from-induction-goal")

## Old definition of get-symbol:
(define (get-symbol f)
  (match f
    ((some-symbol _) f)
    ((some-proc _) (string->symbol (proc-name f)))
    (_ f)))

## Experimental new definition of get-symbol, Sept. 12, 2015:

(load-file "list.ath")

define (list->table L) :=
  let {T := (HashTable.table);
       _ := (map-proc lambda (x) (HashTable.add T [x --> true])
                      L)}
    T

define (pairs->table L) :=
  let {T := (HashTable.table);
       _ := (map-proc lambda (pair)
                        match pair {
                          [x y] => (HashTable.add T [x --> y])
                        }
                      L)}
    T


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

(define (get-symbol f)
  (match f
    (((some-term t) where (constant? f)) (root f))
    ((some-symbol _) f)
    ((some-proc _) (let ((ar (arity-of f)))
                       (match (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 (arity-of f))))
                         ((some-term t) (root t))
                         (_ (string->symbol (proc-name f))))))
    (_ f)))

(declare <> ((S) -> (S S) Boolean))

(set-precedence <> 100)

#(define (===> p q) (if p q))
#(define (<===> p q) (iff p q))
#(define (<=== p q) (if q p))

(define ==> if)
(define <==> iff)
(define & and)
#(define (& p1 p2) (and p1 p2))

(define | or)
```

```
#(define (| p1 p2) (or p1 p2))
(define (<== p q) (if q p))

#(define (==> p q) (if p q))
#
#(define (<===> p q) (iff p q))



(define (~ p) (not p))
(set-precedence ~ 50)
(set-precedence <== 10)
#(set-precedence (===> <=== ==> <==) 10)
#(set-precedence (<===> <==>) 10)
(set-precedence & 30)
(set-precedence | 20)

(define match-sentences match-props)

(define (negation? p)
  (match p
    ((not _) true)
    (_ false)))

(define (conjunction? p)
  (match p
    ((and (some-list _)) true)
    (_ false)))

(define (disjunction? p)
  (match p
    ((or (some-list _)) true)
    (_ false)))

(define (conditional? p)
  (match p
    ((if _ _) true)
    (_ false)))

(define (biconditional? p)
  (match p
    ((iff _ _) true)
    (_ false)))

(define (quant? p)
  (match p
   (((|| forall exists) _ _) true)
   (_ false)))

(define (mark str)
  (let (([str N] (check ((char? str) [[str] 40])
                        (else [str 20]))))
    (letrec ((loop (lambda (i res)
                     (check ((equal? i 0) res)
                            (else (loop (minus i 1) (join str res)))))))
      (print (join "\n" (loop N str) "\n")))))

(define (negate t)
  (match t
    (true false)
    (false true)))

(define (matches? x y)
  (let ((test-with (lambda (f)
                     (match (f x y)
                       ((some-sub _) true)
                       (_            false)))))
    (match [x y]
      ([(some-term _) (some-term _)] (test-with match-terms))
      (_ (test-with match-sentences)))))
```

```
349
350 (define (greater-or-equal? x y)
351   (|| (less? y x) (equal? x y)))
352
353 (define (less-or-equal? x y)
354   (|| (less? x y) (equal? x y)))
355
356 (define (leq? x y)
357   (|| (less? x y) (equal? x y)))
358
359 (define (decompose1 P M)
360   (dmatch P
361      ((and (list-of _ rest))
362         (dlet ((L (!left-and P)))
363            (!decompose1 L (method (left-conjuncts)
364                             (dmatch rest
365                               ([] (!M (add L left-conjuncts)))
366                               (_  (dlet ((R (!right-and P)))
367                                     (!decompose1 R
368                                        (method (right-conjuncts)
369                                          (!M (join [L R] left-conjuncts right-conjuncts))))))))))
370      (_ (!M [P])))))
371
372
373 (define (decompose P M)
374   (!decompose1 P (method (props) (!M (remove-duplicates props)))))
375
376 (define (decompose* conjunctions M)
377    (dletrec ((loop (method (props results)
378                      (dmatch props
379                         ([] (!M results))
380                         ((list-of C more) (!decompose C (method (results')
381                                                           (!loop more (join results' results)))))))))
382       (!loop conjunctions [])))
383
384 (define (conj-intro goal)
385   (dmatch goal
386     ((and (some-list args)) (!map-method conj-intro args (method (_) (!and-intro args))))
387     ((some-list args) (!map-method conj-intro args (method (_) (!and-intro args))))
388     ((some-sent p) (!claim p))))
389
390 (define (conj-elim p C)
391   (dmatch C
392     ((and (some-list args)) (!decompose C (method (_) (!claim p))))
393     (_ (!claim p))))
394
395 (define (dhalt)
396   (dlet ((_ (halt)))
397     (!true-intro)))
398
399 (define (claim-test p)
400   (dtry (dlet ((mprint (lambda (x) ())))
401                (_ (!claim p))
402                (_ (print (join "\nSuccess: " (val->string p) " holds...\n"))))
403           (!claim p))
404         (dlet ((mprint (lambda (x) ())))
405                (_ (mprint (join "\nFailure: " (val->string p) " does not hold...\n"))))
406          (!dhalt))))
407
408 (define (numeric? c)
409   (member? c "0123456789"))
410
411 (define composed-with
412    (lambda (f1 f2)
413      (check ((method? f1) (method (p) (!f2 (!f1 p))))
414             (else (lambda (x) (f2 (f1 x)))))))
415
416
417 (define compose composed-with)
418
```

```
(define (lhs atom)
  (match atom
    (((some-symbol _) s _)  s)
    ((not ((some-symbol _) s _))  s)
    ((if _ ((some-symbol _) s _)) s)))

(define (rhs atom)
  (match atom
    (((some-symbol _) _ t)  t)
    ((not ((some-symbol _) _ t))  t)))

(define (left-rule-side rule)
  (match rule
    ((forall (some-list _) (= left _)) left)
    ((forall (some-list _) (if _ (= left _))) left)))

(define (right-rule-side rule)
  (match rule
    ((forall (some-list _) (= _ right)) right)))

(define (binary-proc? p)
  (&& (proc? p) (equal? (arity-of p) 2)))


(define (has-unique-ex P)
  (match P
    ((some-atom A) false)
    ((not Q) (has-unique-ex Q))
    (((some-sent-con pc) (some-list props)) (for-some props has-unique-ex))
    ((exists-unique _ _) true)
    (((some-quant q) x P') (has-unique-ex P'))))

(define (hold? props)
  (match props
    ([] true)
    ((list-of P more) (&& (holds? P) (hold? more)))))

(define prove-from vprove-from)

(define pf prove-from)

(define (integer? n)
  (match n
    (x:Int true)
    (_ false)))

(define (real? n)
  (match n
    (x:Real true)
    (_ false)))

(define (integer-numeral? n)
  (&& (numeral? n) (integer? n)))

(define (integer-numeral? n)
  (&& (numeral? n) (integer? n)))

(define (real-numeral? n)
  (&& (numeral? n) (real? n)))

(define (var? v)
  (match v
    ((some-var x) true)
    (_ false)))


(define (uspec* P terms)
  (dmatch terms
    ([] (!claim P))
    ((list-of t more-terms) (!uspec* (!uspec P t) more-terms))))
```

```
489
490
491
492 (define (vars* terms)
493   (match terms
494     ([] [])
495     ((list-of t rest) (rd (join (vars t) (vars* rest))))))
496
497 (define (uncurry P)
498   (dmatch P
499     ((if P1 (if P2 P3)) (assume (and P1 P2)
500                           (!mp (!mp P (!left-and (and P1 P2)))
501                                (!right-and (and P1 P2)))))))
502
503
504 (define (wl l)
505   (check ((null? l) (print "None")) (else (map-proc write l))))
506
507 (define (make-and props)
508   (dmatch props
509     ([P] (!claim P))
510     ((list-of P rest) (!both P (!make-and rest)))))
511
512 (define (urep P terms)
513   (match [P terms]
514     ([(forall x Q) (list-of t more)] (urep (replace-var x t Q) more))
515     (_ P)))
516
517 (define (body-of P)
518   (match P
519     (((some-quant Q) _ body) body)))
520
521
522 (define (quant-body P)
523   (match P
524     (((some-quant _) (some-list _) body) body)))
525
526 (define (quant-vars P)
527   (match P
528     (((some-quant _) (some-list vars) _) vars)))
529
530 (define (uquant-body P)
531   (match P
532     ((forall (some-list _) (some-sent body)) body)))
533
534
535
536
537 (define (get-vars-manual t)
538   (match t
539     ((some-var v) (let ((_ ())) # (_ (print "\nVar: " v)))
540                     [v]))
541     (((some-symbol f) (some-list args)) (let ((_ ())) # (_ (print "\nRoot Symbol: " f " and args: " args)))
542                                           (rd (flatten (join (map get-vars-manual args)))))))
543
544 (define (qvars-of P)
545   (match P
546     (((some-quant Q) (list-of (some-var x) rest) _) (add x rest))))
547
548 (define (get-all-vars arg)
549   (match arg
550     ((some-term t) (vars t))
551     ((some-sent P) (vars P))
552     ((some-list args) (foldr join [] (map get-all-vars args)))))
553
554 (define (get-vars arg) (remove-duplicates (get-all-vars arg)))
555
556 (define (conjoin props)
557   (match props
558     ([(some-sent P)] P)
```

```
559     ((list-of (some-sent P) rest) (and P (conjoin rest)))))
560
561 (define (disjoin props)
562   (match props
563     ([P] P)
564     ((list-of P rest) (or P (disjoin rest)))))
565
566 (define (disj-intro goal)
567   (dtry (!claim goal)
568         (dmatch goal
569           ((or (some-list args)) (!map-method-non-strictly disj-intro args (method (_) (!either args)))))))
570
571
572 (define (equality s t)
573  (dlet ((res (check ((equal? s t) (= s t)))))
574    (!force res)))
575
576 (define (equal-args? L1 L2)
577   (match [L1 L2]
578     ([[] []] true)
579     ([(list-of t rest) (list-of t' rest')] (check ((|| (holds? (= t t')) (holds? (= t' t)) (equal? t t'))
580                                                    (equal-args? rest rest'))
581                                                   (else false)))
582     (_ false)))
583
584 (define (println s)
585   (print (join "\n" s "\n")))
586
587 (define (writeln-val v)
588   (seq (print "\n")
589        (write-val v)
590        (print "\n")))
591
592 (define (writeln-val1 msg v)
593   (seq (println msg)
594        (write-val v)
595        (print "\n")))
596
597 (define (show str v)
598   (seq (println str) (write v)))
599
600 (define fv free-vars)
601
602 (define (show-read str v) (seq (print (join "\n" str "\n")) (write v) (read)))
603
604 # This is now a primitive Athena procedure:
605
606 #(define (subterms t)
607 #  (add t (fold join (map subterms (children t)) [])))
608
609 (define (proper-subterms t)
610   (tail (subterms t)))
611
612 (define (constants&vars t)
613   (match t
614     ((some-var v) [v])
615     (((some-symbol f) (some-list args)) (check ((null? args) [f])
616                                                (else (fold join (map constants&vars args) []))))
617     (((some-sent-con sc) (some-list args))
618        (fold join (map constants&vars args) []))
619     (((some-quant _) (some-var _) body) (constants&vars body))))
620
621
622 (define (from-terms P f)
623   (match P
624     ((some-atom A) (f A))
625     ((not body) (from-terms body f))
626     (((some-sent-con pc) P1 P2) (join (from-terms P1 f) (from-terms P2 f)))
627     (((some-quant q) _ body) (from-terms body f))))
628
```

```
(define (from-terms* props f)
  (fold join (map (lambda (P) (from-terms P f)) props) []))

(define (prop-constants-and-vars props)
  (from-terms* props constants&vars))

(define (prop-subterms P)
  (from-terms P subterms))

(define (prop-subterms* props)
  (remove-duplicates (from-terms* props subterms)))

(define (non-var-term t)
  (match t
    (((some-symbol f) (some-list args)) true)
    (_ false)))

(define (prop-constants-and-free-vars props)
  (let ((fvars (fold join (map free-vars props) [])))
   (remove-duplicates (filter (from-terms* props constants&vars)
                              (lambda (t) (|| (member? t fvars) (non-var-term t)))))))


(define (times* L)
  (fold times L 1))

(define (plus* L)
  (fold plus L 0))

(define (term-size t)
  (match t
    (((some-symbol f) (some-list args)) (plus* (add 1 (map term-size args))))
    (_ 1)))

(define (term-less? s t)
  (less? (term-size s) (term-size t)))

(define (no-free-vars? P)
  (null? (free-vars P)))

(define (interesting-prop-subterms P)
  (check ((no-free-vars? P) (prop-constants-and-free-vars [P]))
         (else (prop-subterms P))))

(define (interesting-prop-subterms* props)
  (remove-duplicates (fold join (map interesting-prop-subterms props) [])))



(define (do-props props M K)
  (dletrec ((loop (method (props res)
                    (dmatch props
                      ([] (!K (rev res)))
                      ((list-of (some-sent P) rest) (dlet ((th (!M P)))
                                                      (!loop rest (add th res)))))))))
     (!loop props [])))

(define (make-conjunction props)
  (dmatch props
    ([P] (!claim P))
    ((list-of P more-props) (!both P (!make-conjunction more-props)))))


(define (and* props)
  (match props
    ([] true)
    ([(some-sent p)] p)
    ((list-of p rest) (and p (and* rest)))))

(define (or* props)
```

```
699   (match props
700     ([] false)
701     ([(some-sent p)] p)
702     ((list-of p rest) (or p (or* rest)))))
703
704 (define (unify-props P1 P2)
705     (letrec ((f (lambda (P1 P2)
706                   (match [P1 P2]
707                     ([(some-atom A1) (some-atom A2)] (unify A1 A2))
708                     ([(not Q1) (not Q2)] (f Q1 Q2))
709                     ([((some-sent-con pc) Q1 Q2) (pc Q3 Q4)]
710                          (match (f Q1 Q3)
711                            ((some-sub sub1) (match (f (sub1 Q2) (sub1 Q4))
712                                                    ((some-sub sub2) (compose-subs sub2 sub1))
713                                                    (_ false)))
714                            (_ false)))
715                     ([((some-quant Q) x B1) (Q y B2)]
716                          (let ((v (fresh-var)))
717                            (f (replace-var x v B1) (replace-var y v B2))))
718                     (_ false)))))
719        (f (rename P1) P2)))
720
721 (define (unify-props-with-var-constants P1 P2 var-constants)
722     (letrec ((f (lambda (P1 P2)
723                   (match [P1 P2]
724                     ([(some-atom A1) (some-atom A2)] (unify A1 A2 var-constants))
725                     ([(not Q1) (not Q2)] (f Q1 Q2))
726                     ([((some-sent-con pc) Q1 Q2) (pc Q3 Q4)]
727                          (match (f Q1 Q3)
728                            ((some-sub sub1) (match (f (sub1 Q2) (sub1 Q4))
729                                                    ((some-sub sub2) (compose-subs sub2 sub1))
730                                                    (_ false)))
731                            (_ false)))
732                     ([((some-quant Q) x B1) (Q y B2)]
733                          (let ((v (fresh-var)))
734                            (f (replace-var x v B1) (replace-var y v B2))))
735                     (_ false)))))
736        (f (rename P1) P2)))
737
738 (define up unify-props)
739
740
741 (define (unify* v1 v2)
742   (match [v1 v2]
743     ([(some-term t1) (some-term t2)] (unify t1 t2))
744     ([(some-sent P1) (some-sent P2)] (unify-props P1 P2))))
745
746 (define unified-with unify)
747 (define rvp replace-var)
748 (define rvt replace-var)
749
750 (define (replace-vars vars terms p)
751   (match [vars terms]
752     ([[] _] p)
753     ([(list-of v more-vars) (list-of t more-terms)]
754       (replace-vars more-vars more-terms (replace-var v t p)))))
755
756 (define (get-conjuncts-recursive p)
757   (match p
758     ((and (some-list props)) (fold join (map get-conjuncts-recursive props) []))
759     ((some-sent _) [p])))
760
761 (define get-conjuncts get-conjuncts-recursive)
762
763 (define (get-all-conjuncts p)
764   (match p
765     ((and (some-list props)) (add p (fold join (map get-all-conjuncts props) [])))
766     ((some-sent _) [p])))
767
768 (define (get-disjuncts-recursive p)
```

```
769   (match p
770     ((or (some-list props)) (fold join (map get-disjuncts-recursive props) []))
771     ((some-sent _) [p])))
772
773 (define get-disjuncts get-disjuncts-recursive)
774
775 (define (get-disjuncts* props)
776   (letrec ((loop (lambda (props res)
777                    (match props
778                       ([] (rd res))
779                       ((list-of (some-sent p) more) (loop more (join (get-disjuncts p) res)))))))
780     (loop props [])))
781
782
783 (define (consistent-ab? time-limit)
784   (let ((c (cell [])))
785     (match (dtry (!vprove-from false (ab) [c ['poly true] ['subsorting false] ['max-time time-limit]]) (!true-intro))
786       (false [false (ref c)])
787       (_ true))))
788
789
790 (define (byCases P1 P2 hyps)
791   (dmatch [P1 P2]
792     ([(if P Q) (if P' Q)]
793        (dlet ((one-of-two (!spf (or P P') hyps)))
794          (!cases one-of-two P1 P2)))))
795
796 ###############-----------------------------------------------------------
797
798 (define (by-contradiction conclusion p)
799   (dmatch [conclusion p]
800     ([q (if (not q) false)] (dseq (suppose-absurd (not q)
801                                     (!mp p (not q)))
802                                   (!dn (not (not q)))))
803     ([(not q) (if q false)] (suppose-absurd q
804                               (!mp p q)))))
805
806 (define by-contradiction' by-contradiction)
807
808 (define (contradiction p)
809   (dmatch p
810     ((if (not q) false) (dseq (suppose-absurd (not q)
811                                     (!mp p (not q)))
812                                   (!dn (not (not q)))))
813     ((if q false) (suppose-absurd q (!mp p q)))))
814
815
816 (define (from-false goal)
817   (!by-contradiction goal
818     (assume (not goal)
819       (!claim false))))
820
821 (define (from-complements p q1 q2)
822   (dlet ((M (method (goal q not-q)
823               (dseq (!absurd q not-q)
824                     (!from-false goal)))))
825     (dmatch [q1 q2]
826       ([q (not q)] (!M p q1 q2))
827       ([(not q) q] (!M p q2 q1)))))
828
829
830 (define (false-elim)
831   (!by-contradiction (~ false)
832    assume false
833      (!claim false)))
834
835 (define (contradiction-from H)
836   (method (_ q)
837     (!by-contradiction q
838        (assume (complement q)
```

```
          (!from-complements false H (complement H))))))

(define inconsistent-with contradiction-from)

(define (idn premise)
  (!by-contradiction (not (not premise))
      assume -premise := (not premise)
         (!absurd premise -premise)))

(define (bdn premise)
  (dmatch premise
    ((not (not _)) (!dn premise))
    (_ (!idn premise))))

(define (and-comm premise)
  (dmatch premise
    ((and p q) (!both (!right-and premise)
                      (!left-and premise)))))

(define (or-comm premise)
  (dmatch premise
    ((or p q) (!cases (or p q)
                      (assume p
                        (!right-either q p))
                      (assume q
                        (!left-either q p))))))

(define (iff-comm premise)
   (dmatch premise
     ((iff p q) (!equiv (!right-iff premise) (!left-iff premise)))))


(define (unequal-sym premise)
  (dmatch premise
    ((not (= (some-term s) (some-term t)))
      (!by-contradiction (not (= t s))
        (assume hyp := (t = s)
           (!absurd (!sym hyp) premise))))))

(define ineq-sym unequal-sym)

(define (comm premise)
  (dmatch premise
    ((and _ _) (!and-comm premise))
    ((= _ _) (!sym premise))
    ((not (= _ _)) (!unequal-sym premise))
    ((or _ _)  (!or-comm premise))
    ((iff _ _) (!iff-comm premise))))

(define (contra-pos premise)
  (dmatch premise
    ((if p q) (assume (complement q)
                (!by-contradiction (complement p)
                   (assume p
                     (dseq (conclude q
                             (!mp premise p))
                           (!from-complements false q (complement q))))))))))

(define (dsyl-1 premise-1 premise-2)
  (dmatch [premise-1 premise-2]
    ([(or p q) p']
        (dcheck ((complements? p p')
                   (!cases premise-1
                          (assume p
                            (!from-complements q p p'))
                          (assume q
                            (!claim q))))))))

(define (dsyl premise-1 premise-2)
  (dmatch [premise-1 premise-2]
```

```
909     ([(or p q) r] (dcheck ((complements? p r) (!dsyl-1 premise-1 premise-2))
910                           (else (!dsyl-1 (!comm premise-1) premise-2))))))
911
912
913 (define (neg-cond premise)
914   (dmatch premise
915     ((not (if (some-sent p) (some-sent q)) )
916       (!both (!by-contradiction p
917                  (assume (not p)
918                    (!absurd (assume p
919                                (!from-complements q p (not p)))
920                             premise)))
921              (!by-contradiction (not q)
922                 (assume q
923                   (!absurd (assume p (!claim q))
924                            premise)))))))
925
926 (define (neg-cond-ant premise)
927   (dmatch premise
928     ((not (if p q)) (!by-contradiction p
929                       (assume (not p)
930                         (dseq (assume p
931                                 (!from-complements q p (not p)))
932                               (!absurd (if p q) premise)))))))
933
934 (define (neg-cond-con premise)
935   (dmatch premise
936     ((not (if p q)) (!by-contradiction (not q)
937                       (assume q
938                         (dseq (assume p (!claim q))
939                               (!absurd (if p q) premise)))))))
940
941
942 (define (neg-cond-conv premise)
943   (dmatch premise
944     ((and p (not q)) (!by-contradiction (not (if p q))
945                        (assume-let (hyp (if p q))
946
947                          (dseq (conclude q
948                                  (!mp hyp (!left-and premise)))
949                                (!absurd q (!right-and premise))))))))
950
951 (define (neg-cond premise)
952   (dmatch premise
953     ((not (if _ _)) (!both (!neg-cond-ant premise)
954                            (!neg-cond-con premise)))
955     (_ (!neg-cond-conv premise))))
956
957 (define neg-cond1 neg-cond-ant)
958 (define neg-cond2 neg-cond-con)
959
960
961 (define (ex-middle-1 p)
962   (dlet ((goal (or p (not p))))
963     (!by-contradiction goal
964        (assume (not goal)
965          (dseq (!by-contradiction p
966                   (assume (not p)
967                     (!absurd (!either p (not p))
968                              (not goal))))
969                (!by-contradiction (not p)
970                   (assume p
971                     (!absurd (!either p (not p))
972                              (not goal))))
973                (!absurd p (not p)))))))
974
975 (define (ex-middle p)
976   (dmatch p
977     ((or (some-sent q) (not q)) (!ex-middle-1 q))
978     (_ (!ex-middle-1 p))))
```

```
(define excluded-middle ex-middle)


(define (two-cases cond-1 cond-2)
  let {M := method (p q)
              (!cases conclude (p | ~ p)
                       (!ex-middle p)
               (p ==> q)
               (~ p ==> q))}
  match [cond-1 cond-2] {
    [(if p q) (if (not p) q)] => (!M p q)
  | [(if (not p) q) (if p q)] => (!M p q)
  })

(define (normalize-disjunction disj norm-disj normalized-disjuncts)
   (!map-method (method (d)
                   (assume d (!either norm-disj)))
                normalized-disjuncts
                (method (L)
                  (!cases disj L))))

###############-----------------------------------------------------------------
#
#               NEW DEFINITION OF DE MORGAN AND COND-DEF
#
###############-----------------------------------------------------------------

(define (make-dm-3 complement)
  (method (premise)
    (dlet ((c complement))
      (dmatch premise
        ((not (disjunction as (or (some-list args))))
           (!map-method (method (p)
                          (!by-contradiction (c p)
                             (assume p
                                (!absurd (!either disjunction) premise))))
                        args
                        (method (complemented-args)
                          (!conj-intro (and complemented-args)))))))))

(define dm-3 (make-dm-3 not))
(define dm-3' (make-dm-3 complement))

(define (make-dm-1 c)
 (method (premise)
   (dmatch premise
     ((not (premise-body as (and (some-list conjuncts))))
        (dlet ((goal (or (map c conjuncts))))
            (!by-contradiction goal
               (assume -goal := (not goal)
                 (dlet ((p1 (!(make-dm-3 not) -goal))
                        (dn (method (p) (dtry (!dn p) (!claim p)))))
                    (!map-method dn
                                 (get-conjuncts p1)
                                 (method (_)
                                    (!absurd (!conj-intro premise-body) premise)))))))))))

(define dm-1 (make-dm-1 not))
(define dm-1' (make-dm-1 complement))

(define (make-dm-2 c)
 (method (premise)
   (dmatch premise
     ((or (some-list args))
        (dlet ((goal (not (and (map c args)))))
          (!by-contradiction goal
             (assume -goal := (not goal)
                (dlet ((_ (!dn -goal))) ## Now have (and (map c args))
                  (!map-method (method (p_i)
```

```
                                    (assume p_i
                                       (!from-complements false p_i (c p_i))))
                                  args
                                  (method (conditionals)
                                    (!cases premise conditionals))))))))))))


(define dm-2 (make-dm-2 not))
(define dm-2' (make-dm-2 complement))

(define (make-dm-4 c)
  (method (premise)
    (dmatch premise
      ((and (some-list args))
         (dlet ((-args  (map c args))
                (goal (not (or -args))))
           (!by-contradiction goal
              (assume -goal := (not goal)
                (dlet ((disjunction (!dn -goal)))
                  (!decompose premise
                     (method (_)
                       (!map-method (method (-p_i)
                                      (assume -p_i
                                        (dmatch -p_i
                                           ((not (some-sent q))
                                              (dtry (!absurd q -p_i)
                                                    (!absurd -p_i (not -p_i))))
                                           (_ (!absurd -p_i (not -p_i))))))
                                    -args
                                    (method (conditionals)
                                       (!cases disjunction
                                               conditionals))))))))))))


(define (make-dm c)
  (method (premise)
    (dmatch premise
      ((not (and (some-list _))) (!(make-dm-1 c) premise))
      ((not (or (some-list _))) (!(make-dm-3 c) premise))
      ((and (some-list _)) (!(make-dm-4 c) premise))
      ((or (some-list _)) (!(make-dm-2 c) premise)))))


(define dm (make-dm complement))
(define dm' (make-dm not))


(define (dm-rec p)
  (dmatch p
    ((and (some-list _)) (!dm' p))
    ((or (some-list _)) (!dm' p))
    ((not (and (some-list _))) (!dm' p))
    ((not (or (some-list _))) (dmatch (!dm' p)
                                ((res as (and (some-list args)))
                                    (!decompose res (method (_) (!map-method dm-rec args (method (results) (!conj-int
    (_ (!claim p))))


(define (dm2c premise target)
   (dmatch [premise target]
     ([(or (some-list disjuncts)) (not (big-c as (and (some-list conjuncts))))]
        (dlet ((d-c (list-zip disjuncts conjuncts)))
          (!by-contradiction target
             (assume big-c
                  (!map-method (method (pair)
                                 (dmatch pair
                                    ([d c]
                                       (assume d
```

```
                                          (!from-complements false c d)))))
                                d-c
                                (method (conditionals)
                                  (!cases premise conditionals))))))))


(define (dm3c premise target)
      (dmatch [premise target]
        ([(not (disjunction as (or (some-list disjuncts)))) (and (some-list conjuncts))]
           (dlet ((d-c (list-zip disjuncts conjuncts)))
             (!map-method (method (pair)
                            (dmatch pair
                              ([(some-sent p) (some-sent q)]
                                 (!by-contradiction q
                                    (assume p
                                      (!absurd (!either disjunction) premise))))))
                          d-c
                          (method (complemented-args)
                            (!conj-intro (and complemented-args))))))))


(define (dm1c premise target)
   (dmatch [premise target]
     ([(not (premise-body as (and (some-list conjuncts)))) (or (some-list disjuncts))]
        (dlet ((c-d  (list-zip conjuncts disjuncts)))
            (!by-contradiction target
               (assume -target := (not target)
                  (!absurd (!dm3c -target premise-body)
                           premise)))))))

(define (dm4c premise target)
  (dmatch [premise target]
    ([(and (some-list conjuncts)) (not (disjunction as (or (some-list disjuncts))))]
      (!by-contradiction target
        (assume disjunction
          (!map-method (method (pair)
                         (dmatch pair
                           ([c d] (assume d
                                    (!from-complements false c d)))))
                       (list-zip conjuncts disjuncts)
                       (method (conditionals)
                         (!cases disjunction conditionals)))))))

(define (dm-binary premise target)
  (dmatch premise
    ((not (and (some-list _))) (!dm1c premise target))
    ((not (or (some-list _))) (!dm3c premise target))
    ((and (some-list _)) (!dm4c premise target))
    ((or (some-list _)) (!dm2c premise target))))

define dm-2 := dm-binary
define (make-cond-def-1 neg) :=
  method (premise)
    match premise {
      (p ==> q) => let {p' := (neg p);
                        goal := (p' | q)}
                     (!by-contradiction goal
                        assume -goal := (~ goal)
                          let {_ := (!dm' -goal);
                               p := try { (!dn (~ ~ p)) | (!claim p) }}
                            (!absurd (!mp premise p)
                                     (~ q)))
     }


define (make-cond-def-2 neg) :=
  method (premise)
    match premise {
```

```
        (p | q) => assume p' := (neg p)
                     (!cases premise
                             assume p
                               (!from-complements q p p')
                             assume q
                               (!claim q))
      }

define (make-cond-def neg) :=
  method (premise)
     match premise {
       (_ ==> _) => (!(make-cond-def-1 neg) premise)
     | (_ | _)   => (!(make-cond-def-2 neg) premise)
     }


define cond-def := (make-cond-def complement)
define cond-def' := (make-cond-def not)

## Binary version of cond-def

define cond-def-2 :=
  method (premise goal)
    match [premise goal] {
      [(p ==> q) (p' | q)] =>
         (!two-cases
            (assume p
              (!right-either p' (!mp premise p)))
            assume h := (not p)
              let {_ := match p {
                          (not _) => (!dn h)
                        | _ => (!claim h)}}
                (!left-either p' q))
    | [(p | q) (p' ==> q)] =>
         (!cases premise
                 assume p
                   assume p'
                     (!from-complements q p p')
                 assume q
                   assume p'
                     (!claim q))
    }
###############-----------------------------------------------------------------
##
##                       END OF NEW DM DEFINITION
##
###############-----------------------------------------------------------------

#=================================================================
# In the following implementation of dsyl, the first argument is an
# arbitrarily structured disjunction with (get-disjuncts) [d1,...,dn]
# and the second argument is a list of sentences N = [d_i_1',...d_i_k']
# where each d_i_j' is complementary to some unique d_x.
# E.g., the first argument might be (or A (not B) C (not D)) and
# N could be [(not A) (not (not B))]. All elements of N must be in
# the assumption base, as must be the first argument (the disjunction).
#
# Let D be the list of those disjuncts in [d1,...,dn] that
# have no complementary entry in N (in left-to-right order as
# they appear in [d1 ... dn]). The method derives the conclusion
# (or D), or else false if D is empty. If D has only one element p,
# then p -- rather than (or p) -- is returned.
#
# For added flexibility, the second argument could be a single
# conjunction of N = [d_i_1',...d_i_k'], rather than a list.
# Or, in case n = 2 (which is the "traditional" formulation of disjunctive
# syllogism, whereby the first premise is (p1 | p2) and the second
# premise is either p1' or p2'), the second argument could also
# be a single sentence that is complementary to one of the two
# disjuncts. This accommodates the traditional (and simplest)
```

```
# way in which one would call this method.
#=====================================================================

(define (derive-single-disjunct premise)
  (dmatch premise
     ((or [(some-sentence q)])
          (!cases premise
                  (assume q (!claim q))))
     (_ (!claim premise))))

(define (dsyl big-disjunction N)
 (!decompose N # Works even if N is a list. Will do real work
                          # only when N is a conjunction.
   (method (_)
      (dlet ((disjuncts (get-disjuncts big-disjunction))
             (negation-list (check ((list? N) N)
                                   ((sentence? N)
                                       (check ((for-some disjuncts
                                                         (lambda (d) (complements? d N)))
                                                 [N])
                                              (else (get-conjuncts N))))))
             (remaining-disjuncts
               (filter-out disjuncts
                           (lambda (p)
                              (for-some negation-list
                                        (lambda (q) (complements? p q))))))
             (goal (check ((null? remaining-disjuncts) false) (else (or remaining-disjuncts))))
             (_ (!by-contradiction goal
                   (assume -goal := (not goal)
                     (!decompose (dtry (!dm' -goal) (!claim -goal))
                                 (method (_)
                                     (!map-method (method (d)
                                                    (assume d
                                                      (dtry (!from-complements false d (complement d))
                                                            (!from-complements false d (not d)))))
                                                  disjuncts
                                                    (method (conditionals)
                                                        (!cases big-disjunction
                                                                conditionals)))))))))
          (dmatch goal
             ((or [_]) (!derive-single-disjunct goal))
             (_ (!claim goal)))))))


(define (dmark c)
  (dlet ((_ (mark c)))
    (!true-intro)))

(define (horn-clause-antecedent p)
  (match p
    ((forall (some-list _) (if antecedent _)) antecedent)))

(define (antecedent P)
  (match P
    ((if (some-sent p1) _)  p1)
    ((iff (some-sent p1) _) p1)))

(define get-antecedent antecedent)


(define (consequent P)
  (match P
    ((if _ P2)  P2)
    ((iff _ P2) P2)))

(define dt-comp-method-cell (cell ()))

(primitive-method (numeric-comparison p)
  (match p
    (((< x y) where (less? x y)) p)
```

```
    (((> x y) where (less? y x)) p)
    (((<= x y) where (leq? x y)) p)
    (((>= x y) where (leq? y x)) p)))

(define (prove-components-of P)
  (dcheck ((holds? P) (!claim P))
          (else (dmatch P
                  (true (!true-intro))
                  ((and P1 P2) (!both (!prove-components-of P1)
                                      (!prove-components-of P2)))
                  ((and (some-list args)) (!do-props args prove-components-of (method (theorems) (!and-intro theorems)
                  ((or P1 P2)  (dtry (!either (!prove-components-of P1) P2)
                                     (!either P1 (!prove-components-of P2))))
                  ((or (some-list args)) (!map-method-non-strictly prove-components-of args (method (_) (!either args)
                  ((some-atom A) (dmatch A
                                   ((= (some-term t) t) (!reflex t))
                                   (_ (!numeric-comparison A))))
                  ((= (some-term s) (some-term t)) (dtry (!(ref dt-comp-method-cell) P)
                                                         (!sym (= t s))))
                  ((not (= (some-term s) (some-term t))) (dtry (!(ref dt-comp-method-cell) P)
                                                               (!by-contradiction P
                                                                  (assume hyp := (s = t)
                                                                    (!absurd (!sym hyp) (not (= t s)))))))
                  ((not (some-term s)) (!(ref dt-comp-method-cell) P))
                  ((not (and (some-list args)))
                     (!map-method-non-strictly (method (arg)
                                                 (!prove-components-of (complement arg)))
                                               args
                                               (method (results)
                                                 (!dm (!either (map complement args))))))
                  ((not (or (some-list args))) (!map-method (method (arg)
                                                              (!prove-components-of (complement arg)))
                                                            args
                                                            (method (results)
                                                              (!dm (!conj-intro (and results)))))))))))

(define prove-components-harder prove-components-of)

(define (all-components-hold P)
  (try (match (!prove-components-of P)
         ((val-of P) true)
         (_ false))
       false))

(define (prove-antecedent P)
  (!prove-components-of (antecedent P)))

(define (fire premise terms)
  (dmatch premise
    ((forall (some-list vars) (if P _)) (dlet ((th (!uspec* premise terms)))
                                          (!mp th (!prove-antecedent th))))
    ((forall (some-list vars) (iff _ _))
       (dlet ((th (!uspec* premise terms)))
         (dtry (dlet ((P (!left-iff th)))
                 (!mp P (!prove-antecedent P)))
               (dlet ((P (!right-iff th)))
                 (!mp P (!prove-antecedent P))))))
    ((forall (some-list vars) _) (!uspec* premise terms))))

(define (fire-aux instance)
  (dmatch instance
    ((if _ _) (!mp instance (!prove-antecedent instance)))
    ((iff _ _) (dtry (dlet ((p (!left-iff instance)))
                       (!mp p (!prove-antecedent p)))
                     (dlet ((p (!right-iff instance)))
                       (!mp p (!prove-antecedent p)))))
    (_ (!claim instance))))

# (define (instance p terms)
#   (dmatch terms
```

```
#     ((some-list _) (!fire p terms))
#     ((some-term t) (!fire p [t])))))

(define (instance p terms)
  (dletrec ((named? (lambda (L)
                      (match L
                        ((list-of [_ --> _] _) true)
                        (_ false))))
            (make-vmap (lambda (L)
                         (match L
                           ([] (lambda (_) ()))
                           ((list-of [v --> t] more)
                             (lambda (v') (check ((equal? v v') t)
                                                 (else ((make-vmap more) v')))))))))
           (terms' (match terms
                     ((some-list _) terms)
                     ((some-term t) [t])))))
    (dmatch p
      ((forall (some-list uvars) _)
        (dcheck ((named? terms') (dlet ((vmap (make-vmap terms')))
                                       (!uspec* p (map vmap (take uvars (length terms'))))))
                (else (!uspec* p terms')))))))

(define (fire p terms)
  (dletrec ((named? (lambda (L)
                      (match L
                        ((list-of [_ --> _] _) true)
                        (_ false))))
            (make-vmap (lambda (L)
                         (match L
                           ([] (lambda (_) ()))
                           ((list-of [v --> t] more)
                             (lambda (v') (check ((equal? v v') t)
                                                 (else ((make-vmap more) v')))))))))
           (terms' (match terms
                     ((some-list _) terms)
                     ((some-term t) [t])))))
    (dmatch p
      ((forall (some-list uvars) _)
        (dcheck ((named? terms') (dlet ((vmap (make-vmap terms'))
                                        (instance (!uspec* p (map vmap (take uvars (length terms'))))))
                                       (!fire-aux instance)))
                (else (!fire-aux (!uspec* p terms')))))))

## (define (gp s) (match (and s s) ((and p p) p)))

## Proves x = y from the assumption [x] in {[y]}


(define (all-props) (fetch-all (lambda (P) true)))

(define (not-equal x y)
  (negate (equal? x y)))

(define get-assumption-base ab)

(define (reflexive R)
  (forall ?x (R ?x ?x)))

(define (symmetric R)
  (forall ?x ?y (if (R ?x ?y) (R ?y ?x))))

(define (anti-symmetric R)
  (forall ?x ?y
    (if (and (R ?x ?y) (R ?y ?x))
        (= ?x ?y))))

(define (asymmetric R)
  (forall ?x ?y
    (if (R ?x ?y)
```

```
        (not (R ?y ?x)))))

(define (irreflexive R)
  (forall ?x (not (R ?x ?x))))

(define (transitive R)
  (forall ?x ?y ?z
    (if (and (R ?x ?y) (R ?y ?z))
        (R ?x ?z))))

(define (connected R)
  (forall ?x ?y
    (or (= ?x ?y)
        (R ?x ?y)
        (R ?y ?x))))

(define (unique-condition P)
  (let ((x (fresh-var))
        (y (fresh-var)))
    (forall x y
      (if (and (P x) (P y))
        (= x y)))))

(define (show-unique-existence P)
  (dmatch P
    ((exists (some-var x) (some-sent Q))
       (pick-witness w P
         (dlet ((v (fresh-var)))
           (!egen-unique (exists-unique v (replace-var x v Q)) w))))))

(define (pick-two-witnesses P M)
  (pick-witness w1 P Q1
    (pick-witness w2 Q1 Q2
          (!M w1 w2 [Q1 Q2]))))

(define (pick-three-witnesses P M)
  (pick-witness w1 P Q1
    (pick-witness w2 Q1 Q2
      (pick-witness w3 Q2 Q3
          (!M w1 w2 w3 [Q1 Q2 Q3])))))

(define (pick-four-witnesses P M)
  (pick-witness w1 P Q1
    (pick-witness w2 Q1 Q2
      (pick-witness w3 Q2 Q3
        (pick-witness w4 Q3 Q4
          (!M w1 w2 w3 w4 [Q1 Q2 Q3 Q4]))))))

(define (pick-five-witnesses P M)
  (pick-witness w1 P Q1
    (pick-witness w2 Q1 Q2
      (pick-witness w3 Q2 Q3
        (pick-witness w4 Q3 Q4
          (pick-witness w5 Q4 Q5
            (!M w1 w2 w3 w4 w5 [Q1 Q2 Q3 Q4 Q5])))))))


(define (unequal? a b)
  (check ((equal? a b) false)
         (else true)))

(define (get-term-syms t)
  (match t
    ((some-var x) [])
    (_ (add (root t) (foldr join [] (map get-term-syms (children t)))))))

(define (get-term-syms* terms)
  (fold join (map get-term-syms terms) []))

(define (get-prop-syms P)
```

```
  (match P
    ((some-atom A) (get-term-syms A))
    ((not Q) (get-prop-syms Q))
    (((some-sent-con pc) (some-list props)) (foldr join [] (map get-prop-syms props)))
    (((some-quant q) x P') (get-prop-syms P'))))

(define (get-prop-syms* props)
  (rd (flatten (map get-prop-syms props))))

(define (new-term-syms t syms)
                            (match t
                              ((some-var x) [])
                              (((some-symbol f) (some-list args))
                                  (check ((member? f syms) (new-term-syms* args syms))
                                         (else (add f (new-term-syms* args (add f syms))))))))
           (new-term-syms* terms syms)
                             (match terms
                               ([] [])
                               ((list-of t rest) (let ((syms' (new-term-syms t syms)))
                                                   (join syms' (new-term-syms* rest (join syms' syms))))))
           (new-prop-syms P syms)
                             (match P
                               ((some-atom A) (new-term-syms A syms))
                               ((not Q) (new-prop-syms Q syms))
                               (((some-sent-con pc) (some-list props)) (new-prop-syms* props syms))
                               (((some-quant q) x P') (new-prop-syms P' syms)))
           (new-prop-syms* props syms)
                             (match props
                               ([] [])
                               ((list-of P rest) (let ((syms' (new-prop-syms P syms)))
                                                   (join syms' (new-prop-syms* rest (join syms' syms))))))
           (gs s syms)
                          (match (get-sym-def s)
                             ((some-sent P) [P (new-prop-syms P syms)])
                             (_ ())))

(define (get-defs symbols)
  (letrec ((loop (lambda (syms defs new-syms seen)
                   (match syms
                     ([] defs)
                     ((list-of s rest) (check ((member? s seen) (loop rest defs (add s new-syms) (add s seen)))
                                              (else (match (gs s (join new-syms symbols))
                                                      ([P syms'] (loop (join rest syms') (add P defs)
                                                                       (add s (join syms' new-syms)) (add s seen)))
                                                      (_ (loop rest defs (add s new-syms) (add s seen))))))))))))
    (loop symbols [] [] [])))

(define (get-prop-sym-defs P)
   (get-defs (get-prop-syms P)))

(define gpsd get-prop-sym-defs)

(define (gpsd* P)  (get-prop-sym-defs P))

(define (test P)
    (map write (get-prop-sym-defs P)))

(define (gsd props syms res)
  (let ((syms' (foldr join [] (map get-prop-syms props)))
        (new-syms (list-diff syms' syms))
        (foo (write new-syms)))
    (match new-syms
      ([] res)
      (_  (let ((new-sym-defs (map get-sym-def new-syms)))
            (gsd new-sym-defs (join new-syms syms) (join new-sym-defs res)))))))


(define (sym-defs P)
  (gsd [P] [] []))
```

```
1609
1610 (define (find-witness P facts M)
1611   (dlet ((P1 (!vpf P facts)))
1612     (pick-witness x P P1-inst
1613       (!M x P1-inst))))
1614
1615 ## ZF VERSION:
1616 #(define (find-witness P facts M)
1617 #  (dlet ((P1 (!vpf* P facts)))
1618 #    (pick-witness x P P1-inst
1619 #      (!M x P1-inst))))
1620
1621 (define (fw P facts M)
1622   (dlet ((P1 (!vpf P facts)))
1623     (pick-witness x P P1-inst
1624       (!M x P1-inst))))
1625
1626 (define (find-witness-2 f facts M)
1627   (dlet ((v1 (fresh-var)) (v2 (fresh-var))
1628          (statements (f v1 v2))
1629          (P1 (exists v1 (first statements)))
1630          (P2 (exists v2 (second statements))))
1631    (!find-witness P1 facts
1632       (method (w1 P1-inst)
1633         (dlet ((P2' (exists v2 (second (f w1 v2)))))
1634           (!find-witness P2' (add P1-inst facts)
1635              (method (w2 P2-inst)
1636                (!M w1 w2 P1-inst P2-inst))))))))
1637
1638 (define (spf* P facts)
1639  (dlet ((Q (conjoin (add P facts))))
1640    (!spf P (join facts (gpsd* Q)))))
1641
1642
1643 (define (vpf* P facts)
1644  (dlet ((Q (conjoin (add P facts))))
1645    (!vpf P (join facts (gpsd* Q)))))
1646
1647
1648 (define fm get-model)
1649
1650 (define (univ-sort-axioms syms)
1651   (letrec ((f (lambda (syms x)
1652                 (match syms
1653                   ([P] (P x))
1654                   ((list-of P rest) (or (P x) (f rest x))))))
1655            (g (lambda (fsyms bsyms res x)
1656                 (match fsyms
1657                   ([] res)
1658                   ((list-of P rest) (let ((new-prop (if (P x)
1659                                                         (conjoin (map (lambda (Q) (not (Q x)))
1660                                                                       (join rest bsyms))))))
1661                                       (g rest (add P bsyms) (add new-prop res) x)))))))
1662     (let ((x (fresh-var)) (y (fresh-var))
1663           (gres (forall x (conjoin (g syms [] [] x))))
1664           (fres (forall y (f syms y))))
1665       [fres gres])))
1666
1667 (define (univ-sort-axioms syms)
1668   (letrec ((f (lambda (syms x)
1669                 (match syms
1670                   ([P] (P x))
1671                   ((list-of P rest) (or (P x) (f rest x))))))
1672            (g (lambda (fsyms bsyms res x)
1673                 (match fsyms
1674                   ([] res)
1675                   ((list-of P rest) (let ((new-prop (if (P x)
1676                                                         (conjoin (map (lambda (Q) (not (Q x)))
1677                                                                       (join rest bsyms))))))
1678                                       (g rest (add P bsyms) (add new-prop res) x)))))))
```

```
1679      (let ((x (fresh-var)) (y (fresh-var))
1680            (fres (forall y (f syms y))))
1681        (match syms
1682          ((list-of _ (list-of _ _)) [fres (forall x (conjoin (g syms [] [] x)))])
1683          (_ [fres])))))
1684
1685  (define (fresh-vars n)
1686    (check ((less? n 1) [])
1687           (else (add (fresh-var) (fresh-vars (minus n 1))))))
1688
1689  (define (sortify-prop P smap)
1690    (match P
1691      ((some-atom A) P)
1692      ((not Q) (not (sortify-prop Q smap)))
1693      (((some-sent-con pc) (some-list props)) (pc (map (lambda (P) (sortify-prop P smap)) props)))
1694      ((forall x Q) (match (smap (sort-of-var-in-prop x Q))
1695                      ((some-symbol D) (forall x (if (D x) (sortify-prop Q smap))))))
1696      ((exists x Q) (match  (smap (sort-of-var-in-prop x Q))
1697                      ((some-symbol D) (exists x (and (D x) (sortify-prop Q smap)))))))
1698
1699
1700
1701  (define (exclusive-constructors sname)
1702   (letrec  ((clist (constructors-of sname))
1703             (f  (lambda (cl axioms)
1704                   (match cl
1705                     ([] axioms)
1706                     ((list-of (some-symbol c)
1707                               (some-list more))
1708                             (let ((PL (map (lambda (c') (let ((Q (excl-constructors c c')))
1709                                                           Q)) more)))
1710                               (f more (join PL axioms)))))))))
1711      (f clist [])))
1712
1713
1714  (define (datatype-axioms dname)
1715    (let ((dname (qualify-sort-name dname))
1716          (clist (constructors-of dname)))
1717      (join (exclusive-constructors dname)
1718            (map (lambda (c) (let ((P (!constructor-injectivity c))) P))
1719                 (filter clist (lambda (c) (less? 0 (arity-of c)))))
1720            [(!constructor-exhaustiveness dname)])))
1721
1722
1723  (define (structure-axioms sname)
1724      (join (exclusive-constructors sname) [(!constructor-exhaustiveness sname)]))
1725
1726
1727
1728  (define (show-props L)
1729    (seq (print "\n")
1730           (map write L)
1731           (print "\n")))
1732
1733  (define (show-ab)
1734    (let ((n (length (ab))))
1735      (seq (match n
1736               (0 (print "\nThe assumption base is currently empty.\n"))
1737               (1 (print "\nThere is one sentence in the assumption base:\n\n"))
1738               (_ (seq (print (join "\nThere are " (val->string n)))
1739                       (print " sentences in the assumption base:\n\n"))))
1740           (map (lambda (P) (seq (write-val P) (print "\n\n"))) (ab))
1741           ())))
1742
1743  (define show-assumption-base show-ab)
1744
1745  (define (write-prop P) (write P))
1746
1747  (define (show P props)
1748    (dmatch props
```

```
    ([] (!vpf P (ab)))
    (_  (!vpf P props))))

(define (em-cases cond goal props)
  (!byCases (assume cond
              (!vpf goal (add cond props)))
            (assume (not cond)
              (!vpf goal (add (not cond) props))) []))


(define (case-analysis cond1 cond2 goal props cprops)
  (!byCases (assume cond1
              (!vpf goal (add cond1 props)))
            (assume cond2
              (!vpf goal (add cond2 props))) cprops))

(define (force1 P props)
  (!force P))

(define force-from force1)

(define (prove P)
  (match (dtry (!sprove-from P (ab) [['poly true] ['subsorting false] ['max-time 5000]])
               #(!spf P (ab))
               (!force (not P)))
    ((not P) (seq (print "\nUnable to derive the sentence\n")
                  (write-val P)
                  (print "\nfrom the current assumption base.\n")))
    (_ (seq (print "\nSuccess! The sentence\n")
            (write-val P)
            (print "\nfollows from the current assumption base.\n")))))


(primitive-method (ds P1 P2)
  (match [P1 P2]
    ([(or (some-sent P) (some-sent Q)) (not P)] (check ((hold? [P1 P2]) Q)
                                                       (else (error "Error: Incorrect application of ds."))))
    ([(or (some-sent P) (some-sent Q)) (not Q)] (check ((hold? [P1 P2]) P)
                                                       (else (error "Error: Incorrect application of ds."))))
    (_ (error "Error: Incorrect application of ds."))))


(define (ds P1 P2)
  (dlet ((res (match [P1 P2]
                ([(or (some-sent P) (some-sent Q)) (not P)] (check ((hold? [P1 P2]) Q)
                                                                   (else (error "Error: Incorrect application of ds."
                ([(or (some-sent P) (some-sent Q)) (not Q)] (check ((hold? [P1 P2]) P)
                                                                   (else (error "Error: Incorrect application of ds."
                (_ (error "Error: Incorrect application of ds."))))))
     (!claim res)))


(define (dni P)
  (dlet ((res (check ((hold? [P]) (not not P))
                     (else (error "Error: Incorrect application of dni."))))))
    (!force res)))


(primitive-method (false-intro)
  (not (false)))

(define (false-intro)
  (!force (not (false))))

(define (existentialize P)
    (exists* (free-vars P) P))


(define (byCases* disj ML)
  (dmatch [disj ML]
```

```
    ([(or P (bind rest1 (or _ _))) (list-of (some-method M) rest2)]
        (!byCases (assume P (!M P))

                  (assume rest1
                    (!byCases* rest1 rest2))
                  [disj]))
   ([(or P1 P2) [M1 M2]] (!byCases (assume P1 (!M1 P1))
                                   (assume P2 (!M2 P2))
                                   [disj])))))

(define (poly-sort s)
  (match s
    ((split _ (list-of `` _)) true)
    (_ false)))

(define (gnt t)
  (letrec ((f (lambda (s-lst eqn-lst fv-lst)
                (match s-lst
                  ((list-of s rest)
                     (check ((var? s) (f rest eqn-lst fv-lst))
                            (else (let ((fv (fresh-var))
                                        (new-eqns-fvs (check ((&& (null? (children s))
                                                                  (poly-sort (sort-of (root s)))) eqn-lst)
                                                             (else (add (= s fv) eqn-lst)))))

                                    (f rest new-eqns-fvs (add fv fv-lst))))))
                  (_ [eqn-lst fv-lst])))))
     (f (proper-subterms t) [] [])))


(define (gnt1 t)
  (letrec ((f (lambda (s-lst eqn-lst sigs)
                (match s-lst
                  ((list-of s rest)
                     (check ((var? s) (f rest eqn-lst sigs))
                            (else (let ((root-sym (root s))
                                        (args (children s))
                                        (arg-num (length args))
                                        (fv-args (fresh-vars arg-num))
                                        (fv-res (fresh-var))
                                        (new-eqns-sigs (check ((&& (null? args)
                                                                   (poly-sort (sort-of (root s))))
                                                               [eqn-lst sigs])
                                                              (else (letrec
                                                                      ((make-ae (lambda (args afvs res)
                                                                                  (match [args afvs]
                                                                                    ([[] _] res)
                                                                                    ([(list-of t1 rest)
                                                                                      (list-of fv rest1)]
                                                                                       (make-ae rest rest1
                                                                                         (add (= fv t1) res)))))))
                                                                       (let ((arg-eqns (make-ae args fv-args [])))
                                                                         [(join arg-eqns [(= s fv-res)] eqn-lst)
                                                                          (add [root-sym [fv-args fv-res]] sigs)])))))))
                                    (f rest (first new-eqns-sigs) (second new-eqns-sigs))))))
                  (_ [eqn-lst sigs])))))
     (f (proper-subterms t) [] [])))

##result for a given term s: A triple of the form [new-term eqns fvars]

(define (make-eqns f args vars)
  (let ((t (make-term f args)))
    (letrec ((loop (lambda (front-args tail-args vars res)
                     (match [vars tail-args]
                       ([[] _] res)
                       ([(list-of (some-var x) rest) (list-of targ rest2)]
                           (loop (join front-args [targ]) rest2 rest
                                 (add (= (make-term f (join front-args [x] rest2)) t) res)))))))
      (loop [] args vars []))))
```

```
(define (gnt2 t)
  (letrec ((f (lambda (s)
                (match s
                  ((some-var x) [[s [] []] [] []])
                  (((some-symbol g) (some-list args))
                      (let ((fv-args (fresh-vars (length args)))
                            (fv-res (fresh-var))
                            (all-fvs (add fv-res fv-args))
                            (eqns (make-eqns g all-fvs (add s args))))
                        (match (fLst args [] [] [])
                          ([new-terms new-eqns new-fvs] [[g (rev new-terms) [fv-args fv-res]] (join eqns new-eqns)
                                                         (join all-fvs new-fvs)])))))))
           (fLst (lambda (terms new-terms new-eqns new-fvs)
                   (match terms
                     ([] [new-terms new-eqns new-fvs])
                     ((list-of t rest) (match (f t)
                                         ([new-term eqns fvars] (fLst rest (add new-term new-terms)
                                                                           (join eqns new-eqns)
                                                                           (join fvars new-fvs)))))))))
    (f t)))


(define (gnt3 t)
  (letrec ((f (lambda (s)
                (match s
                  ((some-var x) [[s [] []] [] []])
                  (((some-symbol g) []) [[g [] []] [] []])
                  (((some-symbol g) (some-list args))
                      (let ((fv-args (fresh-vars (length args)))
                            (fv-res (fresh-var))
                            (all-fvs (add fv-res fv-args))
                            (aeqns (make-eqns g args fv-args))
                            (eqns (add (= fv-res s) aeqns)))
                        (match (fLst args [] [] [])
                          ([new-terms new-eqns new-fvs] [[g (rev new-terms) [fv-args fv-res]] (join eqns new-eqns)
                                                         (join all-fvs new-fvs)])))))))
           (fLst (lambda (terms new-terms new-eqns new-fvs)
                   (match terms
                     ([] [new-terms new-eqns new-fvs])
                     ((list-of t rest) (match (f t)
                                         ([new-term eqns fvars] (fLst rest (add new-term new-terms)
                                                                           (join eqns new-eqns)
                                                                           (join fvars new-fvs)))))))))
    (f t)))

## ##result for a given prop P: A triple of the form [big-P new-prop fvars],
## where big-P is P plus the variable-sort equations

(define (gnp2 P)
  (letrec ((f (lambda (P)
                (match P
                       ((some-atom A) (match (gnt3 A)
                                        ([new-term eqns fvl] [(conjoin (add P eqns)) new-term fvl])))
                       ((not (some-sent Q)) (match (f Q)
                                              ([big-prop new-prop fvl] [(not big-prop) [not [new-prop]] fvl])))
                       (((some-sent-con pc) P1 P2)
                                 (match [(f P1) (f P2)]
                                   ([[big-prop1 newP1 fvl1] [big-prop2 newP2 fvl2]]
                                        [(and big-prop1 big-prop2) [and [newP1 newP2]] (join fvl1 fvl2)])))
                       (((some-quant q) (some-var x) (some-sent B))
                               (match (f B)
                                 ([big-prop new-prop fvl] [(q x big-prop) [q [x new-prop]] fvl])))))))
    (f P)))


(define (sorts P)
     (match (gnp2 P)
       ([big-prop new-prop fvars]
          (let ((sorts (rd (map (lambda (v) (sort-of-var-in-prop v big-prop)) fvars))))
            (map (lambda (sort) (print (join "\n" sort "\n"))) sorts)))))
```

```
(define (nprop P)
  (letrec ((loop (lambda (P)
                   (match P
                     ((some-atom A) (match (gnt A)
                                      ([eqns fvs] [(conjoin (add P eqns)) fvs])))
                     ((not (some-sent Q)) (match (loop Q)
                                            ([Q' fvars'] [(not Q') fvars'])))
                     (((some-sent-con pc) P1 P2)
                          (match [(loop P1) (loop P2)]
                            ([[P1' fvars1] [P2' fvars2]] [(pc P1' P2') (join fvars1 fvars2)])))
                     (((some-quant q) (some-var x) B)
                         (match (loop B)
                           ([B' fvs] [(q x B') fvs])))))))
    (loop P)))



(define (nprop1 P)
  (letrec ((loop (lambda (P)
                   (match P
                     ((some-atom A) (match (gnt1 A)
                                      ([eqns sigs] [(conjoin (add P eqns)) sigs])))
                     ((not (some-sent Q)) (match (loop Q)
                                            ([Q' sigs'] [(not Q') sigs'])))
                     (((some-sent-con pc) P1 P2)
                          (match [(loop P1) (loop P2)]
                            ([[P1' sigs1] [P2' sigs2]] [(pc P1' P2') (join sigs1 sigs2)])))
                     (((some-quant q) (some-var x) B)
                         (match (loop B)
                           ([B' sigs] [(q x B') sigs])))))))
    (loop P)))

(define (prop-sorts P)
  (match (nprop P)
    ([Q fvs] (rd (map (lambda (v) (sort-of-var-in-prop v Q)) fvs)))))

(define lparen (head "("))
(define lbrack (head "["))
(define rbrack (head "]"))
(define blank (head " "))
(define rparen (head ")"))
(define new-line "\n")
(define quote "\"")

(define (join-strings sl sep)
  (match sl
    ([] "")
    ([str] str)
    ((list-of str rest) (join str [sep] (join-strings rest sep)))))


(define (join-expl-strings sl sep)
  (let ((quote (lambda (str) (join ['\"] str ['\"]))))
    (match sl
      ([] "")
      ([str] (quote str))
      ((list-of str rest) (join (quote str) [sep] (join-expl-strings rest sep))))))

(define (is-left-paren c)
  (equal? c lparen))

(define (mono-symbol-sort s)
  (let ((ar (arity-of s))
        (arg-vars (fresh-vars (string->id (symbol->string ar)))))
```

```
2029          (res-var (fresh-var))
2030          (P (= (make-term s arg-vars) res-var))
2031          (sovp (lambda (v P)
2032                  (let ((sort (sort-of-var-in-prop v P)))
2033                    (check ((is-left-paren (head sort))
2034                              (error (join "Error---the sort of symbol "
2035                                           (symbol->string s) " is polymorphic: " sort ".")))
2036                           (else sort))))))
2037      [(map (lambda (v) (sovp v P)) arg-vars) (sovp res-var P)]))
2038
2039
2040
2041
2042  (define mss mono-symbol-sort)
2043
2044  (define (mono-prop-sorts P)
2045    (letrec ((syms (list-diff (rd (get-prop-syms P)) [true false =]))
2046             (predicate? (lambda (sig)
2047                           (equal? (second sig) "Boolean")))
2048             (get (lambda (sl sorts sigs)
2049                    (match sl
2050                      ([] [(rd sorts) sigs])
2051                      ((list-of s rest) (let ((sig (mono-symbol-sort s)))
2052                                          (check ((predicate? sig)
2053                                                  (get rest (join sorts (head sig)) (add [s sig] sigs)))
2054                                                 (else
2055                                                   (get rest (join sorts (head sig) (tail sig))
2056                                                             (add [s sig] sigs)))))))))))
2057       (join [syms] (get syms [] []))))
2058
2059
2060  (define mps mono-prop-sorts)
2061
2062  (define (sortify-symbol f smap)
2063    (let ((n (arity-of f))
2064          (fvars (fresh-vars (plus n 1)))
2065          (arg-vars (tail fvars))
2066          (eqn (= (make-term f arg-vars) (head fvars)))
2067          (sorts (map (lambda (v) (sort-of-var-in-prop v eqn)) fvars))
2068          (arg-sort-syms (map smap (tail sorts))))
2069      (match (head sorts)
2070        ("Boolean" (forall* arg-vars
2071                     (if (make-term f arg-vars)
2072                         (conjoin (map (lambda (D-v) (match D-v
2073                                       ([D v] (D v)))) (zip arg-sort-syms arg-vars))))))
2074        (_ (forall* arg-vars
2075             (check ((equal? n 0) ((smap (head sorts)) (lhs eqn)))
2076                    (else (if (conjoin (map (lambda (D-v) (match D-v
2077                                              ([D v] (D v)))) (zip arg-sort-syms arg-vars)))
2078                              ((smap (head sorts)) (lhs eqn)))))))))
2079
2080
2081  (define (makeSingleSortedModel props file)
2082    (let ((univ "Single-Universe")
2083          (lp lparen)
2084          (rp rparen)
2085          (lb lbrack)
2086          (rb rbrack))
2087      (letrec ((repeat (lambda (n)
2088                         (check ((less? n 1) [])
2089                                ((equal? n 1) univ)
2090                                (else (join univ " " (repeat (minus n 1))))))))
2091               (write-line (lambda (s) (write-file file (join new-line s))))
2092               (declare-sort-sym (lambda (sort-name)
2093                                   (join new-line [lparen] "declare is" sort-name
2094                                         [blank lparen] "(T) -> " [lparen] "T"
2095                                         [rparen blank] "Boolean" [rparen rparen] new-line)))
2096               (define-smap (lambda (sort-names)
2097                              (seq
2098                                 (write-line (join [lp] "define " [lp] "sort-map str" [rp] new-line
```

```
                                                  "  " [lp] "match str"))
                                (map (lambda (sort-name)
                                       (write-line (join "    " [lp] quote sort-name quote
                                                         [blank] "is" sort-name [rp]))) sort-names)
                                (write-file file [rp rp])))))
     (match (mono-prop-sorts (conjoin props))
      ([syms sort-names sigs]
        (let ((is-sort-names (map (lambda (sn) (join "is" sn)) sort-names))
              (syms-string (join-strings (map symbol->string syms) blank))
              (string-sigs (map (lambda (p)
                                  (match p
                                    ([s x] [(symbol->string s) x]))) sigs)))
          (seq (map (lambda (s) (write-file file (declare-sort-sym s))) sort-names)
               (define-smap sort-names)
               (write-line (join new-line [lp] "define prop-list-1 " [lp] "univ-sort-axioms ["
                                 (join-strings is-sort-names blank) "]" [rp rp]))
               (write-line (join new-line [lp] "define prop-list-2 " [lp]
                                 "map (lambda (f) (sortify-symbol f sort-map" [rp rp blank lb]
                                  syms-string [rb rp rp]))
               (write-line (join new-line [lp] "define given-props " [lb] new-line
                                (join-strings (map val->string props) '\n) [rb rp]))
               (write-line (join new-line [lp] "define prop-list-3 " [lp]
                                 "map (lambda (P) (sortify-prop P sort-map" [rp rp blank]
                                 "given-props" [rp rp]))
               (write-line (join new-line "(get-multi-sorted-model (join prop-list-1 prop-list-2 prop-list-3) "
                                 [lb] (join-expl-strings is-sort-names blank) [rb blank]
                                 (val->string string-sigs) [rp])))))))))


(define mm makeSingleSortedModel)

(define gms get-multi-sorted-model)

(define prove-from vpf)

(define derive vpf)

(define (all-distinct terms)
  (letrec ((loop (lambda (terms res)
                   (match terms
                     ([] res)
                     ((list-of t more)
                      (loop more (join (map (lambda (s) (not (= t s))) more) res)))))))
    (loop terms [])))


(define (all-distinct-pairs terms)
  (filter (cprod terms terms)
          (lambda (pair)
            (match pair
              ([x y] (unequal? x y))))))

(define (distinct x y) (not (= x y)))

(define all-props ab)

(define show-all-props show-ab)

(define (axiom P)
  (forall* (rev (free-vars P)) P))

# The following method proves and returns the n-th conjunct of P

(define (conjunct n P)
  (dmatch P
    ((and (list-of _ _))
     (dcheck ((equal? n 1) (!left-and P))
             (else (!conjunct (minus n 1) (!right-and P)))))
    (_ (dcheck ((equal? n 1) (!claim P))
```

```
2169                (else (!proof-error "Error in conjunct call: index too large."))))))
2170
2171  (define (put-together p C)
2172    (dmatch C
2173      ((and (some-list args)) (!decompose C (method (_) (!conj-intro p))))
2174      (_ (!claim p))))
2175
2176  (define (print-vals L)
2177    (map (lambda (v) (print (join "\n" (val->string v) "\n"))) L))
2178
2179  (define (get P facts)
2180    (dmatch P
2181      ((some-atom A) (!vpf A facts))
2182      ((not (some-sent Q)) (dtry (!vpf P facts)
2183                                 (suppose-absurd Q
2184                                   (!vpf false (add Q facts)))))
2185      ((and (some-sent P1) (some-sent P2))  (dseq   (!get P1 facts)
2186                                                    (!get P2 facts)
2187                                                    (!both P1 P2)))
2188      ((or (some-sent P1) (some-sent P2))
2189         (dtry (dlet ((L1 (!get (if (not P1) P2) facts)))
2190                 (!spf (or P1 P2) [L1]))
2191               (!vpf (or P1 P2) facts)))
2192      ((if (some-sent P1) (some-sent P2))
2193                     (assume P1
2194                        (!get P2)))
2195      ((iff (some-sent P1) (some-sent P2))
2196         (dlet ((L1 (!get (if P1 P2) facts))
2197                (L2 (!get (if P2 P1) facts)))
2198           (!equiv L1 L2)))
2199      ((forall (some-var x) (some-var Q))
2200                     (pick-any v
2201                       (!get (replace-var x v Q) facts)))
2202      (_ (!vpf P facts))))
2203
2204
2205  ##                   (!decompose P1 (method (conjuncts)
2206  ##                                    (!get P2 (join (rd conjuncts) facts))))))
2207
2208
2209  (define (show-from P facts)
2210    (dlet ((Q (conjoin (add P facts))))
2211      (!get P (join facts (get-prop-sym-defs Q)))))
2212
2213  (define  (functional R)
2214    (let ((vars (fresh-vars (minus (arity-of R) 1)))
2215          (r1 (fresh-var))
2216          (r2 (fresh-var)))
2217      (forall* (join vars [r1 r2])
2218        (if (and (make-term R (join vars [r1]))
2219                 (make-term R (join vars [r2])))
2220            (= r1 r2)))))
2221
2222  (define (injectiveRel R)
2223    (forall ?x ?y ?z
2224      (if (and (R ?x ?z)
2225               (R ?y ?z))
2226          (= ?x ?y))))
2227
2228
2229  (define (uclos P)
2230    (forall* (free-vars P) P))
2231
2232  (define (eol? c)
2233    (equal? c '\n))
2234
2235  (define (id-chain plist)
2236    (dletrec ((loop (method (plist seed)
2237                      (dmatch plist
2238                        ([] (!claim seed))
```

```
                      ([eq premises] (!derive eq (add seed premises)))
                      ((list-of eq (list-of premises rest))
                          (dlet ((new-seed (!derive eq (add seed premises))))
                            (!loop rest new-seed)))))))
    (dmatch plist
      ((list-of eq (list-of premises rest))
         (dlet ((seed (!derive eq premises)))
           (!loop rest seed))))))

(define (neq s t)
  (not (= s t)))

(define (tprove  P max) (!derive P (ab) max))


(define (make-relation-prop R extension)
  (let ((x (fresh-var))
        (y (fresh-var)))
    (forall x y
       (iff (R x y)
            (or (map (lambda (pair)
                       (match pair
                         ([a b] (and (= x a) (= y b)))))) extension))))))


(define (commutative f)
  (forall ?x ?y
    (= (f ?x ?y) (f ?y ?x))))

(define (associative f)
  (forall ?x ?y ?z
    (= (f ?x (f ?y ?z))
       (f (f ?x ?y) ?z))))

(define (AC f)
  (and (commutative f) (associative f)))

(define (not-equal? x y) (negate (equal? x y)))

(define (repeat f n)
  (check ((equal? n 0) (f))
         (else (seq (f) (repeat f (minus n 1))))))



(define (univ-absurd P)
  (dmatch P
    ((forall (some-list vars) false) (false BY (!uspec* P (map (lambda (_) ?foo) vars))))))


(define (egen* goal witnesses)
  (dmatch [goal witnesses]
    ([_ [t]] (!egen goal t))
    ([(exists x body) (list-of t more)]
        (dseq (!egen* (replace-var x t body) more)
              (!egen goal t)))))

(define (continue)
  (seq (print "\nPress any key to continue...\n")
       (read) ()))

########################################################################
#                       SOME NEW METHODS (June 2008)
########################################################################


(define (or-same p)
  (dmatch p
    ((or _q _q)
     (!cases p
```

```
        (assume _q
          (!claim _q))
        (assume _q
          (!claim _q))))))

(define newline "\n")

(define (quant-swap premise)
  (dmatch premise
    ((forall _ (forall _ (some-sent _)))
      (pick-any y x
        (!uspec* premise [x y])))
    ((exists x (exists y (some-sent p)))
      (pick-witnesses (w1 w2) premise
        (!egen* (exists y x p) [w2 w1])))
    ((exists x (forall _ (some-sent p)))
      (pick-any y
        (pick-witness w premise witness-hyp
          (dlet ((q (!uspec witness-hyp y)))
            (!egen (exists x (replace-var w x q)) w)))))))




(define (neg-bicond prop)
  (dmatch prop
    ((not (iff P Q))
       (!two-cases (assume (if P Q)
                     (!either (not (if P Q))
                              (suppose-absurd (if Q P)
                                (!absurd (!equiv (if P Q) (if Q P)) prop))))
                (assume (not (if P Q))
                  (!either (not (if P Q)) (not (if Q P))))))))



(define (generalize vars M)
  (dletrec ((loop (method (vars eigen-vars)
                    (dmatch vars
                      ([] (!M (rev eigen-vars)))
                      ((list-of _ rest) (pick-any x
                                          (!loop rest (add x eigen-vars))))))))
    (!loop vars [])))


(define (mt premise-1 premise-2)
  (dmatch [premise-1 premise-2]
    ([(if (some-sent p) (some-sent q)) _]
      (dcheck ((complements? premise-2 q)
                (!by-contradiction (complement p)
                  (assume p
                    (!from-complements false
                                       (conclude q (!mp premise-1 p))
                                       premise-2))))
              (else (!fail (join "\nInvalid second argument given to mt: " (val->string premise-2))))))))

(define (app-dm p)
  (match p
    ((not (and (some-list args))) (or  (map app-dm (map complement args))))
    ((not (or (some-list args)))  (and (map app-dm (map complement args))))
    ((and (some-list args))  (not (or  (map app-dm (map complement args)))))
    ((or (some-list args))   (not (and (map app-dm (map complement args)))))
    (_ p)))

(define (quant-star q)
  (match q
    (forall forall*)
    (exists exists*)))
```

```
(define (app-dm-deep p)
  (match p
    ((not (and (some-list args))) (or  (map app-dm (map complement args))))
    ((not (or (some-list args)))  (and (map app-dm (map complement args))))
    (((some-sent-con pc) (some-list props)) (pc (map app-dm-deep props)))
    (((some-quant q) (some-list qvars) (some-sent body))
       ((quant-star q) qvars (app-dm-deep body)))
    (_ p)))


define (negated-bicond-1 premise) :=
  match premise {
    (~ (p1 <==> p2)) =>
      (!two-cases
         assume p1
           let {-p2 := (!by-contradiction (~ p2)
                         assume p2
                           (!absurd (!equiv assume p1 (!claim p2)
                                            assume p2 (!claim p1))
                                    premise))}
             (!left-either (!both p1 -p2) (~ p1 & p2))
         assume -p1 := (~ p1)
           let {p2 := (!by-contradiction p2
                         assume (~ p2)
                          (!absurd (!equiv assume p1
                                           (!from-complements p2 p1 (~ p1))
                                           assume p2
                                           (!from-complements p1 p2 (~ p2)))
                                   premise))}
             (!right-either (p1 & ~ p2) (!both -p1 p2)))
  }

define (negated-bicond-2 premise) :=
  match premise {
    ((p1 & (~ p2)) | ((~ p1) & p2)) =>
      let {goal := (~ (p1 <==> p2));
           M := method (equiv-detach case)
                  (!by-contradiction goal
                      assume goal' := (p1 <==> p2)
                        (!absurd (!equiv-detach goal')
                                 (!neg-cond case)))}
         (!cases premise
                assume case1 := (p1 & ~ p2)
                   (!M left-iff case1)
                assume case2 := (~ p1 & p2)
                   (!M right-iff (!comm case2)))
  }

define (negated-bicond premise) :=
  match premise {
    (~ (_ <==> _)) => (!negated-bicond-1 premise)
  | _ => (!negated-bicond-2 premise)
  }

(define eq-sym sym)

define (bicond-def premise) :=
  match premise {
    (p <==> q) => (!both (!left-iff premise)
                         (!right-iff premise))
  | ((p ==> q) & (q ==> p)) => (!equiv (!left-and premise)
                                       (!right-and premise))
  }

(define (bicond-def-2 premise)
  (dmatch premise
    ((iff p q)
      (!two-cases (assume p
                    (dseq (conclude q
                            (!mp (!left-iff premise) p))
```

```
                          (!either (!both p q)
                                   (and (not p) (not q)))))
                  (assume (not p)
                    (dseq (conclude (not q)
                            (!mt (!right-iff premise) (not p)))
                          (!either (and p q)
                                   (!both (not p) (not q)))))))
    ((or (and p q) (and (not p) (not q)))
      (!cases premise
         (assume-let (case1 (and p q))
           (!equiv (assume p
                     (!right-and case1))
                   (assume q
                     (!left-and case1))))
         (assume-let (case2 (and (not p) (not q)))
           (!equiv (assume p
                     (!from-complements q p (!left-and case2)))
                   (assume q
                     (!from-complements p q (!right-and case2)))))))))


define (bicond-def' premise) :=
  match premise {
    (p <==> q) =>
      let {[p' q'] := [(complement p) (complement q)]}
        (!two-cases
          assume p
            let {q := (!mp (!left-iff premise) p)}
              (!left-either (!both p q) (p' & q'))
          assume p'
            let {_ := conclude q'
                        (!mt (!right-iff premise) p')}
             (!right-either (p & q) (!both p' q')))
  | ((p & q) | (p' & q')) =>
      (!cases premise
         assume (p & q)
           (!equiv assume p (!claim q)
                   assume q (!claim p))
         assume (p' & q')
           (!equiv assume p (!from-complements q p p')
                   assume q (!from-complements p q q')))
  }


define (bicond-def-2 premise goal) :=
  match [premise goal] {
    [(p <==> q) (or (and p q) (and p' q'))] =>
        (!two-cases
          assume p
            let {q := (!mp (!left-iff premise) p)}
              (!left-either (!both p q) (p' & q'))
          assume p'
             (!two-cases assume q  (!from-complements goal (!mp (!right-iff premise) q) p')
                         assume q' (!right-either (and p q) (!both p' q'))))
  | [((p & q) | (p' & q')) _] =>
      (!cases premise
         assume (p & q)
           (!equiv assume p (!claim q)
                   assume q (!claim p))
         assume (p' & q')
           (!equiv assume p (!from-complements q p p')
                   assume q (!from-complements p q q')))
  }


(define (neg-bicond-def premise)
  (dmatch premise
    ((iff p q) (!equiv (assume h := (complement p)
                         (!mp (!contra-pos (!right-iff premise)) h))
                       (assume h := (complement q)
```

```
                          (!mp (!contra-pos (!left-iff premise)) h))))))


define (cd-dist-1 premise) :=
  match premise {
    (p & (q | r)) =>
      let {_ := (!left-and premise)}
        (!cases ((q | r) by (!right-and premise))
           assume q
             (!left-either (!both p q) (p & r))
           assume r
            (!right-either (p & q) (!both p r)))
  }

define (cd-dist-2 premise) :=
  match premise {
    ((p & q) | (p & r)) =>
      (!cases premise
         assume (p & q)
           (!both p (!left-either q r))
         assume (p & r)
           (!both p (!right-either q r)))
  }

define (dc-dist-1 premise) :=
  match premise {
    (p | (q & r)) =>
      (!cases premise
        assume p
          (!both (!left-either p q) (!left-either p r))
        assume (q & r)
          (!both (!right-either p q)
                 (!right-either p r)))
  }

define (dc-dist-2 premise) :=
  match premise {
    ((p | q) & (p | r)) =>
      let {_ := (!left-and premise);
           _ := (!right-and premise)}
        (!cases (p | q)
          assume p
            (!left-either p (q & r))
          assume q
            (!cases (p | r)
               assume p
                 (!left-either p (q & r))
               assume r
                 (!right-either p (!both q r))))
  }

define (dist premise) :=
  match premise {
    ((p | _) & (p | _)) => (!dc-dist-2 premise)
  | (_ & (_ | _))       => (!cd-dist-1 premise)
  | ((p & _) | (p & _)) => (!cd-dist-2 premise)
  | (p | (q & r))       => (!dc-dist-1 premise)
  }

(define (sym premise)
  (dmatch premise
    ((= _ _) (!eq-sym premise))
    ((not (= s t)) (!by-contradiction (not (= t s))
                      (assume (= t s)
                        (!absurd (!eq-sym (= t s)) premise))))))

(define (ref-equiv p)
  (!equiv (assume p (!claim p))
          (assume p (!claim p))))
```

```
2589
2590 (define (equiv-tran premise-1 premise-2)
2591   (dmatch [premise-1 premise-2]
2592     ([(iff p1 p2) (iff p2 p3)] (!equiv (assume p1
2593                                          (!mp (!left-iff premise-2)
2594                                               (!mp (!left-iff premise-1) p1)))
2595                                        (assume p3
2596                                          (!mp (!right-iff premise-1)
2597                                               (!mp (!right-iff premise-2) p3)))))))
2598
2599
2600
2601 (define (not-cong premise)
2602   (dmatch premise
2603     ((iff p1 p2)
2604       (dlet ((cond-1 (assume (not p1)
2605                        (!by-contradiction (not p2)
2606                          (assume p2
2607                            (!absurd (!mp (!right-iff premise) p2)
2608                                     (not p1))))))
2609              (cond-2 (assume (not p2)
2610                        (!by-contradiction (not p1)
2611                           (assume p1
2612                             (!absurd (!mp (!left-iff premise) p1)
2613                                      (not p2)))))))
2614         (!equiv cond-1 cond-2)))))
2615
2616 (define (not-cong' premise)
2617   (dmatch premise
2618     ((iff p1 p2)
2619       (dlet (([p1' p2'] [(complement p1) (complement p2)])
2620              (cond-1 (assume p1'
2621                        (!by-contradiction p2'
2622                          (assume p2
2623                            (!from-complements false (!mp (!right-iff premise) p2) p1')))))
2624              (cond-2 (assume p2'
2625                        (!by-contradiction p1'
2626                           (assume p1
2627                             (!from-complements false (!mp (!left-iff premise) p1) p2'))))))
2628         (!equiv cond-1 cond-2)))))
2629
2630
2631 (define (and-cong premise1 premise2)
2632   (dmatch [premise1 premise2]
2633     ([(iff p1 q1) (iff p2 q2)]
2634        (dlet ((cond-1 (assume (and p1 p2)
2635                         (!both (!mp (!left-iff premise1)
2636                                     (!left-and (and p1 p2)))
2637                                (!mp (!left-iff premise2)
2638                                     (!right-and (and p1 p2))))))
2639                (cond-2 (assume (and q1 q2)
2640                          (!both (!mp (!right-iff premise1)
2641                                      (!left-and (and q1 q2)))
2642                                 (!mp (!right-iff premise2)
2643                                      (!right-and (and q1 q2)))))))
2644          (!equiv cond-1 cond-2)))))
2645
2646 (define (uni-and-cong premise1 premise2)
2647   (dmatch [premise1 premise2]
2648     ([(if p1 q1) (if p2 q2)]
2649       (assume (and p1 p2)
2650         (!both (!mp premise1
2651                     (!left-and (and p1 p2)))
2652                (!mp premise2
2653                     (!right-and (and p1 p2))))))))
2654
2655 (define (decompose-conditionals Cs)
2656    (letrec ((loop (lambda (Cs antecedents consequents)
2657                     (match Cs
2658                       ([] [(rev antecedents) (rev consequents)])
```

```
                        ((list-of (if (some-sent p1) (some-sent p2)) more)
                            (loop more (add p1 antecedents) (add p2 consequents)))))))
        (loop Cs [] [])))

(define (uni-and-cong* premises)
    (dlet (([antecedents consequents] (decompose-conditionals premises)))
       (assume (and antecedents)
         (!map-method (method (cond)
                         (!mp cond (antecedent cond)))
                      premises
                      (method (_)
                        (!and-intro consequents))))))

(define (or-cong premise1 premise2)
  (dmatch [premise1 premise2]
    ([(iff p1 q1) (iff p2 q2)]
      (dlet ((cond-1 (assume (or p1 p2)
                        (!cases (or p1 p2)
                                (assume p1
                                  (!either (!mp (!left-iff premise1)
                                                p1)
                                           q2))
                                (assume p2
                                  (!either q1
                                           (!mp (!left-iff premise2)
                                                p2))))))
             (cond-2 (assume (or q1 q2)
                       (!cases (or q1 q2)
                               (assume q1
                                 (!either (!mp (!right-iff premise1)
                                               q1)
                                          p2))
                               (assume q2
                                 (!either p1
                                          (!mp (!right-iff premise2)
                                               q2)))))))
        (!equiv cond-1 cond-2)))))


(define (uni-or-cong premise1 premise2)
  (dmatch [premise1 premise2]
    ([(if p1 q1) (if p2 q2)]
      (assume (or p1 p2)
        (!cases (or p1 p2)
                (assume p1
                  (!either (!mp premise1 p1) q2))
                (assume p2
                  (!either q1
                    (!mp premise2 p2))))))))

(define (uni-or-cong* premises)
    (dlet (([antecedents consequents] (decompose-conditionals premises)))
       (assume h := (or antecedents)
          (!map-method (method (cond)
                          (dmatch cond
                             ((if (some-sent p1) (some-sent p2))
                                (assume p1
                                   (dlet ((_ (!mp cond p1)))
                                     (!either consequents))))))
                       premises
                       (method (conditionals)
                         (!cases h conditionals))))))

(define (if-cong premise1 premise2)
  (dmatch [premise1 premise2]
    ([(iff p1 q1) (iff p2 q2)]
        (dlet ((cond-1 (assume (if p1 p2)
                         (assume q1
                            (!mp (!left-iff premise2)
                                 (!mp (if p1 p2)
```

```
                                        (!mp (!right-iff premise1) q1))))))
                (cond-2 (assume (if q1 q2)
                          (assume p1
                            (!mp (!right-iff premise2)
                                 (!mp (if q1 q2)
                                      (!mp (!left-iff premise1) p1)))))))
           (!equiv cond-1 cond-2)))))


(define (iff-cong premise1 premise2)
  (dmatch [premise1 premise2]
    ([(iff p1 q1) (iff p2 q2)]
         (dlet ((cond-1 (assume (iff p1 p2)
                          (!equiv (assume q1
                                    (!mp (!left-iff premise2)
                                         (!mp (!left-iff (iff p1 p2))
                                              (!mp (!right-iff premise1) q1))))
                                  (assume q2
                                    (!mp (!left-iff premise1)
                                         (!mp (!right-iff (iff p1 p2))
                                              (!mp (!right-iff premise2) q2)))))))
                (cond-2 (assume (iff q1 q2)
                          (!equiv (assume p1
                                    (!mp (!right-iff premise2)
                                         (!mp (!left-iff (iff q1 q2))
                                              (!mp (!left-iff premise1) p1))))
                                  (assume p2
                                    (!mp (!right-iff premise1)
                                         (!mp (!right-iff (iff q1 q2))
                                              (!mp (!left-iff premise2) p2))))))))
           (!equiv cond-1 cond-2)))))


(define (ugen-cong p q M)
  (dmatch [p q]
    ([(forall (some-var v1) body1) (forall (some-var v2) body2)]
       (conclude (iff p q)
         (!equiv (assume p
                   (pick-any v
                     (dlet ((r (!uspec p v))
                            (body2' (replace-var v2 v body2))
                            (th (!M r body2')))
                       (!mp (!left-iff th) r))))
                 (assume q
                   (pick-any v
                     (dlet ((r (!uspec q v))
                            (body1' (replace-var v1 v body1))
                            (th (!M r body1')))
                       (!mp (!left-iff th) r)))))))))

(define (egen-cong p q M)
  (dmatch [p q]
    ([(exists (some-var v1) body1) (exists (some-var v2) body2)]
       (conclude (iff p q)
         (!equiv (assume p
                   (pick-witness v p
                     (dlet ((body1' (replace-var v1 v body1))
                            (body2' (replace-var v2 v body2))
                            (th (!M body1' body2'))
                            (_  (!mp (!left-iff th) body1')))
                       (!egen q v))))
                 (assume q
                   (pick-witness v q
                     (dlet ((body2' (replace-var v2 v body2))
                            (body1' (replace-var v1 v body1))
                            (th (!M body2' body1'))
                            (_  (!mp (!left-iff th) body2')))
                       (!egen p v)))))))))

(define (choose-cong-method pc)
```

```
  (match pc
    (and and-cong)
    (or  or-cong)
    (if  if-cong)
    (iff iff-cong)))

# (define (prove-equiv p q methods)
#   (dtry (!find-some methods
#                     (method (M)
#                       (!equiv (assume p (!M p))
#                               (assume q (!M q))))
#                     (method (_) (!true-intro)))
#         (dcheck ((equal? p q) (!ref-equiv p))
#                 (else (dmatch [p q]
#                         ([(not p1) (not q1)]
#                           (!not-cong (!prove-equiv p1 q1 methods)))
#                         ([((some-sent-con pc) p1 p2) (pc q1 q2)]
#                           (!(choose-cong-method pc)
#                             (!prove-equiv p1 q1 methods)
#                             (!prove-equiv p2 q2 methods))))))))

(define (prove-equiv p q methods)
  (dtry (!find-some methods
                    (method (M)
                        (dtry (!equiv (assume p (!M p))
                                      (assume q (!M q)))
                              (!equiv (assume p (!M p q))
                                      (assume q (!M q p)))))
                    (method (_) (!true-intro)))
        (dcheck ((equal? p q) (!ref-equiv p))
                (else (dlet ((M (method (p1 p2)
                                  (!prove-equiv p1 p2 methods))))
                        (dmatch [p q]
                          ([(not p1) (not q1)]
                            (!not-cong (!prove-equiv p1 q1 methods)))
                          ([((some-sent-con pc) p1 p2) (pc q1 q2)]
                            (!(choose-cong-method pc)
                              (!prove-equiv p1 q1 methods)
                              (!prove-equiv p2 q2 methods)))
                          ([(forall _ _) (forall _ _)]
                            (!ugen-cong p q M))
                          ([(exists _ _) (exists _ _)]
                            (!egen-cong p q M))))))))

 (define (replace p q methods)
   (dseq (conclude (iff p q)
            (!prove-equiv p q methods))
         (conclude q
           (!mp (!left-iff (iff p q)) p))))

(define transform replace)

(define (method? x)
  (match x
    ((some-method _) true)
    (_ false)))

(define
  (term-leaves t)
    (match t
      (((some-symbol _) (list-of s rest)) (flatten (map term-leaves (add s rest))))
      (_ [t]))
  (prop-leaves p)
    (match p
      ((some-atom _) (term-leaves p))
      (((some-sent-con _) (some-list args)) (flatten (map prop-leaves args)))
      (((some-quant _) (some-var _) body) (prop-leaves body))))

(define (leaves v)
  (match v
```

```
2869      ((some-term t) (rd (term-leaves t)))
2870      ((some-sent p) (rd (prop-leaves p)))))
2871
2872  (define (dedup L)
2873    (let ((T (table 500))
2874          (_ (map-proc (lambda (x) (let ((already (try (table-lookup T x) false))) (check (already ()) (else (table-add 
2875      (map first (table->list T))))
2876
2877  (define (dedup L)
2878    (let ((T (table 500))
2879          (L-out (cell [])))
2880          (_ (map-proc (lambda (x) (let ((already (try (table-lookup T x) false)))
2881                                     (check (already ())
2882                                            (else (let ((_ (table-add T [x --> true]))
2883                                                        (_ (set! L-out (add x (ref L-out)))))
2884                                                    (()))))))
2885                       L)))
2886      (rev (ref L-out))))
2887
2888  (define (atoms p)
2889    (letrec ((atoms (lambda (p)
2890                      (match p
2891                        ((some-atom _) [p])
2892                        (((some-sent-con _) (some-list args)) (flatten (map atoms args)))
2893                        (((some-quant _) (some-list _) (some-sent body)) (atoms body))))))
2894      (dedup (atoms p))))
2895
2896  (define
2897    (replace-atoms-aux p new-atoms)
2898      (match p
2899         ((some-atom _) [(first new-atoms) (tail new-atoms)])
2900         (((some-sent-con pc) (some-list args)) (match (replace-atoms-lst args new-atoms [])
2901                                                       ([args' rest] [(pc args') rest])))
2902         (((some-quant q) (some-var x) body) (match (replace-atoms-aux body new-atoms)
2903                                                    ([body' rest] [(q x body') rest])))
2904    (replace-atoms-lst props new-atoms res)
2905     (match props
2906       ([] [(rev res) new-atoms])
2907       ((list-of p more) (match (replace-atoms-aux p new-atoms)
2908                           ([p' rest-atoms] (replace-atoms-lst more rest-atoms (add p' res)))))))
2909
2910  (define (replace-atoms p new-atoms)
2911     (first (replace-atoms-aux p new-atoms)))
2912
2913  (define (get-subterms t vars)
2914    (letrec ((f  (lambda (t L)
2915                   (match t
2916                     (((some-symbol f) (some-list args)) (f* args (add t L)))
2917                     (_ (add t L)))))
2918             (f* (lambda (terms L)
2919                   (match terms
2920                     ([] L)
2921                     ((list-of t rest) (f* rest (f t L)))))))
2922      (f t vars)))
2923
2924  (define (choice-prop-subterms p)
2925    (letrec ((f (lambda (p var-list)
2926                  (match p
2927                    ((some-atom A) (remove A (get-subterms A var-list)))
2928                    ((not Q) (f Q var-list))
2929                    (((some-sent-con pc) p1 p2) (f p2 (f p1 var-list)))
2930                    (((some-quant Q) (some-var x) (some-sent B))
2931                      (filter (f B var-list)
2932                              (lambda (t) (negate (member? x (vars t))))))))))
2933      (rd (f p []))))
2934
2935  (define (choice-prop-subterms p)
2936    (letrec ((f (lambda (p)
2937                  (match p
2938                    ((some-atom A) (remove A (get-subterms A []))))
```

```
                  ((not Q) (f Q))
                  (((some-sent-con pc) (some-list props)) (fold join (map f props) []))
                  (((some-quant Q) (some-var x) (some-sent B))
                    (filter (f B)
                            (lambda (t) (negate (member? x (vars t))))))))))
    (rd (f p))))

(define (choice-prop-subterms* props)
  (fold join (map choice-prop-subterms props) []))


## (find-max L f success failure) takes a list L
## and a unary scoring procedure f that can receive
## any element x of L and will return a non-negative integer
## (f x). If L is empty or if there is no element x in L
## with a positive score (i.e., such that (f x) > 0), then
## the nullary 'failure' continuation is invoked. Otherwise,
## the element of L with the maximum score is passed
## to the unary 'success' continuation.

(define (find-max L f success failure)
  (letrec ((loop (lambda (L current-best current-max)
                   (match L
                     ([] [current-best current-max])
                     ((list-of x rest) (let ((x-score (f x)))
                                         (check ((less? current-max x-score)
                                                 (loop rest x x-score))
                                                (else (loop rest current-best current-max)))))))))
    (match L
      ([] (failure))
      ((list-of x _) (match (loop L x (- 1))
                       ([best max] (check ((less? 0 max) (success best))
                                          (else (failure)))))))))


(define (log-floor n)
  (check ((less? n 2) 0)
         (else (plus 1 (log-floor (div n 2))))))

## The method breadth-first takes a starting premise 'start' (in the a.b.);
## a target sentence 'target'; a list of unary methods [M_1... M_k],
## 'methods', each of which takes a single premise as input; a maximum
## depth 'max-depth'; and a nullary failure continuation 'failure'.
## The method will derive the target iff the latter can be obtained
## from the starting premise via any finite sequence of applications
## of methods taken from the set {M_1,...,M_k}, provided that the
## said sequence is of length <= max_depth (the max_depth parameter
## is there to ensure termination, since, if the target is not
## in fact obtainable from the premise via the given methods, the
## search would go on indefinitely. The search tree is expanded
## in a breadth-first manner to ensure completeness. If we have
## exceeded max-depth without yet deriving the target, the failure
## continuation is invoked. [Note that any method M_i can freely
## fail on any sentence. The search will simply discount such
## failures. This means that the list [M_1 ... M_k] can actually
## contain methods of arbitrary arity; in fact it can contain
## values of any type, e.g. copies of the unit value ().]

(define (breadth-first start target methods max-depth failure)
  (dletrec ((loop (method (queue n)
                    (dmatch queue
                      ((list-of p rest)
                        (dcheck ((equal? p target) (!claim p))
                                ((less? max-depth (log-floor n)) (!failure))
                                (else (!map-method (method (M) (dtry (!M p)
                                                                     (!true-intro)))
                                                   methods
                                                   (method (new-theorems)
                                                     (!loop (join rest new-theorems) (plus n 1))))))))))))
    (!loop [start] 1)))
```

```
3009
3010 (define bf breadth-first)
3011
3012 # Examples:
3013 #
3014 # (define start (and ?A (and ?B (and ?C ?D))))
3015 #
3016 # (define target ?D)
3017 #
3018 # (define (failure)
3019 #   (!proof-error "Failure..."))
3020 #
3021 # (assume start
3022 #   (!bf start target [right-and] 4 failure))
3023 #
3024 # (define start
3025 #   (not (not (not (not (and ?A (and ?B (and ?C ?D))))))))
3026 #
3027 # (define target ?C)
3028 #
3029 # (assume start
3030 #   (!bf start target [dn right-and left-and] 10 failure))
3031 #
3032 # (define start (and ?A (and ?B ?C)))
3033 # (define target (or ?C ?D))
3034 #
3035 # (assume start
3036 #   (!bf start target [right-and comm true-intro (method (p) (!either p ?D))] 5 failure))
3037
3038 (define (rule-antecedent R)
3039   (match R
3040     ((forall (some-list _) (if p _)) p)
3041     ((forall (some-list _) (iff p _)) p)))
3042
3043 (define (rule-consequent R)
3044   (match R
3045     ((forall (some-list _) (if _ q)) q)
3046     ((forall (some-list _) (iff _ q)) q)))
3047
3048 (define (rule->method R)
3049   (method (p)
3050     (dmatch (match-props p (rule-antecedent R))
3051       ((some-sub sub) (!instance R (sub (qvars-of R))))
3052       (_ (dmatch (match-props p (rule-consequent R))
3053            ((some-sub sub) (dlet ((p (!right-iff (!uspec* R (sub (qvars-of R))))))
3054                              (!mp p (!prove-antecedent p)))))))))
3055
3056 (define (fact->cond fact)
3057   (dmatch fact
3058     ((forall (some-list uvars) body)
3059       (dmatch body
3060         ((if _ _) (!fail))
3061         ((iff _ _) (!fail))
3062         (_ (!generalize uvars
3063               (method (eigen-vars)
3064                  (assume true
3065                    (!uspec* fact eigen-vars)))))))))
3066
3067 (define (ufact->cond fact)
3068   (dmatch fact
3069     ((forall (list-of _ _) _)
3070       (!fact->cond fact))))
3071
3072 (define (fact->bicond fact)
3073   (dmatch fact
3074     ((forall (some-list uvars) body)
3075       (dmatch body
3076        ((if _ _) (!fail))
3077        ((iff _ _) (!fail))
3078        (_ (!generalize uvars
```

```
            (method (eigen-vars)
               (!equiv (assume true
                         (!uspec* fact eigen-vars))
                       (assume (replace-vars uvars eigen-vars body)
                         (!true-intro))))))))))

## K: New, tentative definition of fact->bicond, June 15, 2010:

(define (fact->bicond fact)
 (dlet ((M (method (uvars body)
             (!generalize uvars
                (method (eigen-vars)
                   (!equiv (assume true
                             (!uspec* fact eigen-vars))
                           (assume (replace-vars uvars eigen-vars body)
                             (!true-intro))))))))))
  (dmatch fact
    ((forall (some-list uvars) body)
      (dmatch [body (null? uvars)]
       ([(if _ _) false] (!fail))
       ([(if _ _) true] (!M uvars body))
       ([(iff _ _) false] (!fail))
       ([(iff _ _) true] (!M uvars body))
       (_ (!M uvars body))))))

(define (ufact->bicond fact)
  (dmatch fact
    ((forall (list-of _ _) _)
       (!fact->bicond fact))))

(define (identity? p)
  (match p
    ((= _ _) true)
    (_ false)))

(define old-egen egen)

(define (egen' target)
  (dmatch target
    ((exists (some-var x) (some-sent body))
       (dlet ((sub-cell (cell ()))
              (_ (fetch (lambda (p)
                          (match (match-props p body)
                            ((some-sub sub) (seq (set! sub-cell sub) true))
                            (_ false))))))
         (dmatch (ref sub-cell)
           ((some-sub sub) (!old-egen target (sub x))))))))



(define (augment-left p)
  (method (q)
    (!both p q)))

(define augment augment-left)

(define (augment-right q)
  (method (p)
    (!both p q)))

(define (alternate left right)
  (!either right))

(define (comm-absurd P1 P2)
  (dmatch [P1 P2]
    ([P (not P)] (!absurd P1 P2))
    ([(not P) P] (!absurd P2 P1))))
```

```
(define (qn-2 premise)
  (dmatch premise
    ((exists x p)
       (!by-contradiction' (not (forall x (complement p)))
         (assume (forall x (complement p))
           (pick-witness w premise witness-premise
              (!from-complements false
                                 (!uspec (forall x (complement p)) w)
                                 witness-premise)))))))

(define (qn2-strict premise)
  (dmatch premise
    ((exists x (not (some-sent P)))
       (suppose-absurd (forall x P)
         (pick-witness w premise
            (!absurd (!uspec (forall x P) w)
                     (replace-var x w (not P))))))))

(define (qn-3 premise)
  (dmatch premise
    ((not (exists x (some-sent p)))
      (pick-any y
        (dlet ((q (replace-var x y p)))
          (!by-contradiction' (complement q)
            (assume q
              (!absurd (!egen (exists y q) y)
                       premise)))))))))


(define (qn3-strict premise)
  (dmatch premise
    ((not (exists x (some-sent p)))
       (pick-any y
         (suppose-absurd-let (hyp (replace-var x y p))
           (!absurd (!egen (exists x p) y) premise))))))

(define (qn-1 premise)
  (dmatch premise
    ((not (forall x (some-sent p)))
       (!by-contradiction' (exists x (complement p))
         (assume-let (hyp (not (exists x (complement p))))
           (!absurd (conclude (forall x p)
                      (pick-any x (!uspec (!qn-3 hyp) x)))
                    premise))))))

(define (qn1-strict premise)
  (dmatch premise
    ((not (forall x P))
       (!dn (suppose-absurd-let (hyp (not (exists x (not P))))
              (!absurd (pick-any x (!dn (!uspec (!qn3-strict hyp) x))) premise))))))



(define (qn-4 premise)
  (dmatch premise
    ((forall x (some-sent p))
       (!by-contradiction' (not (exists x (complement p)))
          (assume-let (hyp (exists x (complement p)))
            (pick-witness w hyp witness-premise
              (!from-complements false witness-premise
                                 (!uspec premise w)))))))


(define (qn4-strict premise)
  (dmatch premise
    ((forall x (not (some-sent p)))
       (suppose-absurd-let (hyp (exists x p))
         (pick-witness w hyp
           (!absurd (replace-var x w p)
                    (!uspec premise w)))))))
```

```
3219
3220 (define (qn premise)
3221   (dmatch premise
3222     ((not (forall x P)) (!qn-1 premise))
3223     ((exists x (not P)) (!qn-2 premise))
3224     ((not (exists x P)) (!qn-3 premise))
3225     ((forall x (not P)) (!qn-4 premise))))
3226
3227
3228 (define (qn-strict premise)
3229   (dmatch premise
3230     ((not (forall x P)) (!qn1-strict premise))
3231     ((exists x (not P)) (!qn2-strict premise))
3232     ((not (exists x P)) (!qn3-strict premise))
3233     ((forall x (not P)) (!qn4-strict premise))
3234     (_ (!claim premise))))
3235
3236
3237 ## qn* is a version of the qn ("quantifier negation") methods that works
3238 ## iteratively, with arbitrarily long chains of quantifiers preceded by a
3239 ## negation symbol. For instance, suppose that p := (not (forall ?x (forall ?y q)))
3240 ## is in the a.b. Then (!qn p) will produce (exists ?x (not (forall ?y q))),
3241 ## i.e., it will only push the negation sign inward by one position, flipping
3242 ## only one (the very first) quantifier in the process. By constrast,
3243 ## (!qn* p) will produce (exists ?x (exists ?y (not q))).
3244 ## Likewise, (!qn* (not (forall ?x (exists ?y (forall ?z q))))) will produce
3245 ## (exists ?x (forall ?y (exists ?z (not q)))), and so on.
3246 ## There is also a strict version of the method, called qn-strict*,
3247 ## which, like the strict version of qn (qn-strict), determines whether
3248 ## the matrix of the given sentence (the body of the quantified sentence)
3249 ## will be explicitly negated or not. For instance,
3250 ## (!qn-strict* (not (forall ?x (forall ?y (not q))))) will produce
3251 ## (exists ?x (exists ?y (not (not q)))), explicitly negating the body
3252 ## (not q), whereas the non-strict version of the method will apply
3253 ## double negation, resulting in (exists ?x (exists ?y q)).
3254
3255 (define (qn*-comp p)
3256   (match p
3257     ((not (forall (some-var x) body)) (exists x (qn*-comp (not body))))
3258     ((not (exists (some-var x) body)) (forall x (qn*-comp (not body))))
3259     ((not (not q)) q)
3260     ((forall (some-var x) body) (not (exists x (qn*-comp (not body)))))
3261     ((exists (some-var x) body) (not (forall x (qn*-comp (not body)))))
3262     (_ p)))
3263
3264 (define (qn*-comp-strict p)
3265   (match p
3266     ((not (forall (some-var x) body)) (exists x (qn*-comp-strict (not body))))
3267     ((not (exists (some-var x) body)) (forall x (qn*-comp-strict (not body))))
3268     ((forall (some-var x) body) (not (exists x (qn*-comp-strict (not body)))))
3269     ((exists (some-var x) body) (not (forall x (qn*-comp-strict (not body)))))
3270     (_ p)))
3271
3272 (define (qn0* premise)
3273   (dmatch premise
3274     ((not (forall (some-var x) (some-sent body)))
3275       (dmatch (!qn premise)
3276         ((th as (exists y (body' as (not ((some-quant _) (some-var _) (some-sent _))))))
3277           (pick-witness w th witness
3278            (dlet ((body'' (!qn0* (replace-var y w body'))))
3279               (!egen (exists w body'') w))))
3280         (res (!claim res))))
3281     ((not (exists (some-var x) (some-sent body)))
3282       (dmatch (!qn premise)
3283         ((th as (forall (some-var y) (body' as (not ((some-quant _) (some-var _) (some-sent _))))))
3284            (pick-any w:(sort-of y)
3285              (!qn0* (!uspec th w))))
3286         (res (!claim res))))
3287     (_ (!qn premise))))
3288
```

```
(define (qn* p)
 (conclude (qn*-comp p)
    (!qn0* p)))

(define (qn-strict0* premise)
  (dmatch premise
    ((not (forall (some-var x) (some-sent body)))
      (dmatch (!qn-strict premise)
        ((th as (exists y body'))
          (pick-witness w th witness ##
           (dlet ((body'' (!qn-strict0* (replace-var y w body'))))
             (!egen (exists w body'') w))))
        (res (!claim res))))
    ((not (exists (some-var x) (some-sent body)))
      (dmatch (!qn-strict premise)
        ((th as (forall (some-var y) body'))
          (pick-any w:(sort-of y)
            (!qn-strict0* (!uspec th w))))
        (res (!claim res))))
    (_ (!qn-strict premise))))

(define (qn-strict* p)
  (conclude (qn*-comp-strict p)
    (!qn-strict0* p)))

(define (inc c)
  (let ((res (ref c)))
    (seq (set! c (plus res 1))
         res)))


(define (reiterate p)
  (method (_)
    (!claim p)))

(define (look-up table str)
  ((table "look-up") str))

(define (enter table str v)
  ((table "enter") str v))

(define (remove table str)
  ((table "remove") str))

(define (hash-table-size table)
  ((table "size")))

(define (show-table table)
  ((table "show")))

(define (term-look-up table t)
  ((table "look-up") t))

(define (term-enter table t v)
  ((table "enter") t v))

(define (term-table-remove table t)
  ((table "remove") t))

(define (term-enter table t v)
  ((table "enter") t v))

(define (term-table-size table)
  ((table "size")))

(define (show-term-table table)
  ((table "show")))

(define make-var-hash-table make-hash-table)

```

```
(define (look-up-var table v)
  (look-up table (var->string v)))

(define (enter-var table variable key)
  (enter table (var->string variable) key))

(define (var-table-size table)
  (table-size table))

(define (show-var-table table)
  (show-table table))

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

(define (look-up-symbol table v)
  (look-up table (symbol->string v)))

(define (enter-symbol table sym v)
  (enter table (symbol->string sym) v))

(define (symbol-table-size table)
  (table-size table))

(define (show-symbol-table table)
  (show-table table))


## The following returns all permutations of k elements
## taken from list L.

(define (permutations L k)
  (letrec ((f (lambda (x)
                (let ((perms (permutations L (minus k 1))))
                  (map (lambda (P) (add x P)) perms)))))
   (check ((equal? k 0) [[]])
          (else (fold join (map f L) [])))))

(define
  (prop-size p)
    (match p
      ((some-atom _) 1)
      (((some-sent-con _) (some-list props)) (plus 1 (prop-size* props)))
      (((some-quant _) (some-var _) (some-sent q)) (plus 2 (prop-size q))))
  (prop-size* props)
    (match props
      ([] 0)
      ((list-of p more) (plus (prop-size p) (prop-size* more)))))


(define
  (prop-size1 p)
    (match p
      ((some-atom t) (term-size t))
      (((some-sent-con _) (some-list props)) (plus 1 (prop-size1* props)))
      (((some-quant _) (some-var _) (some-sent q)) (plus 2 (prop-size1 q))))
  (prop-size1* props)
    (match props
      ([] 0)
      ((list-of p more) (plus (prop-size1 p) (prop-size1* more)))))




(define (fcong1 id fsym)
  (dmatch id
    ((= s t) (!fcong (= (fsym s) (fsym t))))))

(define (cong-method fsym)
  (method (id) (!fcong1 id fsym)))

```

```
3429
3430 (define (ground? t)
3431   (&& (term? t)
3432       (null? (vars t))))
3433
3434 (define (all-ground? terms)
3435    (for-each terms ground?))
3436
3437 (define (syms t)
3438   (rd (get-term-syms t)))
3439
3440 (define (canonical? t)
3441   (&& (ground? t)
3442       (for-each (syms t) constructor?)))
3443
3444 (define (super-canonical? t)
3445   (&& (ground? t)
3446       (for-each (syms t) (lambda (c) (|| (constructor? c) (real-numeral? c) (meta-id? c))))))
3447
3448 (primitive-method (int-comp i j)
3449   (check ((&& (integer-numeral? i) (integer-numeral? j))
3450           (check ((num-equal? i j) (= i j))
3451                  (else (not (= i j)))))))
3452
3453 (define (int-comp i j)
3454   (!force (check ((&& (integer-numeral? i) (integer-numeral? j))
3455           (check ((num-equal? i j) (= i j))
3456                  (else (not (= i j))))))))
3457
3458 (primitive-method (real-comp i j)
3459   (check ((&& (real-numeral? i) (real-numeral? j))
3460           (check ((num-equal? i j) (= i j))
3461                  (else (not (= i j)))))))
3462
3463 (define (real-comp i j)
3464   (!force (check ((&& (real-numeral? i) (real-numeral? j))
3465                  (check ((num-equal? i j) (= i j))
3466                         (else (not (= i j))))))))
3467
3468
3469 (primitive-method (id-comp i j)
3470   (check ((&& (meta-id? i) (meta-id? j))
3471           (check ((equal? i j) (= i j))
3472                  (else (not (= i j)))))))
3473
3474 (define (id-comp i j)
3475   (!force (check ((&& (meta-id? i) (meta-id? j))
3476           (check ((equal? i j) (= i j))
3477                  (else (not (= i j))))))))
3478
3479 (primitive-method (show-unequal t1 t2)
3480   (check ((&& (canonical? t1)
3481               (canonical? t2)
3482               (equal? (sort-of t1) (sort-of t2))
3483               (negate (equal? t1 t2)))
3484             (not (= t1 t2)))
3485          ((&& (meta-id? t1) (meta-id? t2) (negate (equal? t1 t2)))
3486             (not (= t1 t2)))))
3487
3488 (define (show-unequal t1 t2)
3489   (!force (check ((&& (canonical? t1)
3490               (canonical? t2)
3491               (equal? (sort-of t1) (sort-of t2))
3492               (negate (equal? t1 t2)))
3493             (not (= t1 t2)))
3494          ((&& (meta-id? t1) (meta-id? t2) (negate (equal? t1 t2)))
3495             (not (= t1 t2))))))
3496
3497 (define (symbol-lemmas f)
3498   (fetch-all (lambda (p)
```

```
3499                (match p
3500                  ((forall (some-list _) ((val-of f) (some-list _)))           true)
3501                  ((forall (some-list _) (= ((val-of f) (some-list _)) _))     true)
3502                  ((forall (some-list _) (= _ ((val-of f) (some-list _))))     true)
3503                  ((forall (some-list _) (if _  ((val-of f) (some-list _))))   true)
3504                  ((forall (some-list _) (if _  (= ((val-of f) (some-list _)) _))) true)
3505                  ((forall (some-list _) (if _  (= _ ((val-of f) (some-list _))))) true)
3506                  ((forall (some-list _) (iff _ ((val-of f) (some-list _))))   true)
3507                  ((forall (some-list _) (iff   ((val-of f) (some-list _)) _)) true)
3508                  ((forall (some-list _) (iff _  (= ((val-of f) (some-list _)) _))) true)
3509                  ((forall (some-list _) (iff _  (= _ ((val-of f) (some-list _))))) true)
3510                  (_ false)))))
3511
3512  (define (lemmas arg)
3513    (match arg
3514      ((some-symbol _) (remove-duplicates (symbol-lemmas arg)))
3515      ((some-list _)   (remove-duplicates (fold join (map symbol-lemmas arg) [])))))
3516
3517
3518  (define properties lemmas)
3519
3520  (define (list->vector L)
3521    (let ((V (make-vector (length L) ())))
3522      (letrec ((loop (lambda (L i)
3523                       (match L
3524                         ([] V)
3525                         ((list-of x rest) (seq (vector-set! V i x)
3526                                                (loop rest (plus i 1))))))))
3527        (loop L 0))))
3528
3529  (define (vector->list V)
3530   (let ((max (vector-size V)))
3531     (letrec ((loop (lambda (i L)
3532                      (check ((less? i max) (loop (plus i 1) (add (vector-sub V i) L)))
3533                             (else (rev L))))))
3534      (loop 0 []))))
3535
3536
3537  (define (vec-app f vector)
3538    (letrec ((n (vector-size vector))
3539             (loop (lambda (i)
3540                     (check ((less? i n) (seq (f i (vector-sub vector i))
3541                                              (loop (plus i 1))))
3542                            (else ()))))))
3543      (loop 0)))
3544
3545
3546  (define (vector-copy v1 v2)
3547    (vec-app (lambda (i x)
3548               (vector-set! v2 i x)) v1))
3549
3550  (define (vector-swap V i j)
3551    (let ((x (vector-sub V i))
3552          (_ (vector-set! V i (vector-sub V j))))
3553      (vector-set! V j x)))
3554
3555  (define (smallest i j)
3556    (check ((less? i j) i)
3557           (else j)))
3558
3559  (define (greatest i j)
3560    (check ((less? i j) j)
3561           (else i)))
3562
3563  (define (fix f p)
3564    (let ((p' (f p)))
3565      (check ((equal? p' p) p)
3566             (else (fix f p')))))
3567
3568  (define (dfix M p)
```

```
  (dlet ((p' (!M p)))
    (dcheck ((equal? p p') (!claim p))
            (else (!dfix M p')))))


## The following applies a procedure f to a sentence p
## top-down, at most once:

(define (top-down f p)
  (match (f p)
   ((val-of p) (match p
                 ((some-atom _) p)
                 (_ ((root p) (map (lambda (p) (top-down f p))
                                   (children p))))))
   (res res)))

## The following applies a procedure f to a sentence p
## bottom-up, at most once:

(define (bottom-up f p)
  (letrec ((g (lambda (p)
                (match (f p)
                   ((val-of p) [p false])
                   (q [q true])))))
           (f* (lambda (p)
                 (match p
                  ((some-atom _) (g p))
                  (((some-sent-con pc) (some-list args))
                    (let ((args' (map f* args)))
                      (check ((for-some args' (lambda (x) (second x))) [(pc (map first args')) true])
                             (else (g p))))))))))
    (first (f* p))))

(define (abs x)
  (check ((less? x 0) (times (- 1) x))
         (else x)))


(define (raise x n)
  (check ((less? n 1) 1)
         (else (times x (raise x (minus n 1))))))

define raised-to := raise

(define (transform* p q rules methods)
  (dletrec ((limit 100)
            (same-connective? (lambda (p1 p2)
                                (equal? (root p1) (root p2))))
            (distance (lambda (r)
                        [r (plus (check ((same-connective? r q) 0)
                                        (else 1000))
                                 (abs (minus (prop-size r) (prop-size q))))]))
            (loop (method (current i)
                   (dcheck ((greater? i limit) (dlet ((_ (writeln-val i))) (!claim current)))
                           ((equal? current q) (dlet ((_ (writeln-val i))) (!claim current)))
                           (else (dlet  ((results (filter (map (lambda (R) (top-down R current)) rules)
                                                          (lambda (res) (negate (equal? res current)))))
                                         (res-distances  (map distance results))
                                         (pair-min (lambda (x y)
                                                     (check ((less? (second x) (second y)) x)
                                                            (else y))))
                                         (best (min-or-max res-distances (first res-distances)
                                                           pair-min)))
                                   (dcheck ((equal? best current) (dlet ((_ (writeln-val i))) (!claim current)))
                                           (else (!loop (!transform current (first best) methods)
                                                        (plus i 1))))))))))
    (!loop p 0)))

(define t* transform*)

```

```
# (!t* p q [R] [bdn])


# (define p (if (not A) B))
# (define q (or A B))
# (define rules [(lambda (p) (match p ((if p1 p2) (or (not p1) p2)) (_ p)))
#                (lambda (p) (match p ((not (not q)) q) (_ p)))])

# (assert p)

# (!t* p q rules [cond-def bdn])


(define (make-sub bindings)
  (letrec ((loop (lambda (bindings sub)
                   (match bindings
                     ((list-of [(some-var v) (some-term t)] rest) (loop rest (extend-sub sub v t)))
                     (_ sub)))))
     (loop bindings empty-sub)))

(define (id->atom p)
  (dmatch p
   ((= s true) (!by-contradiction s
                 (assume (not s)
                   (!absurd (!(evaluate "chain-last") [true ==> s [p]])
                            (not s)))))
   ((= s false) (!by-contradiction (not s)
                  (!(evaluate "chain") [s ==> false [p]])))))


(define (identity->atom p)
  (dmatch p
   ((= _ true) (!id->atom p))
   ((= true _) (!id->atom (!sym p)))
   ((= _ false) (!id->atom p))
   ((= false _) (!id->atom (!sym p)))))

(define (height t)
  (match t
    (((some-symbol _) (bind args (list-of _ _))) (plus 1 (max* (map height args))))
    (_ 0)))

(set-precedence (+ - plus minus) 200)
(set-precedence (* times / div mod) 300)
(set-precedence = 100)
(set-precedence Cons 280)
#(set-precedence Pair 200)

(define (augment p q)
  (dmatch q
    ((and p1 p2) (!both p1 p2))))

(define (augment p q)
  (!prove-components-harder q))

(define (unequal x y) (not (x = y)))

(set-precedence unequal (get-precedence =))

(define =!= unequal)
(define =/ unequal)
(define /= unequal)

(define =/= unequal)

(define (unequal-to x)
  (lambda (y) (unequal? y x)))

(define (equal-to x)
  (lambda (y) (equal? y x)))
```

```
3709
3710 (define (combined-with M args)
3711   (method (p q) (dlet ((args' (match args
3712                                 ((some-list _) args)
3713                                 (_ [args])))))
3714                   (!find-some (weave p args')
3715                               (method (arg-combo) (conclude q (!app-method M arg-combo)))
3716                               fail))))
3717
3718 (define with combined-with)
3719
3720 (define (canonical-components p get-components)
3721    (sort (get-components p)
3722          (lambda (p1 p2)
3723             (match (compare-strings (val->string p1) (val->string p2))
3724               ('less true)
3725               (_ false)))))
3726
3727 (define (match-props-modulo-CD p q)
3728   (match [p q]
3729     ([(and (some-list _)) (and (some-list _))] (match-props (and* (canonical-components p get-conjuncts)) (and* (canon
3730     ([(or (some-list _)) (or (some-list _))] (match-props (or* (canonical-components p get-disjuncts)) (or* (canonical
3731     (_ (match-props p q))))
3732
3733 ## The above, sort-based definition of match-props-modulo-CD fails
3734 ## on inputs like these (which should match but don't):
3735 ## p = (m N.< n & n N.< n S m)
3736 ## q = (m N.< ?FOO & ?FOO N.< S m)
3737 ## Thus reverting to the old, simpler definition.
3738
3739 (define (match-props-modulo-CD p q)
3740   (match-props p q))
3741
3742 (define (simplify p q)
3743   (!decompose p (method (_) (!claim q))))
3744
3745 (define (complement-conjunction C p)
3746   (!by-contradiction (complement C)
3747     (assume C
3748       (!from-complements false p (!conj-elim (complement p) C)))))
3749
3750 (define (comm-opt p)
3751   (dmatch p
3752     ((and _ _) (!comm p))
3753     ((or _ _) (!comm p))
3754     (_ (!claim p))))
3755
3756 (define (negate-disjunct negated-disjunction complemented-component)
3757   (!by-contradiction complemented-component
3758     (assume hyp := (complement complemented-component)
3759       (dlet ((_ (!comm-opt hyp)))
3760          (!from-complements false negated-disjunction (!disj-intro (complement negated-disjunction)))))))
3761
3762 (define (disjuncts-of p)
3763   (match p
3764     ((or (some-list args)) args)
3765     (_ [p])))
3766
3767 (define (deep-disjuncts p)
3768   (match p
3769     ((or (some-list args)) (flatten (map deep-disjuncts args)))
3770     (_ [p])))
3771
3772 (define (deep-conjuncts p)
3773   (match p
3774     ((and (some-list args)) (flatten (map deep-conjuncts args)))
3775     (_ [p])))
3776
3777 (define (decompose-equation eqn)
3778   (match eqn
```

```
3779      ((forall (some-list uvars) (= left right))         [uvars left right])
3780      ((forall (some-list uvars) (if _ (= left right)))  [uvars left right])
3781      ((forall (some-list uvars) (iff _ (= left right))) [uvars left right])))
3782
3783  (define (match-props-segment L1 L2 uvars)
3784    (let (([n1 n2] [(length L1) (length L2)]))
3785      (check ((leq? n1 n2) (match-props-3 L1 (take L2 n1) uvars))
3786             (else false))))
3787
3788  (define epf eprove-from)
3789
3790  (define es make-e-prop-string)
3791
3792  (define (running-time f n)
3793    (let ((t1 (time))
3794          (_  (repeat f n))
3795          (t2 (time))
3796          (t  (minus t2 t1))
3797          (_ (print "\nTime: " t "\n")))
3798      t))
3799
3800  (define test running-time)
3801
3802
3803  (define (expand-input-unary f converter)
3804    (lambda (x)
3805      (f (converter x))))
3806
3807  (define (expand-input-binary f converter-1 converter-2)
3808    (lambda (x y)
3809      (f (converter-1 x) (converter-2 y))))
3810
3811  (define (transform-output-unary f converter-list)
3812    (lambda (x)
3813      (let ((res (f x)))
3814         (letrec ((loop (lambda (L last-result)
3815                          (match L
3816                           ([] last-result)
3817                           ((list-of converter more) (let ((new (converter last-result)))
3818                                                       (check ((equal? new last-result) (loop more new))
3819                                                              (else new))))))))
3820           (loop converter-list res)))))
3821
3822  (define (self-equal p q)
3823    (dmatch q
3824      ((= _x _x) (!reflex _x))))
3825
3826  (declare (returns --->) Boolean)
3827
3828  (define (function clauses)
3829    (letrec ((process
3830              (lambda (clauses)
3831                (match clauses
3832                  ((split [_lhs returns _rhsides] _rest)
3833                   (match _rhsides
3834                     ([_id _rhs]
3835                      (join [_id (close (sentence _lhs _rhs))] (process _rest)))
3836                     (_ (join (process-conditions _lhs _rhsides [])
3837                              (process _rest)))))
3838                  ([] []))))
3839             (process-conditions
3840              (lambda (lhs rhsides negated-prior-conditions)
3841                (match rhsides
3842                  ((split [_id _condition ---> _rhs] _more)
3843                   (match _condition
3844                     (true [_id (close (if (conjoin
3845                                            (rev negated-prior-conditions))
3846                                          (sentence lhs _rhs)))])
3847                     (_ (join [_id (close (if (conjoin
3848                                               (rev (add _condition
```

```
                                                          negated-prior-conditions)))
                                          (sentence lhs _rhs)))]
                             (process-conditions
                              lhs _more
                              (add (not _condition) negated-prior-conditions))))))
                 ([] [])))))
            (sentence
             (lambda (lhs rhs)
               (match (sort-of lhs)
                 ("Boolean"
                  (match rhs
                    (true lhs)
                    (false (not lhs))
                    (_ (iff lhs rhs))))
                 (_ (= lhs rhs))))))
     (process clauses)))



(define (pick-all-witnesses premise M)
  (dletrec ((loop (method (premise witnesses)
                    (dmatch premise
                      ((exists x p) (pick-witness w premise w-premise
                                      (!loop w-premise (add w witnesses))))
                      (_ (!M (rev witnesses) premise))))))
    (!loop premise [])))

(set-precedence close 5)


(define (lower-case-alpha-char? c)
  (&& (member? (compare-chars c `a) ['greater 'equal])
      (member? (compare-chars c `z) ['less 'equal])))

(define (upcase c)
  (check ((lower-case-alpha-char? c) (char (minus (char-ord c) 32)))
         (else c)))

(define (upcase-string str)
  (map upcase str))

(define (alpha-char? c)
  (|| (upper-case-alpha-char? c) (lower-case-alpha-char? c)))

(define (numeric-char? c)
  (&& (member? (compare-chars c `0) ['greater 'equal])
      (member? (compare-chars c `9) ['less 'equal])))

(define (alpha-numeric-char? c)
  (|| (alpha-char? c) (numeric-char? c)))

(define digit? numeric-char?)

(define (all-digits? str)
  (for-each str digit?))

(define (alpha-numeric-string? str)
  (for-each str alpha-numeric-char?))

(define (delete-files files)
  (seq (map delete-file files) ()))

(define (datatype-sort? str)
  (negate (equal? (constructors-of str) [])))

(define (printable-char? c)
  (&& (less? (char-ord c) 128)
      (greater? (char-ord c) 32)))

(define (printable-string? str)
```

```
3919   (for-each str printable-char?))
3920
3921 (define (selector? f)
3922   (try (match (all-but-last (get-signature f))
3923          ([arg-sort] (check ((datatype-sort? arg-sort)
3924                              (let ((f-name (symbol->string f))
3925                                    (constructors (constructors-of arg-sort)))
3926                                (for-some constructors
3927                                  (lambda (c)
3928                                     (for-some (selector-names c)
3929                                       (lambda (sel-name)
3930                                         (equal? f-name sel-name)))))))
3931                             (_ false))))
3932        false))
3933
3934
3935 (define (even? n)
3936   (equal? (mod n 2) 0))
3937
3938 (define (odd? n)
3939   (negate (even? n)))
3940
3941 (define (flip-coin)
3942   (equal? (random-int 2) 1))
3943
3944 (define (make-random-int)
3945   (let ((i (random-int max-int)))
3946     (check ((flip-coin) i)
3947            (else (- i)))))
3948
3949 (define (make-random-integer-valued-real)
3950   (let ((r (plus (random-int max-int) 0.0)))
3951     (check ((flip-coin) r)
3952            (else (- r)))))
3953
3954 (define (constructors? L)
3955   (for-each L constructor?))
3956
3957
3958
3959
3960 (define (size' p)
3961   (letrec ((loop (lambda (p sum)
3962                    (match p
3963                      ((some-atom t) (plus sum (term-size t)))
3964                      ((((|| not and or if iff) (some-list args))
3965                        (loop* args (plus sum 1)))
3966                      (((some-quant _) (some-var _) (some-sent q))
3967                        (loop q (plus 2 sum))))))
3968            (loop* (lambda (props sum)
3969                     (match props
3970                       ([] sum)
3971                       ((list-of p more) (loop* more (loop p sum)))))))
3972     (loop p 0)))
3973
3974 (define (alpha-variants? t1 t2)
3975   (let ((sub1 (match-terms t1 t2))
3976         (sub2 (match-terms t2 t1)))
3977     (check ((|| (equal? sub1 false) (equal? sub2 false)) false)
3978            (else sub1))))
3979
3980
3981 ###========================== NEW CODE FOR AUTO EVAL ==========================
3982
3983 (define empty-map [])
3984
3985 (define (add-binding x y map)
3986   (add [x y] map))
3987
3988 (define (extend map pairs)
```

```
3989    (match pairs
3990      ([] map)
3991      ((list-of [x y] more) (extend (add-binding x y map) more))))
3992
3993  (define (remove-binding x map)
3994    (match map
3995      ([] [])
3996      ((list-of [key value] more) (check ((equal? key x) more)
3997                                         (else (add-binding key value (remove-binding x more)))))))
3998
3999  (define (apply-map map x)
4000    (match map
4001      ([] ())
4002      ((list-of [a b] more) (check ((equal? a x) b)
4003                                   (else (apply-map more x))))))
4004
4005  (define (dom m)
4006    (letrec ((loop (lambda (m res)
4007                     (match m
4008                       ([] res)
4009                       ((list-of [a b] more) (check ((member? a res) (loop more res))
4010                                                    (else (loop more (add a res)))))))))
4011      (loop m [])))
4012
4013
4014  (define (map-range m)
4015    (map (lambda (x) (apply-map m x))
4016         (dom m)))
4017
4018  (define (dom-range-list m)
4019    (map (lambda (x) [x (apply-map m x)])
4020         (dom m)))
4021
4022
4023  (define (in-dom? a m)
4024    (match m
4025      ([] false)
4026      ((list-of [x _] rest) (|| (equal? a x)
4027                                (in-dom? a rest)))))
4028
4029
4030  (define [bar comma lp rp lb rb blank colon scolon quot-mark]
4031          [" | " "," "(" ")" "[" "]" " " ":" ";" "\""])
4032
4033  (define newline "\n")
4034
4035  (define tab "  ")
4036
4037  (define (conjuncts-of p)
4038    (add p (get-conjuncts-recursive p)))
4039
4040  (define (rename-term t)
4041    (let ((fv (vars t))
4042          (fv' (map (lambda (v)
4043                      (fresh-var (sort-of v))) fv)))
4044      (letrec ((loop (lambda (fvars t)
4045                       (match fvars
4046                         ([] t)
4047                         ((list-of [v v'] more) (loop more (replace-var v v' t)))))))
4048        (loop (zip fv fv') t))))
4049
4050  (set-precedence And 30)
4051  (set-precedence Or 20)
4052  (set-precedence If 10)
4053  (set-precedence Iff 10)
4054  (set-precedence Not 50)
4055
4056  (define (simp-or L)
4057     (letrec ((loop (lambda (L res)
4058                      (match L
```

```
                      ([] res)
                      ((list-of true _) [true])
                      ((list-of false more) (loop more res))
                      ((list-of p more) (loop more (add p res)))))))
     (match (loop L [])
       ([] false)
       ([p] p)
       (L' (or L')))))

(define (simp-and L)
   (letrec ((loop (lambda (L res)
                    (match L
                      ([] res)
                      ((list-of false _) [false])
                      ((list-of true more) (loop more res))
                      ((list-of p more) (loop more (add p res)))))))
     (match (loop L [])
       ([] true)
       ([p] p)
       (L' (and L')))))


(define (normalize p)
  (match p
    ((or (some-list _))  (or (rev (rd deep-disjuncts p))))
    ((and (some-list _)) (and (rev (rd deep-conjuncts p))))
    ((forall (as uvars (list-of _ _)) body) (forall* uvars (normalize body)))
    (_ p)))


(define (sole-constructor? f t)
   (match (constructors-of (sort-of t))
     ([(val-of f)] true)
     (_ false)))

(define (one-of-each? f g)
  (|| (&& (constructor? f) (selector? g))
      (&& (selector? f) (constructor? g))))

(define (diff1 s t)
  (match [s t]
    ([(some-var _) (f (some-list args))] (check ((sole-constructor? f t)
                                                 (check ((for-each args var?) false)
                                                        (else (try (let ((selectors (map string->symbol (selector-nam
                                                                                      (left-term (make-term f (map (lambda (sel)
                                                                                   (diff1 left-term t))
                                                                                 false))))
                                                 (else (not (= s t)))))
    ([((some-symbol f) (some-list args1)) ((some-symbol g) (some-list args2))]
       (check ((for-each [f g] (lambda (c) (|| (constructor? c) (selector? c))))
                (let ((_ ())
                      ##(_ (print "\nHere is f: " f " and here is g: " g "\n"))
                      (_ ()))
                (check ((one-of-each? f g) (not (= s t)))
                       ((negate (equal? f g)) true)
                       (else (simp-or (map (lambda (pair)
                                             (match pair
                                               ([s' t'] (diff1 s' t'))))
                                           (zip args1 args2)))))))
              (else (error (join "\nNon-constructors found in a function definition clause: " (val->string f) " and "
    (_ false)))

(define (diff lhs lhs')
  (letrec ((loop (lambda (L1 L2 res)
                   (match [L1 L2]
                     ([[] []] (rev res))
                     ([(list-of s1 rest1) (list-of s1' rest1')]
                        (loop rest1 rest1' (add (diff1 s1 s1') res)))))))
    (match [lhs lhs']
      ([(f (some-list args)) (f (some-list args'))]
```

```
4129          (normalize (simp-or (loop args args' [])))))))
4130
4131  (define (diff* lhs lh-list)
4132     (normalize (simp-and (map (lambda (lhs')
4133                                 (diff lhs (rename-term lhs')))
4134                               lh-list))))
4135
4136
4137  (define (filter-out L pred)
4138    (filter L (lambda (x) (negate (pred x)))))
4139
4140  (define (specify terms)
4141    (method (P) (!instance P terms)))
4142
4143  (define (file-path names)
4144    (match (try (foldr make-path "" names) ())
4145      (() (error (join "\nUnable to make a path from these:\n" (separate names ", "))))
4146      (res res)))
4147
4148 define ATHENA_LIB := (file-path [ATHENA_HOME "lib" "basic"])
4149
4150  (define (&&* L) (for-each L (lambda (x) x)))
4151
4152  (define (||* L) (for-some L (lambda (x) x)))
4153
4154  (define (&&R L)
4155    (match L
4156      ([] true)
4157      ((list-of x rest) (match x
4158                          (true (&&R rest))
4159                          (false false)
4160                          (_ (and* L))))))
4161
4162  (define (||R L)
4163    (match L
4164      ([] false)
4165      ((list-of x rest) (match x
4166                          (true true)
4167                          (false (||R rest))
4168                          (_ (or* L))))))
4169
4170  (define (arg-sorts c)
4171    (all-but-last (get-signature c)))
4172
4173  (define (arg-sorts-unified c sort)
4174    (all-but-last (get-signature-unified c sort)))
4175
4176  (define (constructor-range c)
4177    (match (last (get-signature c))
4178      ((split "(" struc-name " " _) struc-name)
4179      (struc-name struc-name)))
4180
4181  (define (reflexive? c)
4182            (takes-args-of-sort c (last (get-signature c)))
4183          (takes-args-of-sort c S)
4184            (let ((sig (get-signature c))
4185                  (range-sort (last sig))
4186                  (arg-sorts (all-but-last sig)))
4187              (|| (for-some arg-sorts (lambda (T) (unifiable-sorts? S T)))
4188                  (for-some (filter-out arg-sorts (lambda (asort) (equal? asort range-sort)))
4189                            (lambda (asort)
4190                              (for-some (constructors-of asort)
4191                                        (lambda (c')
4192                                         (takes-args-of-sort c' S))))))))
4193
4194
4195  (define [reflexive-unif? takes-args-of-sort]
4196    (let (([ht1 ht2] [(table 31) (table 31)])
4197          ([arg-ht1 arg-ht2] [(table 31) (table 31)]))
4198      (letrec ((ref-u (lambda (c sort)
```

```
                        (let ((arg [c sort])
                              (already-called? (try (table-lookup arg-ht1 arg) false)))
                          (check (already-called?
                                   (let ((memoized-result (try (table-lookup ht1 arg) ())))
                                     (match memoized-result
                                       (() false)
                                       (_  memoized-result))))
                                 (else  (let ((_ (table-add arg-ht1 [arg --> true]))
                                              (result (accepts-args-of-sort c (last (get-signature-unified c sort))))
                                              (_ (table-add ht1 [arg --> result])))
                                          result))))))
            (accepts-args-of-sort
               (lambda (c S)
                  (let ((arg [c S])
                        (already-called? (try (table-lookup arg-ht2 arg) false)))
                    (check (already-called?
                             (let ((memoized-result (try (table-lookup ht2 arg) ())))
                               (match memoized-result
                                 (() false)
                                 (_  memoized-result))))
                           (else  (let ((_ (table-add arg-ht2 [arg --> true]))
                                        (result (try
                                                  (let ((sig (get-signature-unified c S))
                                                        (range-sort (last sig))
                                                        (arg-sorts (all-but-last sig)))
                                                    (|| (for-some arg-sorts (lambda (T) (unifiable-sorts? S T)))
                                                        (for-some (filter-out arg-sorts (lambda (asort) (equal? asort
                                                                  (lambda (asort)
                                                                    (for-some
                                                                      (constructors-of asort)
                                                                      (lambda (c')
                                                                        (&& (unequal? c' c) (accepts-args-of-sort c' S
                                                  false))
                                        (_ (table-add ht2 [arg --> result])))
                                      result)))))))
       [ref-u accepts-args-of-sort])))

(define (irreflexive? c)
  (negate (reflexive? c)))

(define (irreflexive-unif? c sort)
  (negate (reflexive-unif? c sort)))

(define (reflexive-constructors-of dt)
  (filter (constructors-of dt) (lambda (c) (reflexive-unif? c dt))))

(define (irreflexive-constructors-of dt)
  (filter (constructors-of dt) (lambda (c) (irreflexive-unif? c dt))))

(define (random-shuffle L)
  (let (([e-first bit-o bit-e] [(flip-coin) (flip-coin) (flip-coin)])
        ([odds evens] [(odd-positions L) (even-positions L)])
        (odds' (check (bit-o (rev odds))
                      (else odds)))
        (evens' (check (bit-e (rev evens))
                       (else evens))))
    (check (e-first (join evens' odds'))
           (else (join odds' evens')))))


(define (choose L)
  (nth (random-int (length L)) L))

(define (choose-and-remove L)
  (let ((x (choose L)))
    [x (list-remove x L)]))

(define (choose-without-reps k L)
  (letrec ((loop (lambda (k L results)
                   (check ((less? k 1) [results L])
```

```
                          (else (let (([x L'] (choose-and-remove L)))
                                  (loop (minus k 1) L' (add x results))))))))
    (loop k L [])))

(define (choose-subset L k)
  (first (choose-without-reps k L)))

(define (starify connective)
  (lambda (terms)
     (letrec ((loop (lambda (terms)
                      (match terms
                        ([t] t)
                        ((list-of t more) (connective t (loop more)))))))
       (loop terms))))

(define And* (starify And))

(define Or* (starify Or))

(define (sent->term p)
  (match p
    ((and (some-list args)) (And* (map sent->term args)))
    ((or (some-list args)) (Or* (map sent->term args)))
    ((if p1 p2) (If (sent->term p1) (sent->term p2)))
    ((iff p1 p2) (let ((p1' (sent->term p1))
                       (p2' (sent->term p2)))
                   (And (If p1' p2') (If p2' p1'))))
    ((not p) (Not (sent->term p)))
    (_ p)))

()

expand-input And [sent->term sent->term]
expand-input Or [sent->term sent->term]
expand-input If [sent->term sent->term]
expand-input Iff [sent->term sent->term]
expand-input ite [sent->term sent->term sent->term]
expand-input Not [sent->term]


define distinct-counter := (cell 0)

(define (all-distinct-functor terms)
  (match terms
    ([] [])
    ((list-of t _)
       (let ((S (sort-of t))
             (new-fsym-name (join "distinct-functor-" (val->string (inc distinct-counter))))
             (command (join "(declare " new-fsym-name " (-> (" S ") Int))"))
             (_ (process-input-from-string command))
             (new-fsym (string->symbol new-fsym-name))
             (counter (cell 0)))
         (map (lambda (t) (= (new-fsym t) (inc counter))) terms)))))

(define (ground p)
  (match p
    (((some-quant q) (some-var v) body)
       (let ((S (sort-of v)))
         (check ((datatype-sort? S)
                   (let ((C (constructors-of S)))
                     (match q
                       (forall (and (map (lambda (c) (replace-var v c (ground body))) C)))
                       (_ (or (map (lambda (c) (replace-var v c (ground body))) C))))))
                (else p))))
    (((some-sent-con sc) (some-list args)) (sc (map ground args)))
    (_ p)))
```

```
(define (split-string str ch)
    (letrec ((loop (lambda (str current results)
                     (match str
                       ([] (rev (add (rev current) results)))
                       ((list-of c more) (check ((equal? c ch) (loop more [] (add (rev current) results)))
                                                (else (loop more (add c current) results))))))))
      (loop str [] [])))


(set-precedence join 105)
define joined-with := join

define mapped-to := map

(set-precedence equal? 100)

(define (from-negation left right)
  (!by-contradiction right
      (assume (not right)
         (!absurd left (not left)))))


(define (get-defined-prop p)
  (match p
    ((forall (some-list uvars) (iff (= (some-term left)
                                       (some-var x))
                                    (= x (some-term right))))
       (let ((uvars' (list-remove x uvars)))
         (forall* uvars' (= left right))))
    (_ p)))


(define make-term' make-term)

(define (make-term f args)
  (match args
    ([] f)
    (_ (make-term' f args))))

(define (try-looking-up x ht)
  (try [(table-lookup ht x)]
       ()))

(define (memoize-unary f)
  (let ((ht (table 101)))
    (lambda (x)
       (match (try-looking-up x ht)
         ([y] y)
         (_ (let ((y (f x))
                  (_ (table-add ht [x --> y])))
              y))))))

(define (memoize-unary f) f)

(define (memoize-binary f)
  (let ((ht (table 101)))
    (lambda (x y)
       (match (try-looking-up [x y] ht)
         ([z] z)
         (_ (let ((z (f x y))
                  (_ (table-add ht [[x y] --> z])))
              z))))))


(define (memoize-ternary f)
  (let ((ht (table 101)))
    (lambda (x y z)
       (match (try-looking-up [x y z] ht)
         ([r] r)
```

```
          (_ (let ((r (f x y z))
                   (_ (table-add ht [[x y z] --> r])))
               r))))))


(define (memoize-ternary f) f)



(define (string? L)
  (match L
    ((some-list _) (for-each L char?))
    (_ false)))

(define (log2 x)
  (check ((leq? x 0.0) (- 1.0))
         (else (div (log10 x) (log10 2)))))

(define (float x) (times 1.0 x))


(define (find-min L compare)
  (letrec ((loop (lambda (L min)
                   (match L
                     ([] min)
                     ((list-of x more) (check ((compare x min) (loop more x))
                                              (else (loop more min))))))))
    (match L
      ((list-of x rest) (loop rest x)))))



(define (get-remaining-patterns pats)
  (list-diff (get-all-remaining-patterns pats) pats))

(define (contains-quants? p)
  (match p
    (((some-quant q) (some-list _) _) true)
    (((some-sent-con _) (some-list args)) (for-some args contains-quants?))
    (_ false)))

define and-conv :=
  method (P)
    match P {
      (and P1 P2 P3) =>
         (!both (!left-and P) (!both (!left-and (!right-and P))
                                     (!right-and (!right-and P))))
    | (and P1 P2 P3 P4) =>
         (!both (!left-and P) (!and-conv (and P2 P3 P4)))
    }



(define (sderive s props)
    (!sprove-from s props [['poly true] ['subsorting false] ['max-time 1000]]))

(define (vderive s props)
    (!vprove-from s props [['poly true] ['subsorting false] ['max-time 1000]]))

(define fsd0 fsd)

(define (fsd f)
  (match f
    ((some-proc _) (match (fsd0 (string->symbol (proc-name f)))
                     (() (seq #(print "\nNo info for this function symbol: " (string->symbol (proc-name f)))
                              ()))
                     (res res)))
    (_ (match (fsd0 f)
         (() (seq  #(print "\nNo info for this function symbol: " (val->string f))
                   ())))
```

```
4479          (res res)))))
4480
4481  (define (code f)
4482    (try
4483      (print ((fsd f) 'code))
4484      (print (join "\nNo code for " (val->string f) ".\n"))))
4485
4486  (define (dcode f)
4487    (print ((fsd f) 'deduction-code)))
4488
4489  (define (red-code f)
4490    (print ((fsd f) 'red-code)))
4491
4492  (define (needed-by f)
4493    ((fsd f) 'needed-by-syms))
4494
4495  (define (get-obsolete-axioms f)
4496    ((fsd f) 'obsolete-axioms))
4497
4498  (define (get-bicond-sources f)
4499    ((fsd f) 'bicond-axiom-sources))
4500
4501  (define (occurring f)
4502    ((fsd f) 'occurring-syms))
4503
4504  (define (guard-syms f)
4505    ((fsd f) 'guard-syms))
4506
4507  (define (eqns f)
4508    ((fsd f) 'defining-equations))
4509
4510  (define (ysolve p)
4511   (let ((ht (table 10))
4512         (_ (table-add ht ['solver --> 'yices])))
4513     (match p
4514       ((some-sent _) (let ((q (check ((poly? p) (make-monomorphic-instance p)) (else p))))
4515                        (smt-solve q ht)))
4516      ((some-list L) (let ((q (and (map (lambda (p) (check ((poly? p) (make-monomorphic-instance p)) (else p))) L))))
4517                       (smt-solve q ht))))))
4518
4519
4520  (define (ground0 p N)
4521    (letrec ((loop (lambda (p)
4522                     (match p
4523                       (((some-quant q) (some-var v) body)
4524                          (let ((S (sort-of v)))
4525                            (check ((datatype-sort? S)
4526                                       (let ((C (constructors-of S)))
4527                                          (match q
4528                                           (forall (and (map (lambda (c) (replace-var v c (loop body))) C)))
4529                                           (_ (or (map (lambda (c) (replace-var v c (loop body))) C))))))
4530                                   ((equal? S "Int")
4531                                      (match q
4532                                         (forall (and (map (lambda (c) (replace-var v c (loop body))) (from-to 0 N))))
4533                                         (exists (or (map (lambda (c) (replace-var v c (loop body))) (from-to 0 N))))))
4534                                   (else p))))
4535                       (((some-sent-con sc) (some-list args)) (sc (map loop args)))
4536                       (_ p)))))
4537      (loop p)))
4538
4539
4540  (define defining-axioms0 defining-axioms)
4541
4542  (define (defining-axioms f)
4543   (try
4544    (match f
4545      ((some-symbol _) (defining-axioms0 f))
4546      ((some-proc _) (defining-axioms0 (root (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 (arity-of f)))))))
4547      (_ (defining-axioms0 f))) []))
4548
```

```
(define (sholds? p)
  (print "\nDoes\n" p "\nhold?: " (holds? p)))

(define (shold? props)
  (map-proc sholds? props))

define when := 'when

define (define-axiom L name) :=
  let {new-axiom := match L {
                      [(some-sent p)] => p
                    | (some-list L) => L
                    };
       cmd := (join "(define " (id->string name) " " (val->string new-axiom) ")")}
    (process-input-from-string cmd)

define (fun-clause left op right) :=
    letrec {var := (fresh-var);
            aux := lambda (right)
                     match right {
                       (split [t 'when c ((some-term name) where (meta-id? name))] rest) =>
                            let {res := (if c (op var t));
                                    _ := (define-axiom [res] name)}
                              (add res (aux rest))
                     | (split [t 'when c] rest) =>
                          (add (if c (op var t)) (aux rest))
                     | [] => []
                     | (some-list _) => (error (join "\nIll-formed fun sub-clause(s):" (val->string right) "\n"))
                     }}
    match right {
      (some-list r) =>
        (close (map lambda (p) (urep (forall var p) [left])
                    (aux r)))
    | _ => [(close (op left right))]
    }


define (fun L) :=
    match L {
      (split [left = right] [((some-term name) where (meta-id? name))] rest) =>
           let {L := (fun-clause left = right);
                _ := (define-axiom L name)}
             (join L (fun rest))
    | (split [left = right] rest) => (join (fun-clause left = right) (fun rest))
    | (split [left <==> right] [((some-term name) where (meta-id? name))] rest) =>
           let {L := (fun-clause left <==> right);
                _ := (define-axiom L name)}
             (join L (fun rest))
    | (split [left <==> right] rest) => (join (fun-clause left <==> right) (fun rest))
    | [] => []
    | (some-list _) => (error (join "\nIll-formed fun call:" (val->string L) "\n"))
    }

define (overload-binary f1 f2) :=
  lambda (x y)
    try { (f2 x y) | (f1 x y) }

(define (sort-starts-with t str)
   (equal? str (first (tokenize (sort-of t) "() "))))

(define (from-list structure-name pre-process)
  (let (#(_ (print "\nInside from-list, given structure-name: " structure-name))
        ([c1 c2] (constructors-of structure-name))
        ([i r] (check ((less? (arity-of c1) 1) [c1 c2]) (else [c2 c1]))))
    (letrec ((loop (lambda (f)
                     (lambda (L)
                       (match L
                        ([] i)
                        ((list-of x (some-list rest))
                          (r (f x) ((loop f) rest)))
```

```
4619                          (_ L))))))
4620        (lambda (f) (lambda (L) ((loop f) (pre-process L)))))))
4621
4622  (define (to-list structure-name post-process)
4623    (let (([c1 c2] (constructors-of structure-name))
4624          ([i r] (check ((less? (arity-of c1) 1) [c1 c2]) (else [c2 c1])))
4625          (i-name (symbol->string i)))
4626      (letrec ((loop (lambda (f)
4627                       (lambda (t)
4628                         (check ((equal? i-name (symbol->string (root t))) [])
4629                                ((equal? (get-symbol (root t)) r)
4630                                    (let (([x t'] (children t)))
4631                                      (add (f x) ((loop f) t'))))
4632                                (else t))))))
4633         (lambda (f) (lambda (t) (try (post-process ((loop f) t)) t))))))
4634
4635  (define (mod-ht-lookup mod-ht-pair names)
4636    (let (([val-ht mod-ht] mod-ht-pair))
4637      (match names
4638          ([name] (table-lookup val-ht name))
4639          ((list-of name rest) (mod-ht-lookup (table-lookup mod-ht name) rest)))))
4640
4641  (define (module->proc M)
4642      (lambda (x)
4643        (let ((str (check ((meta-id? x) (id->string x))
4644                          (else x)))
4645              (name-parts (tokenize str "."))
4646              (mod-ht-pair (module->table M)))
4647          (mod-ht-lookup mod-ht-pair name-parts))))
4648
4649
4650  (define (hashable? v)
4651    (|| (term? v) (prop? v) (char? v) (string? v) (unit? v) (symbol? v)
4652          (match v
4653            ((some-list _) (for-each v hashable?))
4654            (_ false))))
4655
4656  (define (get-mod-size mod-ht-pair mod-path)
4657    (match mod-ht-pair
4658      ([val-ht mod-ht]
4659         (match mod-path
4660            ([M] (match (table-lookup mod-ht M)
4661                   ([val-ht' mod-ht'] (plus (table-size val-ht') (table-size mod-ht')))))
4662            ((list-of M rest) (get-mod-size (table-lookup mod-ht M) rest))))))
4663
4664  (define inverted-index-structure [(cell ()) (table 100)])
4665
4666  (define (build-inverted-index)
4667    (let ((inverted-index (second inverted-index-structure)))
4668       (letrec ((loop (lambda (mod-ht-pair mod-path)
4669                        (match mod-ht-pair
4670                          ([val-ht mod-ht]
4671                             (seq (map-proc (lambda (p)
4672                                              (match p
4673                                                ([name val] (check ((hashable? val)
4674                                                                    (let ((whole-name (separate (join mod-path [name])
4675                                                                      (try (let ((existing-names (table-lookup inverte
4676                                                                                   (table-add inverted-index [val --> (add w
4677                                                                                 (table-add inverted-index [val --> [whole-na
4678                                                                           (else ())))))
4679                                            (table->list val-ht))
4680                                       (map-proc (lambda (p)
4681                                                   (match p
4682                                                     ([submodule-name contents] (loop contents (join mod-path [submodule-nam
4683                                                   (table->list mod-ht))))))))
4684           (let ((top-mod-ht-pair (module->table "Top"))
4685                 (_ (set! (first inverted-index-structure) top-mod-ht-pair)))
4686             (loop top-mod-ht-pair []))))))
4687
4688  (define (reverse-lookup v)
```

```
4689    (try (rd (table-lookup (second inverted-index-structure) v)) []))
4690
4691  (define module-ab0 module-ab)
4692
4693  (define (module-ab-aux M props)
4694    (let ((_ (match (ref (first inverted-index-structure))
4695                    (() (build-inverted-index))
4696                    (ht-pair (let ((size-1 (module-size M))
4697                                   (size-2 (get-mod-size ht-pair (tokenize M "."))))
4698                               (check ((unequal? size-1 size-2) (build-inverted-index))
4699                                      (else ()))))))))
4700     (map (lambda (p)
4701            (let ((names (dedup (reverse-lookup p)))
4702                  (kind (check ((assertion? p) 'AXIOM) (else 'THEOREM)))
4703                  (result (make-map [['names names] ['kind kind] ['sentence p]]))
4704                  (_ ()))
4705              result))
4706          props)))
4707
4708  (define (module-theory M)
4709     (module-ab-aux M (dedup (module-ab0 M))))
4710
4711  (define (module-theory* M)
4712    (letrec ((loop (lambda (M)
4713                     (join (module-ab0 M)
4714                           (flatten (map (lambda (mname) (loop (join M "." mname)))
4715                                         (sub-modules M))))))))
4716      (module-ab-aux M (dedup (loop M)))))
4717
4718  (load "graph-draw")
4719
4720  (define (draw-theory M)
4721    (let ((L (module-theory M))
4722          (props (map (lambda (record)
4723                        (record 'sentence))
4724                      L))
4725           (G (Graph-Draw.make-graph 0))
4726           (_ (build-inverted-index))
4727           (name (lambda (p)
4728                   (match (reverse-lookup p)
4729                     ([] p)
4730                     (L (first (rev L))))))
4731           (counter (cell 0))
4732           (_ (map-proc (lambda (p)
4733                          (let ((premises (match (dependencies p)
4734                                                 (() [])
4735                                                 (res res))))
4736                            (map-proc (lambda (d)
4737                                        (Graph-Draw.add-edge G (name p) (name d) (inc counter)))
4738                                      premises)))
4739                        props)))
4740      (Graph-Draw.draw-and-show G Graph-Draw.viewer)))
4741
4742  (define (literal? x)
4743    (match x
4744      ((some-atom _) true)
4745      ((not (some-atom _)) true)
4746      (_ false)))
4747
4748
4749  (define (negation-body q) (match q ((not (some-sent body)) body)))
4750
4751
4752  (define (map-remove* m keys)
4753    (letrec ((loop (lambda (m keys)
4754                     (match keys
4755                       ([] m)
4756                       ((list-of k rest) (loop (map-remove m k) rest))))))
4757      (loop m keys)))
4758
```

```
(define (cnf p) (cnf-core p 'dimacs-list))

(define (equiv? p q)
  (try (seq (!vprove-from (iff p q) []) true) false))

define (apply-top-down f) :=
  lambda (p)
    match (f p) {
     (some-sent q) => q
    | _ => match p {
             ((some-sent-con sc) (some-list args)) =>
                (sc (map (apply-top-down f) args))
           | _ => p
           }
    }

(define [tc0 check-fun-def0] [tc check-fun-def])

(define (tc f)
  (tc0 (get-symbol f)))

(define (check-fun-def f)
  (check-fun-def0 (get-symbol f) (defining-axioms f)))


(define (check-fun-defs fsyms)
  (map-proc check-fun-def fsyms))

(define (rearrange p)
                               (let ((move-to-front (lambda (L pred)
                                                                       (letrec ((loop (lambda (L yes no)
                                                                 (match L
                                                                    ([] (join (rev yes) (rev no)))
                                                                    ((list-of x more) (check ((pred x) (loop more (add
                                                                                            (else (loop more yes (add
                                                  (loop L [] [])))))
                         (match p
                            ((forall (some-list uvars) body)
                                (let ((uvars' (move-to-front uvars (lambda (x) (datatype-sort? (sort-of x))))))
                                  (forall* uvars' body)))
                            (_ p))))


(define (induction0* p)
   (dmatch p
     (((forall (some-var x) (some-sent body)) where (datatype-sort? (sort-of x)))
        (dlet ((method-name (join (downcase-string (first (tokenize (sort-of x) " ()"))) "-induction-with"))
               (M (evaluate method-name)))
          (!M p induction0*)))
     (_ (!vprove-from p (ab) [['poly true] ['subsorting false] ['max-time 300]]))))

(define (induction* p)
  (dlet ((q (rearrange p))
         (_ (!induction0* q))
         (cond (!vpf (if q p) [])))
    (!mp cond q)))

(define (induction-from0* p premises)
   (dmatch p
     (((forall (some-var x) (some-sent body)) where (datatype-sort? (sort-of x)))
        (dlet ((method-name (join (downcase-string (first (tokenize (sort-of x) " ()"))) "-induction-with"))
               (M (evaluate method-name)))
          (!M p (method (g) (!induction-from0* g premises)))))
     (_ (!sprove-from p premises [['poly true] ['subsorting false] ['max-time 4000]]))))

(define (induction-from* p premises)
  (dlet ((q (rearrange p))
         (_ (!induction-from0* q premises))
         (cond (!spf (if q p) [])))
    (!mp cond q)))
```

```
4829
4830 (define (induction-with p atp-method)
4831    (dmatch p
4832      (((forall (some-var x) (some-sent body)) where (datatype-sort? (sort-of x)))
4833         (dlet ((method-name (join (downcase-string (first (tokenize (sort-of x) " ()"))) "-induction-with"))
4834                (M (evaluate method-name)))
4835           (!M p (method (g) (!induction-with g atp-method)))))
4836      (_ (!atp-method p))))
4837
4838 (define (induction*-with p atp-method)
4839   (dlet ((q (rearrange p))
4840          (_ (!induction-with q atp-method))
4841          (cond  (!sprove-from (if q p) [] [['poly true] ['subsorting false] ['max-time 4000]])))
4842     (!mp cond q)))
4843
4844 (define (ite* L)
4845   (letrec ((loop (lambda (L)
4846                    (match L
4847                       ((split [(some-term s) --> (some-term t) or] more)
4848                          (ite s t (loop more)))
4849                       ((|| [_ --> (some-term t)]
4850                            [(some-term t)])        t)))))
4851     (loop L)))
4852
4853 (define (module-results module-name)
4854   (map (lambda (r) (r 'sentence))
4855        (module-theory* module-name)))
4856
4857 define (theory-results module-name) :=
4858    (map (lambda (r) (r 'sentence)) (module-theory module-name))
4859
4860 define (test-eval x y) :=
4861   (check ((equal? x y) (print "\nEval worked...\n"))
4862          (else (print "\nEVAL FAILED!\n")))
4863
4864
4865 (define (get-horn-clauses p)
4866   (match p
4867     ((forall (some-list _) (body as (if _ _))) [body])
4868     ((forall (some-list uvars) (iff (some-sent ant) (some-sent con)))
4869          (join (get-horn-clauses (forall* uvars (if ant con)))
4870                (get-horn-clauses (forall* uvars (if con ant)))))
4871     ((forall (some-list _) (some-atom _)) [])
4872     ((forall (some-list _) (not (some-atom _)))  [])
4873     ((and (some-list _)) [])
4874     (_ [])))
4875
4876
4877 define [x y z w x' y' z' w'] := [?x ?y ?z ?w ?x' ?y' ?z' ?w']
4878
4879
4880 define (sub-sentences p) :=
4881   match p {
4882     (some-atom _) => [p]
4883   | ((some-sent-con _) (some-list args)) => (add p (flatten (map sub-sentences args)))
4884   | ((some-quant _) (some-list _) body) => (add p (sub-sentences body))
4885    }
4886
4887
4888 (define (idf f max-depth max-branches)
4889   (letrec ((loop (lambda (i)
4890                    (check ((less? i max-depth) (try (f i max-branches)
4891                                                     (let ((_ (print "\nIncreading depth to: " (plus i 1) "\n"))) (loop
4892                            (else (print "\nOut of iterations, exiting idf...\n")))))))
4893     (loop 1)))
4894
4895
4896
4897
4898
```

```
4899 module Polarities {
4900
4901 define (flip pol) :=
4902   match pol {
4903     'p => 'n
4904   | 'n => 'p
4905   | 'pn => 'pn}
4906
4907
4908 define (polarities p q) :=
4909   let {prepend-and-process :=
4910          lambda (i f)
4911            lambda (pos-pol-pair)
4912              match pos-pol-pair {
4913                [pos pol] => [(add i pos) (f pol)]
4914              };
4915        id := lambda (x) x;
4916        make-pos-neg := lambda (_) 'pn}
4917     match q {
4918       (val-of p) => [[[] 'p]]
4919     | (~ q1) => (map (prepend-and-process 1 flip)
4920                      (polarities p q1))
4921     | (q1 ==> q2) =>  (join (map (prepend-and-process 1 flip)
4922                                  (polarities p q1))
4923                             (map (prepend-and-process 2 id)
4924                                  (polarities p q2)))
4925     | (q1 <==> q2) => (join (map (prepend-and-process 1 make-pos-neg)
4926                                  (polarities p q1))
4927                             (map (prepend-and-process 2 make-pos-neg)
4928                                  (polarities p q2)))
4929     | ((some-sent-con _) (some-list args)) =>
4930         let {i := (cell 1)}
4931           (flatten (map lambda (q)
4932                         (map (prepend-and-process (inc i) id)
4933                              (polarities p q))
4934                         args))
4935      | _ => []
4936      }
4937
4938
4939 #(polarities (A ==> (C | D | A | B)) (A ==> (C | D | A | B)))
4940
4941 } # close module Polarities
4942
4943
4944 define (quant-dist premise) :=
4945   match premise {
4946     (forall (some-var v) (p1 & p2)) =>
4947       let {all-p1 := pick-any x
4948                      conclude (replace-var v x p1)
4949                        (!left-and (!uspec premise x));
4950            all-p2 := pick-any x
4951                      conclude (replace-var v x p2)
4952                        (!right-and (!uspec premise x))}
4953         (!both all-p1 all-p2)
4954   | ((forall (some-var v1) (some-sent p1)) & (forall (some-var v2) (some-sent p2))) =>
4955       pick-any x
4956         (!both (!uspec (!left-and premise) x)
4957                (!uspec (!right-and premise) x))
4958   | (exists (some-var v) (p1 | p2)) =>
4959      pick-witness w for premise wp
4960        (!cases wp
4961              assume (replace-var v w p1)
4962                let {some-p1 := (!egen (exists x (replace-var v x p1))  w)}
4963                  (!either some-p1 (exists x (replace-var v x p2)))
4964              assume (replace-var v w p2)
4965                let {some-p2 := (!egen (exists x (replace-var v x p2)) w)}
4966                  (!either (exists x (replace-var v x p1)) some-p2))
4967
4968   | ((exists (some-var v1) (some-sent p1)) | (exists (some-var v2) (some-sent p2))) =>
```

```
      let {goal := (exists x (or (replace-var v1 x p1) (replace-var v2 x p2)))}
        (!cases premise
           assume case-1 := (exists v1 p1)
              pick-witness w for case-1 wp # we now have (P w) in the a.b.
                let {p1w|p2w := (!either wp (replace-var v2 w p2))}
                   (!egen goal w)
           assume case-2 := (exists v2 p2)
             pick-witness w for case-2 wp
               let {Pw|Qw := (!either (replace-var v1 w p1) wp)}
                 (!egen goal w))
  | (exists (some-var v) (p1 & p2)) =>
      pick-witness w for premise wp
         (!both (!egen (exists v p1) w)
                (!egen (exists v p2) w))
  | ((forall (some-var v1) p1) | (forall (some-var v2) p2)) =>
      pick-any x
        (!cases premise
          assume case1 := (forall v1 p1)
            (!either (!uspec case1 x) (replace-var v2 x p2))
          assume case2 := (forall v2 p2)
            (!either (replace-var v1 x p1) (!uspec case2 x)))
  }

define stopgap := force

# Programmatic way to introduce a datatype by a given name and an arbitrary
# name of constant constructors of the form c_1, c_2, ..., c_N, where both
# the letter 'c' and the number N are specified as inputs:

define (make-datatype datatype-name constructor-letter N) :=
  let {constructors := (map lambda (i)
                             (join constructor-letter (val->string i))
                           (1 to N));
       cmd := (join "datatype " datatype-name " := " (separate constructors " | "))}
           (process-input-from-string cmd)
```