This module implements the bookkeeping part of trace scheduling -- that part which, given a trace and a schedule for the trace, unsplices the trace from the MI flow graph and splices in the schedule, generating compensation code and keeping track of live variables and USE/DEFs.

**TR.INITIALIZE-BOOKKEEP**
Initializes this module by clearing out global state.

**TR.BOOKKEEP**
Performs bookkeeping on the current trace and its schedule. The bookkeeping is broken up into several main passes.

Each main pass is described in detail at its source; here is a summary of the passes:

- **BK.RECORD-THE-TRACE**
Records the trace on a vector for easy access later on.

- **BK.BUFFER-THE-TRACE**
Isolates the trace from the rest of the MI flow graph by splicing in dummies on each edge entering and exiting the edge. These dummies act as sentinels and make the implementation much easier.

- **BK.COALESCE**
Coalesces the machine operations in each cycle of the schedule into a compacted MI; the compacted MIs are stored on a vector for easy access.

- **BK.FIX-SUCCESSORS**
Sets the successors of each coalesced, compacted MI and generates split compensation code and DEFs at each split.

- **BK.FIX-PREDECESSORS**
Sets the predecessors of each coalesced, compacted MI and generates rejoin compensation code and USEs at each join.

- **BK.PROPAGATE-LIVE-VARIABLES**
Performs incremental live analysis on all the rejoin and split compensation code and USE/DEFs generated during bookkeeping.

- **BK.MISCELLANEOUS-CLEAN-UP**
Miscellaneous clean ups not deserving of a separate pass.

```lisp
(defun tr.initialize-bookkeep ()
  (setf *tr.rejoin-count* 0)
  (setf *tr.split-count* 0)
  (setf *tr.partial-rejoin-count* 0)
  (setf *tr.partial-split-count* 0)
  (setf *bk.use-def-count* 0)
  (setf *bk.dunny-nls* ())
  (setf *bk.rejoin-dunnels* ())
  (setf *bk.split-dunnels* ())
  (setf *bk.new-permanent-als* ())
  (setf *bk.use-def-as* ())
  (setf *bk.cycle-as* ())
  (setf *bk.trace-pos:mi* (makevector (+ 1 *tr.trace-size*))
  (loop (for trace-als in *tr.trace-als*
    (incr trace-pos fron 1)
    (Initial prev-trace-Bl () )
    (do
      (:= (makevector (+ 1 *tr.trace-size*))
      (loop (for trace-mi in *tr.trace-als*
        (incr trace-pos from 1)
        (Initial prev-trace-mi () )
        (do
          (:= ([*bk.trace-pos:mi* trace-pos] trace-mi)
```

---

**BK.RECORD-THE-TRACE**
Records the trace in *TR.TRACE-MIS* in the array *BK.TRACE-POS:MI*.

The :TRACE-PRED and :TRACE-SUCC fields of each trace MI are set to be the predecessor and successor trace MIs. The :TRACE-PRED of the first MI is set to be () (causing all of its predecessors to be treated as off-trace predecessors). The :TRACE-SUCC of the last MI is set to be its first successor (just so that we have both an on-trace and off-trace successor for later code).
This "buffers" the trace with dummy MIs that act as algorithmic sentinels. A dummy is spliced in between every conditional jump on the trace and its off-trace successor. A dummy is also spliced in between a join point in the trace and all the MIs that jump to that single point. The edge into the beginning of the trace is considered a join and buffered as such; all the edges leaving the last MI of the trace are considered off-trace split edges and also buffered.

One of the dummies joining to the beginning of the trace is distinguished as the "on-trace" predecessor of the first trace MI. Likewise, one of the dummies buffering the edges from the end is distinguished as the "on-trace" successor of the last MI. This guarantees that every conditional jump on the trace has an on-trace and off-trace successor.

But isolating the trace with dummies, we are guaranteed that every flow edge coming into and out of the trace has its tail or head off the trace, and that every trace MI is jumped to by at most one off-trace predecessor.

This buffering makes several things tractable, e.g. the handling of jumps on the trace that rejoin back to the trace and the merging of N-way jumps.

Rejoin dummies have the outgoing live variables stored in the :COPY-LIVE-OUT field. Similarly, split dummies have the incoming live variables stored in :COPY-LIVE-OUT. (Do we need the rejoin live info?) After producing the split and rejoin copies, we propagate the live info from the split dummies up into the split copies.

This "buffers" the trace with dummy MIs that act as algorithmic sentinels. A dummy is spliced in between every conditional jump on the trace and its off-trace successor. A dummy is also spliced in between a join point in the trace and all the MIs that jump to that single point. The edge into the beginning of the trace is considered a join and buffered as such; all the edges leaving the last MI of the trace are considered off-trace split edges and also buffered.

One of the dummies joining to the beginning of the trace is distinguished as the "on-trace" predecessor of the first trace MI. Likewise, one of the dummies buffering the edges from the end is distinguished as the "on-trace" successor of the last MI. This guarantees that every conditional jump on the trace has an on-trace and off-trace successor.

But isolating the trace with dummies, we are guaranteed that every flow edge coming into and out of the trace has its tail or head off the trace, and that every trace MI is jumped to by at most one off-trace predecessor.

This buffering makes several things tractable, e.g. the handling of jumps on the trace that rejoin back to the trace and the merging of N-way jumps.

Rejoin dummies have the outgoing live variables stored in the :COPY-LIVE-OUT field. Similarly, split dummies have the incoming live variables stored in :COPY-LIVE-OUT. (Do we need the rejoin live info?) After producing the split and rejoin copies, we propagate the live info from the split dummies up into the split copies.
An empty, permanent compacted MI is always added to the end of the schedule. This gives rejoiners to the "end of the schedule" a place to jump to and insures that even empty schedules are represented by one compacted MI (helping to maintain the assertion that all the predecessors of DEFs and all the successors of USEs are compacted).

(defun bk.coalesce ()
  (:= bk.cycleol* (makevector (+ 2 tr.schedule-size)) )
  (loop (incr cycle from 1 to *tr.schedule-size) (do
    (:= ([] bk.cycle:nl* cycle)
    (bk.mi:coalesce (schedule: [] *tr.schedule* cycle) )
    (loop (for mi in (mi:constituents ([] bk.cycle:mi* cycle) ) ) (do
      (if mi (then
        (:= (mi:first-cycle al) cycle) 
        (:= (mi:last-cycle al) cycle) )
      )))
    )))

  (add the an empty permanent MI to the end of the schedule.
  This MI is made to point at the on-trace dummy buffering
  the end of the trace, and the dummy made to point at
  the MI. This is done here since it won't follow out nicely in
  later passes.

  (let ( (last-nl (bk.nl:coalesce () )
    (last-dunmy (ni:trace-succ ([] bk.trace-poso1 tr.trace-size)))
    (last-mil (:= (mi:succ last-nl) (list last-dunmy) )
    (:= (mi:preds last-dunmy) (list last-mil) )
    
    (result
     (push succ-list ([] bk.cycle:mi* (+ 1 cycle) )
     (:= (mi:succ mi) (reverse succ-list) )
     ))
    )))

  (before splicing each split edge looks like:
  mi -> dummy -> off-trace
  After, it looks like:
  mi -> partial-sched -> def -> split-copies -> dummy -> off-trace

  (defun bk.schedule-mi:fix-succs ( mi cycle )
    (let ( (jump-mi (bk.mi:cond-jump-constituents mi cycle) )
      (loop (for junp-ai in jump-mi)
        (bind SUCC-BI (nioff-trace-succ junp-ai) )
        (initial succ-llst () ) /
        (do
          (push succ-llst succ-nl)
          (:= (nl:preds succ-ai) (top-level-substq nl Junp-al ftftft) )
          (result
            (push succ-list ([] bk. cycle:mi* (+ 1 cycle) )
            (:= (mi:succ mi) (reverse succ-list) )
            )))
      )))

  (check to see if any elements were before the conditional jump
  on the trace, but have now been scheduled below. For each,
  call it ABOVE-MI, we make a copy of it for placement before
  each off-trace follower. Note that these ABOVE-MIs are forced
  in reverse trace order, necessary since data precedence must
  be preserved in the sequence of copied instructions. Thus
  the DECS in the loop below. After the copies are made, we
  splice in the DEF and partial schedule of multi-cycle
  operations that spanned the split point that the code
  generator told us about.

  Before splicing each split edge looks like:
  mi -> dummy -> off-trace
  After, it looks like:
  mi -> partial-sched -> def -> split-copies -> dummy -> off-trace

  (loop (for junp-ai in jump-mi)
    (incr Jump-number froa 1)
    (bind Junplng-Bl ni
      junaped-to-al (nioff-trace-succ Junp-al)
      partial-sched&def
      (schedule:split *tr.schedule* cycle Jump-number))
    (do
      (loop (decr cycle2 froa (- (nl:trace-poa Junp-nl) 1) to 1)
        (bind above-al ([] bk.trace-poso1 cycle2) )
        (do
          (if (ftft (al:first-cycle above-al) ; scheduled?
            (> (al:first-cycle above-al) cycle) )
          then
            (car (mi:succ last-mi) )
            )))
    )
  )
)
(defun bk.splice-def-partial-schedule  
(coalesced-ni (def partial-schedule) off-trace-ni)
(let* (;
;jumped-to-ni (list coalesced-ni) 
(jumped-to-mi)

(if partial-schedule (then
(:= partial-schedule (list coalesced-ni) 
(schedule:length partial-schedule) )
)
(loop (descr cycle from (schedule:length partial-schedule) to 1) (do
(:= jumped-to-mi 
(bk.splice (list coalesced-ni) 
(jumped-to-mi) )
)
)
)

(let*( (trace-rejoined-ni (car (nlonccs rejolnlng-nl) ) )

(bk.rejoin-ing-al:fix-rejoin (rejoinlng-ni )
(let* (;
;"was produced to follow the jump " 
(mi:source jump-op) 
" schedule in cycles " 
(mi:first-cycle jump-op) 
" (mi:last-cycle jump-op) 
)
)
)

(bk.print-copy-after-message (copied-op jump-op)
(if *tr.print-copying? (then
(msg 0 t t "A new copy of " (mi:source copied-op) 
" schedule in cycles " 
(mi:first-cycle copied-op) 
" (mi:last-cycle copied-op) 
" )
)
)
)

(bk.rejoin-ing-al:fix-rejoin (rejoinlng-ni )
(let* (;
;"Note how having a special empty MI at the end of the schedule makes 
;"things work out nice — we always have a spot "at the end of the 
;"schedule" to make a rejoin to. 

(bk.print-copy-after-message (copied-op jump-op)
(if *tr.print-copying? (then
(msg 0 t t "A new copy of " (mi:source copied-op) 
" schedule in cycles " 
(mi:first-cycle copied-op) 
" (mi:last-cycle copied-op) 
" )
)
)
)

(bk.rejoin-ing-al:fix-rejoin (rejoinlng-ni )
(let* (;
;"Note how having a special empty MI at the end of the schedule makes 
;"things work out nice — we always have a spot "at the end of the 
;"schedule" to make a rejoin to. 

(bk.print-copy-after-message (copied-op jump-op)
(if *tr.print-copying? (then
(msg 0 t t "A new copy of " (mi:source copied-op) 
" schedule in cycles " 
(mi:first-cycle copied-op) 
" (mi:last-cycle copied-op) 
" )
)
)
)
Finally, figure out which operations were scheduled too early for the rejoin, but really need to be jumped to. Make sure that they are copied before the rejoin, so they get executed too. After the copies are made, we splice in the DEF and partial schedule of multi-cycle operations that spanned the join point that the code generator told us about. Each join edge then looks like:

```
off-trace—>join-copies—>use—>partial-schedule—>rejoined-nl
```

If a condition jump is copied up into the rejoin, then on the "off-trace" edge of the copy, we need to copy in all operations which were originally between the rejoin and the conditional on the trace, but haven't been copied up into the rejoin.

```
if (mi:cond-jump? mi-to-copy) (then
  (loop (incr trace-pos from (mi:trace-pos trace-rejoined-nl) to (- (mi:trace-pos mi-to-copy) 1))
    (bind below-nl (bk.trace-pos mi-to-copy) )
    (initial pred-ni copied-nl)
    (when (ftft (mi:first-cycle below-nl) (mi:last-cycle below-nl) rejoln-cycle) )
    (do (:= pred-al (bk.splice (list pred-ni) coalesced-ni) )
      (+ tr.rejoin-count)
      (bk.print-copy-before-message mi-to-copy trace-rejoined-nl))))
```

This copies a source MI up into a rejoin, doing the necessary extra copying for conditional jumps.

```
defun bk.copy-into-rejoin
  (rejoin-mi mi-to-copy trace-rejoined-mi)
```

```
defun bk.splice-use-partial-schedule
  (off-trace-al use partial-schedule coalesced-al)
```

```
defun bk.splice
  (list joining-ni off-trace-ni)
```

```
defun bk.print-copy-before-message
  (mi-to-copy trace-rejoined-nl)
```
(defun bk.print-copy-before-message (copied-ni rejoin-ni)
  (if *et.print-copying* (then
    (msg 0 t t "A new copy of "
      (mi:source copied-ni)
      " schedule in cycles "
      (mi:first-cycle copied-ni) ";" (mi:last-cycle copied-ni)
      " was produced to precede the rejoin at "
      (mi:source rejoin-ni) ";"
    ) )
  )
)

(defun bk.trace-ni:rejoin-cycle (ni)
  (if *et.trace-ni:rejoin-cycle* (al:trace-pos ni) )
)

(defun bk.build-trace-pos:rejoin-cycle ()
  (let ( (to-do () )
    (loop (for use-ni in *bk.use-def-nis*
      (when (ss 'use (al:operator use-ni) )
        (do
          (:s (nl:copy-live-out use-ni)
            (for ( (var loc) in (oper:part (nloper use-ni) 'body) )
              (save var) ) ) ) )
    )
    (loop (for split-ni in *bk.split-dummies*)
      (do
        (loop (for pred-ni in (mi:preds split-ni)
          (when (! (ni:conpacted? pred-ni) )
            (when (! (neaq to-do pred-ni) )
              (do
                (push to-do pred-ni) ) )
          )
        )
      )
    (loop (for use-ni in *bk.use-def-nis*
      (when (ss 'use (al:operator use-ni) )
        (do
          (loop (for pred-ni in (nl:preds use-ni)
            (when (! (nl:conpacted? pred-ni) )
              (when (! (neaq to-do pred-ni) )
                (do
                  (push to-do pred-ni) ) )
            )
          )
        )
      )
    )
  )
)

(defun bk.propagate-live-variables ()
  (let ( (to-do () )
    (loop (for use-ni in *bk.use-def-nis*
      (when (ss 'use (al:operator use-ni) )
        (do
          (loop (for pred-ni in (nl:preds use-ni)
            (when (! (nl:conpacted? pred-ni) )
              (when (! (neaq to-do pred-ni) )
                (do
                  (push to-do pred-ni) ) )
            )
          )
        )
      )
    )
  )
)

;*** (BK.PRINT-COPY-BEFORE-MESSAGE COPIED-MI REJOIN-MI)
;*** Used for debugging.
;***

;*** (BK.TRACE-MI:REJOIN-CYCLE MI)
;*** Finds the point in the schedule where we can make a rejoin that was
;*** originally to MI (an operation on the trace). That point is the
;*** highest point in the schedule below which are only operations which
;*** were originally below MI in the trace.
;***

;*** (BUILD-TRACE-POS:REJOIN-CYCLE)
;*** Builds the array *BK.TRACE-POS:REJOIN-CYCLE*, amazingly enough.
;*** T-P:R-C(t) is the earliest legal cycle in the schedule
;*** at which a join originally at trace position t can be made; e.g.
;*** given a join originally at trace position t, we make the join at
;*** cycle c in the schedule. T-P:R-C is calculated by observing:
;***
;*** Let Tc be the minimum trace position of all the operations scheduled
;*** at cycle c or later. Joins to trace positions after Tc must be remade
;*** at cycle c+1 or later.
;***

;*** (BK.PROPAGATE-LIVE-VARIABLES)
;*** This calculates live variables for all the split and join copies
;*** that were made. The live info is propagated up from the buffering
;*** split dummies up into the split copies; and from the USEs at joins
;*** up into the join copies.
;***

;*** (BUILD-TRACE-POS:REJOIN-CYCLE)
;*** Builds the array *BK.TRACE-POS:REJOIN-CYCLE*, amazingly enough.
;*** T-P:R-C(t) is the earliest legal cycle in the schedule
;*** at which a join originally at trace position t can be made; e.g.
;*** given a join originally at trace position t, we make the join at
;*** cycle c in the schedule. T-P:R-C is calculated by observing:
;***
;*** Let Tc be the minimum trace position of all the operations scheduled
;*** at cycle c or later. Joins to trace positions after Tc must be remade
;*** at cycle c+1 or later.
;***

;*** (BK.PROPAGATE-LIVE-VARIABLES)
;*** This calculates live variables for all the split and join copies
;*** that were made. The live info is propagated up from the buffering
;*** split dummies up into the split copies; and from the USEs at joins
;*** up into the join copies.
;***

;*** (BUILD-TRACE-POS:REJOIN-CYCLE)
;*** Builds the array *BK.TRACE-POS:REJOIN-CYCLE*, amazingly enough.
;*** T-P:R-C(t) is the earliest legal cycle in the schedule
;*** at which a join originally at trace position t can be made; e.g.
;*** given a join originally at trace position t, we make the join at
;*** cycle c in the schedule. T-P:R-C is calculated by observing:
;***
;*** Let Tc be the minimum trace position of all the operations scheduled
;*** at cycle c or later. Joins to trace positions after Tc must be remade
;*** at cycle c+1 or later.
;***

;*** (BK.PROPAGATE-LIVE-VARIABLES)
;*** This calculates live variables for all the split and join copies
;*** that were made. The live info is propagated up from the buffering
;*** split dummies up into the split copies; and from the USEs at joins
;*** up into the join copies.
;***

;*** (BUILD-TRACE-POS:REJOIN-CYCLE)
;*** Builds the array *BK.TRACE-POS:REJOIN-CYCLE*, amazingly enough.
;*** T-P:R-C(t) is the earliest legal cycle in the schedule
;*** at which a join originally at trace position t can be made; e.g.
;*** given a join originally at trace position t, we make the join at
;*** cycle c in the schedule. T-P:R-C is calculated by observing:
;***
;*** Let Tc be the minimum trace position of all the operations scheduled
;*** at cycle c or later. Joins to trace positions after Tc must be remade
;*** at cycle c+1 or later.
;***

;*** (BK.PROPAGATE-LIVE-VARIABLES)
;*** This calculates live variables for all the split and join copies
;*** that were made. The live info is propagated up from the buffering
;*** split dummies up into the split copies; and from the USEs at joins
;*** up into the join copies.
;***

;*** (BUILD-TRACE-POS:REJOIN-CYCLE)
;*** Builds the array *BK.TRACE-POS:REJOIN-CYCLE*, amazingly enough.
;*** T-P:R-C(t) is the earliest legal cycle in the schedule
;*** at which a join originally at trace position t can be made; e.g.
;*** given a join originally at trace position t, we make the join at
;*** cycle c in the schedule. T-P:R-C is calculated by observing:
;***
;*** Let Tc be the minimum trace position of all the operations scheduled
;*** at cycle c or later. Joins to trace positions after Tc must be remade
;*** at cycle c+1 or later.
;***

;*** (BK.PROPAGATE-LIVE-VARIABLES)
;*** This calculates live variables for all the split and join copies
;*** that were made. The live info is propagated up from the buffering
;*** split dummies up into the split copies; and from the USEs at joins
;*** up into the join copies.
(loop (while to-do)
  (initial nl () )
  (do
    (:= nl
      (loop (for possible-nl ln to-do) (do
        (lf (for-every (succ-ni in (nl:succs possible-nl) )
          (!== *tr.unknown-copy-live-out*
            ai:copy-live-out succ-ai) )
          then
            return poaalble-nl ) )
      result () ) )
    )
  )
)

*** We better have found an MI and it better be a copy
*** or USE/DEF and it better have one predecessor.
(assert (ftft nl
  (ss i (length (nl:preds nl) ) )
  (il (ni : copy-type ni)
    nenq (aloperator al) '(use def) ) ) )
(:s (al:copy-live-out ni)
  (loop (for succ-ai ln (al:succs al) )
    (reduce unlonq () (ai:live-in succ-ni) ) )
)
)

;:; to-do (top-level-renoveq nl to-do) 
(loop (for pred-nl ln (nl:preds al) )
  (when (! (al:coapacked? pred-ai) )
    (when (! (nenq to-do pred-ni) )
      (do
        (push to-do pred-al) ) )
  )
)
)

) )

(BK.MISCELLANEOUS-CLEAN-UP)
Perforas alscellaneous clean ups that aren't deserving of a separate
pass.
(defun bk.Blscellaneous-clean-up ()

;** Dispose of the source Mia on the trace:
(:= *tr.B* (aet-diffq *tr.s* *tr.trace-ni8*) )
;*** Dispose of the dunny Hat:

(loop (for ni in »bk.dunny-nis») (do
  bk.unsplice nl ) )
(:= *tr.a* (aet-diffq *tr.s* *bk.dunny-nls*) )
(:= *bk.dunny-nls* () )

;*** Update the expect value of each new elenent:
(loop (for nl ln »bk.new-pernanent-als») (do
  (al: find-expect al) )
  := »bk.new-peraanent-BlB* () )

;** Rénove any new USEs that have no unconpacted predecessors.
;* and any DEFs that have no unconpacted successors. They are
* useless.

(loop (for use-def-ni in »bk.use-def-nis») (do
  (if (for-every (ai in (if (= *use* (mi:operator use-def-ni) )
    (mi:preds use-def-ni) )
    (mi:successes use-def-ni) )
  (mi:compactdef mi) )
    then
      (bk.use-def-count
        bk.unsplice use-def-ni
        (:= *tr.s* (top-level-removeq use-def-ni *tr.s*) )) )
  (:= »bk.use-def-mis* () )

;*** Clear out the constituents field, which contains only old
;* MIs which we won't want to use again, so we can gc em.
(loop (incr cycle fron 1 to »tr.schedule-size») (do
  bk.unsplice use-def-ni
  (:= (al:constituents () »bk.cycleol* cycle) ) )))

;* Clear out the successors and predecessors of the trace elements
;* just in case.
(loop (for ni in »tr.trace-nia») (do
  (:= (ni:preds nl) ()
    (:= (niouccs nl) ()
      (:= (ni:trace-succ nl) ()
        (:= (ni:trace-pred al) 0 ) )
  )
)

;* If requested, add (TRACE 1) to each compacted instruction.
(if »tr.generate-trace-lnfo? (then
  (loop (incr cycle fron 1 to »tr.schedule-size») (do
    (push (ni:source () »bk.cycleol* cycle) )
    ' (trace ,*tr.trace-number») ) )
  )
)

;* Now we know how many new elements we've generated, so we
;* update »tr.ops-left. Don't need the rejoin list anymore,
;* either:
( := »tr.ops-left* (+ ftftft
  (+ »tr.rejoin-count
    (+ »tr.split-count* »bk.use-def-count) ) )

(:= »tr.rejoin-total* (+ ftftft »tr.rejoln-count»)
  (+ »tr.partlal-rejoln-count
    »bk.use-def-count) )
(:= »tr.split-total* (+ ftftft *tr.split-count*)
  (+ »tr.partial-split-total
    »bk.use-def-count) )
(:= »tr.partlal-spllt-total» (+ ftftft »tr.partial-split-count»)
  (+ »tr.partlal-rejoln-total
    »bk.use-def-count) )
(:= *tr.split-dunnles* () )
(:= *bk.rejoln-dunallea* () )
)

(BK.MI:COALESCE CYCLE-LIST)
*** Makes a new compacted MI representing the machine operations of a
*** cycle in the schedule. CYCLE-LIST represents a cycle in the schedule,
*** and is is a list of pairs of the form:

where SOURCE-MI represents the source operation that caused MACHINE-OPERATION to be generated (SOURCE-MI may be () ). The new MI is marked as compacted, its :SOURCE is set to be the list of machine operations in the cycle, its :CONSTITUENTS the SOURCE-MIs, and its :TRACE-DIRECTION to be a list of LEFT/RIGHT indicating which way the trace goes for that machine operation (significant only for jumps).

(defun bk.ai:coalesce (cycle-list)
  (let ((new-ai (aioew
                 number (+ str.mi-number*)
                 trace (str.trace-number*)
                 compacted? t)
                 (push *tr.s* new-ai)
                 (push *bk.new-permanent-mis* new-ai)
                 (loop (for (oper Bourca-nl) in cycle-list)
                   (initial current-prob 1.0)
                   (do (push (nl:source new-nl) oper )
                        (push (nl:constituents new-nl) source-nl)
                        (if (mi:cond-jump? source-nl )
                            (then (push (mi:trace-direction new-nl)
                                          current-prob)
                                   (:= current-prob
                                       (* current-prob
                                          (nr.edge-prob source-nl)
                                          (mi:off-trace-succ source-nl) )
                                   )
                            )
                        )
                        )
                   (result
                    (push (mi:edge-prob new-nl)
                         (- 1.0)
                    )
                   (loop (for prob in (mi:edge-prob new-nl) )
                     (initial sum 0.0)
                     (next sum (+ sum prob )
                     (result sum )
                     )
                   (:= (mi:source new-nl)
                       (dreverse (mi:edge-prob new-nl) )
                   )
                   (:= (mi:constituents new-nl)
                       (dreverse (mi:constituents new-nl) )
                   )
                   (:= (mi:trace-direction new-nl)
                       (dreverse (mi:trace-direction new-nl) )
                   )
                   new-ai )

(defun bk.ai:new-dummy ()
  (let ((new-nl (ni:new
                 number (+ str.mi-number*)
                 source (mi:source old-ni)
                 copy-type ()
                 trace (str.trace-number*)
                 compacted? ()
                 edge-prob (mi:edge-prob old-ni) )
                 (push *tr.s* new-ai)
                 (push *bk.new-permanent-mis* new-ai)
                 new-ni )

(defun bk.ai:new-use-def (use-or-def )
  (let ((new-nl (aioew
                 number (+ str.mi-number*)
                 source (list use-or-def)
                 copy-type ()
                 trace (str.trace-number*)
                 compacted? ()
                 edge-prob (1.0) )
                 (:= (bk.use-def-count) (+ 1 bk.use-def-count*)
                 (push *tr.s* new-ni)
                 (push *bk.new-permanent-mis* new-ni)
                 (push *bk.use-def-mis* new-ni)
                 new-ni )

PS: <C.S.BULLDOG.TRACE>BOOKKEEPER.LSP.1
(defun bk.mi:cond-jump-constituents (mi mi-cycle)
  (loop (for source-mi in (mi:constituents mi)
    (initial result () )
    (do (if (mi:cond-jump? source-mi)
        (mi:cond-jump? source-mi)
      (then (push result source-mi ) ) )
    (result (reverse result)) ) )

(defun bk.splice (preds target succ)
  (assert (subset? preds (nl:preds succ)) )
  (assert (not (neaq target (al:preds succ)) )
  (let ( (preds (ni:preda target))
    (succ (car (nl:preds target)) )
    (for (pred ln preds) (do
      (:= (nl:preds pred)
        (top-level-substq succ target (al:succs pred) ) )
    )
    (result (reverse result)) ) )

(defun bk.unsplice (target)
  (let ( (preds (mi:preds target))
    (succ (car (nl:preds target)) )
    (for (pred ln preds) (do
      (:= (nl:preds pred)
        (top-level-substq succ target (mi:preds pred) ) )
    )
    (result (reverse result)) ) )

(:= (ni:preds succ)
   (unionq preds (top-level-removelq target (ni:preds succ) ))
)

(:= (ni:succs target) ()
(():= (ni:preds target) ()
(())

::* (build *tr.build-module-list*)
::* (build.compile *tr.build-module-list*)

(:= *tr.build-module-list* '(
   trace:n1
   trace:naddr-rec
   trace:nis-to-pnaddr
   trace:trace-picker
   trace:bookkeeper
   trace:display
   trace:compact
   trace:compact-options
))

(:= *build-module-list* (append *build-module-list* *tr.build-module-list*))

1
PS:<C.S.BULLDOG.TRACE>BUILD.LSP.4
Initialization of all the trace modules. This is all gross organization, but it will have to do until the bookkeeper is rewritten.

(defun tr.Initialize ()
  (tr.initialize-bookkeeper)
  (tr.initialize-trace-picker)
  (initialize-code-generator)
  ;; Initial ops 0
  (= *tr.start-length* (length *tr.s*));;; Saved for final print-out
  (= *tr.ops-left* 0) ;;; For running total of ops
  (loop (for elt in *tr.s*) ;;; still needing compaction.
    (lf (! (ai:coapacted? elt) )
      (++ ops) ) ) )

(defun compact ( naddr )
  (let ( (tr.space-node*  »tr.space-node») ) ;;; Dynamically bind these vars
    (= *tr.space-node* (car *tr.trace-nis*) ) ;;; in case we change them during
    (= last-nl (last-elt *tr.trace-nis*) ) ;;; compaction below — dynamic
    (= live-before (ni:live-in first-nl) ) ;;; binding will restore their values.
    (= live-after )

    ;; Initialization (done once per setting of *tr.s*):
    (tr.initialize)
    (set-conpactor-flags) ;;; Sets flags for space-saving.
    ;;; display methodology, etc.
    (tr.get-conpactor-flags) ;;; Builds the global list *tr.s*
    (= *tr.start-length* (length *tr.s*));;; Saved for final print-out
    (= *tr.ops-left* 0) ;;; For running total of ops
  )

(defun conpact ( naddr )
  (let ( (»tr.space-node» »tr.space-node») ) ;;; Dynamically bind these vars
    (= *tr.space-node* (car *tr.trace-nis*) ) ;;; in case we change them during
    (= last-nl (last-elt *tr.trace-nis*) ) ;;; compaction below — dynamic
    (= live-before (ni:live-in first-nl) ) ;;; binding will restore their values.
    (= live-after )

    ;; Initialization (done once per setting of *tr.s*):
    (tr.initialize)
    (set-conpactor-flags) ;;; Sets flags for space-saving.
    ;;; display methodology, etc.
    (tr.get-conpactor-flags) ;;; Builds the global list *tr.s*
    (= *tr.start-length* (length *tr.s*));;; Saved for final print-out
    (= *tr.ops-left* 0) ;;; For running total of ops
  )

COMPACT OPTIONS

This module contains the definitions of options dealing with the trace compactor.

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

(def-option *tr.display-level* 4 trace: "Controls how much is printed out during compaction.
  0 - absolutely nothing.
  1 - only final stats.
  2 - like 1, but dumps a histogram of uncompacted MIs left during scheduling. What fun.
  3 - trace info for each trace, no schedule.
  4 - trace info + schedule for each trace.
  5 - All of 4, plus the elements of each trace.
"
)

(def-option *tr.space-node* () trace: "Controls the way that space saving is done.
  () no space saving.
  'MRT minimum release times. Won't let jumps get scheduled until the source order preceding ops could have been scheduled, had there been no resource conflicts.
  'NSC no splice copies. Refuses to schedule jumps until all the source order preceding ops HAVE been scheduled. Thus no splice copies could ever be generated.
  'CJO preserve the source order of conditional jumps by not allowing a conditional to move above a previous conditional.
"
)

(def-option *tr.window* () trace: "When *TR.SPACE-MODE* = 'MRT is used, *TR.WINDOW* is subtracted from the calculated release time. Yes. Virginia, you can set *TR.WINDOW* to -100 and get a lot of empty cycles.
"
)

(def-option *tr.delayed-space-mode* () trace: "If *TR.DELAYED-SPACE-MODE* is non-(), then starting with trace *TR.DELAYED-SPACE-MODE-TRACE*, the *TR.DELAYED-SPACE-MODE* and *TR.DELAYED-WINDOW* replace *TR.SPACE-MODE* and *TR.WINDOW*.
"
)

(def-option *tr.delayed-window* () trace: "See *TR.DELAYED-SPACE-MODE*.
"
)

(def-option *tr.delayed-space-mode-trace* 2 trace: "See *TR.DELAYED-SPACE-MODE*.
"
)

(def-option *tr.trace-picker* "normal trace: "
  Controls the way that traces are picked.
  'NORMAL conservatively goes past splits and joins.
  'BB basic block compaction only.
  'LIBERAL goes past splits and joins whenever it can, ignoring edge-probability and just using expect.
"
)

(def-option *tr.generate-trace-info* () trace: "Dumps certain trace-information into the output parallel NADDR.
  () do nothing.
  T put (TRACE 1) in each compacted instruction.
"
)

(def-option *tr.generate-code-hook* () trace: "Undocumented, sorry.
"
)

(def-option *tr.dag-hook* () trace: "Undocumented, sorry.
"
)

(def-option *tr.break-before-scheduling* () trace: "If T then the compactor will stop at a breakpoint right before the current trace is scheduled (given to the codegenerator). If it is a list of trace numbers, then the compactor will stop only on those traces.
"
)

(def-option *tr.break-after-bookkeeping* () trace: "If T then the compactor will stop at a breakpoint right after the current trace has been compacted and sunged by the bookkeeper (i.e. right before picking the next trace. If it is a list of trace numbers, then the compactor will stop only on those traces.
"
)

(def-option *tr.print-copying?* () trace: "If T then the bookkeeper will print out information about all join and split copies as they are made.
"
)
Compiler Declarations.

Any module that needs these declarations must do:

(INCLUDE TRACE DECLARATIONS)

(eval-when (compile)
  (build '(
    utilities:sharp-sharp
    trace:ni
    interpreter:naddr
  )))

(declare (special
  *tr.break-before-scheduling*
  *tr.break-after-bookkeeping*
  *tr.dag-hook*
  *tr.delayed-space-node*
  *tr.delayed-window*
  *tr.delayed-space-node-trace*
  *tr.display-level*
  *tr.generate-code-hook*
  *tr.generate-trace-info?*
  *tr.ml-number*
  *tr.ops-left*
  *tr.partial-rejoin-count*
  *tr.partial-rejoin-total*
  *tr.partial-split-count*
  *tr.partial-split-total*
  *tr.print-copying?*
  *tr.rejoin-count*
  *tr.rejoin-total*
  *tr.schedule-size*
  *tr.schedule*
  *tr.space-node*
  *tr.split-count*
  *tr.split-total*
  *tr.start-length*
  *tr.*
  *tr.total-cycles*
  *tr.trace-number*
  *tr.trace-picker*
  *tr.trace-size*
  *tr.unknown-copy-live-out*
  *tr.window*)
  )
)
(eval-when (compile load)
  (include trace-declarations)
)

(defun tr.display-trace-info (traceno)
  (caseq *tr.display-level*
    (2
      (if (= traceno 1) (then
        (msg 0 t "Histogram of total M1s left (10 M1s):")
        (loop (incr 1 from 1 to *tr.ops-left* by 10) (do
          (print t)))))
    (3 4 5
      (msg 0 t "Trace No.: " traceno " Trace size: " *tr.trace-size*
        " Offtrace OPs left: " *tr.ops-left* t)
      (msg 0 t "Trace elements: " (* mi-list:numbers *tr.trace-size* 10000 10000)
        (loop (for (oper direction mi off-live) in source-record-list)
          (msg 1 t "Arguments passed GENERATE-CODE:"
            "LIVE-BEFORE = " (h live-before 100 100) t
            "LIVE-AFTER = " (h live-after 100 100) t)
          (loop (for (oper direction mi off-live) in source-record-list)
            (incr 1 from 1)
            (do
              (msg 1 t "(mi: number mi 4) ":
                (h oper 100 100) (t 86) "
                (j direction -7) "
                (j (mi:trace mi 8) "
                (h off-live 100 100) t)
              (msg t)))))
    (6
      (msg 0 t "Partial-rejoin/split copies from last trace:"
        "str.partial-rejoin-count" "//" "str.partial-split-count" t)
    )))

(defun tr.display-generate-code-arguments
  (live-before source-record-list live-after)
  (if (= 5 *tr.display-level*) (then
    (msg 0 t "Arguments passed GENERATE-CODE:"
      "LIVE-BEFORE = " (h live-before 100 100) t
      "LIVE-AFTER = " (h live-after 100 100) t)
    (loop (for (oper direction mi off-live) in source-record-list)
      (incr 1 from 1)
      (do
        (msg 1 t "(mi: number mi 4) ":
          (h oper 100 100) (t 86) "
          (j direction -7) "
          (j (mi:trace mi 8) "
          (h off-live 100 100) t)
        (msg t)))))

(defun tr.display-one-schedule()
  (caseq *tr.display-level*
    (4 6
      (msg 0 t "Schedule:"
        (loop (incr cycleno from 1 to *tr.schedule-size*) (do
          (msg 0 t (j cycleno 3 ""))
          (for (oper source-mi) in (schedule [] *tr.schedule* cycleno))
            (do
              (if (> (flat-size oper) (chrct)) (then
                (msg 0 t "")))))
        (msg 0 ""))))

(defun tr.display-whole-sched-info (traceno)
  (caseq *tr.display-level*
    (1 2 3 4 5
      (msg 0 t "FINAL SCHEDULING TOTALS:"
        "Length of original sequence:" (t 36) *tr.start-length* t
        "Total rejoin copies:" (t 36) *tr.rejoin-total* t
        "Total split copies:" (t 36) *tr.split-total* t
        "Total partial-rejoin copies:" (t 36) *tr.partial-rejoin-total* t
        "Total partial-split copies:" (t 36) *tr.partial-split-total* t
        "Total traces:" (t 36) (+ -1 traceno) t
        "Total cycles scheduled:" (t 36) *tr.total-cycles* t))
    (6
      (msg 0 "")
    )))

PS:<C.S.BULLDOG.TRACE>DISPLAY.LSP.11
MI

An MI (micro-instruction) represents either an uncompact ed source operation or else a compacted instruction. Taken as a group the MIs form the flow graph used by the trace scheduler.

(def-struct mi)

The following fields are filled in when a record is made from a source instruction and handed to the compactor in the first place, and when a new record is made from old ones.

number : A unique number used for naming this MI.
source : The list of the single source NADDR operation this represents (if uncompact ed) or the list of machine operations in the instruction (if compact ed).
expect : The relative probability of the MI's execution.
(succs () suppressed) : A list of the MIs to which control could next flow from this one. SU is for successor.
(preds () suppressed) : A list of the MIs from which control could have flowed to this one. PR is for predecessor.
edge-prob : An assoc list. Having (a4 .6) on n2's list means that a4 is a successor with probability .6 of being jumped to next after the execution of m4. All of the probs on the list should total 1, though that's not necessary for the code to work correctly. Functions edge-prob and edge-prob: set manipulate the list.
compacted? : True if the MI represents a compacted machine instruction or if the trace-scheduler no longer wants to consider this record for scheduling. False if this represents an uncompact ed source operation.

The following depend upon the choice of a trace and are filled in after an operation has been placed on a trace:

(trace-succ () suppressed) : The next element on the current trace.
(trace-pred () suppressed) : The previous element on the current trace.
trace-pos : Which element (1-based) of the current trace this is.
first-cycle : First and last cycles of the generated machine.

last-cycle : machine operations corresponding to this source operation.
trace-direction : Used in an MI built from several MIs compacted together. Used when n > 0 cond-jumps are compacted to form part of the MI. A list containing n elements, each LEFT or RIGHT and each corresponding to the succs in the same position in the list succs (which is itself, however, of length n+1). Also corresponds to the cond-jumps in the MI:SOURCE field, which is expected to have its cond-jumps in source order. A LEFT in the field indicates that the jump is to the next trace element if true, off-trace if false. RIGHT is the reverse.

The following fields are used during bookkeeping:

(constituents () suppress) : As a new MI is being formed out of a list of ops which were scheduled in the same cycle, this field buffers the list of those ops.
(copy-type : For uncompact ed MIs only, one of:
() - if this MI is an original source instructions.
JOIN - if this MI is a rejoin copy.
SPLIT - if this MI is a split copy.
(copy-live-out : For uncompact ed rejoin or split copy MIs only; this contains the variables live on exit from the MI.
() for original source MIs.
trace : This field is used for picture drawing and debugging only. It is the number of the trace that produced this MI. If the MI is compacted then it is the number of the trace that compacted it; if uncompact ed, it is the trace that made it.
(translated-to-source : Contain s the actual pnaaddr produced from the MI.
(MI:OPERATOR MI) : For uncompact ed MIs, the NADDR operator. () for compacted MIs.
(MI:OPER MI) : For uncompact ed MIs, the NADDR operation. () for compacted MIs.
(MI:COND-JUMP? MI) : True if MI represents a conditional jump (has more than one successor).
(MI:REJOIN? MI)
True if MI is jumped to by other MIs (has more than one predecessor).

(MI-LIST:NUMBERS MI-LIST)
Takes a list of MIs and returns a corresponding list of their numbers.

#»MI 36
Returns the MI numbered 36.

(MI:LIVE-OUT MI)
Returns the list of names live on exit from MI.

(MI:LIVE-IN MI)
Returns the list of names live on entrance to MI.

(MI:LIVE-OUT-ON-EDGE MI DIRECTION)
Returns the list of variables live on entrance to one of MI's successors. DIRECTION is LEFT or RIGHT or (), selecting either the left or right successor of MI or neither successor.

(MI:ON-TRACE-DIRECTION MI)
(MI:OFF-TRACE-DIRECTION MI)
These functions return which way (LEFT or RIGHT) the on-trace and off-trace edges of an MI go (LEFT always for a non-conditional-jump). If MI has no on-trace successor marked yet, the trace is assumed to go to the left.

(MI:OFF-TRACE-SUCC MI)
The off-trace successor of MI. Valid only during bookkeeping.

(MI:OFF-TRACE-PREDS MI)
The list of off-trace predecessors of MI. Valid only during bookkeeping.

(eval-when (compile)
  (build '(utilities :sharp-sharp
             interpreter:naddr) ))

(defvar *tr.unknown-copy-live-out* (cons **tr.unknown-copy-live-out*) )
;*** Special marker showing that we don't know the value of
;*** MI:COPY-LIVE-OUT field.

(defun mi:operator ( mi )
  (assert (mi:is mi) )
  (if (mi:compacted? mi)
      ()
      (oper:operator (car (mi:source mi)) ) ) )

(defun mi:oper ( mi )
  (assert (mi:is mi) )
  (if (mi:compacted? mi)
      ()
      (car (mi:source mi)) ) )

(defun mi:cond-jump? ( mi )
  (assert (mi:is mi) )
  (> (length (mi:succs mi)) 1) )

(defun mi:rejoin? ( mi )
  (assert (mi:is mi) )
  (> (length (mi:preds mi)) 1) )

(defun mi-list:numbers ( mi-list )
  (for (mi in mi-list) (save
    (assert (mi:is mi) )
    (mi:number mi)) ) )

(def-sharp-sharp mi
  '(mi-with-number ,(read) ) )

(declare (special *tr.* ))

(defun mi:live-out ( mi )
  (assert (mi:is mi) )
  (if (= *tr.unknown-copy-live-out* (mi:copy-live-out mi) )
    (oper:live-out (car (mi:source mi)) )
    (mi:copy-live-out mi)) )

(defun mi:live-in ( mi )
  (assert (mi:is mi) )
  (if (= *tr.unknown-copy-live-out* (mi:copy-live-out mi) )
    (oper:live-in (car (mi:source mi)) )
    (mi:copy-live-out mi)) )

(defun mi:live-out-on-edge ( mi direction )
  (assert (mi:is mi) )
  (if (= *tr.unknown-copy-live-out* (mi:copy-live-out mi) )
    (oper:live-out-on-edge (car (mi:source mi)) direction) )
  (t (intersectonq
    (oper:part (car (mi:source mi)) 'written)
    (mi:copy-live-out mi) ) )
    (union)
    (top-level-reaoveq (oper:part (car (mi:source mi)) 'written)
    (mi:copy-live-out mi) ) )
  (else
    (union)
    (cop-level-removq (oper:part (car (mi:source mi)) 'written)
    (mi:copy-live-out mi) )
    (loop (for-each-oper-operand-read (car (mi:source mi)) name)
    (save name) ) ) )

(defun mi:live-out-on-edge ( mi direction )
  (assert (mi:is mi) )
  (if (= *tr.unknown-copy-live-out* (mi:copy-live-out mi) )
    (oper:live-out-on-edge (car (mi:source mi)) direction) )
  (t (intersectonq
    (oper:part (car (mi:source mi)) 'written)
    (mi:copy-live-out mi) ) )
    (union)
    (top-level-reaoveq (oper:part (car (mi:source mi)) 'written)
    (mi:copy-live-out mi) ) )
  (else
    (union)
    (cop-level-removq (oper:part (car (mi:source mi)) 'written)
    (mi:copy-live-out mi) )
    (loop (for-each-oper-operand-read (car (mi:source mi)) name)
    (save name) ) ) )


(defun mi:copy-live-out (mi)
  (mi:live-in
   (caseq direction
     (left (car (mi:succs mi))
     (right (cadr (mi:succs mi)))))

(defun mi:on-trace-direction (mi)
  (assert (mi:is mi))
  (if (or (not (mi:trace-succ mi))
      (<= (car (mi:succs mi)) (mi:trace-succ mi)))
     'left
     'right)

(defun mi:off-trace-direction (mi)
  (assert (mi:is mi))
  (assert (<= 2 (length (mi:succs mi))))
  (if (or (not (mi:trace-succ mi))
      (<= (car (mi:succs mi)) (mi:trace-succ mi)))
     'right
     'left)

(defun mi:off-trace-succ (mi)
  (assert (mi:is mi))
  (if (or (not (mi:trace-succ mi)) (car (mi:succs mi))
      (cadr (mi:succs mi))
      (car (mi:succs mi))))

(defun mi:off-trace-preds (mi)
  (assert (mi:is mi))
  (top-level-removelq (mi:trace-pred mi) (mi:preds mi)))
This module converts compacted microinstructions into parallel naddr code.

To convert a list of MI's (such as returned by the compacter) into parallel naddr (PNADDR):

\[
\text{mis->pnaddr mi-list}
\]

The conversion is very simple minded—preorder traversal of the flow graph, starting at the one node that has the (START) source. Each node is visited only once (using the MI:TRANSLATED-TO-SOURCE flag to remember visits).

It is assumed that there are one or two flow successor for each MI. Conditional boolean jumps (TRUEGO, FALSEGO) are converted into the parallel naddr form with two explicit labels, like the if-then-else naddr operations.

\[
\text{eval-when (compile load)}
\]
\[
\text{(include trace:declarations)}
\]

\[
\text{(declare (special *als-to-do* ;*** stack of al's possibly not yet processed)}
\]

\[
\text{*** Translate the single MI MI into a parallel naddr statement. As a side effect, push on *MIS-TO-DO* the successors of this MI. Only}}
\]
\[
\text{*** the labels of source naddr statements are changed (for jumps). *** The new instruction is also placed in the translated-to-source}}
\]
\[
\text{*** field of the MI.}
\]

\[
\text{(defun mi:pnaddr ( mi )}
\]
\[
\text{(:s */a-to-do* () )}
\]
\[
\text{(assert (mi:1s mi) )}
\]
\[
\text{(:= (mi:translated-to-source mi) t) )}
\]
\[
\text{(let ( (pnaddr-stream () ) (pnaddr () ) )}
\]
\[
\text{*** clear the "translated" flag of every MI, and if it has}
\]
\[
\text{*** no predecessors, push it on our to-do stack.}
\]
\[
\text{(for (mi in mi-list) )}
\]
\[
\text{(do (assert (mi:1s mi) ) )}
\]
\[
\text{(*= (mi:translated-to-source mi) ) )}
\]
\[
\text{(push *mis-to-do* mi) ) )}
\]
\[
\text{*** while there are untranslated MI's, do}
\]
\[
\text{*** convert each one to pnaddr}
\]
\[
\text{(loop (initial mi () )}
\]
\[
\text{(while *mis-to-do* )}
\]
\[
\text{(do (assert (mi:1s mi) ) )}
\]
\[
\text{(pop *mis-to-do* mi) )}
\]
\[
\text{(*= (mi:translated-to-source mi) ) ) (then (}
\]
\[
\text{(if (}
\]
\[
\text{(*= (mi:translated-to-source mi) ) ) ) ) ) )}
\]
\[
\text{(pop pnaddr-stream pnaddr) ) ) ) )}
\]
\[
\text{)(reverse pnaddr-stream ) ) )}
\]

1

PS: <C.S.BULLDOG.TRACE>MIS-TO-PNADDR.LSP.6
(to
  (push popers oper)))

;*** record the conditional jump, if any, as a COND
  (if cond-oper (then
    (push cond-oper '(goto ,(mi:pnaddr-label (car succs) ))
    (push popers '(f,(dreverse cond-oper) )))
    ;*** push the successors on the stack
    (:= succs (ni:8uccs nl)) ;*** in case you forgot (ha ha).
    (for (succ ln succs)
      (do (if (! (nl:tranelated-to-eource succ))
        (push *nis-to-do* succ) ) )
      ;*** Hack, if there is one successor and it's already
      ;*** been translated, we need to do an explicit GOTO.
      ;*** O.w. it will be done next, it's code will follow
      ;*** immediately.
      (if (li (= 1 (length succs))
        (mi:translated-to-source (car succs))
        (push popers '(goto ,(mi:pnaddr-label (car succs) )) ) )
      )
    )
    (:= (mi:translated-to-source ni) (dreverse popers))
  )
)

;*** Return the label of an MI.
;***
(defun mi:pnaddr-label ( mi )
  (assert (mi:is mi) )
  (atomconcat '1 (mi:number mi) )
)
; N-ADDRESS CODE INTO COMPACT RECORDS TRANSLATION. J. Fisher
; 11/30/81

(defun tr.get-*tr.s* (naddr) (nr.naddr-to-records naddr))

;; nr.naddr-to-records does the actual conversion. First, an end statement is provided by nr.cleanly-end-program if none exists. Then records are formed, but these have actual names rather than pointers where other records belong (i.e. as flow successors) and have names, not the needed numbers, for registers. A final routine fixes that...

(defun nr.naddr-to-records (naddr-list)
  (nr.form-records (nr.cleanly-end-program naddr-list))
  (nr.replace-names-with-pointers)
  (nr.eliminate-gotos)
  (nr.set-all-edge-probabilities)
  (nr.calculate-expects)
  (:= »nr.xref-list* ())
  nil)

;; To form records, we the following. The list is processed statement by statement. If the statement is not a label or an expect, a record is built by a call to nr.make-new-record. Then all the labels that referred to this statement are placed in the label table as doing so.
;; When a label is encountered, it is placed on the currently active list, where labels are accumulated until an operative statement is encountered.

(defvar special
  *nr.xref-list* )

(defun nr.form-records (naddr-list)
  (:= *tr.s* nil)
  (:= *nr.xref-list* nil)
  (let ((active-labels nil)
        (stat-count 0)
        (current-record nil) )
    (for (stat in naddr-list)
      do
        (cond
          ((nr.type-of-stat-that-makes-records stat)
            (let* ( (nane (nr.get-record stat)
                          (fall-through (+ 1 stat-count))
                          (source (repr source
                                    (nr:source)
                                    (nr:source (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
                                                (nr:get-record source
                                                (nr:source)
::: change default field values for special operators

(caspeq (oper:group stat))

(cond-jump
 (:= (mi:succ mi)
 '((oper:part stat 'label1)
 (oper:part stat 'label2)
 (oper:part stat 'label12)
 (oper:part stat 'label12)
 (fail-through) ) )
)

(if-then-else
 (:= (mi:succ mi)
 '((oper:part stat 'label1)
 (oper:part stat 'label2)
 (fail-through) ) )
)

(goto
 (:= (mi:succ mi) (list (cadr stat) ) )
)

(live def-block esc
 (:= (mi:compacted? mi) t )
)

(end stop
 (:= (mi:succ mi) ()
 (:= (mi:compacted? mi) t )
)
)

::: We've built only flowauxa, so here we build the
::: set of flowprs...

(defun nr.build-flow-pre ()
 (for (a ln →tr.s«)
   (do
     (for (succ in (mi:succs a))
       (push (mi:preds succ) a))))
)

::: THE FOLLOWING IS THE STUFF RELATED TO FILLING IN THE EXPECT AND JUMP
::: JUMP PROBABILITY FIELDS

::: He's making a list (ta ta), checking it twice (la la).

(defun nr.calculate-expects ()
 (for (elt in (reverse →tr.s«))
   (do
     (mi:find-expect elt))))

::: mi:find-expect retrieves the mi:expect value of an MI, or forces
::: a calculation of it if it hasn't been done yet. Because of possible
::: flakiness in various codegenerators, we guarantee the expect is at
::: least a tiny number.

(defun mi:find-expect ( elt )
  (if (! (mi:expect elt)) (then
    ;; protect against circularity
    (:= (mi:expect elt) 1.0)
  (else
    (:= (mi:expect elt) 1.0)
    (:= (mi:expect elt)
      (loop (for pred in (mi:preds elt))
        (when (! (trace-fence:is pred))
          (initial expect 0.0)
          (do
            (:= expect (+ expect
              (if (mi:cond-jump? pred)
                (* (mi:find-expect pred)
                  (nr.edge-prob pred elt)
                  (mi:find-expect pred)))))
            (result expect))))
      (if (loop-start:is elt) (then
        (:= (mi:expect elt)
          (+ (mi:expect elt) (mi:iteration-count elt)))))
    (else (if (loop-end:is elt) (then
      (let ( (loop-start (nr.find-loop-start (loop-end:naae elt)) )
        (:= (mi:expect elt)
          (/ (mi:find-expect loop-start)
            (mi:iteration-count loop-start)))))
      (:= (mi:expect elt) (max 1.0E-50 (mi:expect elt)))))
    (:= (mi:expect elt) (aax 1.0E-80 (mi:expect elt)))
  )))

(defun mi:iteration-count (elt)
 (caddar (mi:source elt)))

(defun nr.find-loop-start (name)
 (loop
   (for elt in →tr.s«)
   (do
     (if (eq (loop-start:la elt) (loop-start:nane elt))
       (eq (loop-start:name elt) name))
       (return elt)))
 (result nil)))

(defun loop-start:is (elt)
  (≡ 'loop-start (oper:operator (car (mi:source elt)))))

(defun trace-fence:is (elt)
  (≡ 'trace-fence (oper:operator (car (mi:source elt))))

(defun loop-start:name (elt)
  (oper:part (car (mi:source elt)) 'label1))

(defun loop-end:is (elt)
  (≡ 'loop-end (oper:operator (car (mi:source elt))))

(defun loop-end:name (elt)
  (oper:part (car (mi:source elt)) 'label1))
::: Edge prob is formed here, and passed along when copies are made in the ::: bookkeeping phase.

(defun nr.edge-prob (pred succ)
  (assert (memq succ (ni:succs pred)))
  (loop
    (for next-succ in (ni:succs pred))
    (for next-edge-prob in (mi:edge-prob pred))
    (do (if (eq next-succ succ)
          (then (return next-edge-prob)))))

(defun nr.set-all-edge-probabilities ()
  (for (elt in *tr.s*)
    (do
      (if (ni:cond-jump elt)
        (then
          (:= (ni:edge-prob elt) (- 1.0 (ni:jump:prob elt))))
        (else
          (:= (ni:edge-prob elt) '1.0)))))

(defun ni:jump:prob (elt)
  (oper:part (car (mi:source elt)) 'probability))

;::: Here we attempt to eliminate all the goto statements. It is remarkably;::: easy if this really works. It is only difficult in that we are altering;::: *tr.s* at the same time that we're running through it.

(defun nr.eliminate-gotos ()
  (loop
    (initial goto-ni (nr.find-goto-ni))
    (while goto-ni)
    (do (bk.unsplice goto-ni)
      (:= *tr.s* (set-diffq *tr.s* (list goto-ni)))
      (next goto-ni (nr.find-goto-ni)))))

(defun nr.find-goto-ni ()
  (loop
    (for elt in *tr.s*)
    (do
      (if (eq (oper:group (car (mi:source elt)) goto)
        (return elt)))
    (result nil)))
TRACE PICKER

This module implements the picking of the next trace of uncompacted MIs from
the MI flow graph.

(TR.INITIALIZE-TRACE-PICKER)
Initializes the module by clearing *TR.TRACE-MIS*.

(TR.PICK-TRACE)
Picks the next trace from the flow graph, setting *TR.TRACE-MIS* to
be the trace and *TR.TRACE-SIZE* the size of the trace and
decreasing *TR.OPS-LEFT*. *TR.TRACE-MIS* is set to be () if there
are no uncompacted MIs left.

Traces are picked by first finding the uncompacted MI with the largest
:EXPECT value; that MI becomes the seed of the trace. The trace picker
then moves forward, incrementally growing the trace from the end. To
find the next MI, it looks at the current end of the trace and finds
the successor that meets the trace criteria and adds it to the end of
the trace. After growing forward, the trace picker grows the trace
backwards analogously.

Whichever direction we are growing the trace, the same criteria are
used to see if an edge between two MIs belongs to the trace. Suppose
we have two MIs, PRED-MI and SUCC-MI. If we are growing the trace
forward, PRED-MI is the current end of the trace. If we are growing
the trace backward, then SUCC-MI is the current beginning of the trace.
In either case, we look at the edge from PRED-MI to SUCC-MI; both of
the MIs must be uncompacted.

The setting of *TR.TRACE-PICKER* determines which edge criteria are used:

NORMAL
The edge from PRED-MI to SUCC-MI has the highest edge probability
of all exits from PRED-MI.

The edge from PRED-MI to SUCC-MI is the most likely to be
executed of all the predecessor edges coming into SUCC-MI. That
is, the edge contributes the most :EXPECT to SUCC-MI of all
of the predecessor edges.

BB
"Basic block" — traces consist exactly of basic blocks.

LIBERAL
PRED-MI has the highest :EXPECT of all the predecessors of SUCC-MI.
SUCC-MI has the highest :EXPECT of all the successors of PRED-MI.

In all three criteria, PRED-MI and SUCC-MI must be uncompacted. To
guarantee that a TRACE-FENCE pseudo-op can only occur at the end of
a trace, PRED-MI can't be a TRACE-FENCE pseudo-op.

(eval-when (compile load)
(inclure trace:declarations))

(defun tr.initialize-trace-picker ()

(defun tr.pick-trace ()
(let ((seed-ni (tp.mi-list:max-expect-ni *tr.s*)))
  (:= *tr.trace-mis* ()
      (loop (initial next-ni seed-ni)
        (while next-ni)
          (do (:= (ni:trace-pos next-ni) t)
            (push *tr.trace-mis* next-ni)
            (next next-ni (tp.mi:next-forward-trace-mi next-ni))
            ;*** Reverse the trace and remove the seed from the front.
            (:= *tr.trace-mis* (dreverse &amp;))
            (pop *tr.trace-mis*)
            ;*** Gobble MIs backward from the seed.
            (loop (initial next-ni seed-ni)
              (while next-ni)
                (do (:= (ni:trace-pos next-ni) t)
                  (push *tr.trace-mis* next-ni)
                  (next next-ni (tp.mi:next-backward-trace-mi next-ni))
                  ;*** Set the trace size and number of operations left.
                  (:= *tr.trace-size* (length *tr.trace-mis*)
                    (:= *tr.ops-left* (- *tr.ops-left* *tr.trace-size*)
                        0))))

(defun tp.mi:next-forward-trace-mi (mi)
  (loop (for succ-ni in (mi:succ mi)) (do
    (if (&amp; (ni:trace-pos succ-ni)
      (tp.pred-mi:succ-mi:right-edge? mi succ-ni)
        (then
          (return succ-ni)
        ))
      (result () ) )
  )
)

(defun tp.mi:next-backward-trace-mi (mi)
  
)
```lisp
(defun tp.Bioext-backward-trace-ai (al)
  (loop (for pred-ai ln (al:preds ai) ) (do
    (lf (ftft (! (Bi:trace-pos pred-al) )
        (tp.pred-al:succ-ai:right-edge? pred-ni nl) )
    (then
      (return pred-ni) ) )
  (result () ) )
)
```

```lisp
(defun tp.pred-ml:succ-ni: right-edge? ( pred-ni aucc-ni )
  (if (II (! pred-ni)
      (! aucc-nl)
      (al:coapacted? pred-ni)
      (al:coapacted? succ-ai)
      (== 'trace-fence (aioperator pred-al) ) )
  (then
    nil)
  (else
    (caseq »tr.trace-picker*
      (normal
        (ftft (>= (nr.edge-prob pred-ni aucc-nl) .6)
          (then
            (:= max-expect (ni:expect pred-ni) )
            (:s max-elt-so-far ni) ) )
      (bb
        (&& (! (ni:cond-jump? pred-ni) )
          (! (ni:rejoin? succ-ni) ) )
        (liberal
          (&& (= succ-ni
          "PS: <C.S. BULLDOG TRACE>TRACE-PICKER.LSP.1"
```

```lisp
(defun tp.ml-list:max-expect-ml (ml:ml-list) )
  (loop (for ni in ml:ml-list)
    (initial max-expect -1
    max-elt-so-far ()
    (do
      (if (&& (! (nl:conpacted? nl) )
        (> (al:expect al) aax-expect) )
        (then
          (:= aax-expect (al:expect al) )
          (:s aax-elt-so-far ni) ) )
    (result aax-elt-so-far) )
  )
)
```

```lisp
(defun tp.ml-list:max-expect-ml (ml:ml-list) )
  (loop (for ni in ml:ml-list)
    (initial max-expect -1
    max-elt-so-far ()
    (do
      (if (&& (! (ni:cond-jump? pred-ni) )
        (! (ni:rejoin? succ-ni) )
        (liberal
          (&& (= succ-ni
```