(def-block (x (a (1 10))) ; input
(= r 1)
(while (<= k a))
(do
(= j (/ (+ k a) 2))
(if (= x (a j)) (then (= r j) (leave lupl j))))
(if (< x (a j)) (then (= m (- j 1)))
(else (= k (+ j 1))))
(result (= r 0)))

(def-block (x a1 a2 a3 a4 a5 a6 a7 a8 a9 a10) ; input
(= r -1)
(if (= x a5) (then (= r 5))
(else if (< x a5) (then (= x < a5)
(if (= x a2) (then (= r 2))
(else if (< x a2) (then (= x < a2)
(if (= x a1) (then (= r 1))
(else (= r 0))))
(else if (= x a4) (then (= r 4))
(else if (< x a4) (then (= a2 < x < a4)
(if (= x a3) (then (= r 3))
(else (= r 0))))
(else (= r 0)))))))))
:: a4 < x < a5

(else if (= x a7) (then (= r 7)) :: a5 < x < a10
(else if (< x a7) (then (if (= x a6) (then (= r 6))
(else (= r 0))))
(else if (= x a9) (then (= r 9))
(else if (< x a9) (then (if (= x a8) (then (= r a8))
(else (= r 0))))
(else if (= x a10) (then (= r 10))
(else (= r 0)))))))))))

:: (skex bin '(* 11 22 33 44 55 66 77 88 99 1010 ) )

(= bin ; w/o array
(def-block (x a1 a2 a3 a4 a5 a6 a7 a8 a9 a10) ; input
(= r -1)
(if (= x a5) (then (= r 5))
(else if (< x a5) (then (= x < a5)
(if (= x a2) (then (= r 2))
(else if (< x a2) (then (= x < a2)
(if (= x a1) (then (= r 1))
(else (= r 0))))
(else if (= x a4) (then (= r 4))
(else if (< x a4) (then (= a2 < x < a4)
(if (= x a3) (then (= r 3))
(else (= r 0))))
(else (= r 0)))))))))
:: a4 < x < a5

(else if (= x a7) (then (= r 7)) :: a5 < x < a10
(else if (< x a7) (then (if (= x a6) (then (= r 6))
(else (= r 0))))
(else if (= x a9) (then (= r 9))
(else if (< x a9) (then (if (= x a8) (then (= r a8))
(else (= r 0))))
(else if (= x a10) (then (= r 10))
(else (= r 0)))))))))))
;;; Build the latest version of SKEP

;;;

(reprop 'loop 'build-information) ;; Get rid of Yale Loop info.
(:= *build-module-list* () )

(load 'utilities:build) ;; This must go first.
(load 'interpreter:build)
(load 'traces:build)
(load 'ideal-code-generator:build)
(load 'diophantine:build)
(load 'flow-analysis:build)

(:= *exp.build-module-list* '(
  experiments:skex-opt1
  experiments:skex
 ))

(:= *build-module-list* (append *build-module-list* *exp.build-module-list*))
(build)
Convolution, with k weights and vectors of length n.

```lisp
(= convoi
  (def-block
    (N :real size N (max 128))
    (K :real size K (max 32))
    (x (1 128))
    (y (1 128))
    (v (1 32))
    (w (1 32))
    (y (1 128))
    (loop
      (incr i from 1 to n)
      (unroll 4)
      (do
        (loop
          (incr j from 1 to k)
          (unroll 8 fold-step-vars no-back-jump no-exit-tests)
          (do
            (:= (y i) (+ (y i) (+ (w j) (x (- (+ i j) 1)))))
          
          
          (:= convo-args
            "'
            64
            8
            (1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
             1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
             1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
             1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
             1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
             1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
             (0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
              (1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
               1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
               1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
               1.0 0.0 8.0 4.0 1.0 0.0 8.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0)
            )
          )
        )
      )
    )
  )
)
```

```lisp
1
PS:<C.S.BULLDOG.EXPERIMENTS>CONVO.LSP.1
```
(def-block nil (P) 

(declare P (O 5))

(:= a (+ b c))
(:= d (+ e a))
(:= e (+ f d))
(:= f (+ b e))
(:= x (+ y f))
(:= y (+ x x))
(:= u (+ u u))
(:= v (+ v v))
(:= w (+ w w))

(declare P (P 1))
(:= j1 (+ j1))
(:= j2 (+ j1 j1))
(:= j3 (+ j2 j1))
(:= j4 (+ j3 j1))
(:= j5 (+ j4 j1))

))
(defun dotprod.unroll (n)
  (subst n 'unroll* dotprod.template*))

(:= dotprod1 (dotprod.unroll 1))
(:= dotprod2 (dotprod.unroll 2))
(:= dotprod4 (dotprod.unroll 4))
(:= dotprod5 (dotprod.unroll 5))
See FFT.FOR and FFT1.FOR for some meaningful FORTRAN source.

```
(fft.unroll 8) FFT-ARGS
(fft.unroll 16) FFT-ARGS
```

Correct answer for FFT-ARGS:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.04308E-7</td>
</tr>
<tr>
<td>2</td>
<td>2.81724E-8</td>
</tr>
<tr>
<td>3</td>
<td>15.99991E9</td>
</tr>
<tr>
<td>4</td>
<td>4.09013E-6</td>
</tr>
<tr>
<td>5</td>
<td>2.66227E-6</td>
</tr>
<tr>
<td>6</td>
<td>1.5662E-6</td>
</tr>
<tr>
<td>7</td>
<td>1.08865E-8</td>
</tr>
<tr>
<td>8</td>
<td>1.08864E-8</td>
</tr>
<tr>
<td>9</td>
<td>0.52688E-7</td>
</tr>
</tbody>
</table>

```
**** FFT adapted from "Introduction to Discrete Systems" by Kenneth Steiglitz
See FFT.FOR and FFT1.FOR for some meaningful FORTRAN source.
*****
```

```lisp
(def-block ( (s (1 256))
  (r (1 256))
)

(defun fft.unroll (n)
  (sub8t n 'fft.template))

(:= fft1 (fft.unroll 1))
(:= fft2 (fft.unroll 2))
(:= fft4 (fft.unroll 4))
```

```
1
PS:<C.S.BULLDOG.EXPERIMENTS>FFT.LSP.11
```
defun fft-gen-vals ( n )
    (loop (initial result () )
        (incr i from 1 to n)
        (do
            (push result (sin (* (- 1 i) (/ 3.14159 8.0) ) ) )
            (result
                (dreverse result) ) ) )

3
PS:<C.S.BULLDOG.EXPERIMENTS>FFT.LSP.11
Using the Maclaurin Series for the natural log of \( x \), we have two parameters, \( x \) (where \( 0 < x < 2 \)), and \( k \), the number of terms summed.

```
(:= ln-args '(1.8 80))
(:= ln-templates ')
(def-block (x k) (log)
  (:= y (-$ 1.0 x))
  (loop (unroll »unroll)
    (incr n from 1 to k)
    (step n-real from 0.0
      using (+$ n-real 1.0))
    (step numerator from -1.0
      using (*$ numerator y))
    (step log from 0.0
      using (+$ log (//$ numerator n-real)))
  )
)

(defun ln-unroll (n)
  (subat n '»unroll »ln-templates))

(:= ln1 (ln-unroll 1))
(:= ln2 (ln-unroll 2))
(:= ln4 (ln-unroll 4))
(:= ln20 (ln-unroll 20))
```
Matrix Multiply

(SKEX HAMTUL1 MATMUL-ARGS)
(SKEX HAMTUL2 MATMUL-ARGS)
(SKEX HAMTUL4 MATMUL-ARGS)

(= matmul-args
  '( (1.0 0.0 3.0 4.0
     2.0 2.0 0.0 0.0
     0.0 0.0 0.0 1.0
     0.0 0.0 0.0 0.0
   )
     (4.0 4.0 4.0 4.0
     3.0 3.0 3.0 3.0
     2.0 2.0 2.0 2.0
     1.0 1.0 1.0 1.0))

(= *matmul.template*
  (def-block ((a (1 4) (1 4))
              (b (1 4) (1 4))
              (p (1 4) (1 4)))
    (loop (Incr u from 1 to 4)
      (do
        (loop (Incr v from 1 to 4)
          (unroll *unroll*)
          (do
            (= SUB 0.0)
            (loop (Incr 1 from 1 to 4)
              (unroll 4 no-back-jump)
              (do
                (= sum (+$ sum (*$ (a u 1) (b 1 v))))
                (= (p u v) sum)))
          )
        )
      )
    )
  )

(defun matmul.unroll ( n )
  (subst n "unroll" "matmul.template")

(= matmul1 (matmul.unroll 1))
(= matmul2 (matmul.unroll 2))
(= matmul4 (matmul.unroll 4))

1

PS:<C.S.BULLDOG.EXPERIMENTS>MATMUL.LSP.3
Silly Tiny-Lisp test program to find the max of 16 numbers using a tree of comparisons. The 16 is VERY hardwired into this. This was chosen to get a lot of mileage out of the bookkeeper and make it likely that paths other than the main will be exercised... This is really pathological and ought to stand a good chance of breaking the compactor.

(SKEK MAX MAX-ARGS)

(:= aax-args '(( 18 15 14 13 12 11 10 9 8 7 6 6 4 3 2 1) )
(:= aax-args1 '((10 11 12 22 22 33 33 44 44)))
(:= aax-args2 '((1 1 1 1 2 2 2 2 33 33 44 44 44 44)))
(:= aax-args3 '((1 1 1 1 2 2 2 2 33 33 44 44 44 44)))

(:= aax (def-block ( (a 16) )
    (max)
    (:= max0 (if (> (a 0) (a 1))
        (then (a 0))
        (else (a 1)) ) )
    (:= max1 (if (> (a 2) (a 3))
        (then (a 2))
        (else (a 3)) ) )
    (:= max2 (if (> (a 4) (a 6))
        (then (a 4))
        (else (a 6)) ) )
    (:= max3 (if (> (a 6) (a 7))
        (then (a 6))
        (else (a 7)) ) )
    (:= max4 (if (> (a 8) (a 9))
        (then (a 8))
        (else (a 9)) ) )
    (:= max5 (if (> (a 10) (a 11))
        (then (a 10))
        (else (a 11)) ) )
    (:= max6 (if (> (a 12) (a 13))
        (then (a 12))
        (else (a 13)) ) )
    (:= max7 (if (> (a 14) (a 15))
        (then (a 14))
        (else (a 15)) ) )
    (:= max8 (if (> max0 max1)
        (then max0)
        (else max1)) )
    (:= max9 (if (> max2 max3)
        (then max2)
        (else max3)) )
    (:= max10 (if (> max4 max5)
        (then max4)
        (else max5)) )
)}
::: Uses sieve to find primes from 1-100. Should have very little compaction.

---

PRIME

::: Uses a sieve to calculate the primes between 1 and 100.

(SKEX PRIME1)
(SKEX PRIME2)
(SKEX PRIME3)
(SKEX PRIME4)
(SKEX (PRIME.UNROLL N))

---

(:= *prime.template* ')

(defun prime.unroll (n)
  (substitute n 'unroll 'prime.template))

(:= prime1 (prime.unroll 1))
(:= prime2 (prime.unroll 2))
(:= prime4 (prime.unroll 4))
Top Level Driver for SKeduling EXperlnents.

(defvar «skex.code* () ) ;*** The most recent tiny-lisp source code.
(defvar «skex.actual-paraas* 0 ) ;*«« The most recent actual parameters.
(defvar «skex.seq-naddr-unoptlalzed* () )

;««« The most recent sequential naddr
;*** from «SKEX.CODE*.
(defvar *8kex.seq-naddr-optlnlzed* 0 )

;*** Optimized version of the above.
(defvar *skex.par-naddr* () ) ;*** The most recent parallel naddr.

;*** List of functions to be timed.
(defvar »int.operation-count* ) ;*« Froa INTERPRETER.
(defvar »Int.Instruction-count«) ;*** Froa INTERPRETER.
(defvar *nunber-of-banks* ) ;*«* Nunber of memory banks.

(declare
(lexpr tiae-functions)
(lexpr options.print)
(lexpr coapile-tiny-lisp)
(lexpr print-functlons-tlaes)
(lexpr Interpret)
)

*** Options. Do (OPTIONS.HELP) for a description of each option and its
*** current value.

(declare (special ;*** Froa SKEX-OPTIONS
«skex.tiae-functions?«
«skex.elialnate-conaon-subexpressions?»
«skex.nove-loop-invariants?«
«skex.dlsaabiguator-tool?*
«skex.coapact?«
) )

(options.reset)

(= «skex.timed-functions* '(
 compile-tiny-lisp
 Interpret
 fg.analyzeeoptinize
 (fg.collect-names
 fg.set-reaching-defs
)
)

PS: <C.S.BULLDOG.EXPERIMENTS>SKEK.LSP.17
(defun skex.initialize ()
  (tz.initialize)
  (fg.initialize)
  (de.initialize)
  (:= «skex.code» (defun))
  (:= «skex.actual-paraas» (defun))
  (:= «skex.seq-naddr-unoptinlzed*» (defun))
  (:= «skex.seq-naddr-optlalzed*» (defun))
  (:= «skex.par-naddr» (defun))
  (let ( (old (gcgag t) )
         (gc)
         (gcgag old) ) ) )

;;; Run the sequential and compacted code from the last experiment but
;;; with new actual parameters.
;;; Run a tiny lisp program.

(defun skex.again ( actual-paraas )
  (:= «skex.actual-paraas* actual-paraas) freshmen t
  (skex.run-programs) )

;;; Execute the sequential and compacted code on the actual parameters
;;; (all saved away in the global variables), and print some pretty
;;; statistics and the results of each execution.

(defun skex.run-programs ()
  (let ( (sequential-instructions 0)
          (parallel-instructions 0)
          (parallel-operations 0)
          (foraal-paraas 0)
          (output-vars () ) )
    (asg 0 t
        (:= naddr (coapple-tlny-lisp code) )
        (lf (== 'def-block (caar naddr) ) (then
            (desetq (() foraal-paraas output-vars) (car naddr) ) ) )
        (Interpret naddr actual-paraas)
        (:= sequential-Instructions »int.operation-count»)
        (asg 0 »int.instruction-count* " instructions." t)
        (BBg "Uncoapacted prograa results:" t)
        (lnt.print-prograa-variables output-vars)
        (msg t "Uncoapacted program results:" t)
        (int.print-program-variables output-vars)
        (seg t "Compacted program results:" t)
        (int.print-program-variables output-vars)
        (seg t "Compacted instructions = " sequential-instructions t)
SKEX OPTIONS

This module contains the definitions of top level options dealing with
the whole compiler (SKEX).

(eval-when (compile)
  (build '(utilities:options)) )

(def-option *skex.time-functions?* () experiments: "
If T then all the functions listed in *SKEX.TIME-FUNCTIONS* are timed and
statistics printed out at the end of each SKEX run.
")

(def-option *skex.compact?* t experiments: "
If T then the NADDR program is compiled using trace-scheduling and the
currently loaded codegenerator; the optimized sequential NADDR is compared
with the compacted parallel NADDR. If () compacting is not invoked, and
the TinyLisp generated NADDR is compared with the optimized sequential
NADDR.
")

(def-option *number-of-banks* 8 experiments: "
This variable contains the number of memory banks to compile for.
")
LU DECOMPOSITION SOLVER. From Forsythe & Moler.

Given $A$ & $b$, solves $Ax=b$ by decomposing $A$ into $L$ & $U$, then solving the
easy resultant triangular systems $Ly=b$ and $Ux=y$. $L$ is lower triangular.

$U$ upper.

([SKEX SOLVE1 SOLVE-ARGS]
[SOLVE1 SOLVE-ARGS]
[SOLVE4 SOLVE-ARGS]
[SOLVE (SOLVE.UNROLL 16) SOLVE-ARGS]

(:= norarow 0.0)
(:= (pa 1) 1)
•*• the row, ps(i) to 1
*** decoapostlon natrlcies combined.

(loop (Incr j from 1 to 16)
  (x (1 16) )
)

(unsroll unroll*)

(cos (format t "%Singularity in a row." ) )

:*** Gaussian elimination with partial pivoting:

(loop incr i from 1 to (- 16 1) do
  (:= biggest 0.0)
  (:= pivot 0)
  (loop incr i from k to 16)
    (unroll *unroll*)
    (do
      (:= size (*$ (aba$ (lu (ps 1) k) ) (scales (ps 1) ) )
      (if (> size biggest) then
        (:= biggest size)
        (:= pivotx k) ) )
      (if (= 0.0 biggest) then
        (esc (format t "%Singular matrix.") ) )
      (if (= pivotx k) then
        (assert (= pivotx k) )
        (:= (ps 1) k)
        (:= (ps pivotx) )
        (:= (ps pivotx) )
      (:= pivot (lu (ps k) k) )
      (loop incr i from (+ k 1) to 16)
        (do
          (:= (scales l) (//$ 1.0 norarow) )
          (:= norarow (nax$ norarow (abs$ (lu 1 J) ) ) )
        (:= dot (+t dot (•* (lu (ps 1) 1) (x j) ) ) )
      (assert (/= mult (lu (ps k) j) )
      (:= (lu (ps 1) j)
        (- (lu (ps 1) j)
          (*$ (lu (ps k) j) ) ) )
    )
  )
)(:= (ls 1 1) (a 1 1) )
(:= norarow (max$ norarow (abs$ (lu 1 1) ) ) )
(if (= norarow 0.0) then
  (= (scales i) (/$ 1.0 norarow ) )
(else
  (esc (format t "%Singularity in a row." ) )
) ) )

:*** SOLVE: Using the LU found in DECOMPOSE, solves the linear
*** equation $Ax=b$.

(loop incr i from 1 to 16) do
  (:= dot 0.0)
  (loop incr j from 1 to (- 1 1) )
    (unroll *unroll*)
    (do
      (:= dot (+$ dot (*$ (lu (ps 1) j) (x j) ) )
      (:= dot 0.0)
      (loop incr j from (+ 1 1) to 16)
      )

PS:<C.S.BULLDOG.EXPERIMENTS>SOLVE.LSP.2
(unroll *unroll*)

(do
    (:= dot (+$ dot (+$ (lu (ps i) j) (x j)) ) ) )
    (:= (x 1) (/$ (-$ (x 1) dot) (lu (ps i) i)) ) )
)

;*** For a check, we find Ax and see if it's equal to b...
(loop (incr i from 1 to 16)
    (do
        (:= tot 0.0)
        (loop (incr j from 1 to 16)
            (unroll *unroll*)
            (do
                (:= tot (+$ tot (** (a 1 j) (x j)) ) )
                (:= (ax i) tot) )
        )
    )
)

(defun solve.unroll ( n )
    (subset n *unroll* *solve.template* )
)

(:= solve1 (solve.unroll 1) )
(:= solve2 (solve.unroll 2) )
(:= solve4 (solve.unroll 4) )
(skex.run-tiny-lisp ins '(1 10 (6 3 6 2 1 10 9 8 7) ))

(:= ins
       '(def-block (h n (1 (1 10) )
                 (loop (incr j from 2 to n) (do
                      (:= key (1 j))
                      (:= k
                         (loop lup (step 1 from (- j h) using (- 1 h)
                                      while (> 1 0))
                                (unroll 2)
                                (do
                                   (if (< (1 i) key) (then
                                        (:= (1 (+ h 1) ) (1 i) )
                                   (else
                                        (leave lup i) )))
                      (:= (1 (+ k h)) key) ) ) )

        (:= ins
       '(def-block (h n (1 (1 10) )
                 (loop (incr j from 2 to n) (do
                      (:= key (1 j))
                      (loop lup (step 1 from (- j h) using (- 1 h)
                                      while (if (> 1 0)
                                          (then (< (1 i) key )
                                          (else 0) )
                     (result-live i)
                     (unroll 2)
                     (do
                        (:= (1 (+ h 1) ) (1 i) )
                        (:= (1 (+ h) h) key) ) ) )

1
PS:<C.S.BULLDOG.EXPERIMENTS>SORT.LSP.1
Square Root

Using Newton's method.

(SKEX SQRT1 '(*15))
(SKEX SQRT2 '(*15))
(SKEX SQRT4 '(*16))
(SKEX SQRT10 '(*16))
(SKEX (SQRT.UNROLL n) '(*15))

(= *sqrt.template* '(x y z))
(def-block ( x ) ( y )
  (:= y-init (if (<= x 4.0) (then x) (else (/ x 2.0)))
  (loop 1
    (unroll *unroll*)
    (step y-old from x
      using y-init)
    (step y from y-init
      using (- y (/ (- (* y y) x) (* y y y) ))
      using (- y-old (/ (- (* y y) x) (* y y y) )))
    (while (> (- y-old y) 0.0000000001) )
  )
)
(defun sqrt.unroll ( n )
  (subst n '*unroll* *sqrt.template*)
)(:= sqrt1 (sqrt.unroll 1))
(:= sqrt2 (sqrt.unroll 2))
(:= sqrt4 (sqrt.unroll 4))
(:= sqrt10 (sqrt.unroll 10))
This is designed to show off the variable folder.

TRANPOSE1 doesn't do any unrolling at all

(comment ;; Stuff me please.
skex.set-option 'eliminate-common-subexpressions)

(skek transpose1 '(
  1.0 0.0 3.0 4.0  1.0 0.0 4.0 4.0
  2.0 2.0 0.0 0.0  2.0 2.0 8.0 8.0
  0.0 0.0 3.0 1.0  0.0 0.0 2.0 2.0
  0.0 0.0 0.0 0.0  0.0 0.0 1.0 1.0
  4.0 4.0 4.0 4.0  4.0 4.0 8.0 8.0
  3.0 3.0 3.0 3.0  3.0 3.0 8.0 8.0
  2.0 2.0 2.0 2.0  2.0 2.0 3.0 1.0
  1.0 1.0 1.0 1.0  1.0 1.0 1.0 1.0
))

endcomment)

(:= transpose1
  (def-block ((a (1 8) (1 8)))
    (at (1 8) (1 8))
    (loop (incr u from 1 to 8)
      (do
        (loop (incr v from 1 to 8)
          (do
            (= (at v u) (a u v))))))))

**************

**

** TRANPOSE2 does the unroll keywords to the hilt.

**

(comment ;; Stuff me please.
(skek.set-option 'eliminate-common-subexpressions)

(skek transpose2 '(
  1.0 0.0 3.0 4.0  1.0 0.0 4.0 4.0
  2.0 2.0 0.0 0.0  2.0 2.0 8.0 8.0
  0.0 0.0 3.0 1.0  0.0 0.0 2.0 2.0
  0.0 0.0 0.0 0.0  0.0 0.0 1.0 1.0
  4.0 4.0 4.0 4.0  4.0 4.0 8.0 8.0
  3.0 3.0 3.0 3.0  3.0 3.0 8.0 8.0
  2.0 2.0 2.0 2.0  2.0 2.0 3.0 1.0
  1.0 1.0 1.0 1.0  1.0 1.0 1.0 1.0
))

endcomment)

(:= transpose2
  (def-block ((a (1 8) (1 8)))
    (at (1 8) (1 8))
    (loop (incr u from 1 to 8)
      (do
        (loop (incr v from 1 to 8)
          (do
            (= (at v u) (a u v))))))))

**************

**

** TRANPOSE3 does unrolling without my neat new keywords. No sense in unrolling the outer loop, traces won't go beyond.

**

(comment ;; Stuff me please.
(skek.set-option 'eliminate-common-subexpressions)

(skek transpose3 '(
  1.0 0.0 3.0 4.0  1.0 0.0 4.0 4.0
  2.0 2.0 0.0 0.0  2.0 2.0 8.0 8.0
  0.0 0.0 3.0 1.0  0.0 0.0 2.0 2.0
  0.0 0.0 0.0 0.0  0.0 0.0 1.0 1.0
  4.0 4.0 4.0 4.0  4.0 4.0 8.0 8.0
  3.0 3.0 3.0 3.0  3.0 3.0 8.0 8.0
  2.0 2.0 2.0 2.0  2.0 2.0 3.0 1.0
  1.0 1.0 1.0 1.0  1.0 1.0 1.0 1.0
))

endcomment)

(:= transpose3
  (def-block ((a (1 8) (1 8)))
    (at (1 8) (1 8))
    (loop (incr u from 1 to 8)
      (do
        (loop (incr v from 1 to 8)
          (enroll 8)
          (do
            (= (at v u) (a u v))))))))

**************

**

** TRANPOSE4 is a silly attempt at doing a 12x12 transpose. What a fool.

**

(comment ;; Stuff me please.
(skek.set-option 'eliminate-common-subexpressions)

(skek transpose4 '(
  1.0 0.0 3.0 4.0  1.0 0.0 4.0 4.0  1.0 0.0 4.0 4.0
  2.0 2.0 0.0 0.0  2.0 2.0 8.0 8.0  2.0 2.0 8.0 8.0
  0.0 0.0 3.0 1.0  0.0 0.0 2.0 2.0  2.0 2.0 8.0 8.0
  0.0 0.0 0.0 0.0  0.0 0.0 1.0 1.0  2.0 2.0 8.0 8.0
  4.0 4.0 4.0 4.0  4.0 4.0 8.0 8.0  4.0 4.0 8.0 8.0
  3.0 3.0 3.0 3.0  3.0 3.0 8.0 8.0  3.0 3.0 8.0 8.0
  2.0 2.0 2.0 2.0  2.0 2.0 3.0 1.0  2.0 2.0 3.0 1.0
  1.0 1.0 1.0 1.0  1.0 1.0 1.0 1.0  1.0 1.0 1.0 1.0
))

endcomment)

(:= transpose4
  (def-block ((a (1 8) (1 8)))
    (at (1 8) (1 8))
    (loop (incr u from 1 to 8)
      (do
        (loop (incr v from 1 to 8)
          (enroll 8)
          (do
            (= (at v u) (a u v))))))))

1

PS:<C.S.BULLDOG.EXPERIMENTS>TRANSPOSE.LSP.1
Transitive Closure

The result should be all 1's.

(SKEX TRCL TRCL-ARGS)

(:= trcl-args '((0 1 0 0 0 1 1 0))

(:= trcl •
(def-block { (a (1 3) (1 3))
    { (a (1 3) (1 3))
      (loop (incr k from 1 to 3) (do
        (loop (incr 1 from 1 to 3)
        (do
          (if (= (a 1 k) 1) (then
            (loop (incr J from 1 to 3)
            (unroll S no-back-Jump)
            (do
              (if (= (a J k) 1) (then
                (:= (a 1 J) 1) ))))))))))
    )}
  )
)

1
PS:<C.S.BULLDOG.EXPERIMENTS>TRCL.LSP.2