• (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*)) )
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 (dexpri dexpr2)
  (assert (fc* (consp dexpri)
                (== '*. (car dexpri) ) )
  (assert (** (consp dexpr2)
               (== '* (car dexpr2) ) )
  (= dexpri (cadr dexpri))
  (= dexpr2 (cadr dexpr2))
  (= DEXPR1 0)
  (if* (== (length dexpri) (length dexpr2))
       (> (length dexpri) 2))
  (let ( (prod2 (if (= 2 (length dexpr2))
                   (cadr dexpr2)
                   (caddr dexpr2)) )
  (loop (for prodl in (cdr dexpri))
    (bind prod-result (ded.prod:divide prodl prod2))
    (initial result ()))
  (if (prod-result)
    (then
      (return 0)
    )
  )
  )
)

Divides product PRODI 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 (for-every (prodl in (cdr dexpri))
    (prod2 in (cdr dexpr2))
    (tt (= prodl '(* 0) )
     (= prod2 '(* 0) )
     (= result (ded.prod:divide prodl prod2) )
    )
    (then
      (de:normalize result)
    )
  )
  (else () )
)
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 unnormallzed. & 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 unnormallzed DEXPR has the following syntax:

\[ \text{dexpr} \rightarrow (\& \text{dexpr} \text{dexpr} ...) \]
\[ (+ \text{dexpr} \text{dexpr}) \]
\[ (* \text{dexpr} \text{dexpr}) \]
\[ (- \text{dexpr} \text{dexpr}) \]
\[ (- \text{dexpr}) \]
\[ \text{constant} \]
\[ \text{variable} \]

Notice that variables don't need to be symbols; they can be records, for example.

A normalized DEXPR has the following syntax:

\[ \text{dexpr} \rightarrow (\& \text{sum} \text{sum} ...) \]
\[ \text{sum} \rightarrow (+ (* \text{constant}) \text{prod} \text{prod} ...) \]
\[ \text{prod} \rightarrow (* \text{constant} \text{variable}) \]

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 guaranteed 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 \rightarrow (\& (+ (* 5)) \) \]
\[ (+ (* a b) (+ b a)) \rightarrow (\& (+ (* 0) (+ 2 a b))) \]
\[ (+ (* b a) (+ a b)) \rightarrow (\& (+ (* 0) (+ 0))) \]
\[ (+ (* 3 a) (+ 4 b)) \rightarrow (\& (+ (* 7) (+ (* 8)))) \]

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 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 unnormallzed 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))

(defun de:normalize ( expr )
  (? ( (numberp expr)
    (den.de:normallze-constant expr) )
  ( (llstp expr)
    (caseq (car expr)
      (- (If (= 2 (length expr) )
        (de:normalize ' (+ -1 , (cadr expr) )
          (de:normalize ' (+ , (cadr expr) (* -1 , (caddr expr) )))
        (de:normalize ' (+ , (cadr expr) (* -1 , (caddr expr) )))
      )
      (+ (den.de : normalize-+ expr) )
      (* (den.de:normalize-* expr) )
      (ft (den.de:normallze-ft expr) )
      (t (error (list expr "Unknown expression in DE:NORMALIZE." )))
    )
    (t (den.de:noraallze-var expr) ) ) ) )

(defun den.de:normallze-constant ( expr )
  '(+ (* expr) )
)

(defun den.de:normallze-var ( expr )
  '(* (* 0 (* 1 expr) ) )
)

(defun den.de : normalize-* ( expr )
  (assert (<= 2 (length expr) )
    (If (== 2 (length expr) ) (then
      (de:normallze-constant (cadr expr) )
    )
    (else
      (let ( (x (de:normallze-constant (cadr expr) )
        (y (de:normallze-constant (caddr expr) )
          (if (== 3 (length expr) )
            (caddr expr) )
        )
      )
        (loop (for xl in (cdr x) 
          (Initial result ()
            (do (for yl In (cdr y) 
              (push result (den.de:+-of-+* xl yl) ) )
            )
          )
        )
      (result '(ft ..(noduples result) ))
      )
    )
  )))

(defun den.denormalize-* ( 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) )
        (cadr expr) )
          (initial result ()
            (do (for yl In (cdr y) 
              (push result (den.de:-of-+* xi yl) ) )
            )
          )
        )
      )
    )
      (result '(ft ..(noduples result) ))
    )
  )

(defun de:normalize ( expr )
  (? ( (numberp expr)
    (den.de:normalize-constant expr) )
  (1istp expr)
    (caseq (car expr)
      (- (If (= 2 (length expr) )
        (de:normalize ' (+ -1 , (cadr expr) )
          (de:normalize ' (+ , (cadr expr) (* -1 , (caddr expr) )))
        (de:normalize ' (+ , (cadr expr) (* -1 , (caddr expr) )))
      )
      (+ (den.de:normalize-+ expr) )
      (* (den.de:normalize-* expr) )
      (ft (den.de:normallze-ft expr) )
      (t (error (list expr "Unknown expression in DE:NORMALIZE. ")))
    )
  )
)

PS:<C.S.BULLDOG.DIOPHANTINE>DE-NORMALIZE.LSP.18
(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-+* xl yl))))
         (result '(..(noduples result)))))))

(defun den.de:normallze-ft (expr)
  '(ft ,,(noduples (for (x in (cdr expr)) (splice (cdr (deformalize x))) ))))

Defining the function `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) (* 6 A) (* 7 B) )
RESULT = (+ (• 6) (* 7 A) (* 7 B) )

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

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

(defun den.de:+-of-list-* (e-list)
  (let ( (terms () ) )
    (*** Remove terms with constant 0.
    (loop (for x in terms)
      (initial new-terms())
      (do
       (loop (for yi in (cdr y)) (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)))
      ...

(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
          (caaeq (den.de:compare-variables xl yl)
          (-1 (return t))
          (:= (result ()))))))
  ...
**PS:** C.S.BULLDOG.DIOPHANTINE>DE-NORMALIZE.LSP.
(if (inject gcd) (then
  (loop (for product in (cdr sum) ) (do
    (:= (cadr product) (o (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 1 (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 where the particular assertion was made. The database of assertions is searched if diophantine analysis doesn't yield a conclusive answer.

(DE:RESIDUE DEXPR M STAT)
Returns the residue of DEXPR modulo M if known, () otherwise. The assertion database is examined for applicable =0-M0D 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* () )
(defvar *des.=0-mod-assertions* () )

;*** These two lists represent the known equality assertions about the program. Each list contains 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 (SUM1 STAT1) means normalized SUM1 = 0

(PS: <C.S.BULLDOG.DIOPHANTINE>DE-SOLVE.LSP.9
```lisp
(defun des.prod:constant (c sum) )
(a (des.prod:constant (cadr sum) ) )
(b (des.prod:constant (caddr sum) ) )
g (gcd a b )
(if (|| (< 3 (length (cadr sum) ) )
(< 3 (length (cadddr sum) ) )
(= 0 c)
(= 0 ( c g) ) )
(then
'maybe)
(else
'no))
(t 'maybe))
(if (ff (== 'maybe sun-answer)
(des.sun:asserted-non-zero? sum stat) )
then
(:- sun-answer 'no) ) )
caseq sum-answer
(no (:= any-nos? t))
(maybe (:= any-maybes? t))
(yes (:= any-yeses? t))
(result (it any-maybes?
'maybe)
(&& any-yeses? (! any-nos?) )
'yes)
(&& (1 any-yeses?) any-nos?)
(no)
(&& any-yeses? any-nos?)
'maybe)
(t (error "DES.DE:POSSIBLE-SOLUTIONS?: Oh shit!)) ) )
```

These functions actually add the sums in a normalized equation to the databases of assertions. STAT is the point in the flow graph where we are asking the question.

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

Returns true if the normalized SUM (not a full DEXPR) has been asserted to be != 0.

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

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.

```lisp
(defun des.sun:asserted->0? (sua stat)
(let*( (valid-assertions (des.Btat:valid-assertions stat
*des.>0-assertions*) )
(used (makevector (length valid-assertions) ) )
(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; ([I USED] 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 recursively prove that A-SUM1 <= 0.
    (return (loop (for asserted-sum in valid-assertions)
                   (incr 1 from 0)
                   (when (! (des.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))))
                         (:= (des.used 1) t)
                         (if (des.sum:recursively-prove->0? (des.sum:+ (des.sum:negate remaining-sum) valid-assertions)
                             (then (return t))))
                         (:= (des.used 1) 0))))

(defvar 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)
          (save asserted-sum))

(defvar 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))

(defvar des.sum:contains-prod? (sun 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? (sun prod)
  (for-soae (sum-prod in (cddr sum))
    (des.de:= sum-prod prod))

(defvar 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))

(defvar des.sum:negate (sum)
  *** Returns the negation of SUM.

(defun des.sum:negate (sum)
  (let ((new-sum (copy sum))
         (big-constant (des.sum:constant 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))
    new-sum))
(defun des:sum:+l! (sum)
  (let ((prod (cadr sum)))
    (= (cadr prod) (+ 1 &sum))
    sum))

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

(defun des:sun:constant (sun)
  (cadr (cadr sun))

(defun des:de:= (dexpri dexpr2)
  (? (consp dexpri) (consp dexpr2)
    (des:de:= (car dexpri) (car dexpr2)
      (des:de:= (cadr dexpri) (cadr dexpr2)))
    (let* ((valid-assertions (des:stat:valid-assertions sun)
                (suns (cdr (deformalize sun)))
                (residue (des:sun:residue (car sans) n valid-assertions)))
      (if (for-every (sun in (cdr suns))
                    (== residue (des:sun:residue sun n valid-assertions))
        then residue)
      else ()))))

(defun des:de:assert-0-mod (dexpr n stat)
  (assert (lnunp m) "The modulus of an (ASSERT (=0-MOD ... must be an integer.")
    (let ((a-list (assoc m *de8.=0-nod-assertions*)
      (push *de8.=0-nod-assertions*
        (:n-llst (list n)))
      (loop (for sum in (cdr (de:nornallze dexpr))
        (push (cdr m-list) "(.sun .stat)")
      )
    )
    ) )

(defun de:residue (dexpr m stat)
  (assert (stat:ls stat)
    (assert (lnunp m)
      (let* ((valid-assertions (des:stat:valid-assertions sun)
                (suns (cdr (deformalize dexpr)))
                (residue (des:sun:residue (car sans) n valid-assertions)))
        (if (for-every (sum in (cdr suns))
                      (== residue (des:sun:residue sum n valid-assertions))
          then residue)
        else ()))))

(defun des:sum:residue (sun m valid-assertions)
  (let* ((valid-assertions (des:stat:valid-assertions sun)
            (suns (cdr (deformalize sun)))
            (residue (des:sun:residue sum m valid-assertions)))
    (if (for-every (sum in (cdr suns))
                  (== residue (des:sun:residue sum m valid-assertions))
      then residue)
    else ()))

PS: C.S.BULLDOG.DIOPHANTINE>DE-SOLVE.LSP.9
(defun des:sum:residue (sum m valid-assertions)
  (loop (for asserted-sum in valid-assertions)
    (when (des.de:= (cddr sum) (cddr asserted-sum))
      (do
        (return (nod (- (des.sum:constant sum)
                       (des.sum:constant asserted-sum)
                       m)))
      (result () )))
  (loop (initial residue 0)
    (for prod in (cdr sum))
    (bind prod-residue (des.prod:residue prod n valid-assertions))
    (do
      (if (! prod-residue) (then
                       (return ()))
       (next residue (+ residue prod-residue))
      (result (nod residue m))))
  )
)

(des.prod:residue (prod m valid-assertions)
  (when (= 2 (length prod))
    (nod (des.prod:constant prod) m)
    (when (= 0 (mod (des.prod:constant prod) m))
      (0)
      (when (= 3 (length asserted-sum))
        (loop (for asserted-sum in valid-assertions)
          (when (des.de:= prod (cddr asserted-sum))
            (do
              (return (mod (- * (des.sum:constant asserted-sum) m)))
            (result () )))
          (result ()))
      (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 ()))
        (loop (for asserted-sum in valid-assertions)
          (when (des.de:= (cddr sum) (cddr asserted-sum))
            (do
              (return (nod (- (des.sum:constant sum) (des.sum:constant asserted-sum) m)))
            (result () )))))
    (return (mod (- prod-residue m)))
  (result ()))
)

******

(des.prod:residue prod m valid-assertions)

- Returns the residue of normalized product RED 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)
  (when (= 2 (length prod))
    (nod (des.prod:constant prod) m)
    (when (= 0 (mod (des.prod:constant prod) m))
      (0)
      (when (= 3 (length asserted-sum))
        (for asserted-sum in valid-assertions)
          (when (des.de:= prod (cddr asserted-sum))
            (do
              (return (mod (- * (des.sum:constant asserted-sum) m)))
            (result () )))
          (result ()))
    (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 ()))
        (loop (for asserted-sum in valid-assertions)
          (when (des.de:= (cddr sum) (cddr asserted-sum))
            (do
              (return (nod (- (des.sum:constant sum) (des.sum:constant asserted-sum) m)))
            (result () ))))))
  (return (mod (- prod-residue m)))
  (result ()))
)