lib/basic/fundef.ath

```
(define prover vprove-from)

(define (make-eqn0 c l r)
  (match c
    (true (close (= l r)))
    ((and [p]) (close (if p (= l r))))
    (_ (close (if c (= l r))))))

(define (make-combinations L1 L2)
   (cprods L1))

(define (con-term? t uvars)
  (match t
    ((((some-symbol _) (some-list _)) where (&& (for-each (syms t) constructor?) (subset? (vars t) uvars))) true)
    (_ false)))

(define (dt-inequality? p uvars lvars)
  (match p
    ((not (= (some-term s) (some-term t)))
           (&& (con-term? t uvars)))
    (_ false)))


(define (collect lhsides)
  (let ((T (table 7))
        (_ (map-proc (lambda (lhs)
                       (match lhs
                         ((not (= l r)) (table-add T [l --> (add r (try (table-lookup T l) []))])))))
                     lhsides)))
    (table->list T)))


(define [cc lc rc] [(cell ()) (cell ()) (cell ())])

(define (make-id p)
  (match p
    ([(some-term s) (some-term t)] (= s t))))

(define (complement-constructors L)
  (let ((allowed-constructors (list-diff (constructors-of (sort-of (first L)))
                                         (map root L))))
   (map (lambda (c) (let ((t (make-term c (map (lambda (_) (fresh-var)) (from-to 1 (arity-of c))))))
                      (lhs (= t (first L)))))
        allowed-constructors)))

(define (apply-case L p)
  (let ((sub (make-sub (map (lambda (id)
                              [(lhs id) (rhs id)]) L))))
    (sub p)))


(define (make-subs ac lvars)
  (let ((T (table 500))
        (_ (map-proc (lambda (e)
                       (let ((v (lhs e)))
                          (table-add T [v --> (add (rhs e) (try (table-lookup T v) []))])))
                     ac))
        (list (map (lambda (p) [(first p) (rd (second p))])
                   (table->list T))))
   (flatten (map (lambda (pair)
                   (match pair
                      ([lside rsides] (map (lambda (rside) (make-sub [[lside rside]])) rsides))))
          list))))

(define (possibly-sole-disjunction? p)
  (match p
    ((some-sent _) true)
    (_ false)))
```

```
69
70  (define (wildcard-clause? l)
71    (match l
72      ((((some-symbol _) (some-list args)) where (for-each args var?)) true)
73      (_ false)))
74
75  (define (non-explicit-wildcard-patterns?)
76    (match (get-flag "explicit-wildcard-patterns")
77      ("0" true)
78      (_ false)))
79
80  (define (grp L) (flatten (join (map (lambda (p) (get-remaining-patterns [p])) L))))
81
82  (define (make-eqn c l r previous-lhsides)
83   (try
84    (let (##(_ (print "\nCalling make-eqn on c: " c "\nand l: " l "\nand r: " r "\nand previous-lhsides:
    " previous-lhsides "\n"))
85          (lvars (vars l)))
86      (match c
87        (((forall (some-list uvars) (body as (|| (and (some-list props))
88                                                 (some-sent prop))))
89                where (let ((props (try props [prop])))
90                        (for-each props (lambda (p) (&& (possibly-sole-disjunction? p)
91                                                        (for-each (get-disjuncts p)
92                                                                  (lambda (d) (dt-inequality? d uvars lvars))))))))
93         (check ((wildcard-clause? l)
94                 (match (get-flag "explicit-wildcard-patterns")
95                   ("1" (let (#(_ (print "\nWILDCARD CLAUSE DETECTED!\n"))
96                              (rem-pats (get-remaining-patterns previous-lhsides)))
97                              (map (lambda (pat)
98                                     (close (= pat r)))
99                                   rem-pats)))
100                  ("2" (let (#(_ (print "\nWILDCARD CLAUSE DETECTED!\n"))
101                             (rem-pats (get-remaining-patterns previous-lhsides)))
102                             (join [(make-eqn0 c l r)]
103                                   (map (lambda (pat)
104                                          (close (= pat r)))
105                                        rem-pats))))
106                 ("0" [(make-eqn0 c l r)])))
107           ((non-explicit-wildcard-patterns?)  [(make-eqn0 c l r)])
108           (else
109          (let ((_ ())
110                (props (try props [prop]))
111  #              (_ (print "\nLeft-hand side " l " is not a wildcard clause...\n"))
112  #              (_ (print "\nc: " c "\nl: " l "\nr: " r "\n"))
113                (_ (seq (set! lc l) (set! rc r) (set! cc c)))
114  #             (_ (print "\nl: " l))
115  #             (_ (print "\nlvars: " lvars))
116                (combinations (make-combinations (map get-disjuncts props) lvars))
117                (process-inequalities
118                    (lambda (inequalities)
119                       (let ((lhsides (collect inequalities))
120                             (allowed-eqns (map (lambda (p)
121                                                  [(first p) (let ((pats (second p)))
122                                                               (join (grp pats)
123                                                                     (complement-constructors pats)))])
124                                                lhsides))
125                             (allowed-rhsides (cprods (map second allowed-eqns)))
126                             (allowed-lhsides (map first allowed-eqns)))
127                         (flatten (map (lambda (r)
128                                         (map make-id (match allowed-lhsides
129                                                        ([_] (cprod allowed-lhsides r))
130                                                        (_   (list-zip allowed-lhsides r)))))
131                                       (map (lambda (r)
132                                              (match r
133                                                 ((some-list _) r)
134                                                 (_ [r])))
135                                            allowed-rhsides))))))
136                (allowed-cases (map process-inequalities combinations))
137                (resulting-equations (table 500))
```

```
138               (_ (map-proc (lambda (ac)
139                              (match ac
140                                ([] ())
141                                (_ (let ((subs (make-subs ac lvars)))
142                                     (map-proc (lambda (sub)
143                                                 (let ((eq (close (sub (= l r)))))
144                                                   (table-add resulting-equations [eq --> true])))
145                                               subs)))))
146                            allowed-cases)))
147            (check ((equal? (get-flag "explicit-wildcard-patterns") "1") (map first (table->list resulting-equations)))
148                   (else (join [(make-eqn0 c l r)] (map first (table->list resulting-equations)))))))))
149     (_ (seq  [(make-eqn0 c l r)])))
150     (seq [(make-eqn0 c l r)])))
151
152
153 (define (apply-equation eqn t)
154   (match eqn
155      ((= (some-term left) (some-term right)) (replace-term-in-term left t right))
156      (_ t)))
157
158 (define (apply-equations eqns t)
159   (letrec ((loop (lambda (eqns res)
160                    (match eqns
161                      ([] res)
162                      ((list-of e more) (loop more (apply-equation e res)))))))
163     (loop eqns t)))
164
165 (define (make-eqn' c l r previous-lhsides)
166  (try
167   (let ( ## (_ (print "\nCalling make-eqn on c: " c "\nand l: " l "\nand r: " r "\nand previous-lhsides:
    " previous-lhsides "\n"))
168         (lvars (vars l)))
169     (match c
170       (((forall (some-list uvars) (body as (|| (and (some-list props))
171                                                (some-sent prop))))
172                 where (let ((props (try props [prop])))
173                          (for-each props (lambda (p) (&& (possibly-sole-disjunction? p)
174                                                          (for-each (get-disjuncts p)
175                                                                    (lambda (d) (dt-inequality? d uvars lvars))))))))
176        (check ((wildcard-clause? l)
177                (match (get-flag "explicit-wildcard-patterns")
178                  ("1" (let (#(_ (print "\nWILDCARD CLAUSE DETECTED!\n"))
179                             (rem-pats (get-remaining-patterns previous-lhsides)))
180                             (map (lambda (pat)
181                                    (close (= pat r)))
182                                  rem-pats)))
183                  ("2" (let (#(_ (print "\nWILDCARD CLAUSE DETECTED!\n"))
184                             (rem-pats (get-remaining-patterns previous-lhsides)))
185                             (join [(make-eqn0 c l r)]
186                                   (map (lambda (pat)
187                                          (close (= pat r)))
188                                        rem-pats))))
189                  ("0" [(make-eqn0 c l r)])))
190          ((non-explicit-wildcard-patterns?)  [(make-eqn0 c l r)])
191          (else
192         (let ((_ ())
193               (props (try props [prop]))
194 #              (_ (print "\nLeft-hand side " l " is not a wildcard clause...\n"))
195 #              (_ (print "\nc: " c "\nl: " l "\nr: " r "\n"))
196               (_ (seq (set! lc l) (set! rc r) (set! cc c)))
197 #             (_ (print "\nl: " l))
198 #             (_ (print "\nlvars: " lvars))
199 ## Each member of all-inequalities is now a list of of the form [s1 =/= t1 ... sn =/= tn]
200               (all-inequalities (map get-disjuncts props))
201               (unwanted-equations (map (lambda (inequality)
202                                          (map complement inequality))
203                                        all-inequalities))
204 ## So now each member of unwanted-equations is a list of the form [s1 = t1 ... sn = tn]
205               (make-pattern (lambda (unwanted-equation)
206                               (apply-equations unwanted-equation l)))
```

```
207               (unwanted-patterns (map make-pattern unwanted-equations))
208               (remaining-pats-and-subs (map-select
209                                         (lambda (pat)
210                                            (match (unify pat l)
211                                              ((some-sub sub) [pat sub])
212                                              (_ ())))
213                                         (get-remaining-patterns unwanted-patterns)
214                                         (unequal-to ())))
215               (all-remaining-equations (map (lambda (pair)
216                                               (match pair
217                                                 ([pat sub] (close (sub (= pat r))))))
218                                             remaining-pats-and-subs)))
219            (check ((equal? (get-flag "explicit-wildcard-patterns") "1") all-remaining-equations)
220                   (else (join [(make-eqn0 c l r)] all-remaining-equations)))))))
221     (_ (seq  [(make-eqn0 c l r)])))
222     (seq [(make-eqn0 c l r)])))
223
224 (define (make-eqn c l r previous-lhsides)
225   (let ((res (make-eqn' c l r previous-lhsides)))
226     res))
227
228 (define decompose-equation'
229   (lambda (eqn)
230     (match (rename eqn)
231       ((forall (some-list uvars) (if guard (= pattern res))) [pattern guard res])
232       ((forall (some-list uvars) (= pattern res)) [pattern () res]))))
233
234
235 (define (make-equivalence-classes triples)
236   (letrec ((loop (lambda (remaining-triples classes-so-far)
237                    (match remaining-triples
238                      ([] (rev classes-so-far))
239                      ((list-of (as triple [pat guard res]) more)
240                         (match (for-some' classes-so-far
241                                           (lambda (triple-list)
242                                              (unifiable pat (pat-of (first triple-list)))))
243                           ([classes-1 triple-list classes-2]
244                              (let ((classes' (join classes-1 [(join triple-list [triple])] classes-2)))
245                                (loop more classes')))
246                           (_ (loop more (add [triple] classes-so-far)))))))))
247       (loop triples []))))
248
249 (define (analyze equations)
250   (let ((all-triples (map decompose-equation' equations))
251         (list-of-classes (make-equivalence-classes all-triples))
252         (list-of-classes (map sort-class list-of-classes)))
253     (map sort-class (map process-equivalence-class list-of-classes))))
254
255 (define (conditionalize0 remaining-equations previous-lhsides results)
256                    (match remaining-equations
257                      ([] (rev results))
258                      ((list-of (l = r) more)
259                         (let ((c (diff* l previous-lhsides))
260                               (previous-vars (list-diff (vars c) (vars l)))
261                               (c (forall* (intersection (vars c) previous-vars) c)))
262                           (conditionalize0 more (add l previous-lhsides)
263                                                 (join (make-eqn c l r previous-lhsides) results))))
264                      ((list-of (g ==> (l = r)) more)
265                         (let ((c (diff* l previous-lhsides))
266                               (c-vars (vars c))
267                               (previous-vars (list-diff c-vars (vars l)))
268                               (c (forall* (intersection c-vars previous-vars) c))
269                               (cond (normalize (simp-and [g c])))
270                               (eqns' (make-eqn cond l r previous-lhsides))
271                               (previous' (match more
272                                            ((list-of (= _ _) _) (add l previous-lhsides))
273                                            (_ previous-lhsides))))
274                           (conditionalize0 more previous' (join eqns' results))))))
275
276 (define (conditionalize identities)
```

```
277    (conditionalize0 identities [] []))
278
279  (define (get-left-terms L)
280    (letrec ((loop (lambda (L res)
281                     (match L
282                        ((list-of (some-term s) (list-of or more)) (loop more (add s res)))
283                        ((list-of (some-term s) []) (rev (add s res)))))))
284      (loop L [])))
285
286  (define (proper-term? s)
287    (&& (term? s) (negate (equal? s -->))))
288
289  (define (proper left-hand-sides)
290    (for-each left-hand-sides (lambda (x) (|| (proper-term? x) (equal? x or)))))
291
292
293   (define (negate-guard guard lhs-list)
294    (let ((temp-prop (if guard (and (map (lambda (l) (= l (fresh-var))) lhs-list))))
295          (guard (antecedent temp-prop))
296          (temp-vars (list-diff (vars guard) (vars* lhs-list)))
297          (body (match guard
298                      ((((|| (= s pattern) (= pattern s)) where (&& (equal-lists-as-sets (vars pattern) temp-vars)
299                                                                    (equal-lists-as-sets [] (intersection temp-vars (vars
300                             (match pattern
301                               ((((some-symbol c) (some-list args)) where (&& (constructor? c) (for-each args var?)))
302                                     (match (list-remove c (constructors-of (constructor-range c)))
303                                       (([c'] where (equal? (arity-of c') 0)) (= s (c')))
304                                       (_ (not guard))))
305                               (_ (not guard))))
306                      (_ (not guard))))
307          (body' (match (get-flag "simplify-fun-defs")
308                    ("false" (not guard))
309                    (_ body)))
310          (condition (forall* (intersection temp-vars (vars body')) body')))
311      condition))
312
313  (define (simplify-condition' c rhs)
314    (match c
315      ((forall (some-list uvars) (|| (and (some-list conds)) (some-sent cond)))
316          (let ((conds (try conds [cond]))
317                (conds' (map app-dm-deep conds))
318                (constructor-equalities (filter conds' (lambda (c)
319                                                         (match c
320                                                           ((= (some-term s) (some-term t)) (&& (constructor? (root t))
321                                                           (_ false)))))
322                (lhsides (map lhs constructor-equalities))
323                (other-conds (list-diff conds' constructor-equalities))
324                 (other-conds' (filter-out other-conds
325                                 (lambda (cond)
326                                   (match cond
327                                     ((|| (or (some-list inequalities)) (some-sent inequality))
328                                      (let ((inequalities (try inequalities [inequality])))
329                                       (for-some inequalities
330                                          (lambda (i)
331                                             (match i
332                                               (((forall (some-list _) (not (= (some-term l) (some-term r))))
333                                                                                                  where (for-some constructor-equa
334                                                                                                         (lambda (e)
335                                                                                                           (match e
336                                                                                                             ((= (val-of l) t) (un
337                                                                                                             (_ false))))) true)
338                                                                                                  (_ false))))))
339                                     (_ false)))))
340                (all-conds (join constructor-equalities other-conds'))
341                (res (match all-conds
342                       ([p] p)
343                       (L (and L)))))
344            res))
345      (_ c)))
346
```

```
(define (simplify-condition c rhs)
  (match (get-flag "simplify-fun-defs")
     ("false" c)
     (_ (let ((res (try (simplify-condition' c rhs)
                        c))
##             (_ (print "\nOriginal condition: " c "\nSimplified condition: " res))
               (_ ()))
         res))))


(define (non-identity? x) (negate (identity? x)))

(define (simplify-clause' clause)
  (match clause
    ((if (|| (and (some-list conds))
             (some-sent cond))
         (body as (= (some-term lhs) (some-term rhs))))
      (let ((conds (try [cond] conds))
#             (_ (print "\nAll conds: " conds "\n"))
#             (_ (print "\nWhole clause: " clause))
            (V (vars lhs))
            (bindings (cell []))
            (eqns (cell []))
            (sub (let ((_ (map-proc (lambda (c)
                                      (match c
                                        ((= (some-var v) t) (seq (set! eqns (add c (ref eqns)))
                                                                 (set! bindings (add [v t] (ref bindings)))))
                                        (_ ()))))
                                    conds)))
                       (make-sub (ref bindings))))
           (sub-supp (supp sub))
#          (_ (print "\n(ref eqns): "  (ref eqns)))
           (non-equality-vars (filter (vars* conds)
                                      (lambda (v)
                                        (for-some conds (lambda (c)
                                                          (&& (non-identity? c) (member? v (vars c))))))))
           (movable-vars (cell []))
           (movable-eqns (filter (ref eqns)
                                 (lambda (e)
                                    (match e
                                       (((= (some-term s) (some-term t)) where (negate (member? s non-equality-vars)))
                                       (_ false)))))
           (conds' (list-diff conds movable-eqns))
           (supp' (filter sub-supp (lambda (v) (member? v (ref movable-vars)))))
           (sub (make-sub (zip supp' (sub supp'))))
           (body' (sub body))
           (conds' (filter-out conds'
                       (lambda (e)
                          (match e
                            (((= (some-var x) _) where (negate (member? x (vars body')))) true)
                            (((not (= (some-var x) _)) where (&& (negate (member? x (vars body'))) (negate (member? x
                            (_ false))))))
       (check ((subset? (supp sub) V)
                 (match conds'
                   ([] body')
                   ([true] body')
                   (_ (match conds'
                        ([(some-sent p)] (if p body'))
                        (_ (if (and conds') body')))))))
              (else clause))))))


(define
 (sent-cons p)
    (match p
      (((some-sent-con sc) (some-list args)) (add sc (sent-cons* args)))
      (((some-quant q) (some-list _) (some-sentence body)) (sent-cons body))
      (_ []))
 (sent-cons* props)
   (match props
```

```
      ([] [])
      ((list-of p more) (join (sent-cons p) (sent-cons* more)))))

(define (all-sent-cons p) (rd (sent-cons p)))

(define (entails? premises goal axioms)
  (let ((_ ())
#         (_ (print "\nInside entails...\n"))
#         (_ (print "\npremises: " premises "\ngoal: " goal "\naxioms: " axioms))
        (smt-goal (match premises
                    ([] goal)
                    ((some-list _) (if (and premises) goal))))
#         (_ (print "\nsmt-goal: " smt-goal))
        (_ ()))
    (check ((contains-quants? smt-goal)
                (let ((goal (if (and (join premises axioms)) goal))
                      (proved (cell false))
                      (_ (dtry (dlet ((_ (!prover goal [] [['poly true] ['subsorting false] ['max-time 3]]))
                                      (_ (set! proved true)))
                                 (!true-intro))
                               (!true-intro))))
                  (ref proved)))
           (else (match (smt-solve (not smt-goal) (table [['solver --> 'cvc]]))
                   ('Unsatisfiable true)
                   (_  false))))))

(define (equivalent? p1 p2 axioms)
  (let ((goal (make-monomorphic-instance (if (and axioms) (iff p1 p2))))
#         (_ (print "\nInside equivalent. Goal: " goal "\n"))
        (proved (cell false))
        (_ (dtry (dlet ((_ (!prover goal [] [['poly true] ['subsorting false] ['max-time 3]]))
                        (_ (set! proved true)))
                   (!true-intro))
                 (!true-intro))))
    (ref proved)))


(define (make-or props)
  (match props
    ([p] p)
    (_ (or props))))

(define (simplify-neg-cond p pos-conds axioms)
  (match p
    ((or (some-list _)) (let ((props (get-disjuncts p)))
                          (make-or (filter-out props (lambda (d) (entails? pos-conds (complement d) axioms))))))
    (_ p)))

(define (extract-sub conds)
  (letrec ((loop (lambda (rem-conds front-conds res)
                   (match rem-conds
                      ([] [front-conds res])
#                      (((list-of (= (some-var x) (some-term t)) more) where (null? (intersection (add x (vars t)) (va
                      (((list-of (= (some-var x) (some-term t)) more) where (negate (member? x (vars* (join front-cond
                           (loop more front-conds (add [x t] res)))
                      ((list-of c more) (loop more (add c front-conds) res))))))
     (let (([conds' bindings] (loop conds [] [])))
       [conds' (make-sub bindings)])))


(define (simplify-clause' clause)
  (match clause
    ((if (orig-cond as (|| (and (some-list conds))
                           (some-sent cond)))
         (body as (= (some-term lhs) (some-term rhs))))
      (let ((conds (try [cond] conds))
            (constructors (filter (get-prop-syms clause) constructor?))
            (axioms (rd (flatten (map datatype-axioms (map constructor-range constructors)))))
            ([pos-conds neg-conds] (filter-and-complement conds (lambda (c) (negate (member? not (all-sent-cons c))))))
            ([pos-conds' neg-conds'] [(map uquant-body pos-conds) (map uquant-body neg-conds)])
```

```
#            (_ (print "\nAbout to call entails on " (length neg-conds') " neg conds...\n"))
             (neg-conds' (filter-out neg-conds' (lambda (p) (entails? pos-conds' p axioms))))
#            (_ (print "\nDONE WITH ENTAILMENT...\n"))
             (neg-conds'' (map (lambda (nc) (simplify-neg-cond nc pos-conds' axioms)) neg-conds'))
             (new-conds (join pos-conds' neg-conds''))
             ([new-conds sub] (extract-sub new-conds))
             (final-new-cond (match new-conds
                               ([(some-sent p)] p)
                               ([] true)
                               (_ (and new-conds))))
             (new-clause (match new-conds
                           ([] (sub body))
                           ([true] (sub body))
                           (_ (if final-new-cond (sub body)))))
#            (_ (print "\nORIG CLAUSE: " clause))
#            (_ (print "\nNEW CLAUSE: " new-clause))
            (p1 (close clause))
            (p2 (close new-clause))
#           (_ (print "\np1: " p1 "\np2: " p2 "\naxioms: " axioms "\n"))
            (eq-test (try (equivalent? (close clause) (close new-clause) axioms) false))
#           (_ (print "\neq-test-result: " eq-test))
            (_ ()))
          (check (eq-test new-clause)
                 (else clause))))))

(define (simplify-clause clause)
  (match (get-flag "simplify-fun-defs")
     ("false" clause)
     (_ (let ((res  (try (simplify-clause' clause)
                         clause))
              (_ ()))
          res))))

(define (make-fresh-term e)
  (match e
    ((= l _) (let ((f (root l)))
               (make-term f (map (lambda (_) (fresh-var)) (from-to 1 (arity-of f))))))
    ((if _ c) (make-fresh-term c))))

(define (desugar-element x)
  (match x
    ((list-of 'case (list-of discrim rest))
        (let ((rest' (map desugar-element rest)))
          (letrec ((loop (lambda (rem result)
                           (match rem
                              ([] (rev result))
                              ((list-of left (list-of (id-op as (|| = -->)) more))
                                (loop more (add id-op (add (= discrim left) result))))
                              ((list-of x more) (loop more (add x result)))))))
             (loop rest' []))))
   ((list-of 'cond rest) (map desugar-element rest))
   ((some-list L) (map desugar-element L))
   (_ x)))

(define (desugar-list L)
   (map desugar-element L))


(define (make-new-list discrim match-clauses)
  (letrec ((loop (lambda (clauses res)
                   (match clauses
                     ([] res)
                     ([(left as (some-var _)) (|| = -->) right] (join res clauses))
                     ([or (left as (some-var _)) (|| = -->) right] (join res clauses))
                     ((split [(left as (some-term _)) (id-op as (|| = -->)) right] more) (loop more (join res [(= disc
                     ((split [or (left as (some-term _)) (id-op as (|| = -->)) right] more) (loop more (join res [(= d
                     ((split [(left as (some-term _)) (id-op as (|| = -->)) right] more) (loop more (join res [(= disc
                     ((split [or (left as (some-term _)) (id-op as (|| = -->)) right] more) (loop more (join res [(= d
      (loop match-clauses []))))

```

```
(define [case-of cond] ['case 'cond])

(define (fun-def-ids L)
  (letrec ((guard-ids (lambda (remaining previous-guards results bindings lhs-list)
                        (match remaining
                          ([] [results bindings])
                          ((|| (split [guard --> rhs] (list-of or more))
                               (split [guard --> rhs] [((some-term clause-name) where (meta-id? clause-name)) or] more)
                               (split [guard --> rhs] [((some-term clause-name) where (meta-id? clause-name))] more)
                               (split [guard --> rhs] more))
                            (let ((guard' (match guard
                                            ((some-var _) true)
                                            (_ guard)))
                                  (c (normalize (simp-and (add guard' (map (lambda (g) (negate-guard g lhs-list)) pre
                                  # (_ (print "\nNormalized guard condition to be simplified: " c "\nand the rhs: " rhs
                                  (c (simplify-condition c rhs))
                                  # (_ (print "\nAnd the simplified result: " c))
                                  (new-bindings (try (match lhs-list
                                                       ([lhs] (add [clause-name (simplify-clause (if c (= lhs rhs)))]
                                                       (_     (add [clause-name (map (lambda (lhs) (simplify-clause (
                                                     bindings)))
                              (guard-ids more (add guard previous-guards)
                                          (join (map (lambda (lhs) (simplify-clause (if c (= lhs rhs)))) lhs-lis
          (loop (lambda (L res bindings)
                  (match L
                    ([] [(rev res) (rev bindings)])


                    ((split [(some-var lv) (|| --> =) (some-term r)] more)
                       (let (#(_ (print "\nWILDCARD CLAUSE FOUND...\n"))
                             (_ (match more
                                  ((list-of _ _) (error "A wildcard pattern can only appear in the last clause."))
                                  (_ ()))))
                         (match (get-flag "mlstyle-fundef")
                           ("true" (match res
                                     ((list-of e _) (let ((_ ()))
                                                      (loop (join [(= (make-fresh-term e) r)] more) res bindings))
                           ("false" (let ((previous-lhsides (map lhs res)))
                                      (match res
                                        ((list-of e _) (let ((new-identity (= (make-fresh-term e) r))
                                                             (new-identities (conditionalize0 [new-identity] prev
                                                         (loop more (join new-identities res) bindings)))))))))


                    ((split [((|| = -->) l (some-term r))] (as nl (|| [] [((some-term n) where (meta-id? n))])))
(list-of or more))
                       (match nl
                         ([] (loop more (add (l = r) res) bindings))
                         ([(x where (meta-id? x))]
                             (loop more (add (l = r) res) (add [x (l = r)] bindings)))))
                    ((list-of ((|| = -->) l (some-term r)) more)
                        (match more
                          (((list-of x more') where (meta-id? x))
                             (loop more' (add (l = r) res) (add [x (= l r)] bindings)))
                          ((list-of (|| = -->) (list-of (some-term r') more'))
                             (loop more' (add (= (= l r) r') res) bindings))
                          (_ (loop more (add (l = r) res) bindings))))


                    (((split left-hand-sides [(|| --> =) (some-term r) or] more) where (proper left-hand-sides))
                       (let ((left-terms (get-left-terms left-hand-sides)))
                         (loop more (join (rev (map (lambda (left-term) (= left-term r)) left-terms)) res) bindings))
                    (((split left-hand-sides [(|| --> =) (some-term r) ((some-term n) where (meta-id? n)) or] more) wh
                       (let ((left-terms (get-left-terms left-hand-sides))
                             (eqns (map (lambda (left-term) (= left-term r)) left-terms))
                             (bindings' (match left-terms
                                          ([l] (add [n (= l r)] bindings))
                                          (_   (add [n eqns] bindings)))))
                         (loop more (join (rev eqns) res) bindings')))
                    (((split left-hand-sides [(|| --> =) (some-term r) ((some-term n) where (meta-id? n))] more) where
```

```
626                          (let ((left-terms (get-left-terms left-hand-sides))
627                                (eqns (map (lambda (left-term) (= left-term r)) left-terms))
628                                (bindings' (match left-terms
629                                             ([l] (add [n (= l r)] bindings))
630                                             (_   (add [n eqns] bindings)))))
631                            (loop more (join (rev (map (lambda (left-term) (= left-term r)) left-terms)) res)
632                                       bindings')))
633                       (((split left-hand-sides [(|| --> =) (some-term r)] more) where (proper left-hand-sides))
634                          (let ((left-terms (get-left-terms left-hand-sides)))
635                            (loop more (join (rev (map (lambda (left-term) (= left-term r)) left-terms)) res) bindings))
636
637
638                       (((split left-hand-sides [(id-op as (|| --> =)) (list-of 'case (list-of discrim match-clauses)) o
639                            (let ((new-list (make-new-list discrim match-clauses))
640                                  (L (join left-hand-sides [id-op] [new-list] [or] more)))
641                              (loop L res bindings)))
642
643                       (((split left-hand-sides [(id-op as (|| --> =)) (list-of 'case (list-of discrim match-clauses))] r
644                            (let ((new-list (make-new-list discrim match-clauses))
645                                  (L (join left-hand-sides [id-op] [new-list] more)))
646                              (loop L res bindings)))
647
648
649                       (((split left-hand-sides [(id-op as (|| --> =)) (list-of 'cond cond-clauses) or] more) where (prop
650                          (loop (join left-hand-sides [id-op] [cond-clauses] [or] more) res bindings))
651
652                       (((split left-hand-sides [(id-op as (|| --> =)) (list-of 'cond cond-clauses)] more) where (proper
653                          (loop (join left-hand-sides [id-op] [cond-clauses] more) res bindings))
654
655                       (((split left-hand-sides [(|| --> =) (some-list g) or] more) where (proper left-hand-sides))
656                          (let ((left-terms (get-left-terms left-hand-sides))
657                                ([eqns bindings'] (guard-ids g [] [] bindings left-terms)))
658                            (loop more (join eqns res) bindings')))
659
660                       (((split left-hand-sides [(|| --> =) (some-list g)] more) where (proper left-hand-sides))
661                          (let ((left-terms (get-left-terms left-hand-sides))
662                                ([eqns bindings'] (guard-ids g [] [] bindings left-terms)))
663                            (loop more (join eqns res) bindings')))))))
664     (loop L [] [])))
665
666
667 (define (get-symbol eqn)
668   (match eqn
669     ((forall (some-list _) (= ((some-symbol f) (some-list _)) _)) f)
670     ((forall (some-list _) (if _ (= ((some-symbol f) (some-list _)) _))) f)))
671
672 (define (classify eqns)
673   (letrec ((loop (lambda (eqns results)
674                    (match eqns
675                       ([] results)
676                       ((list-of e more) (let ((f (get-symbol e)))
677                                           (match results
678                                             ((split L1 [[(val-of f) f-eqns]] L2) (loop more (join L1 [[f (join f-eqns
679                                             (_ (loop more (add [f [e]] results)))))))))))
680     (loop eqns [])))
681
682 (define (redo-bindings eqns bindings)
683   (map (lambda (binding)
684          (match binding
685            ([name eqn] (let ((eqn-index (member-index eqn eqns)))
686                          [name eqn eqn-index]))))
687        bindings))
688
689 (define (find-bindings eqns bindings)
690   (map (lambda (binding)
691          (match binding
692            ([name old-eqn index] (let ((new-eqn (ith eqns index)))
693                                    [name new-eqn]))))
694        bindings))
695
```

```
696
697 (define (prim-list? L)
698   (letrec ((loop (lambda (L)
699                    (match L
700                       ((list-of (|| --> =) rest)
701                           (match rest
702                             ([] true)
703                             ((list-of (some-list _) _) false)
704                             ((list-of _ more) (loop more))))
705                       ((list-of x rest) (loop rest))
706                       (_ true)))))
707     (loop L)))
708
709 (define (transform-left L f)
710   (letrec ((loop (lambda (L res)
711                    (match L
712                      ([] (rev res))
713                      ((list-of l (list-of (id-op as (|| --> =)) rest))
714                         (loop rest (add id-op (add (f l) res))))
715                      ((list-of x more) (loop more (add x res)))))))
716     (loop L [])))
717
718 (define (transform-right L f)
719   (letrec ((loop (lambda (L res)
720                    (match L
721                      ([] (rev res))
722                      ((list-of left (list-of (id-op as (|| --> =)) (list-of right rest)))
723                         (loop rest (add (f right) (add id-op (add left res)))))
724                      ((list-of x more) (loop more (add x res)))))))
725     (loop L [])))
726
727 (define (flatten-prim-triple g id-op prim-list)
728   (transform-left prim-list (lambda (guard)
729                               (match guard
730                                 ((|| true (some-var _)) g)
731                                 (_ (and g guard))))))
732
733 (define (flatten-guard-list L)
734   (letrec ((loop (lambda (L)
735                    (match L
736                       ((list-of g (list-of (id-op as (|| --> =)) (list-of ((some-list gl) where (prim-list? gl)) rest
737                            (join (flatten-prim-triple g id-op gl) rest))
738                       ((list-of g (list-of (id-op as (|| --> =)) (list-of (some-list gl) rest)))
739                          (let ((gl' (loop gl)))
740                            (join (flatten-prim-triple g id-op gl') rest)))
741                       ((list-of g (list-of (id-op as (|| --> =)) (list-of t rest)))
742                         (join [g id-op t] (loop rest)))
743                       ((list-of x rest) (add x (loop rest)))
744                       (_ L)))))
745
746      (loop L)))
747
748 (define (get-neg-pat p D)
749   (match p
750     ((forall (some-list uvars) (not (= (val-of D) (some-term pat)))) pat)))
751
752 (define (split-diff-conds conds)
753  (try
754     (match (find-in-list conds identity?)
755        ([left-part id right-part]
756          (match id
757            ((= (some-term D) _)
758               (let ((neg-pats (map (lambda (p)
759                                      (get-neg-pat p D))
760                                    (join left-part right-part))))
761                 [id neg-pats])))))
762     ())))
763
764
765 (define (diff-simplify eqn)
```

```
  (match eqn
    ((forall (some-list uvars) (if (some-sent guard) (conclusion as (= (some-term left) (some-term right)))))
       (let ((conds (get-conjuncts guard))
             # (_ (print "\nconclusion: " conclusion))
             (_ ()))
         (match (split-diff-conds conds)
           ([(pos-cond as (= (some-term D) (some-term FORM))) (some-list neg-pats)]
              (let ((new-conds (diff* FORM neg-pats)))
                (check ((member? D (subterms left))
                        (forall* uvars (if (simp-and [new-conds]) (replace-term-in-term D conclusion FORM))))
                       (else (forall* uvars (if (simp-and (add pos-cond [new-conds])) conclusion))))))
           (_ eqn))))
    (_ eqn)))


(define (diff-simplify* eqns)
  (try (map diff-simplify eqns)
       eqns))

(define (fun-def L0)
  (let ((L (transform-right (desugar-list L0) flatten-guard-list))
        ([eqns bindings]
             (match (get-flag "mlstyle-fundef")
               ("true" (match  (fun-def-ids L)
                         ([eqns bindings] (let ((bindings' (redo-bindings eqns bindings))
                                                (eqns (diff-simplify* eqns))
                                                (partitions (classify eqns))
                                                (eqns' (flatten (join (map (lambda (partition) (conditionalize (secon
                                                (bindings'' (find-bindings eqns' bindings')))
                                           [eqns' bindings'']))))
               (_ (map close (fun-def-ids L))))))
    [(classify eqns) bindings]))

# For use with axioms resulting from fun-def of a Boolean-valued function:

define true-conv := (forall ?x . ?x = true ==> ?x)

conclude true-conv
  pick-any x
    assume A := (x = true)
      (!by-contradiction x
       assume (~ x)
         (!absurd
          (!true-intro)
          (!chain-last [(~ x) ==> (~ true) [A]])))

define false-conv := (forall ?x . ?x = false ==> ~ ?x)

conclude false-conv
  pick-any x
    assume A := (x = false)
      (!by-contradiction (~ x)
       assume x
         (!chain-last [x ==> false [A]]))
```