

lib/basic/rewriting.ath

```

1 (load-file "lib/basic/prop-tab.ath")
2
3 (load-file "lib/basic/tableaux.ath")
4
5 (load-file "lib/basic/msr.ath")
6
7 ## (standard-reduce-proc-name-suffix)
8
9 (define (dm-rec premise)
10   (!prop-taut premise (app-dm premise)))
11
12 ## NEW DEFS, JULY 03 2015:
13 # (define (get-eval-proc-name' f)
14 #   (let ((N (arity-of f))
15 #         (t (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 N))))))
16 #     (get-eval-proc-name (root t))))
17
18 # (define (get-eval-proc-name-1' f)
19 #   (let ((N (arity-of f))
20 #         (t (app-proc f (map (lambda (_) (fresh-var)) (from-to 1 N))))))
21 #     (get-eval-proc-name-1 (root t))))
22 # (define get-eval-proc-name get-eval-proc-name')
23 # (define get-eval-proc-name-1 get-eval-proc-name-1')
24
25 (define (get-reduce-proc-name f)
26   (join (get-eval-proc-name f) (standard-reduce-proc-name-suffix)))
27
28 (define (get-reduce-proc-name-1 f)
29   (join (get-eval-proc-name-1 f) (standard-reduce-proc-name-suffix)))
30
31 (define derive-theorem
32   (method (goal premises)
33     (!vprove-from goal premises [['poly true] ['subsorting false] ['max-time 60000]])))
34
35 (define mderive-theorem
36   (method (goal premises)
37     (!vprove-from goal premises [['poly false] ['subsorting false] ['max-time 600000]])))
38
39 (define sderive-theorem
40   (method (goal premises)
41     (!sprove-from goal premises [['poly true] ['subsorting false] ['max-time 60]])))
42
43
44 (define (existence p q)
45   (dlet ((q-body (quant-body q)))
46     (dmatch (match-props-modulo-CD p q-body)
47       ((some-sub sub) (dseq (!mp (!taut (if p (sub q-body))) p)
48                             (!egen* q (sub (qvars-of q)))))))
49
50 (define (restrict-right-hand-side-sorts left right)
51   (let ((left-fvars (vars left))
52         (left-fvar-names (map var->string left-fvars))
53         (right-fvars (filter (vars right)
54                               (lambda (v) (member? (var->string v) left-fvar-names)))))
55     (match (match-terms left-fvars right-fvars)
56       ((some-sub sub) (sub right))
57       (_ right))))
58
59
60 (define (sorted-tran eq1 eq2)
61   (dtry (!tran eq1 eq2)
62     (dlet (([eq1' eq2'] (match (and eq1 eq2)
63                                ((and (some-sent p1) (some-sent p2)) [p1 p2])))
64           (eq1'' (!sort-instance eq1 eq1'))
65           (eq2'' (!sort-instance eq2 eq2'))))
66     (!tran eq1'' eq2''))))
67

```

```

68 (define (restrict-right-hand-side-sorts l r) r)
69
70 (define deval-cell (cell ()))
71
72 (primitive-method (leibniz t1 t2 newP fv)
73   (iff (replace-var fv t1 newP)
74         (replace-var fv t2 newP)))
75
76 (define (leibniz t1 t2 newP fv)
77   (!force (iff (replace-var fv t1 newP)
78                (replace-var fv t2 newP))))
79
80
81 (define vpf' vpf)
82
83 (define (vpf goal props)
84   (dlet ((_ (print "\nCalling external theorem prover...\n")))
85     (!vpf' goal props)))
86
87 (define (vpf goal props)
88   (dlet ((_ ()))
89     (!fail "About to call external ATP, failing instead...\n")))
90
91 (declare unary ((T) -> (T) T))
92
93 # (declare (==> <-- <--> <== ==> <==>) Boolean)
94
95 # (declare (==> <-- <-->) Boolean)
96 # (define ==> ==>)
97 # (define <== <==)
98 # (define <==> <==>)
99
100 conclude neg-id-lemma :=
101   (forall ?x . false <==> forall ?y . ?x /= ?y)
102   pick-any x
103     (!equiv
104       assume false
105         (!from-false (forall ?y . x /= ?y))
106         assume hyp := (forall ?y . x /= ?y)
107           (!absurd conclude (x = x)
108             (!reflex x)
109             conclude (x /= x)
110             (!uspec hyp x)))
111
112 #####
113 # (define (direction? x)
114   # (member? x [= --> <-- <--> <== ==> <==>]))
115
116 # (define (equational? x)
117   # (member? x [= --> <--]))
118
119 # (define (chain-symbol? x)
120   # (member? x [--> <-- <--> <== ==> <==> =]))
121
122 (define (direction? x)
123   (|| (member? x [= --> <-- <--> ==> <==>])
124       (&& (binary-proc? x) (let ((res (x true true)))
125         (|| (equal? res (if true true)) (equal? res (iff true true)))))))
126
127 (define (equational? x)
128   (member? x [= --> <--]))
129
130 (define chain-symbol? direction?)
131
132 (define (direction->string x)
133   (match x
134     (if "==">)
135     (iff "<==>")
136     (_ (symbol->string x))))
137

```

[illegible]

```

208
209 #####
210 # If a proposition of the form (and Q R), (or Q R), or (iff Q R) is in
211 # the assumption base, prove the same proposition with the arguments
212 # exchanged. E.g.,
213 # (assume (and A B) (!reorder (and A B)))
214 # proves (if (and A B) (and B A))
215
216 (define (reorder P)
217   (dmatch P
218     ((and Q R)
219      (!both (!right-and (and Q R))
220              (!left-and (and Q R))))
221     ((or Q R)
222      (!cases (or Q R)
223               (assume Q
224                       (!either R Q))
225               (assume R
226                       (!either R Q))))
227     ((iff Q R)
228      (!equiv
229       (assume R
230        (!mp (!right-iff (iff Q R)) R))
231       (assume Q
232        (!mp (!left-iff (iff Q R)) Q))))))
233
234 ## The following method takes two terms t1 and t2 where
235 ## the equality (= t1 t2) is in the assumption base, and
236 ## derives the theorem (= t2 t1).
237
238 ## This method takes three terms t1, t2, and t3 such that
239 ## t1 = t2 and t2 = t3 hold, and derives the equality t1 = t3.
240
241 ## The method below takes a term t1, a theorem P, and a term t2,
242 ## where the equality (= t1 t2) holds, and returns the proposition
243 ## obtained from P by replacing every occurrence of t1 by t2.
244
245 (define (replace-term-in-term t1 t t2)
246   (match t
247     ((val-of t1) t2)
248     (((some-symbol f) (some-list args))
249      (make-term f (map (lambda (t) (replace-term-in-term t1 t t2)) args)))
250     (s s)))
251
252 (define (replace-term-in-prop t1 P t2)
253   (match P
254     ((some-atom t) (replace-term-in-term t1 t t2))
255     ((not _Q) (not (replace-term-in-prop t1 _Q t2)))
256     (((some-sent-con pc) (some-list props))
257      (pc (map (lambda (p)
258                  (replace-term-in-prop t1 p t2)) props)))
259     (((some-quant quant) v B) (quant v (replace-term-in-prop t1 B t2))))))
260
261 define replace-term-in-sentence := replace-term-in-prop
262
263 (define (substitute-equals t1 P t2)
264   (dlet ((fv (fresh-var (sort-of t1)))
265          (newP (replace-term-in-prop t1 (rename P) fv))
266          (biconditional (dseq (dcheck ((holds? (= t1 t2)) (!sym (= t1 t2)))
267                                     (else (!sym (= t2 t1))))
268                               (!leibniz t1 t2 newP fv))))
269     (!mp (!left-iff biconditional) P)))
270
271 (define (equality-to-equivalence p)
272   (dmatch p
273     ((= s t) (!equiv (assume s
274                        (!substitute-equals s s t))
275                       (assume t
276                        (!substitute-equals t t s))))))
276
277

```

```

278  ## The following is a more selective, positional version of
279  ## substitute-equals. It takes a term t1, a theorem P, a position
280  ## pos (represented as a list of numeric terms, say [2 1 4]) and
281  ## a term t2, where the equality (= t1 t2) must hold. It returns
282  ## the proposition obtained from P by replacing the occurrence of
283  ## t1 in P at position pos by t2.
284
285  (define (pos-substitute-equals t1 P pos t2)
286    (dlet ((t1=t2 (!claim (= t1 t2)))
287          (v (fresh-var (sort-of t1)))
288          (newP (prop-pos-replace P pos v))
289          (biconditional (!leibniz t1 t2 newP v)))
290      (!mp (!left-iff biconditional) P)))
291
292  ## The method eq-congruence takes two terms t1 and t2, where
293  ## the equality (= t1 t2) must hold, a term t, and a variable v
294  ## and returns the equality (= t1' t2'), where t1' is obtained from
295  ## t by replacing every occurrence of v by t1, and t2' is obtained
296  ## from t by replacing every occurrence of v by t2.
297
298  # (define eq-congruence
299  #   (method (t1 t2 t v)
300  #       (dlet ((t1=t2 (!claim (= t1 t2)))
301  #             (v' (fresh-var))
302  #             (newt (replace-var v v' t))
303  #             (newt{t2/v'} (replace-var v' t2 newt))
304  #             (prop (= newt newt{t2/v'}))
305  #             (newt{t1/v'}=newt{t2/v'}<=>newt{t2/v'}=newt{t2/v'} (!leibniz t1 t2 prop v'))
306  #             (newt{t2/v'}=newt{t2/v'}=>newt{t1/v'}=newt{t2/v'}
307  #               (!right-iff newt{t1/v'}=newt{t2/v'}<=>newt{t2/v'}=newt{t2/v'})))
308  #       (newt{t2/v'}=newt{t2/v'} (!reflex newt{t2/v'})))
309  #       (!mp newt{t2/v'}=newt{t2/v'}=>newt{t1/v'}=newt{t2/v'}
310  #           newt{t2/v'}=newt{t2/v'})))
311
312  ## The following method, positional congruence, works with positions
313  ## instead of variables. It takes again two terms t1 and t2 such that
314  ## (= t1 t2) is a theorem, a term t, and a position pos, and returns
315  ## the equality (= t1' t2'), where t1' is obtained from t by replacing
316  ## plugging t1 at position pos, and t2' is obtained from t by plugging
317  ## t2 at position pos.
318
319  # (define (pos-congruence t1 t2 t pos)
320  #   (dlet ((v (fresh-var [t1 t2 t]))
321  #         (newt (term-replace t pos v)))
322  #       (!eq-congruence t1 t2 newt v)))
323
324  ##=====
325  ##
326  ##=====
327
328  ## The method fun-cong below takes a function symbol f (of arbitrary arity),
329  ## and two lists of terms, s-terms = [s1 ... sn] and t-terms = [t1 ... tn],
330  ## such that s_i = t_i is in the assumption base for every i = 1,...,n,
331  ## and derives the equality f(s1,...sn) = f(t1,...,tn).
332
333  (define (fun-cong f s-terms t-terms)
334    (dletrec ((v (fresh-var (join (vars* s-terms) (vars* t-terms))))
335              (do-args (method (first-s_i first-t_i rem-s_j rem-t_j eq)
336                                (dmatch [rem-s_j rem-t_j]
337                                  ([[] []] (!claim eq))
338                                  ([ (list-of s_j more-s_j) (list-of t_j more-t_j)]
                                   (dlet ((F (= (make-term f s-terms)
339                                                (make-term f (join first-t_i (join [v] more-s_j)))))
340                                       (bi-cond (!leibniz s_j t_j F v))
341                                       (new-eq (!mp (!left-iff bi-cond) eq)))
342                                       (!do-args (join first-s_i [s_j])
343                                                  (join first-t_i [t_j])
344                                                  more-s_j more-t_j new-eq))))))
345              (!do-args [] [] s-terms t-terms (!reflex (make-term f s-terms)))))
346
347

```

```

348 (define (fun-cong f s-terms t-terms)
349   (!fcong (= (make-term f s-terms) (make-term f t-terms))))
350
351 #####
352 ## RELATION CONGRUENCE
353 #####
354
355 ## The method rel-cong below takes an atomic theorem P of the form (R s1 ... sn),
356 ## the terms s-terms [s1...sn], and the terms t-terms [t1...tn], where si = ti
357 ## must be in the assumption base for all i, and returns the theorem (R t1 ... tn).
358
359 (define (rel-cong P s-terms t-terms)
360   (dletrec ((do-args (method (s-terms t-terms theorem)
361     (dmatch [s-terms t-terms]
362       ([[] []] (!claim theorem))
363       ([ (list-of s more-s) (list-of t more-t)]
364         (dlet ((new-theorem (!substitute-equals s theorem t)))
365           (!do-args more-s more-t new-theorem))))))
366     (!do-args s-terms t-terms P)))
367
368 ## DRM: the above definition applies each substitution to the whole
369 ## theorem P, which is incorrect (according to the comment preceding
370 ## it). The following is a modified version that uses
371 ## pos-substitute-equals instead of substitute-equals, to restrict
372 ## application of individual substitution to the corresponding
373 ## argument position only.
374
375 (define (rel-cong P s-terms t-terms)
376   (dletrec ((do-args (method (s-terms t-terms theorem n)
377     (dmatch [s-terms t-terms]
378       ([[] []] (!claim theorem))
379       ([ (list-of s more-s) (list-of t more-t)]
380         (dlet ((new-theorem
381           (!pos-substitute-equals s theorem [n] t)))
382           (!do-args more-s more-t new-theorem (plus n 1))))))
383     (!do-args s-terms t-terms P 1)))
384
385 ## The argument list s-terms in the method rel-cong above is superfluous, since
386 ## it can be extracted from the atomic theorem P. Hence rel-cong-2 below simply
387 ## takes a theorem P, which again must be of the form (R s1 ... sn), and a list
388 ## of terms t-terms [t1 ... tn], where si = ti must be in the asm. base, and
389 ## derives the theorem (R t1 ... tn)
390
391 (define (rel-cong-2 P t-terms)
392   (dcheck ((atom? P) (!rel-cong P (children P) t-terms))))
393
394 #####
395 ## RECURSIVE CONGRUENCE
396 #####
397
398 ## This is a powerful recursive congruence method. If any
399 ## subterms of t1 and t2 in corresponding positions are equal
400 ## (with the equality in the assumption base), everything else
401 ## being the same, then the theorem (= t1 t2) is returned.
402
403 (define (rec-cong t1 t2)
404   (dmatch (equal? t1 t2)
405     (true (!equality t1 t1))
406     (_ (dmatch (fetch (lambda (P)
407       (|| (equal? P (= t1 t2))
408         (equal? P (= t2 t1))))))
409       (()) (dlet ((root1 (root t1))
410         (root2 (root t2))
411         (args1 (children t1))
412         (args2 (children t2)))
413         (dmatch (equal? root1 root2)
414           (true (dletrec ((do-args
415             (method (s-terms t-terms)
416               (dmatch [s-terms t-terms]
417                 ([[] []] (!fun-cong root1 args1 args2))

```

```

418                                     ([ (list-of s1 more) (list-of t1 rest)]
419                                     (dseq
420                                     (!rec-cong s1 t1)
421                                     (!do-args more rest))))))
422                                     (!do-args args1 args2))))))
423 (P (dmatch P
424       ((= (val-of t1) (val-of t2)) (!claim P))
425       (_ (!sym (= t2 t1))))))
426
427
428
429
430 (define (rec-cong t1 t2)
431   (dcheck
432     ((equal? t1 t2) (!reflex t1))
433     ((holds? (= t1 t2)) (!claim (= t1 t2)))
434     ((holds? (= t2 t1)) (!sym (= t2 t1)))
435     (else (dlet ((root1 (root t1))
436                   (root2 (root t2))
437                   (args1 (children t1))
438                   (args2 (children t2)))
439               (dmatch (equal? root1 root2)
440                 (true (dletrec ((do-args
441                                   (method (s-terms t-terms)
442                                     (dmatch [s-terms t-terms]
443                                       ([[] []] (!fun-cong root1 args1 args2))
444                                       ([ (list-of s1 more) (list-of t1 rest)]
445                                         (dseq
446                                           (!rec-cong s1 t1)
447                                           (!do-args more rest)))))))
448                                     (!do-args args1 args2))))))
449
450
451 (define (rec-rel-cong p q)
452   (dlet ((p-terms (children p))
453         (q-terms (children q)))
454     (!map-method
455       (method (term-pair)
456         (dmatch term-pair
457           ([ (some-term s) (some-term t)] (!rec-cong s t)))
458         (list-zip p-terms q-terms)
459         (method (results)
460           (!rcong p q))))))
461
462
463
464 ## The procedure try-rewrite determines whether a term s rewrites into a term t
465 ## on the basis of a given rewrite rule left -> right, and if so, returns the
466 ## corresponding substitution. Specifically, a call (try-rewrite s t left right K)
467 ## will return a substitution theta whenever (a) s matches left under theta
468 ## and (b) applying theta to right yields t. If either (a) or (b) is false, then
469 ## the failure continuation K is invoked.
470
471
472 # (define match-terms-core match-terms)
473
474 # (define (match-terms s t uvars)
475 #   (match [s t]
476 #     ([_ (- (- (val-of s)))] true)
477 #     ([(- (- (some-term x))] x] true)
478 #     (_ (match-terms s t uvars))))
479
480 # (define (equal-up-to-double-negation s t)
481 #   (|| (equal? s t)
482 #       (&& (numeral? s)
483 #           (|| (equal? (- (- s)) t) (equal? s (- (- t))))))
484
485 (define (try-rewrite s t left right uvars K)
486   (let ((eqn (= s t)))
487     (match (match-terms eqn (= left right) uvars)

```

```

488      ((some-sub sub) (check (||| (equal? t (sub right)) (sort-instance? (rhs eqn) (sub right))) sub)
489                        (else (K))))
490      (_ (K))))
491
492  ## A call (rewrites? s t rule direction), where rule is a universally quantified
493  ## identity (forall v1 ... vk (= t1 t2)) or (forall v1 ... vk (if p (= t1 t2))),
494  ## possibly with zero quantifiers, will return the relevant substitution
495  ## if s rewrites into t on the basis of rule, or if t rewrites into s on
496  ## the basis of rule, depending on the given direction (or if either holds,
497  ## if the direction is =). If neither holds, then the constant 'false' is returned.
498
499  (define (get-identity p)
500    (match p
501      ((= _ _) p)
502      ((if _ (bind consequent (= _ _)) consequent)
503       ((iff _ (bind consequent (= _ _)) consequent)
504        (_ ())))
505
506  ## UQM
507
508  (define (rewrites? s t rule direction)
509    (match rule
510      ((forall (some-list uvars) (= (some-term L) (ite _ (some-term R1) (some-term R2))))
511       (match (rewrites? s t (forall* uvars (= L R1)) direction)
512         ((some-sub sub) sub)
513         (_ (rewrites? s t (forall* uvars (= L R2)) direction))))
514      ((forall (some-list uvars) body)
515       (match (get-identity body)
516         ((= left right) where (negate (&& (var? left) (var? right)))
517          (let ((failure-cont (lambda () false)))
518            (match direction
519              (--> (try-rewrite s t left right uvars failure-cont))
520              (<-- (try-rewrite t s left right uvars failure-cont))
521              (= (try-rewrite s t left right uvars
522                    (lambda () (try-rewrite t s left right uvars failure-cont))))))
523          (_ false)))
524      (_ false)))
525
526  (define (show t1 t2 direction show-left-term?)
527    (let ((f (lambda ()
528                (seq (indent (plus (get-trace-level) 1))
529                    (check (show-left-term? (seq (indent-print (plus (times 4 (get-trace-level)) 2) t1)
530                                                (print newline)
531                                                (indent (plus (get-trace-level) 1))))
532                    (else (indent (minus (get-trace-level) 1))))
533                    (print (join (direction->string direction) newline))
534                    (indent (plus (get-trace-level) 1))
535                    (indent-print (plus (times 4 (get-trace-level)) 2) t2)
536                    (print newline))))))
537      (check ((equal? t1 t2) (f))
538              (else (match (get-debug-mode)
539                          ("rewriting" (f))
540                          ("detailed" (f))
541                          (_ ()))))))
542
543  (define (prove-condition p methods)
544    (!find-some methods (method (M) (!M p)) fail))
545
546  (define (negateR x)
547    (match x
548      (true false)
549      (false true)
550      (_ (not x))))
551
552
553  (define (orient rule)
554    (dmatch rule
555      ((forall (some-list var-list) (if ant (= s t)))
556       (dcheck (||| (subset? (vars t) (vars s))
557                    (negate (subset? (vars s) (vars t))))))

```



```

558         (negate (null? (fv rule))))
559         (!claim rule))
560     (else (!generalize var-list (method (eigen-vars)
561         (dlet ((rule' (!uspec* rule eigen-vars))
562             (ant' (antecedent rule'))))
563             (assume ant'
564                 (!sym (!mp rule' ant'))))))))
565     ((forall (some-list var-list) (iff ant (= s t)))
566     (dcheck (||| (subset? (vars t) (vars s))
567         (negate (subset? (vars s) (vars t)))
568         (negate (null? (fv rule))))
569         (!claim rule))
570     (else (!generalize var-list (method (eigen-vars)
571         (dlet ((rule' (!uspec* rule eigen-vars))
572             (rule'' (!left-iff rule'))
573             (ant' (antecedent rule'))))
574             (assume ant'
575                 (!sym (!mp rule'' ant'))))))))
576     ((forall (some-list var-list) (= s t))
577     (dcheck (||| (subset? (vars t) (vars s))
578         (negate (subset? (vars s) (vars t)))
579         (negate (null? (fv rule))))
580         (!claim rule))
581     (else (!generalize var-list (method (eigen-vars)
582         (dlet ((rule' (!uspec* rule eigen-vars))
583             (!sym rule'))))))))
584
585 # (define rule (forall ?x ?y (= ?x (Plus ?y ?y)))
586
587 # (assume rule (!orient rule))
588
589 # (define orient claim)
590
591 ## A call (!rewrite* t1 t2 rules direction) attempts to derive the identity (= t1 t2)
592 ## by rewriting t1 into t2 or vice versa (depending on the 'direction' argument)
593 ## on the basis of the given rules, each of which is of the form
594 ##      (forall v1 ... vk (= s t))
595 ##      or (forall v1 ... vk (if _ (= s t))),
596 ##      or (forall v1 ... vk (iff _ (= s t))),
597 ## where we might have zero universal quantifiers. The rules must be in the a.b.
598 ## Any number of subterms of each term may be rewritten, by any of the given rules.
599 ## However, if the given direction is other than =, i.e., if it's either --> or
600 ## <--, then the rewriting can only proceed in one direction, e.g., in the case of -->,
601 ## the rules are applied only to subterms of t1, and in the case of <--, only
602 ## to subterms of t2. This can limit the usefulness of the method, since in the
603 ## general case there is no reason to explicitly restrict the direction of the
604 ## rewriting (from a cognitive perspective, equalities are generally perceived
605 ## as inherently symmetric). So the default direction should be =.
606 ## Note that if t1 and t2 are identical then no rules need to be supplied,
607 ## i.e., (!rewrite* t t []) will always derive (= t t). Obviously, if the
608 ## rule is conditional then the condition(s) must obtain for the rule
609 ## to be applied successfully.
610
611 (define (rewrite-one-redex t1 t2 equation sub)
612     (dlet ([uvars left right] (decompose-equation equation)))
613     (dletrec ((loop (method (s t)
614         (dcheck ((&& (equal? (sub left) s)
615             (equal? (sub right) t))
616             (dmatch (!uspec* equation (sub uvars))
617                 ((bind p (= (val-of s) _)) (!claim p))
618                 ((bind p (if ant (= (val-of s) _)) (!mp p (!prove-components-harder ant)))
619                 ((bind p (if ant (= _ _)) (!sym (!mp p (!prove-components-harder ant))))
620                 ((bind p (iff _ (= (val-of s) _))
621                     (dlet ((p' (!left-iff p)))
622                         (!mp p' (!prove-components-harder (antecedent p')))))
623                 ((bind p (iff _ (= _ _))
624                     (dlet ((p' (!left-iff p)))
625                         (!sym (!mp p' (!prove-components-harder (antecedent p'))))))))
626         ((equal? s t) (!equality s t))
627         (else (!map-method

```

```

628         (method (term-pair)
629           (dmatch term-pair
630             ([s' t'] (!rewrite-one-redex s' t' equation sub))))
631         (zip (children s) (children t))
632         (method (results)
633           (!fcong (= s t)))))))))
634     (!loop t1 t2)))
635
636 (define (rewrite-one-redex t1 t2 equation sub)
637   (dlet ([uvars left right] (decompose-equation equation))
638     (body (!uspec* equation (sub uvars))))
639   (dletrec ((loop (method (s t)
640     (dcheck ((|| (equal? (sub left) s) (equal? (sub right) s))
641       (dmatch body
642         ((bind p (= (val-of s) _)) (!claim p))
643         ((bind p (= _ _)) (!sym p))
644         ((bind p (if ant (= (val-of s) _)) (!mp p (!prove-components-harder ant)))
645         ((bind p (if ant (= _ _)) (!sym (!mp p (!prove-components-harder ant))))
646         ((bind p (iff _ (= (val-of s) _))
647           (dlet ((p' (!left-iff p))
648             (!mp p' (!prove-components-harder (antecedent p')))))
649         ((bind p (iff _ (= _ _))
650           (dlet ((p' (!left-iff p))
651             (!sym (!mp p' (!prove-components-harder (antecedent p'))))))))
652         ((equal? s t) (!equality s t))
653         (else (!map-method
654           (method (term-pair)
655             (dmatch term-pair
656               ([s' t'] (!loop s' t'))))
657           (zip (children s) (children t))
658           (method (results)
659             (!fcong (= s t))))))))))
660     (!loop t1 t2))))))
661
662 ## ite-ir takes an equality of the form (= s (ite C R1 R2)) and produces
663 ## (= s R1) if C holds, or else (= s R2) if (~ C) holds.
664
665 (primitive-method (ite-ir ite-equality)
666   (match ite-equality
667     ((= (some-term s) (ite condition res1 res2))
668       (check ((holds? condition) (= s res1))
669         ((holds? (complement condition)) (= s res2)))))
670
671 (primitive-method (ite-ir* ite-equality)
672   (letrec ((loop (lambda (ite-equality)
673     (match ite-equality
674       ((= (some-term s) (ite condition res1 (nested-ite-term as (ite _ _ _)))
675         (check ((holds? condition) (= s res1))
676         (else (loop (= s nested-ite-term))))
677       ((= (some-term s) (ite _ _ _)) (let ((res (!ite-ir ite-equality)) res))))))
678     (loop ite-equality)))
679
680 (define (ite-ir' C R1 R2)
681   (dlet ((ite-term (ite C R1 R2))
682     (!ite-ir (!reflex ite-term)))
683
684 (define (ite-term? t)
685   (match t
686     ((ite _ _ _) true)
687     (_ false)))
688
689
690 (define (search t1 t2 rules direction)
691   (dtry (!drs-bf t1 t2 rules rewrite-one-redex)
692     (dcheck ((equal? direction =)
693       (!sym (!drs-bf t2 t1 rules rewrite-one-redex)))))
694
695
696 (define (find-ite-match-0 t1 t2 lhs C rhs1 rhs2)
697   (let ((eqn (= t1 t2)))

```

```

698 (match (match-terms eqn (= lhs rhs1))
699   ((some-sub _) C)
700   (_ (match (match-terms eqn (= lhs rhs2))
701     ((some-sub _) (complement C))
702     (_ (match rhs1
703       ((ite (some-term C') (some-term rhs1') (some-term rhs2'))
704         (find-ite-match-0 t1 t2 lhs C' rhs1' rhs2'))
705       (_ ())))))))))
706
707 (define (find-ite-match t1 t2 lhs C rhs1 rhs2)
708   (let (
709     #(_ (print "\nAbout to call find-ite-match on t1: " t1 ", t2: " t2 ", lhs: " lhs ", C: " C
710     #      ", rhs1: " rhs1 ", and rhs2: " rhs2)
711     )
712   (match (find-ite-match-0 t1 t2 lhs C rhs1 rhs2)
713     (()) (match (find-ite-match-0 t2 t1 lhs C rhs1 rhs2)
714       ((some-sent cond) [cond 'reversed])
715       (_ (error "No match..."))))
716     ((some-sent cond) [cond 'normal]))))
717
718 (define (rewrite** t1 t2 rules direction)
719   (dletrec ((loop (method (terms1 terms2)
720     (dmatch [terms1 terms2]
721       ([[[] []] (!fcong (= t1 t2))]
722       ([([list-of s rest1] [list-of t rest2])
723         (dseq (!rewrite** s t rules direction)
724           (!loop rest1 rest2))))))
725     (rules' (match rules
726       ((some-list _) rules)
727       ((some-sent P) [P])))
728   (dlet ((methods' (filter rules' method?)))
729     (dcheck ((equal? t1 t2) (!equality t1 t2))
730       (else (!find-some rules'
731         (method (rule)
732           (dlet ((rule' (!orient rule))
733             (dmatch [rule' (rewrites? t1 t2 rule' direction)]
734               ([([forall (some-list vars) _) (some-sub sub)]
735                 (dlet (# (_ (print "\nMATCHING SUB for rewriting: "
736                 # t1 " into " t2 ":\n" sub "\nand rule':\n" rule' "\nand vars:\n" vars))
737                 (res
738                 (dmatch (!uspec* rule' (sub vars))
739                   ((bind p (= (some-term lhs) (ite (some-term C)
740                     (some-term rhs1)
741                     (some-term rhs2))))
742                   (dlet ((res (dmatch (find-ite-match t1 t2 lhs C rhs1 rhs2)
743                     ([([some-sent cond] 'reversed)
744                       (dlet ((_) (!prove-condition cond
745                         (add prove-components-of methods'))))
746                       (!sym (!ite-ir* p))))
747                     ([([some-sent cond) _]
748                       (dlet ((_) (!prove-condition cond
749                         (add prove-components-of methods'))))
750                       (!ite-ir* p))))))
751                   (dcheck ((equal? res (= t1 t2)) (!claim res))
752                     (else (!sort-instance res (= t1 t2))))))
753                   ((bind p (= _ _)) (dmatch (match-terms (= t1 t2) p vars)
754                     ((some-sub _) (!claim p))
755                     (_ (dmatch (match-terms (= t2 t1) p vars)
756                       ((some-sub _) (!sym p))))))
757                   ((bind p (if _ (= (val-of t1) _)) (dlet ((_) ()
758                     ## (_ (print "\nAntecedent: " (antecedent p))
759                     (th (!prove-condition (antecedent p)
760                       (add prove-component.
761                       ## (_ (print "\nPROVED ANTECEDENT!\n"))
762                       (_ ()))
763                       (!mp p (!prove-condition (antecedent p)
764                         (add prove-components-
765                   ((bind p (if _ (= _ _)) (dlet (## (_ (print "\nTrying antecedent: " p))
766                     (th (!prove-condition (antecedent p)
767                       (add prove-components-of meth

```

```

768                                     (_ ()))
769                                     (!sym (!mp p th)))
770      ((bind p (iff _ (= (val-of t1) _)))
771       (dlet ((p' (!left-iff p)))
772        (!mp p' (!prove-condition (antecedent p') (add prove-components-of methods')))))
773
774      ((bind p (iff _ (= _ _)))
775       (dlet ((p' (!left-iff p)))
776        (!sym (!mp p' (!prove-condition (antecedent p') (add prove-components-of methods'))
777         (dtry (conclude (= t1 t2)
778                  (!claim res))
779                  (conclude (= t1 t2)
780                           (!sort-instance res (= t1 t2)))
781                  (dmatch t1
782                   ((ite C (some-term R1) (some-term R2))
783                    (dcheck ((equal? t2 R1) (dlet ((_) (!prove-condition C (add prove-component
784                                                                (!ite-ir' C R1 R2)))
785                                                                ((equal? t2 R2) (dlet ((_) (!prove-condition (complement C) (add pr
786                                                                (!ite-ir' C R1 R2))))))
787                   ))))))
788      (method ()
789       (dmatch [t1 t2]
790        ([((some-symbol f) (some-list args1)) (f (some-list args2))]
791         (dtry (!loop args1 args2)
792          (!search t1 t2 rules direction)))
793         (_ (!search t1 t2 rules direction)))))))))
794
795
796 (define (rewrite*** t1 t2 rules direction)
797   (dtry (!rewrite** t1 t2 rules direction)
798    (!fail "\nAbout to call ATPs, failing instead... \n")))
799
800 (define (rewrite*** t1 t2 rules direction)
801   (dcheck ((get-boolean-flag "atps-with-chain")
802    (!vprove-from (= t1 t2) rules [['poly true] ['subsorting false] ['max-time 160]]))
803    (else (!rewrite** t1 t2 rules direction)))
804
805 # (define (rewrite*** t1 t2 rules direction)
806 #   (!thread-methods [(method () (!rewrite** t1 t2 rules direction))
807 #                     (method () (!derive-theorem (= t1 t2) rules))])
808
809
810 ## This method takes two equational theorems eq1 and eq2, where eq1 is
811 ## t=u and eq2 is v=u, and derives the equational theorem t=v.
812 ## ****> Supersceded by combine (which is used with reduce and expand)
813
814 (define combine-equations
815   (method (eq1 eq2)
816    (dmatch eq1
817     ((= t11 t12)
818      (dmatch eq2
819       ((= t21 t22)
820        (dcheck
821         ((equal? t12 t22)
822          (dseq
823           (!sym (= t21 t22))
824           (!tran (= t11 t22) (= t22 t21))))))))))
825
826 #####
827 ##
828 ## Rewriting methods: setup, reduce, expand, combine
829
830 (define (universal-quantifiers P)
831   (match P
832    ((forall _x _Q) (add _x (universal-quantifiers _Q)))
833    (_ [])))
834
835 (define (universal-quantifierless P)
836   (match P
837    ((forall x _Q) (universal-quantifierless _Q))

```

```

838     (_ P)))
839
840 (define (positions&subterms t k)
841   (add [[] t]
842     (fold join (map
843       (lambda (child)
844         (let ((n (cell 0))
845               (p&s (positions&subterms child n)))
846           (seq
847             (set! k (plus (ref k) 1))
848             (map
849               (lambda (position&subterm)
850                 (match position&subterm
851                   ([position subterm]
852                    [(add (ref k) position) subterm]))))
853             p&s))))
854     (children t))
855   [])))
856
857 (define (positions-and-subterms t)
858   (positions&subterms t (cell 0)))
859
860 (define (attempt-rewrite current-equation new-term
861         proposition position subterm direction)
862   (dlet ((term0
863     (match (universal-quantifierless proposition)
864       ((= _lhs _rhs) _lhs)
865       ((if condition (= _lhs _rhs))
866        _lhs)
867       (_p (let ((dummy
868         (!proof-error (join "Left-hand-side of a proposition used "
869                           "in rewriting must be\n (with quantifiers removed) "
870                           "an equality or a conditional equality.\n"
871                           "Instead it was\n" (val->string _p) "\n"))))
872         dummy))))
873     (subst (unify term0 subterm)))
874   (dcheck
875     ((negate (equal? subst false))
876      (dlet ((proposition1
877        (!uspec* proposition
878          (subst (universal-quantifiers proposition))))
879        (result
880          (dmatch proposition1
881            ((= _lhs _rhs)
882              (dtry
883                (dmatch direction
884                  (--> (!pos-substitute-equals
885                    _lhs (ref current-equation) position _rhs))
886                  (<-- (dseq
887                    (!sym proposition1)
888                    (!pos-substitute-equals
889                     _rhs (ref current-equation) position _lhs))))
890                (!true-intro)))
891            ((if condition (= _lhs _rhs))
892              (dtry
893                (dseq
894                  (dcheck ((holds? condition)
895                    (!true-intro))
896                    (else (!claim false)))
897                  (dmatch direction
898                    (--> (dseq
899                      (!mp proposition1 condition)
900                      (!pos-substitute-equals
901                       _lhs (ref current-equation) position _rhs)))
902                      (<-- (dseq
903                        (!sym (!mp proposition1 condition))
904                        (!pos-substitute-equals
905                         _rhs (ref current-equation) position _lhs))))
906                (!true-intro))))))

```

```

908         (hit-target? (match result
909                       ((= _lhs _rhs)
910                        (equal? _rhs new-term))
911                        (_ false))))
912     (dcheck (hit-target?
913              (dlet ((dummy (set! current-equation result)))
914                    (!claim result)))
915              (else (!true-intro))))))
916     (else (!true-intro))))))
917
918 (define (try-all-terms current-equation new-term equation
919               positions&subterms direction)
920   (dmatch positions&subterms
921     ((list-of [position subterm] more)
922      (dlet ((attempt
923              (!attempt-rewrite current-equation new-term
924                                equation position subterm direction)))
925        (dcheck ((equal? attempt true)
926                  (!try-all-terms current-equation new-term equation
927                                    more direction))
928                  (else (!claim attempt)))))))
929
930 # For debugging uncomment the following redefinition and change
931 # the position being checked to a position where you think the error is
932 # occurring.
933 # (define (try-all-terms current-equation new-term equation
934 #                       positions&subterms direction)
935 #   (dmatch positions&subterms
936 #     ((list-of [position subterm] more)
937 #       (dlet ((dummy (seq
938 #                 (print "\nposition: ") (write position)
939 #                 (write subterm))))
940 #         (dcheck ((equal? position [2 1 2])
941 #                   (!attempt-rewrite current-equation new-term equation
942 #                                     position subterm direction))
943 #                   (else
944 #                     (dlet ((success (cell true))
945 #                           (ce (cell (ref current-equation))))
946 #                     (dseq
947 #                       (dtry (!attempt-rewrite ce new-term equation position
948 #                                     subterm direction)
949 #                         (dlet ((dummy (set! success false))
950 #                               (!true-intro)))
951 #                         (dcheck ((equal? (ref success) true)
952 #                                   (!attempt-rewrite current-equation new-term equation
953 #                                     position subterm direction))
954 #                                   (else (!try-all-terms current-equation new-term equation
955 #                                     more direction))))))))))
956
957 (define previous-equation (cell (cell true)))
958
959 (define (in-rewriting-trace-mode)
960   (member? (get-debug-mode) ["rewriting" "simple" "detailed"]))
961
962 (define (do-rewrite current-equation new-term equation direction)
963   (dlet ((old-term (match (ref current-equation)
964                           ((= _lhs _rhs) _rhs)))
965         (goal-eqn (= old-term new-term))
966         ([old-term new-term] (match goal-eqn
967                                   ((= (some-term L) (some-term R)) [L R])))
968         (dummy (check ((in-rewriting-trace-mode)
969                        (check ((negate (equal? current-equation
970                                             (ref previous-equation)))
971                                (seq
972                                  (indent (plus (get-trace-level) 1))
973                                  (print "Rewriting\n")
974                                  (indent (plus (get-trace-level) 1))
975                                  (indent-print (plus (times 4 (get-trace-level)) 2) old-term)
976                                  (print "\n")
977                                  ))
978                        ))
979         (seq
980           (indent (plus (get-trace-level) 1))
981           (print "Rewriting\n")
982           (indent (plus (get-trace-level) 1))
983           (indent-print (plus (times 4 (get-trace-level)) 2) old-term)
984           (print "\n")
985           ))
986   ))

```

```

978         (else ()))
979     (else ()))
980 (result (dmatch direction
981     (--> (!try-all-terms current-equation
982         new-term
983         (rename equation)
984         (positions&subterms (unary old-term) (cell 1))
985         direction))
986     (<-- (!try-all-terms current-equation
987         new-term
988         (rename equation)
989         (positions&subterms (unary new-term) (cell 1))
990         direction))
991     (= (dseq
992         (!derive (= old-term new-term) equation)
993         (dlet ((new (!tran (ref current-equation) (= old-term new-term)))
994             (dummy (set! current-equation new)))
995             (!claim new))))))
996 (dummy1
997 (check ((in-rewriting-trace-mode)
998     (seq
999     (indent (plus (get-trace-level) 1))
1000     (match direction
1001         (--> (print "-->\n"))
1002         (<-- (print "<--\n"))
1003         (= (print " = \n"))
1004         (indent (plus (get-trace-level) 1))
1005         (indent-print (plus (times 4 (get-trace-level)) 2) new-term)
1006         (print "\n")
1007         (set! previous-equation current-equation)))
1008     (else ())))
1009 (!claim result)))
1010
1011 (define (reduce current-equation new-term equation)
1012     (!do-rewrite current-equation new-term equation -->))
1013
1014 (define (expand current-equation new-term equation)
1015     (!do-rewrite current-equation new-term equation <--))
1016
1017 (define neither-left-nor-right (cell (cell true)))
1018
1019 (define (setup current-equation term)
1020     (dlet ((equation (!equality term term))
1021         (dummy (seq
1022             (set! current-equation equation)
1023             (set! previous-equation neither-left-nor-right))))
1024         (!claim equation)))
1025
1026 (define (combine left right)
1027     (!combine-equations (ref left) (ref right)))
1028
1029
1030 #####
1031 # More powerful rewriting method: chain
1032 #
1033 # (!chain [t_0 = t_1 [P_1] = t_2 [P_2] = ... = t_n [P_n]])
1034 # proves and returns the equation (= t_0 t_n), provided each
1035 # equation (= t_{i-1} t_i) can be obtained with reduce or expand
1036 # using P_i. Instead of =, one can use --> to restrict to
1037 # the use of reduce or <-- to restrict to the use of expand.
1038
1039 (define (chain L)
1040     (dletrec ((c (cell true))
1041         (unbracket (lambda (P) (match P ([_P] (unbracket _P)) (_ P))))
1042         (bracket (lambda (P) (match P ((list-of _x _more) P) (_ [P]))))
1043         (chain-help
1044         (method (L)
1045             (dmatch L
1046                 ((list-of --> (list-of _y (list-of _P _rest)))
1047                 (dseq

```

```

1048         (!reduce c _y (unbracket _P))
1049         (!chain-help _rest)))
1050     ((list-of <-- (list-of _y (list-of _P _rest)))
1051      (dseq
1052       (!expand c _y (unbracket _P))
1053       (!chain-help _rest)))
1054     ((list-of = (list-of _y (list-of _P _rest)))
1055      (dseq
1056       (!do-rewrite c _y (bracket _P) =)
1057       (!chain-help _rest)))
1058     ([[] (!claim (ref c)))]))
1059 (dmatch L
1060  ((list-of t rest)
1061   (dseq
1062    (!setup c t)
1063    (!chain-help rest))))))
1064
1065
1066 ## The current version of chain* supports directional rewriting, so this should work:
1067 ## (!chain* [(Plus (succ ?foo) zero) --> (succ ?foo) [Plus-zero-axiom]])
1068 ## and so should this:
1069 ## (!chain* [(succ ?foo) <-- (Plus (succ ?foo) zero) [Plus-zero-axiom]])
1070 ## but this should not:
1071 ## (!chain* [(Plus (succ ?foo) zero) <-- (succ ?foo) [Plus-zero-axiom]])
1072 ## and nor should this:
1073 ## (!chain* [(succ ?foo) --> (Plus (succ ?foo) zero) [Plus-zero-axiom]])
1074 ## But this of course should work:
1075 ## (!chain* [(succ ?foo) = (Plus (succ ?foo) zero) [Plus-zero-axiom]])
1076 ## as should this:
1077 ## (!chain* [(Plus (succ ?foo) zero) = (succ ?foo) [Plus-zero-axiom]])
1078
1079 (define (show-equiv p1 p2 direction show-left-prop?)
1080  (let ((f (lambda ()
1081             (seq (indent (plus (get-trace-level) 1))
1082                  (check (show-left-prop? (seq (indent-print (plus (times 4 (get-trace-level)) 2) p1)
1083                                             (print newline)
1084                                             (indent (plus (get-trace-level) 1))))
1085                  (else (indent (minus (get-trace-level) 1))))
1086                  (print (join (direction->string direction) newline))
1087                  (indent (plus (get-trace-level) 1))
1088                  (indent-print (plus (times 4 (get-trace-level)) 2) p2)
1089                  (print newline))))))
1090  (check ((equal? p1 p2) ())
1091         (else (match (get-debug-mode)
1092                     ("rewriting" (f))
1093                     ("detailed" (f))
1094                     (_ ())))))
1095
1096 (define (equate-subterms atom1 atom2 rules K)
1097  (dlet ((left-subterms (children atom1))
1098        (right-subterms (children atom2))
1099        (term-pairs (zip left-subterms right-subterms)))
1100    (!map-method
1101     (method (term-pair)
1102              (dmatch term-pair
1103               ([s t] (!rewrite*** s t rules =))))
1104     term-pairs K)))
1105
1106 (define (equate-atoms s t rules)
1107  (dlet ((identity (!rewrite*** s t rules =)))
1108    (!force t)))
1109
1110 (define (score atom)
1111  (lambda (atom')
1112    (check ((equal? (root atom) (root atom'))
1113           (plus 1 (check ((subset? (leaves atom') (leaves atom)) 1)
1114                          (else 0))))
1115          (else 0))))
1116
1117

```



```

1118
1119
1120 (define (align atoms-1 atoms-2)
1121   (letrec ((loop (lambda (atoms-1 atoms-2 res)
1122     (match atoms-1
1123       ([] (rev res))
1124       ((list-of atom-1 rest-1)
1125        (find-max atoms-2 (score atom-1)
1126         (lambda (A) (loop rest-1 (remove A atoms-2) (add A res)))
1127         (lambda () (loop rest-1 atoms-2 (add atom-1 res))))))))
1128     (loop atoms-1 atoms-2 [])))
1129
1130 (define (apply-tran premise-1 premise-2)
1131   (dmatch [premise-1 premise-2]
1132     ([[(iff p1 p2) (if p2 p3)] (assume p1
1133       (!mp premise-2 (!mp (!left-iff premise-1) p1)))]
1134      [(iff p1 p2) (if p3 p2)] (assume p3
1135        (!mp (!right-iff premise-1)
1136         (!mp premise-2 p3)))]
1137      [(iff p1 p2) (iff p3 p2)] (assume p3
1138        (!mp (!right-iff premise-1)
1139         (!mp (!left-iff premise-2) p3)))]
1140
1141      [(iff p1 p2) (iff p2 p3)] (!equiv-tran premise-1 premise-2))
1142     ([[(if p1 p2) (if p2 p3)] (assume p1
1143       (!mp premise-2 (!mp premise-1 p1)))]
1144      [(if p1 p2) (if p3 p1)] (!apply-tran premise-2 premise-1))
1145     ([[(if p1 p2) (iff p2 p3)] (assume p1
1146       (!mp (!left-iff premise-2)
1147        (!mp premise-1 p1)))]
1148      [(if p2 p1) (iff p2 p3)] (assume p3
1149        (!mp premise-1
1150         (!mp (!right-iff premise-2) p3)))]
1151     ([p1 (if p1 p2)] (!claim premise-2))
1152     ([p1 (iff p1 p2)] (!claim premise-2))
1153     ([p2 (iff p1 p2)] (!claim premise-2))))
1154
1155 (define (commute p)
1156   (match p
1157     ((and p1 p2) (and p2 p1))
1158     (_ p)))
1159
1160 (define (match-props-modulo-conj p q)
1161   (match (match-props p q)
1162     ((some-sub sub) sub)
1163     (_ (match [p q]
1164       ([[(if (and p1 p2) body) (if (and q1 q2) body')]
1165         [(match-props p (if (and q2 q1) body'))]]
1166        [(iff (and p1 p2) body) (iff (and q1 q2) body')]
1167         [(match-props p (iff (and q2 q1) body'))]]
1168        (_ ()))))))
1169
1170
1171 (define (get-components p kind)
1172   (match [p kind]
1173     ([[(and (some-list args)) 'conj] args)
1174      [(or (some-list args)) 'disj] args)
1175     (_ ())))
1176
1177 (define (match-props-AC p1 p2 uvars)
1178   (match [(match-props-3 p1 p2 uvars) p2]
1179     ([[(some-sub sub) _] [sub p2 'not-reversed]]
1180      # ([_ (= (some-term s) (some-term t))]
1181         # (match (match-props-3 p1 (= t s) uvars)
1182           # ((some-sub sub) [sub p2 'reversed])
1183           # (_ false)))
1184      ([_ (and q1 q2)] (let ((p2' (and q2 q1)))
1185        (match (match-props-3 p1 p2' uvars)
1186          ((some-sub sub) [sub p2' 'reversed])
1187          (_ false))))))

```

```

1188      ([_ (or q1 q2)] (let ((p2' (or q2 q1)))
1189                          (match (match-props-3 p1 p2' uvars)
1190                                ((some-sub sub) [sub p2' 'reversed])
1191                                (_ false))))
1192      ([_ (not p2')]
1193       (match p1 ((not p1') (match-props-AC p1' p2' uvars))
1194                (_ false)))
1195      (_ false)))
1196
1197 (define (is-id? dual-proc)
1198   (equal? (not true) (dual-proc (not true))))
1199
1200 (define (match-some-component p C uvars kind dual?)
1201   (match (match-props-AC p (dual? C) uvars)
1202     ([ (some-sub sub) C' rev-flag] [[C' sub rev-flag]])
1203     (_ (match (check ((is-id? dual?) []))
1204               (else (match-props-AC (dual? p) C uvars)))
1205         ([ (some-sub sub) C' rev-flag] [[C' sub rev-flag]])
1206         (_ (match (get-components C kind)
1207                   ((some-list args) (match-some-component* p args uvars kind dual? []))
1208                   (_ []))))))
1209   (match-some-component* p Cs uvars kind dual? results)
1210   (match Cs
1211     ([] results)
1212     ((list-of (some-sent C) rest)
1213      (match-some-component* p rest uvars kind dual?
1214                            (join (match-some-component p C uvars kind dual?) results)))))
1215
1216 (define (match-some-conjunct p C uvars)
1217   (match-some-component p C uvars 'conj (lambda (x) x))
1218   ([] (let ((p-conjuncts (get-conjuncts-recursive p)))
1219         (find-element p-conjuncts (lambda (p')
1220                                     (negate (null? (match-some-component p' C uvars 'conj (lambda (x) x))))))
1221         (lambda (p')
1222           (match-some-component p' C uvars 'conj (lambda (x) x)))
1223         (lambda () []))))
1224   (res res)))
1225
1226
1227 (define (match-some-conjunct p C uvars)
1228   (find-element'
1229     (get-all-conjuncts p)
1230     (lambda (L)
1231       (negate (null? L)))
1232     (lambda (q)
1233       (match-some-component q C uvars 'conj id))
1234     id
1235     (lambda () [])))
1236
1237 (define (match-some-dual-conjunct p C uvars)
1238   (match-some-component p C uvars 'conj complement))
1239
1240 (define (match-some-disjunct p C uvars)
1241   (match-some-component p C uvars 'disj (lambda (x) x)))
1242
1243 (define (match-some-dual-disjunct p C uvars)
1244   (match-some-component p C uvars 'disj complement))
1245
1246 (define (label-list L label)
1247   (let ((f (lambda (x) (add label x))))
1248     (map f L)))
1249
1250 (define (match-antecedent p left q right uvars)
1251   (let ((L1 (match-some-conjunct q right uvars))
1252         (Lb (label-list (match-some-conjunct p left uvars) 'conj))
1253         (Lc (label-list (match-some-disjunct p left uvars) 'disj))
1254         (L2 (join Lb Lc)))
1255     (#(mprint (lambda (x y) (print (join (val->string x) " " (val->string y)))))
1256      (mprint (lambda (x y) ()))
1257      (find-first' (cpred L1 L2)

```

```

1258     (lambda (pair-res)
1259       (match pair-res
1260         ([right-conjunct _ q-flag]
1261          [label left-component left-sub p-flag]]
1262         (let ((_ ()))
1263           #(_ (print "\nAbout to try matching " (if (left-sub left-component) q) " against: " (if left-compon
1264             (_ ()))
1265           (match (match-props-3 (if (left-sub left-component) q) (if left-component right-conjunct) uvars)
1266             ((some-sub sub) (seq (mprint "\nleft-component: " left-component)
1267                                  (mprint "\nright-conjunct: " right-conjunct)
1268                                  [sub label [left-component p-flag] [right-conjunct q-flag]])))
1269             (_ false))))
1270         (_ false)))
1271     (lambda () ())))
1272
1273 (define (apply-dm p q)
1274   (match p
1275     ((not (and (some-list _))) (app-dm p))
1276     ((not (or (some-list _))) (app-dm p))
1277     ((and (some-list _)) (app-dm p))
1278     ((or (some-list _)) (app-dm p))
1279     (_ (not q))))
1280
1281
1282 (define (subset-matching L1 L2 uvars)
1283   (letrec ((loop (lambda (remaining sub)
1284                     (match remaining
1285                       ([ sub)
1286                        ((list-of p rest) (find-first' L2
1287                                                       (lambda (q)
1288                                                         (match (match-props-3 (sub p) (sub q) uvars)
1289                                                           ((some-sub sub') (loop rest (compose-subs sub' sub)))
1290                                                           (_ false)))
1291                                                         (lambda () false)))))))
1292             (loop L1 empty-sub)))
1293
1294
1295 (define (match-consequent p right q left uvars)
1296   (let ((L1 (match-some-dual-conjunct p right uvars))
1297         (Lb (label-list (match-some-dual-disjunct q left uvars) 'no-dm))
1298         (Lc (label-list (match (match-props-AC q (apply-dm (complement left) q) uvars)
1299                           (false [])
1300                           ([sub y z] [[y sub z]]) 'dm))
1301         (L2 (join Lb Lc))
1302         (mprint (lambda (x y) ())))
1303   (find-first' (cprod L1 L2)
1304     (lambda (pair-res)
1305       (match pair-res
1306         ([dual-right-conjunct _ p-flag]
1307          [dm-flag dual-left-disjunct _ q-flag]]
1308         (match [dm-flag (match-props-3 (if p q) (if dual-right-conjunct dual-left-disjunct) uvars)]
1309           ([dm (some-sub sub)] (seq (mprint "\ndual-right-conjunct: " dual-right-conjunct)
1310                                     (mprint "\ndual-left-disjunct: " dual-left-disjunct)
1311                                     [sub [(complement dual-right-conjunct) p-flag]
1312                                           [(complement dual-left-disjunct) q-flag] 'dm])))
1313           ([no-dm (some-sub sub)] [sub [(complement dual-right-conjunct) p-flag]
1314                                         [(complement dual-left-disjunct) q-flag]]))
1315         (_ false)))
1316     (_ false)))
1317   (lambda ()
1318     (find-first' L1
1319       (lambda (res)
1320         (match res
1321           ([dual-right-conjunct (some-sub right-sub) p-flag]
1322            (let ((La (right-sub (get-conjuncts-recursive q)))
1323                  (Lb (right-sub (get-conjuncts-recursive (apply-dm (complement left) q))))
1324              (res-sub (subset-matching La Lb uvars)))
1325            (match res-sub
1326              ((some-sub _) [(compose-subs res-sub right-sub)
1327                             [(complement dual-right-conjunct) p-flag]]

```



```

1399 (define (conj-elim' p C)
1400   (dmatch C
1401     ((and (some-list args)) (!decompose C (method _) (!claim p))))
1402     (_ (dtry (!claim p)
1403       (!sort-instance C p)))))
1404
1405 (define (make-cond-method p q rule uvars left right M)
1406   (match (match-antecedent p left q right uvars)
1407     [(some-sub sub) 'conj [p-pat rev-flag-p] [q-pat rev-flag-q]]
1408     (method _ _ (dlet (
1409       (R (method ()
1410         (dseq (!comm-opt p)
1411           (!decompose p (method _)
1412             (dlet ((left-side (!conj-intro (sub left)))
1413               (!mp (!M (!uspec* rule (sub uvars))) left-side))))))
1414       (dmatch rev-flag-q
1415         ('reversed (!comm (!conj-elim' (commute q) (!R))))
1416         (_ (!conj-elim' q (!R)))))
1417     [(some-sub sub) 'disj rev-flag-left rev-flag-right]
1418     (method _ _ (dlet (#/_ (print "p: " p ", q: " q ", left: " left ", right: " right ", uvars: " uvars "\nsub:
1419       (inst-rule (!M (!uspec* rule (sub uvars))))
1420       (conclusion (!mp inst-rule (!disj-intro (sub left))))
1421       #/_ (mprint "\ninst-rule: " inst-rule " conclusion: " conclusion))
1422       (_ ()))
1423     (dseq (!comm-opt p)
1424       (!conj-elim' q conclusion)))))
1425   (_ (match (match-consequent p right q left uvars)
1426     [(some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q]]
1427     (method _ _ (dlet ((irule (!M (!uspec* rule (sub uvars))))
1428       ([ileft  iright] [(antecedent irule) (consequent irule)])
1429       (_ (conclude (complement iright)
1430         (!complement-conjunction iright (!commute? p rev-flag-p))))
1431       (not-ileft (conclude (complement ileft)
1432         (!mt irule (complement iright))))
1433       (!negate-disjunct not-ileft q))))
1434     [(some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q] 'dm]
1435     (method _ _ (dlet ((mprint (lambda (x y) (print x y)))
1436       (irule (!M (!uspec* rule (sub uvars))))
1437       ([ileft  iright] [(antecedent irule) (consequent irule)])
1438       (iright' (conclude (complement iright)
1439         (!complement-conjunction iright (!commute? p rev-flag-p))))
1440       (ileft' (conclude (complement ileft)
1441         (!mt irule iright')))
1442       (res (!dm-rec ileft')))
1443     (dmatch res
1444       ((and (some-list _)) (!prove-components-harder q))
1445       (_ (!commute? res rev-flag-q)))))
1446     (_ ())))))
1447
1448 (define make-cond-method-old make-cond-method)
1449
1450 (define (make-cond-method p q rule uvars left right M rule-body)
1451   (match (match-antecedent p left q right uvars)
1452     [(some-sub sub) 'conj [p-pat rev-flag-p] [q-pat rev-flag-q]]
1453     (method _ _ (dlet ((R (method ()
1454       (dseq (!comm-opt p)
1455         (!decompose p (method _)
1456           (dlet ((left-side (!conj-intro (sub left)))
1457             (rule-instance (!uspec* rule (sub uvars)))
1458             (rule-instance' (!sort-instance rule-instance (sub rule)
1459               (!mp (!M rule-instance') left-side))))))
1460       (dmatch rev-flag-q
1461         ('reversed (!comm (!conj-elim' (commute q) (!R))))
1462         (_ (!conj-elim' q (!R)))))
1463     [(some-sub sub) 'disj rev-flag-left rev-flag-right]
1464     (method _ _ (dlet (#/_ (print "p: " p ", q: " q ", left: " left ", right: " right ", uvars: " uvars "\nsub:
1465       (rule-instance (!uspec* rule (sub uvars)))
1466       (rule-instance' (!sort-instance rule-instance (sub rule-body)))
1467       (inst-rule (!M rule-instance))

```

```

1468         (conclusion (!mp inst-rule (!disj-intro (sub left))))
1469         #(_ (mprint "\ninst-rule: " inst-rule " conclusion: " conclusion))
1470         (_ ()))
1471     (dseq (!comm-opt p)
1472           (!conj-elim' q conclusion))))
1473 (_ (match (match-consequent p right q left uvars)
1474   ([ (some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q]
1475     (method (_ _) (dlet ((rule-instance (!uspec* rule (sub uvars)))
1476                          (rule-instance (!sort-instance rule-instance (sub rule-body)))
1477                          (irule (!M rule-instance))
1478                          ([ileft iright] [(antecedent irule) (consequent irule)])
1479                          (_ (conclude (complement iright)
1480                                       (!complement-conjunction iright (!commute? p rev-flag-p))))
1481                          (not-ileft (conclude (complement ileft)
1482                                               (!mt irule (complement iright)))))
1483                          (!negate-disjunct not-ileft q))))
1484     ([ (some-sub sub) [p-dual rev-flag-p] [q-dual rev-flag-q] 'dm]
1485       (method (_ _) (dlet ((mprint (lambda (x y) (print x y)))
1486                            (rule-instance (!uspec* rule (sub uvars)))
1487                            (rule-instance (!sort-instance rule-instance (sub rule-body)))
1488                            (irule (!M rule-instance))
1489                            ([ileft iright] [(antecedent irule) (consequent irule)])
1490                            (iright' (conclude (complement iright)
1491                                                  (!complement-conjunction iright (!commute? p rev-flag-p))))
1492                            (ileft' (conclude (complement ileft)
1493                                              (!mt irule iright')))
1494                            (res (!dm-rec ileft')))
1495         (dmatch res
1496           ((and (some-list _)) (!decompose res (method (_) (!prove-components-harder q))))
1497           (_ (!commute? res rev-flag-q))))))
1498   (_ ())))))
1499
1500
1501 (define (make-method p q rule)
1502   (match rule
1503     ((forall (some-list uvars) (rule-body as (if left right)))
1504      (make-cond-method p q rule uvars left right claim rule-body))
1505     ((forall (some-list uvars) (rule-body as (iff left right)))
1506      (check ((equal? left true)
1507              (match (match-props-3 q right uvars)
1508                ((some-sub sub) (method (p q)
1509                                       (!mp (!left-iff (!uspec* rule (sub uvars)))
1510                                             (!true-intro))))
1511              (_ ())))
1512      (else (let ((p->q (match (make-cond-method p q rule uvars left right left-iff rule-body)
1513                               ((some-method M) M)
1514                               (_ (make-cond-method p q rule uvars right left right-iff rule-body))))
1515              (q->p (match (make-cond-method q p rule uvars left right left-iff rule-body)
1516                        ((some-method M) M)
1517                        (_ (make-cond-method q p rule uvars right left right-iff rule-body))))))
1518      (match [(method? p->q) (method? q->p)]
1519        ([false false] ())
1520        (_ (method (premise goal)
1521                    (dcheck ((equal? premise p) (!p->q premise goal))
1522                          (else (!q->p premise goal)))))))
1523     ((some-atom A) (match p ## p is the identity here, so it can be ignored.
1524       ((= _ _) (method (_ q)
1525                        (!rec-rel-cong A q)))
1526       (_ ())))
1527     ((not (some-atom A))
1528      (match p
1529        ((= _ _)
1530         (method (_ q)
1531           (dmatch q
1532             ((not (some-atom B))
1533              (!by-contradiction q
1534               (assume B
1535                (dlet ((res (!rec-rel-cong B A)))
1536                  (!absurd res rule)))))))
1537         (_ ())))

```

```

1538     (_ ()))
1539
1540 (define (augment-methods methods rules p q)
1541   (let ((f (lambda (rule) (make-method p q rule))))
1542     (join (filter (map f rules) method?) methods)))
1543
1544 (define (try-implication-method M p q)
1545   (conclude (if p q)
1546     (dtry (conclude (if p q)
1547       (assume p
1548         (!M p)))
1549       (conclude (if p q)
1550         (!M (if p q)))
1551       (conclude (if p q)
1552         (assume p
1553           (!M p q))))))
1554
1555 (define (try-equivalence-method M p q)
1556   (dtry (conclude (iff p q)
1557     (!M (iff p q)))
1558     (conclude (iff p q)
1559       (!equiv (assume p (!M p))
1560         (assume q (!M q))))
1561     (conclude (iff p q)
1562       (!equiv (assume p (!M p q))
1563         (assume q (!M q p))))))
1564
1565 (define (get-implication-structurally p q rules methods)
1566   (!find-some (augment-methods methods rules p q)
1567     (method (M) (!try-implication-method M p q))
1568     (method ()
1569       (dmatch [p q]
1570         ([_ (val-of p)] (assume p (!claim q)))
1571         [(= (some-term s) (some-term t))
1572           (= ((some-symbol f) s) (f t))]
1573         (assume p
1574           (!fcong q)))
1575         [(some-atom _) (some-atom _)]
1576         (assume p
1577           (dtry (!equate-subterms p q rules
1578             (method (identities)
1579               (!map-method sym identities)
1580               (method (_)
1581                 (dtry (!rcong p q)
1582                   (!mp (!left-iff (!equality-to-equivalence (= p q)) p))))))
1583             (!equate-atoms p q rules))))
1584         [(and p1 p2) (and q1 q2)]
1585         (!uni-and-cong (!get-implication-structurally p1 q1 rules methods)
1586           (!get-implication-structurally p2 q2 rules methods)))
1587         [(and (some-list props1) (and (some-list props2))]
1588           (!map-method (method (pair)
1589             (dmatch pair
1590               ([p1 q1] (!get-implication-structurally p1 q1 rules methods))))
1591             (list-zip props1 props2)
1592             (method (results)
1593               (!uni-and-cong* results))))
1594         [(forall (some-var v1) (some-sentence p1)) (forall (some-var v2) (some-sentence p2))]
1595         (assume p
1596           (pick-any x2
1597             (dlet ((step-1 (!uspec p x2))
1598               (step-2 (!get-implication-structurally step-1 (replace-var v2 x2 p2) rules methods)))
1599               (!mp step-2 step-1))))
1600         [(exists (some-var v1) (some-sentence p1)) (exists (some-var v2) (some-sentence p2))]
1601         (assume hyp := p
1602           (pick-witness w hyp witness-premise
1603             (dlet ((step-1 (!get-implication-structurally witness-premise (replace-var v2 w p2) rules methods))
1604               (body (!mp step-1 witness-premise)))
1605               (!egen q w))))
1606         [(or p1 p2) (or q1 q2)]
1607         (!uni-or-cong (!get-implication-structurally p1 q1 rules methods)

```

```

1608      (!get-implication-structurally p2 q2 rules methods)))
1609      ((or (some-list props1) (or (some-list props2)))
1610       (!map-method (method (pair)
1611        (dmatch pair
1612         ([p1 q1] (!get-implication-structurally p1 q1 rules methods))))
1613        (list-zip props1 props2)
1614        (method (results)
1615         (!uni-or-cong* results)))))))))
1616
1617 (define (get-equivalence-structurally p q rules methods)
1618   (dletrec ((loop (method (p q)
1619    (!find-some (augment-methods methods rules p q)
1620     (method (M) (!try-equivalence-method M p q))
1621     (method ()
1622      (dmatch [p q]
1623       ([_ (val-of p)] (!ref-equiv p))
1624       ([ (some-atom _) (some-atom _)]
1625        (!equate-subterms p q rules
1626         (method (identities)
1627          (!map-method sym identities
1628           (method (_)
1629            (dtry (!equiv (assume p
1630             (!rcong p q))
1631              (assume q
1632               (!rcong q p))))
1633             (!equality-to-equivalence (= p q)))))))
1634       ([ (not p1) (not q1)] (!not-cong (!loop p1 q1)))
1635       ([ ((some-sent-con pc) p1 p2) (pc q1 q2)]
1636        (! (choose-cong-method pc)
1637         (!loop p1 q1) (!loop p2 q2)))
1638       ([ (forall (some-var v1) body1) (forall (some-var v2) body2)]
1639        (!ugen-cong p q loop))
1640       ([ (exists (some-var v1) body1) (exists (some-var v2) body2)]
1641        (!egen-cong p q loop)))))))))
1642   (!loop p q)))
1643
1644 (define (get-implication-through-structure p q rules methods)
1645   (dtry (!get-implication-structurally p q rules methods)
1646    (dlet ((p<==>q (!get-equivalence-structurally p q rules methods)))
1647     (assume p
1648      (!mp (!left-iff p<==>q) p))))))
1649
1650
1651 # Blocking this out so as to move get-equivalence lower, after get-implication,
1652 # since this meth-trans is hardly ever used, it seems.
1653
1654 #(define (meth-trans p1 p2 methods rules-or-methods max)
1655 # (!breadth-first p1 p2 methods max
1656 # (method () (!left-iff (!get-equivalence p1 p2 rules-or-methods))))
1657
1658 (define (meth-trans p1 p2 methods rules-or-methods max)
1659   (dlet (#(_ (mark 'H))
1660    (_ ()))
1661    (!fail "")))
1662
1663 (define (sentential-components-of p)
1664   (match p
1665    ((and (some-list _)) (get-conjuncts p))
1666    ((or (some-list _)) (get-disjuncts p))
1667    ((not q) [q])
1668    (_ [p]))))
1669
1670
1671 (define (derive-nested-rule' R p q)
1672   (dmatch R
1673    ((forall (some-list uvars) ((sc as (|| if iff)) left (forall (some-list uvars') ((|| if iff) left' right'))))
1674     (dtry (dmatch (match-props-3 (if p q) (if left' right') (join uvars' uvars))
1675      ((some-sub sub) (dlet ((M (match sc
1676       (if claim)
1677       (iff left-iff))))))

```



```

1678             (!mp (!M (!uspec* R (sub uvars)))
1679              (sub left))))))
1680 (dlet ((all-vars (join uvars' uvars))
1681       (V (join (constants&vars p) (constants&vars q))))
1682   (!find-some
1683     (ab)
1684     (method (fact)
1685       (dmatch (match-props-3 fact left all-vars)
1686         (((some-sub sub) where (non-null (fast-intersection (constants&vars fact) V)))
1687           (dlet ((M (match sc
1688             (if claim)
1689             (iff left-iff))))
1690             (!mp (!M (!uspec* R (sub uvars)))
1691              (sub left))))))
1692       (method () (!proof-error ""))))))
1693
1694
1695 (define (derive-nested-rule-new R p q)
1696   (dmatch R
1697     ((forall (some-list uvars) ((sc as (|| if iff)) left (forall (some-list uvars') ((|| if iff) left' right'))))
1698       (dtry (dmatch (match-props-3 (if p q) (if left' right') (join uvars' uvars))
1699         ((some-sub sub) (dlet ((M (match sc
1700           (if claim)
1701           (iff left-iff))))
1702           (!mp (!M (!uspec* R (sub uvars)))
1703            (sub left))))))
1703       (dlet ((all-vars (join uvars' uvars))
1704         (V (join (constants&vars p) (constants&vars q))))
1705         (!map-method-non-strictly
1706           (method (fact)
1707             (dmatch (match-props-3 fact left all-vars)
1708               (((some-sub sub) where (non-null (fast-intersection (constants&vars fact) V)))
1709                 (dlet ((M (match sc
1710                   (if claim)
1711                   (iff left-iff))))
1712                   (!mp (!M (!uspec* R (sub uvars)))
1713                    (sub left))))))
1713             (ab)
1714             (method (rules)
1715               (dmatch rules
1716                 ([[] (!proof-error "")
1717                   [(some-sent res)] (!claim res))
1718                 ((list-of _ (list-of _ _)) (!conj-intro rules)))))))))
1719
1720
1721
1722 (define (get-implication-new p q rules-or-methods)
1723   (dlet (([methods rules] (filter-and-complement rules-or-methods method?)))
1724     (!map-method-non-strictly
1725       fact->bicond
1726       rules
1727       (method (bc-rules)
1728         (!map-methods-non-strictly [(method (R) (!derive-nested-rule-new R p q))] rules
1729           (method (new-rules)
1730             (!decompose* new-rules
1731               (method (new-rules')
1732                 (dlet ((rules' (join new-rules' bc-rules rules))
1733                   (methods' (augment-methods methods rules' p q)))
1734                   (!find-some methods'
1735                     (method (M) (!try-implication-method M p q))
1736                     (method ()
1737                       (dtry (conclude (if p q)
1738                         (!get-implication-through-structure p q rules' methods'))
1739                       (!meth-trans p q methods rules-or-methods 7))))))))))
1740
1741
1742 (define (get-implication p q rules-or-methods)
1743   (dlet (([methods rules] (filter-and-complement rules-or-methods method?))
1744     )
1745     (!map-method-non-strictly [fact->bicond (method (R) (!derive-nested-rule' R p q))] rules
1746       (method (new-rules)
1747         (dlet ((rules' (join new-rules rules))
1748           (methods' (augment-methods methods rules' p q)))

```

```

1748      (!find-some methods'
1749        (method (M) (!try-implication-method M p q))
1750        (method ()
1751          (dtry (conclude (if p q)
1752            (!get-implication-through-structure p q rules' methods'))
1753            (!meth-trans p q methods rules-or-methods 7)))))))))
1754
1755 (define get-implication get-implication-new)
1756
1757 (define (get-equivalence p q rules-or-methods)
1758   (dlet (([methods rules] (filter-and-complement rules-or-methods method?)))
1759     (!map-methods-non-strictly [fact->bicond (method (R) (!derive-nested-rule R p q))] rules
1760       (method (new-rules)
1761         (dtry (dlet ((rules' (join new-rules rules))
1762           (methods' (augment-methods methods rules' p q)))
1763           (!get-equivalence-structurally p q rules' methods'))
1764           (!equiv (!get-implication p q rules-or-methods)
1765             (!get-implication q p rules-or-methods)))))))
1766
1767 (define (get-equivalence* p q rules-or-methods)
1768   (dlet ((writeln-val (lambda (x y) ()))
1769     (_ (writeln-val "p: " p))
1770     (_ (writeln-val "q: " q))
1771     (dtry (!get-equivalence p q rules-or-methods)
1772       (dlet ((_ (print "\nCalling external ATP for equivalence...\n")))
1773         (!vpf (iff p q) (filter rules-or-methods (lambda (x) (negate (method? x))))))))))
1774
1775
1776 (define (get-equivalence* p q rules-or-methods)
1777   (dlet ((writeln-val (lambda (x y) ()))
1778     (_ (writeln-val "p: " p))
1779     (_ (writeln-val "q: " q))
1780     (dcheck ((get-boolean-flag "atps-with-chain")
1781       (!vprove-from (iff p q) (filter rules-or-methods (lambda (x) (negate (method? x))))
1782         [['poly true] ['subsorting false] ['max-time 160]]))
1783       (else (!get-equivalence p q rules-or-methods))))))
1784
1785 #   (!thread-methods [(method () (!get-equivalence p q rules-or-methods))
1786 #                     (method () (!derive-theorem (iff p q) (filter rules-or-methods (lambda (x) (negate (method? x))))
1787 #                       (!fail "About to call external ATP, failing instead...\n"))))
1788
1789 (define (get-implication* p q rules-or-methods)
1790   (dlet ((writeln-val (lambda (x y) ()))
1791     (_ (writeln-val "p: " p))
1792     (_ (writeln-val "q: " q))
1793     (dtry (!get-implication p q rules-or-methods)
1794       (dlet ((_ (print "\nCalling external ATP for implication...\n")))
1795         (!vpf (if p q) (filter rules-or-methods (lambda (x) (negate (method? x))))))))))
1796
1797
1798
1799
1800 (define (get-implication* p q rules-or-methods)
1801   (dlet ((writeln-val (lambda (x y) ()))
1802     (dcheck ((get-boolean-flag "atps-with-chain")
1803       (dtry
1804         (!vprove-from (if p q) (filter rules-or-methods (lambda (x) (negate (method? x))))
1805           [['poly true] ['subsorting false] ['max-time 160]]))
1806         (!get-implication p q rules-or-methods))
1807       (else (!get-implication p q rules-or-methods))))))
1808
1809 #   (!thread-methods [(method () (!get-implication p q rules-or-methods))
1810 #                     (method () (!derive-theorem (if p q) (filter rules-or-methods (lambda (x) (negate (method? x))))
1811 #                       (!fail "About to call external ATP, failing instead...\n"))))
1812
1813 (define gi get-implication)
1814 (define gi* get-implication*)
1815
1816 (define (get-list L)
1817   (letrec ((loop (lambda (L results)

```

```

1818         (match L
1819           ([[] (rev results))
1820           ((list-of (some-list L1) more) (loop more (join L1 results)))
1821           ((list-of x more) (loop more (add x results))))))
1822     (loop L [])))
1823
1824 (define (make-list x)
1825   (match x
1826     ((some-list L) (get-list L))
1827     (_ [x])))
1828
1829 (define (rcong-rec premise goal)
1830   (dmatch [premise goal]
1831     ([[(some-symbol R) (some-list args1)] (R (some-list args2))]
1832      (dletrec ((loop (method (args1 args2)
1833                               (dmatch [args1 args2]
1834                                ([[] []] (!rcong premise goal))
1835                                ([[(list-of s rest1) (list-of t rest2)]
1836                                 (dseq (!rec-cong s t)
1837                                       (!loop rest1 rest2)))))))
1837       (!loop args1 args2))))))
1838
1839 (define (equational-step current-theorem left right L direction new-object rest
1840         get-rules no-rules-given? iteration show-info)
1841   (dlet (#(_ (print "\nCurrent theorem: " current-theorem " and new-object: " new-object))
1842         ([old-term old-term' diff?] (match [right (atom? right) (identity? current-theorem)]
1843          ([[= _ (some-term t)] _ _]
1844           [t right true])
1845           ([[(some-symbol _) (bind L (list-of _ _))] true false]
1846            [(first (rev L)) right true])
1847           ([[(some-symbol _) (bind L (list-of _ _))] true true]
1848            [right right false])
1849            [_ [right right false]))))
1850         ([old-term new-object] (match (= old-term new-object)
1851          ((= (some-term L) (some-term R)) [L R])))
1852         (rules (check (no-rules-given? (ab)) (else (make-list (head rest)))))
1853         (rule-cell (cell rules))
1854         (_ (dcheck (no-rules-given? (!true-intro)) (else (!get-rules rules rule-cell))))
1855         (rules' (ref rule-cell))
1856         (rules'' (map rename' rules'))
1857         (_ (show-info current-theorem old-term new-object direction iteration))
1858         (tran' (check ((equal? iteration 0) sorted-tran) (else tran)))
1859         (new-equation (!map-multi-method decompose rules"
1860          (method (results)
1861            (dlet (#(_ (print "\nAbout to rewrite*** with the following rules: " results))
1862                  (_ ()))
1863            (dtry (!rewrite*** old-term new-object results direction)
1864              (dcheck (diff? (!rewrite*** old-term' new-object results direction))
1865                (else (!fail)))
1866              (dlet ((eval-res (!ref deval-cell) old-term))
1867                # (eval-res (!reflex old-term))
1868                #(_ (!fail))
1869                )
1870              (dcheck ((equal? (rhs eval-res) new-object) (!claim eval-res))
1871                (else (!fail))))))))))
1872
1873 (dmatch current-theorem
1874   ([[= _ _] (!tran' current-theorem new-equation))
1875   ((some-atom A) (dlet ((terms (children A))
1876                        (terms' (join (all-but-last terms) [new-object]))
1877                        (new-th (make-term (root A) terms')))
1878     (!rcong-rec current-theorem new-th)))
1879   ((if ant (= _ _)) (assume ant
1880     (!tran' (!mp current-theorem ant) new-equation)))
1881   ((if ant (bind A ((some-symbol f) (list-of _ _)))
1882    (dlet ((terms' (join (all-but-last (children A)) [new-object]))
1883          (new-th (make-term (root A) terms')))
1884      (assume ant
1885        (!rcong (!mp current-theorem ant) new-th))))))
1886
1887

```

```

1888 (define (first-iteration? n)
1889   (equal? n 0))
1890
1891 (define (show-info writer)
1892   (lambda (ct ot no di it)
1893     (seq (writer "current-theorem: " ct)
1894           (writer "old-term: " ot)
1895           (writer "new-object " no)
1896           (show ot no di (first-iteration? it)))))
1897
1898 # (define (infer p1 p2 rules direction)
1899 #   (dmatch direction
1900 #     (<=> (!get-equivalence* p1 p2 rules))
1901 #     (==> (!get-implication* p1 p2 rules))
1902 #     (<== (!get-implication* p2 p1 rules))
1903 #     ( _ (!fail (join "Error: " (val->string direction)
1904 #                       " found where <==>, ==>, or <== was expected."))))))
1905
1906 (define (infer p1 p2 rules direction)
1907   (dcheck ((binary-proc? direction) (dmatch (direction true false)
1908                                               ((iff true false) (!get-equivalence* p1 p2 rules))
1909                                               ((if true false) (!get-implication* p1 p2 rules))
1910                                               ((if false true) (!get-implication* p2 p1 rules))
1911                                               ( _ (!fail (join "Error: " (val->string direction)
1912                                                                 " found where <==>, ==>, or <== was expected."))))))
1913   (else (!fail (join "Error: " (val->string direction)
1914                       " found where <==>, ==>, or <== was expected."))))))
1915
1916 (define (infer p1 p2 rules direction)
1917   (dmatch direction
1918     (if (!get-implication* p1 p2 rules))
1919     (iff (!get-equivalence* p1 p2 rules))
1920     ( _ (dcheck ((binary-proc? direction) (dmatch (direction true false)
1921                                                     ((iff true false) (!get-equivalence* p1 p2 rules))
1922                                                     ((if true false) (!get-implication* p1 p2 rules))
1923                                                     ((if false true) (!get-implication* p2 p1 rules))
1924                                                     ( _ (!fail (join "Error: " (val->string direction)
1925                                                                 " found where <==>, ==>, or <== was expected."))))))
1926     (else (!fail (join "Error: " (val->string direction)
1927                           " found where <==>, ==>, or <== was expected.")))))))))
1928
1929 (define (implication-step current-theorem side new-object direction
1930         no-rules-given? rest get-rules iteration wv)
1931   (dlet ((rules (check (no-rules-given? (ab))
1932                         (else (make-list (head rest)))))
1933          (rule-cell (cell rules))
1934          ( _ (dcheck (no-rules-given? (!true-intro))
1935                      (else (!get-rules rules rule-cell))))
1936          (rules' (ref rule-cell))
1937          (rules'' (map rename' rules'))
1938          ([side new-object] (match (if side new-object)
1939                                    ((if (some-sent l) (some-sent r)) [l r])))
1940          (new-theorem (!map-multi-method decompose rules'
1941                                           (method (results)
1942                                             # Add the following if you want prop-taut to be included by default when no rules are specif
1943                                             (dlet ((results' (check (no-rules-given? (add prop-taut results))
1944                                                                    (else results)))))
1945                                             (!infer side new-object results' direction)))))
1946          ( _ (wv "current-theorem: " current-theorem))
1947          ( _ (wv "side: " side))
1948          ( _ (wv "new-object " new-object))
1949          ( _ (wv "direction: " direction))
1950          ( _ (wv "new-theorem " new-theorem))
1951          ([current-theorem' new-theorem'] (match (and current-theorem new-theorem)
1952                                                    ((and (some-sent p1) (some-sent p2)) [p1 p2])))
1953          ( _ (show-equiv side new-object direction (first-iteration? iteration)))
1954          ( _ (!sort-instance current-theorem current-theorem'))
1955          (!apply-tran current-theorem' new-theorem'))))
1956
1957

```

```

1958 (define (do-end current-theorem end-choice wv)
1959   (dmatch current-theorem
1960     ((bind th (iff p1 p2)) (dmatch end-choice
1961       ('last (!mp (!left-iff th) (!prove-components-harder p1)))
1962       ('first (!mp (!right-iff th) (!prove-components-harder p2)))
1963       (_ (!claim th))))
1964     ((bind th (if p1 _)) (dmatch end-choice
1965       ('last (dlet ((wv "th: " th))
1966         (_ (wv "" (join "\nWill try to prove components of: "
1967           (val->string p1)))))
1968         (!mp th (!prove-components-harder p1))))
1969       ('first (!mp th (!prove-components-harder p1)))
1970       (_ (!claim th))))
1971     (_ (!claim current-theorem))))
1972
1973 (define (get-all-rules generic-chain)
1974   (method (L list-cell)
1975     (dcheck ((for-some L chain-symbol?)
1976       (dlet ((th (!generic-chain L 'last))
1977         (_ (set! list-cell [th])))
1978         (!claim th)))
1979       (else (dlet ((L (set! list-cell L))
1980         (!true-intro))))))
1981
1982 (define (backward-implication-direction? dir)
1983   (&& (binary-proc? dir)
1984     (equal? (dir true false) (if false true))))
1985
1986
1987 (define (ordinal-string i)
1988   (let ((suffix (match (mod i 10)
1989     (1 (check ((equal? i 11) "th")
1990       (else "st"))
1991     (2 "nd")
1992     (3 "rd")
1993     (_ "th"))))
1994     (join (val->string i) suffix)))
1995
1996
1997 (define (dcatch' M1 M2)
1998   (dlet ((failed (cell false))
1999     (res (dtry (!M1)
2000       (dlet ((L (set! failed true))
2001         (!true-intro))))))
2002     (dmatch (ref failed)
2003       (true (!M2))
2004       (_ (!claim res)))))
2005
2006
2007 (define (generic-chain given-list end-choice)
2008   (dletrec ((writeln-val (lambda (x y) (print x "\n" (val->string y) "\n")))
2009     (writeln-val (lambda (x y) ()))
2010     (#(mprint (lambda (x) (print x)))
2011     (mprint (lambda (x) ()))
2012     (continue (lambda () ()))
2013     (_ (check ((in-rewriting-trace-mode) (seq (indent (plus (get-trace-level) 1))
2014       (print "Chaining:\n"))
2015       (else ())))
2016     (get-rules (get-all-rules generic-chain))
2017     (loop (method (current-theorem L backwards? iteration)
2018       (dmatch [current-theorem L]
2019         [(left right) (split [direction new-object] rest)]
2020         (dcheck ((equational? direction)
2021           (dlet ((no-rules-given? (|| (null? rest) (direction? (head rest))))
2022             (rest' (check (no-rules-given? rest) (else (tail rest))))
2023             (current-theorem' (!dcatch'
2024               (method () (!equational-step current-theorem left right)
2025                 (restrict-right-hand-side-sorts right)
2026                 (get-rules no-rules-given? iteration)
2027                 (method () (!proof-error (join "Equational chaining error

```

```

2028                                     " step of the chain, in going from
2029                                     "\nto:\n" (val->string new-object)
2030                                     (!loop current-theorem' rest' backwards? (plus iteration 1))))
2031 (else (dlet ((side (check ((identity? current-theorem) current-theorem)
2032                             (backwards? left)
2033                             (else right))))
2034         (finished? (null? rest))
2035         (no-rules-given? (|| finished? (direction? (head rest))))
2036         (rest' (check (no-rules-given? rest)
2037                       (else (tail rest))))
2038         (_ (mprint (join "\ncurrent-theorem: " (val->string current-theorem)
2039                          "\nside: " (val->string side) "\nnew-object: " (val->string new-object)
2040                          "\nno-rules-given?: " (val->string no-rules-given?)
2041                          "\nrest: " (val->string rest) "\niteration: " (val->string iteration)
2042                          "\n")
2043                (new-current-theorem (!dcatch'
2044                                     (method ()
2045                                     (!implication-step current-theorem side
2046                                     (restrict-right-hand-side-sorts side new-object)
2047                                     no-rules-given? rest get-rules iteration write!
2048                                     (method () (dlet ((previous (match (& (term? right)
2049                                     (true right)
2050                                     (_ (match current-theorem
2051                                     ((iff LBS RBS) RBS)
2052                                     (_ current-theorem))))))
2053                                     (!proof-error (join "Implicational chaining error: "
2054                                     " step of the chain, in going from "
2055                                     "\nto:\n" (val->string new-object)
2056                                     (backwards (|| backwards? (backward-implication-direction? direction))))
2057                                     (!loop new-current-theorem rest' backwards (plus iteration 1))))))
2058         (_ (!do-end current-theorem end-choice writeln-val))))))
2059 (dmatch given-list
2060 ((list-of first-object (bind rest (list-of dir _)))
2061  (dcheck ((equational? dir) (!loop (!reflex first-object) rest false 0))
2062           (else (dlet (_ ()) (!loop (!ref-equiv first-object) rest false 0))))))
2063 (define (chain-last L)
2064   (!generic-chain L 'last))
2065
2066 (define (chain-first L)
2067   (!generic-chain L 'first))
2068
2069 (define (chain L)
2070   (!generic-chain L 'none))
2071
2072 (define [chain-> chain<-] [chain-last chain-first])
2073 (define [c-> c<-] [chain-last chain-first])
2074
2075 define chain-transformer :=
2076   method (sentence-transformer L modifier)
2077     letrec {insert-given := lambda (x)
2078             match x {
2079               (some-sent _) => try {(sentence-transformer x) | x}
2080               | (some-method _) => x
2081               | (some-list L) => (map insert-given L)
2082               | _ => (sentence-transformer x)
2083             };
2084     insert-givens :=
2085       lambda (L)
2086         match L {
2087           (list-of direction (list-of y (list-of x rest))) =>
2088             (join [direction y (insert-given x)]
2089                  (insert-givens rest))
2090           | [] => []
2091         }
2092     match L {
2093       (list-of h rest) => let {L' := (add h (insert-givens rest))}
2094         (!generic-chain L' modifier)
2095     }
2096
2097

```

```

2098
2099 (define (left-side equivalence)
2100   (dmatch equivalence
2101     ((bind th (iff _ p2)) (!mp (!right-iff th) (!prove-components-of p2))))))
2102
2103 (define (right-side equivalence)
2104   (dmatch equivalence
2105     ((bind th (iff p1 _)) (!mp (!left-iff th) (!prove-components-of p1)))
2106     ((bind th (if p1 _)) (!mp th (!prove-components-of p1))))))
2107
2108 ##### Some more examples of new uses of chain:
2109 #
2110 ##### Example 1: Here p2 is obtainable from p1 via double
2111 ##### negation, as well as rewriting with Plus-zero-axiom
2112 ##### and Times-Commutativity:
2113 #
2114 # (define p1 (not (not (< (Plus ?x zero) (Times ?z1 ?z2)))))
2115 #
2116 # (define p2 (< ?x (Times ?z2 ?z1)))
2117 #
2118 # (!chain [p1 <==> p2 [bdn Plus-zero-axiom Times-Commutativity]])
2119 #
2120 ##### Example 2: Here p2 is obtainable from p1 by commuting
2121 ##### the conjuncts of p1, and also rewriting various subterms
2122 ##### in accordance with the commutativity of times and the
2123 ##### plus-zero axiom:
2124 #
2125 # (define p1 (or (not true) (and (< (Plus ?x zero) ?y)
2126 #                                (pos (Times ?z1 ?z2)))))
2127 #
2128 # (define p2 (or (not true) (and (pos (Times ?z2 ?z1))
2129 #                                (< ?x ?y))))
2130 #
2131 # (!chain [p1 <==> p2 [comm Plus-zero-axiom Times-Commutativity]])
2132 #
2133 ##### Example 3: Similar to example 1, but uni-directional:
2134 #
2135 # (define p1 (and (= ?foo ?goo)
2136 #                 (not (not (< (Plus ?x zero)
2137 #                                (Times ?z1 ?z2)))))
2138 #
2139 # (define p2 (< ?x (Times ?z2 ?z1)))
2140 #
2141 # (!chain [p1 ==> p2 [bdn right-and Plus-zero-axiom Times-Commutativity]])
2142 #
2143 ##### Example 4: Backward goal reduction:
2144 ##
2145 ## (!chain [?A <== (and ?A ?B) [left-and]
2146 ##           <== (and ?C (and ?A ?B)) [right-and]])
2147 ##
2148 ##### Example 5: Quantified propositions:
2149 ##
2150 ## (define p (forall ?x (and (= ?x (Plus ?foo ?goo))
2151 ##                           (= (Plus ?x zero) ?y))))
2152 ##
2153 ## (define q (forall ?w (and (= ?w (Plus ?goo ?foo))
2154 ##                           (= ?w ?y))))
2155 ##
2156 ## (!chain [p <==> q [Plus-Commutativity Plus-zero-axiom]])
2157 ## (!chain [p ==> q [Plus-Commutativity Plus-zero-axiom]])
2158 ## (!chain [p <== q [Plus-Commutativity Plus-zero-axiom]])
2159 ##
2160 ## These examples would work with 'exists' in place of 'forall'
2161 ## as well.
2162 #####
2163
2164 (define (test-proof p)
2165   (dlet ((dummy (write p)))
2166     (!true-intro)))
2167

```

```

2168 (define (test-proof p)
2169   (seq
2170     (write p)
2171     (print "Proved!")))
2172
2173 (define (spec-proposition terms)
2174   (dlet ((spec-proposition (!uspec* proposition terms)))
2175     (dmatch spec-proposition
2176       ((if _P _Q)
2177        (!mp spec-proposition
2178         _P))
2179       ((iff _P _Q)
2180        (!mp (!left-iff spec-proposition)
2181         _P))
2182       (_ (!claim spec-proposition))))))
2183
2184 (define (spec-right-proposition terms)
2185   (dlet ((spec-proposition (!uspec* proposition terms)))
2186     (dmatch spec-proposition
2187       ((iff _P _Q)
2188        (!mp (!right-iff spec-proposition)
2189         _Q))))))
2190
2191 # New names for same methods (old spec names are kept temporarily
2192 # for backward compatibility):
2193
2194 ## It turns out that the method instance was similar to fire, in util.ath.
2195 ## I think the name 'instance' is more appropriate than 'fire', so I have
2196 ## been using 'instance' instead of 'fire', having defined 'instance' as
2197 ## 'fire' in util.ath (since 'fire' is a slight generalization of 'instance').
2198
2199 # (define (instance-proposition terms)
2200 #   (dlet ((instance-proposition (!uspec* proposition terms)))
2201 #     (dmatch instance-proposition
2202 #       ((if _P _Q)
2203 #        (!mp instance-proposition
2204 #         _P))
2205 #       ((iff _P _Q)
2206 #        (!mp (!left-iff instance-proposition)
2207 #         _P))
2208 #       (_ (!claim instance-proposition))))))
2209
2210 (define (right-instance-proposition terms)
2211   (dlet ((instance-proposition (!uspec* proposition terms)))
2212     (dmatch instance-proposition
2213       ((iff _P _Q)
2214        (!mp (!right-iff instance-proposition)
2215         _Q))))))
2216
2217 (define (left-instance-proposition terms)
2218   (dlet ((instance-proposition (!uspec* proposition terms)))
2219     (dmatch instance-proposition
2220       ((iff _P _Q)
2221        (!mp (!left-iff instance-proposition)
2222         _P))))))
2223
2224 # A similar method, for use where you want to be explicit
2225 # about the assumption used to discharge the antecedent of the
2226 # specialized implication or equivalence.
2227
2228 (define (mp-instance-proposition terms assumption)
2229   (dlet ((instance-proposition (!uspec* proposition terms)))
2230     (dmatch instance-proposition
2231       ((if _P _Q)
2232        (!mp instance-proposition
2233         (dcheck ((equal? _P assumption) (!claim _P))
2234          (else (dlet ((dummy
2235            (write "Error: Failed application of mp-instance, due to:"))
2236              (!claim assumption))))))
2237       ((iff _P _Q)

```



```

2238      (!mp (!left-iff instance-proposition)
2239        (dcheck ((equal? _P assumption) (!claim _P))
2240          (else (dlet ((dummy
2241            (write "Error: Failed application of mp-instance, due to:"))
2242              (!claim assumption))))))
2243      (_ (!claim instance-proposition))))))
2244
2245
2246 # A method that subsumes uspec and uspec*
2247
2248 (define (special-case P terms)
2249   (dmatch terms
2250     ([ (!claim P))
2251       ((list-of t more-terms) (!special-case (!uspec P t) more-terms))
2252       (_ (!uspec P terms))))))
2253
2254 # Methods for which the first argument is the desired special case,
2255 # or an antecedent or consequent of the desired special case,
2256 # and the second argument is the property to be specialized. The
2257 # instantiation is deduced by unification.
2258
2259 (define (left-extract P1 P)
2260   (dmatch (universal-quantifierless P)
2261     ((iff _A _B) (dlet ((subst (unify-props _B P1))
2262       (term-list (universal-quantifiers P)))
2263       (!mp (!left-iff (!special-case P (subst term-list))
2264         (!prove-components-of (subst _A)))))))
2265
2266 (define (right-extract P1 P)
2267   (dmatch (universal-quantifierless P)
2268     ((iff _A _B) (dlet ((subst (unify-props _A P1))
2269       (term-list (universal-quantifiers P)))
2270       (!mp (!right-iff (!special-case P (subst term-list))
2271         (!prove-components-of (subst _B)))))))
2272
2273 (define (extract P1 P)
2274   (conclude P1
2275     (dmatch (universal-quantifierless P)
2276       ((if _A _B)
2277         (dlet ((subst (match-props P1 _B))
2278           (dcheck ((negate (equal? subst false))
2279             (dlet ((term-list (universal-quantifiers P)))
2280               (!mp (!special-case P (subst term-list))
2281                 (!prove-components-of (subst _A))))))))))
2282       (_P
2283         (dtry
2284           (!left-extract P1 P)
2285           (!right-extract P1 P)
2286           (dlet ((subst (match-props P1 _P)))
2287             (dcheck ((negate (equal? subst false))
2288               (!special-case P (subst (universal-quantifiers P))))))))))
2289
2290 #####
2291
2292 (define tracing (cell false))
2293
2294 (define (conclude-old prop)
2295   (seq
2296     (check ((ref tracing)
2297       (seq
2298         (set! level (plus (ref level) 1))
2299         (indent (ref level))
2300         (print "Proving at level ")
2301         (write-val (ref level))
2302         (print ":\n")
2303         (indent (ref level))
2304         (write-val prop)
2305         (print "\n"))
2306       (else ()))
2307     (method (proof)

```

```

2308     (dseq
2309       (prop BY (!claim proof))
2310       (dlet ((dummy
2311         (check ((ref tracing)
2312           (seq
2313             (indent (ref level))
2314             (print "Done at level ")
2315             (write-val (ref level))
2316             (print "\n")
2317             (set! level (minus (ref level) 1))))
2318           (else ())))))
2319       (!claim prop))))))
2320
2321 # To turn on tracing of concludes, enter
2322 #   (set-debug-mode "conclude")
2323 # To turn it off, enter
2324 #   (set-debug-mode "off")
2325
2326 # To turn on tracing of rewrite rule applications (and concludes):
2327 #   (set-debug-mode "rewriting")
2328 # To turn it off, enter
2329 #   (set-debug-mode "off")
2330 # or, if you still want concludes traced:
2331 #   (set-debug-mode "conclude")
2332
2333 # For legacy code that uses deduce:
2334 (define (trace)
2335   (seq
2336     (set! tracing true)
2337     (set! level 0)))
2338 # Also at top-level do
2339 #   (set-debug-mode "rewriting")
2340
2341 # The following redefines rel-cong so that it
2342 # doesn't need to have equations of the form (= x x) in the
2343 # assumption base (to use the built-in rel-cong you must
2344 # do (!equality x x) for terms x that appear unchanged in
2345 # corresponding positions of s-terms and t-terms)
2346
2347 (define (rel-cong P s-terms t-terms)
2348   (dletrec ((do-args (method (s-terms t-terms theorem n)
2349     (dmatch [s-terms t-terms]
2350       ([[] []] (!claim theorem))
2351       ([ (list-of s more-s) (list-of t more-t) ]
2352         (dlet ((new-theorem
2353           (!pos-substitute-equals s theorem [n] t)))
2354           (!do-args more-s more-t new-theorem (plus n 1)))))))
2355     (!do-args s-terms t-terms P 1)))
2356
2357 # Using this rel-cong, we define (rewrite P1 P2) where P1 and P2
2358 # are predicates with the same predicate symbols and same number of
2359 # arguments, and P1 is in the assumption base. If for each argument
2360 # position with corresponding terms s and t the equation (= s t) is in the
2361 # assumption base, then P2 is entered as a theorem in the assumption base.
2362
2363 (define (rewrite P1 P2)
2364   (dletrec
2365     ((loop (method (L1 L2 theorem n)
2366       (dmatch [L1 L2]
2367         ([ (list-of x1 rest1) (list-of x2 rest2) ]
2368           (dcheck ((negate (equal? x1 x2))
2369             (dlet ((new-theorem
2370               (!pos-substitute-equals x1 theorem [n] x2)))
2371               (!loop rest1 rest2 new-theorem (plus n 1))))
2372           (else (!loop rest1 rest2 theorem (plus n 1))))))
2373         (_ (!claim theorem))))))
2374     (!loop (children P1) (children P2) P1 1)))
2375
2376 (define cl chain-last)
2377

```

```

2378 (define (dt=? s t)
2379   (dmatch [s t]
2380     ([((some-symbol f1) (some-list terms1))
2381       ((some-symbol f2) (some-list terms2))])
2382     (dcheck ((for-each [f1 f2] constructor?)
2383               (dcheck (equal? f1 f2)
2384                         (dletrec ((loop
2385                                   (method (L)
2386                                     (dmatch L
2387                                       ([] (!fcong (= s t)))
2388                                       ((list-of term-pair rest)
2389                                         (dmatch (!dt=? (first term-pair) (second term-pair))
2390                                           ((= _ _) (!loop rest))
2391                                           (res (!chain-last [res ==> (s /= t) [(datatype-axioms (sort-of s))]])))
2392                                         (!loop (zip terms1 terms2))))
2393                                       (else (dlet ((s-sort (sort-of s))
2394                                                    (dt-axioms (datatype-axioms s-sort))
2395                                                    (_ (check ((hold? dt-axioms) ())
2396                                                                (else (print "\nWarning: the datatype axioms for " s-sort
2397                                                                "are not in the assumption base.\n")))))
2398                                         (dtry (!chain-last [true ==> (s /= t) [dt-axioms]]
2399                                                  (!chain-last [true ==> (t /= s) [dt-axioms]
2400                                                                ==> (s /= t) [unequal-sym]]))))))
2401               ((&& (real-numeral? s) (real-numeral? t)) (!real-comp s t))
2402               ((&& (meta-id? s) (meta-id? t)) (!id-comp s t))))))
2403
2404 # The dt-comp-method-cell is a cell defined in util.ath right before the
2405 # definition of prove-components-of and uses whatever method is inside that
2406 # cell to prove equalities and inequalities. By setting the contents of
2407 # that cell to dt=?, we make dt=? the default method for proving equalities
2408 # and inequalities in antecedents of conditional equations.
2409
2410 (define (dt-comp p)
2411   (dmatch p
2412     ((= (some-term s) (some-term t))
2413      (dcheck ((&& (super-canonical? s) (super-canonical? t)) (!dt=? s t))
2414                ((all-ground? [s t]) (!chain [s = t]))
2415                (else (!dt=? s t))))
2416     ((not (= (some-term s) (some-term t))) (!dt=? s t))
2417     ((not (some-term s)) (dcheck ((ground? s) (!identity->atom (!chain [s = false]))))))
2418
2419
2420 (set! dt-comp-method-cell
2421       (method (p)
2422               (dmatch p
2423                 ((= (some-term l) (some-term r)) (!dt=? l r)))))
2424
2425 (set! dt-comp-method-cell dt-comp)
2426
2427
2428 (define (dt-comp' _ p)
2429   (!dt-comp p))
2430
2431 (define dt-comps [dt-comp dt-comp'])
2432
2433 #;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2434 #;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2435 #
2436 #
2437 #
2438 #
2439 (define decompose-equation'
2440   (lambda (eqn)
2441     (match (rename eqn)
2442       ((forall (some-list uvars) (if guard (= pattern res))) [pattern guard res])
2443       ((forall (some-list uvars) (= pattern res)) [pattern () res]))))
2444
2445 (define pat-of first)
2446 (define guard-of second)
2447 (define res-of third)

```

```

2448
2449 (define (unifiable x y)
2450   (try (match (unify* x y)
2451     ((some-sub _) true)
2452     (_ false))
2453     false))
2454
2455 (define (make-equivalence-classes triples)
2456   (letrec ((loop (lambda (remaining-triples classes-so-far)
2457     (match remaining-triples
2458       ([] (rev classes-so-far))
2459       ((list-of (as triple [pat guard res]) more)
2460        (match (for-some' classes-so-far
2461          (lambda (triple-list)
2462            (unifiable pat (pat-of (first triple-list))))))
2463          ([classes-1 triple-list classes-2]
2464           (let ((classes' (join classes-1 [(join triple-list [triple]]) classes-2)))
2465             (loop more classes')))
2466          (_ (loop more (add [triple] classes-so-far))))))))))
2467   (loop triples []))
2468
2469 (define (simplify-guard triple previous-triples)
2470   (match triple
2471     ([_ () _] triple)
2472     ([pat guard res]
2473      (let ((guard-conjuncts (get-conjuncts-recursive guard))
2474            (structural-inequalities (conjuncts-of (diff* pat (map pat-of previous-triples))))
2475            (guard-conjuncts' (filter guard-conjuncts (lambda (c) (negate (member? c structural-inequalities)))))
2476            ([pat fgc res] (letrec ((loop (lambda (prev-triples final-triple)
2477              (match [prev-triples final-triple]
2478                ([[] _] final-triple)
2479                ((list-of [_ () _] more) _) (loop more final-triple))
2480                ((list-of [pat0 guard0 res0] more) [pat conjuncts res])
2481                 (match (alpha-variants? pat0 pat)
2482                   ((some-sub sub)
2483                    (let ((guard0-conjuncts' (sub (conjuncts-of guard0)))
2484                          ([pat' conjuncts' res'] [(sub pat) (sub conjuncts) (sub
2485                          (conjuncts" (filter-out conjuncts'
2486                          (lambda (c)
2487                            (member? (complement c) guard
2488                              (loop more [pat' conjuncts" res'])))
2489                          (_ (loop more final-triple))))))
2490                  (loop previous-triples [pat guard-conjuncts' res]))))
2491            [pat (and* fgc) res])))))
2492
2493
2494
2495 # (define (glean-all-sure-negations previous-triples pat guard res)
2496 #   (letrec ((loop (lambda (previous-triples results)
2497 #     (match previous-triples
2498 #       ([[] results)
2499 #       ((list-of [pat0 guard0 res0] more)
2500 #        (match (non-linear-instance? pat0 pat)
2501 #          ((some-sub sub) (match (get-non-linear-constraints pat0)
2502 #            ([c] (sub c)
2503
2504 (define (linearize-pattern pat)
2505   (let ((T (table 10))
2506         (T' (table 10))
2507         (already-seen? (lambda (v) (try (seq (table-lookup T v) true) false))))
2508     (letrec ((loop (lambda (t)
2509       (match t
2510         ((some-var _) (let ((new-var (lhs (= (fresh-var) t)))
2511           (_ (table-add T' [t --> new-var]))
2512           (constraints (check ((already-seen? t)
2513             (let ((old-var (table-lookup T t))
2514               (_ (table-add T [t --> new-var]))
2515               [(= old-var new-var]))
2516             (else (let ((_ (table-add T [t --> new-var]))
2517               []))))))

```

```

2518         [new-var constraints]))
2519         (((some-symbol f) (some-list args)) (let (([args' constraints] (loop* args [[] []])))
2520         [(make-term f args') constraints]])))
2521     (loop* (lambda (terms res)
2522     (match [terms res]
2523     ([[] [terms' constraints]] [(rev terms') (rev constraints)])
2524     ([([list-of s more) [terms' constraints]]
2525     (let (([s' constraints'] (loop s)))
2526     (loop* more [(add s' terms') (join constraints' constraints)]))))))
2527     (let ((res (loop pat))
2528     (sub (make-sub (table->list T))))
2529     [res sub])))
2530
2531 (define (make-and props)
2532   (match props
2533     ([p] p)
2534     ([] true)
2535     (_ (and props))))
2536
2537 (define (augment-guards guard equalities)
2538   (match [equalities guard]
2539     ([[] _] guard)
2540     ([_ ()] (make-and equalities))
2541     (_ (and (join (get-conjuncts guard) equalities)))))
2542
2543 (define (apply-sub sub guard)
2544   (check ((equal? guard ()) guard)
2545     (else (sub guard))))
2546
2547 (define (linearize triple previous-triples)
2548   (match triple
2549     ([pat guard res] (let (([pat' equalities] sub] (linearize-pattern pat))
2550     #      (_ (print "\npat: " pat "\nguard: " guard "\nres: " res "\npat': " pat' "\nequalities: " e
2551     (_ ()))
2552     [pat' (apply-sub sub (augment-guards guard equalities)) (sub res)]))))))
2553
2554 # (define (linearize triple previous-triples)
2555 #   triple)
2556
2557 (define (process-equivalence-class class)
2558   (letrec ((loop (lambda (remaining-triples processed-triples results)
2559   (match remaining-triples
2560   ([[] (rev results))
2561   ((list-of triple more-triples)
2562   (let ((triple' (linearize triple processed-triples)))
2563   (loop more-triples (add triple processed-triples)
2564   (add triple' results)))))))
2565   (loop class [] [])))
2566
2567
2568 (define size' (lambda (x)
2569   (match x
2570     ([()] 0)
2571     (_ (size x)))))
2572
2573
2574 (define sort-class (lambda (class)
2575   (merge-sort class
2576     (lambda (triple1 triple2)
2577     (less? (size' (guard-of triple1)) (size' (guard-of triple2)))))))
2578
2579 (define (analyze equations)
2580   (let ((all-triples (map decompose-equation' equations))
2581   (list-of-classes (make-equivalence-classes all-triples))
2582   (list-of-classes (map sort-class list-of-classes)))
2583   (map sort-class (map process-equivalence-class list-of-classes))))
2584
2585 (define triple->equation (lambda (triple)
2586   (match triple
2587     ([pat guard res]

```

```

2588             (let ((eqn (check ((member? guard [()] true)) (= pat res))
2589                     (else (if guard (= pat res))))))
2590             (close eqn))))))
2591
2592 (define (all-vars? L) (for-each L var?))
2593
2594
2595
2596 (define (all-constructors? t)
2597   (for-each (syms t) (lambda (f) (|| (constructor? f) (numeral? f) (meta-id? f)))))
2598
2599 (define (all-constructors* L)
2600   (&&* (map all-constructors? L)))
2601
2602 (define (guards-should-be-matched? guard-res-list uvars)
2603   (try
2604     (let ((all-identical (lambda (L) (equal? (length (rd L)) 1)))
2605           (guards (filter-out (map first guard-res-list) (lambda (x) (member? x [true ()]))))
2606           (guard-lhsides (map lhs guards))
2607           (guard-rhsides (map rhs guards))
2608           (results (map second guard-res-list))
2609           (cond1 (all-identical guard-lhsides))
2610           (cond2 (all-constructors* guard-rhsides))
2611           (cond3 (for-some guard-rhsides
2612                         (lambda (rhs)
2613                           (for-some (vars rhs) (lambda (v) (negate (member? v uvars))))))))
2613       (&& cond1 cond2 cond3))
2614     (seq # (print "\nFailed try...\n")
2615           false)))
2616
2617
2618 (define (rhs-vars guard)
2619   (match guard
2620     ((= _ (some-term t)) (get-vars-manual t))
2621     (_ [])))
2622
2623 (define (get-extra-vars guard-res-list uvars)
2624   (let ((guards (map first guard-res-list))
2625         (guard-rhside-vars (rd (flatten (map rhs-vars guards)))))
2626     (list-diff guard-rhside-vars uvars)))
2627
2628
2629 (define (split-point L1 L2)
2630   (match [L1 L2]
2631     ([ (list-of m rest1) (list-of m rest2)] (split-point rest1 rest2))
2632     ([ (list-of m rest) _] L1)
2633     (_ [])))
2634
2635 (define (get-eval-proc-name-generic f mod-path)
2636   (let ((long-name (get-eval-proc-name-1 f))
2637         (toks (tokenize long-name "."))
2638         ([mods epc] [(all-but-last toks) (last toks)])
2639         (mods' (split-point mods mod-path)))
2640     (match mods'
2641       ([] epc)
2642       (_ (join (separate mods' ".") "." epc)))))
2643
2644 (define (get-reduce-proc-name-generic f mod-path)
2645   (join (get-eval-proc-name-generic f mod-path) (standard-reduce-proc-name-suffix)))
2646
2647 (define [translate-symbol compile-term compile-guard compile-terms]
2648   (letrec
2649     ((translate-symbol (lambda (g)
2650                          (let ((long-name (get-eval-proc-name-1 g))
2651                                #(_ (print "\nMOD PATH: " mod-path))
2652                                (_ ()))
2653                            (check ((constructor? g)
2654                                    (symbol->string g)
2655                                    ((prefix? (all-but-last (split-string long-name `.) mod-path)
2656                                                 (get-eval-proc-name g))
2657                                    (else (get-eval-proc-name-generic g mod-path)))))))

```

```

2658 (compile-term (lambda (term mapping translate-symbol)
2659   (let (#(_ (print "\nAbout to translate this term: " term))
2660     (_ ()))
2661     (letrec ((loop (lambda (t)
2662       (match t
2663         ((some-var x) (apply-map mapping x))
2664         (((some-symbol g) []) (val->string g))
2665         (((some-symbol g) (some-list args))
2666           (check (equal? g =)
2667             (join lp (translate-symbol g) blank (separate (map loop args) l
2668             (else (join lp (translate-symbol g) blank (separate (map loop ar
2669         (loop term))))))
2670 (compile-guard (lambda (guard mapping translate-symbol)
2671   (letrec ((loop (lambda (g)
2672     (match g
2673       ((some-term _) (compile-term g mapping translate-symbol))
2674       ((not g') (join lp "negate " (loop g') rp))
2675       ((and g1 g2) (join lp "&& " (loop g1) blank (loop g2) rp))
2676       ((or g1 g2) (join lp "|| " (loop g1) blank (loop g2) rp))))))
2677     (loop guard))))
2678 (compile-terms (lambda (terms mapping translate-symbol)
2679   (map (lambda (t) (compile-term t mapping translate-symbol)) terms))))
2680 [translate-symbol compile-term compile-guard compile-terms]))
2681
2682 (define (constant? t)
2683   (&& (symbol? t) (equal? (arity-of t) 0)))
2684
2685 (define (poly-constant? t)
2686   (&& (constant? t) (poly? t)))
2687
2688 (define (mono? t) (negate (poly? t)))
2689
2690 (define (compile-entry equation mod-path)
2691   (let (
2692     #(_ (print "\nCalling compile-entry on equation: " equation))
2693     (translate-symbol (lambda (g)
2694       (let ((long-name (get-eval-proc-name-1 g))
2695         #(_ (print "\nMOD PATH: " mod-path))
2696         (_ ()))
2697       (check ((constructor? g)
2698         (symbol->string g))
2699         ((prefix? (all-but-last (split-string long-name `.) mod-path)
2700         (get-eval-proc-name g))
2701         (else (get-eval-proc-name-generic g mod-path))))))
2702     (compile-term (lambda (term mapping translate-symbol)
2703       (let (#(_ (print "\nAbout to translate this term: " term))
2704         (_ ()))
2705         (letrec ((loop (lambda (t)
2706           (match t
2707             ((some-var x) (apply-map mapping x))
2708             (((some-symbol g) []) (val->string g))
2709             (((some-symbol g) (some-list args))
2710               (check (equal? g =)
2711                 (join lp (translate-symbol g) blank (separate (map loop args) l
2712             (else (join lp (translate-symbol g) blank (separate (map loop ar
2713           (loop term))))))
2714     (compile-guard (lambda (guard mapping translate-symbol)
2715       (letrec ((loop (lambda (g)
2716         (match g
2717           ((some-term _) (compile-term g mapping translate-symbol))
2718           ((not g') (join lp "negate " (loop g') rp))
2719           ((and g1 g2) (join lp "&& " (loop g1) blank (loop g2) rp))
2720           ((or g1 g2) (join lp "|| " (loop g1) blank (loop g2) rp))))))
2721         (loop guard))))
2722     (compile-terms (lambda (terms mapping translate-symbol)
2723       (map (lambda (t) (compile-term t mapping translate-symbol)) terms))))
2724 (match equation
2725   ([uvars (as pattern (f (some-list args))) guard-res-list]
2726     (let ((f-name (symbol->string f))
2727       (count (length guard-res-list))

```

```

2728     (new-vars (map (lambda (i)
2729                     (join "x" (val->string i)))
2730                     (from-to 1 (length uvars))))
2731     (input-vars (map (lambda (i)
2732                       (join "t" (val->string i)))
2733                       (from-to 1 (length uvars))))
2734     (mapping (extend empty-map (list-zip uvars new-vars)))
2735     (input-mapping (extend empty-map (list-zip uvars input-vars)))
2736     (are-all-vars (all-vars? args))
2737     (mapping (check (are-all-vars input-mapping)
2738                     (else mapping)))
2739     (pattern-translation (check (are-all-vars "_")
2740                                  (else (join lb (separate (compile-terms args mapping symbol->string) blank) rp)))
2741     (translate-clause (lambda (guard-res-pair i)
2742                        (match guard-res-pair
2743                          ([guard res] (let ((cg (check ((&& (equal? guard true) (equal? i count)) "else")
2744                                                         (else (compile-guard guard mapping translate-symbol))
2745                                                         #(_ (print "\nCalling compile-term on this arg: " res " and this
2746                                                         (fourth-result (compile-term res mapping translate-symbol))
2747                                                         #(_ (print "\nFourth result: " fourth-result))
2748                                                         _ ()))
2749                                (join lp cg blank (compile-term res mapping translate-symbol) rp))))))
2750     (check
2751      ((negate (guards-should-be-matched? guard-res-list uvars))
2752       (let ((check-clauses (separate (map-with-index translate-clause guard-res-list) newline))
2753             (check-translation (join lp "check " check-clauses rp)))
2754         (join lp pattern-translation blank check-translation rp)))
2755      (else (let ((extra-vars (get-extra-vars guard-res-list uvars))
2756                  (compile-pattern (lambda (p mapping ts)
2757                                     (compile-term p mapping ts)))
2758                  (mapping' (extend mapping (list-zip extra-vars
2759                                                         (map (lambda (i)
2760                                                             (join "p" (val->string i)))
2761                                                             (from-to 1 (length extra-vars)))))))
2762              (translate-match-clause
2763               (lambda (guard-res-pair i)
2764                 (match guard-res-pair
2765                   ([guard res] (let ((cg (check ((&& (equal? guard true) (equal? i count)) "_")
2766                                                         (else (compile-pattern (rhs guard) mapping' translate-symbol)
2767                                                         (join lp cg blank (compile-term res mapping' translate-symbol) rp))))))
2768                   (discriminant (compile-term (lhs (first (first guard-res-list))) mapping' translate-symbol))
2769                   (match-clauses (separate (map-with-index translate-match-clause guard-res-list) newline))
2770                   (match-translation (join lp "match " discriminant "\n"
2771                                             match-clauses rp)))
2772              (join lp pattern-translation blank match-translation rp))))))
2773    ((forall (some-list uvars)
2774             (= (f (some-list args)) rhs))
2775     (let ((f-name (symbol->string f))
2776           (new-vars (map (lambda (i)
2777                           (join "x" (val->string i)))
2778                           (from-to 1 (length uvars))))
2779           (mapping (extend empty-map (zip uvars new-vars)))
2780           (lhs' (join lb (separate (compile-terms args mapping symbol->string) blank) rb))
2781           (rhs' (compile-term rhs mapping translate-symbol)))
2782      (join lp lhs' blank rhs' rp))
2783    ((forall (some-list uvars)
2784             (if antecedent (= (f (some-list args)) rhs)))
2785     (let ((f-name (symbol->string f))
2786           (new-vars (map (lambda (i)
2787                           (join "x" (val->string i)))
2788                           (from-to 1 (length uvars))))
2789           (mapping (extend empty-map (zip uvars new-vars)))
2790           (input-vars (map (lambda (i)
2791                             (join "t" (val->string i)))
2792                             (from-to 1 (length uvars))))
2793           (input-mapping (extend empty-map (zip uvars input-vars)))
2794           (are-all-vars (all-vars? args))
2795           (mapping (check (are-all-vars input-mapping)
2796                           (else mapping)))
2797           (lhs' (check (are-all-vars "_")

```



```

2798         (else (join lb (separate (compile-terms args mapping symbol->string) blank) rb))))
2799         (guard (compile-guard antecedent mapping translate-symbol))
2800         (lhs'' (join lp lhs' " where " guard rp))
2801         (rhs' (compile-term rhs mapping translate-symbol)))
2802     (join lp lhs'' blank rhs' rp))))))
2803
2804 (primitive-method (d+' s t)
2805   (let ((sym (string->symbol "+"))
2806         (rator (evaluate (get-eval-proc-name sym)))
2807         (res (rator s t)))
2808     (= (sym s t) res)))
2809
2810 (define (d+' s t)
2811   (!force (let ((sym (string->symbol "+"))
2812                 (rator (evaluate (get-eval-proc-name sym)))
2813                 (res (rator s t)))
2814             (= (sym s t) res))))
2815
2816 (primitive-method (d-' s t)
2817   (let ((sym (string->symbol "-"))
2818         (rator (evaluate (get-eval-proc-name sym)))
2819         (res (rator s t)))
2820     (= (sym s t) res)))
2821
2822 (define (d-' s t)
2823   (!force
2824     (let ((sym (string->symbol "-"))
2825           (rator (evaluate (get-eval-proc-name sym)))
2826           (res (rator s t)))
2827       (= (sym s t) res))))
2828
2829 (define (ded-unary-minus s)
2830   (!force
2831     (let ((sym (string->symbol "-"))
2832           (rator (evaluate (get-eval-proc-name sym)))
2833           (= (sym s) (rator s))))))
2834
2835 (primitive-method (d*' s t)
2836   (let ((sym (string->symbol "*"))
2837         (rator (evaluate (get-eval-proc-name sym)))
2838         (res (rator s t)))
2839     (= (sym s t) res)))
2840
2841 (define (d*' s t)
2842   (!force (let ((sym (string->symbol "*"))
2843                 (rator (evaluate (get-eval-proc-name sym)))
2844                 (res (rator s t)))
2845             (= (sym s t) res))))
2846
2847 (primitive-method (d/' s t)
2848   (let ((sym (string->symbol "/"))
2849         (rator (evaluate (get-eval-proc-name sym)))
2850         (res (rator s t)))
2851     (= (sym s t) res)))
2852
2853 (define (d/' s t)
2854   (!force (let ((sym (string->symbol "/"))
2855                 (rator (evaluate (get-eval-proc-name sym)))
2856                 (res (rator s t)))
2857             (= (sym s t) res))))
2858
2859 (primitive-method (d=' s t)
2860   (match ((evaluate (get-eval-proc-name (string->symbol "="))) s t)
2861     (true (= (= s t) true))
2862     (_ (= (= s t) false))))
2863
2864 (define (d=' s t)
2865   (!force (match ((evaluate (get-eval-proc-name (string->symbol "="))) s t)
2866                 (true (= (= s t) true))
2867                 (_ (= (= s t) false))))))

```

```

2868
2869 (primitive-method (d<' s t)
2870   (let ((sym (string->symbol "<")))
2871     (match ((evaluate (get-eval-proc-name sym)) s t)
2872       (true (= (sym s t) true))
2873       (_ (= (sym s t) false)))))
2874
2875 (define (d<' s t)
2876   (!force (let ((sym (string->symbol "<")))
2877     (match ((evaluate (get-eval-proc-name sym)) s t)
2878       (true (= (sym s t) true))
2879       (_ (= (sym s t) false)))))
2880
2881 (primitive-method (d<=' s t)
2882   (let ((sym (string->symbol "<=")))
2883     (match ((evaluate (get-eval-proc-name sym)) s t)
2884       (true (= (sym s t) true))
2885       (_ (= (sym s t) false)))))
2886
2887 (define (d<=' s t)
2888   (!force (let ((sym (string->symbol "<=")))
2889     (match ((evaluate (get-eval-proc-name sym)) s t)
2890       (true (= (sym s t) true))
2891       (_ (= (sym s t) false)))))
2892
2893 (primitive-method (d>=' s t)
2894   (let ((sym (string->symbol ">=")))
2895     (match ((evaluate (get-eval-proc-name sym)) s t)
2896       (true (= (sym s t) true))
2897       (_ (= (sym s t) false)))))
2898
2899 (define (d>=' s t)
2900   (!force (let ((sym (string->symbol ">=")))
2901     (match ((evaluate (get-eval-proc-name sym)) s t)
2902       (true (= (sym s t) true))
2903       (_ (= (sym s t) false)))))
2904
2905 (primitive-method (d>' s t)
2906   (let ((sym (string->symbol ">")))
2907     (match ((evaluate (get-eval-proc-name sym)) s t)
2908       (true (= (sym s t) true))
2909       (_ (= (sym s t) false)))))
2910
2911 (define (d>' s t)
2912   (!force (let ((sym (string->symbol ">")))
2913     (match ((evaluate (get-eval-proc-name sym)) s t)
2914       (true (= (sym s t) true))
2915       (_ (= (sym s t) false)))))
2916
2917 ## stran:
2918
2919 (define (stran eq1 eq2)
2920   (dtry (!tran eq1 eq2)
2921     (dmatch [eq1 eq2]
2922       ([ (= l1 r1) (= l2 r2)]
2923         (dcheck ((sort-instance? r1 l2)
2924           (dlet ((th (!sort-instance eq2 (= r1 r2)))
2925             (!tran eq1 th)))))))
2926
2927
2928 (define (deductive-version str)
2929   (match (rev str)
2930     ((split L1 (as L2 (list-of \. rest))) (join (rev L2) "d" (rev L1)))
2931     (_ (join "d" str)))
2932
2933 (define (unary-minus? f arity)
2934   (&& (equal? (symbol->string f) "-")
2935     (equal? arity 1)))
2936
2937 (define (deval0 t)

```

```

2938 (dmatch t
2939   ((some-var _) (!reflex t))
2940   ((some-symbol f) []) (!reflex t))
2941   ((bind term ((some-symbol f) (as args (list-of t more))))
2942     (dcheck ((&& (negate (free-constructor? f)) (unequal? (fsd f) ()))
2943       (dtry
2944         (!map-method deval0 args
2945           (method (eqns)
2946             (dlet ((rhs-list (map rhs eqns))
2947               (rhs' (make-term f rhs-list))
2948               (th1 (= term rhs')))
2949               (_ (!fcong th1))
2950               (f-method (check ((unary-minus? f (length args)) ded-unary-minus)
2951                 (else (try (evaluate (deductive-version (get-eval-proc-name-1 f)))
2952                   (evaluate (join "d" (get-eval-proc-name f)))))))
2953               ((as th2 (= 1 r)) (!app-method f-method rhs-list))
2954               (th (!stran th1 th2)))
2955               (dcheck ((equal? rhs' r) (!claim th2))
2956                 (else (dlet ((th3 (!deval0 r))
2957                   (!stran th th3)))))))
2958         (!map-method deval0 args
2959           (method (eqns)
2960             (dlet ((rhs-list (map rhs eqns))
2961               (term' (make-term f rhs-list))
2962               (!fcong (= term term')))))
2963         (else (!map-method deval0 args
2964           (method (eqns)
2965             (dlet ((rhs-list (map rhs eqns))
2966               (rhs' (make-term f rhs-list))
2967               (th (= term rhs')))
2968               (!fcong th)))))))
2969
2970 (define (deval t)
2971   (dlet ((res (dtry (!deval0 t) (!true-intro))))
2972     (dmatch res
2973       (true (!proof-error (join "\nUnable to reduce the term:\n" (val->string t) "\nto a normal form.\n")))
2974       (_ (!claim res))))
2975
2976 (set! deval-cell deval)
2977
2978 (define (evaluate-guard guard)
2979   (dmatch guard
2980     ((some-symbol pred) (some-list args))
2981     (dmatch (!deval guard)
2982       ((as res (= _ true)) (!identity->atom res))
2983       ((as res (= _ false)) (!identity->atom res))))
2984   ((not p) (dmatch (!evaluate-guard p)
2985     ((val-of p) (!by-contradiction (not (not p))
2986       (assume (not p)
2987         (!absurd p (not p)))))
2988     ((as res (not (val-of p))) (!claim res))))
2989   ((and p1 p2) (dmatch (!evaluate-guard p1)
2990     ((val-of p1) (dmatch (!evaluate-guard p2)
2991       ((val-of p2) (!both p1 p2))
2992       (not-p2 (!by-contradiction (not guard)
2993         (assume guard
2994           (!absurd (!right-and guard) not-p2))))))
2995     (not-p1 (!by-contradiction (not guard)
2996       (assume guard
2997         (!absurd (!left-and guard) not-p1))))))
2999   ((or p1 p2) (dmatch (!evaluate-guard p1)
3000     ((val-of p1) (!either p1 p2))
3001     (not-p1 (dmatch (!evaluate-guard p2)
3002       ((val-of p2) (!either p1 p2))
3003       (not-p2 (!by-contradiction (not guard)
3004         (assume guard
3005           (!cases guard
3006             (assume p1 (!absurd p1 not-p1))
3007             (assume p2 (!absurd p2 not-p2))))))))))

```

```

3008
3009 (define (do-clauses lhs guard-rhs-list justification)
3010   (dletrec ((loop (method (guard-res-list)
3011     (dmatch guard-res-list
3012       ((list-of [guard res] more)
3013         (!evaluate-guard guard
3014           (method ()
3015             (!chain [lhs = res justification]))
3016           (method ()
3017             (!loop more)))))))
3018     (!loop guard-rhs-list)))
3019
3020
3021 (define (prove-clause-step lhs rhs guard axioms)
3022   (dlet ((justification' (match axioms
3023     ((some-list _) (add guard axioms))
3024     ((some-sent _) [guard axioms])
3025     (_ axioms))))
3026     (!chain [lhs = rhs justification'])))
3027
3028 (define (prove-clause-step lhs rhs guard axioms)
3029   (dlet ((original-axioms axioms)
3030     (given (= lhs rhs))
3031     (rename-first (lambda (L)
3032       (match L
3033         ((list-of (some-sent p) more) (add (rename p) more))
3034         (_ L)))))
3035     (dletrec ((loop (method (axioms)
3036       (dmatch (rename-first axioms)
3037         ((list-of (|| (first-axiom as (forall (some-list uvars) (axiom-body as (if _ con))))
3038           (first-axiom as (forall (some-list uvars) (axiom-body as con))))
3039         rest)
3040         (dlet (#_ (print "\nLooking at this axiom: " first-axiom "\n"))
3041           )
3042         (dmatch (match-sentences given con)
3043           ((some-sub sub)
3044             (dtry (dmatch axiom-body
3045               ((if _ _)
3046                 (dlet ((body' (!uspec* first-axiom (sub uvars)))
3047                   (#_ (print "\nGot this body': " (val->string body') "\n"))
3048                   ([ant con] (match body' ((if (some-sent A) (some-sent B)) [A B])))
3049                   #_ (print "\nWill now try to prove components of this guard: "
3050                     (val->string ant)
3051                     "\ngiven this guard: " (val->string guard) "\n"))
3052                   (_ (!prove-components-of ant)))
3053                   (!mp body' ant)))
3054                   ((= _ _) (!uspec* first-axiom (sub uvars))))
3055                   (!loop rest))))
3056                   (_ (!loop rest))))
3057         (_ (dlet ( #_ (print "\nFailed! prove-clause loop on this lhs:\n" (val->string lhs)
3058           "\nand this rhs:\n" (val->string rhs)
3059           "\nand this guard:\n" (val->string guard)
3060           "\nand these axioms:\n" original-axioms))
3061           (_ ())
3062           )
3063           (!fail)))
3064         ))))
3065     (!loop axioms))))
3066
3067 (define (prove-clause-step lhs rhs guard axioms)
3068   (!force (= lhs rhs)))
3069
3070 (define (do-clauses lhs guard-rhs-list justification)
3071   (dletrec ((loop (method (guard-res-list)
3072     (dmatch guard-res-list
3073       ((list-of [guard res] more)
3074         (dlet (#_ (print "\nAbout to evaluate this guard: " (val->string guard) "\n"))
3075           (th (!evaluate-guard guard))
3076           (#_ (print "\nFinished guard evaluation...\n"))
3077         )

```

```

3078         (dmatch th
3079           ((val-of guard)
3080            (dlet (# (λ (print "\nThis guard was true: " (val->string guard)
3081                        #
3082                        "\nand we'll now chain from this lhs:\n" (val->string lhs)
3083                        #
3084                        " to this rhs side:\n" (val->string res) "\n"))
3085                     (justification' (match justification
3086                                     ((some-list _) (add guard justification))
3087                                     ((some-sent _) [guard justification])
3088                                     (_ justification)))
3089                     )
3090                     #(!chain [lhs = res justification'])
3091                     (!prove-clause-step lhs res guard justification)
3092                     ))
3093         (!loop guard-rhs-list)))
3094
3095 (define [success failure] [(method () (!dmark 'S)) (method () (!dmark 'F))])
3096
3097 (define (dcompile-entry equation def-eqn-id mod-path)
3098   (let (#(λ (print "\nGive equation: " equation))
3099         (translate-symbol (lambda (g) (symbol->string g)))
3100
3101         # (let ((long-name (get-eval-proc-name-1 g)))
3102             # (check ((constructor? g)
3103                       (symbol->string g)
3104                       ((equal? mod-path (all-but-last (split-string long-name '.))
3105                          (get-eval-proc-name g))
3106                       (else long-name))))))
3107   #
3108   (λ (print "\nAbout to dcompile-entry the following entry:\n" equation "\nwith the following def-eqn-id: " def-eqn-id)
3109     (compile-term (lambda (term mapping)
3110                   (letrec ((loop (lambda (t)
3111                                   (match t
3112                                    ((some-var x) (apply-map mapping x))
3113                                    ((g []) (val->string g))
3114                                    ((g (some-list args))
                                     (join lp (translate-symbol g)
                                             blank (separate (map loop args) blank) rp))))))
3115                     (loop term))))
3116     (compile-term' (lambda (term mapping)
3117                    (letrec ((loop (lambda (t)
3118                                    (match t
3119                                     ((some-var x) (apply-map mapping x))
3120                                     ((g []) (val->string t))
3121                                     ((g (some-list args))
                                      (let ((λ (λ (λ (print "\nTerm being translated: " t))
3122                                             (λ (print "\nRoot symbol is translated to this: " (translate-symbol g))
3123                                             )
3124                                      (join lp
3125                                             (translate-symbol g)
3126                                             blank
3127                                             (separate (map loop args) blank)
3128                                             rp))))))
3129                      (loop term))))
3130     (compile-guard (lambda (guard mapping)
3131                    (letrec ((loop (lambda (g)
3132                                    (match g
3133                                     ((some-term _) (compile-term' g mapping))
3134                                     ((not g') (join lp "not " (loop g') rp))
3135                                     ((and g1 g2) (join lp "and " (loop g1) blank (loop g2) rp))
3136                                     ((or g1 g2) (join lp "or " (loop g1) blank (loop g2) rp))))))
3137                      (loop guard))))
3138     (compile-terms (lambda (terms mapping)
3139                    (map (lambda (t) (compile-term t mapping)) terms)))
3140     (compile-terms' (lambda (terms mapping)
3141                      (map (lambda (t) (compile-term' t mapping)) terms))))
3142 (match equation
3143   ([uvars (as pattern (f (some-list args))) guard-res-list]

```

```
(let ((f-name (symbol->string f))
      (count (length guard-res-list))
      (new-vars (map (lambda (i)
                       (join "x" (val->string i)))
                     (from-to 1 (length uvars))))
      (input-vars (map (lambda (i)
                        (join "t" (val->string i)))
                      (from-to 1 (length uvars))))
      (mapping (extend empty-map (list-zip uvars new-vars)))
      (input-mapping (extend empty-map (list-zip uvars input-vars)))
      (are-all-vars (all-vars? args))
      (mapping (check (are-all-vars input-mapping)
                     (else mapping)))
      (_ (print "\nmapping: " mapping))
      (pattern-translation (check (are-all-vars "_")
                                   (else (join lb (separate (compile-terms' args mapping) blank) rb))))
      (translate-clause (lambda (guard-res-pair i)
                         (match guard-res-pair
                          ([guard res] (let ((cg (compile-guard guard mapping))
                                              (RHS (compile-term' res mapping)))
                                           (join lb cg blank RHS rb)))))))

(check
  ((negate (guards-should-be-matched? guard-res-list uvars))
   (let ((LHS (compile-term' pattern mapping))
         (check-clauses (join lb (separate (map-with-index translate-clause guard-res-list) blank) rb))
         (check-translation (join lp "!do-clauses " LHS blank check-clauses blank def-eqn-id rp)))
     (join lp pattern-translation blank check-translation rp)))
  (else (let ((extra-vars (get-extra-vars guard-res-list uvars))
              (compile-pattern (lambda (p mapping ts)
                                (compile-term' p mapping)))
              (mapping' (extend mapping (list-zip extra-vars
                                                  (map (lambda (i)
                                                         (join "p" (val->string i)))
                                           (from-to 1 (length extra-vars)))))))
            (translate-match-clause
             (lambda (guard-res-pair i)
               (match guard-res-pair
                ([guard res] (let ((cg (check ((&& (equal? guard true) (equal? i count)) "_")
                                              (else (join "(res as (= _ " (compile-pattern (rhs guard
                                                                 (result-term (compile-term' res mapping'))
                                                                 RHS (join "(!chain [" (compile-term pattern mapping') " = " result
                                                                 (join lp cg blank RHS rp))))))
                (discriminant (join lp "!deval " (compile-term' (lhs (first (first guard-res-list))) mapping')
                (match-clauses (separate (map-with-index translate-match-clause guard-res-list) newline))
                (match-translation (join lp "dmatch " discriminant "\n"
                                         match-clauses rp)))
              (join lp pattern-translation blank match-translation rp))))))

(forall (some-list uvars)
  (= (as pattern (f (some-list args))) rhs))
(let ((f-name (symbol->string f))
      (new-vars (map (lambda (i)
                       (join "x" (val->string i)))
                     (from-to 1 (length uvars))))
      (mapping (extend empty-map (zip uvars new-vars)))
      (LHS (compile-term' pattern mapping))
      (lhs' (join lb (separate (compile-terms' args mapping) blank) rb))
      (rhs' (compile-term rhs mapping))
      (RHS (compile-term' rhs mapping))
      (dbody (join "(!chain " lb LHS blank " = " RHS blank lb def-eqn-id rb rb rp)))
      (join lp lhs' blank dbody rp)))
(forall (some-list uvars)
  (if antecedent (= (as pattern (f (some-list args))) rhs)))
(let ((f-name (symbol->string f))
      (new-vars (map (lambda (i)
                       (join "x" (val->string i)))
                     (from-to 1 (length uvars))))
      (mapping (extend empty-map (zip uvars new-vars)))
      (input-vars (map (lambda (i)
                        (join "t" (val->string i)))
                      (from-to 1 (length uvars))))
```

```

3218         (input-mapping (extend empty-map (zip uvars input-vars)))
3219         (are-all-vars (all-vars? args))
3220         (mapping (check (are-all-vars input-mapping)
3221                         (else mapping))))
3222         (LHS (compile-term' pattern mapping))
3223         (lhs' (check (are-all-vars "_")
3224                     (else (join lb (separate (compile-terms' args mapping) blank) rb))))
3225         (guard (compile-guard antecedent mapping))
3226         (lhs'' lhs')
3227         (rhs' (compile-term' rhs mapping))
3228         (RHS (compile-term' rhs mapping))
3229         (dbody (join "(!chain " lb LHS blank " = " RHS blank lb def-eqn-id rb rp))
3230         (dbody (join "(!do-clauses " LHS blank lb lb guard blank RHS rb rb blank def-eqn-id rp)))
3231         (join lp lhs'' blank dbody rp))))))
3232
3233 (define (all-alpha-variants? ec)
3234   (let ((are-alpha-variants (lambda (s t)
3235                               (match (alpha-variants? s t)
3236                                     ((some-sub _) true)
3237                                     (_ false)))))
3238     (letrec ((loop (lambda (current-pat remaining-triples)
3239                     (match remaining-triples
3240                           ([] true)
3241                           ((list-of [pat guard res] more) (&& (are-alpha-variants pat current-pat)
3242                                                                (loop pat more)))
3243                           (_ false)))))
3244       (match ec
3245         ([] true)
3246         ((list-of [pat _ _] more) (loop pat more))))))
3247
3248 (define (get-guard g)
3249   (match g
3250     (()) true)
3251     (_ g)))
3252
3253 (define (simple-simplify gr-list pat-vars)
3254   (match gr-list
3255     ([] gr-list)
3256     ((list-of [guard1 res1] more)
3257      (let ((conjuncts1 (get-conjuncts guard1)))
3258        (check ((negate (equal? (length conjuncts1) 1)) gr-list)
3259              (else (let ((sole-first-conjunct (first conjuncts1)))
3260                      (letrec ((loop (lambda (rem-pairs sure-so-far accum)
3261                                      (match rem-pairs
3262                                            ([] (add [guard1 res1] (rev accum)))
3263                                            ((list-of [guard res] more)
3264                                             (let ((conjuncts (get-conjuncts guard))
3265                                                  (conjuncts' (filter-out conjuncts (lambda (c) (member? (complement c)
3266                                                                 (guard' (make-and conjuncts'))
3267                                                                 (sure-so-far' (match conjuncts'
3268                                                                    ([_] (add (first conjuncts') sure-so-far))
3269                                                                    (_ sure-so-far))))
3270                                                  (loop more sure-so-far' (add [guard' res] accum)))))))
3271                    (loop more [sole-first-conjunct] []))))))))))
3272
3273 (define (make-single-entry ec)
3274   (match ec
3275     ([first-and-only-triple] (triple->equation first-and-only-triple))
3276     ((list-of (as first-triple [pat1 guard1 res1]) more-triples)
3277      (letrec ((get-pgr-list (lambda (remaining-triples results sub)
3278                              (match [remaining-triples results]
3279                                    ([[] _] [(rev results) sub])
3280                                    ((list-of [pat guard res] more) [])
3281                                     (let ((guard' (get-guard guard)))
3282                                       (get-pgr-list more (add [pat guard' res] results) sub))))
3283              ((list-of [pat guard res] more) (list-of [previous-pat previous-guard previous-res]
3284                (let ((guard' (get-guard guard))
3285                      (sub' (unify previous-pat pat)))
3286                  (get-pgr-list more (add [pat guard' res] results) (compose subs sub' sub))))))
3287      (let ((get-vars (lambda (x)

```

```

3288         (match x
3289           ((() []))
3290           (_ (vars x))))))
3291 #   (uvars (rev (rd (join (get-vars-manual pat1) (get-vars guard1)))))
3292 (uvars (rd (get-vars-manual pat1)))
3293 (pat pat1)
3294 ([pat-guard-res-list sub] (get-pgr-list ec [] empty-sub))
3295 ((answer as [uvars' pat' gr-list'])
3296   [(sub uvars) (sub pat) (map (lambda (triple)
3297                                (match triple
3298                                  ([p g r] [(sub g) (sub r)])))
3299                                pat-guard-res-list)])
3300 (pvars (sub (vars pat1)))
3301 ##   (_ (print "\nanswer: " answer "\nsub: " sub "\npvars: " pvars "\nand gr-list': " gr-list'))
3302 (gr-list" (simple-simplify gr-list' pvars)))
3303 [uvars' pat' gr-list"])))
3304
3305 (define reprocess-ec (lambda (ec)
3306   (check ((all-alpha-variants? ec) [(make-single-entry ec)])
3307     (else (map triple->equation ec)))))
3308
3309 (define (compile-symbol f mod-path)
3310   (let ((arity (arity-of f))
3311         (f-name (get-eval-proc-name f))
3312         (params (map (lambda (i) (join "t" (val->string i)))
3313                       (from-to 1 arity)))
3314         (params' (separate params blank))
3315         (all-equations (defining-axioms f))
3316         (processsed-equivalence-classes (analyze all-equations))
3317         (reprocessed-equivalence-classes (map reprocess-ec processsed-equivalence-classes))
3318         (all-entries (flatten reprocessed-equivalence-classes))
3319         (clauses (map (lambda (e) (compile-entry e mod-path)) all-entries))
3320         (body (match clauses
3321                  ([([split "(_" rest]) (all-but-last rest))
3322                   (_ (join lp "match " lb params' rb newline tab tab (separate clauses (join newline tab tab)) rp)))]
3323                    (res (join lp "letrec " lp lp f-name blank lp "lambda " lp params' rp newline tab body rp rp rp newline tab tab
3324                              res)))
3325
3326
3327 (define (escape str)
3328   (escape-string str))
3329
3330 (define (dcompile-symbol f mod-path)
3331   (let ((arity (arity-of f))
3332         (f-name (get-eval-proc-name f))
3333         (symbol-name (symbol->string f))
3334         (safe-symbol-name (join lp "string->symbol " quote (escape symbol-name) quote rp))
3335         (params (map (lambda (i) (join "t" (val->string i)))
3336                       (from-to 1 arity)))
3337         (params' (separate params blank))
3338         (all-equations (defining-axioms f))
3339         (defining-axioms-id "defining-axioms")
3340         (processsed-equivalence-classes (analyze all-equations))
3341         (reprocessed-equivalence-classes (map reprocess-ec processsed-equivalence-classes))
3342         (all-entries (flatten reprocessed-equivalence-classes))
3343         (clauses (map (lambda (e) (dcompile-entry e defining-axioms-id mod-path)) all-entries))
3344         (body (match clauses
3345                  ([([split "(_" rest]) (all-but-last rest))
3346                   (_ (join lp "dmatch " lb params' rb newline tab tab (separate clauses (join newline tab tab)) rp)))]
3347                    (res (join lp "let " lp lp defining-axioms-id blank lp "defining-axioms" blank safe-symbol-name rp rp rp newlin
3348                              lp "method " lp params' rp newline tab body rp rp)))
3349         res))
3350
3351 (define (compile-symbols fsyms mod-path)
3352   (let ((arities (map arity-of fsyms))
3353         (names (map get-eval-proc-name fsyms))
3354         (param-lists (map (lambda (fsym)
3355                             (map (lambda (i) (join "t" (val->string i)))
3356                                   (from-to 1 (arity-of fsym)))))
3357         fsyms))

```



```

3358 (param-lists' (map (lambda (param-list) (separate param-list blank)) param-lists))
3359 (equation-lists (map defining-axioms fsyms))
3360 (clause-lists (map (lambda (equation-list)
3361   (let ((processed-equivalence-classes (analyze equation-list))
3362         (reprocessed-equivalence-classes (map reprocess-ec processed-equivalence-classes))
3363         (all-entries (flatten reprocessed-equivalence-classes)))
3364     (map (lambda (e) (compile-entry e mod-path)) all-entries)))
3365   equation-lists))
3366 (bodies (map (lambda (pcl)
3367   (match pcl
3368     ([param-list clause-list]
3369      (match clause-list
3370        ([ (split "_" rest)] (all-but-last rest))
3371        (_ (join lp "match " lb param-list rb newline tab tab (separate clause-list (join newline
3372 (zip param-lists' clause-lists)))
3373 (lams (map (lambda (pbl)
3374   (match pbl
3375     ([param-list body] (join lp "lambda " lp param-list rp newline tab body rp))))
3376     (zip param-lists' bodies)))
3377 (bindings (map (lambda (name-and-lam)
3378   (match name-and-lam
3379     ([name lam] (join lp name blank lam rp)))
3380     (zip names lams)))
3381 (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb rp
3382 res)))
3383
3384 (define (compile-symbols-with-default fsyms mod-path)
3385 (let ((arities (map arity-of fsyms))
3386       (names (map get-eval-proc-name fsyms))
3387       (param-lists (map (lambda (fsym)
3388         (map (lambda (i) (join "t" (val->string i)))
3389              (from-to 1 (arity-of fsym))))
3390         fsyms))
3391 (param-lists' (map (lambda (param-list) (separate param-list blank)) param-lists))
3392 (equation-lists (map defining-axioms fsyms))
3393 (clause-lists (map (lambda (equation-list-fsym-params)
3394   (let (([equation-list [fsym params]] equation-list-fsym-params)
3395         (processed-equivalence-classes (analyze equation-list))
3396         (reprocessed-equivalence-classes (map reprocess-ec processed-equivalence-classes))
3397         (all-entries (flatten reprocessed-equivalence-classes))
3398         (last-entry (join "t" (val->string fsym) " " params "))))
3399     (join (map (lambda (e) (compile-entry e mod-path)) all-entries)
3400           [last-entry])))
3401   (list-zip equation-lists (list-zip fsyms param-lists'))))
3402 (bodies (map (lambda (pcl)
3403   (match pcl
3404     ([param-list clause-list]
3405      (match clause-list
3406        ([ (split "_" rest)] (all-but-last rest))
3407        (_ (join lp "match " lb param-list rb newline tab tab (separate clause-list (join newline
3408 (zip param-lists' clause-lists)))
3409 (lams (map (lambda (pbl)
3410   (match pbl
3411     ([param-list body] (join lp "lambda " lp param-list rp newline tab body rp))))
3412     (zip param-lists' bodies)))
3413 (bindings (map (lambda (name-and-lam)
3414   (match name-and-lam
3415     ([name lam] (join lp name blank lam rp)))
3416     (zip names lams)))
3417 (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb rp
3418 res)))
3419
3420 # (assert* ite-axioms :=
3421 #   (fun-def [(ite true ?x _) = ?x
3422 #             | (ite false _ ?y) = ?y]))
3423
3424 # (define (ite' x y z)
3425 #   (match x
3426 #     (true y)
3427 #     (_ z)))

```

```

3428
3429
3430 (define [+ ' -' *' /' %'] [plus minus times div mod])
3431 (define [+ ' -' *' /' %'] [+ ' -' *' /' %'])
3432
3433 (define [= ' <' >' <=' >='] [struc-equal? less? greater? less-or-equal? greater-or-equal?])
3434
3435 (define [= ' <' >' <=' >='] [= ' <' >' <=' >='])
3436
3437 (define [+ 'R -'R *'R /'R %'R] [+ ' -' *' /' %'])
3438
3439 (define (+ 'R x y)
3440   (match [x y]
3441     ([0 _] y)
3442     ([_ 0] x)
3443     (_ (plus x y))))
3444
3445 (define (- 'R x y)
3446   (match [x y]
3447     ([_ 0] x)
3448     ([ (some-term z) z] 0)
3449     (_ (minus x y))))
3450
3451 (define (* 'R x y)
3452   (match [x y]
3453     (([| [0 _] [_ 0]] 0)
3454     ([1 _] y)
3455     ([_ 1] x)
3456     (_ (times x y))))
3457
3458 (define [= 'R <'R >'R <='R >='R] [= ' <' >' <=' >='])
3459
3460 (define (= 'basic s t)
3461   (let ((res (struc-equal? s t)))
3462     (check ((equal? res true) res)
3463       ((&& (super-canonical? s) (super-canonical? t)) res)
3464       (else (let ((f (root s))
3465                    (g (root t)))
3466                 (check ((&& (free-constructor? f) (free-constructor? g) (unequal? f g)) false)
3467                   (else (let ((eq (= s t)))
3468                             (check ((holds? eq) true)
3469                               ((holds? (not eq)) false)
3470                               (else (equal? s t)))))))))))
3471
3472 (define (= 's t)
3473   (check ((| (poly? s) (poly? t))
3474     (match (= s t)
3475       ((= (some-term l) (some-term r)) (= 'basic l r)))
3476     (else (= 'basic s t))))
3477
3478 (define = ' =')
3479
3480 (define (= 'R s t)
3481   (let ((res (struc-equal? s t)))
3482     (check ((equal? res true) res)
3483       ((&& (super-canonical? s) (super-canonical? t)) res)
3484       (else (let ((f (root s))
3485                    (g (root t)))
3486                 (check ((&& (free-constructor? f) (free-constructor? g) (unequal? f g)) false)
3487                   (else (let ((eq (= s t)))
3488                             (check ((holds? eq) true)
3489                               ((holds? (= t s)) true)
3490                               ((holds? (not eq)) false)
3491                               ((equal? s t) true)
3492                               (else (= s t)))))))))))
3493
3494 (define silent-eval-mode (cell false))
3495
3496 (define
3497   (eval1 t)

```

```

3498 (match t
3499   (((some-symbol f) (some-list args))
3500     (check ((null? args) t)
3501       ((constructor? f)
3502         (match (defining-axioms f)
3503           ([] (make-term f (map eval args)))
3504           (_ (let ((f' (try (evaluate (get-eval-proc-name-generic f (mod-path)))
3505                             (evaluate (get-eval-proc-name f))))
3506                 (args' (map eval args)))
3507             (try (app-proc f' args') (make-term f args')))))
3508         (else (let ((f' (try (evaluate (get-eval-proc-name-generic f (mod-path)))
3509                             (evaluate (get-eval-proc-name f))))
3510                 (app-proc f' (map eval args)))))
3511       ((not p) (negate (eval p)))
3512       ((and (some-list args)) (&* (map eval args)))
3513       ((or (some-list args)) (||* (map eval args)))
3514       ((if p1 p2) (eval (or (not p1) p2)))
3515       ((iff p1 p2) (eval (and (if p1 p2) (if p2 p1))))
3516       (_ t))
3517   (eval t)
3518   (try (let ((res (evall t)))
3519         (try (rhs (= t res))
3520           res))
3521     (check ((ref silent-eval-mode) ())
3522       (else (print "\nUnable to reduce the term:\n" t "\nto a normal form.\n")))))
3523
3524
3525 (define
3526   (evall t)
3527     (match t
3528       (((some-symbol f) (some-list args))
3529         (check (((| (canon? t) (meta-id? t) (&& (null? args) (null? (defining-axioms f)))) t)
3530           ((constructor? f)
3531             (match (defining-axioms f)
3532               ([] (make-term f (map evall args)))
3533               (_ (let ((f' (try (evaluate (get-eval-proc-name-generic f (mod-path)))
3534                               (evaluate (get-eval-proc-name f))))
3535                   (args' (map evall args))
3536                   (res (try (app-proc f' args') (make-term f args'))))
3537                 (check ((poly? res) (rhs (= t res))) (else res))))))
3538             (else (let ((f' (try (let ((name (get-eval-proc-name-generic f (mod-path)))
3539                                   )
3540                                   (evaluate name))
3541                               (evaluate (get-eval-proc-name f))))
3542                   (res (app-proc f' (map evall args)))
3543                   (check ((poly? res) (rhs (= t res))) (else res))))))
3544               ((not p) (negate (evall p)))
3545               ((and (some-list args)) (&* (map evall args)))
3546               ((or (some-list args)) (||* (map evall args)))
3547               ((if p1 p2) (evall (or (not p1) p2)))
3548               ((iff p1 p2) (evall (and (if p1 p2) (if p2 p1))))
3549               (_ t))
3550           (eval t)
3551           (check ((has-vars? t) (print "\nNon-ground term given as input, unable to reduce the term:\n" t "\nto a normal form
3552             (else (try (evall t)
3553               (check ((ref silent-eval-mode) ())
3554                 (else (print "\nUnable to reduce the term:\n" t "\nto a normal form.\n"))))))))
3555
3556
3557 (make-private "evall")
3558
3559 (define (eval-silent t)
3560   (let ((x (ref silent-eval-mode))
3561         (_ (set! silent-eval-mode true))
3562         (res (eval t))
3563         (_ (set! silent-eval-mode x))
3564         res))
3565
3566
3567 (define

```

```

3568 (reduce1 t)
3569   (match t
3570     (((some-symbol f) (some-list args))
3571       (check ((|| (canon? t) (meta-id? t) (&& (null? args) (null? (defining-axioms f)))) t)
3572         ((&& (constructor? f) (null? (defining-axioms f))) (make-term f (map reduce1 args)))
3573         (else (let ((f' (try (evaluate (get-reduce-proc-name-generic f (mod-path)))
3574                               (evaluate (get-reduce-proc-name f))
3575                               ())))
3576           (match f'
3577             ((|| () (some-symbol _)) (make-term f (map reduce1 args)))
3578             (_ (app-proc f' (map reduce1 args)))))))
3579     ((not p) (negateR (reduce1 p)))
3580     ((and (some-list args)) (&&R (map reduce1 args)))
3581     ((or (some-list args)) (||R (map reduce1 args)))
3582     ((if p1 p2) (reduce1 (or (not p1) p2)))
3583     ((iff p1 p2) (reduce1 (and (if p1 p2) (if p2 p1))))
3584     ((forall (some-list vars) (some-sentence body)) (forall* vars (reduce1 body)))
3585     ((exists (some-list vars) (some-sentence body)) (exists* vars (reduce1 body)))
3586     (_ t))
3587 (reduce t)
3588   (match t
3589     ((some-sentence p) (check ((holds? p) true)
3590       ((|| (holds? (complement p)) (holds? (not p))) false)
3591       (else (let ((res (reduce1 t)))
3592         (try (rhs (= t res))
3593           res))))))
3594     (_ (let ((res (reduce1 t)))
3595       (try (rhs (= t res))
3596         res))))))
3597
3598
3599 (define (eval2 t)
3600   (match t
3601     (((some-symbol f) (some-list args))
3602       (check ((|| (canon? t) (meta-id? t) (&& (null? args) (null? (defining-axioms f)))) t)
3603         ((constructor? f) (make-term f (map eval2 args)))
3604         (else (let ((error? (cell false))
3605           (f' (try (evaluate (get-eval-proc-name-1 f))
3606             (evaluate (get-eval-proc-name f))
3607             (set! error? true))))
3608           (check ((ref error?) t)
3609             (else (let ((res (try (app-proc f' args)
3610               (set! error? true))))
3611               (check ((ref error?) t)
3612                 (else (eval2 res))))))))))
3613       ((not p) (negate (eval2 p)))
3614       ((and (some-list args)) (&&* (map eval2 args)))
3615       ((or (some-list args)) (||* (map eval2 args)))
3616       ((if p1 p2) (eval2 (or (not p1) p2)))
3617       ((iff p1 p2) (eval2 (and (if p1 p2) (if p2 p1))))
3618       (_ t)))
3619
3620 (define (eval' t)
3621   (try (eval2 t)
3622     t))
3623
3624 (set-precedence eval 5)
3625 (set-precedence reduce 5)
3626
3627 (define (matches-with l r theta-cell)
3628   (let (#(_ (print "\nInside matches-with, with left: " l " and right: " r))
3629     (_ ()))
3630     )
3631   (match [l r]
3632     [(some-term left) (some-term right)] (match (match-terms left ((ref theta-cell) right))
3633       ((some-sub sub) (seq (set! theta-cell (compose-subs sub (ref theta-cell)
3634         (_ false))))
3635       [(some-sentence left) (some-sentence right)]
3636         (match (match-sentences left ((ref theta-cell) right))
3637           ((some-sub sub) (seq (set! theta-cell (compose-subs sub (ref theta-cell))) true))

```

```

3638         (_ false)))
3639     (_ false))))
3640
3641 (define (matches-with left right theta-cell)
3642   (let ((theta (ref theta-cell)))
3643     (match (match-terms (theta left) (theta right))
3644       ((some-sub sub) (seq (set! theta-cell (compose-subs sub theta)) true))
3645       (_ false))))
3646
3647 (define (translate-symbol-new g mod-path evaluator-name)
3648   (let ((reducing? (check ((equal? evaluator-name "eval1") false) (else true)))
3649         (long-name (check (reducing? (get-reduce-proc-name-1 g))
3650                             (else (get-eval-proc-name-1 g))))
3651         (_ ()))
3652     (check ((constructor? g)
3653             (symbol->string g)
3654             ((prefix? (all-but-last (split-string long-name `.) mod-path)
3655                       (check (reducing? (get-reduce-proc-name g))
3656                             (else (get-eval-proc-name g)))
3657             )
3658             (else (check (reducing? (get-reduce-proc-name-generic g mod-path))
3659                           (else (get-eval-proc-name-generic g mod-path)))))))
3660
3661 (define (translate-arg-symbol-new g mod-path evaluator-name)
3662   (symbol->string g))
3663
3664 (define (compile-constant-term t mod-path evaluator-name)
3665   (check ((null? (defining-axioms t)) (val->string t))
3666         (else (join lp (translate-symbol-new (root t) mod-path evaluator-name) rp))))
3667
3668 (define (compile-constant-term-arg t mod-path evaluator-name)
3669   (check ((poly? t) (let ((v (var->string (fresh-var))))
3670                       (join (" v " as " (val->string t) ")"))
3671         (else (val->string t))))
3672
3673 (define (compile-term-new term mapping mod-path evaluator-name)
3674   (letrec ((loop (lambda (t)
3675                     (match t
3676                       ((some-var x) (try (match (apply-map mapping x)
3677                                                (() (val->string t))
3678                                                (res res))
3679                                           (val->string t)))
3680                       ((g []) (compile-constant-term t mod-path evaluator-name))
3681                       ((g (some-list args))
3682                        (check ((&& false (equal? evaluator-name "reduce")
3683                                         (equal? (fsd0 g) ()))
3684                               (join lp (symbol->string g) " " (separate (map loop args) blank) rp))
3685                        (else (join lp (translate-symbol-new g mod-path evaluator-name) blank (separate (map loop
3686 (loop term)))
3687
3688 (define (compile-arg-new term mapping mod-path evaluator-name)
3689   (letrec ((loop (lambda (t)
3690                     (match t
3691                       ((some-var x) (try (match (apply-map mapping x)
3692                                                (() (val->string t))
3693                                                (res res))
3694                                           (val->string t)))
3695                       ((g []) (compile-constant-term-arg t mod-path evaluator-name))
3696                       ((g (some-list args))
3697                        (check ((&& (equal? evaluator-name "reduce")
3698                                         (equal? (fsd0 g) ()))
3699                               (join lp (symbol->string g) " " (separate (map loop args) blank) rp))
3700                        (else (join lp (translate-arg-symbol-new g mod-path evaluator-name) blank (separate (map loop
3701 (loop term)))
3702
3703 (define (compile-term-and-boolean-combination term mapping mod-path evaluator-name)
3704   (letrec ((loop (lambda (t)
3705                     (match t

```

```

3708 ((some-var x) (try (match (apply-map mapping x)
3709                        (() (val->string t))
3710                        (res res))
3711                  (val->string t)))
3712 ((not (some-sentence arg)) (join "(negate " (loop arg) ")"))
3713 ((and (some-list args) (let ((results (map loop args)))
3714                          (join "&& " (separate results " " " "))))
3715 ((or (some-list args) (let ((results (map loop args)))
3716                         (join "|| " (separate results " " " "))))
3717 ((g []) (compile-constant-term t mod-path evaluator-name))
3718 ((g (some-list args)
3719      (join lp (translate-symbol-new g mod-path evaluator-name) blank (separate (map loop args) blank)
3720            (loop term)))
3721
3722 (define (compile-term-with-no-eval term mapping mod-path evaluator-name)
3723   (letrec ((loop (lambda (t)
3724                   (match t
3725                     ((some-var x) (try (match (apply-map mapping x)
3726                                           (() (val->string t))
3727                                           (res res))
3728                                       (val->string t)))
3729                     ((g []) (compile-constant-term t mod-path evaluator-name))
3730                     ((g (some-list args)
3731                      (join lp (symbol->string g) blank (separate (map loop args) blank) rp))))))
3732         (loop term)))
3733
3734 (define (guard->where-condition guard mapping left-uvars fsym-being-defined needs-sub? mod-path evaluator-name)
3735   (let ((orient (lambda (s t t-vars)
3736                 (check ((negate (subset? t-vars left-uvars)) (= s t))
3737                       ((negate (subset? (vars s) left-uvars)) (= t s))
3738                       ((constructor? (root s)) (= t s))
3739                       ((constructor? (root t)) (= s t))
3740                       ((member? fsym-being-defined (syms s)) (= s t))
3741                       ((member? fsym-being-defined (syms t)) (= t s))
3742                       (else (= s t))))))
3743     (_ (set! needs-sub? false))
3744     (_ ()))
3745   (letrec ((loop (lambda (guard)
3746                   (match guard
3747                     ((= s t) (let ((t-vars (vars t))
3748                                     #(_ (print "\nDoing this guard: " guard ", with left-uvars: " left-uvars))
3749                                     (oriented-guard (orient s t t-vars))
3750                                     #(_ (print "\nOriented guard: " oriented-guard))
3751                                     (_ ()))
3752                               (match oriented-guard
3753                                 ((= s t) (check ((subset? (vars t) left-uvars)
3754                                                    (compile-term-new (= s t) mapping mod-path evaluator-name))
3755                                                  (else (seq (set! needs-sub? true)
3756                                                           (join "(matches-with " (compile-term-new s mapping mod-path evaluator-name)
3757                                                                (compile-term-new t mapping mod-path evaluator-name)
3758                                                                (some-atom A)
3759                                                                (let (#(_ (print "\nDoing this guard: " guard ", with left-uvars: " left-uvars))
3760                                                                    (_ ()))
3761                                                                (check ((subset? (vars A) left-uvars) (compile-term-new A mapping mod-path evaluator-name)
3762                                                                      (else (seq (set! needs-sub? true)
3763                                                       (join "(eval1 ((ref theta-cell) "
3764                                                            (val->string A) "))))))
3765                                                                ((and (some-list guards) (let ((results (map loop guards))
3766                                                                    (check ((equal? evaluator-name "eval1")
3767                                                                           (join "&& " (separate results " " " ")))
3768                                                                           (else (join "&&R [" (separate results "
3769                                                                 " " " "))))))
3770                                                                ((or (some-list guards) (let ((results (map loop guards))
3771                                                                    (check ((equal? evaluator-name "eval1")
3772                                                                           (join "|| " (separate results " " " ")))
3773                                                                           (else (join "||R [" (separate results "
3774                                                                 " " " "))))))
3775                                                                ((not guard) (check ((equal? evaluator-name "eval1")
3776                                                                    (join "(negate " (loop guard) ")")
3777                                                                    (else (join "(negateR " (loop guard) "))))))))))

```

```

3776     (match guard
3777       (true "_")
3778       (_ (loop guard))))))
3779
3780
3781 (define (compile-equation-into-single-clause eqn mod-path evaluator-name)
3782   (let ((needs-sub? (cell false))
3783         (res (match eqn
3784                 ((forall (some-list uvars) (= (as left ((some-symbol f) (some-list args))) (some-term right)))
3785                  (let ((proper-uvars (vars left))
3786                        # (nums-and-uvars (list-zip (from-to 1 (length proper-uvars)) proper-uvars))
3787                        (params (map (lambda (i)
3788                                      (join "x" (val->string i)))
3789                                      (from-to 1 (length proper-uvars)))))
3790                  (mapping (extend empty-map (list-zip proper-uvars params)))
3791                  (arg-translations (separate (map (lambda (a) (compile-arg-new a mapping mod-path evaluator-name)) args) "
3792                  (pattern (join "[" arg-translations "]"))
3793                  (result (compile-term-new right mapping mod-path evaluator-name)))
3794                  (join "(" pattern " " result "\n"))))
3795                 ((forall (some-list uvars) (if (some-sentence guard)
3796                                                  (= (as left ((some-symbol f) (some-list args)))
3797                                                    (some-term right))))
3798                  (let ((proper-uvars (vars left))
3799                        (params (map (lambda (i)
3800                                      (join "x" (val->string i)))
3801                                      (from-to 1 (length proper-uvars)))))
3802                  (mapping (extend empty-map (list-zip proper-uvars params)))
3803                  (where-condition (guard->where-condition guard mapping proper-uvars f needs-sub? mod-path evaluator-name))
3804                  (arg-translations (separate (map (lambda (a) (compile-term-new a mapping mod-path evaluator-name)) args)
3805                  (pattern (check ((equal? where-condition "_")
3806                                   (join "[" arg-translations "]"))
3807                                   (else (join "([ arg-translations " where " where-condition " ]))))))
3808                  (result (check ((ref needs-sub?) (join "(" evaluator-name " (ref theta-cell) " (compile-term-with-no-eval
3809                                   (else (compile-term-new right mapping mod-path evaluator-name)))))
3810                                   (join "(" pattern " " result "\n"))))
3811                 ((forall (some-list uvars) (iff (as left ((some-symbol f) (some-list args))) (some-sentence right)))
3812                  (let ((proper-uvars (vars left))
3813                        (params (map (lambda (i)
3814                                      (join "x" (val->string i)))
3815                                      (from-to 1 (length proper-uvars)))))
3816                  (mapping (extend empty-map (list-zip proper-uvars params)))
3817                  (arg-translations (separate (map (lambda (a) (compile-term-new a mapping mod-path evaluator-name)) args)
3818                  (pattern (join "[" arg-translations "]"))
3819                  (result (compile-term-and-boolean-combination right mapping mod-path evaluator-name)))
3820                  (join "(" pattern " " result "\n")))))
3821     [res (ref needs-sub?)])
3822
3823 (define (compile-equation-into-single-clause' eqn mod-path evaluator-name)
3824   (match eqn
3825     ((forall (some-list uvars) (and (some-list args)))
3826      (map (lambda (arg) (compile-equation-into-single-clause (forall* uvars arg) mod-path evaluator-name)) args))
3827     (_ (let ((res (compile-equation-into-single-clause eqn mod-path evaluator-name))
3828              # (print "\nResult: " res))
3829          [res])))
3830
3831
3832 (define (compilable-bicond-axiom p)
3833   (match p
3834     ((forall (some-list uvars) (iff (some-atom A) (some-sentence RHS)))
3835      (match A
3836        (((some-symbol f) (some-list _)) (unequal? f =))
3837        (_ false)))
3838     (_ false)))
3839
3840 (define (eqn-guard e)
3841   (match e
3842     ((forall (some-list _) (if g _) g)
3843      (forall (some-list _) _))
3844     (_))
3845
3846 (define (follows-from-0 c lhs eqns)

```

```

3846 (let ((entails? (lambda (p1 p2)
3847   (equal? (complement p1) (complement p2))))
3848   (apply-dm (lambda (p)
3849     (match p
3850       ((not (or (some-list props))) (and* (map (lambda (q) (complement q)) props))
3851       (_ p)))))
3852   (for-some eqns
3853     (lambda (e)
3854       (match (rename e)
3855         ((forall (some-list _) (if g (= lhs' _)))
3856          (match (unify lhs lhs')
3857            (false false)
3858            ((some-sub sub)
3859             (let ((c' (sub c))
3860                 (g' (sub g)))
3861               (|| (equal? c' (complement g'))
3862                  (let ((conjunctions (get-conjunctions (app-dm (not g')))))
3863                    (for-some conjunctions
3864                      (lambda (c)
3865                        (entails? c c'))))))))))
3866         (_ false)))))
3867
3868 (define (follows-from c lhs eqns)
3869   (let ((left-sides (map left-rule-side eqns))
3870         (diff-props (try (get-conjunctions (diff* lhs left-sides)) [])))
3871     (|| (follows-from-0 c lhs eqns)
3872         (member? c diff-props)))
3873
3874
3875 # rearrange-and-simplify takes a list of eqns E1 ... En and reorders and simplifies them
3876 # so that redundant guards in the newly produced permutation are eliminated. A guard in
3877 # an equation Ei' in the new list E1' ... En' is considered redundant if it follows from
3878 # the negation of the guard of some preceding equation E1' .. E{i-1}'. What constitutes
3879 # "follows from" is relative, and handled by the internal procedure entails? inside
3880 # follows-from above. Right now it's just syntactic identity, but entails? could be
3881 # redefined to, e.g., take into account datatype axioms.
3882
3883 (define (rearrange-and-simplify eqns)
3884   (let ((#(_ (print "\nAbout to rearrange and simplify the following eqns:\n" eqns))
3885         (guard-size (lambda (g) (match g (() 0) (_ (size g)))))
3886         (eqns (merge-sort eqns
3887                           (lambda (e1 e2)
3888                             (less? (guard-size (eqn-guard e1))
3889                                   (guard-size (eqn-guard e2)))))))
3889     (letrec ((loop (lambda (remaining-eqns previous-eqns)
3890                     (match remaining-eqns
3891                       ([] (rev previous-eqns))
3892                       ((list-of e rest)
3893                        (match e
3894                          ((forall (some-list uvars) (if g (as body (= (some-term l) (some-term r)))))
3895                           (let ((conjunctions (get-conjunctions g))
3896                               (conjunctions' (filter-out conjunctions (lambda (c) (follows-from c l previous-eqns))))
3897                             (new-guard (match conjunctions'
3898                                         ([] true)
3899                                         (_ (and* conjunctions'))))
3900                             (loop rest (add (forall* uvars (if new-guard body)) previous-eqns))))
3901                        (_ (loop rest (add e previous-eqns)))))))
3902         (let ((res (loop eqns []))
3903               #(_ (print "\nRESULT:\n" res))
3904               )
3905           res))))
3906
3907
3908 # To debug the simplifier rearrange-and-simplify while making sure that compilation works,
3909 # uncomment the following two lines to redefine rearrange-and-simplify as the identity function:
3910
3911 #(define (rearrange-and-simplify eqns)
3912 # eqns)
3913
3914 (define (compile-symbols-simple fsyms mod-path)
3915   (let ((arities (map arity-of fsyms))

```


[illegible]

```

3986         (match pair
3987           ([eqn bc] (compilable-bicond-axiom bc))))))
3988         (bicond-axioms (rd (map second compilable-elements)))
3989         (equational-axioms (list-diff axioms (rd (map first compilable-elements)))))
3990         (join equational-axioms bicond-axioms)))
3991 (equation-lists (map get-defining-axioms fsyms))
3992 # (equation-lists (map (lambda (f) (rearrange-and-simplify (get-defining-axioms f))) fsyms))
3993 (clause-lists (map (lambda (equation-list)
3994   (map (lambda (e) (flatten (compile-equation-into-single-clause' e mod-path "reduce"))) equa
3995     equation-lists))
3996 (bodies (map (lambda (pcl)
3997   (match pcl
3998     ([[param-list fsym] clause-list]
3999     (let ((clause-list' (map first clause-list))
4000 #       (last-clause (join "(_ (make-term " (val->string fsym) " [" param-list "]))"))
4001       (term-to-be-reduced (join "(" (val->string fsym) " " param-list ")))
4002       (last-clause (join "(_ " term-to-be-reduced ")))
4003       (clause-list' (join clause-list' [last-clause]))
4004       (needs-sub (for-some (map second clause-list) (lambda (x) x)))
4005       (body1 (join lp "match " lb param-list rb newline tab tab (separate clause-list' (join
4006       (body1' (check (needs-sub (join "(let ((theta-cell (cell empty-sub))) " body1 ")))
4007       (else body1)))
4008       (body1' (join "(try (resolve-redex " term-to-be-reduced ")\n
" body1' "))))))
4009       body1'))))
4010       (list-zip param-lists' clause-lists)))
4011 (lams (map (lambda (pbl)
4012   (match pbl
4013     ([[param-list fsym] body] (join lp "lambda " lp param-list rp newline tab body rp))))
4014   (list-zip param-lists' bodies)))
4015 (bindings (map (lambda (name-and-lam)
4016   (match name-and-lam
4017     ([name lam] (join lp name blank lam rp))))
4018   (list-zip names lams)))
4019 (res (join lp "letrec " lp (separate bindings (join newline tab)) rp newline tab lb (separate names blank) rb rp
4020 res))
4021
4022 (define (cd f)
4023   (println (compile-symbols-with-default [f] [])))
4024
4025
4026 (define (vars-to-rid left-args guard-free-vars)
4027   (letrec ((loop (lambda (rem-args previous-args results)
4028     (match rem-args
4029       ([] results)
4030       ((list-of (some-var v) more)
4031        (let ((V (vars* (join previous-args more))))
4032          (check ((|| (member? v V) (member? v guard-free-vars))
4033            (loop more (add v previous-args) results))
4034          (else (loop more (add v previous-args) (add v results))))))
4035       ((list-of arg more)
4036        (loop more (add arg previous-args) results))))))
4037   (loop left-args [] []))
4038
4039
4040 (define (econd f eqns)
4041   (let ((eqns (map rename eqns))
4042         (N (arity-of f))
4043         (g (get-symbol f))
4044         (freshvars (map (lambda (_) (fresh-var)) (1 to N)))
4045         (get-cond (lambda (eqn)
4046           (match eqn
4047             ((forall (some-list uvars) (if (some-sent p) (= (left as ((val-of g) (some-list args))) R)))
4048             (let ((pfv (free-vars p))
4049                   (vars-to-discard (vars-to-rid args pfv))
4050                   (evars (filter-out (dedup (join (vars left) pfv))
4051                     (lambda (v)
4052                       (member? v vars-to-discard)))))
4053             (conds1 (map-select
4054               (lambda (var-arg-pair)

```

[illegible]

```

4125         (lambda (v)
4126           (member? v vars-to-discard)))
4127     (conds1 (map-select
4128       (lambda (var-arg-pair)
4129         (match var-arg-pair
4130           (([v a] where (negate (member? a vars-to-discard))) (= v a))
4131           (_ ())))
4132       (list-zip freshvars args)
4133       (unequal-to ())))
4134     (and-conds (join conds1))
4135     (body (match and-conds
4136       ([ true)
4137        ([ (some-sent q)] q)
4138        (_ (and and-conds))))))
4139     (match evars
4140       ([ body)
4141        (_ (exists* evars body))))))
4142 (make-cond-2 (lambda (freshvars eqn1 eqn2)
4143   (forall* freshvars
4144     (if (make-cond freshvars eqn1)
4145         (not (make-cond freshvars eqn2))))))
4146 (letrec ((loop (lambda (eqns res)
4147   (match eqns
4148     ([ (rev res))
4149      ((list-of eqn more) (let ((res' (map (lambda (eqn2)
4150                                         (make-cond-2 freshvars eqn eqn2))
4151                                         more)))
4152        (loop more (join res' res)))))))
4153   (loop eqns [])))
4154
4155 (define (fun-def-conds-e f)
4156   (match f
4157     ((some-symbol _) (econd f (defining-axioms f)))
4158     ((some-proc _)
4159      (let ((fsym (root (app-proc f (map (lambda (__) (fresh-var)) (from-to 1 (arity-of f))))))
4160        (econd fsym (defining-axioms fsym)))))
4161
4162 (define get-ec fun-def-conds-e)
4163
4164 (define (fun-def-cond-d0 f)
4165   (let ((res (match (dcond f (defining-axioms f))
4166     ([ (some-sent p)] [p])
4167     ([ true])
4168     ((some-list L) L)))
4169     (clean-up (lambda (res)
4170       (match res
4171         ((forall (some-list uvars) body)
4172          (let ((body-fv (free-vars body))
4173            (uvars' (filter uvars (lambda (v) (member? v body-fv))))
4174            (forall* uvars' body))))))
4175       (map clean-up res)))
4176
4177 (define (fun-def-conds-d f)
4178   (match f
4179     ((some-symbol _) (fun-def-cond-d0 f))
4180     ((some-proc _)
4181      (let ((fsym (root (app-proc f (map (lambda (__) (fresh-var)) (from-to 1 (arity-of f))))))
4182        (fun-def-cond-d0 fsym))))
4183
4184 (define get-dc fun-def-conds-d)
4185
4186 (define (def-obligations f)
4187   (let ((p (get-ec f))
4188     ([error at_least_one_conditional_equation] (check-fun-def f))
4189     (_ (check ((&& (negate error) (negate at_least_one_conditional_equation))
4190       (print "\nThe definition of " f " is well-formed; the following conditions hold.\n"))
4191       (else ())))
4192     (qs (get-dc f)))
4193   (add p qs))
4194

```

```

4195
4196 (define fun-def-conds def-obligations)
4197
4198 (define (spf s props N)
4199   (!sprove-from s props [['poly true] ['subsorting false] ['max-time N]]))
4200
4201 (define (vpf s props N)
4202   (!vprove-from s props [['poly true] ['subsorting false] ['max-time N]]))
4203
4204 (define (prove goal premises)
4205   (!spf goal premises 100))
4206
4207 define (derive-from goal premises options) :=
4208   let {max := try {(options 'max-time) | 100};
4209       c := try {(options 'used-premises) | () }}
4210   check {(equal? (options 'atp) 'vampire) =>
4211     match c {
4212       (some-cell _) => (!vprove-from goal premises [c ['poly true] ['subsorting false] ['max-time max]])
4213       | _ => (!vpf goal premises max)}
4214   | else => (!spf goal premises max)}
4215
4216 define top-smt-solve := smt-solve
4217
4218 define make-all-terms-thunk-cell := (cell ())
4219
4220 module SMT {
4221
4222   define (in x range) :=
4223     match range {
4224       [l h] => ((x <= h) & (l <= x))
4225     }
4226
4227   define (check-option-key-and-value k v) :=
4228     let {error := lambda (k v) (error (join "\nWrong key-value entry given as an option: " (val->string k) " := " (val->string v)))}
4229     match k {
4230       'results => match v { (some-table _) => () | _ => (error k v) }
4231       | 'solver => check {(member? v ['yices 'cvc]) => () | else => (error k v)}
4232       | _ => ()
4233     }
4234
4235   #=====
4236
4237   define (basic-augment-answer-table answer-table eqn) :=
4238     match eqn {
4239       (= l r) => let {prior-terms := try {(HashTable.lookup answer-table l) | [] }}
4240         (HashTable.add answer-table [l --> (add r prior-terms)])
4241     }
4242
4243   define (get-ints-from-code code-string) :=
4244     let {toks := (tokenize code-string " \t-")();
4245         numerals := (dedup (filter toks all-digits?))}
4246     (join (map string->num numerals)
4247       (map lambda (n) (- (string->num n)) numerals))
4248
4249   define (get-reals-from-code code-string) :=
4250     let {toks := (tokenize code-string " \t-")();
4251         numerals := (dedup (filter toks lambda (t) (for-each t lambda (c) (|| (digit? c) (equal? c \.))))}
4252     (join (map string->num numerals)
4253       (map lambda (n) (- (string->num n)) numerals))
4254
4255   define (get-metatypes-from-code code-string) :=
4256     let {toks := (tokenize code-string " \t")();
4257         ids := (dedup (filter toks lambda (t) try { (seq (id->string (evaluate t)) true) | false })))}
4258     (map (o id->string evaluate) ids)
4259
4260   define (non-datatype-sort D) := (negate (datatype-sort? D))
4261
4262   define (make-all-equations fsym code-string) :=
4263     let {input-domains := (all-but-last (get-signature fsym));
4264         #_ := (print "\nAbout to make all equations for fsym: " fsym ", with input-domains: " (separate input-domains

```

```

4265     make-all-terms := (ref make-all-terms-thunk-cell);
4266     make-all-terms' := lambda (sort)
4267         check {(equal? sort "Int") => (get-ints-from-code code-string)
4268             | (equal? sort "Real") => (get-reals-from-code code-string)
4269             | (equal? sort "Ide") => (get-metataids-from-code code-string)
4270             | else => (make-all-terms sort)}}
4271     check {(=& false (for-some input-domains non-datatype-sort)) => []
4272         | else => let {all-inputs := (cprod* (map make-all-terms' input-domains));
4273             #_ := (print "\n\tLength of all-inputs for symbol " fsym " is: " (length all-inputs));
4274             get-answer := lambda (input)
4275                 try {
4276                     let {inputs := (separate (map val->string input) " ");
4277                         exp := (join "(" code-string inputs ")");
4278                         result := (evaluate exp)}
4279                     (= (make-term fsym input) result)
4280                     | () };
4281             results := (map-select get-answer all-inputs (unequal-to ()));
4282             #_ := (print "\n\tThere are " (length results) " resulting equations for symbol " fsym);
4283             #_ := (print "\n\tAnd the results:\n" (separate (map val->string results) "\n"));
4284             _ := ()
4285         }
4286     results}
4287
4288 define (cvc-code->equations code-table answer-table) :=
4289     (map-proc lambda (key-val-pair)
4290         match key-val-pair {
4291             [fsym code] => let {eqns := (make-all-equations fsym code)}
4292                 (map-proc lambda (eqn) (basic-augment-answer-table answer-table eqn) eqns)
4293         }
4294     (HashTable.table->list code-table))
4295
4296 define augment-cvc-answer-table :=
4297     lambda (answer-table)
4298     let {#_ := (print "\n\tEntering augment-cvc-answer-table, size of incoming answer-table: " (HashTable.size answer-table));
4299         L := (table->list answer-table);
4300         code-table := (HashTable.table);
4301         _ := (map-proc lambda (key-val-pair)
4302             match key-val-pair {
4303                 [(some-list f-str) (some-list code)] where (& (string? f-str) (string? code)) =>
4304                     let {_ := (HashTable.remove answer-table f-str);
4305                         f := (string->symbol f-str)}
4306                     (HashTable.add code-table [f --> code])
4307                 | _ => ()
4308             }
4309             L);
4310         #_ := (print "\n\tHere's code-table:\n" code-table "\n\tand here's answer-table:\n" answer-table);
4311         _ := (cvc-code->equations code-table answer-table);
4312         #_ := (print "\n\tExiting augment-cvc-answer-table, size of outgoing answer-table: " (HashTable.size answer-table));
4313         _ := ()
4314     }
4315     ()
4316
4317 define (extract-cvc-extension-table answer-table) :=
4318     let {extension-table := (HashTable.table);
4319         L := (HashTable.table->list answer-table);
4320         _ := (map lambda (key-val-pair)
4321             match key-val-pair {
4322                 [(some-term l) where (ground? l)) term-list] =>
4323                     match (dedup term-list) {
4324                         [(some-term r)] => (HashTable.add extension-table [l --> r])
4325                         | res => () # (print "\n\tCould not extract a sole answer for the term " l, got this instead:
4326                             res)
4327                     }
4328             L)}
4329     extension-table
4330
4331 define (smt-solve C options) :=
4332     let {ht := (HashTable.table);
4333         _ := (map-proc lambda (k)
4334             let {_ := (check-option-key-and-value k (options k))}
```

```

4335         (HashTable.add ht [k --> (options k)])
4336         (Map.keys options));
4337     #_ := (print "\nHere's ht right before the solving: " ht);
4338     #_ := (print "\nAnd here's options right before the solving: " options);
4339     res := (top-smt-solve C ht);
4340     #_ := (print "\nAnd here's ht right AFTER the solving: " ht);
4341     #_ := (print "\nAnd here's options right after the solving: " options);
4342     post-process-cvc-answers := lambda (t)
4343         let {#_ := (print "\nHere's (downcase-string (options 'solver)): " (downcase-string
4344             cond1 := (prefix? "cvc" (downcase-string (val->string (options 'solver))));
4345             cond := (&& cond1
4346                 (negate (prefix? "un" (downcase-string (val->string t))))))}
4347         try {check {cond =>
4348             let {#_ := (print "\nHere's (options 'results'): " (options 'results'))
4349                 _ := (augment-cvc-answer-table (options 'results))
4350             }
4351             t
4352             | else => t}
4353         | t}}
4354     match res {
4355         (some-term t) => let {#_ := (print "\nThe main call to smt-solve produced this TERM: " (val->string t));
4356             _ := (post-process-cvc-answers t)}
4357         t
4358         | _ => let {#_ := (print "\nThe main call to smt-solve produced this result: " (val->string res));
4359             final-res := 'Satisfiable;
4360             _ := (post-process-cvc-answers final-res)}
4361         final-res
4362     }
4363
4364 define (replace-all-subterms t ht) :=
4365     check {(| (meta-id? t) (numeral? t)) => t
4366     | else =>
4367         try { (first (table-lookup ht t))
4368             | match t {
4369                 ((some-symbol f) (some-list args)) =>
4370                     (make-term f (map lambda (s) (replace-all-subterms s ht)
4371                         args))
4372                 | _ => t
4373             }
4374         }
4375
4376 define (solve p) :=
4377     let {ht := (table)}
4378     match (top-smt-solve p (table [['results --> ht] ['solver --> 'yices]])) {
4379         (some-term t) => t
4380         | _ => let {pairs := (map lambda (pair)
4381             match pair {
4382                 [x vals] => [x (first vals)]
4383             }
4384             (filter (table->list ht)
4385                 lambda (p) (var? first p)))};
4386         pairs' := (map (lambda (p) [(first p) (replace-all-subterms (second p) ht)]) pairs)}
4387     (make-sub pairs')
4388
4389
4390 define built-in-symbols :=
4391     (map string->symbol ["<" ">" "<=" ">=" "=" "+" "-" "*" "/"])
4392
4393 define (built-in? f) := (member? f built-in-symbols)
4394
4395 define (cost-term-leaves cost-term) :=
4396     match cost-term {
4397         ((some-symbol f) (some-list args)) =>
4398             check {(built-in? f) => (flatten (map cost-term-leaves args))
4399                 | else => [cost-term]}
4400         | _ => [cost-term]
4401     }
4402
4403 define (apply-solution sub s) :=
4404     let {res := (sub s)}

```

```

4405     match (res equal? s) {
4406       true => ()
4407       | _ => res
4408     }
4409
4410   define (apply-solution-new ht s) :=
4411     let {res := try {(HashTable.lookup ht s) | true}}
4412     match res {
4413       true => ()
4414       | (list-of h _) => h
4415       | (some-term t) => t
4416       | v => let {_ := (print "\nGot the following value by applying ht to the term " s ": " v)}
4417         ()
4418     }
4419
4420   define (get-cost solution cost-terms) :=
4421     let {costs := (map lambda (cost-term)
4422       (apply-solution solution cost-term)
4423       cost-terms)}
4424     (eval (foldl + 0 costs))
4425
4426   define (get-cost-new ht cost-terms) :=
4427     let {#_ := (print "\nCost-terms: " cost-terms);
4428       costs := (map lambda (cost-term)
4429         (apply-solution-new ht cost-term)
4430         cost-terms);
4431       #_ := (print "\nAnd their costs: " costs);
4432       _ := ()}
4433     (eval (foldl + 0 costs))
4434
4435   define (midpoint l h) := (l plus ((h minus l) div 2))
4436
4437   define (solve-and-minimize constraint cost-term max-cost) :=
4438     let {counter := (cell 0);
4439       cost-terms := (cost-term-leaves cost-term)}
4440     letrec {loop := lambda (lo hi)
4441       let {_ := (inc counter)}
4442       check {(hi less? lo) => 'Unsatisfiable
4443         | (lo equal? hi) => (solve (and constraint (cost-term = hi)))
4444         | else => let {mid := (lo midpoint hi);
4445           cost-constraint := (cost-term in [lo mid])}
4446           match (solve (and constraint cost-constraint)) {
4447             (some-sub sub) => check {(less? lo mid) =>
4448               let {total-cost := (get-cost sub cost-terms)}
4449               (loop lo total-cost)
4450             | else => sub}
4451             | _ => (loop (plus mid 1) hi)
4452           }}}
4453     match (loop 0 max-cost) {
4454       (some-sub sub) => let {_ := (print "\nTotal cost: " (get-cost sub cost-terms) "\n")}
4455         sub
4456     }
4457     | res => res
4458   }
4459
4460   define (range->string a b suffix) :=
4461     (join "[" (val->string a) "," (val->string b) "]" suffix)
4462
4463   define (solve-and-minimize-new constraint cost-term max-cost options) :=
4464     let {counter := (cell 0);
4465       cost-terms := (cost-term-leaves cost-term);
4466       get-minutes := lambda (s)
4467         (div s 60.0);
4468       solve := lambda (core-constraint cost-constraint)
4469         let {#_ := (print "\nSolving with cost constraint: " cost-constraint "\n");
4470           _ := (HashTable.clear (options 'results));
4471           #_ := (print "\nEntering solver...");
4472           res := (smt-solve (and core-constraint cost-constraint) options);
4473           #_ := (print "\nExiting solver...");
4474         }

```



```

4475         _ := (augment-cvc-answer-table (options 'results)))
4476     res}
4477     letrec {loop := lambda (lo hi iteration)
4478         let {_ := (inc counter);
4479             _ := (print (join "\n----- Iteration #" (val->string iteration) " -----\n"))}
4480         check {(hi less? lo) => 'Unsatisfiable
4481             | (lo equal? hi) => let {_ := (print "Cost constraint specifying that the cost term is in "
4482                 (solve constraint (cost-term = hi))
4483             | else => let {mid := (lo midpoint hi);
4484                 #_ := (print "\nGiven lo: " lo ", hi: " hi ", and midpoint: " mid ".");
4485                 _ := (print (join "Cost constraint specifying that the cost term is in " (ra
4486                     cost-constraint := (cost-term in [lo mid]));
4487                 t1 := (time);
4488                 solver-result := (solve constraint cost-constraint);
4489                 t2 := (time);
4490                 elapsed := (get-minutes (minus t2 t1))}
4491                 match solver-result {
4492                     'Unsatisfiable => let {_ := (print "\nFailed in " elapsed
4493 " minutes, will now try again with a properly adjusted range...\n")}
4494                         check {(iteration equals? 1) => 'Unsatisfiable
4495                             | else => (loop (plus mid 1) hi (plus iteration 1))}
4496                     | 'Satisfiable => check {(less? lo mid) =>
4497                         let {total-cost := (get-cost-new (options 'results) c
4498                             _ := (print (join "\nSuccess in " (val->string e
4499                                 (val->string total-cost)
4500                                     ", will now try again with a proper
4501                                     (loop lo total-cost (plus iteration 1))
4502                             | else => 'Satisfiable)
4503                     | other => let {str := (take (val->string other) 200);
4504                         _ := (print (join "\nIndeterminate result after " (val->string c
4505 " minutes: " str));
4506                         _ := (print ", will now try again with a properly adjusted rang
4507                             (loop (plus mid 1) hi (plus iteration 1))
4508                     }}}
4509         match (loop 0 max-cost 1) {
4510             'Satisfiable => let {_ := (print "\nTotal cost: " (get-cost-new (options 'results) cost-terms) "\n")}
4511             'Satisfiable
4512             | res => res
4513         }
4514     }
4515
4516     define (holds? p) :=
4517         match (solve (not p)) {
4518             'Unsatisfiable => true
4519             | _ => false
4520         }
4521
4522     set-precedence solve 2
4523     set-precedence holds? 2
4524
4525     (define (sc->string sc)
4526         (match sc
4527             (and "and")
4528             (or "or")))
4529
4530     (define (make-constraint sc strings)
4531         (let ((sc-string (sc->string sc)))
4532             (match strings
4533                 ([s] s)
4534                 (_ (join lparen sc-string blank strings))))))
4535
4536     (define (sum-all terms)
4537         (match terms
4538             ([x] x)
4539             ((list-of x (bind rest (list-of _ _))) (+ x (sum-all rest)))))
4540
4541     (define (sum n)
4542         (check ((less? n 1) 0)
4543             (else (plus n (sum (minus n 1))))))
4544
4545     (define (make-cost-term t)

```

```

4543 (match t
4544   ((some-var x) (string->var (join "cost" (var->string x) ":Int"))
4545    ((some-symbol f) (some-list _) (string->var (join "cost" (symbol->string f) ":Int")))))
4546
4547 (define (make-constraint n)
4548   (let ((span (from-to 1 n))
4549         (vars (map (lambda (l) (fresh-var "Int")) span))
4550         (counter (cell 1))
4551         (cost (cell 1))
4552         (range-sentences-and-var-values-1
4553          (map (lambda (v)
4554                 (let ((low ((inc counter) times 10))
4555                     (hi (plus low 5)))
4556                   [(in v [low hi])
4557                    (= v (plus low 1))]))
4558                vars))
4559         (range-sentences-and-var-values-2
4560          (map (lambda (v)
4561                 (let ((low ((inc counter) times 100))
4562                     (hi (plus low 10)))
4563                   [(in v [low hi])
4564                    (= v (plus low 2))]))
4565                vars))
4566         ([range-sentences-1 var-values-1] (unzip range-sentences-and-var-values-1))
4567         ([range-sentences-2 var-values-2] (unzip range-sentences-and-var-values-2))
4568         (constraint (or (and* range-sentences-1) (and* range-sentences-2)))
4569         (mid (midpoint 1 n))
4570         (values-1 (take var-values-1 mid))
4571         (values-2 (second (split-list var-values-2 mid)))
4572         (values (join values-1 values-2))
4573         (cost-constraints (map (lambda (var-val)
4574                                  (match var-val
4575                                    ((= v val) (let ((v-cost-term (make-cost-term v)))
4576                                                    (ite (= v val) (= v-cost-term 0) (= v-cost-term (inc cost))))))
4577                                values))
4578         (cost-variables (map make-cost-term vars))
4579         (cost-term (sum-all cost-variables))
4580         (cost-constraint (and* cost-constraints))
4581         (max-cost (sum (length vars))))
4582   [constraint vars cost-constraint cost-term max-cost]))
4583
4584 (define [constraint-30 vars-30 cost-constraint-30 cost-term-30 max-cost-30] (make-constraint 30))
4585
4586 (define [constraint-100 vars-100 cost-constraint-100 cost-term-100 max-cost-100] (make-constraint 100))
4587
4588 # (running-time (lambda () (solve-and-minimize (and constraint-30 cost-constraint-30) cost-term-30 max-cost-30)) 0)
4589
4590 # (solve-and-minimize (and constraint-100 cost-constraint-100) cost-term-100 max-cost-100)
4591
4592 }
4593
4594 EOF
4595 (println (cd [N.Plus] ["N"]))
4596 (define css compile-symbols-simple)
4597 (define csd compile-symbols-with-default)
4598
4599 (println (css [append] []))
4600 (println (cs [append] []))
4601
4602 (println (css [mem] []))
4603
4604 (println (css [I] []))
4605
4606 (define ri (iff (= ?A_1062
4607                  null)
4608                (forall ?v1080:'S
4609                  (iff (in ?v1080
4610                          ?A_1062)
4611                        (in ?v1080
4612                            null)))))
4612

```

```
4613
4614
4615 (define sri (iff (= ?A_1062
4616   null)
4617   (forall ?v1085:'T
4618     (iff (in ?v1085
4619       ?A_1062)
4620       (in ?v1085
4621         null))))))
4622
4623 (assume ri
4624   (!sort-instance ri sri))
4625
4626 (load-file "cc")
```