```
;* (build *dio.build-module-list*)
;* (build.compile *dio.build-module-list*)

(:= *dio.build-module-list* '(

    diophantine:de-normalize
    diophantine:de-solve
    diophantine:de-divide
    ) )

(:= *build-module-list* (append *build-module-list* *dio.build-module-list*) )
```

```lisp
;==============================================================================
;
; (DE:DIVIDE DEXPR1 DEXPR2)
;
; Divides normalized diophantine expressions DEXPR1 by DEXPR2 and returns
; the normalized result if the remainder is zero, () if there is a
; remainder.
;
;==============================================================================

(defun de:divide ( dexpr1 dexpr2 )
    (assert (&& (consp dexpr1)
                (== '& (car dexpr1) ) ) )
    (assert (&& (consp dexpr2)
                (== '& (car dexpr2) ) ) )

    (:= dexpr1 (cadr dexpr1) )
    (:= dexpr2 (cadr dexpr2) )

    (?       ;*** DEXPR1 = 0
             ;
      ( (= dexpr1 '(+ (* 0) ) )
        '(& (+ (* 0) ) ) )


             ;*** DEXPR2 = (+ C) or (+ 0 PROD)
             ;
      ( (|| (== 2 (length dexpr2) )
            (&& (== 3 (length dexpr2) )
                (== 0 (cadr (cadr dexpr2) ) ) ) )

        (let ( (prod2 (if (== 2 (length dexpr2) )
                          (cadr  dexpr2)
                          (caddr dexpr2) ) ) )

             (loop (for prod1 in (cdr dexpr1) )
                   (bind prod-result (ded.prod:divide prod1 prod2) )
                   (initial result () )
             (do
                 (if (! prod-result) (then
                     (return () ) ) )
                 (push result prod-result) )
             (result (de:normalize '(+ ,.result) ) ) ) ) )


             ;*** DEXPR1 = (+ C1 P11 P12 ... P1n)
             ;*** DEXPR2 = (+ C2 P21 P22 ... P2n)
             ;
      ( (&& (== (length dexpr1) (length dexpr2) )
            (> (length dexpr1) 2) )

        (let ( (result (ded.prod:divide (caddr dexpr1) (caddr dexpr2) ) ) )
             (if (for-every (prod1 in (cdr dexpr1) )
                            (prod2 in (cdr dexpr2) )
                     (|| (&& (= prod1 '(* 0) )
                             (= prod2 '(* 0) ) )
                         (= result (ded.prod:divide prod1 prod2) ) ) )
             (then
                 (de:normalize result) )
             (else
                 () ) ) ) )

      ( t
```



```lisp
         () ) ) )


;***===========================================================================
;***
;*** (DED.PROD:DIVIDE PROD1 PROD2)
;***
;*** Divides product PROD1 by product PROD2, returning the result product
;*** if there is no remainder, () if there is a remainder.
;***
;***===========================================================================

(defun ded.prod:divide ( (() c1 . vars1) (() c2 . vars2) )
    (if (|| (== c1 0)
            (&& (subset? vars2 vars1)
                (== 0 (\ c1 c2) ) ) )
    (then
        '(* ,(// c1 c2) ,.(set-diffq vars1 vars2) ) )
    (else
        () ) ) )
```

```lisp
;===========================================================================
;
; This module normalizes "diophantine expressions" (DEXPR's), "diophantine
; equations" (a diophantine equation is just a DEXPR assumed equal to
; 0), and "diophantine inequalities" (DEXPR > 0).  A DEXPR consists of
; constants (numbers), variables, +, *, - (unary and binary), &.  DEXPRs
; have two forms:  normalized and unnormalized.  & is the alternative
; operator; (& e1 e2) means either e1 or e2.  & is used in the derivations
; of expressions received from the flow analysis components; alternatives
; arise when multiple definitions reach a single use of a variable.
;
; An unnormalized DEXPR has the following syntax:
;
;     dexpr -> (& dexpr [dexpr dexpr ...] )
;              (+ dexpr dexpr)
;              (* dexpr dexpr)
;              (- dexpr dexpr)
;              (- dexpr)
;              constant              ;;; a Lisp number
;              variable              ;;; anything that isn't the above
;
; Notice that variables don't need to be symbols; they can be records,
; for example.
;
; A normalized DEXPR has the following syntax:
;
;     dexpr -> (& sum [sum sum ...] )
;     sum   -> (+ (* constant) [prod prod prod ...] )
;     prod  -> (* constant variable [variable variable ...] )
;
; The variables in a PROD are in alphabetic order if they are symbols,
; in STAT:NUMBER order if they are STATs, or otherwise in LEXORDER.  PRODs
; with the same variables are collapsed into one PROD.  PRODs are
; guarranteed to have non-zero constants.  The PRODs are ordered first
; by length, then by variable order.
;
; Negation is replaced by addition and multiplication by -1.  Examples:
;
;     5                     => (& (+ (* 5) ) )
;     (+ (* a b) (* b a) ) => (& (+ (* 0) (* 2 a b) ) )
;     (- (* a b) (* b a) ) => (& (+ (* 0) ) )
;     (+ 3 (& 4 5) )        => (& (+ (* 7) ) (+ (* 8) ) )
;
; A diophantine equation or inequality is represented by a DEXPR that
; is assumed equal to or greater than 0.  Such a DEXPR is normalized by
; reducing all its constant coefficients by by the GCD of all the
; constants, and for equations only, by making the first non-zero constant
; positive (by multiplying by -1 if necessary).
;
; (DE:NORMALIZE EXPR)
;     Returns the normalized DEXPR corresponding to EXPR, an unnormalized
;     DEXPR.
;
; (DE:NORMALIZE-EQUATION EXPR)
;     Returns the normalized diophantine equation equivalent to EXPR = 0.
;
; (DE:NORMALIZE-INEQUALITY EXPR)
;     Returns the normalized diophantine inequality equivalent to EXPR > 0.
;
;===========================================================================

(eval-when (compile)
    (build '(flow-analysis:stat) ) )
```



```lisp
(defun de:normalize ( expr )
    (? ( (numberp expr)
         (den.de:normalize-constant expr) )

       ( (listp expr)
         (caseq (car expr)
            (-  (if (= 2 (length expr) )
                    (de:normalize '(* -1 ,(cadr expr) ) )
                    (de:normalize '(+ ,(cadr expr) (* -1 ,(caddr expr) )))))
            (+  (den.de:normalize-+ expr) )
            (*  (den.de:normalize-* expr) )
            (&  (den.de:normalize-& expr) )
            (t  (error (list expr "Unkown expression in DE:NORMALIZE.") ))))

       ( t
         (den.de:normalize-var expr) ) ) )


;***========================================================================
;***
;*** (DEN.DE:NORMALIZE-CONSTANT EXPR)
;*** (DEN.DE:NORMALIZE-VAR      EXPR)
;*** (DEN.DE:NORMALIZE-+        EXPR)
;*** (DEN.DE:NORMALIZE-*        EXPR)
;*** (DEN.DE:NORMALIZE-&        EXPR)
;***
;*** These functions each normalize a subset of unnormalized DEXPRs.  For
;*** example, DEN.DE:NORMALIZE-+ normalizes and unnormalized DEXPR whose
;*** outermost operator is +.
;***
;***========================================================================

(defun den.de:normalize-constant ( expr )
    '(& (+ (* ,expr) ) ) )


(defun den.de:normalize-var ( expr )
    '(& (+ (* 0) (* 1 ,expr) ) ) )


(defun den.de:normalize-+ ( expr )
    (assert (<= 2 (length expr) ) )
    (if (== 2 (length expr) ) (then
        (de:normalize (cadr expr) ) )

    (else
        (let ( (x (de:normalize (cadr expr) ) )
               (y (de:normalize (if (== 3 (length expr) )
                                    (caddr expr)
                                    '(+ ,,(cddr expr) ) ) ) ) )
            (loop (for xi in (cdr x) )
                  (initial result () )
            (do
                (loop (for yi in (cdr y) ) (do
                    (push result (den.de:+-of-+* xi yi) ) ) ) )
            (result '(& ,,(noduples result) ) ) ) ) ) ) )


(defun den.de:normalize-* ( expr )
    (assert (<= 2 (length expr) ) )
    (if (== 2 (length expr) ) (then
```

```
        (de:normalize (cadr expr) ) )

    (else
        (let ( (x (de:normalize (cadr expr) ) )
               (y (de:normalize (if (== 3 (length expr) )
                                   (caddr expr)
                                   '(* ,,(cddr expr) ) ) ) ) )
            (loop (for xi in (cdr x) )
                  (initial result () )
            (do
                (loop (for yi in (cdr y) ) (do
                    (push result (den.de:*-of-+* xi yi) ) ) ) )
            (result '(& ,,(noduples result) ) ) ) ) ) ) )


(defun den.de:normalize-& ( expr )
    '(& ,,(noduples (for (x in (cdr expr) ) (splice
                        (cdr (de:normalize x) ) ) ) ) ) )



;***========================================================================
;***
;*** (DEN.DE:+-OF-+* X Y)
;***
;*** X and Y are normalized SUMs (see the grammar above).  They are
;*** combined into a single normalized SUM.  Example:
;***
;***     X      = (+ (* 3) (* 2 A) )
;***     Y      = (+ (* 2) (* 5 A) (* 7 B) )
;***     RESULT = (+ (* 5) (* 7 A) (* 7 B) )
;***
;***========================================================================

(defun den.de:+-of-+* ( x y )
    (den.de:+-of-list-* '(,,(cdr x) ,,(cdr y) ) ) )


;***========================================================================
;***
;*** (DEN.DE:+-OF-LIST-* *-LIST)
;***
;*** *-LIST is a list of normalized PRODs (see the grammar above).  The
;*** PRODs are added into a single SUM.  Example:
;***
;***      *-LIST = ( (* 3) (* 2 A) (* 2) (* 5 A) (* 7 B) )
;***      RESULT = (+ (* 5) (* 7 A) (* 7 B) )
;***
;***========================================================================

(defun den.de:+-of-list-* ( *-list )
    (let ( (terms () ) )

                ;*** First, add up all the terms.
                ;
        (loop (for x in *-list) (do
            (loop (for y in terms) (do
                (if (= (cddr x) (cddr y) ) (then
                    (:= (cadr y) (+ (cadr y) (cadr x) ) )
                    (return () ) ) ) )
            (result
                (push terms (copy x) ) ) ) ) )
```



```
                ;*** Remove terms with constant 0.
                ;
        (loop (for x in terms)
            (initial new-terms () )
        (do
            (if (|| (! (cddr x) )
                    (!= 0 (cadr x) ) )
            (then
                (push new-terms x) ) ) )
        (result
            (:= terms new-terms) ) )

                ;*** Now sort all the terms
                ;
        (:= terms (sort terms 'den.de:compare-*) )

                ;*** Add in a 0 constant term at the front if not there
                ;
        (if (!= 2 (length (car terms) ) ) (then
            (push terms (copy '(* 0) ) ) ) )

        '(+ ,,terms) ) )


;***========================================================================
;***
;*** (DEN.DE:COMPARE-* X Y)
;***
;*** X and Y are normalized PRODS (see the grammar above).  They are
;*** compared; true is returned iff X is "less" than Y.  PRODs with fewer
;*** variables come first; PRODS with the same number of variables are
;*** order by variable ordering of the variables.
;***
;***========================================================================

(defun den.de:compare-* ( x y )
    (if (!= (length x) (length y) ) (then
        (< (length x) (length y) ) )

    (else
        (loop (for xi in (cddr x) )
              (for yi in (cddr y) )
        (do
            (caseq (den.de:compare-variables xi yi)
                (-1 (return t) )
                (+1 (return () ) ) ) )
        (result () ) ) ) ) )


;***========================================================================
;***
;*** (DEN.DE:COMPARE-VARIABLES X Y)
;***
;*** X and Y are variables.  Returns -1 if X is less than Y, 0 if equal,
;*** 1, if greater.  The ordering used if X and Y are of different types
;*** is:  numbers, litatoms, STATs, anything else.  If they are of the
;*** same type, then they are compared numerically or lexically if atomic,
;*** using STAT:NUMBER if they are STATs, and using EQUAL/LEXORDER
;*** otherwise.
;***
;***========================================================================

(defun den.de:compare-variables ( x y )
```

```
    (?( (numberp x)
        (?( (numberp y)
            (?( (<  x y) -1)
              ( (=  x y)  0)
              ( t        +1) ) )
          ( t -1) ) )

      ( (litatom x)
        (?( (numberp y) +1)
          ( (litatom y)
            (?( (eqstr    x y)  0)
              ( (lexorder x y) -1)
              ( t              +1) ) )
          ( t -1) ) )

      ( (stat:is x)
        (?( (numberp y) +1)
          ( (litatom y) +1)
          ( (stat:is y)
            (?( (<  (stat:number x) (stat:number y) ) -1)
              ( (== (stat:number x) (stat:number y) )  0)
              ( t                                     +1) ) )
          ( t -1) ) )

      ( t
        (?( (=         x y)  0)
          ( (lexorder x y) -1)
          ( t              +1) ) ) ) )


;***=========================================================================
;***
;*** (DEN.DE:*-OF-+* X Y)
;***
;*** X and Y are normalized SUMs (see the grammar above).  They are
;*** multiplied together and normalized.  Example:
;***
;*** X      = (+ (* 2) (* 3 A) )
;*** Y      = (+ (* 3) (* 4 A) )
;*** RESULT = (+ (* 6) (* 17 A) (* 12 A A) )
;***
;***=========================================================================

(defun den.de:*-of-+* ( x y )
    (den.de:+-of-list-*
        (loop (for xi in (cdr x) )
              (initial result () )
        (do
            (loop (for yi in (cdr y) ) (do
                (push result (den.de:*-of-* xi yi) ) ) ) )
        (result result) ) ) )


;***=========================================================================
;***
;*** (DEN.DE:*-OF-* X Y)
;***
;*** X and Y are normalized PRODs (see the grammar above).  They are
;*** multiplied together and normalized.  Example:
;***
;*** X      = (* 3 A D)
;*** Y      = (* 2 C B)
;*** RESULT = (* 6 A B C D)
```



```
;***
;***=========================================================================

(defun den.de:*-of-* ( x y )
    `(* ,(* (cadr x) (cadr y) )
        ,.(sort (append (cddr x) (cddr y) )
                (f:l ( var1 var2 )
                    (== -1 (den.de:compare-variables var1 var2) ) ) ) ) )

;***=========================================================================
;***
;*** (DE:NORMALIZE-EQUATION   EXPR)
;*** (DE:NORMALIZE-INEQUALITY EXPR)
;***
;***=========================================================================

(defun de:normalize-equation ( expr )
    (de:normalize-equation-inequality expr t) )

(defun de:normalize-inequality ( expr )
    (de:normalize-equation-inequality expr () ) )

(defun de:normalize-equation-inequality ( expr equation? )
    (let ( (dexpr (de:normalize expr) ) )
        (loop (for sum in (cdr dexpr) ) (do
            (den.de:normalize-equation-sum! sum equation?) ) )
        dexpr) )


;***=========================================================================
;***
;*** (DEN.DE:NORMALIZE-EQUATION-SUM! SUM EQUATION?)
;***
;*** SUM is a normalized sum (see the grammar above) that is assumed to
;*** represent a diophantine equation or inequality (= 0 or > 0).  The
;*** constants of the PRODs are all reduced by their GCD.  If EQUATION?,
;*** then SUM is assumed to be an equation and the first non-zero constant
;*** is made positive (by multiplying by -1 if necessary).  This is all
;*** done destructively to SUM.
;***
;***=========================================================================

(defun den.de:normalize-equation-sum! ( sum equation? )
    (let ( (gcd      0)
           (negate? () ) )

            ;*** Find the GCD of all the product coefficients.  Remember
            ;*** if the first non-zero coefficient was positive or negative.
            ;
    (loop (for (() constant . ()) in (cdr sum) )
          (initial found-non-zero? () )
    (do
        (if (&& (! found-non-zero?)
                (!= 0 constant) )
        (then
            (:= found-non-zero? t)
            (:= negate? (< constant 0) ) ) )
        (:= gcd (gcd gcd constant) ) ) )

            ;*** Normalize all the constants if necessary.
            ;
    (if (&& equation? negate?) (then
        (:= gcd (- 0 gcd) ) ) )
```

```
(if (!= 1 gcd) (then
    (loop (for product in (cdr sum) ) (do
        (:= (cadr product) (// (cadr product) gcd) ) ) ) ) )

sum) )
```

```
;==============================================================================
;
; DE-SOLVE
;
; This module answers the question, "Could two diophantine expressions
; possibly be equal?"  Currently, only linear diophantine equations of
; 0, 1, or 2 variables are handled; any others are assumed to have possible
; solutions.
;
; The operators allowed in the expressions are +, - (unary and binary),
; *, and & (alternative).  See DE-NORMALIZE.LSP for a formal definition
; of the diophantine expressions allowed.
;
; (DE:POSSIBLY-EQUAL? DEXPR1 DEXPR2 STAT)
;     Returns true if the two diophantine expressions could possibly be
;     equal.  Returns false only if we know for sure that the two
;     expressions could not possibly be equal.  The "true" answer is
;     YES if we know for sure the two expressions are equal, MAYBE if
;     we only suspect they could be equal;  the "false" answer is NO.
;     STAT is the statement where we want to consider equality -- it is
;     used to find out which client-supplied assertions might be applied
;     in determining equality;  only those assertions are considered that
;     dominate STAT.
;
; (DE:ASSERT COMPARISON-OP DEXPR1 DEXPR2 STAT)
;     Asserts that DEXPR1 COMPARISON-OP DEXPR2; COMPARISION-OP is one
;     of the NADDR comparison operators.  STAT is the assertion STAT
;     containing the relational.  All assertions in the "database" are
;     examined first before solving the diophantine equation solver when
;     trying to decide if two expressions could be equal.
;
; (DE:RESIDUE DEXPR M STAT)
;     Returns the residue of DEXRP modulo M if known, () otherwise.  The
;     assertion database is examined for applicable =0-MOD assertions.
;
; (DE.INITIALIZE)
;     Initializes this module, by forgetting all previous assertions.
;
;==============================================================================

(eval-when (compile)
    (build '(flow-analysis:stat) ) )


(defvar *des.!=0-assertions*     () )
(defvar *des.>0-assertions*      () )
    ;
    ;*** These two lists represent the known equality assertions about
    ;*** the program.  Each list contain pairs of the form (DEXPR STAT).
    ;*** DEXPR is a normalized diophantine sum (not equation), and STAT
    ;*** is the flow graph assertion STAT where the particular assertion
    ;*** was made.  The first list contains expressions known just to
    ;*** be not equal to zero; the second list contains expressions known
    ;*** to be greater than 0.

(defvar *des.=0-mod-assertions* () )
    ;
    ;*** This list represents the known MOD assertions.  It is an association
    ;*** list keyed by modulus M whose elements are in the form:
    ;***
    ;***     (M (SUM1 STAT1) (SUM2 STAT2) ...)
    ;***
    ;*** where M is a modulus and (SUMi STATi) means normalized SUMi = 0
```



```
    ;*** mod M at point STATi in the flow graph.


(defun de.initialize ()
    (:= *des.!=0-assertions*     () )
    (:= *des.>0-assertions*      () )
    (:= *des.=0-mod-assertions* () )
    () )


(defun de:possibly-equal? ( dexpr1 dexpr2 stat )
    (assert (stat:is stat) )
    (des.de:possible-solutions? `(- ,dexpr1 ,dexpr2) stat) )


(defun de:assert ( comparison-op dexpr1 dexpr2 stat )
    (assert (stat:is stat) )

    (caseq comparison-op
        (ine     (des.de:assert-!=0      `(- ,dexpr1 ,dexpr2)   stat) )
        (ilt     (des.de:assert->0       `(- ,dexpr2 ,dexpr1)   stat) )
        (ile     (des.de:assert->0  `(+ 1 (- ,dexpr2 ,dexpr1) ) stat) )
        (ige     (des.de:assert->0  `(+ 1 (- ,dexpr1 ,dexpr2) ) stat) )
        (igt     (des.de:assert->0       `(- ,dexpr1 ,dexpr2)   stat) )
        (ie0mod  (des.de:assert-=0-mod   dexpr1 dexpr2          stat) )
        (t   (error (list comparison-op
                 "DE:ASSERT: Unsupported relational operator.") ) ) ) )


;***===========================================================================
;***
;*** (DES.DE:POSSIBLE-SOLUTIONS? DEXPR STAT)
;***
;*** DEXPR is a diophantine expression.  Returns NO if it can be proven
;*** that DEXPR is always non-zero.  Returns YES if it can be proven DEXPR
;*** is always zero.  Returns MAYBE otherwise.
;***
;*** Currently we handle the following forms:
;***
;***      c            = 0
;***      ax + c       = 0
;***      ax + by + c = 0
;***
;*** The database of assertions is searched if diophantine analsys doesn't
;*** yield a conclusive answer.
;***
;***===========================================================================

(defun des.de:possible-solutions? ( dexpr stat )
    (loop (for sum in (cdr (de:normalize-equation dexpr) ) )
          (initial any-yeses?  ()
                   any-maybes? ()
                   any-nos?    ()
                   sum-answer  () )

    (do
        (:= sum-answer
           (caseq (length (cdr sum) )
               (1
                   (if (= 0 (des.prod:constant (cadr sum) ) )
                       'yes
                       'no) )
```

```
                ( (2 3)
                    (let*( (c (des.prod:constant (cadr   sum) ) )
                           (a (des.prod:constant (caddr  sum) ) )
                           (b (des.prod:constant (cadddr sum) ) )
                           (g (gcd a b) ) )
                        (if (|| (< 3 (length (caddr  sum) ) )
                                (< 3 (length (cadddr sum) ) )
                                (= 0 c)
                                (= 0 (\ c g) ) )
                        (then
                            'maybe)
                        (else
                            'no) ) ) )

                (t  'maybe) ) )

        (if (&& (== 'maybe sum-answer)
                (des.sum:asserted-non-zero? sum stat) )
        (then
            (:= sum-answer 'no) ) )

        (caseq sum-answer
            (no    (:= any-nos?    t) )
            (maybe (:= any-maybes? t) )
            (yes   (:= any-yeses?  t) ) ) )

    (result
        (?( any-maybes?
            'maybe)
          ( (&& any-yeses? (! any-nos?) )
            'yes)
          ( (&& (! any-yeses?) any-nos?)
            'no)
          ( (&& any-yeses? any-nos?)
            'maybe)
          ( t
            (error "DES.DE:POSSIBLE-SOLUTIONS?:  Oh shit!") ) ) ) ) )


;***======================================================================
;***
;*** (DES.DE:ASSERT->0     DEXPR   STAT)
;*** (DES.DE:ASSERT-!=0    DEXPR   STAT)
;***
;*** These functions actually add the sums in a normalized equation to
;*** the databases of assertions.  STAT is the point in the flow graph
;*** of the assertion.
;***
;***======================================================================

(defun des.de:assert->0 ( dexpr stat )
    (loop (for sum in (cdr (de:normalize-inequality dexpr) ) ) (do
          (push *des.>0-assertions* `(,sum ,stat) ) ) )
    () )

(defun des.de:assert-!=0 ( dexpr stat )
    (loop (for sum in (cdr (de:normalize-equation dexpr) ) ) (do
          (push *des.!=0-assertions* `(,sum ,stat) ) ) )
    () )


;***======================================================================
```



```
;***
;*** (DES.SUM:ASSERTED-NON-ZERO? SUM STAT)
;***
;*** Returns true if it can be proved using the assertions that normalized
;*** sum SUM is non-zero.  STAT is the point in the flow graph where we
;*** are asking the question.
;***
;***======================================================================

(defun des.sum:asserted-non-zero? ( sum stat )
    (|| (des.sum:asserted-~!=0? sum stat)
        (des.sum:asserted->0?  sum stat)
        (des.sum:asserted->0?  (des.sum:negate sum)
                               stat) ) )


;***======================================================================
;***
;*** (DES.SUM:ASSERTED-!=0? SUM STAT)
;***
;*** Returns true if the normalized SUM (not a full DEXPR) has been
;*** asserted to be != 0.
;***
;***======================================================================

(defun des.sum:asserted-!=0? ( sum stat )
    (loop (for (asserted-sum asserted-stat) in *des.!=0-assertions*) (do
        (if (&& (bblock:dominates? (stat:bblock asserted-stat)
                                   (stat:bblock stat) )
                (des.de:= asserted-sum sum) )
        (then
            (return t) ) ) )
    (result () ) ) )


;***======================================================================
;***
;*** (DES.SUM:ASSERTED->0? SUM STAT)
;***
;*** Returns true if the normalized SUM (not a full DEXPR) can be proven
;*** to be > 0.   The method (described in the comments below) is just
;*** a quick crock and might have to be replaced if it is too slow or not
;*** general enough.  Known deficiencies:
;***
;***    Given 3X + 4 > 0, it can't prove 4X + 4 > 0.
;***
;***======================================================================

(defun des.sum:asserted->0? ( sum stat )
    (let*( (valid-assertions (des.stat:valid-assertions stat
                                                  *des.>0-assertions*) )
           (used             (makevector (length valid-assertions) ) ) )
        (des.sum:recursively-prove->0? sum valid-assertions used) ) )


;***======================================================================
;***
;*** (DES.SUM:RECURSIVELY-PROVE->0? SUM VALID-ASSERTIONS USED)
;***
;*** Tries to recursively prove that SUM is > 0, using the sums in
;*** VALID-ASSERTIONS (all asserted > 0).  USED is a boolean vector whose
;*** elements correspond to those of VALID-ASSERTIONS; ([] USED I) is
;*** true iff the Ith sum in VALID-ASSERTIONS has already been used in
```

```
;*** trying to prove some antecedent of SUM greater than 0.
;***
;***========================================================================

(defun des.sum:recursively-prove->0? ( sum valid-assertions used ) (prog ()

        ;*** If SUM is just a constant, then return true or false according
        ;*** to its value.
        ;
    (if (== 1 (length (cdr sum) ) ) (then
        (return (< 0 (des.sum:constant sum) ) ) ) ) )

        ;*** Find all asserted->0 sums A-SUM such that all the terms of
        ;*** SUM are contained in A-SUM; that is, such that A-SUM > 0
        ;*** can be expressed as:
        ;***
        ;***    SUM + A-SUM1 > 0
        ;***
        ;*** If A-SUM1 = 0, we are done.  Otherwise, for each such A-SUM,
        ;*** try to resursively prove that A-SUM1 <= 0.
        ;
    (return
    (loop (for  asserted-sum in valid-assertions)
          (incr i from 0)
          (when (! ([] used i) ) )
          (when (des.sum:contains-sum? asserted-sum sum) )
          (bind remaining-sum (des.sum:remaining-sum asserted-sum sum) )
    (do
        (if (des.de:= remaining-sum '(+ (* 0) ) )  (then
            (return t) ) )

        (:= ([] used i) t)
        (if (des.sum:recursively-prove->0?
                (des.sum:+1! (des.sum:negate remaining-sum) )
                valid-assertions
                used)
        (then
            (return t) ) )
        (:= ([] used i) () ) )

    (result () ) ) ) ) ) )


;***========================================================================
;***
;*** (DES.STAT:VALID-ASSERTIONS STAT SUM&STAT-LIST)
;***
;*** Returns the list of assertions (sums) in the SUM&STAT-LIST that are
;*** valid at point STAT in the flow graph.   An assertion is valid at
;*** a point if its source STAT dominates that point.
;***
;***========================================================================

(defun des.stat:valid-assertions ( stat sum&stat-list )
    (loop (for (asserted-sum asserted-stat) in sum&stat-list)
          (when (bblock:dominates? (stat:bblock asserted-stat)
                                   (stat:bblock stat) ) )
    (save
        asserted-sum) ) )


;***========================================================================
;***
```



```
;*** (DES.SUM:CONTAINS-SUM? BIG-SUM LITTLE-SUM)
;***
;*** Returns true iff all the products in LITTLE-SUM are contained in
;*** BIG-SUM (except for the first constant product).
;***
;***========================================================================

(defun des.sum:contains-sum? ( big-sum little-sum )
    (for-every (prod in (cddr little-sum) )
        (des.sum:contains-prod? big-sum prod) ) )

;***========================================================================
;***
;*** (DES.SUM:CONTAINS-PROD? BIG-SUM PROD)
;***
;*** Returns true iff product PROD is contained in the sum BIG-SUM (the first
;*** constant product is not considered).
;***
;***========================================================================

(defun des.sum:contains-prod? ( sum prod )
    (for-some (sum-prod in (cddr sum) )
        (des.de:= sum-prod prod) ) )


;***========================================================================
;***
;*** (DES.SUM:REMAINING-SUM BIG-SUM LITTLE-SUM)
;***
;*** Returns BIG-SUM minus all the products in LITTLE-SUM.  BIG-SUM is
;*** assumed to contain all the products of LITTLE-SUM (except for the
;*** first constant product, which could be different).
;***
;***========================================================================

(defun des.sum:remaining-sum ( big-sum little-sum )
    (let ( (big-constant    (des.sum:constant big-sum) )
           (little-constant (des.sum:constant little-sum) )
           (prods
                (for (prod in (cddr big-sum) )
                     (when (! (des.sum:contains-prod? little-sum prod) ) )
                (save prod) ) ) )

        '(+ (* ,(- big-constant little-constant) )
            ,,prods) ) )


;***========================================================================
;***
;*** (DES.SUM:NEGATE  SUM)
;***
;*** Returns the negation of SUM.
;***
;***========================================================================

(defun des.sum:negate ( sum )
    (let ( (new-sum (copy sum) ) )
        (loop (for prod in (cdr new-sum) ) (do
            (:= (cadr prod) (* -1 &&&) ) ) )
        new-sum) )


;***========================================================================
```

```
;***
;*** (DES.SUM:+1! SUM)
;***
;*** Destructively adds 1 to SUM.
;***
;***=========================================================================

(defun des.sum:+1! ( sum )
    (let ( (prod (cadr sum) ) )
         (:= (cadr prod) (+ 1 &&&) )
         sum) )


;***=========================================================================
;***
;*** (DES.PROD:CONSTANT PROD)
;***
;*** PROD is a normalized product of a normalized DEXPR (see
;*** DE-NORMALIZE.LSP), e.g. (* 3 A B).  The constant of the PROD is
;*** returned.  If PROD is (), 0 is returned.
;***
;***=========================================================================

(defun des.prod:constant ( prod )
    (if prod
        (cadr prod)
        0) )

;***=========================================================================
;***
;*** (DES.SUM:CONSTANT SUM)
;***
;*** SUM is a normalized sum of a normalized DEXPR.  The constant of the
;*** sum is returned, e.g. 3 is returned for (+ (* 3) (* 4 A) (* 2 B) ).
;***
;***=========================================================================

(defun des.sum:constant ( sum )
    (cadr (cadr sum) ) )


;***=========================================================================
;***
;*** (DES.DE:= DEXPR1 DEXPR2)
;***
;*** Returns true if if EXPR1 is the same expression as DEXPR2;  we don't
;*** use EQUAL because the "atoms" of the expression could be records, and
;*** we certainly don't want to invoke the bogus Maclisp class system.
;***
;***=========================================================================

(defun des.de:= ( dexpr1 dexpr2 )
    (? ( (&& (consp dexpr1)
             (consp dexpr2) )
         (&& (des.de:= (car dexpr1) (car dexpr2) )
             (des.de:= (cdr dexpr1) (cdr dexpr2) ) ) )

       ( (&& (numberp dexpr1)
             (numberp dexpr2) )
         (= dexpr1 dexpr2) )

       ( t
         (== dexpr1 dexpr2) ) ) )
```



```

;***=========================================================================
;***
;*** (DES.DE:ASSERT-=0-MOD DEXPR M STAT)
;***
;*** Adds the assertion DEXPR = 0 mod M at point STAT in the flowgraph
;*** to the database of assertions.  DEXPR is an unnormalized expression.
;***
;***=========================================================================

(defun des.de:assert-=0-mod ( dexpr m stat )
    (assert (inump m)
        "The modulus of an (ASSERT (=0-MOD ... must be an integer." )

    (let ( (m-list (assoc# m *des.=0-mod-assertions*) ) )
        (if (! m-list) (then
            (push *des.=0-mod-assertions*
                (:= m-list (list m) ) ) ) )

        (loop (for sum in (cdr (de:normalize dexpr) ) ) (do
              (push (cdr m-list) '(,sum ,stat) ) ) ) )
    () )


;***=========================================================================
;***
;*** (DE:RESIDUE DEXPR M STAT)
;***
;*** See top comments.
;***
;***=========================================================================

(defun de:residue ( dexpr m stat )
    (assert (stat:is stat) )
    (assert (inump m) )

        ;*** DEXPR is of the form (& SUM1 SUM2 ...).  Return the residue
        ;*** of SUM1 if it is the same as the residues of all the other
        ;*** SUMi; otherwise return ().
        ;
    (let*( (valid-assertions (des.stat:valid-assertions stat
                                (cdr (assoc# m *des.=0-mod-assertions*) ) ) )
           (sums             (cdr (de:normalize dexpr) ) )
           (residue          (des.sum:residue (car sums) m valid-assertions)))

        (if (for-every (sum in (cdr sums) )
               (== residue (des.sum:residue sum m valid-assertions) ) )
        (then
            residue)
        (else
            () ) ) ) )


;***=========================================================================
;***
;*** (DES.SUM:RESIDUE SUM M VALID-ASSERTIONS)
;***
;*** Returns the residue mod M of normalized SUM; returns () if the residue
;*** is not known.  VALID-ASSERTIONS is the list of normalized sums known
;*** to be = 0 mod M.
;***
;***=========================================================================
```

```
(defun des.sum:residue ( sum m valid-assertions )

    (||
            ;*** SUM is of the form (+ C PROD1 PROD2 ...).  First see
            ;*** if there is an assertion (+ AC PROD1 PROD2 ...)  = 0
            ;*** mod M.  If so, the result is the residue of C - AC mod M.
            ;
        (loop (for asserted-sum in valid-assertions)
              (when (des.de:= (cddr sum) (cddr asserted-sum) ) )
        (do
            (return (mod (- (des.sum:constant sum)
                            (des.sum:constant asserted-sum) )
                         m) ) )
        (result () ) )

            ;*** We didn't find the right assertion.  Instead, calculate
            ;*** the residue by finding adding the residues of each of
            ;*** of the products of the sum.
            ;
        (loop (initial residue 0)
        (for prod in (cdr sum) )
        (bind prod-residue (des.prod:residue prod m valid-assertions) )
        (do
            (if (! prod-residue) (then
                (return () ) ) ) )
        (next residue (+ residue prod-residue) )
        (result (mod residue m) ) ) ) )


;***=========================================================================
;***
;*** (DES.PROD:RESIDUE PROD M VALID-ASSERTIONS)
;***
;*** Returns the residue of normalized product PROD mod M if known, ()
;*** otherwise.   VALID-ASSERTIONS is the list of normalized sums known
;*** to be = 0 mod M.
;***
;***=========================================================================

(defun des.prod:residue ( prod m valid-assertions )
    (?
            ;*** PROD is a constant product of the form (* C).  Return
            ;*** C mod M.
            ;
      ( (== 2 (length prod) )
        (mod (des.prod:constant prod) m) )


            ;*** PROD is of the form (* C V1 V2 ...) and C is a multiple
            ;*** of M.   Return 0.
            ;
      ( (== 0 (mod (des.prod:constant prod) m) )
        0)

            ;*** Otherwise, go into the assertion database.
            ;
      ( t
        (||
                ;*** Look for an assertion of the form (+ C PROD).  If
                ;*** we find one, the result is M - C mod M.
                ;
            (loop (for asserted-sum in valid-assertions)
```



```
                  (when (== 3 (length asserted-sum) ) )
                  (when (des.de:= prod (caddr asserted-sum) ) )
            (do
                (return (mod (- m (des.sum:constant asserted-sum) ) m) ) )
            (result () ) )

                ;*** Assume PROD is of the form (* C V1 V2 ...). Look
                ;*** for an assertion of the form (+ AC (* 1 V1 V2 ...)).
                ;*** If we find one, the result is - C * AC mod M.
                ;
            (loop (for asserted-sum in valid-assertions)
                  (when (== 3 (length asserted-sum) ) )
                  (bind asserted-prod (caddr asserted-sum) )
                  (when (== 1 (des.prod:constant asserted-prod) ) )
                  (when (des.de:= (cddr prod) (cddr asserted-prod) ) )
            (do
                (return (mod (- 0 (* (des.prod:constant prod)
                                     (des.sum:constant  asserted-sum) ) )
                             m) ) )
            (result () ) ) ) ) ) ) )
```