This module implements the bank disambiguation. It assumes that the assertions have already been processed and that loop assignments have already been inserted for derivations to work ok.

Each vector reference is examined in turn. The residue of the derivation of the index is calculated, and if it is non-nil (known) the NADDR source operation is destructively modified to include the bank.

Warning: If we ever change this to do non-destructive modification, then remember to recalculate the hash table used in the disambiguator.

(include flow-analysis:flow-analysis-decls)

(defun fg.disambiguate-banks ()
  (loop (for-each-stat stat)
    (when (stat:property? stat 'vector-reference)
      (initial heading-printed? ()
        (do
          (if-let ( (bank (de:resldue (stat:index-derivation stat)
            'number-of-banks* stat) )
            (then
              (:= (cdr (nth (stat:source stat) 4) )
                '(,bank) )
            )
          (else (if *fg.show-unknown-bank-references?* (then
            (if (! heading-printed?) (then
              (msg 0 t "Vector references to unknown banks:" t)
              (:= heading-printed? t) )
            (msg t (a stat) t ) ) ) ))
          )
        ) ) )
  ) ) )
Every basic block in the flow graph is represented by a BBLOCK. All BBLOCKS are numbered and stored in a global array that maps the numbers onto the corresponding BBLOCKS.

```lisp
(def-struct bblock
  nuaber
  dfo-nunber
  first-stat
  last-stat
  (preds)
  (succs)
  (gen)
  (kill)
  (reaching-in)
  (reaching-out)
  (reaching-copies-in)
  (reaching-copies-out)
  (dominators)
  (live-in)
  (live-out)
)
```

- **Number of this block.**
- **Depth-first-order number of this block.**
- **First statement in this block.**
- **Last statement.**
- **List of predecessor blocks.**
- **List of successor blocks.**
- **STAT-SET used for flow analysis algorithms.**
- **STAT-SET used for flow analysis algorithms.**
- **STAT-SET of definitions reaching this block on entry and exit.**
- **STAT-SET of definitions copy-reaching this block on entry and exit.**
- **BBLOCK-SET of dominators of this block.**
- **NAME-SET of names live on entry and exit from this block.**

### BBLOCK Functions

- **(BBLOCK:CREATE)**
  - Creates a new BBLOCK and records it in the array of BBLOCKS. This is the only way a BBLOCK should be created.
  - **(BBLOCK:DELETE BBLOCK)**
    - Unsplices BBLOCK from the flow graph, forgetting it and all of its STATS. A BBLOCK may be deleted only if it satisfies the conditions of BBLOCK:UNSPICE.
  - **(BBLOCK:SPICE BBLOCK NEW-BBLOCK PREDs)**
    - Splices NEW-BBLOCK between BBLOCK and each block in PREDs.
    - PREDs is assumed to be a subset of the preds of BBLOCK.
  - **(BBLOCK:UNSPICE BBLOCK)**
    - Unsplices BBLOCK from the flow graph. A BBLOCK may be unspliced only if it has 1 successor and 0, 1, or 2 predecessors; or 2 successors and 0 or 1 predecessor.
  - **(BBLOCK:APPEND-STAT BBLOCK STAT)**
    - Appends STAT to the end of BBLOCK.
  - **(BBLOCK:DELETE-STATS BBLOCK)**
    - Deletes all of the STATS in the BBLOCK.
  - **(BBLOCK:MERGE-WITH-SUCCESSOR BBLOCK)**
    - Tries to merge BBLOCK with its successor. The merge can take place only if BBLOCK has one successor, and that successor has only one predecessor. If they are mergeable, the successor BBLOCK is deleted and true is returned; otherwise nothing happens and false is returned.

- **(BBLOCK:DOMINATES? BBLOCK BBLOCK1)**
  - Returns true if BBLOCK dominates BBLOCK1.
- **(BBLOCK:EMPTY? BBLOCK)**
  - Returns true if BBLOCK has no STATS in it.
- **(NUMBER:BBLOCK NUMBER)**
  - Returns the BBLOCK of a given number.

### Easy, interactive syntax for (NUMBER:BBLOCK <number>)

- **#SB <number>**

### Flow Analysis

- **(loop (for-each-bblock bblock) ...)**
  - This is the only public way for enumerating through all the BBLOCKS of the flow graph. The enumeration is in the original order of the NADDR source. It is defined via LOOP'S DEF-SIMPLE-LOOP-CLAUSE.
- **(loop (for-each-bblock-stat bblock stat) ...)**
  - This is the recommended way for enumerating through all the STAT's of a BBLOCK. The enumeration goes from first stat to last.
- **(loop (for-each-bblock-stat-reverse bbblock stat) ...)**
  - Enumerates through the STATs of a BBLOCK in reverse order.

### Algebraic Properties

1. PS:<C.S.BULLDOG.FLOW-ANALYSIS>BBLOCK.LSP.15
;;; Remove BBLOCK from the flow graph.
bblock:unsplice bblock

(defun bblock:delete-stat ( bblock )
  (assert (bblock:is bblock) )
  (loop (initial stat (bblock:first-stat bblock) succ-stat () )
    (while stat)
      (next succ-stat (stat:succ stat) )
      (do (stat:delete stat) )
    (next stat succ-stat) )

  (= (bblock:first-stat bblock) () )
  (= (bblock:last-stat bblock) () )

(defun bblock:splice ( bblock new-bblock preds )
  (assert (bblock:is bblock) )
  (assert (bblock:is new-bblock) )

  ;; Make each predecessor point at NEW-BBLOCK, and remove
  ;; that predecessor from the preds of BBLOCK.

  (for (pred-bblock in preds) (do
    (:= (bblock:preds bblock)
        (unionq (top-level-removq pred-bblock bblock new-bblock) ) )
  )

  ;; Add NEW-BBLOCK to the preds of BBLOCK

  (push (bblock:preds bblock) new-bblock)

  ;; The preds of NEW-BBLOCK are PREDs, and the succs are
  ;; just BBLOCK.

  (:= (bblock:succs new-bblock) (list bblock) )

(defun bblock:append-stat ( bblock stat )
  (assert (bblock:is bblock) )
  (assert (stat:is stat) )

  (= (stat:bblock stat) bblock)

  (if (! (bblock:first-stat bblock) ) (then
    (:= (bblock:first-stat bblock) stat)
    (:= (bblock:last-stat bblock) stat) )
  (else
    (= (stat:pred stat) (bblock:last-stat bblock) )
    (= (stat:succ (bblock:last-stat bblock) ) stat)
    (= (bblock:last-stat bblock) stat) )

(defun bblock:merge-with-successor ( bblock )
  (assert (bblock:is bblock) )

  (if (= 1 (length (bblock:succ bblock) ) )
    (= (bblock:succ bblock) (list bblock) )
    (= (bblock:succ bblock) (list bblock (car (bblock:succ bblock) ) ) )
  )
(then
  (let* ((succ-bblock (car (bblock:succs bblock)))
         (succ-bblock-stats
          (loop (for-each-bblock-stat succ-bblock stat) (save stat)))
         (loop (for stat in succ-bblock-stats) (do
             (stat:extract stat)
             (bblock:append-stat bblock stat)))
         (bblock:delete succ-bblock))
       (else t))
)

(defun bblock:dominates? (bblock bblockl)
  (assert (bblock:is bblock))
  (assert (bblock:is bblockl))
  (bblock-set:member? (bblockNominators bblockl) bblock))

(defun bblock:empty? (bblock)
  (assert (bblock:is bblock))
  (assert (! (bblock:first-stat bblock))
  (i (bblock:last-stat bblock)))

(defmacro number:bblock (number)
  '([number.bblock . number]))

(def-sharp-sharp b
  '([bblock . (read)]))

(def-sinple-loop-clause for-each-bblock (clause)
  (let ( (for-each-bblock var) clause)
    (index (Intern (gensya)))
    (lf (! (ftft (= 2 (length clause))
          (lltatoa var))
    (error (list clause "Invalid FOR-EACH-BBLOCK syntax."))
    • (initial . var ()
      (incr . index from 0 to (• -1 »fg.total-bblocks»)
      (next . var (• fg.nuaber:bblock . index)
        (when . var)))
  )
)

eval-when (eval compile load)
(def-sinple-loop-clause for-each-bblock-stat (clause)
  (let ( (for-each-bblock-stat bblock stat-var) clause)
    (bblock-var (Intern (gensyn))
    (lf (! («ft (= 3 (length clause))
          (lltatoa stat-var))
    (error (list clause "Invalid FOR-EACH-BBLOCK-STAT'- syntax."))
    • (initial ,bblock-var . bblock
      ,stat-var 0)
      (next ,stat-var
        (lf (! ,stat-var)
          (bblock:last-stat ,bblock-var)
          (stat:pred ,stat-var))
      (while ,stat-var))
  )
)

(defun bblock:print (bblock bblock-fields)
  (assert (bblock:is bblock))
  (asg 0 (bblock:nuaber bblock) ": succs: "
  (for (succ-bblock In (bblock:succs bblock)) (save
    (bblock:nuaber succ-bblock))
    " preds: ")
  (for (pred-bblock in (bblock:preds bblock)) (save
    (bblock:nuaber pred-bblock))
    t)

  (loop (for field in bblock-fields) (do
    (caseq field
      (gen (asg " Gen: 
    (e (stat-set:print (bblock:gen bblock))
    t))
      (kill (asg " Kill: "
    (e (stat-set:print (bblock:kill bblock))
    t))
      (reaching-in (asg " Reaching-in: "
    (e (stat-set:print (bblock:reaching-in bblock))
    t))
      (reaching-out (asg " Reaching-out: "
    (e (stat-set:print (bblock:reaching-out bblock))
    t))
      (reaching-copies-in (asg " Reaching-copies-in: "
    (e (stat-set:print (bblock:reaching-copies-in bblock))
    t))
      (reaching-copies-out (asg " Reaching-copies-out: "
    (e (stat-set:print (bblock:reaching-copies-out bblock))
    t))
      (doinator (asg " Doinators: "
    (e (bblock-set:print (bblock:doinators bblock))
    t))
  )
)
(live-in
  (msg "Live-in:"
    (e (name-set:print (bblock:live-in bblock))
      t))
)
(live-out
  (msg "Live-out:"
    (e (name-set:print (bblock:live-out bblock))
      t))
)
)
(loop (for-each-bblock-stat bblock stat) (do
  (msg " " (J (stat:nunber stat) 3) " • (stat:source stat)"
    (if (neaq 'reachlng-uses bblock-fields) (then
      (msg t)
      (for (use-stat In (stat:reachlng-uses stat) ) (save
        (stat:nuaber use-stat) ) )
    (msg t)))
  )
)"
Sets of BBLOCKS are represented using BBLOCK-SETS, currently implemented as BIT-SETs.

- **FG.EMPTY-BBLOCK-SET**
  - The empty BBLOCK-SET.
- **(BBLOCK-SET:UNIVERSE)**
  - Set of all BBLOCKS.
- **(BBLOCK-SET:SINGLETON BBLOCK)**
  - Creates a new set containing BBLOCK.
  - Returns true if BBLOCK is a member of SET.
- **(BBLOCK-SET:INTERSECTION SET1 SET2 ...)**
  - Returns a new set that is the intersection of all the given sets.
- **(BBLOCK-SET:UNION SET1 SET2 ...)**
  - Returns a new set that is the union of all the given sets.
- **(BBLOCK-SET:UNION1 SET BBLOCK)**
  - Unions a single BBLOCK into SET.
- **(BBLOCK-SET:DIFEREENCE SET1 SET2)**
  - Returns a new set that contains all elements in SET1 not in SET2.
- **(BBLOCK-SET:= SET1 SET2)**
  - Returns true if the two sets are equal.
- **(BBLOCK-SET:SIZE SET)**
  - Returns the number of elements in the set.

**DEF-SIMPLE-LOOP-CLAUSE**

- (for-each-bblock-set-element (clause))
  - Enumerates BBLOCK through each element in SET. Uses DEF-SIMPLE-LOOP-CLAUSE.

**BBLOCK-SET:PRINT SET**

- Prints SET by printing out the statement numbers.
(build *flow.build-module-list*)

(:= *flow.build-module-list* '(flow-analysis:stat
flow-analysis:stat-set
flow-analysis:cblock
flow-analysis:cblock-set
flow-analysis:naae
flow-analysis:naae-set
flow-analysis:loop
flow-analysis:naddr-to-flow-graph
flow-analysis:flow-graph-to-naddr
flow-analysis:temporary-name
flow-analysis:depth-first-order
flow-analysis:reaching-defs
flow-analysis:reaching-uses
flow-analysis:reaching-copies
flow-analysis:live-names
flow-analysis:dominators
flow-analysis:find-loops
flow-analysis:loop-invariant-motion
flow-analysis:induction-variable-removal
flow-analysis:cse-node
flow-analysis:cse-hash-tables
flow-analysis:common-subexpression-elimination
flow-analysis:copy-propagation
flow-analysis:variable-renaming
flow-analysis:derivations
flow-analysis:disambiguator
flow-analysis:disambiguator-tool
flow-analysis:bank-disambiguator
flow-analysis:statistics
flow-analysis:dependencies
flow-analysis:flow-analysis-options
flow-analysis:flow-analysis)

(:= *build-module-list* (append *build-module-list* *flow.build-module-list*))
(include flow-analysis:flow-analysis-decls)

(defun fg.collect-names ()
  (fg.initialize-names)
  (loop (for-each-stat stat) (do
    (caseq (stat:operator stat)
      (def-block
        (for (name in (stat:part stat 'in-variables)) (do
          (if (litatom name) (then
            (name:create name 'scalar 0)
            (name:add-defining-stat name stat)))
        ))
      (del
        (push *fg.all-vector-names* (stat:part stat 'variable))
        (name:create (stat:part stat 'variable)
          'vector
          (stat:part stat 'length)))
      (t
        (let ((defined-name (stat:part stat 'written)))
          (if defined-name (then
            (name:create defined-name 'scalar 0)
            (name:add-defining-stat defined-name stat))))))

  (loop (for-each-stat stat) (do
    (loop (for-each-stat-operand-read stat read-name) (do
      (name:add-using-stat read-name stat)))))

  ()
  )
COMMON SUBEXPRESSION ELIMINATION

This module implements basic block common subexpression elimination. Currently, a VSTORE into a vector kills all previous values loaded from that vector.

(FA.INITIALIZE-CSE)
Initializes this module for the current flow graph.

(FA.ELIMINATE-COMMON-SUBEXPRESSIONS)
Eliminates the common subexpressions in each BBLOCK of the flow graph, destroying the old STATs of the BBLOCKS and replacing them with new STATs. This invalidates all the flow information except the live variable info.

(CSE.TEST OPER-LIST LIVE-LIST)
A testing function that takes OPER-LIST, a list of NADDR, converts it into a BBLOCK, eliminates the CSEs in the BBLOCK, and then prints the new BBLOCK. LIVE-LIST is a list of names that are live on exit from the block.

(CSE.PRINT)
Prints out the current DAG of CSE-NODEs.

(eval-when (compile load)
  (include flow-analysis:flow-analysis-decls)
)

(eval-when (compile)
  (build '(flow-analysis:cse-node))
)

(declare (special
  +cse.all-cse-nodes+)
)

(defun fa.initialize-cse ()
  (cse.initialize)
  ()
)

(defun fa.eliminate-common-subexpressions ()
  (fa.initialize-cse)
  (loop (for-each-bblock bblock) (do
    (bblock:eliminate-common-subexpressions bblock)
    ()
  ))
)

(defun cse.test ( oper-list live-list )
  (fa.initialize)
  (let ((bblock (bblock:create)))
    (loop (for name in live-list) (do
      (name:create name 'scalar 0)
      (:= (bblock:live-out bblock)
          (name:set:unioni (bblock:live-out bblock)
          ))
    ))
    (loop (for oper in oper-list) (do
      (loop (for name in (oper:part oper 'read)) (do
        (name:create name 'scalar 0)
        (if (oper:part oper 'written)
          (name:create (oper:part oper 'written) 'scalar 0)
          (bblock:append-stat bblock (stat:create oper))
        )
      ))
      (bblock:eliminate-common-subexpressions bblock)
      (bblock:print bblock)
    ))
  )
)

(defun cse.initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)

(defun cse.process-oper (oper)
  (cse.process-oper (stat:source oper))
)

(defun bblock:eliminate-common-subexpressions ( bblock )
  (cse.initialize)
  (loop (for-each-bblock-stat bblock stat) (do
    (cse.process-oper (stat:source stat))
  ))
  (cse.remove-useless-labels (bblock:live-out bblock))
  (cse.make-copy-nodes-for-initial-nodes)
  (cse.assign-names-and-add-pseudo-edges)
  (cse.set-remaining-counts)
  (bblock:delete-stat bblock)
  (loop (for oper in (cse.generate-code)) (do
    (bblock:append-stat bblock (stat:create oper))
  ))
  (if (bblock:empty? bblock) (then
    (bblock:delete bblock)
  )
  )
)

(defun bblock:initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)

(defun cse.process-oper (oper)
  (if (cse.is-oper-external-operand oper)
    (cse.process-oper (stat:source oper))
    (cse.process-oper (stat:source oper))
  )
)

(defun cse.initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)

(defun cse.process-oper (oper)
  (if (cse.is-oper-external-operand oper)
    (cse.process-oper (stat:source oper))
    (cse.process-oper (stat:source oper))
  )
)

(defun bblock:initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)

(defun cse.process-oper (oper)
  (if (cse.is-oper-external-operand oper)
    (cse.process-oper (stat:source oper))
    (cse.process-oper (stat:source oper))
  )
)

(defun bblock:initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)

(defun cse.process-oper (oper)
  (if (cse.is-oper-external-operand oper)
    (cse.process-oper (stat:source oper))
    (cse.process-oper (stat:source oper))
  )
)

(defun bblock:initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)

(defun cse.process-oper (oper)
  (if (cse.is-oper-external-operand oper)
    (cse.process-oper (stat:source oper))
    (cse.process-oper (stat:source oper))
  )
)

(defun bblock:initialize ()
  (cse.initialize-cse-nodes)
  (cse.initialize-hash-tables)
)
(defun cse.process-oper (oper)
  (caseq (oper:group oper)
    (vload (cse.process-vload oper))
    (vstore (cse.process-vstore oper))
    (if-compare (cse.process-n-in-n-out oper))
    (one-in-one-out (cse.process-n-in-n-out oper))
    (two-in-one-out (cse.process-n-in-n-out oper))
    (three-in-one-out (cse.process-n-in-n-out oper))
    (assert (cse.process-n-in-n-out oper))
    (loop-assign (cse.process-n-in-n-out oper))
    (vbase (cse.process-n-in-n-out oper)))
  (t (cse.process-miscellaneous oper))))

(defun cse.process-vload (oper)
  (let* ((children (opert:part oper 'vector))
        (expr-key (opert:operator oper))
        (cse-node (cse.node:create-interior oper children))
        (cse.expr:define-cse-node expr-key cse-node))
    (if (opert:part oper 'written)
      (cse.node:assign-to-name cse-node (opert:part oper 'written)))
    (if (! cse-node)
      (cse.node:create-interior oper children))
    (cse.node:assign-to-name cse-node (opert:part oper 'written)))
)

(defun cse.process-assn (oper)
  (let* ((expr-key (opert:operator oper))
        (cse-node (cse.node:create-interior oper (cse.node:part oper (opert:readi))))
        (cse.node:assign-to-name cse-node (opert:part oper 'written)))
    (cse.node:assign-to-name cse-node (opert:part oper 'written)))
)

(defun cse.process-miscellaneous (oper)
  (let* ((cse-node (cse.node:create-miscellaneous oper)))
    (:= (cse.node:miscellaneous? cse-node) t))
)

(defun cse.process-n-in-n-out (oper)
  (let* ((children (if (= 'vbase (opert:operator oper))
                        (cse.node:cse-node (opert:part oper 'vector))
                        (for (name in (opert:part oper 'readi) (save
                                      (cse.node:cse-node name)))
                          (expr-key (opert:operator oper))
                          (cse.node:cse-node expr-key))))
        (if (! cse-node)
          (cse.node:create-miscellaneous oper children))
        (cse.node:assign-to-name cse-node (opert:part oper 'written)))
    (if (opert:part oper 'written)
      (cse.node:assign-to-name cse-node (opert:part oper 'written)))
    )
)

(defun cse.process-vload (oper)
  (let* ((vector-cse-node (cse.node:cse-node (opert:part oper 'vector)))
        (expr-key (opert:operator oper)))
    (if (cse.node:datatype vector-cse-node (opert:dest-datatype oper))
      (if (! cse-node)
        (cse.node:create-miscellaneous oper children))
      (cse.node:assign-to-name cse-node (opert:part oper 'written)))
    (if (opert:part oper 'written)
      (cse.node:assign-to-name cse-node (opert:part oper 'written)))
    )
)
(defun cse-process-vstore (oper)
  ;; Adds a CSE-NODE representing operation OPER to the DAG. OPER should be a VSTORE.
  ;; A new CSE-NODE is always created for a VSTORE. All previous VLOADs and VSTOREs into the same vector are killed. The new CSE-NODE is added to the list of VLOADs and VSTOREs stored in the initial node for the vector. Note the trickiness of hashing the new CSE-NODE under VLOAD instead of VSTORE -- this lets future VLOADs find this VSTORE as a cse.
  ;; We can make VSTOREs smarter by having them kill only those previous VLOADs and VSTOREs that could possibly have the same index as this VSTORE. We would call the disambiguator oracle for this info.
  (let* ((vector-cse-node (cse-node:make-copy-cse-nodes oper))
         (index-cse-node (cse-node:make-copy-cse-nodes oper))
         (children ((vector-cse-node :read2) (cse-node:make-copy-cse-nodes oper))
         (expr-key (cse-node:make-copy-cse-nodes oper))
         (cse-node (cse-node:create-interior oper children))
         (expr-key (cse-node:make-copy-cse-nodes oper))
         (cse-node (cse-node:assign-to-name oper))
         (cse-node (cse-node:assign-to-name oper))
         (cse-node (cse-node:assign-to-name oper))
         (cse-node (cse-node:assign-to-name oper)))
    (loop (for vload-vstore-cse-node in (cse-node:vloads-vstores vector-cse-node) )
      (when (! (cse-node:killed? vload-vstore-cse-node) )
        (do
          (cse-node:add-pseudo-child cse-node vload-vstore-cse-node)
          (cse-node:assign-to-name oper)
          (push (cse-node:vloads-vstores vector-cse-node) cse-node)
          (cse-node:assign-to-name oper))))
    (t)
    (cse-node:assign-to-name oper))
  )
)

(*=====================================================================
  (defun cse-remove-useless-labels (live-out)
    (loop (for cse-node in (cse-all-cse-nodes) )
      (do
        (loop (for name in (cse-node:labels cse-node) )
          (initial new-labels 0)
          (do
            (if (ftft (name-set:member? live-out name) )
              (then
                (push new-labels name) )
            ) )
          (result
            (:= (cse-node:labels cse-node) new-labels) )
          )
        )
      )
    ) )
  )

  (defun cse.make-copy-nodes-for-initial-nodes ()
    (loop (for cse-node in (cse-all-cse-nodes) )
      (do
        (if (cse-node:initial? cse-node)
          (cse-node:make-copy-cse-nodes cse-node) )
        )
      )
    )
  )

  (defun cse.assign-names-and-add-pseudo-edges ()
    (loop (for cse-node in (cse-all-cse-nodes) )
      (do
        (if (cse-node:initial? cse-node)
          (cse-node:make-copy-cse-nodes cse-node) )
        )
      )
    )
  )

  PS:<C.S.BULLDOG.FLOW-ANALYSIS.TEST>COMMON-SUBEXPRESSION-ELIMINATION.LSP.1
ASSIGN nodes are created for all other names that label the node; each ASSIGN assigns its name with the value of this node. Pseudo edges are added between each new ASSIGN node and the uses of the initial node of its name (as in step 2).

(defun cse.assign-names-and-add-pseudo-edges ()
  (loop (initial rest-cse-nodes *cse.all-c3e-nodes* name () )
    (while rest-cse-nodes)
    (bind cse-node (pop rest-cse-nodes)
      oper (cse-node:source cse-node) )
    (when (II (cse-node:initial? cse-node)
      (cse-node:initial-nane cse-node) )
    (for inltlal-use-cse-node in (cse.name:initial-uses name) )
    (do
      (:= all-inltlal-uses-ok? (cse-node:ancestor? inltlal-use-cse-node cse-node) )
    )
    (if all-inltlal-uses-ok?
      (return name )
    )
    (result
      (cse-node:source cse-node)
      (fa:temporary-name (oper:part (cse-node:source cse-node) written) )
      (fa:temporary-name () )
    )
    )
  )
)

(cse-node:pick-a-good-nane ( cse-node )
(assert (cse-node:is cse-node) )
(? (cse-node:initial? cse-node)
  (cse-node:initial-name cse-node) )
(oper:part (cse-node:source cse-node) 'written)
(loop (for name in (cse-node:labels cse-node) )
  )
)

(cse-node:add-initial-name-edges ( cse-node )
  (if (! (cse-node:initial? cse-node))
    (loop (for inltlal-use-cse-node in (cse.name:initial-uses name) )
      (when (II initial-use-cse-node cse-node )
        (do
          (cse-node:add-pseudo-child cse-node initial-use-cse-node )
        )
      )
    )
  )
)

(cse.set-remaining-counts ()
  (loop (for cse-node in *cse.all-c3e-nodes* )
    (do (cse-node:remaining-count cse-node)
      (+ (length (cse-node:children cse-node) )
        (length (cse-node:pseudo-children cse-node) )
      )
    )
  )
)

(defun cse.node:pick-a-good-nane ( cse-node )
(cse-node:pick-a-good-name cse-node)
)

(defun cse-node:add-initial-name-edges ( cse-node )
(cse-node:add-initial-name-edges cse-node)
)

(defun cse.set-remaining-counts ()
  (loop (for cse-node in *cse.all-c3e-nodes* )
    (do (cse-node:remaining-count cse-node)
      (+ (length (cse-node:children cse-node) )
        (length (cse-node:pseudo-children cse-node) )
      )
    )
  )
)

(defun cse.set-remaining-counts ()
  (loop (for cse-node in *cse.all-c3e-nodes* )
    (do (cse-node:remaining-count cse-node)
      (+ (length (cse-node:children cse-node) )
        (length (cse-node:pseudo-children cse-node) )
      )
    )
  )
)

(defun cse.assign-names-and-add-pseudo-edges ()
  (loop (initial rest-cse-nodes *cse.all-c3e-nodes* name () )
    (while rest-cse-nodes)
    (bind cse-node (pop rest-cse-nodes)
      oper (cse-node:source cse-node) )
    (when (II (cse-node:initial? cse-node)
      (cse-node:initial-nane cse-node) )
    (for inltlal-use-cse-node in (cse.name:initial-uses name) )
    (do
      (:= all-inltlal-uses-ok? (cse-node:ancestor? inltlal-use-cse-node cse-node) )
    )
    (if all-inltlal-uses-ok?
      (return name )
    )
    (result
      (cse-node:source cse-node)
      (fa:temporary-name (oper:part (cse-node:source cse-node) written) )
      (fa:temporary-name () )
    )
    )
  )
)

(cse-node:pick-a-good-nane ( cse-node )
(assert (cse-node:is cse-node) )
(? (cse-node:initial? cse-node)
  (cse-node:initial-nane cse-node) )
(oper:part (cse-node:source cse-node) 'written)
(loop (for name in (cse-node:labels cse-node) )
  )
)

(cse-node:add-initial-name-edges ( cse-node )
  (if (! (cse-node:initial? cse-node) )
    (loop (for inltlal-use-cse-node in (cse.name:initial-uses name) )
      (when (II initial-use-cse-node cse-node )
        (do
          (cse-node:add-pseudo-child cse-node initial-use-cse-node )
        )
      )
    )
  )
)

(cse.set-remaining-counts ()
  (loop (for cse-node in *cse.all-c3e-nodes* )
    (do (cse-node:remaining-count cse-node)
      (+ (length (cse-node:children cse-node) )
        (length (cse-node:pseudo-children cse-node) )
      )
    )
  )
)
Traverses the DAG in topological order (from the leaves up), converting DAG nodes back into NADDR operations. A list of NADDR is returned.

There is one slightly tricky thing going on here — *CSE.ALL-CSE-NODES* is in reverse order of creation. When we enumerate through looking for all leaf nodes, a list of those leaf nodes is created (reversed again) containing those nodes in original source order. Those nodes are processed in depth-first manner. This all guarantees that miscellaneous pseudo-ops at the beginning or end of block maintain their relative position. Neat how that falls out.

But it is slightly more tricky than that. Induction variable removal works best when the induction statements \( I := I +/-/C \) are evaluated as late as position in the list as use the older value of \( I \). So when we are picking the next node off the to-do stack, if the first node on the stack is a possible induction statement, we ignore it and look deeper into the stack for a non-induction node.

**CSE-NODE:POSSIBLE-INDUCTION? CSE-NODE**

Returns true if CSE-NODE describes an operation of the form:

\[ I := I +/-/C \text{ or } I := C +/-/I \]

for any \( C \).

**CSE-NODE:GENERATE CSE-NODE**

Construct a NADDR operation for CSE-NODE, substituting in the name of this node and its children for the old variable names.
(let ( (oper (cse-node:source cse-node) ) )
  (if (== 'vbase (oper:operator oper) )
    (vbase . (cse-node:name cse-node) ,(oper:part oper 'vector) )
    (cse-node:children cse-node)
    (:= oper
      (oper:substitute-for-part oper
        (loop (for child-cse-node in (cse-node:children cse-node) )
          (save (cse-node:name child-cse-node) )
        )
      )
    )
  )
)

(defvar cse-print ()
  (loop (for cse-node In *cse.all-cse-nodes*) (do
    (nag 0 (cse-node:number cse-node) "t")
    (if (cse-node:name cse-node) (msg "name: " (cse-node:name cse-node) t))
    (if (! (cse-node:initial-name cse-node) )
      (msg "initial-name: " (cse-node:initial-name cse-node) t))
    (if (cse-node:labels cse-node)
      (msg "labels: " (cse-node:labels cse-node) t))
    (if (cse-node:source cse-node)
      (msg "source: " (cse-node:source cse-node) t))
    (if (cse-node:parents cse-node)
      (msg "parents: " (cse-node-list:numbers (cse-node:parents cse-node) ) t))
    (if (cse-node:children cse-node)
      (msg "children: " (cse-node-list:numbers (cse-node:children cse-node) ) t))
    (if (cse-node:pseudo-children cse-node)
      (msg "p-children: " (cse-node-list:numbers (cse-node:pseudo-children cse-node) ) t))
    (if (cse-node:remaining-count cse-node)
      (msg "remaining-count: " (cse-node:remaining-count cse-node) t))
    (if (! (bit-set= «bit-set.empty-set* (cse-node:descendants cse-node) ) )
      (msg "descendants: " (cse-node-list:print (cse-node:descendants cse-node) ) t))
  )
)

(defvar cse-node-list:numbers ( list )
  (for (cse-node in list) (save (cse-node:number cse-node) )
    (msg "number: " (cse-node:number cse-node) t))
)

(defvar cse-node-list:numbers ( list )
  (for (cse-node in list) (save (cse-node:number cse-node) )
    (msg "number: " (cse-node:number cse-node) t))
)
Does constant folding, using an iterative algorithm similar to that in the Dragon Book. Given a constant assignment \( S: A := C \) (where \( C \) is a constant number), we substitute \( C \) for \( A \) in all uses of \( S \) that have \( S \) as the only reaching definition of \( A \). If the use now has all constant operands, we evaluate the right hand side and replace it by an assignment statement. We keep iterating over the flow graph until we can't do any more substitutions.

Constant folding also includes the following transformations (for both integer and real arithmetic):

- \( A \{+ / \text{AND} \} 1 \Rightarrow A \)
- \( A \{+ / \text{OR} \} 0 \Rightarrow A \)
- \( 0 \{+ / \text{AND} \} A \Rightarrow 0 \)
- \( A / A \Rightarrow 1 \)
- \( A - A \Rightarrow 0 \)

Eventually, we'll want to do constant folding on conditional jumps also.

Intuitively, it is faster to keep iterating over the flow graph in depth-first order, rather than using a statement-propagation algorithm. The reason is that enumerating in depth-first order causes \( \text{STAT:REACHING-DEFS} \) to be evaluated very efficiently (since it is calculated on the fly from the basic block reaching defs). Whereas a statement-propagation algorithm, keeping a stack of statements that have changed, evaluates \( \text{STAT:REACHING-DEFS} \) in random order, causing essentially \( N^2 \) behaviour with lots of consing.

---

```
(defun fg.fold-constants ()
  (loop (initial change? t)
    (while change?)
      (incr pass from 1)
      (do (:= change? () )
        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
          (do (:= (stat:source stat)
            (oper: substitute-operand 1 read )
            (if (|| any-substitutions?)
              (any-substitutions? t))))
          (if (cf.stat:simplify stat) (then
            (:= change? t) ) ) ) ) )
)
```

---

```
(constant operand 'read)

(IF (|| any-substitutions?)
  (any-substitutions? t)
  (if (cf.stat:simplify stat)
    (then
      (:= change? t) ) )
)
```

---

```
(defun cf.stat:simplify (stat)
  (let* ((operator (stat:operator stat) )
    (read (stat:part stat 'read) )
    (written (stat:part stat 'written) )
    (Identity (cf.operator:Identity operator) )
    (? ( (ftft (== 'assign operator)
      (numberp (stat:part stat 'readl))
    ) t)
      ( (ftft (neq (stat:group stat) '(one-in-one-out two-in-one-out) )
        (for-every (operand in read)
          (numberp operand) )
      )
        (:= (stat:source stat)
          'assign .written .apply (operator:execute-function operator) read )
    ) t)
      ( (ftft (neq operator '(inul fnul land) )
        (member 0 read) )
        (:= (stat:source stat)
          'assign .written 0 )
    ) t)
      ( (ftft (neq operator '(ldlv fdlv isub fsub) )
        (== (car read) (cadr read) )
      )
        (:= (stat:source stat)
          'assign .written .identity )
    ) t)
      ( (ftft (neq operator '(inul faul ladd fadd land lor) )
        (aeaber identity read) )
        (:= (stat:source stat)
          'assign .written .,(top-level-remove Identity read) )
    ) 0)
      ( (ftft (neq operator '(ldlv fdlv isub fsub) )
        (== (car read) (cadr read) )
      )
        (:= (stat:source stat)
          'assign .written .identity )
    ) t)
      ( (ftft (neq operator '(inul faul ladd fadd land lor) )
        (aeaber identity read) )
        (:= (stat:source stat)
          'assign .written .,(top-level-remove Identity read) )
    ) 0)
    )
)
```
(defun cf.operator:Identity (operator)
  (caseq operator
    ((inul ldlv land) 1)
    ((faul fdlv) 1.0)
    ((ladd isub lor) 0)
    ((fadd fsub) 0.0)
    (t () )))

;;==================================================================================
;; (CF.OPERATOR:IDENTITY OPERATOR)
;; ***
;; Returns the identity constant for the given operator.
;;==================================================================================
Propagates copies produced by assignment statements through the flow graph. For each statement \( S: A := B \), all the uses of \( S \) are examined:

- If \( S \) copy reaches a use, then \( B \) can be substituted for \( A \) in the use.

The assignment statements are processed in depth-first order so that copies are propagated through chains of assignments.

This algorithm differs slightly from that of the Dragon Book. The Dragon Book requires that every assignment copy-reaches every use before we substitute in any of the uses; thus, after substitution, the assignment can be deleted since it is useless.

This algorithm substitutes into each use of the assignment independently of whether it copy-reaches every use; we count on dead-code removal to remove any useless assignments so created. Doing the substitution wherever possible can potentially reduce the depth of the data precedence graph and eliminate needless write-after-conditional-read conflicts, even if we can't substitute an assignment into ALL of its uses.

```
(eval-when (compile load)
  (include flow-analysis:flow-analysis-decls)
)

(defun fg.propagate-copies ()
  (let ( (assign-stats 0) )
    ;; Set ASSIGN-STATS to be all assignments. The list is
    ;; created in depth-first order, so that we can propagate
    ;; copies through long chains of assignments.
    (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward))
      (do
        (loop (for-each-bblock-stat bblock stat)
          (when (== 'assign (stat:operator stat))
            (do
              (push assign-stats stat))))
      )
    (= assign-stats (dreverse assign-stats))
    ;; For each assignment \( A := B \), substitute \( B \) into each use
    ;; of the assignment that has it as a reaching copy. We
    ;; destructively change STAT:SOURCE instead of creating a
    ;; new STAT so that we preserve the reaching-uses of the
    ;; changed STAT; otherwise, we couldn't propagate through
    ;; chains of assignments.
    (loop (for assign-stat in assign-stats) (do
      (loop (for use-stat in (stat:reaching-uses assign-stat))
        (when (== 'live (stat:operator use-stat))
          (when (stat-set:member? (stat:reaching-copies use-stat) assign-stat)
            (do
              (stat:source use-stat)
              (oper:substitute-operand (stat:source use-stat)
                (stat:part assign-stat 'read)
                (stat:part assign-stat 'written)
                'read))))))
  )
)
```

PS: <C.S.BULLDOG.FLOW-ANALYSIS>COPY-PROPAGATION.LSP.1
CSE-HASH-TABLES

This module implements the hash tables used by basic block common
subexpression elimination. There are three such tables:

one for mapping operators and operands onto previously found
CSE-NODEs with the same operator and operands;

one that maps a NAME onto the CSE-NODEs currently holding the value
of the NAME;

one that maps a NAME onto the initial (leaf) CSE-NODE whose
:INITIAL-NAME is NAME (the CSE-NODE that represents the NAME's value
on entry into the basic block).

(CSE.INITIALIZE-HASH-TABLES)

Initializes the hash tables.

(CSE.EXPR:CSE-NODE KEY)

Maps KEY, a list of the form (OPERATOR OPERAND1 OPERAND2) onto the
CSE-NODE in the CSE DAG that has that operator and operands. Returns
0 if there is no such node.

(CSE.EXPR:DEFINE-CSE-NODE KEY CSE-NODE)

Associates the CSE-NODE with KEY (see previous function) in the
hash table.

(CSE.NAME:CSE-NODE NAME)

Returns the CSE-NODE currently representing the value of NAME (NAME
is in the :LABELS of CSE-NODE). A new initial node representing
NAME is created if there is no node currently holding NAME.

(CSE.NAME:DEFINE-CSE-NODE NAME CSE-NODE)

Associates CSE-NODE with NAME in the hash table that maps NAMES
onto the CSE-NODEs currently holding those NAMES.

(CSE.NAME:INITIAL-CSE-NODE NAME)

Returns the initial CSE-NODE that represents the value of NAME on
entry to the basic block. Returns () if there is no such node.

(CSE.NAME:DEFINE-INITIAL-CSE-NODE NAME CSE-NODE)

Associates CSE-NODE as the initial node of NAME.

(CSE.NAME:INITIAL-USES NAME)

Returns the parents of the initial CSE-NODE of NAME; that is, all
the direct uses of the initial value of NAME.

(CSE-NODE:ASSIGN-TO-NAME CSE-NODE NAME)

Moves the label NAME to CSE-NODE; if a previous node contained NAME
as a label, NAME is removed from that node. After calling this
function, (CSE.NAME:CSE-NODE NAME) will return CSE-NODE.

PS:<C.S.BULLDOG.FLOW-ANALYSIS>CSE-HASH-TABLES.LSP.15
(defun cse.expr:define-cse-node (key cse-node)
  (assert (cse-node:is cse-node))
  (hash-table:put *cse.expr:cse-node* key cse-node))

(defun cse.name:cse-node (name)
  (let ((cse-node (hash-table:get *cse.name:cse-node* name)))
    (if (== *hash-table.not-found* cse-node)
      (cse-node:create-initial name)
      cse-node))
  (assert (cse-node:is cse-node))
  (hash-table:put *cse.name:cse-node* name cse-node)
  (hash-table:put *cse.name:initial-cse-node* name cse-node)
  (cse-node))

(defun cse.name:define-cse-node (name cse-node)
  (assert (cse-node:is cse-node))
  (hash-table:put *cse.name:cse-node* name cse-node))

(defun cse.name:initial-cse-node (name)
  (let ((cse-node (hash-table:get *cse.name:initial-cse-node* name)))
    (if (== *hash-table.not-found* cse-node)
      cse-node)
    (else
      cse-node)))

(defun cse.name:define-initial-cse-node (name cse-node)
  (assert (cse-node:is cse-node))
  (hash-table:put *cse.name:initial-cse-node* name cse-node))

(defun cse.name:initial-uses (name)
  (let ((initial-cse-node (hash-table:get *cse.name:initial-cse-node* name)))
    (if (== *hash-table.not-found* initial-cse-node)
      cse-node)
    (else
      (cse-node:parents initial-cse-node)))
  (assert (cse-node:is cse-node))
  (hash-table:put *cse.name:initial-cse-node* name cse-node))

(defun cse-node:assign-to-name (cse-node name)
  (assert (cse-node:is cse-node))
  (let ((old-cse-node (hash-table:get *cse.name:cse-node* name)))
    (if (== *hash-table.not-found* old-cse-node)
      (cse-node:labels old-cse-node)
      cse-node))
    (push (cse-node:labels cse-node) name)
    (hash-table:put *cse.name:cse-node* name cse-node))

PS: <C.S.BULLDOG.FLOW-ANALYSIS>CSE-HASH-TABLES.LSP.15
CSE-NODE

This module implements the nodes of the DAG built for basic block common subexpression elimination. Everything here is private to the few modules implementing CSE.

(defvar *cse.all-cse-nodes* ;**« A list of all CSE-NODEs created.
(defvar *cse.total-cse-nodes* ;»»* Total number of CSE-NODEs created.
)

(defun cse.initialize-cse-nodes ()
  (:= *cse.all-cse-nodes* ()
  (= *cse.total-cse-nodes* 0)
  ()
)

(defun cse-node:create ()
  (let ( (cse-node (cse-node:new) ))
    (push *cse.all-cse-nodes* cse-node)
    (:= *cse.total-cse-nodes* (+ *cse.total-cse-nodes* 1))
    cse-node)
)

(defun cse-node:create-initial ( initial-name )
  (let ( (cse-node (cse-node:create) ))
    (:= (cse-node:number cse-node) *cse.total-cse-nodes*)
    cse-node)
)

(defun cse-node:create-interior ( source children )
  (assert (consp source))
  (let ( (cse-node (cse-node:create) ) ))
)
```lisp
(defun cse-node:calculate-descendants (cse-node)
  (assert (cse-node:is cse-node))
  (let ((descendants (bit-set:empty-set)))
    (loop (for child-cse-node in (cse-node:children cse-node))
      (do ((descendants (bit-set:union (bit-set:union1 (cse-node:descendants child-cse-node) (cse-node:number child-cse-node)) descendants)))
        (push (cse-node:parents child-cse-node) cse-node)
        (push (cse-node:parents child-cse-node) cse-node)
    )))

(defun cse-node:propagate-new-descendants (cse-node)
  (let ((new-descendants (cse-node:calculate-descendants cse-node)))
    (if (! (bit-set:= new-descendants (cse-node:descendants cse-node)))
      (then
        (let ((new-descendants (cse-node:calculate-descendants cse-node)))
          (loop (for parent-cse-node in (cse-node:parents cse-node))
            (do (cse-node:propagate-new-descendants parent-cse-node)))))
    )))

(defun cse-node:make-copy-cse-node (cse-node)
  (assert (! (cse-node:labels cse-node))
    (cse-node:datatype cse-node))
  (loop (for name in (cse-node:labels cse-node))
    (initial result-list)
    (copy-cse-node ()
      (:= copy-cse-node
        (cse-node:create-interior
          (if (== 'integer (cse-node:datatype cse-node))
            '(lassign dummy dummy))
            (cse-node:assign-to-name copy-cse-node name)
            (push result-list copy-cse-node)
            (result
              (:= (cse-node:labels cse-node) ()
                result-list)))))
```
This module implements a Unix MAKE-like facility for keeping track of which transformations and computations on the flow graph are still valid. For example, FG.MOVE-LOOP-INVARlANTS destroys several computed values, such as the dominator info, because it changes the block structure of the flow graph. Any succeeding function that needs the loop information will first have to recompute the loop info. We want to automate the process of recomputing values so that we have the flexibility of reordering or turning off transformations.

**(DEF-DEPENDENCY FUNCTION**

**(NEEDS N-FL [N-F2 ...] )
(DESTROYS D-FL [D-F2 ...] )
(REINITIALIZE R-F )**

States that FUNCTION needs the values computed by the functions N-F1 before it can be executed; the values computed by the functions D-F1 are destroyed by invoking FUNCTION. The functions N-F1 and D-F1 should have been declared in DEF-DEPENDENCIES. If another dependency function destroys the values computed by FUNCTION, the reinitializing function R-F will be called after the destroying function finishes. R-F is also called at the very beginning of flow analysis.

**(FG.INITIALIZE-DEPENDENCIES)**
Forgets all previous dependencies declared by DEF-DEPENDENCY.

**(FG.INITIALIZE-DEPENDENT-FUNCTIONS)**
Forgets about any previously computed dependency functions, invoking all the reinitializers whether the corresponding function was previously computed.

**(FG.INVOKE-DEPENDENT-FUNCTION FUNCTION**

If FUNCTION has already been invoked and not subsequently destroyed, this will do nothing. Otherwise, it calls the specified function, which should have been declared in a DEF-DEPENDENCY. Before invoking FUNCTION, all the functions specified in the NEEDS clause of the corresponding DEF-DEPENDENCY are recursively invoked if they have not been previously computed. After FUNCTION terminates, the functions specified in the DESTROYS clause are forgotten; that is, the next time they are invoked by FG.INVOKE-DEPENDENT-FUNCTION either directly or implicitly by a NEEDS clause, the destroyed functions will be called to recompute their values. Any functions depending on the destroyed functions are recursively destroyed.

**(FG.DESTROY-DEPENDENT-FUNCTION FUNCTION**

"Forgets" the fact that FUNCTION may have been previously computed. Any functions that depend on FUNCTION via a NEEDS clause will also be recursively destroyed. The next time FG.INVOKE-DEPENDENT-FUNCTION is called, the a "forgotten" function will actually be recomputed.

(include flow-analysis:flow-analysis-decls)
(defvar *dep.computed-functions* () )
;; List of all function names that
;; have been previously computed
;; and not subsequently destroyed.

(PS:<C.S.BULLDOG.FLOW-ANALYSIS:DEPENDENCIES.LSP> 17)
(assert dep)

(if (! (memq function *dep.computed-functions*) ) (then
    ;; Recursively invoke any needed functions
    (loop (for needed-function in (dep.dependency:needs dep) ) (do
        (fg.invoke-dependent-function needed-function) )
       ;; Invoke the function.
       (msg 0 (t (* 2 «dep.nesting») ) "Invoking " function t)
       (funcall function)
       ;; Destroy the specified functions
       (loop (for destroyed-function in (dep.dependency:destroys dep) )
             (do
                  (fg.destroy-dependent-function destroyed-function) )
       ;; Remember the function as computed
       (if (! (memq function «dep.computed-functions») ) (then
            (push «dep.computed-functions» function) ) )
)

(defun fg.destroy-dependent-function ( function )
  (let ( (dep (dep.function dependency function) )
          (+ «dep.nesting» (+ 1 «dep.nesting») )
    (assert dep)
    ;; Recursively destroy all functions that NEED this
    ;; function
    (loop (for next-dep in *fg.all-dependencies*) (do
        (if (! (memq function (dep.dependency:needs next-dep) ) )
            (fg.destroy-dependent-function (dep.dependency:function next-dep) )
        )
    )))
  ;; Call this functions reinitializer if it has one.
  (let ( (reinitialize (dep.dependency:reinitialize dep) )
    (if (! reinitialize)
        (msg 0 (t (* 2 «dep.nesting») ) "Destroying "
            function t)
        (funcall reinitialize)
        (reinitialize))
    ;; Remove the function from the list of remembered functions.
    (= «dep.computed-functions»
        (top-level-removel function *dep.computed-functions*) )
))

PS: <C.S.BULLDOG.FLOW-ANALYSIS>DEPENDENCIES.LSP.17
This module orders the basic blocks of the flow graph by depth first order.

(FG_SET-DEPTH-FIRST-ORDER)

This should be called each time the basic block structure of the flow graph changes. The two functions are synonyms for clarity in the function dependencies (ugh).

(FG_DEPTH-FIRST-ORDERED-BBLOCK-LIST ORDERING)

Returns a list of BBLOCKs in depth first order (ORDERING is either REVERSE or FORWARD). The results of the last call to this function are remembered, so that a depth first search is actually done only once. The :DFO-NUMBER of each BBLOCK is set to its position in the forward ordering.

(include flow-analysis:flow-analysis-decls)

(declare (special
  *dfo.current-order* ;*** one of (), REVERSE, or FORWARD
  *dfo.ordered-list* ;**• order list of BBLOCKs.
  *dfo.visited-bblocks* ;*** bit set of currently visited BBLOCKs.
  *bit-set.empty-set* ))

(defun fg.set-depth-first-order ()
  (fg.initialize-depth-first-order) )

(defun fg.initialize-depth-first-order ()
  (= *dfo.current-order* ()
  (= *dfo.ordered-list* ()

(defun fg.depth-first-ordered-bblock-list ( ordering )
  (assert (& (litaton ordering
  (neq ordering '(reverse forward) ) ))

  (if (! *dfo.current-order*) (then
    (= *dfo.visited-bblocks* *bit-set.empty-set*
    (dfo.search *fg.entry-bblock*)
    (:= *dfo.visited-bblocks* *bit-set.empty-set*
    (= *dfo.current-order* 'forward)
    (loop (for bblock in *dfo.ordered-list*)
      (incr i from 1)
      (do
        (:= (bblock:dfo-number bblock) 1) )
    (if (== ordering *dfo.current-order*) (then
      *dfo.ordered-list*
    (else
      (:= *dfo.current-order* ordering)
      (= *dfo.ordered-list* (dreverse *dfo.ordered-list*) ) ))

    (for (succ-bblock in (bblock:succs bblock) ) (do
      (if (! (bit-set-member? *dfo.visited-bblocks*
        (block:number succ-bblock) )
        (dfo.search succ-bblock) )
      (push *dfo.ordered-list* bblock) )

(ddefun dfo.search ( bblock )
  (:= *dfo.visited-bblocks* (bit-set:unionl *dfo.visited-bblocks*
    (block:number bblock) )

  (for (succ-bblock in (bblock:succs bblock) ) (do
    (if (! (bit-set-member? *dfo.visited-bblocks*
      (block:number succ-bblock) )
    (dfo.search succ-bblock) )
    (push *dfo.ordered-list* bblock) )

PS:<C.S.BULLDOG.FLOW-ANALYSIS>DEPTH-FIRST-ORDER.LSP.6
DERIVATIONS

This module provides "derivations" of integer variable definitions. A derivation of a variable is an expression for the variable in terms of the induction variables of the innermost loop containing the variable's definition: if the definition is not contained in a loop, then the expression is in terms of initial program inputs.

The derivation expressions are diophantine expressions, which are defined elsewhere; the recognized operators are +, *, - , and & expresses alternative - its operands are the alternative derivations for a definition.

Derivations are computed by first, for each loop, finding the set of induction variables of that loop and splicing in a dummy BBLOCK loop header at the top of the loop that looks like:

(LOOP-ASSIGN I1)
(LOOP-ASSIGN I2)

where the II are the loop's induction variables. In this module, a variable I is an induction variable of a loop if:

1. The variable has a definition in the loop.
2. That definition reaches the loop header via one of the back-edges.
3. The variable is live on entry to the header of the loop.

After the dummy loop assignments are inserted, reaching definitions are re-calculated. This lets us easily identify which reaching definitions that reach an operand are induction variables defined in the previous iteration of the loop body.

A derivation for a definition is found by recursively tracing the reaching definitions for each operand backwards. The backwards chaining stops at definitions that aren't one of the following known operators: IADD, IMUL, ISUB, INEG, and ASSIGN. Note specifically that the backwards chaining stops at a LOOP-ASSIGN that doesn't specify an equivalent expression for the loop induction variable. The unknowns of the derivation are the STATs for which a derivation cannot be derived. Multiple reaching definitions for a variable are included in the derivation using the & operator.

Derivations for a definition are remembered as they are found in the field STAT:KNOWN-DERIVATION, so getting a derivation for all expressions is linear in the size of the flow graph.

(INITIALIZE-DERIVATIONS)
Initializes this module.

(INSERT-LOOP-ASSIGNMENTS)
Inserts a (LOOP-ASSIGN I) for each induction variable of the loop at the beginning of the loop header.

(DERIVATION STAT)
Returns the diophantine expression representing the derivation of the variable defined by STAT. The leaves of the expression are STATs and constants.

(OPERAND-DERIVATION STAT OPERAND)
Returns the derivation of an operand of STAT: OPERAND is just a variable name that is read by STAT.

(INDEX-DERIVATION STAT)
Returns the derivation of the index operand of STAT, which should be a vector reference.

(INITIALIZE-DERIVATIONS)

(INSERT-LOOP-ASSIGNMENTS)
This might be speeded up (who cares?) and simplified (I care) by just looking at the definitions that reach the loop header; it isn't necessary to consider each back edge individually.

(INITIALIZE-DERIVATIONS)

(INSERT-LOOP-ASSIGNMENTS)

(INITIALIZE-DERIVATIONS)

(INSERT-LOOP-ASSIGNMENTS)
**Returns a list of the induction variables of LOOP that have definitions reaching the loop header via the back-edge whose tail is BACK-EDGE-BBLOCK.**

(defun der.loop:back-edge-bblock:induction-vars ( loop back-edge-bblock )
  (let ((induction-names ())
        (edge-reaching-defs
          (stat-set:Intersection
           (loop:stats loop)
           (bblock:reaching-out back-edge-bblock)
           (bblock:reaching-in (loop:header loop)))))
    **EDGE-REACHING-DEFS is the set of definitions contained in the loop that reach the loop header via the back edge.**
    **Enumerate over all the variables that are live on entry to the loop header, and if any one of them has definitions in EDGE-REACHING-DEFS, it is an induction variable.**
    (loop (for-each-name-set-element (bblock:live-in (loop:header loop)) live-name)
      (do (if (! (stat-set:= *fg.empty-stat-set* (stat-set:Intersection
                                                (name:definition-stats live-name)
                                                edge-reaching-defs)))
            (then (push induction-names live-name)))))

(defun stat:derivation ( stat )
  (assert (stat:is stat))
  (if (stat:known-derivation stat) (then (stat:known-derivation stat))
    (else (= (stat:known-derivation stat)
             (caseq (stat:operator stat)
                (iadd isub inul)
                (ladd lsub lmul)
                . (stat:operand-derivation
                    stat (stat:part stat 'read1))
                . (stat:operand-derivation
                    stat (stat:part stat 'read2)))))
    (ineg (- . (stat:operand-derivation
     stat (stat:part stat 'read1)))))

(defun stat:operand-derivation ( stat operand )
  (if (numberp operand) (then operand)
    (else (if-let ( (var&derivation
                    (assoc operand (stat:known-operand-derivations stat))))
             (then (cadr var&derivation))
             (let*( (reaching-defs (stat:operand-reaching-defs stat operand))
                 (derivation
                  (loop (for-each-stat-set-element reaching-defs def-stat)
                    (initial result ())
                    (do (push result (stat:derivation def-stat))
                        (result '(& . (reverse result)))))
                    (push (stat:known-operand-derivations stat)
                        (operator-derivation)
                        derivation)))))))

(defun stat:index-derivation ( vector-stat )
  (stat:operand-derivation
   vector-stat
   (stat:part vector-stat 'index)))
DISAMBIGUATOR

This module implements the interface functions of the "disambiguator" as described in DOC:DISAMB.DOC. The description of the interface won't be repeated here (to make sure that there is one, and only one, description that is kept up to date).

(FG.INITIALIZE-NADDR:STAT-MAPPING)
(FG.INITIALIZE-DISAMBIGUATOR)
Initializes this module, clearing any old data structures.

(FG.DISAMBUATE)
Prepares for disambiguation by:
1. Adding all the assertions to the assertion database.
2. Obtaining and storing derivations for each vector index.

(FG.CREATE-NADDR:STAT-MAPPING)
Creates the mapping from source NADDR to STATs.

(START-TRACE)
See DOC:DISAMB.DOC.

(PREDECESSORS SOURCE-OPERATION TRACE-DIRECTION DATUM)
See DOC:DISAMB.DOC.

(OPER:LIVE-IN OPER)
See DOC:DISAMB.DOC.

(OPER:LIVE-OUT OPER)
See DOC:DISAMB.DOC.

(OPER:LIVE-OUT-ON-EDGE OPER DIRECTION)
See DOC:DISAMB.DOC.

(include flow-analysis:flow-analysis-decls)

(DIS.NAME:RWS NAME)
Maps a name to a list of DIS.RW records describing reads and writes of that name. NAME may be a vector.

(DIS.NAME:DEFINE-RWS NAME RWS)
Associates a list of DIS.RW records (RWS) with NAME, removing any previous association.

(DIS.NAME:ADD-RW NAME RW)
Adds RW to the list of DIS.RW records associated with NAME.

(def-atruct dla.rw
atat ; The STAT reading/writing.
name ; The scalar or vector variable being read/written (or name of the vector).
name-type ; SCALAR or VECTOR.
operand-number ; Operand number of the variable in the NADDR.
op (STAT:PART numbering).
type ; READ, CONDITIONAL-READ, WRITTEN
(date ; Random codegenerator value associated with :STAT.
() suppress);

(destruct dla.rw)

(defun dis.nane:rws ( nane )
(let ( (rws (hash-table:get *dis.nane:rws* nane) )
(= rws (hash-table.not-found) ()
(rws) )
)
)

(defun dla.nane:deflne-rwa ( nane rws )
(hash-table:put *dis.nane:rws* nane rws) )

(defun dis.nane:add-rw ( nane rw )
(let ( (rws (hash-table:get *dis.nane:rws* nane) )
(= rws (hash-table.not-found)) (then
(hash-table:put *dis.nane:rws* nane *(,rw) )
(else
(hash-table:put *dis.nane:rws* nane '(.rw ..rws) )
)
)
)

1
PS:<C.S.BULLDOG.FLOW-ANALYSIS>DISAMBIGUATOR.LSP.10


that has NADDR operations (lists) as EQ keys, and STATs as associated values.

(defun fg.initialize-disambugutator ()
  (:= «dis.name:rw» () )
)

(defun fg.initialize-naddr:stat-mapping ()
  (:= «dis.oper:stat» (hash-table:create) )
)

(defun fg.disambiguate ()
  ;*** Collect all the assertions in the program.
  (loop (for-each-stat stat)
    (when (== 'assert (stat:operator stat) )
      (de:assert (stat:part stat 'compare-op)
        (stat:operand-derivation stat (stat:part stat 'read1) )
        (stat:operand-derivation stat (stat:part stat 'read2) )
      )
    )
  ;*** Obtain derivations for the indices of vector references.
  (loop (for-each-stat stat)
    (when (stat:property? stat 'vector-reference) 
      (stat:index-derivation stat)
    )
  )

(defun fg.create-naddr:stat-mapping ()
  (loop (for-each-stat stat) (do
    (hash-table:put «dis.oper:stat» (stat:source stat) stat)
  )
)

(defun start-trace ()
  (:= «dis.name:rw» (hash-table:create) )
)

(defun predecessors ( source-operation trace-direction datum )
  (let*( (stat-written-rw (dis.stat:create-written-rw stat datum trace-direction) )
      (stat-written-name
        (if stat-written-rw (dis.rw:name stat-written-rw) () )
      )
      (stat-read-rws
        (dis.stat:create-read-rws stat datum trace-direction)
      )
      (conflicts ()
        (dis.rw:get «dis.stat:conflicts» stat trace-direction datum)
      )
    )
)

This is a test function for testing this module. STAT-NUMBER-LIST is a list of STAT numbers that are to be the trace; conditional jumps are always assumed to go to the right. The trace predecessors of each trace STAT are printed out in complete, gory detail.

(defun dis.stat:conflicts ( stat trace-direction datum )
  (let*( (stat-written-rw (dis.stat:create-written-rw stat datum trace-direction) )
      (stat-written-name
        (if stat-written-rw (dis.rw:name stat-written-rw) () )
      )
      (stat-read-rws
        (dis.stat:create-read-rws stat datum trace-direction)
      )
      (conflicts ()
        (dis.rw:get «dis.stat:conflicts» stat trace-direction datum)
      )
    )
)

==<C.S.BULLDOG.FLOW-ANALYSIS>DISAMBIGUATOR.LSP.10==
The conflicts that will be the result of this function.

(new-prev-rw () )

The new list of DIS.RW records representing the read/writes of the variable written by STAT. This list is the old such list, plus any reads of the variable by STAT, minus all reads/writes killed by STAT.

Construct the conflicts between the operands read by this STAT and previous STATs. Add the STAT's RW to the mapping NAME:RWS for each read name (only if the name isn't also defined by this stat).

(loop (for stat-read-name in stat-read-names) (do
    (loop (for prev-rw in (dis.name:rw:stat stat-read-name) )
        (lf-let ( (conflict (dis.rw:conflict rw prev-rw) )
            (push conflicts conflict)
        )
    )
    (lf (!= stat-written-name (dis.name:rw:name stat-read-name) ) (then
        (dis.name:define-rw (dis.name:rw:name stat-read-name) (dis.name:rw:rw stat-read-name) )
    )
)
)

Construct the conflicts between the operand written by this STAT and previous STATs. Remember all the previous RWs listed under the name written by STAT that weren't definitely known conflicts, forgetting all others (the ones killed by STAT); add the the write RW to the list.

(if stat-written-rw (then
    (loop (for prev-rw in (dis.name:rw:stat stat-written-name) )
        (lf-let ( (conflict (dis.rw:conflict rw prev-rw) )
            (push conflicts conflict)
        )
    )
    (lf-let ( (conflict 'possible-operand-conflict (cadr conflict) )
        (push new-prev-rws prev-rw)
    )
)
)

push new-prev-rws stat-written-rw
(dis.name:define-rw stat-written-name new-prev-rws)

conflicts)

(DIS.STAT:CREATE-READ-RWS STAT DATUM TRACE-DIRECTION)

Returns a list of DIS.RW records representing the variables read by STAT, including the "conditional-reads" of variables that are live on the off-trace edge. For now, we assume that every vector is live on every off-trace edge.

But we now do live analysis of vector names. A vector write should move up above a conditional jump if the vector is dead on the off-trace edge, no?

(dis.stat:create-read-rws (stat datum trace-direction )
(let ( (result () )
    (if (stat:property? 'conditional-jump) (then
        (let* ( (bblock (stat:bblock stat) )
            (*If this is a conditional jump, make a record for every variable that is live on the off-trace edge. Also make a record for every array name, assuming for now that every vector that is read on the off-trace edge.
        ))
    )
)

(DIS.RW:CONFICT RW PREV-RW)

Returns the PREDECEDSORS-format conflict representing the conflict, if any, between a read/write of a variable by one trace element and the read/write of the same variable by another previous trace element. For example, a read of a variable (PREV-RW) followed by another read (RW) is not a conflict, but a write after a read would be.

The form of the conflict (from DISAMB.DOC) is:

(PRED REASON SOURCE-OPERAND SOURCE-TYPE PRED-OPERAND PRED-TYPE)

(dis.stat:conflict (stat datum trace-direction )
(let ( (result () )
    (if (stat:property? 'conditional-jump) (then
        (let* ( (bblock (stat:bblock stat) )
            (*If this is a conditional jump, make a record for every variable that is live on the off-trace edge. Also make a record for every array name, assuming for now that every vector that is read on the off-trace edge.
        ))
    )
)

(dis.stat:conflict (stat datum trace-direction )
(let ( (result () )
    (if (stat:property? 'conditional-jump) (then
        (let* ( (bblock (stat:bblock stat) )
            (*If this is a conditional jump, make a record for every variable that is live on the off-trace edge. Also make a record for every array name, assuming for now that every vector that is read on the off-trace edge.
        ))
    )
)
(succ-bblock (if (== 'left trace-direction)
  (cadr (block:succ bblock)))
  (car (block:succ bblock)))

(succ-stat (block:first-stat succ-bblock)))

;*** First the off-trace live scalars.
(loop (for-each-name-set-element (stat:live-in succ-stat)
  live-name)
  (when (!== 'vector (name:type live-name))
  (do
    (push result
      (dis.rw:new
        stat stat
        name live-name
        name-type (name:type live-name)
        operand-number ()
        type 'conditional-read
        datum datum)))))

;*** Then all array names.
(loop (for-each-vector-name vector-name) (do
  (push result
    (dis.rw:new
      stat stat
      name vector-name
      name-type 'vector
      operand-number ()
      type 'conditional-read
      datum datum))))

;*** Make a record for each read variable.
(loop (for-each-stat-operand-read stat operand operand-number) (do
  (push result
    (dis.rw:new
      stat stat
      name operand
      name-type (name:type operand)
      operand-number operand-number
      type 'read
      datum datum)))))

result)

;*** (DIS.NAME-SET:LIST-SCALARS SET)
;*** Returns a list of all the scalar variable names in SET.

(defun dis-name-set:list-scalars (set)
  (loop (for-each-name-set-element set name)
    (when (!== 'vector (name:type name))
      (save name))))
(include flow-analysis:flow-analysis-decls)

(declare (special
  *dt.de-calls*
  *dt.de-successes*
  *dt.conflicts*) )

(defun fg.disambiguator-tool ()
  (let* ((vector-stats (dt.collect-vector-stats))
         (total-vector-stats (length vector-stats))
         (*dt.de-calls* 0)
         (*dt.de-successes* 0)
         (*dt.conflicts* () )
         (msg 0 t (length vector-stats) 4) " vector STATs" t)
  (loop (incr i from 1 to (length vector-stats))
    (bind stat1 (nth-elt vector-stats i))
    (do (loop (incr j from (+ 1 i) to (length vector-stats))
      (bind stat2 (nth-elt vector-stats j))
      (if (dt.conflicting-vector-stats? stat1 stat2)
        (then
          (dt.print-conflicting-vector-stats stat1 stat2)
          (msg (j (- (// (* total-vector-stats (+ -1 total-vector-stats) )
                        2) " pairs were trivial (different vectors, different loops)." t)
          (msg (j *dt.de-calls* 4) " pairs required the diophantine equation solver." t)
          (msg (j *dt.de-successes* 4) " of those pairs were possibly conflicting." t)
          (msg (j (length *dt.conflicts*) 4) " unique conflicting pairs of indices." t)
          () )
        )))
  (nsg 0 t (length vector-stats) 4) " vector STATs." t)
  (nsg (J (- (// (* total-vector-stats (+ -1 total-vector-stats) )
                        2) " pairs were trivial (different vectors, different loops)." t)
          4) " pairs required the diophantine equation solver." t)
  (nsg (J (length *dt.conflicts*) 4) " unique conflicting pairs of indices." t)
)

(defun dt.conflicting-vector-stats? ( stat1 stat2 )
  (let* ((deriv1 (stat:index-derivation stat1))
         (deriv2 (stat:index-derivation stat2))
         ([& := *dt.de-calls* (+ 1 *dt.de-calls*)]
          (= "maybe (de:possibly-equal? deriv1 deriv2 stat2)"
            ;*** Passing in STAT2 is a crock -- it just usually
            ;*** happens that STAT2 comes "after" STAT1. Sigh.
            ;*** which stat should we pass in to pick up which
            ;*** "valid" assertions?
            (& := *dt.de-successes* (+ 1 *dt.de-successes*) ) )
          )
        (dt.print-conflicting-vector-stats stat1 stat2)
        (msg "Same as"
          (stat:number (nth-elt prev-conflict 2) ) "/
          (stat:number (nth-elt prev-conflict 3) ) ") t)
        (else
          (push *dt.conflicts* '((der1 . stat1 . stat2) )
          (hprinl (dt.de:pretty (dt:normalize deriv1) )
          (terpri)
          (hprinl (dt.de:pretty (dt:normalize deriv2) )
          (terpri)
          (hprinl '(= 0 , (dt.de:pretty dexpr) )
          (terpri)
          ()
        )))
  )))

(defun dt.print-conflicting-vector-stats ( stat1 stat2 )
  (let* ((deriv1 (stat:index-derivation stat1))
         (deriv2 (stat:index-derivation stat2))
         ([& := *dt.de-calls* (+ 1 *dt.de-calls*)]
          (= "maybe (de:possibly-equal? deriv1 deriv2 stat2)"
            ;*** Passing in STAT2 is a crock -- it just usually
            ;*** happens that STAT2 comes "after" STAT1. Sigh.
            ;*** which stat should we pass in to pick up which
            ;*** "valid" assertions?
            (& := *dt.de-successes* (+ 1 *dt.de-successes*) ) )
          )
        (dt.print-conflicting-vector-stats stat1 stat2)
        (msg "Same as"
          (stat:number (nth-elt prev-conflict 2) ) "/
          (stat:number (nth-elt prev-conflict 3) ) ") t)
        (else
          (push *dt.conflicts* '((der1 . stat1 . stat2) )
          (hprinl (dt.de:pretty (dt:normalize deriv1) )
          (terpri)
          (hprinl (dt.de:pretty (dt:normalize deriv2) )
          (terpri)
          (hprinl '(= 0 , (dt.de:pretty dexpr) )
          (terpri)
          ()
        )))
  )))

(defun dt.collect-vector-stats ()
  (loop (for-each-stat stat)
    (when (stat:property? stat 'vector-reference)
      (Initial result ()
        (do (push result stat) )
      (result (dreverse result) ) )
  )))

(defun dt.stat:containing-loop ( stat )
  (loop (for-each-loop loop)
    (initial containing-loop ()
      (do (it (loop:bblockmember? loop (stat:bblock stat) ) (then
        (: = containing-loop loop) )
      )
    (result containing-loop) )
  )))

(defun dt.de:pretty ( expr )
  (* (consp expr)
    (loop (for sub-expr in (cdr expr) )
      (initial result ()
        (do (:= sub-result (dt.de:pretty sub-expr) )
      (result (dreverse result) )
    (do (:= sub-result (dt.de:pretty sub-expr) )
      (casseq (car expr)
        (+ (if (= 0 sub-result) (then
          (push result sub-result) )
        (+ 1 sub-result) )
        (return 0) )
    (result (push result sub-result) )
    (result (push result sub-result) )
  )))

PS:<C.S.BULLDOG.FLOW-ANALYSIS>DISAMBIGUATOR-TOOL.LSP.11
(stat:is expr
  (if (== 'loop-assign (stat:operator expr) )
    '((stat:part expr 'written) (stat:number expr))
    '((stat:source expr) (stat:number expr)))
  (t expr)))
This module sets the :DOMINATORS field of each BBLOCK to be the set
of BBLOCKS that dominate that BBLOCK. See chapter 13 of the Dragon
Book for the definition of dominators and an explanation of the
algorithms used to calculate them.

(defun fg.set-dominators ()
  (loop (for-each-bblock bblock)
    (initial universe (bblock-set:universe) )
    (do
      (:= (bblock:dominators bblock) universe) )
    (result
      (:= (bblock:dominators *fg.entry-bblock*)
          (bblock-set:singleton *fg.entry-bblock*) )
      )
    )
  )
)

(include flow-analysis:flow-analysis-decls)

(loop (initial change ()
  (next change ()
    (do
      (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
        (initial new-dominators ()
          (do
            (:= new-dominators
              (bblock-set:union:
                (apply 'bblock-set:intersection
                  (for (pred-bblock in (bblock:preds bblock) )
                    (save (bblock:dominators pred-bblock) )
                  )
                bblock) )
              )
            (if (! (bblock-set:= new-dominators
                            (bblock:dominators bblock) )
              (:= change t) )
              (:= (bblock:dominators bblock) new-dominators) )
            )
          (while change) )
        )
      )
    )
  )
)
Find Loops

This module finds the loops in the flow graph. A loop is defined here differently than in the Dragon Book. Each loop has a loop header that dominates all the nodes in the body of the loop. A loop header is identified by an edge $T \rightarrow H$ in the flow graph such that $H$ dominates $T$. A loop consists of the header $H$, all the backedges $T \rightarrow H$, and all the predecessors of each $T$ that are dominated by $H$.

(FG.FIND-LOOPS)
Finds all the loops in the flow graph.

(FG.PRINT-LOOPS)
Prints out all the loops.

(include flow-analysis:flow-analysis-decls)

(defun fg.find-loops ()
(fg.initialize-loops)

;; Consider each edge in turn; if the head of the edge
;; dominates the tail, then the edge is a backedge of a loop.
(loop (for-each-block bblock) (do
  (loop (for succ-bblock in (bblock:succs bblock)) (do
    (if (bblock:dominates? succ-bblock bblock) (then
      (loop:create bblock succ-bblock) ))))))

;; For each loop, find all the BLOCKs in the loop.
;; Create the :STATS of the loop (the set of all statements
;; within the loop.
(loop (for-each-loop loop) (do
  (fl.loop:find-body loop)

  (loop (for-each-block-set-element (loop:back-edges loop) bblock) (do
    (loop (for-each-block-stat bblock stat) (do
      (:= (loop:stats loop)
        (stat-set:union (loop:stats loop) stat) ))))))

;; For each loop, find all the BLOCKs of the loop that
;; are exits (having at least one successor not in the loop).
(loop (for-each-loop loop) (do
  (loop (for-each-block-set-element (loop:blocks loop) bblock) (do
    (loop (for succ-bblock in (bblock:succs bblock)) (do
      (if (! (loop:bblock:aeaber? loop succ-bblock)) (then
        (loop:create bblock succ-bblock) )
      ))))))

;; Sort the loops according to containment
(fg.sort-loops)

(defun fl.loop:find-body ( loop )
  (loop (for-each-block-set-element (loop:back-edges loop) back-edge-bblock)
    (do
      (if (!= back-edge-bblock (loop:header loop)) (then
        (loop (initial bblock)
          stack (list back-edge-bblock)
          (while stack)
          (do
            (pop stack bblock)
            (loop (for pred-bblock in (bblock:preds bblock)) (do
              (if (! (loop:bblock:aeaber? loop pred-bblock)) (then
                (loop:create bblock pred-bblock)
                (push stack pred-bblock))))))))))

(defun fg.print-loops ()
  (loop (for-each-loop loop) (do
    (asg 0 t)
    (asg "Header: " (bblock:number (loop:header loop)) t)
    (asg "Bblocks: " (e (bblock-set:print (loop:bblocks loop)) ) t)
    (asg "Stats: " (e (stat-set:print (loop:stats loop)) ) t)
    (asg "Back edges: " (e (bblock-set:print (loop:back-edges loop)) ) t)
    (asg "Exits: " (e (bblock-set:print (loop:exits loop)) ) t)
    (asg "Invariants: "
      (for (stat in (loop:invariants loop))
        (save (stat:number stat))) )
    t))))
(include flow-analysis:flow-analysis-decls)
(build '(flow-analysis:dependencies))

;* * *========================================================================
;* * (FG.INITIALIZE)
;* * Initializes the flow-graph by forgetting all previous STATS and BBLOCKS,
;* * and prepares for the creation of a new flow-graph.
;* *=========================================================================

(defun fg.Initialize ()
  (fg.initialize-naddr-to-flow-graph)
  (fg.initialize-stats)
  (fg.initialize-bblocks)
  (fg.initialize-temporary-name)
  (fg.initialize-dependent-functions)
)

;* * (FG.ANALYZE&OPTIMIZE NADDR)
;* *========================================================================

(defun fg.analyze-optimize (naddr)
  (let ( (result-naddr ())
    (fg.initialize)
    (fg.naddr-to-flow-graph naddr)
    (fg.rename-variables? (then
      (fg.invoke-dependent-function 'fg.rename-variables)))
    (fg.move-loop-invariants? (then
      (fg.invoke-dependent-function 'fg.move-loop-invariants)))
    (fg.remove-induction-variables? (then
      (fg.invoke-dependent-function 'fg.remove-induction-variables))
    (fg.eliminate-common-subexpressions? (then
      (fg.invoke-dependent-function 'fg.eliminate-common-subexpressions)))
    (fg.propagate-copies? (then
      (fg.invoke-dependent-function 'fg.propagate-copies)))
    (fg.remove-dead-code? (then
      (let ()
        (fg.remove-dead-code?)))
    (fg.redefine-assertions? (then
      (fg.invoke-dependent-function 'fg.redefine-assertions))))
  (fg.fold-constants? (then
    (fg.invoke-dependent-function 'fg.fold-constants)))
  (fg.eliminate-common-subexpressions? (then
    (fg.invoke-dependent-function 'fg.eliminate-common-subexpressions)))
  (fg.propagate-copies? (then
    (fg.invoke-dependent-function 'fg.propagate-copies)))
  (result-naddr))
)

;* * (FG.PRINT-FLOW-GRAH)
;* * Dumps out the current flow graph in semi-readable format.
;* *========================================================================

(defun fg.print-flow-graph (optional bblock-fields)
  (loop (for-each-bblock bblock) (do
    (bblock:print bblock bblock-fields))))

;* * DEPENDENCIES
;* * The dependencies between the different modules that crunch on the
;* * flow graph are recorded here to keep them all together for
;* * maintainability. See DEPENDENCIES.LSP for details of
;* * def-dependencies.
;* *=========================================================================

(defun fg.initialize-dependencies)
  (def-dependency fg.collect-names
    (reinitialize fg.initialize-names))
  (def-dependency fg.set-depth-first-order
    (reinitialize fg.initialize-depth-first-order))
  (def-dependency fg.set-reaching-defs
    (needs fg.collect-names
      fg.set-depth-first-order
      (reinitialize fg.initialize-reaching-defs))
    (needs fg.set-reaching-copies
      (reinitialize fg.initialize-reaching-copies)))
)

PS:C.S.BULLDOG.FLOW-ANALYSIS>FLOW-ANALYSIS.LSP.6
Inter-module declarations for the flow analysis modules.

Modules that manipulate the flow graph should INCLUDE this file at compile time.

(declare (special
  *hash-table.not-found*
  *bit-set.empty-set*
  *number-of-banks*
  *fg.total-stats*
  *fg.number:stat*
  *fg.total-bblocks*
  *fg.number:bblock*
  *fg.entry-bblock*
  *fg.empty-stat-set*
  *fg.number-of-reaching-iterations*
  *fg.name:name-descriptor*
  *fg.total-names*
  *fg.all-vector-names*
  *fg.empty-name-set*
  *fg.all-dependencies*
  *fg.all-loops*
  *fg.rename-variables?*
  *fg.eliminate-common-subexpressions?*
  *fg.move-loop-invariants?*
  *fg.remove-induction-variables?*
  *fg.fold-constants?*
  *fg.propagate-copies?*
  *fg.remove-dead-code?*
  *fg.remove-assertions?*
  *fg.disambiguator-tool?*
  *fg.disambiguate-banks?*
  *fg.show-unknown-bank-references?*

* Options defined in FLOW-ANALYSIS-OPTIONS

*skex.compact?*
)
)

(declare
  (lexpr vector-map:initialize)
  (lexpr vector-map:add-element)
  (lexpr bblock:print)
  (lexpr fg.print-flow-graph)
)

(eval-when (compile)
  (build `(
    interpreter:naddr
    utilities:bit-set
    utilities:sharp-sharp
    flow-analysis:stat
    flow-analysis:stat-set
    flow-analysis:bblock
    flow-analysis:bblock-set
    flow-analysis:name
    flow-analysis:name-set
    flow-analysis:loop
  ) )
)

PS:<C.S.BULLDOG.FLOW-ANALYSIS>FLOW-ANALYSIS-DECLS.LSP.14
FLOW ANALYSIS OPTIONS
This module contains the definitions of options dealing with flow analysis.

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

(def-option *fg.rename-variables* t flow-analysis: "
If T then variables are renamed wherever possible.
"
)

(def-option *fg.eliminate-common-subexpressions* t flow-analysis: "
If T then common subexpression elimination is done on NADDR basic blocks during optimization.
"
)

(def-option *fg.move-loop-invariants* t flow-analysis: "
If T then invariants are moved out of loops during NADDR optimization.
"
)

(def-option *fg.remove-induction-variables* t flow-analysis: "
If T then induction variables are eliminated and simplified.
"
)

(def-option *fg.fold-constants* t flow-analysis: "
If T then constant folding is performed.
"
)

(def-option *fg.propagate-copies* t flow-analysis: "
If T then copy propagation is performed.
"
)

(def-option *fg.remove-dead-code* t flow-analysis: "
If T then unreachable or useless code is removed from the flow graph.
"
)

(def-option *fg.remove-assertions* t flow-analysis: "
If T then assertions are removed during dead code removal.
"
)

(def-option *fg.disambiguator-tool* () flow-analysis: "
If T then the disambiguator tool is invoked, printing out all possible vector conflicts in the program.
"
)

(def-option *fg.disambiguator-banks* () flow-analysis: "
If T then the bank disambiguator is invoked, modifying vector references to contain the bank they refer to.
")
This module converts the current flow graph back into NADDR. It is primarily useful for testing.

Returns the list of NADDR corresponding to the current flow-graph.

The conversion is done in a single pass. Each basic block has a LABEL generated of the form L<number> where <number> is the number of the block. This lets us convert edges in the flow graph into GOTOs easily. A GOTO is never generated for a block that "falls through" to another block.

The :SOURCE field of each STAT is replaced by the new source operation generated for it. At present, only conditional jumps generate new source (because the labels have changed).

(defun fg.flow-graph-to-naddr ()
  (let ((naddr () ) )
    (loop (initial prev-succ-bblock () )
      (for-each-bblock bblock)
      (do
        (if (&& prev-succ-bblock
                 (= prev-succ-bblock block))
            (then
              (push naddr (goto (fgtn-bblock:label prev-succ-bblock) ) ) ) )
          (if (bblock:preds bblock)
               (push naddr (label (fgtn-bblock:label bblock) ) ) )
        (loop (for-each-bblock-stat bblock stat)
              (initial new-source () )
              (do
                (:= new-source
                    (caseq (stat:group stat)
                        (if-then-else
                            (fgtn-bblock:label (car (bblock:succs bblock) ) )
                           (fgtn-bblock:label (cadr (bblock:succs bblock) ) ) )
                        (cond-jump
                            (fgtn-bblock:label (car (bblock:succs bblock) ) )
                           (fgtn-bblock:label (cadr (bblock:succs bblock) ) ) )
                        (cond-else
                            (fgtn-bblock:label (car (bblock:succs bblock) ) )
                           (fgtn-bblock:label (cadr (bblock:succs bblock) ) ) )
                        (setf (stat:operator stat) ""))))))
      (next prev-succ-bblock
        (if (> (length (bblock:succs bblock) )
              (car (bblock:succs bblock) )
          )
        )
      (result
        (if prev-succ-bblock (then
                    (push naddr
                        (goto (fgtn-bblock:label prev-succ-bblock) ) ) )
                  (dreverse naddr ) ) )
    )
  )
))
INDUCTION VARIABLE REMOVAL

This module implements the induction variable removal algorithm of the Dragon book with some minor modifications.

(FA.REMOVE-INDUCTION-VARIABLES)
Simplifies induction variables, changing the whole program.

Here is a summary of the algorithm. Details of the implementation are provided in the procedure comments below.

A primary induction variable of a loop is a variable whose only assignments within the loop are of the form:

\[ I := I \pm C \]

where C is a loop invariant. Each such assignment to I in the loop is called a primary induction statement. This module assumes a name is loop invariant iff there are no assignments to it within the loop.

A secondary induction variable is a variable K that is assigned exactly once within the loop by a statement of one of the forms:

\[ K := C \div J \quad K := J \div C \quad K := C \]

where J is either a primary induction variable or a secondary induction variable. If J is secondary, there are more requirements. Let I be primary variable of J. There can be no assignment to I between the single assignment to J and the single assignment to K, and the definition of J in the loop must be the only reaching definition of J reaching the assignment to K.

Each secondary variable is expressed as a linear function of its primary variable:

\[ K := A \cdot I + B \]

All the secondary induction statements of a primary with the same linear function are grouped together.

A secondary or primary induction variable is "useless" if its only uses are for calculating other induction variables in the same family or within comparisons and conditional jumps.

The program is rewritten as follows: Each group of secondary induction statements with the same linear function are assigned a new unique name. Each assignment to a secondary I.V. K is replaced by:

\[ K' := K^* \]

where K' is the new name. Then at the end of the loop header, each new secondary variable is initialized:

\[ K' := A \cdot I + B \]

Then after each assignment to the primary induction variable within the loop, all the new secondary variables are stepped in parallel. If the assignment is of the form:

\[ I := I \pm C \]

Then conditional jumps and comparisons are rewritten to use the "simplest" non-useless secondary induction variable possible; this will possibly allow us to delete the original secondary induction variable.

Suppose there is a comparison:

\[ J \text{ RELOP } X \]

where J is either a primary induction variable or else a secondary I.V. with linear function:

\[ J = AJ + BJ \]

We look for another useful secondary I.V. K' that has the form:

\[ K' = AK + BJ \] where \( AK = R \cdot AJ \) for some constant R.

If we find such a K', we can rewrite the conditional test to be:

\[ TEMP := R \cdot X - BJ + BK \]
\[ K' \text{ RELOP } TEMP \]

The code for TEMP can be placed in the loop header if X is a loop constant. If there are many choices for K', we favor ones that have R = 1 or BK = 0.

After all this rewriting, many of the secondary and primary induction variables may now be truly dead — dead code removal will eliminate them entirely from the program.

When writing the assignments to the new secondary I.V.s and when rewriting comparisons, we rely on the fact that constant folding/simplification will clean up the code considerably later on.

---

### An IV-FAMILY represents one primary induction variable and its family of secondary induction variables.

#### (def-struct iv-family

- **name** Name of the primary induction variable.
- **primary-stats** **STAT-SET of primary induction statements.**
- **secondary-stats** **STAT-SET of secondary induction statements.**
- **(secondary-ivs** List of SECONDARY-IV records representing the secondary induction variables of this family.)

### A SECONDARY-IV represents all the secondary induction variables that are the same function of the primary induction variable.

#### (def-struct secondary-iv;***

---

PS:<C.S.BULLDOG.FLOW-ANALYSIS>INDUCTION-VARIABLE-REMOVAL.LSP.4
stats
naae
orlglnal-nanes
(lv-fanily (lv-suppress))
a
b
a-address
b-address
useful?

| •••=========T~==
| S
| ;*••
| •*«
| •»»
| •**
| ••»
| ••*
| *•*
| **•
| ***
| •••
| •••
| •••
| •••
| •••
| **•
| •••
| •••
| •••
| •••
| •••
| •••
| •••
| •••
| •••
| •••
| •••
| •••
| •••

maa
STAT-SET of secondary induction STATs all with the same linear function of the primary variable. The new variable naae generated for these induction STATs.

List of naaes of original secondary i.v.s.
The Induction family to which this belongs.

Constants of the linear function for these naaes, of the for A • B for I the primary induction variable. Constants are represented as normalized diophantine expressions.

The loop-Invariant variable naaes holding these constants' values, or else the numeric value of the constant if known.

True if any of the STATs forming this secondary induction variable are used for anything other than computing other induction variables.

(eval-when (compile load)
  (include flow-analysis:flow-analysis-decls) )

(defvar elvr.debugT* () ) ;••• If T, then dump out debugging info.

(declare (special
  «lvr.loop*«
  «lvr.lv-families«
  ;•*» Current loop being optimized.
  «lvr.nane:#defs«
  ;••• Hash tables used in our local loop flow analysis,
  «lvr.nane:defining-stats«
  «lvr.nane:using-stats«
  ;••• flow analysis,
  «lvr.naae:lv-faally*«
  «lvr.naae:secondary-lv«
  ;•** Hash tables aapping i.v. naaes onto IV-FAMILYs and SECONDARY-IVs.
))

(defun fa.remove-induction-variables ()
  (loop (for each-loop loop)
    (bind «lvr.loop*«
      «lvr.lv-families«
      «lvr.nane:#defs«
      «lvr.naae:defining-stats«
      «lvr.nane:using-stats«
      «lvr.naae:lv-faally*«
      «lvr.naae:secondary-lv«
      )
    (do
      (ivr.analyse-uses&defines)
      (ivr.collect-primary-iv*)
      (ivr.collect-secondary-iv*)
      (if «lvr.debug?* (then
        (msg 0 t "Loop: " t)
        (bblock:print (loop:pre-header loop) )
        (loop (for each-bblock-set-element (loop:bblocks «lvr.loop*») bblock)
          (do
            (bblock:print bblock) )
        )
      )
      )
      )
    )
  )

(defun ivr.print-iv-families ()
  (msg 0 t "Loop: " t)
  (loop (for each-bblock-set-element (loop:bblocks «lvr.loop*») bblock)
    (do
      (bblock:print bblock) )
  )

  (loop (for iv-family in «lvr.iv-families*»)
    (do
      (ivr.iv-family:simplify-induction-variables iv-family) )
    (if «lvr.debug?* (then
      (msg 0 t "Loop: " t)
      (bblock:print (loop:pre-header loop) )
      (loop (for each-bblock-set-element (loop:bblocks «lvr.loop*») bblock)
        (do
          (bblock:print bblock) )
      )
    )
    )

  (loop (for iv-family in «lvr.iv-families*»)
    (do
      (msg 0 t " ")
      (loop (for each-bblock-set-element (loop:bblocks «lvr.loop*») bblock)
        (do
          (bblock:print bblock) )
      )
    )

  )

  (defun ivr.analyse-uses&defines()
    (ivr.analyse-uses&defines)
  )

  (ivr.print-iv-families)

  (ivr.analyse-uses&defines)

  (ivr.print-iv-families)

  (ivr.analyse-uses&defines)

  (ivr.print-iv-families)

  (ivr.analyse-uses&defines)

  (ivr.print-iv-families)

  ;**»:
  ;***:
  ;*••:
  ;*••:
  ;••*
  ;*••:
  ;•••
  ;•••
  ;•••
  ;•••
  ;•••
  ;•••
  ;•••
  ;•••
  ;•••

  LOCAL LOOP FLOW ANALYSES

  Optimization of nested loops requires incremental flow analysis, since optimizing an outer loop affects the analysis of an inner one and vice versa. So we do our own simple, special-case analysis at the beginning of optimizing each loop. Groan.

  ;**: (ivr.analyse-uses&defines)
  ;**: Performs the analysis accessed by the two functions below on the current loop. Notice that we recalculate LOOP:STATS, since the optimization of an outer loop may have added new stats to

  PS:<C.S.BULLDOG.FLOW-ANALYSIS>INDUCTION-VARIABLE-REMOVAL.LSP.4
the loop.

(ivr.name:loop-def-count name)
Returns the number of defs of NAME in the current loop.

(ivr.name:loop-defining-stats NAME)
Returns the set of STATS that define NAME in the current loop.

(ivr.name:loop-using-stats NAME)
Returns the set of STATS that use NAME in the current loop.

(ivr.name:constant? name)
True if NAME is a constant within the loop (either a number or else not defined in the loop — we rely on the fact that loop invariants have already been moved out.)

(ivr.name:iv-family NAME)
If NAME has been identified as a primary induction variable, returns the corresponding IV-FAMILY; otherwise, returns ().

(ivr.name:secondary-iv NAME)
If NAME has been identified as a secondary induction variable, returns the corresponding SECONDARY-IV; otherwise, returns ().

(ivr.name:iv-family&constants NAME)
NAME must be an induction variable. Returns a list of the form:

(iv-family A B)
where IV-FAMILY is the primary family of NAME, and A and B are its linear function constants (1 and 0 if NAME is a primary iv).

(defun ivr.analyze-use&defs ()
  (= *ivr.name:defs* 0)
  (= *ivr.name:defining-stats* (hash-table:create () () (stat-set.empty-set) ))
  (= *ivr.name:using-stats* (hash-table:create () () (stat-set.empty-set) ))
  (= *ivr.name:iv-family* (hash-table:create () () (stat-set.empty-set) ))
  (= *ivr.name:secondary-iv* (hash-table:create () () (stat-set.empty-set) ))
  (= (loop:stats *ivr.loop*) (stat-set.empty-set))
  (loop (for-each-bblock-set-element (loop:blocks *ivr.loop*) bblock) (do
    (loop (for-each-bblock-stat bblock stat) (when (= (loop:assign (stat:operator stat) ) )
      (do
        (= (loop:stats *ivr.loop*) (stat-set:unioni *ivr.defs* stat) )
        (if-let ( (written (stat:part stat 'written) ) )
          (let ( (count (hash-table:get *ivr.name:defining-stats* written) )
            (def-stat (hash-table:get *ivr.name:defining-stats* written) )
            (hash-table:put *ivr.name:defining-stats* written
              (stat-set:unioni *ivr.defs* stat) )
            (loop:stats *ivr.loop*)
          )
        )
      )
    )
  )
)

(defun ivr.collect-primary-ivs ()
  (ivr.collect-primary-iv)
```
(ivr.primary-induction-variables)
loop (for var in (ivr.primary-induction-variables))
  (bind iv-family
    name var
    primary-stats (ivr.name:loop-defining-stats var))
  (save
    (hash-table:put *ivr.name:*var-family* var iv-family))
()

(ivr.stat:possible-primary-induction? stat)
(assert (stat:is-stat stat))
(let (written (stat:part stat 'written)
      (operator (stat:operator stat))
      (eq (msgq operator '(lad lsub))
        (let ((read1 (stat:part stat 'read1))
              (read2 (stat:part stat 'read2))
              (eq (= written read1)
                (ivr.name:constant? read1))
              (eq (= 'lad operator)
                (ivr.name:constant? read2))
        )))
  )
```

The `ivr.stat:possible-primary-induction?` function checks if `stat` qualifies as a primary induction stat of some IV-family; if so, it adds the primary variable `I` to the family and true is returned.

To be secondary IV, `stat` must have the form:

- `STAT: K := I */=- C` or `K := = I`
- where `J` is an already-discovered primary or secondary IV.
- If `J` is a secondary IV, then only one definition of `J` may reach `K` and no assignment of the primary variable, `I`, of `J` occurs between the assignment to `J` and `STAT`.
- Because an complicated analysis is needed to check this requirement in general (using reaching copies, which would need to be incrementally computed), we use more restricted criteria suggested by the Dragon Book:
  - If the assignments to `J` and `K` are in the same block, then the assignment to `J` must occur first and there must be no assignment to `I` in between.
  - If the assignments to `J` and `K` are in different blocks, then `J`'s block must dominate `K`'s block, and all the assignments to `I` must be in back-edge blocks of the loop (blocks whose successor is the loop header); there must be no assignment to `K` in those back-edge blocks after the assignment to `I`.
  - Both of these conditions satisfy the more general requirement. The
first condition is suggested by the Dragon Book. The second condition adds a little more generality needed for unrolling loops and "folding" the induction variable.

(defun lvr.stat:secondary-induction? (stat) (prog () (assert (stat:is stat) ) (let* ((k () ) (j 0 ) (c () ) (read1 () ) (read2 () ) ) (lf (! (nenq (stat:operator stat) '(Iadd Isub Ineg) ) ) (then (return 0 ) ) )
(desetq (read1 read2) (atat:part stat 'read) )
(?( (lvr.naae: constant? read1)
(:- c read1)
(:= j read2)
) )
( (lvr.naae:constant? read2)
(:= c read2)
(:= j read1) )
(t (return 0 ) )
(:3k (atat:part stat 'written) )
(If (II (== j k)
(1= 1 (lvr.naae:loop-def-count k) ) )
(then (return () ) )
(lf (I (lvr.stat:j:k:secondary-induction? j-k-stat) ) (then
desetq j-stat k-stat (return () ) )
(lvr.atat:add-secondary-lv stat k j c)
(return t) )
)

These requirements identify a subset of the following general condition required for K-STAT to be a secondary lv: J is primary, or if J is a secondary, then only one definition of J may reach K and no assignment of the primary variable, I, of J occurs between the assignment to J and STAT.
The more general condition can be checked using reaching copies, but it needs incremental flow analysis (ugh). So we use these restricted conditions which are easier to check and will get most cases.

(defun lvr.stat:j:k:secondary-induction? (k-stat j k)
(let ( (i 0 )
(secondary-lv () )
(j-stat () )
) (?( (lvr.naae:lv-family j)
(i-j-stat () ) )
( (:= secondary-lv (lvr.naae:secondary-lv j) )
(:= j-atat (stat-set:choose (lvr.naae:loop-defining-atats j) )
(:= i (lv-fam:lv-piece (secondary-lv:lv-fam secondary-lv) )
) )
(t (return () ) )
)
(if (! (lvr.stat:j:k:secondary-induction? j k) ) (then
(return () )
) )
(livr.stat:add-secondary-lv stat k j c)
(return t) )
)

These requirements identify a subset of the following general condition required for K-STAT to be a secondary lv: J is primary, or if J is a secondary, then only one definition of J may reach K and no assignment of the primary variable, I, of J occurs between the assignment to J and STAT.
The more general condition can be checked using reaching copies, but it needs incremental flow analysis (ugh). So we use these restricted conditions which are easier to check and will get most cases.

(livr.stat:j:k:secondary-induction? k-stat j k)
(let ( (i 0 )
(secondary-lv () )
(j-stat () )
) (?( (lvr.naae:lv-family j)
(i-j-stat () ) )
( (:= secondary-lv (lvr.naae:secondary-lv j) )
(:= j-atat (stat-set:choose (lvr.naae:loop-defining-atats j) )
(:= i (lv-fam:lv-piece (secondary-lv:lv-fam secondary-lv) )
) )
(t (return () ) )
)
(if (== i k)
(then (return 0 ) )
)
(livr.stat:j:k:secondary-induction? stat j k) )
(else
(if (at-stat:choose-stat-set j-stat k-stat)
(loop (for-each-stat-set-element
(livr.naae:loop-defining-stats i-stat)
1-stat)
(bind i-bblock (stat-bblock 1-stat) )
(do (if (! (bblock-set:member? (loop:back-edges lvr.loops) i-bblock) )
(then (return () ) )
)
(if (! (lv-fam:loop-defining-stats i-stat)
(bblock:back-edge-block i-bblock) )
(then (return () ) )
) )
(result t) )
) )))

(livr.stat:j:k:secondary-induction? k-stat j k)
(let ( (i 0 )
(secondary-lv () )
(j-stat () )
) (?( (lvr.naae:lv-family j)
(i-j-stat () ) )
( (:= secondary-lv (lvr.naae:secondary-lv j) )
(:= j-atat (stat-set:choose (lvr.naae:loop-defining-atats j) )
(:= i (lv-fam:lv-piece (secondary-lv:lv-fam secondary-lv) )
) )
(t (return () ) )
)
(if (== i k)
(then (return 0 ) )
)
(livr.stat:j:k:secondary-induction? stat j k) )
(else
(if (at-stat:choose-stat-set j-stat k-stat)
(loop (for-each-stat-set-element
(livr.naae:loop-defining-stats i-stat)
1-stat)
(bind i-bblock (stat-bblock 1-stat) )
(do (if (! (bblock-set:member? (loop:back-edges lvr.loops) i-bblock) )
(then (return () ) )
)
(if (! (lv-fam:loop-defining-stats i-stat)
(bblock:back-edge-block i-bblock) )
(then (return () ) )
) )
(result t) )
) )))

(livr.stat:j:k:secondary-induction? k-stat j k)
(let ( (i 0 )
(secondary-lv () )
(j-stat () )
) (?( (lvr.naae:lv-family j)
(i-j-stat () ) )
( (:= secondary-lv (lvr.naae:secondary-lv j) )
(:= j-atat (stat-set:choose (lvr.naae:loop-defining-atats j) )
(:= i (lv-fam:lv-piece (secondary-lv:lv-fam secondary-lv) )
) )
(t (return () ) )
)
(if (== i k)
(then (return 0 ) )
)
(livr.stat:j:k:secondary-induction? stat j k) )
(else
(if (at-stat:choose-stat-set j-stat k-stat)
(loop (for-each-stat-set-element
(livr.naae:loop-defining-stats i-stat)
1-stat)
(bind i-bblock (stat-bblock 1-stat) )
(do (if (! (bblock-set:member? (loop:back-edges lvr.loops) i-bblock) )
(then (return () ) )
)
(if (! (lv-fam:loop-defining-stats i-stat)
(bblock:back-edge-block i-bblock) )
(then (return () ) )
) )
(result t) )
) )))

(livr.stat:j:k:secondary-induction? k-stat j k)
(let ( (i 0 )
(secondary-lv () )
(j-stat () )
) (?( (lvr.naae:lv-family j)
(i-j-stat () ) )
( (:= secondary-lv (lvr.naae:secondary-lv j) )
(:= j-atat (stat-set:choose (lvr.naae:loop-defining-atats j) )
(:= i (lv-fam:lv-piece (secondary-lv:lv-fam secondary-lv) )
) )
(t (return () ) )
)
(if (== i k)
(then (return 0 ) )
)
(livr.stat:j:k:secondary-induction? stat j k) )
(else
(if (at-stat:choose-stat-set j-stat k-stat)
(loop (for-each-stat-set-element
(livr.naae:loop-defining-stats i-stat)
1-stat)
(bind i-bblock (stat-bblock 1-stat) )
(do (if (! (bblock-set:member? (loop:back-edges lvr.loops) i-bblock) )
(then (return () ) )
)
(if (! (lv-fam:loop-defining-stats i-stat)
(bblock:back-edge-block i-bblock) )
(then (return () ) )
) )
(result t) )
) )))
(defun ivr.bblock:no-assignments? (stat1 stat2 stat )
  (loop (initial stat (stat:succ stat)) (do
    (if (stat) (then
      (return (1)) )
    (if (= stat stat2) (then
      (return 1)) )
    (if (= i (stat:part stat 'written)) (then
      (return 1)) )
    (:= (stat:succ stat) ) ) ) ) )

1 and JB = 0 if J is a primary induction variable.

**•**

**•»• • K := J • C k := C • J**

**•»• • K := J - C k := C - J**

**••« K := - J**

**••* K := J • C k := C • J**

where J is an already-known induction variable whose form is:

**•**

J = JA • I + JB (for primary induction variable I)

**•**

JA = 1 and JB = 0 if J is a primary induction variable.

**•**

The linear function constants of the new secondary induction variable

K are calculated by substituting in for the constants of J.

**•**

Note that if there is already a secondary induction variable with
the same constants, we just add STAT to the set of STATs of that
secondary variable.

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

(defun ivr.stat:add-secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (lv-faally ja jb) (ivr.stat:lv-faally:constants j) )
    (a 0 )
    (b () )
    (secondary-lv 0 ) )
  ;** Calculate the linear function constants of STAT by
  ;** substituting in for the constants of the induction variable.
  ;**
  (caseeq (stat:operator stat)
    (add ;••• K := J + C ➞> JA • I + (JB + C)
      (:° a ja)
      (:= b '(- ,c ,jb) ) )
    (sub ;••• K := J - C ➞> JA • I + (JB - C)
      (:° a ja)
      (:= b '(- ,c ,jb) ) )
    (if (= c (stat:part stat 'read2)) (then
      (:= a j)
      (:= b '(- ,c ,jb) ) ) )
  (error (list stat "Unexpected operator.") ) )
  (else
    (:= a '(e ,ja) )
    (:= b '(- ,c ,jb) ) )
  (ineq ;••• K := -J ➞> -JA • I - JB
    (:° a '(- ,ja) )
    (:= b '(- ,c ,jb) ) )
  (imul ;••• K := C • J ➞> C • JA • I + C • JB
    (:° a '(- ,ja) )
    (:= b '(- ,c ,jb) ) )
  )
  (error (list stat "Unexpected operator.") ) )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

;** (ivr.stat:add-secondary-lv:stat k j c)

;***

(defun lvr.stat:add-secondary-lv:stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***

(defun lvr.stat:add-secondary-lv:secondary-lv (stat k j c)
  (assert (stat:ls stat) )
  (let«( ( (secondary-lv:stats aecondary-lv) )
    (atat-aet:unionl ftftft stat) )
  )

:*** Add STAT to the SECONDARY-IV that already has the same
:*** constants, or else create a new SECONDARY-IV for STAT
:*** if no previous secondary IV has the same constants.

:***
Sets the :USEFUL? field of SECONDARY-IV. A secondary variable is useful if at least one of the secondary induction STATs is used for something other than calculating other secondary induction variables in the same family or conditional jumps.

(defun lvr.secondary-lv:mark-if-useful (secondary-lv)
  (assert (secondary-lv:is secondary-lv))
  (if (for-some (var in (secondary-lv:orignal-names secondary-lv))
        (lvr.lv-family:induction-var:useful? secondary-lv var))
    (then (:= (secondary-lv:useful? secondary-lv) t) )))

Returns true if the primary induction variable is used for something other than conditional jumps, asserts, or computing induction variables within the family.

(defun lvr.lv-family:useful? (lv-family)
  (assert (lv-family:is lv-family))
  (lvr.lv-family:induction-var:useful? lv-family)
  (lvr.lv-family:name lv-family))

Returns true if VAR is used for something other than conditional jumps, asserts, or computing other induction variables within the family.

(defun lvr.lv-family:induction-var:useful? (lv-family var)
  (assert (lv-family:is lv-family))
  (let ((lv-stat (stat-set:union (lv-family:primary-stat lv-family)
                                (lv-family:secondary-stat lv-family)))
        (loop (for-each-stat-set-element lv-stat)
              (when (if-compare (stat:group use-stat) lv-family)
                    (do (lvr.lv-family:induction-var:jumap-stat:simplify lv-family:
                                      (lv-family:name lv-family) use-stat))))))

Rewrite each conditional jump use of a useless secondary lv.

(defun lvr.lv-family:simplify-induction-variables (lv-family)
  (assert (lv-family:is lv-family))
  (defun lvr.lv-family:generate-loop-algn secondary-lv)
    (do (for secondary-lv in (lv-family:secondary-lvs lv-family))
        (lvr.secondary-lv:generate-loop-algn secondary-lv) )}

Rewrite each conditional jump use of the primary lv, provided the primary lv is used only for conditional jumps and secondary lv.

Rewrite each conditional jump use of a useless secondary lv.

Generate code in the pre-header for the linear constants A and B of each secondary variable if needed, assigning :A-ADDRESS and :B-ADDRESS.

Generate pseudo-ops in the loop header that relate the secondary variables to the primary variable, for use by the assertion facility.

Rewrite each conditional jump use of the primary lv, provided the primary lv is used only for conditional jumps and secondary lv.
(loop (for var in (secondary-iv:original-names secondary-iv)) (do
  (loop (for-each-stat-set-element
    (ivr.name:loop-using-stats var)
    use-stat
    (when (= 'if-compare (stat:group use-stat)))
    (do
      (ivr.iv-family:induction-var:jump-stat:simplify
        iv-family var use-stat)
    )
  )
)
)

• Generate code for each of the secondary lvs that decouples then from the primary lv.

• (ivr.secondary-lv:slapllfy secondary-lv)

•• (ivr.secondary-lv:generate-constant-code secondary-lv)
  (assert (secondary-lv:la aecondary-lv))
  (let*
    (a-address (ivr.de:generate-and-insert-naddr secondary-lv 'pre-header () ))
    (b-address (ivr.de:generate-and-insert-naddr secondary-lv 'pre-header () ))
    (:= (secondary-lv:a-address secondary-lv) a-address)
    (:= (secondary-lv:b-address secondary-lv) b-address)
  )

•• (ivr.secondary-lv:generate-loop-asslgn secondary-lv)
  (assert (secondary-lv:ls secondary-lv))
  (let*
    (lv-faally (secondary-lv:lv-faally secondary-lv))
    (l (lv-faally:naae lv-faally))
    (tenp naddr)
    (1vr.dexpr:generate-naddr
      (* (* .1 .(secondary-lv:a-address secondary-lv))
        (secondary-lv:b-address secondary-lv))
    )
    (ivr.insert-naddr
      append naddr
      (loop-asslgn
        ,(secondary-lv:naae secondary-lv)
        tenp)
      before
      (bblock:flrat-atat (loop:header +ivr.loop-8) )
    )
  )


•• Simplifies a conditional jump that uses a useless secondary or primary induction variable by rewriting the conditional jump to use a simpler secondary induction variable.

•• The conditional jump test has the form:
  J RELOP X

  where J is either a secondary or primary iv and expressed in terms of the primary as:
  J = AJ • I • BJ

•• We look for another useful secondary lv K that has the form:
  K = AK • I • BK where AK = R • AJ for some known number R.

•• If we find such a K, we can rewrite the conditional test to be:
  TEMP := R • X - R * BJ • BK
  K RELOP TEMP (if R is positive)
  TEMP RELOP K (if R is negative)

•• The code for TEMP can be placed in the loop header if X is a loop constant.

•• Note that when we “replace” a STAT, we really only change its source, so that reaching uses, etc. will continue to be available (sigh).
(defun lvr.iv-family:a:simplest-secondary-iv*r (iv-family a)  
  (assert (iv-family:is iv-family))  
  (loop (for secondary-iv in (iv-family:secondary-ivs iv-family))  
    (when (secondary-iv:useful? secondary-iv))  
      (bind r-exp (de:divide (secondary-iv:a secondary-iv) a))  
      (when r-exp)  
        (bind r-sum (cadr r-exp))  
        (when (= 2 (length r-sum)))  
          (bind r-prod (cadr r-sum))  
          (when (= 2 (length r-prod)))  
          (bind ((0 r) r-prod))  
          (bind b (secondary-iv:b secondary-iv))  
          (initial simplest-secondary-iv  
            simplest-r  
            simplest-cost 10000  
            cost 0)  
          (do  
            (:= cost 0)  
            (if (!= r 1) (then  
              (:= cost 2) ) )  
            (lf (!= b '(ft (• (• 0) ) ) ) (then  
              (-s coat (• cost 1) ) ) )  
            (lf (< cost simplest-cost) (then  
              := simplest-cost  
              := simplest-r  
              := simplest-secondary-iv) ) ) ) )  
  (result  
    (if simplest-secondary-iv  
      '(simplest-secondary-iv . simplest-r)  
      () ) )  
  ))))

(IVR.SIMPLIFY SECONDARY-IV)
---
Replaces all assignments to secondary induction variables of IV-FAMILY  
in the current by assignments rewritten in terms of their linear  
functions of the primary variable, decoupling secondary variables  
from the primary ones.
---
Each assignment of a secondary variable in SECONDARY-IV is replaced by:
---  
    J := NEW-NAME
---
where NEW-NAME is the name of a new induction variable created to  
replace the set of all the secondary ivs that have the same linear  
function of the primary (SECONDARY-IV represents that set).
---
After each primary induction statement I := I + C, we insert
---
    NEW-NAME := NEW-NAME + A * C
---
In the loop header, NEW-NAME is initialized to be:
---
    NEW-NAME := A * I + B.
---
Note that when we “replace” a STAT, we really only change its source.
so that reaching uses, etc. will continue to be available (sigh).

(defun lvr.secondary-lv:simplify (secondary-lv)
 (assert (secondary-lv:ls secondary-lv))
 (let*( (lv-faally (secondary-lv:lv-faally secondary-lv))
        (lv-faally:naae lv-faally)
        (new-naae (secondary-lv:naae secondary-lv))
        (a-address (secondary-lv:a-address secondary-lv))
        (a (secondary-lv:a secondary-lv))
        (b (secondary-lv:b secondary-lv)))
  ; Replace each assignment of secondary variables J by
  ; J := NEW-NAME.
  (loop (for each-stat-set-element (secondary-lv:stats secondary-lv) stat)
    (bind J (stat:part stat 'written))
    (do (:= (stat:source stat) '(lasslgn ,J ,new-naae) ) )

  ; Generate statements in the pre-header of the form that
  ; evaluate:
  ; NEW-NAME := A \* I + B
  ; where I is the primary induction variable and A and B are
  ; the linear function constants of the secondary variable.
  (let ( (address naddr) (lvr.de:generate-naddr dexpr) )
    (lvr.lnaert-naddr naddr where? stat)
    (caseq where?
      (before (loop (for oper in naddr) (do (stat:append-stat stat (stat:create oper) ) ) )
      (after (loop (for oper in (reverse naddr) ) (do (stat:append-stat stat (stat:create oper) ) ) )
      (pre-header (loop (for oper in naddr) (initial pre-header (loop:pre-header «lvr.loop«) )
        (do (stat:append-stat stat (stat:create oper) ) )
      (t (error (list where? "Invalid WHERE? position." ) ) )
    )

(defun lvr.de:generate-and-insert-naddr (dexpr where? stat)
 (assert ([[ (stat:ls stat) (stat:ls stat) ) ]
 (let ( (address naddr) (lvr.de:generate-naddr dexpr) )
   (lvr.lnaert-naddr naddr where? stat)
   (caseq where?
     (before (loop (for oper in naddr) (do (stat:append-stat stat (stat:create oper) ) ) )
     (after (loop (for oper in (reverse naddr) ) (do (stat:append-stat stat (stat:create oper) ) )
     (pre-header (loop (for oper in naddr) (initial pre-header (loop:pre-header «lvr.loop«) )
       (do (stat:append-stat stat (stat:create oper) ) )
     (t (error (list where? "Invalid WHERE? position." ) ) )
   )

(defun lvr.de:generate-naddr (dexpr)
 (assert ([[ (stat:ls stat) ]

(defvar (IVR.DE:GENERATE-AND-INSERT-ADDR DEXPR WHERE? STAT)
 Generates NADDR for an unnormalized diophantine expression and inserts
 it somewhere in the current, returning the name of the temporary
 variable holding the expression's value (or a number if the value
 of the expression is constant). WHERE? and STAT specify where the
 NADDR is to be inserted:
 WHERE? = 'BEFORE Right before STAT.
 WHERE? = 'AFTER Right after STAT.
 WHERE? = 'PRE-HEADER At the end of the pre-header of the current loop.

(defvar (IVR.INSERT-ADDR NADDR WHERE? STAT)
 Inserts a list of NADDR operations into a spot in the current loop.
 WHERE? and STAT have the same meaning as above.

(defvar (IVR.DEXPR:GENERATE-ADDR DEXPR)
 Generates NADDR for unnormalized diophantine expression DEXPR, which
 represents a linear function constant of an induction variable. The
 result returned has the form:
ADDRESS NADDR

where ADDRESS is the temporary name generated to hold the constant and NADDR is a list of NADDR operations generating that constant. If DEXPR represents a single known number, then ADDRESS will be that number and NADDR will be ()..

(defun lvr.dexpr:generate-naddr ( dexpr )
  (lvr.dexpr-*:generate-naddr (cadr (de:noraallze dexpr)) ) )

(defun lvr.dexpr-*:generate-naddr ( (() (()
  (constant) . prods) )
  (lf (I prods) (then
    (:= prods (sort prods (f:1 ( prodl prod2 )
          (lf (> (cadr prodl) (cadr prod2) )
              t
              (lexorder (cddr prodl) (cddr prod2) ) ) ) )
    (loop (for prod in prods)
      (initial address (lf (== 0 constant) () constant)
          naddr ()
          tenp () )
      (bind (prod-address prod-naddr) (lvr.dexpr-*:gonerate-naddr prod address) )
      (do
        (lf address (then
          (:= tenp (fa.teaporary-naae () )
          (:= naddr (append naddr *( (Inul .teap .address .var) ) )
          (:= address teap) )
          (else
            (:* address var) ) )
      ) )
    )
  ) ) ) )
  (result '(.address .naddr) ) ) )

(defun lvr.dexpr-«.-generate-naddr ( (() constant . vars) lgnore-slgn? )
  (let ( (address ()
            naddr ()
            teap () )
    (lf lgnore-slgn? (then
      (:= constant (abs constant) )
    )
    (?( (== -1 constant)
       (:= tenp (fa.teaporary-naae () )
       (:= naddr '( (lneg .teap .(car vars) ) )
       (:= address teap)
       (:= vars (cdr vars) )
     )
    (!== 1 constant)
    (:= address constant) )
  )
)

(loop (for var in vars) (do
  (if address (then
    (:= temp (fa.teaporary-name () )
    (:= naddr (append naddr '((isub .temp .address .var) )
            (:= address temp)
            (else
              (:= address var) )
            (:= naddr prod-naddr)
            (:= address prod-address) )
    )
  )

(result '(.address .naddr) ) ) )

PS:<C.S.BULLDOG.FLOW-ANALYSIS>INDUCTION-VARIABLE-REMOVAL.LSP.4
This module computes live variable information, using the algorithm in the Dragon Book, Chap. 14. As a special hack, the live-out of every exit BBLOCK is set to be the set of names defined in any LIVE pseudo-ops in the exit block. This guarantees that common sub-expression elimination will optimize the exit block correctly.

(FG.SET-LIVE-NAMES)
- Sets :LIVE-IN and :LIVE-OUT of each BBLOCK to be the NAME-SETS of names live on entry and exit to the block.

(STAT:LIVE-IN STAT)
- Returns the NAME-SET of names live on entry to STAT.

(STAT:LIVE-OUT STAT)
- Returns the NAME-SET of names live on exit to STAT.

.include flow-analysis:flow-analysis-decls
(defmacro bblock:def ( bblock ) *(bblock:gen .bblock) )
(defmacro bblock:use ( bblock ) *(bblock:klll .bblock) )

(defun fg.set-live-names ()
 (ln.set-deftuse)
 (loop (for-each-bblock bblock) (do
 (:= (bblock:live-in bblock) *fg.empty-name-set*)
 (:= (bblock:live-out bblock) *fg.empty-name-set*) )
 )
 (loop (initial change () )
 (next change () )
 (do
 (loop (for block in (fg.depth-first-ordered-bblock-list 'reverse) )
 (initial new-out ()
 (do
 (:= new-out 'name-set:union
 (for (succ-bblock in (bblock:succs bblock) ) (save
 (bblock:live-in succ-bblock) ) )
 )
 (if (/ (name-set:= new-out (bblock:live-out bblock) ) )
 (:= change t) )
 (:= (bblock:live-out bblock) new-out)
 (:= (bblock:live-in bblock)
 (name-set:union (name-set:difference (bblock:live-out bblock)
 (bblock:def bblock))
 (bblock:use bblock) ) )
 (while change )
 (ln.hack-exit-bblocks) )
)
)

(defun ln.set-deftuse ()
 (loop (for-each-bblock bblock)
 (do
 (loop (for-stat-opand-read stat name)
 (:= use (name-set:union use name) )
 )
 (
)
)

(defun ln.hack-exit-bblocks ()
 (loop (for-each-bblock bblock)
 (when (! (bblock:succs bblock) ) )
 (do
 (loop (for-stat-opand-read stat name)
 (:= (bblock:live-out bblock) name )
 )
 ))

;*** We could do some caching here of results, like was done for *** STAT:REACHING-DEFS, but it might not be worth it.

(defun ln.stat:live-in ( stat )
 (ln.stat:live-in-exit stat t )
)
(defun ln.stat:live-out ( stat )
 (ln.stat:live-in-exit stat t )
)
(defun ln.stat:live-in-exit ( stat entry? )
 (assert (stat:is-stat)
 (loop (initial block (stat:bblock block)
 live-out (block:live-out block)
 (for-stat-opand-read block succ-stat)
 (until (/ (1 entry?)
 (:= stat succ-stat) )
 (do
 (if (stat:definition? succ-stat) (then
 (:= live-out
 (name-set:difference live-out
 (name-set:singleton (stat:part succ-stat 'written) ) ) )
 )
 )
 )
 )
 )
)
(loop (for-each-stat-operand-read succ-stat name) (do
  (live-out (name-set:union! live-out name) ))
)

(until (entry?
  (stat succ-stat )
  (result live-out) ))
A LOOP describes one loop in the flow graph:

```
(def-struct loop
  header ; A BBLOCK that is the single entry into the loop,
  found-pre-header ; A BBLOCK that is the pre-header of the entry
                   (this is filled in on the fly).
  back-edges ; BBLOCK-SET of BBLOCKs in the loop whose
              successors include the header,
  bblocks ; BBLOCK-SET of all the BBLOCKs in the loop.
            Include the header and the back edges,
  stats ; STAT-SET of stats in the loop,
  invariants ; Ordered list of STATs that are invariant,
  exits ; All the BBLOCKS in the loop that have a
         successor not in the loop.
)
```

(FG.INITIALIZE-LOOPS)
Forgets all previous loops.

(LOOP:CREATE TAIL-BBLOCK HEAD-BBLOCK)
TAIL-BBLOCK and HEAD-BBLOCK describe a backedge of a loop. If
there is already a loop with HEAD-BBLOCK as the head, then this
backedge is just added to the loop. If not, a new loop is created
(and remembered).

(FG.SORT-LOOPS)
Sorts the loops by order of containment (outer loops come first)
and secondarily by source order of the headers.

(LOOP:BBLOCK-MEMBER? LOOP BBLOCK)
Returns true if BBLOCK is part of LOOP.

(LOOP:STAT-MEMBER? LOOP STAT)
Returns true if STAT is part of LOOP.

(LOOP:NAME:DEFINING-STATS LOOP NAME)
Returns the STAT-SET of definitions of NAME within the loop.

(LOOP:PRE-HEADER LOOP)
Returns the pre-header BBLOCK of LOOP (creating one if necessary).
A pre-header is a block that is a predecessor of the header and that
has only one successor, the header. The pre-header is where any
invariants will be moved to.

(LOOP (FOR-EACH-LOOP LOOP) ...)
Enumerates LOOP through each loop of the flow graph, in order of
containment (outer loops first).

(include flow-analysis:flow-analysis-decls)
(devar *fg.all-loops* () ;** List of all the loops.
(defun fg.initialize-loops ()
  (:= *fg.all-loops* ()
)

GF: <C.S.BULLDOG.FLOW-ANALYSIS>LOOP.LSP.7
(def-simple-loop-clause for-each-loop ( clause )
  (list ( (for-each-loop var) clause ))
  (if (! (= 2 (length clause))
    (litatom var))
    (error (list clause "Invalid FOR-EACH-LOOP syntax.") ) )
' (for ,var in *fg.all-loops* ) ) )

PS:<C.S.BULLDOG.FLOW-ANALYSIS>LOOP.LSP.7
LOOP INVARIANT MOTION

This module moves invariant code out of loops, closely following the algorithms in the Dragon Book, Chapter 13.

(FG.SET-LOOP-INVARIANTS)
Sets the :INVARIANTS field of each LOOP to be an ordered list of STATS that compute values that are invariant for each execution of the loop. The list is ordered by order that the invariant statements are found.

(FG.MOVE-LOOP-INVARIANTS)
Moves as many of the invariants as possible out of each loop.
*** WARNING *** Most of the flow analysis information is invalidated after this; the only thing guaranteed to be correct is the BBLOCK, STAT, and NAME information.

(include flow-analysis :flow-analysis-decls)

(defun fg.set-loop-invariants ()
  (loop (for-each-loop loop) (do
    (lia.loop:set-invariants loop)
  )
)

(defun fg.move-loop-invariants ()
  (list (* lia.moved-invariants* ) )
  
  (loop (for-each-loop loop) (do
    (lia.loop:moved-invariants loop)
  )
)

(defun lia.moved-invariants ()
  (list (* lia.moved-invariants* ) )

(defun lia.loop:moved-invariants ( loop )
  (loop (for-each-loop loop) (do
    (lia.loop:moved-invariants loop)
  )
)

(defun lia.loop:set-invariants ( loop )
  ;; First check every statement in LOOP for possible invariancy.
  ;;
  (loop (for-each-stat-set-element (loop:stats loop) stat) (do
    (lia.loop:stat:process-possible-invariancy loop stat)
  )
)

(defun lia.loop:stat:process-possible-invariancy ( loop stat )
  ;; This checks to see if STAT (assumed to be in LOOP) is an invariant in LOOP that isn't now marked as invariant. If it is, it is added to the :INVARIANTS of LOOP, and true is returned. False is returned otherwise.

(defun lia.loop:stat:process-possible-invariancy ( loop stat )
  (if (ftft (stat:definition? stat)
    (! (memq stat (loop:invariants loop) ) )
    (lia.loop:stat:process-possible-invariancy loop stat)
  )
  (then
    (push (loop:invariants loop) stat)
    t
  )
  (else
    ()
  )
)

(defun lia.loop:stat-invariant? ( loop stat )
  Returns true if STAT is invariant in LOOP, that is, if each operand of STAT is invariant.

(defun lia.loop:stat-invariant? ( loop stat )
  (loop (for-each-stat-operand-read stat name) (do
    (lia.loop:stat-operand-invariant? loop stat name)
  )
  (result t)
)

(defun lia.loop:stat-operand-invariant? ( loop stat name )
  ;; Conceptually, repeated passes are made over the stats of the loop marking invariants, until no new invariants are marked. For efficiency, statements are only rechecked for invariancy if at least one of their reaching definitions were marked as invariant.

(defun lia.loop:stat-invariant? ( loop stat )
  (loop (for-each-stat-operand-read stat name) (do
    (lia.loop:stat-operand-invariant? loop stat name)
  )
  (result t)
)

(defun lia.loop:stat-operand-invariant? ( loop stat name )
  ;; Returns true if STAT is invariant in LOOP, that is, if each operand of STAT is invariant.

(defun lia.loop:stat-invariant? ( loop stat )
  (loop (for-each-stat-operand-read stat name) (do
    (lia.loop:stat-operand-invariant? loop stat name)
  )
  (result t)
)

(defun lia.loop:stat-operand-invariant? ( loop stat name )
  ;; Now make repeated passes over the using stats of the invariants so far discovered, adding any newly discovered invariants to LOOP, stopping when we make an entire pass without discovering a new invariant.

(defun lia.loop:stat-operand-invariant? ( loop stat name )
  (loop (initial change ()
    (next change ()
      (do (loop (for invariant-stat in (loop:invariants loop) ) (do
        (lia.loop:stat:process-possible-invariancy loop invariant-stat)
      )
      (when (stat-set-member? (loop:stats loop) invariant-stat)
        (do
          (if (lia.loop:stat:process-possible-invariancy loop stat) (then
            (! (lia.loop:stat-invariant? loop stat) )
          )
        )
      )
    )
  )
  (while change)
  (:= (lia.loop:move-invariants loop)
    (reverse (lia.loop:move-invariants loop) )
  )

(defun lia.loop:move-invariants ( loop )
  ;; Conceptually, repeated passes are made over the stats of the loop marking invariants, until no new invariants are marked. For efficiency, statements are only rechecked for invariancy if at least one of their reaching definitions were marked as invariant.

(defun lia.loop:move-invariants ( loop )
  (loop (for-each-loop loop) (do
    (lia.loop:move-invariants loop)
  )
)

(defun lia.loop:move-invariants ( loop )
  (loop (for-each-loop loop) (do
    (lia.loop:move-invariants loop)
  )
)

(defun lia.loop:move-invariants ( loop )
  (loop (for-each-loop loop) (do
    (lia.loop:move-invariants loop)
  )
)

(defun lia.loop:move-invariants ( loop )
  (loop (for-each-loop loop) (do
    (lia.loop:move-invariants loop)
  )
)
Returns true if the operand NAME of STAT is invariant in LOOP.

An operand is invariant if either:

1. It is a constant, or
2. All of its reaching defs are outside of the loop, or
3. There is exactly one reaching def and it is already marked invariant.

(defun llim.loop:stat-operand-inv? (loop stat name)
  ; Is the operand a constant?
  (if (numberp name)
    ; Or are all the reaching defs outside of the loop, or
    ; invariant?
    (loop (initial all-outside? t
           all-invariant? t
           count 0)
          (for-each-stat-set-element (stat:operand-reaching-defs stat name)
            def-stat)
          (do (++ count)
               (= all-outside? (all-outside? def-stat)
                             (loop:stat:member? loop def-stat))
               (= all-invariant? (all-invariant? def-stat
                                      (loop:invariants loop))))
   ))

(defun llim.loop:move-inv? (loop)
  ; Moves as many invariant STATS out of LOOP as possible. An invariant
  ; statement S that defines name A may be moved out of the loop if:
  ; 1. A is not defined elsewhere in the loop.
  ; 2. A) A is dead on every exit from the loop, or b) S dominates every
  ;    exit of the loop.
  ; 3. Each use of this definition of A is reached only by this definition
  ;    and has no other defs of A reaching it.
  ; 4. If the reaching definitions of the operands of S come from inside
  ;    the loop, then those definitions have been previously moved out
  ;    of the loop as an invariant.
  (defun llim.loop:move-inv? (loop)
    (loop (label 1)
          (for-each-block-set-element (loop:exits loop) exit)
          (do (loop (for exlt-succ in (block:succs exit)))
               (when (! (block:aeaber? loop exlt-succ))
                (do (if (block:live-in exlt-succ)
                           (leave 1 ()))))
          )
          (result t)
    )
  )

Note that invariants are tested for moving eligibility in the order

that they were marked invariant (in the order of LOOP:INVARIANTS).
This insures that we consider an invariant only after all its
predecessor invariants have been considered; if a def has operands
that depend on previous invariants, those invariants must have been
moved out in order for def to be moved out.

(defun llim.loop:move-inv? (loop)
  (loop (label 1)
        (for-each-block-set-element (loop:exits loop) exit)
        (do (loop (for exlt-succ in (block:succs exit)))
             (when (! (block:aeaber? loop exlt-succ))
              (do (if (block:live-in exlt-succ)
                       (leave 1 ()))))
        )
        (result t)
  )

PS: <C.S.BULLDOG.FLOW-ANALYSIS>LOOP-INVARIANT-MOTION.LSP.7
(defun lim.loop:stat-dominates-all-exits? (loop stat)
  (loop (initial stat-bblock (stat:bblock stat))
    (for-each-bblock-set-element (loop:exits loop) exit-bblock)
    (do
      (if (! (bblock:dominates? stat-bblock exit-bblock))
        (return 1)))
  (result t)))

(defun lim.loop:def-reaches-all-uses? (loop def-stat name)
  (loop (label 1)
    (for use-stat in (atat:reaching-uses def-stat))
    (when (stat-set:member? (loop:stats loop) use-stat))
    (do
      (loop (for-each-stat-set-element
        (stat:operand-reaching-defs use-stat name)
        reaching-stat)
        (do
          (if (!= reaching-stat def-stat)
            (leave 1())))
    (result t)))

(defun lim.loop:operand-defs-moved? (loop stat)
  (loop (for-each-stat-operand-read atat name)
    (initial loop-reaching-defs 0)
    (do
      (= loop-reaching-defs
        (atat-set:intersection (atat:operand-reaching-defs atat name)
          (loop:stats loop)))
      (if (! (atat-set:size empty-stat-set)
          (stat-set:difference loop-reaching-defs
            *lim.moved-invariants*))
        (return 1)))
    (result t)))
This module contains functions for converting NADDR to flow graphs.

(include flow-analysis:flow-analysis-decls)

(FG.INITIALIZE-NADDR-TO-FLOW-GRAPH NADDR)
*** Initializes this module.

(defun fg.initialize-naddr-to-flow-graph ()
  (= *fg.entry-bblock* () )
)

(FG.NADDR-TO-FLOW-GRAPH NADDR)
*** Constructs a flow-graph from NADDR (a list of NADDR operations).

(defun fg.naddr-to-flow-graph ( naddr )
  (ntfg.naddr-to-stats naddr)
  (ntfg.stats-to-bblocks)
  (fg.remove-unreachable-bblocks)
)

(NTFG.NADDR-TO-STATS NADDR)
*** Constructs the graph of STATs corresponding to NADDR (a list of NADDR operations).
*** Two passes are used. The first pass creates a STAT for each NADDR operation (except LABEL and GOTO) and installs the "value" of each NADDR label in a symbol table. The second pass goes over each STAT and replaces the symbol labels in the :SUCC field with the corresponding STAT successor.

When we are finished, the :SUCC and :PRED fields of the STAT are lists of STATs that are the predecessors and successors of the STAT in the flow graph. When we make basic blocks, those fields will be set to single STATs (the pred and succ within the BBLOCK).

(defun ntfg.naddr-to-stats ( naddr )
  (let ( (labels (hash-table:create) ) )
    (loop (Initial stat ()
              current-labels ()
              prev-stat ()
            )
            (for oper in naddr)
            (do
              (caseq (oper:operator oper)
                (label ;*** Remember this label on the list of current equivalent labels. If the previous operation falls through to here, set its successor to be this label.
                  (push current-labels (oper:part oper 'label1) )
                  (if prev-stat (then
                    (push (stat:succ prev-stat)
                          (oper:part oper 'label1) )
                    (:= prev-stat ()))))
                (goto ;** Equate all current equivalent labels with the destination of this GOTO. If the previous operation falls through to here, set its successor to be the destination of the GOTO.
                  (for (label in current-labels) (do
                    (hash-table:put labels label
                                      (oper:part oper 'label1) )
                    (:= current-labels () )
                  ))
                  (if prev-stat (then
                    (push (stat:succ prev-stat)
                          (oper:part oper 'label1) )
                    (:= prev-stat ()))))
                (t ;* Some random operation: Create a STAT for it and equate all current equivalent labels to it. If the previous operation falls through to here set its successor to be the new STAT. If STAT is a conditional jump, set its successors to be the true/false labels.
                  (:= stat (atat:create oper) )
                  (for (label in current-labels) (do
                    (hash-table:put labels label
                                      (oper:part oper 'label1) )
                    (:= current-labels () )
                  ))
                  (if prev-stat)
(push (stat:succ prev-stat) stat)
(if (stat:property? stat 'conditional-jump) (then
  (push (stat:succ stat)
    (if ( oper:part oper 'label1) (then
      (push (stat:succ stat)
        (oper:part oper 'label2)
        (:= prev-stat ())
      (else
        (:= prev-stat stat))))
    (else
      (:= prev-stat stat)))
  ))
)

;; Now make a second pass over all STATS, replacing symbolic
;; label successors with actual STATS, and setting the
;; predecessors field of each STAT.
(loop (for-each-stat stat) (do
  ;; Replace labels with actual STATS
  (loop (Initial rest-succs (stat:succ stat))
    (while rest-succs)
    (do
      (lf (litaton (car rest-succs))
        (:= (car rest-succs)
          (ntfg.label:stat labels (car rest-atat))
          (next rest-succs (cdr rest-succs))))
  )
  ;; The :SUCCE field was built in reverse (tricky!).
  (:= (stat:succ stat) (dreverse (stat:succ stat))
    (:= (stat:succ (stat:pred stat)) (stat:pred stat))
    ;; Set the predecessor field of each successor to
    ;; include this stat.
    (for (succ-stat in (stat:succ stat)) (do
      (push (stat:pred succ-stat) stat)
    ))
)

;; (NTFG.LABEL:STAT LABELS LABEL)
;; Returns the STAT associated with LABEL in the hash table LABELS.
;; Labels can be equated with other labels in the table, so we have
;; to follow such chains until we find an actual STAT.
(defvar ntfg.label:stat (labels label)
  (loop (while (litaton label))
    (do
      (result label)
    )
  )
)

;; (NTFG.STATS-TO-BBLOCKS)
;; Constructs the basic blocks corresponding to the graph of STATS built
;; by NTFG.NADDR-TO-STATS. Once the BBLOCKS are built, the :SUCCE and
;; :PRED field of each STAT are changed to form a doubly linked list of
;; STATS within the BBLOCK.
;; Sets :FG.ENTRY-BBLOCK to the first STAT's BBLOCK (we assume it is
;; the entry node of the graph).
(defvar ntfg.stats-to-bblocks ()
  ;; For each STAT in the graph (in source order), if it is
  ;; the leader of a basic block, start a new basic block.
  ;; If it isn't a leader, add it to the current basic block.
  (loop (Initial initial-bblock ()
    (for-each-stat stat)
    (do
      (if (ntfg.stat:bblock-leader? stat) (then
        (:= bblock (bblock:create))
        (:= (bblock:flrat-atat bblock) stat)
        (loop (Initial next-stat stat)
          (do
            (:= (stat:bblock next-stat) bblock)
            (:= (bblock:last-stat bblock) next-stat)
            (next next-stat (car (stat:succ next-stat)))
            (while (ntfg.Batat:bblock-leader? next-stat) )))
      )))
  )))

;; For each BBLOCK, set its predecessor and successor blocks.
;; Also sets the entry BBLOCK to be the first one.
(:= ntfg.entry-bblock ()
  (loop (for-each-bblock bblock)
    (do
      (if (! ntfg.entry-bblock)
        (:= ntfg.entry-bblock bblock)
        (:= (bblock:preds bblock)
          (for (stat in (stat:pred (bblock:first-stat bblock))))
            (save (stat:bblock stat))))
      (:= (bblock:succs bblock)
        (for (stat in (stat:succ (bblock:last-stat bblock))))
          (save (stat:bblock stat))))
    )))

;; Currently :SUCCE and :PRED of each STAT are lists of
;; the successor and predecessor STATS of that STAT.
;; Change :PRED and :SUCCE so that they now form a doubly
;; linked list of STATS within a BBLOCK. :PRED of the
;; first stat in a BBLOCK is (); likewise of :SUCCE of the
;; last STAT.
(loop (for-each-bblock bblock)
  (do
    (loop (Initial initial-stat (bblock:first-stat bblock))
      (while stat)
      (while (= bblock (stat:bblock stat)))
      (while (lisp (stat:succ stat)))
      (do
        (:= (stat:pred stat) prev-stat)
        (if prev-stat
          (:= (stat:succ stat) succ-stat)
          (:= (stat:succ succ-stat) stat)
        )
      )))
)

(:= (stat:succ prev-stat) stat) )

(next prev-stat stat
stat (car (stat:succ stat) ))

(result
(:= (stat:succ prev-stat) () () ) )

============================================

*** (NTFG.STAT:BBLOCK-LEADER? STAT)
*** Returns true if STAT is the leader (beginning) of a basic block.
*** A STAT is a leader if it has 0 or more than 1 predecessors or if
*** its predecessor has more than one successor.
***

(defun ntfg.stat:bblock-leader? ( stat )
  (if (not (stat:pred stat))
    (< 1 (length (stat:pred stat)) )
    (< 1 (length (stat:succ (car (stat:pred stat)) ) ) ) ) )

PS: <C.S. BULLDOG.FLOW-ANALYSIS>NADDR-TO-FLOW-GRAPH.LSP.7
This module defines the type "name". A name is a variable 
name used within the flow graph. For now, names are 
represented externally as symbols to make handling NADDR 
straightforward; a hash table is used to map names onto 
descriptive information. We'll see how much we pay for this 
hashing.

(NAME:CREATE NAME TYPE SIZE) 
Creates a new name with symbolic name NAME. TYPE is either 
SCALAR or VECTOR. SIZE is the declared size of a vector.

(NAME:DEFINING-STATS NAME) 
Returns the STAT-SET of definitions defining NAME (initially empty).

(NAME:ADD-DEFINING-STAT NAME STAT) 
Adds STAT to the set of definitions defining NAME.

(NAME:USING-STATS NAME) 
Returns the STAT-SET of statements using NAME.

(NAME:TYPE NAME) 
The type of a name, either VECTOR or SCALAR.

(NAME:LENGTH NAME) 
The declared size of a vector name.

(NUMBER:NAME INDEX) 
Returns the name with :NUMBER INDEX.

#Syntax for (NUMBER:NAME INDEX)
(LOOP (FOR-EACH-NAME NAME) ...) 
Enumerates through each known NAME.

(LOOP (FOR-EACH-VECTOR-NAME NAME) ...) 
Enumerates through each known NAME declared to be a vector.

(FG.PRINT-NAMES) 
Prints out all known NAMES and their associated info.

(type ;*** SCALAR or VECTOR.
length ;*** Length of a VECTOR.
defining-stats ;*** The STAT-SET of definitions defining this stat.
using-stats ;*** The STAT-SET of statements using this stat.

(declare (special 
*fg.name:name-descriptor* ;*** A hash table mapping variable names onto 
*fg.number:name* ;*** An array mapping numbers to names.
*fg.total-names* ;*** Total number of names.
*fg.all-vector-names* ;*** List of all vector names.
)

(declare (special 
*hash-table.not-found* )
)

(defun fg.initialize-names () 
(:= »fg.all-vector-names» () ) 
(:= »fg.number:name* (hash-table:create) ) 
(vector-map:initialize »fg.number:name* »fg.total-names* 100) ()
)

(defun name:create ( name type length ) 
(let ( (desc (hash-table:get »fg.name:name-descriptor* name )) )
(if (== »hash-table.not-found* desc) (then 
(hash-table:put »fg.name:name-descriptor* name 
(name-descriptor:new name number »fg.total-names* 
(type type 
(length length 
(defining-stats »fg.empty-stat-set* 
(using-stats »fg.empty-stat-set* ) ) 
(vector-map:add-element »fg.number:name* »fg.total-names* 
name 100 t) )

(else ;*** if we have just discovered NAME to be a vector, 
;*** override any previous guess; sigh. PARAM for 
;*** for vectors seems bogus?
(if (== type "vector") (then 
(:= (name-descriptor:type desc) type) 
(:= (name-descriptor:length desc) length ) ) )
)
)
)

(defun name:define-stat ( name ) 
(let ( (desc (hash-table:get »fg.name:name-descriptor* name )) )
(assert (== desc »hash-table.not-found*))
"NAME:DEFINING-STAT: name = " name 
(name-descriptor:define-stat desc) )

(defun name:add-definining-stat ( name stat ) 
(let ( (desc (hash-table:get »fg.name:name-descriptor* name )) )
(assert (== desc »hash-table.not-found*))
"NAME:ADD-DEFINING-STAT: name = " name * stat = " (h stat) )

(declare (special
*fg.name:name-descriptor* ;*** A hash table mapping variable names onto 
*fg.number:name* ;*** An array mapping numbers to names.
*fg.total-names* ;*** Total number of names.
*fg.all-vector-names* ;*** List of all vector names.
) )

(defun fg.initialize-names () 
(:= »fg.all-vector-names» () ) 
(:= »fg.number:name* (hash-table:create) ) 
(vector-map:initialize »fg.number:name* »fg.total-names* 100) ()
)
(defun name:using-stats (name)
  (let ((desc (hash-table:get «fg.name:name-descriptor* name)))
    (assert (!= desc »hash-table.not-found*)
      "NAME:USING-STATS: name = " name)
    (name-descriptor:using-stats desc) )
)

(defun name:add-using-stat (name stat)
  (let ((desc (hash-table:get «fg.name:name-descriptor* name)))
    (assert (!= desc »hash-table.not-found*)
      "NAME:ADD-USING-STAT: name = " name » stat = " (h stat)
    (:= (name-descriptor:using-stats desc)
      (stat-set:union 1 (name-descriptor:using-stats desc)
        stat) )
    )
)

(defun name:type (name)
  (let ((desc (hash-table:get «fg.name:name-descriptor* name)))
    (assert (!= desc »hash-table.not-found*)
      "NAME:TYPE: name 3 » name)
    (name-descriptor:type desc) )
)

(defun name:length (name)
  (let ((desc (hash-table:get »fg.name:name-descriptor* name)))
    (assert (!= desc »hash-table.not-found*)
      "NAME:LENGTH: name = " name)
    (name-descriptor:length desc) )
)

(defun name:number (name)
  (let ((desc (hash-table:get »fg.name:name-descriptor* name)))
    (assert (!= desc »hash-table.not-found*)
      "NAME:NUMBER: name 3 • name)
    (name-descriptor:number desc) )
)

(defun number:name (index)
  (if *fg.number:name* .(read) )
)

(defun simple-loop-clause for-each-name (clause)
  (let ((for-each-name var clause)
          (index (intern (gensym)) )
    (if (! (= 2 (length clause))
          (error (list clause "Invalid FOR-EACH-NAME syntax."))
    ' ( (initial ,var () )
        (incr .index from 0 to (+ -1 »fg.total-names*)
          next ,var (if (for-each-name var) (t 21)
          (error (list clause "Invalid FOR-EACH-NAME syntax."))) )
        (when ,var ) )
    )
)

(defun simple-loop-clause for-each-vector-name (clause)
  (let ((for-each-vector-name var clause)
    (if (! (= 2 (length clause))
      (error (list clause "Invalid FOR-EACH-VARIABLE-NAME syntax."))) )
  )
)

(defun fg.print-names ()
  (loop (for-each-name name) (do
    (asg 0 naae *: (t 10) (name:type name) (t 17) (name:length name)
      (t 21) (e (stat-set:print (name:describing-stats name)) t) )
    0 )
  )
)

(error (list clause "Invalid FOR-EACH-VECTOR-NAME syntax.")
  ' ( (for ,var in »fg.all-vector-names) )
)

(defun fg.print-names ()
  (loop (for-each-name name) (do
    (asg 0 naae *: (t 10) (name:type name) (t 17) (name:length name)
      (t 21) (e (stat-set:print (name:describing-stats name)) t) )
    0 )
  )
)

(error (list clause "Invalid FOR-EACH-VECTOR-NAME syntax.")
  ' ( (for ,var in »fg.all-vector-names) )
)

(defun fg.print-names ()
  (loop (for-each-name name) (do
    (asg 0 naae *: (t 10) (name:type name) (t 17) (name:length name)
      (t 21) (e (stat-set:print (name:describing-stats name)) t) )
    0 )
  )
)

(error (list clause "Invalid FOR-EACH-VECTOR-NAME syntax.")
  ' ( (for ,var in »fg.all-vector-names) )
)
NAME-SETS

Sets of NAMEs are represented using NAME-SETs, currently implemented as BIT-SETs.

• FG.EMPTY-NAME-SET*
  The empty NAME-SET.

(NAME-SET:UNIVERSE)
  The set of all NAMEs.

(NAME-SET:SINGLETON NAME)
  Creates a new set containing NAME.

(NAME-SET:MEMBER? SET NAME)
  Returns true if NAME is a member of SET.

(NAME-SET:INTERSECTION SETI SET2 ...)
  Returns a new set that is the intersection of all the given sets.

(NAME-SET:UNION SETI SET2 ...)
  Returns a new set that is the union of all the given sets.

(NAME-SET:UNION1 SET NAME)
  Unions a single NAME into SET.

(NAME-SET:DIFERENCE SETI SET2)
  Returns a new set that contains all elements in SETI not in SET2.

(NAME-SET:= SETI SET2)
  Returns true if the two sets are equal.

(NAME-SET:SIZE SET)
  Returns the number of elements in the set.

(LOOP (FOR-EACH-NAME-SET-ELEMENT SET NAME)
  Enumerates NAME through each element in SET. Uses DEF-SIMPLE-LOOP-CLAUSE.

(NAME-SET:PRINT SET)
  Prints SET.

(NAME-SET:LIST SET)
  Returns a list of the names in SET.

===============================================

(include flow-analysis:flow-analysis-decls)

(declare (special *fg.total-names*) )

(defvar *fg.empty-name-set* () ) ;*** the empty NAME-SET.

(defun name-set:universe ()
  (bit-set:universe *fg.total-names* )
)

(defun name-set:singleton ( name )
  (bit-set:singleton (name:number name) )
)

(defun name-set:member? ( set name )
  (bit-set:member? set (name:name) )
)
This module implements the flow analysis algorithm that finds the "reaching copies" of each basic block. The definition of a reaching copy used here is a little more general than that used in the Dragon Book on page 487.

The set of reaching copies at point S2 is the set of statements S1 of the form:

\[ S1: A := f(B1, B2, ...) \]

such that for every path from the initial node to S2:

1. The path includes S1.
2. After the last occurrence of S1 on the path, there are no definitions of the variables A or any of the Bi.

Intuitively, if S1 is a reaching copy at S2, S1 could be copied right before S2 without affecting the program.

(defun fg.set-reaching-copies ()
  (rc.set-gen&kill)
  (loop (for-each-bblock bblock) (do
    (:= (bblock:reaching-copies-in bblock) (stat-set:universe))
    (:= (bblock:reaching-copies-out bblock)
        (stat-set:difference (bblock:reaching-copies-in bblock)
                               (bblock:kill bblock)))
    (:= (bblock:reaching-copies-in bblock) *fg.entry-bblock* *fg.empty-stat-set*)
    (:= (bblock:reaching-copies-out bblock) *fg.entry-bblock*)
    (bblock:gen *fg.entry-bblock*)
    (loop (initial change () new-in () )
      (next change () new-in)
      (do (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward))
             (do (:= new-in
                  (:= (bblock:reaching-copies-in bblock) (apply 'stat-set:intersection
                            (for (pred-bblock in (bblock:preds bblock) ) (save
                              (bblock:reaching-copies-out pred-bblock)))))
                  (:= (bblock:reaching-copies-in bblock) new-in)
                  (:= (bblock:reaching-copies-out bblock)
                      (stat-set:union
                       (stat-set:difference (bblock:reaching-copies-in bblock)
                                            (bblock:kill bblock))
                       (bblock:gen bblock)))
                  (if (! (stat-set:= new-in (bblock:reaching-copies-in bblock))
                         (:= change t))
                    (:= (bblock:reaching-copies-out bblock)
                        (stat-set:difference (bblock:reaching-copies-in bblock)
                                             (bblock:kill bblock))
                        (bblock:gen bblock)))
      (while change)
    (result (:= (bblock:gen bblock) gen)
            (:= (bblock:kill bblock) kill)))))
  ()
)

(defun rc.set-gen&kill ()
  (loop (for-each-bblock bblock) (do
    (loop (for-each-bblock-stat bblock stat)
      (initial gen *fg.empty-stat-set*
                 kill *fg.empty-stat-set*)
      (do
        (lf (stat:definition? stat) (then
          (let ( (killed-defs (rc.stat:killed-copies stat) )
                (:= kill (stat-set:union kill killed-defs))
                (:= gen (stat-set:union1
                             (stat-set:difference gen killed-defs)
                             stat)))
          (result (:= (bblock:gen bblock) gen)
                  (:= (bblock:kill bblock) kill))))))
    (result (:= (bblock:gen bblock) gen)
            (:= (bblock:kill bblock) kill)))))

(defun fg.stat-reaching-copies-stat ()
  (loop (for-each-bblock-stat bblock stat)
    (initial gen *fg.empty-stat-set*
               kill *fg.empty-stat-set*)
    (do
      (lf (stat:definition? stat) (then
        (let ( (killed-defs (rc.stat:killed-copies stat) )
              (:= kill (stat-set:union kill killed-defs))
              (:= gen (stat-set:union1
                       (stat-set:difference gen killed-defs)
                       stat)))
        (result (:= (bblock:gen bblock) gen)
                (:= (bblock:kill bblock) kill))))))
    (result (:= (bblock:gen bblock) gen)
            (:= (bblock:kill bblock) kill)))))

PS:<C.BULLDOG.FLOW-ANALYSIS>REACHING-COPIES.LSP.1
(defun stat:reaching-copies (stat)
  (assert (stat:is-stat stat))
  (loop (initial reaching-copies (bblock:reaching-copies-in
                                   (stat:bblock-stat stat))))
  (for-each-bblock-stat (stat:bblock-stat stat) pred-stat)
  (while (!= pred-stat stat)
    (when (stat:definition? pred-stat))
    (do
      (:= reaching-copies
        (stat-set:union (stat-set:difference &amp;&amp;
                         (rc.stat:killed-copies pred-stat)
                         pred-stat))
      )
    )
  )
)

;===================================================================
;***
;*** (RC.STAT:KILLED-COPIES STAT)
;***
;*** Returns that STAT-SET of definitions that are copy-killed by STAT.
;*** STAT is assumed to be a definition.
;***
;===================================================================
(defun rc.stat:killed-copies (stat)
  (let ( (written (stat:part stat 'written)))
    (stat-set:union
      (name:defining-stats written)
      (name:using-stats written)))
)
REACHING DEFINITIONS

This module implements the flow analysis algorithm that finds the definitions that reach each basic block. See chapters 12 and 14 of the Dragon Book for explanation of the algorithm.

(FG.SET-REACHING-DEFS)
Does the flow analysis of reaching definitions, setting the :REACHING-IN and :REACHING-OUT of each BBLOCK in the flow graph to be a STAT-SET of definitions reaching that BBLOCK.

(STAT:REACHING-DEFS STAT)
Returns the STAT-SET of definitions reaching STAT.

(STAT:OPERAND-REACHING-DEFS STAT NAME)
Returns the STAT-SET of definitions of NAME reaching STAT.

(FG.INITIALIZE-REACHING-DEFS)
Initializes this module.

(STAT:KILLED-DEFS STAT)
Returns the STAT-SET of definitions possibly killed by STAT (which is assumed to be a definition).

(include flow-analysis:flow-analysis-decls)

(declare (special
   *fg.number-of-reaching-iterations* ;*** # of iterations used for calculating
   ;*** reaching def
   *rd.last-reaching-stat* ;*** The last STAT given to STAT:REACHING-DEFS.
   *rd.last-reaching-def* ;*** The last result of the last call to
   ;*** STAT:REACHING-DEFS. These two variables
   ;*** form a one-result cache (hack).
   ))

*** (FG.INITIALIZE-REACHING-DEFS)

*** (FG.SET-REACHING-DEFS)

(defun fg.set-reaching-defs ()
   (fg.initialize-reaching-defs)
)

(defun rd.set-gen&kill ()
   (loop (for-each-bblock bblock)
      (do
         (:= (bblock:reaching-in bblock) *fg.empty-stat-set*)
         (:= (bblock:reaching-out bblock) (bblock:gen bblock))
      )
   )
   (:= *fg.number-of-reaching-iterations* 0)
   (loop (initial change ()
      (next change ()
         (do
            (++; *fg.number-of-reaching-iterations*)
            (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward))
               (do
                  (:= new-in 'stat-set:union
                     (for (pred-bblock in (bblock:preds bblock)) (save
                        (bblock:reaching-out pred-bblock) ) ) ) )
                  (if (! (stat-set:= new-in (bblock:reaching-in bblock))
                     (:= change t) )
                  )
                  (:= (bblock:reaching-in bblock) new-in)
                  (:= (bblock:reaching-out bblock)
                     (stat-set:union (stat-set:difference (bblock:reaching-in bblock)
                        (bblock:kill bblock)) (bblock:gen bblock)) ) ) )
            )
         )
      )
   )
   (rd.set-gen&kill)

   (loop (for-each-bblock bblock)
      (do
         (loop (for-each-bblock-stat bblock stat)
            (initial gen *fg.empty-stat-set*
             kill *fg.empty-stat-set*)
            (do
               (if (stat:definition? stat) (then
                  (let ((killed-defs (stat:killed-defs stat))
                     (:= kill (stat-set:union kill killed-defs))
                     (:= gen (stat-set:union1
gen (stat-set:difference gen killed-defs stat))) ) )
               )
            )
            (result (:= (bblock:gen bblock) gen)
               (:= (bblock:kill bblock) kill) )
            )
         )
      )
   )
)

1 PS:<C.S.BULLDOG.FLOW-ANALYSIS>REACHING-DEFS.LSP.5
(defun stat:reaching-defs (stat)
  (assert (stat:is stat))
  (if (= stat *rd.last-reaching-stat*) (then
    *rd.last-reaching-defs*)
    (else
      (loop (initial reaching-defs ()
          defined-name ()
          pred-stat ()
          begin
            (if (&& *rd.last-reaching-state*
              (stat:bblock stat)
              (stat:bblock *rd.last-reaching-stat* (stat:pred stat)))
              (then
                (:= pred-stat *rd.last-reaching-state*)
                (:= reaching-defs *rd.last-reaching-defs*)
              (else
                (:= pred-stat (bblock:first-stat (stat:bblock stat))
                (:= reaching-defs (bblock:reaching-in
                    (atat:bblock stat) ) ) ) )
            )
          (while (!= stat pred-stat) )
          (do
            (if (stat:definition? pred-stat)
              (:= reaching-defs
                (stat-set:union
                  (stat-set:difference reaching-defs
                    (stat:killed-defs pred-stat)
                    pred-stat) ) ) )
            (next pred-stat (stat:succ pred-stat) )
            )
          )
          result
          (:= *rd.last-reaching-stat* stat)
          (:= *rd.last-reaching-defs* reaching-defs)
          reaching-defs ) )
    ) )
  )
)

(*** (STAT:OPERAND-REACHING-DEFS STAT NAME) ***

(defun stat:operand-reaching-defs (stat name)
  (stat-set:intersection (stat:reaching-defs stat)
    (name:defining-stats (stat:part stat 'written)) )
)

(*** (STAT:KILLED-DEFS STAT) ***

(defun stat:killed-defs (stat)
  (name:defining-stats (stat:part stat 'written)) )

PS: <C.S.BULLDOG.FLOW-ANALYSIS>REACHING-DEFS.LSP.5
REACHING USES

This module sets :REACHING-USES of each STAT to be the set of STATS that use the value of the STAT.

The reaching uses are calculated by inverting STAT:REACHING-DEFS, not by a separate flow-analysis pass. A separate flow analysis pass would require a representation for uses and use-sets, and measurements show that this representation (lists of STATS stored in each STAT) will consume less space as long as the average number of uses reaching a STAT is less than about 4 (for most programs it appears to be 2). Besides, this was a lot easier to implement.

(FG.INITIALIZE-REACHING-USES)
Clears the :REACHING-USES of each STAT in the flow graph.

(FG.SET-REACHING-USES)
Calculates the :REACHING-USES of each STAT.

#include flow-analysis:flow-analysis-decls

(defun fg.initialize-reaching-uses ()
  (loop (for-each-stat stat) (do
    (:= (stat:reachlng-uses stat) () )))
)

(defun fg.set-reaching-uses ()
  (fg.initialize-reaching-uses)
  ;; For each STAT in the flow graph, STAT is pushed on the the list :REACHING-USES of the STATS whose definitions reach the given STAT and are actually used by it. The STATS are enumerated in basic block order so that STAT:REACHING-DEFS will go fastest (due to its cache of previous results).
  (loop (for-each-bblock bblock) (do
    (loop (for-each-bblock-stat bblock stat)
      (bind used-reaching-defs
        (stat-set:intersection
          (stat:reachlng-defs stat)
          (loop (for-each-stat-operand-read stat name)
            (reduce stat-set:union "fg.empty-stat-set" (name:defining-stats name)) )))
      (do (loop (for-each-stat-set-element used-reaching-defs def-stat) (do
        (push (stat:reachlng-uses def-stat) stat) ))))
    ))
  )
)
Dead Code Removal

**FG.REMOVE-ASSERTIONS**
Removes assertions and LOOP-ASSIGNs from the flow graph; a following
(FG.REMOVE-DEAD-CODE) will remove any support code needed for the
assertions.

**FG.REMOVE-DEAD-CODE**
This function removes "dead code" from the flow graph. First all
useless STATs are removed, then all unreachable BBLOCKS are removed.
A STAT is useless if:
1) It is a definition and its value isn't used in ultimately
producing the final live values of the flow graph.
2) It is a conditional jump both of whose branches jump to the
same spot.

**FG.REMOVE-UNREACHABLE-BBLOCKS**
Removes all BBLOCKS from the flow graph that cannot be reached from
the entry point.

```
(include flow-analysis:flow-analysis-decls)
```

```
(defun fg.remove-assertions ()
  ;*** FG.REMOVE-ASSERTIONS
  ;*** Removes assertions and LOOP-ASSIGNs from the flow graph; a following
  ;*** (FG.REMOVE-DEAD-CODE) will remove any support code needed for the
  ;*** assertions.
  (loop (for-each-stat stat)
    (when (not (eq (stat:operator stat) "(assert loop-assign)") ))
      (do (rdc.stat:delete stat) )
  )
)
```

```
(defun fg.remove-dead-code ()
  ;*** FG.REMOVE-DEAD-CODE
  ;*** This function removes "dead code" from the flow graph. First all
  ;*** useless STATs are removed, then all unreachable BBLOCKS are removed.
  ;*** A STAT is useless if:
  ;*** 1) It is a definition and its value isn't used in ultimately
  ;***    producing the final live values of the flow graph.
  ;*** 2) It is a conditional jump both of whose branches jump to the
  ;***    same spot.
  (loop (while (rdc.remove-useless-stats) )
    (fg.remove-unreachable-bblocks)
  )
)
```

```
(defun fg.remove-unreachable-bblocks ()
  ;*** FG.REMOVE-UNREACHABLE-BBLOCKS
  ;*** Removes all BBLOCKS from the flow graph that cannot be reached from
  ;*** the entry point.
  (let ( (unprocessed ()) )
    ;*** Find all BBLOCKS that have no predecessors and are
    ;*** not the entry node.
    (loop (for-each-bblock bblock)
      (when (not (eq (bblock:preds bblock) "(fg.entry-bblock)") ))
        (do (bblock:delete bblock)
          (for (succ-bblock in (bblock:succs bblock) ) (do
            (if (not (eq (bblock:preds succ-bblock) "(fg.entry-bblock)") ))
              (push unprocessed succ-bblock) )
          )
        )
    )
  )
)
```
The set of all STATs that have been marked as useful so far.

(deleted-cond-jump? () )

*** True if we deleted a conditional jump.

*** Initialize STACK to be all statements that aren't one- or two-in-one-out or vector references.

(loop (for-each-stat stat)
  (when (! (eq (stat:group stat) '(one-in-one-out two-in-one-out vstore vload
          loop-assign induction-assign) ) )
  (do (push stack stat) ) )

*** Now iteratively mark the reaching definitions of STATs that are on the stack, pushing the reaching definitions onto the stack for eventual recursive marking. We never trace from INDUCTION-ASSIGNS (sigh).

(loop (while stack)
  (bind stat (pop stack) )
  (when (! (stat-set:member? visited stat) )
  (do (= visited (stat-set:union visited stat) )
    (loop (for-each-stat-set-element reaching-stat)
        (do (push stack reaching-stat) ) ) )

*** Delete all unmarked STATs.

(loop (for-each-stat-set-element
    (stat-set:difference (stat-set:universe) visited)
    stat)
  (rdc.stat:delete stat) )

*** Now delete any cond-jumps whose left and right branches go to the same spot.

(loop (for-each-stat)
  (when (stat:property? stat 'conditional-jump) )
  (bind succ-bblocks (bblock:succs (stat:bblock atat) )
    (when (|| (<= 2 (length succ-bblocks) )
        (<= 1 (length succ-bblocks) ) )
    (do (rdc.stat:delete stat)
        (= deleted-cond-jump? t) ) )

(deleted-cond-jump?)

(defun rdc.stat:delete ( stat )
  (assert (stat:is stat) )
  (let ( (bblock (stat:bblock stat) )
    (stat:delete stat)
    (if (bblock:empty? bblock) (then
      (bblock:delete bblock) )
    ( ) )
  )
)

::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::==::= 
Every NADDR operation in the flow-graph except GOTO and LABEL is represented by a STAT record. All STATs are numbered and are stored in an array that maps the numbers onto the STATs.

(de-struct stat
  source
  number
  (block () suppress)
  (succ () suppress)
  (pred () suppress)
  (known-derivation () suppress)
  (known-operand-derivations () suppress)
  (reaching-uses () suppress)
)

(STAT:CREATE SOURCE)
Creates a new STAT record for NADDR operation SOURCE. This is the only way STATs should be created.

(STAT:SUBSTITUTE-OPERAND STAT NEW-OPERAND OLD-OPERAND PART)
Creates a new STAT from the source of STAT, substituting NEW-OPERAND for OLD-OPERAND. See OPER:SUBSTITUTE-OPERAND for how the substitution occurs.

(STAT:EXTRACT STAT)
Splices out STAT from the flow graph. STAT is eligible to be placed somewhere else in the flow graph.

(STAT:DELETE STAT)
Extracts STAT from the flow graph and then forgets about it.

(STAT:INSERT-STAT STAT1 STAT2)
Inserts STAT2 before STAT1 in the flow graph.

(STAT:APPEND-STAT STAT1 STAT2)
Appends STAT2 after STAT1 in the flow graph.

(STAT:REPLACE-STAT STAT1 STAT2)
Replaces STAT1 by STAT2 in the flow graph.

(LOOP (FOR-EACH-STAT STAT) ...)
This is the only public way for enumerating through all the STATs of the flow graph. The enumeration is in order of the original NADDR source. Defined via LOOP’S DEF-SIMPLE-LOOP-CLAUSE.

(LOOP (FOR-EACH-STAT-OPERAND-READ STAT NAME) ...)
Enumerates NAME through each scalar and vector name read by by STAT.

(STAT:PREDS STAT)
(STAT:SUCCS STAT)
List of STATs that are flow predecessors/successors of STAT.

(STAT:OPERATOR STAT)
(STAT:GROUP STAT)
(STAT:PART STAT PART)
(STAT:PROPERTY? STAT PROPERTY)
Convenient functions that just invoke the equivalent functions on the NADDR source of the STAT.

(STAT:DEFINITION? STAT)
Returns true if STAT is a definition of a variable (or vector element).

(STAT:DOMINATES? STAT1 STAT2)
True if STAT1 dominates STAT2.

(include flow-analysis:flow-analysis-decls)

(defun fg.initialize-stats ()
  (vector-nap:initialize *fg.total-stats* *fg.nunber:stat* 200 t)
  (:= *stat.free-list* () )
  (:= *stat.free-list-reversed?* () )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed?*) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number index) )
    )
  )
  )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed?*) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number *)
    )
  )
  )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed* ) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number *)
    )
  )
  )
)

(declare (special
  *fg.total-stats* ;; Current number of STATs
  *fg.nunber:stat*  ;; Array for mapping STAT numbers onto STATs.
  *stat.free-list*  ;; List of free slots in *FG.NUMBER:STATS*.
  *stat.free-list-reversed?*  ;; True if above list has been reversed.
))

(Number:STAT NUMBER )
Returns the STAT with the given NUMBER. Try not to use this.
Deleted STATs return ()

(*SS number).
This is syntax for easy interactive access to particular STATs.
*SS 30 references STAT number 30.

(STAT:PREDs STAT)
(STAT:SUCCS STAT)
List of STATs that are flow predecessors/successors of STAT.

(STAT:OPERATOR STAT)
(STAT:GROUP STAT)
(STAT:PART STAT PART)
(STAT:PROPERTY? STAT PROPERTY)
Convenient functions that just invoke the equivalent functions on the NADDR source of the STAT.

(STAT:DEFINITION? STAT)
Returns true if STAT is a definition of a variable (or vector element).

(STAT:DOMINATES? STAT1 STAT2)
True if STAT1 dominates STAT2.

(Include flow-analysis:flow-analysis-decls)

(declare (special
  *fg.total-stats* ;; Current number of STATs
  *fg.nunber:stat*  ;; Array for mapping STAT numbers onto STATs.
  *stat.free-list*  ;; List of free slots in *FG.NUMBER:STATS*.
  *stat.free-list-reversed?*  ;; True if above list has been reversed.
))

(defun fg.initialize-stats ()
  (vector-nap:initialize *fg.number:stat* *fg.total-stats* 200 t)
  (:= *stat.free-list* )
  (:= *stat.free-list-reversed?* )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed?*) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number *)
    )
  )
  )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed* ) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number *)
    )
  )
  )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed* ) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number *)
    )
  )
  )
)

(defun stat:create ( source )
  (if *stat.free-list* (then
    (if (! *stat.free-list-reversed* ) (then
      (:= *stat.free-list* (reverse *stat.free-list*) )
      (:= *stat.free-list-reversed?* t) )
    (let ( (index (pop *stat.free-list*) ) )
      (:= (stat:new source source number *)
    )
  )
  )
)
(defun stat:substitute-operand (stat new-operand old-operand part)
  (assert (stat:is-stat stat))
  (stat:create (oper:substitute-operand)
    (stat:source stat)
    new-operand
    old-operand
    part))

(defun stat:extract (stat)
  (assert (stat:is-stat stat))
  (let ((bblock (stat:bblock-stat stat))
        (succ-stat (stat:succ-stat stat))
        (pred-stat (stat:pred-stat stat)))
    ;; Remove STAT from the doubly linked list of STATs in
    ;; the BBLOCK.
    (if succ-stat
        (:= (stat:pred succ-stat) pred-stat)
        (else (:= (bblock:first-stat bblock) succ-stat))
    (:= (stat:bblock-stat stat) bblock))
    (:= (stat:pred-stat stat) [])
    (:= (stat:succ-stat stat) []))

(defun stat:delete (stat)
  (assert (stat:is-stat stat))
  (stat:extract-stat stat)
  ;; Delete this STAT from the array of STATs.
  (:= ([] «fg.number:stat* (stat:NUMBER-stat stat) () ) [])
  (if ^^stat:free-list-reversed? (then
      (:= ^^stat:free-list (reverse ^^stat:free-list*))
    (push ^^stat:free-list (stat:NUMBER-stat stat))
  )

(defun stat:insert-stat (stat1 stat2)
  (assert (stat:is-stat stat1))
  (assert (stat:is-stat stat2))
  (let ((bblock (stat:bblock-stat stat1))
        (aucc-stat (stat:succ-stat stat1))
        (pred-stat (stat:pred-stat stat1)))
    (:= (stat:succ-stat stat1) aucc-stat)
    (:= (stat:pred-stat stat1) pred-stat)
    (if aucc-stat (then
        (:= (stat:pred-aucc-stat aucc-stat) stat2)
      (else
        (:= (bblock:first-stat bblock) stat2)
      )
    (:= (stat:bblock-stat stat2) bblock))
    (:= (stat:pred-stat stat2) [])
    (:= (stat:succ-stat stat2) [])
    )

(defun stat:append-stat (stat1 stat2)
  (assert (stat:is-stat stat1))
  (assert (stat:is-stat stat2))
  (let ((bblock (stat:bblock-stat stat1))
        (aucc-stat (stat:succ-stat stat1))
        (pred-stat (stat:pred-stat stat1)))
    (:= (stat:succ-stat stat1) aucc-stat)
    (:= (stat:pred-stat stat2) stat1)
    (if aucc-stat (then
        (:= (stat:pred-aucc-stat aucc-stat) stat2)
      (else
        (:= (bblock:last-stat bblock) stat2)
      )
    (:= (stat:bblock-stat stat2) bblock))
    (:= (stat:pred-stat stat2) [])
    (:= (stat:succ-stat stat2) [])
    )

(defun stat:replace-stat (stat1 stat2)
  (assert (stat:is-stat stat1))
  (assert (stat:is-stat stat2))
  (stat:insert-stat stat1 stat2)
  (stat:delete-stat stat1)

(defun simple-loop-clause for-each-stat (clause)
  (let ((for-each-stat var clause)
        (Index (intern (gensym) ) )
    (lf (! (ftft (= 2 (length clause) )
        (lltatoa var) ) )
      (error (list clause "Invalid FOR-EACH-STAT syntax."))
    )
  (initial ,var ()
    (incr .Index from 0 to (+ -1 «fg.total-stats«)
      (next ,var ([] «fg.number:atat* .index) )
    (when ,var) )
  )

(defun simple-loop-clause for-each-stat-operand-read (clause)
  (let ((for-each-stat-operand-read (stat:source ,stat var .index) )
    (for-each-oper-operand-read (stat:source ,stat) ,var .index) )
  )

(defmacro number:stat (number)
  "([] *fg.number:stat* .number)"
)

(defmacro number:stat-read (number)
  "([] *fg.number:stat* .read)"
)

(defun stat:preds (stat)
(assert (stat:is stat) )
(let ( (bblock (stat:bblock stat) ) )
  (if (= stat (bblock:first-stat bblock) ) (then
    (loop (for pred-bblock In (bblock:preds bblock) ) (save
      (bblock:last-stat pred-bblock) )
    )
    (else
      (list (stat:pred stat) )
    )
  )
)

(defun stat:succ (stat)
  (assert (stat:is stat) )
  (let ( (bblock (stat:bblock stat) ) )
    (if (= stat (bblock:last-stat bblock) ) (then
      (loop (for succ-bblock in (bblock:succs bblock) ) (save
        (bblock:first-stat succ-bblock) )
    )
    (else
      (list (stat:succ stat) )
    )
  )
)

(defun stat:operator (stat)
  (assert (stat:is stat) )
  (oper:operator (stat:source stat) )
)

(defun stat:group (stat)
  (assert (stat:is stat) )
  (oper:group (stat:source stat) )
)

(defun stat:part (stat part)
  (assert (stat:part stat part) )
  (oper:part (stat:source stat) part) )

(defun stat:property? (stat property)
  (assert (stat:is-stat stat) )
  (oper:property? (stat:source-stat stat) property) )

(defun stat:definition? (stat)
  (stat:part stat 'written) )

(defun stat:dominates? (stat1 stat2)
  (assert (stat:is-stat stat1) )
  (assert (stat:is-stat stat2) )
  (bblock:dominates? (stat:block-stat stat1) (stat:block-stat stat2) )
)
STAT-SETS
Sets of STATs are represented using STAT-SETS, currently implemented as BIT-SETs.

*FG.EMPTY-STAT-SET*
The empty STAT-SET.

(STAT-SET:UNIVERSE)
The set of all STATs.

(STAT-SET:SINGLETON STAT)
Creates a new set containing STAT.

(STAT-SET:MEMBER? SET STAT)
Returns true if STAT is a member of SET.

(STAT-SET:INTERSECTION SET1 SET2 ...)
Returns a new set that is the intersection of all the given sets.

(STAT-SET:UNION SET1 SET2 ...)
Returns a new set that is the union of all the given sets.

(STAT-SET:UNION1 SET STAT)
Unions a single STAT into SET.

(STAT-SET:DIFERENCE SET1 SET2)
Returns a new set that contains all elements in SET1 not in SET2.

(STAT-SET:= SET1 SET2)
Returns true if the two sets are equal.

(STAT-SET:SIZE SET)
Returns the number of elements in the set.

(STAT-SET:CHOOSE SET)
Returns the first element of the set, NIL if the set is empty.

(STAT-SET:CONTAINS? SET1 SET2)
True if SET1 contains SET2.

(LOOP (FOR-EACH-STAT-SET-ELEMENT SET STAT)
Enumerates STAT through each element in SET. Uses DEF-SIMPLE-LOOP-CLAUSE.

(STAT-SET:PRINT SET)
Prints SET by printing out the statement numbers.

(include flow-analysis:flow-analysis-decls)
(declare (special *fg.total-stats*) )
(declare *fg.empty-stat-set* () ) ;** the empty STAT-SET.
(defvar *fg.total-stats* )
(defun stat-set:universe ()
  (bit-set:universe *fg.total-stats* )
)
(defun stat-set:singleton ( stat )
  (bit-set:singleton (stat:number stat ) )
)
(defun stat-set:member? ( set stat )
  (bit-set:member? set (stat:number stat ) )
)
(defun stat-set:intersection args
  (apply 'bit-set:intersection (listify-lexpr-args args ) )
)
(defun stat-set:union args
  (apply 'bit-set:union (listify-lexpr-args args ) )
)
(defun stat-set:union1 ( set stat )
  (bit-set:union1 set (stat:number stat ) )
)
(defun stat-set:difference ( set1 set2 )
  (bit-set:difference set1 set2 )
)
(defun stat-set:= ( set1 set2 )
  (bit-set:= set1 set2 )
)
(defun stat-set:size ( set )
  (bit-set:size set )
)
(defun stat-set:contains? ( set1 set2 )
  (bit-set:contains? set1 set2 )
)
(defun stat-set:choose ( set )
  (if-let ( (index (bit-set:choose set) ) )
    (number:stat index)
    ()
  )
)
(defun stat-set:contains? ( set1 set2 )
  (bit-set:contains? set1 set2 )
)
(def-simple-loop-clause for-each-stat-set-element ( clause )
  (let (( (for-each-stat-set-element set stat) clause)
    (index (intern (gensys) ) )
    (if (! (&& (= 3 (length clause) )
          (let (stat) ) ) )
      (error (list clause "Invalid FOR-EACH-STAT-SET-ELEMENT syntax."))) ) )
  '
    (initial ,stat ()
      (for-each-bit-set-element ,set ,index)
      (next ,stat (number:stat ,index)
        (when ,stat ) ) )
  )
)
(defun stat-set:print ( set )
  (bit-set:print set )
)

This module prints miscellaneous statistics about the flow graph.

(FG.PRINT-STATISTICS)
Prints out whatever statistics we want today.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(include flow-analysis:flow-analysis-decls)

(defun fg.print-statistics ()
  (msg 0)
  (msg "# of iterations in REACHING-DEFS: "
       +fg.number-of-reaching-iterations*
       t t)
  (msg "# of BBLOCKS: " +fg.total-bblocks* t)
  (msg "# of STATS: " +fg.total-statss* t)
  (msg "#STATS//BBLOCK:"
   (// (flonum +fg.total-stats*) (flonum +fg.total-bblocks*)) t)
  (loop (for-each-stat stat)
          (initial ud-length 0)
          (do
            (loop (initial used-defs +fg.empty-stat-set*)
                  (for-each-stat-operand-read stat name)
                  (next used-defs (stat-set:union used-defs
                                    (name:defining-stats name) ))
            (result
             (= :ud-length
                 (+ :ud-length
                   (bit-set :size (bit-set:intersection
                                    used-defs
                                    (stat:reaching-defs stat)) ))))
            (result
             (msg "Average length of ud-chain//STAT: "
                  (// (flonum :ud-length) (flonum +fg.total-stats*)
                       t t) )
             )
          )
  )
)
(FG.TEMPORARY-NAME ROOT)

Returns a new, unshared unique temporary name of the form:

\$ ROOT - n

where n is a unique number. E.g. if \$ROOT = 'FOO' then the result
might be \$FOO-2. \$ROOT may be (), which is the same as the empty
string. If \$ROOT is of the form \$NAME-n, then \$NAME is used as the
the root. Thus if \$ROOT = 'FOO-49', the result might be \$FOO-100.

(FG.INITIALIZE-TEMPORARY-NAME)

Initializes the creation of temporary names by resetting the unique
number counter to 1.

(defvar *fg.temporary-name-counter* 0)

(defun fg.initialize-temporary-name ()
  (= *fg.temporary-name-counter* 0))

(defun fg.temporary-name ( root )
  (if (not root)
      (format nil "$• (•• *fg.temporary-name-counter*)
    (if (equal (aref root 1) $) (format nil "$" root "-" (•• *fg.temporary-name-counter*)
    (t (loop (incr i from 1 to (length root))
        (until (equal (aref root i) $))
        (result (format nil "$" (substring root (- 1 1))
      (= *fg.temporary-name-counter* (••)))))
    )
  )
)}
(FG.RENAME-VARIABLES)

Renames variables as much as possible to avoid false data dependencies. The uses and defs of a variable can be viewed as a bipartite graph. Every use and def is a separate node. There is an edge between a def and a use if the def reaches the use. Within each connected component of this graph, the variable can be given a new unique name. More precisely, the reads of the variable in uses and the writes of the variables within defs may be changed to use a new variable.

We don't actually construct the graph, but instead pick a def of a variable and then trace out the connected component use-def chain.

(eval-when (compile load)
 (include flow-analysis:flow-analysis-decls) )

(defun fg.rename-variables ()
  (let ( (renamings () ) )
    ;; A list of triples of the form:
    ;;   (STAT-SET NEW-NAME OLD-NAME READ-OR-WRITTEN)
    ;;   that specifies that NEW-NAME should be substituted for
    ;;   OLD-NAME in the READ or WRITTEN part of every STAT in
    ;;   STAT-SET. We construct the whole set of renamings for
    ;;   the program before doing any renaming; otherwise
    ;; STAT:REACHING-DEFS will get horribly confused by a
    ;; partially renamed program.
    ;;
    ;; For each NAME do:
    (loop (for-each-name name)
      (when (= 'scalar (name:type name) )
        (bind all-visited-defs 'fg.empty-stat-set)
        (do
          ;; Pick an unvisited defining DEF-STAT of NAME.
          (loop (for-each-stat-set-element (name:definition-stat name) def-stat)
            (when (! (stat-set:member? visited-defs def-stat) )
              (bind trace-stack ()
                visited-uses 'fg.empty-stat-set
                visited-defs 'fg.empty-stat-set
                contains-pseudo-op ()
              )
              (do
                ;; Trace out the connected chain of uses and defs of
                ;; NAME containing DEF-STAT. To do the tracing, we
                ;; keep a stack trace-stack of pairs of the form (STAT
                ;; READ-OR-WRITTEN), where STAT is a possibly unvisited
                ;; use or def of NAME, and READ-OR-WRITTEN indicates
                ;; whether STAT is being considered as a use or def
                ;; of NAME. As the end of this tracing, VISITED-USES
                ;; and VISITED-DEFS will contain STAT-SETS of all the
                ;; uses and defs that are part of the connected train
                ;; that was traced out.
                push trace-stack '(
                  (def-stat written) )
                (loop (while trace-stack)
                  (bind (stat read-or-written) (pop trace-stack) )
                  (when (! (stat-set:member? visited-defs stat) )
                    ;; If this STAT is a special pseudo-op, remember
                    ;; it so as to disable renaming of this chain.
                    (if (eqq (stat:operator stat) 'def-block live paran) (then
                      (:= contains-pseudo-op? t) )
                    )
                  )
                  (do
                    ;; Mark the use or def as visited, and trace out
                    ;; reaching uses (defs) of this def (use).
                    (if (= 'written read-or-written) (then
                      (:= visited-defs (stat-set:union refl stat) )
                      (loop (for use-stat in (stat:reaching-uses stat) ) (do
                        (push trace-stack '(
                          (use-stat . use-stat read) ) )
                      ))
                    )
                    (else
                      (:= visited-uses (stat-set:union refl stat) )
                      (loop (for-each-stat-set-element
                        (stat:operand-reaching-defs stat name)
                        reaching-def-stat
                      ) (do
                        (push trace-stack '(
                          (reaching-def-stat written) )
                        )))
                    )
                  )
                )
              )))
            )))
          )))
          ;; The tracing is finished. Remember all the visited
          ;; defs of NAME.
          (:= all-visited-defs (stat-set:union refl visited-defs) )
          ;; Make an entry on the RENAMINGS list if this is not the
          ;; first chain of NAME and the chain contains no pseudo-op.
          (if (and (> (++ name-counter) 1)
            (contains-pseudo-op?)
            (then
              (let ( (new-name (fg.temporary-name name) ) )
                (push renamings '(
                  visited-uses . new-name . name read
                )
                (push renamings '(
                  visited-defs . new-name . name written
                )))
              )))
          )))
      )))
    )))
    ;; All the partitions (chains) of every NAME have been found.
    ;; Now do the actual renaming as specified by the RENAMINGS list.
    (loop (for (stat-set new-name old-name read-or-written) in renamings) (do
      (loop (for-each-stat-set-element stat-set stat)
        (do
          (:= (stat:source-stat)
            (oper:substitute-operand (stat:source-stat)
              new-name
              old-name
              read-or-written ) )
        ))
      )))
  )))
)

PS:<C.S.BULLDOG.FLOW-ANALYSIS>VARIABLE-RENAMING.ISP.1