SETL NEWSLETTER 195

# AN ALGORITHM FOR COPY OPTIMIZATION, BASED ON NL176.

MICHA SHARIR
JULY 11 1977

THIS COPY OPTIMIZATION ALGORITHM IS A FLOW ANALYSIS OF ≠SHADOW VARIABLES≠, AS DEFINED IN NL. 176. IT LOCATES THE POTENTIALLY DESTRUCTIVE USES AND INSTRUCTIONS WHERE SHARE BITS ARE POSSIBLY SET. THEN, BY A STRAIGHT FORWARD TECHNIQUE WHICH COMBINES LIVE/DEAD INFORMATION AND CONSTANT PROPAGATION FOR THESE SHARE BIT VARIABLES, WE MOVE COPIES OUT OF LOOPS, SUPRESS SETTINGS OF SHARE BITS, AND FIND ALL UNCONDITIONAL COPIES AND NO-COPIES.

ALTHOUGH IT IS POSSIBLE TO PERFORM THIS COPY OPTIMIZATION BY AN INTERVAL-ORIENTED ALGORITHM, WITH TWO PASSES THROUGH THE DERIVED SEQUENCE, IT SEEMS BEST TO DO IT IN THE STRAIGHT-FORWARD WAY EXPLAINED BELOW, FOR THE FOLLOWING REASONS,

A) MUCH OF THE DATA THAT OUR ALGORITHM USES IS ALREADY CONTAINED IN THE BFROM MAP, WHICH WAS INDEED OBTAINED BY SUCH AN INTERVAL ORIENTED ALGORITHM.

B) IT IS NOT CLEAR WHICH OF THESE TWO ALGORITHMS WILL BE MORE EFFICIENT. THE ALGORITHM GIVEN BELOW IS ALMOST LINEAR IN THE NUMBER OF POTENTIALLY DESTRUCTIVE USES, AND THE INTERVAL ORIENTED ALGORITHM REQUIRES A TIME THAT CONTAINS AN ADDITIONAL FACTOR ROUGHLY PROPORTIONAL TO THE NUMBER OF BASIC BLOCKS IN THE PROGRAM, AND SO MIGHT BE LESS EFFICIENT, ESPECIALLY IF WE ANTICIPATE THE NUMBER OF DESTRUCTIVE USES TO BE A RATHER SMALL PORTION OF THE PROGRAM LENGTH.

C) OUR ALGORITHM ALLOWS US TO PERFORM THE COPY OPTIMIZATION INTER-PROCEDURALLY. THE INTERVAL-ORIENTED ONE IS NOT SUITABLE FOR INTER-PROCEDURAL ANALYSIS.

WE FIRST BEGIN BY SCANNING THE CODE, LOCATING ALL POSSIBLE INSTRUCTIONS WHICH MIGHT ≠DEFINE≠ OR ≠USE≠ THE SHARE BITS.
A ≠DEFINITION≠ IS EITHER A VALUE CREATING INSTRUCTION, WHICH ASSIGNS 0 TO THE SHARE BIT OF ITS OVARIABLE (MEANING THAT THIS NEW VALUE IS DEFINITELY NOT SHARED), OR IS A VALUE TRANSMITTING, RETRIEVING OR EMBEDDING INSTRUCTION, SUCH AS A SIMPLE ASSIGNMENT, MAP RETRIEVAL, SET INSERTION ETC. , ALL OF WHICH USUALLY ASSIGN ≠1≠ TO THE SHARE BIT OF ONE OR MORE OF THEIR VARIABLES, MEANING THAT THE VALUE OF THIS VARIABLE BECOMES SHARED AFTER THIS INSTRUCTION.
A ≠USE≠ OF THE SHARE BIT IS A POTENTIALLY DESTRUCTIVE USE OF THE CORRESPONDING VARIABLE.

AN ADDITIONAL COMPLICATION ARISES IN THE CASE OF SIMPLE ASSIGNMENTS AND ASSIGNMENT-LIKE OPERATIONS SUCH AS ARGIN AND ARGOUT. AN ASSIGNMENT X = Y SHOULD SET THE SHARE BIT OF X, UNLESS Y IS BOTH DEAD AND UNSHARED AT THIS POINT. THE ORIGINAL APPROACH, SUGGESTED BY J. SCHWARTZ IN NL. 176, IS TO SET THE SHARE BIT OF X UNCONDITIONALLY AT THIS ASSIGNMENT. HOWEVER, THERE ARE TWO VERY FREQUENT CASES FOR WHICH THIS OVER-ESTIMATION MIGHT PROPAGATE A CONSIDERABLE AMOUNT OF UNNECESSARY COPY OPERATIONS.

THE FIRST CASE CONCERNS RETURN VALUES OF FUNCTIONS AND WRITE PARAMETERS. AS CURRENTLY DESIGNED, THE VALUE TRANSFERS FROM THE FORMAL WRITE PARAMETERS AND THE RETURN VALUE TO THE ACTUAL ARGUMENTS ARE PRESENTED EXPLICITLY IN THE CODE BY ARGOUT ASSIGNMENTS FOLLOWING THE CALL. THE PREVIOUS APPROACH WOULD IMPLY THAT THESE VALUES WILL ALWAYS BE JUDGED TO BE SHARED, WHEREAS IN MOST CASES THEY ARE NOT, AS THE FORMAL RETURN VALUE ACTS ONLY AS A TRANSFER MEDIUM FOR THIS VALUE.

THE OTHER, MUCH MORE FREQUENT, CASE CONCERNS TEMPORARIES. ANY EXPRESSION COMPUTATION, SUCH AS A = B + C WILL EXPAND INTO T = B + C; A = T; WHERE T IS A TEMPORARY NAME UNIQUELY REPRESENTING THE COMPUTATION B + C. UNLESS B + C IS A COMMON SUBEXPRESSION WHICH WILL LATER BE USED WITHOUT BEING RE-DEFINED (AFTER THE COMMON SUBEXPRESSION ELIMINATION PHASE), THERE IS NO NEED TO SET THE SHARE BIT OF A, SINCE IN THE OTHER CASES T WILL BE AN UNSHARED (BEING CREATED JUST NOW) AND DEAD VARIABLE AT THE ASSIGNMENT. THE PREVIOUS APPROACH WOULD THEREFORE IMPLY THAT ALL NEWLY CREATED VALUES WOULD IMMEDIATELY BE JUDGED TO BE SHARED, AN ASSUMPTION WHICH WILL NEGATE MOST OF THE GAINS OF COPY OPTIMIZATION.

WE SHALL THEREFORE TREAT ASSIGNMENTS IN A SPECIAL WAY IN OUR ALGORITHM, CONSIDERING THEM AS PARTLY TRANSMITTING, PARTLY SETTING THE SHARE BIT. THAT MEANS THAT WE SHALL SOMETIMES TRACE THE EARLIEST POINTS OF SETTING THE SHARE BIT, CROSSING ASSIGNMENTS WHILE DOING SO.

IN THE FIRST SCAN OF THE CODE WE COMPUTE THE FOLLOWING OBJECTS.

SHAREBIT - A MAP ON OCCURENCES, WITH VALUES 0 OR 1 IF AN OCCURENCE TURNS OFF OR ON THE SHARE BIT OF ITS VARIABLE, AND WITH VALUE 2 IF THE OCCURENCE IS AN OVARIABLE OF AN ASSIGNMENT FOR WHICH THE IVARIABLE IS DEAD (AND THEREFORE THIS ASSIGNMENT ACTS AS A TRANSMISSION OF THE SHARE BIT FROM THE IVARIABLE TO THE OVARIABLE).

OTHERWISE SHAREBIT OF AN OCCURENCE IS UNDEFINED, INDICATING THAT THIS OCCURENCE DOES NOT AFFECT THE SHARE BIT.

SHAREBITUSE - ALL IVARIABLE OCCURENCES WHICH APPEAR IN A POTENTIALLY DESTRUCTIVE USE.

AFTER THIS PRELIMINARY PASS, WE ITERATE OVER ALL IVARIABLES IV WHICH REPRESENT A USE OF THE SHARE BIT OF SOME VARIABLE V. FOR EACH SUCH OCCURENCE, WE FIND THE LARGEST GROSS INTERVAL I CONTAINING IV, SUCH THAT THE SHARE BIT OF V WAS NEVER SET IN I WITHOUT BEING DROPPED BEFORE IV WAS REACHED. IF I IS THE ENTIRE PROGRAM THEN NO COPY IS REQUIRED, AND WE SET COPYFLAG(IV) TO COPY¬NO. IF THERE IS NO SUCH I, THEN EITHER A COPY WILL ALWAYS BE REQUIRED, IF ALL PREVIOUS OCCURENCES OF THE CURRENT VALUE OF V SET THE SHARE BIT (AND THEN WE SET COPYFLAG(IV) TO COPY¬PRE), OR A RUN TIME TEST OF THE SHARE BIT IS NEEDED, AND THEN WE SET COPYFLAG(IV) TO COPY¬TEST. IN ALL OTHER CASES, WE MOVE THE COPY TO THE TARGET BLOCK OF I. HOWEVER, THIS MOVED COPY CAN STILL BE EITHER CONDITIONAL OR UNCONDITIONAL, DEPENDING ON WHETHER THERE WAS OR THERE WAS NOT A NEW CREATION OF THIS VALUE PRIOR TO THIS INTERVAL.

IN ORDER TO FIND THE INTERVAL I, WE COMPUTE A SELECTIVE TRANSITIVE CLOSURE SET, USING THE BFROM MAP AND ASSIGNMENTS WHICH ACT AS A SHARE BIT TRANSFER. MORE PRECISELY, WE FIND THE SET S OF ALL PREVIOUS OCCURENCES OI OF THE VALUE OF IV, FROM WHICH IV CAN BE REACHED VIA A PATH, SUCH THAT ALL OTHER OCCURENCES OF THIS VALUE ALONG THIS PATH DO NOT SET THE SHARE BIT OF THIS VALUE (BUT MAY TRANSFER IT FROM ONE VARIABLE TO ANOTHER). (THIS CAN BE COMPARED TO THE CRTHIS MAP OF THE ≠SHADOW VARIABLE OCCURENCE≠ CORRESPONDING TO THE SHARE BIT OF IV).

THEN, FOR EACH OIV IN S WHICH SETS THE SHARE BIT, WE DETERMINE THE LARGEST GROSS INTERVAL I WHICH CONTAINS IV BUT NOT OIV; THE MINIMUM OF THESE INTERVALS I IS THE REQUIRED INTERVAL.

TO DETERMINE WHETHER THE COPY SHOULD BE CONDITIONAL OR UNCONDITIONAL, WE ALSO CALCULATE AN AUXILIARY INTERVAL I1 WHICH IS THE SMALLEST INTERVAL CONTAINING IV AND ALL THE PREVIOUS NEW VALUE CREATIONS OF THIS VALUE. THE COPY SHOULD BE CONDITIONAL IF AND ONLY IF I1 STRICTLY CONTAINS I. I1 IS COMPUTED IN A SIMILAR WAY TO THE COMPUTATION OF I.

A SECOND RELATED OPTIMIZATION, WHICH IS SOMEWHAT MARGINAL, IS TO SUPPRESS A SETTING OF THE SHARE BIT IF THIS BIT IS NEVER GOING TO BE USED, I.E. - THERE IS NO SUBSEQUENT POTENTIALLY DESTRUCTIVE USE OF THE CORRESPONDING VALUE.

SINCE IN THE ABSENCE OF ANY FLAG INFORMATION CONCERNING SHARE BITS, THE CODE GENERATOR WILL NOT TOUCH THE SHARE BITS AT ALL, WE HAVE TO ASSIGN COPY¬SET TO COPYFLAG(IV) IF AND ONLY IF THE INSTRUCTION OF IV ESSENTIALLY SETS THIS BIT, AND THERE IS A SUBSEQUENT USE OF THE SHARE BIT OF THE CORRESPONDING VALUE. (BY ≠SUBSEQUENT≠ WE MEAN A SELECTIVE TRANSITIVE CLOSURE, USING THE FFROM MAP AND SHARE BIT TRANSFER ASSIGNMENTS, COMPLETELY ANALOGOUS TO THE ABOVE MENTIONED CLOSURE OF BFROM.)

NOTE THAT THE SECOND PHASE IS INDEPENDENT OF THE FIRST, EVEN THOUGH COPIES MIGHT HAVE BEEN MOVED OUT OF INTERVALS IN THE FIRST PHASE, WHICH MIGHT HAVE CHANGED THE CRTHISINV MAP OF THE SHADOW VARIABLES. HOWEVER, IT IS EASILY CHECKED THAT A SHARE BIT WAS SET PRIOR TO EXECUTING THE COPY IF AND ONLY IF IT WAS SET PRIOR TO THE DESTRUCTIVE USE (UNLESS IT WAS LATER DROPPED BEFORE THIS DESTRUCTIVE USE, IN WHICH CASE WE DO NOT CONSIDER THIS SETTING AT ALL).

MOREOVER, IT CAN BE SHOWN THAT THE ONLY CASE OF INSERTING A REDUNDANT COPY OPERATION, IS THE INSERTION OF TWO IDENTICAL COPY OPERATIONS INTO THE TARGET BLOCK OF THE SAME INTERVAL. INDEED, IF TWO COPY OPERATIONS OF THE SAME VARIABLE ARE FOUND TO BE INSERTTED INTO THE TARGET BLOCKS OF TWO INTERVALS $I_1$ AND $I_2$, THEN EITHER $I_1 = I_2$, OR NEITHER OF THEM CONTAINS THE OTHER, OR IF $I_1$, SAY, CONTAINS $I_2$ THEN THE OCCURENCE WHICH INDUCED THE COPY AT THE ENTRY OF $I_1$ IS IN A LOOP-FREE PART OF $I_1$, BEFORE THE ENTRY OF $I_2$, SO THAT BOTH COPY OPERATIONS ARE NOT REDUNDANT.

WE MIGHT EVEN IMPROVE THIS OPTIMIZATION, BY SUPPRESSING THE SETTING OF THE SHARE BIT, IF NONE OF ITS SUBSEQUENT USES HAVE THEIR COPYFLAG = COPY¬TEST, WHICH MEANS THAT THE SHARE BIT WILL NEVER BE TESTED, SINCE ALL SUBSEQUENT COPIES OF THIS VARIABLE ARE NOW EXPLICIT IN THE CODE.

FOR THE SAKE OF EFFICIENCY, WE INSERT THIS OPTIMIZATION INTO THE MAIN ITERATION OVER THE SHARE BIT USES. SPECIFICALLY, WHENEVER WE FIND A USE WHOSE COPYFLAG HAS BEEN SET TO COPY¬TEST, THEN, AND ONLY THEN, WE EXPLICITLY SET THE SHARE BIT OF ALL PRIOR OCCURENCES WHICH ESSENTIALLY SET THIS BIT.

THE IMPLEMENTATION OF THIS ALGORITHM IS VERY STRAIGHT FORWARD. FOR THE SAKE OF COMPLETENESS WE INCLUDE HERE A SETL VERSION OF IT. LATER WE SHALL GIVE SOME EXAMPLES OF COPY OPTIMIZATIONS THAT ARE CAUGHT BY THIS ALGORITHM, AND SOME THAT ARE NOT CAUGHT.

```
      MODULE COPYOP;

      DEFINE COPY-OPTIMIZE PUBLIC;
  $ THIS IS THE MAIN DRIVING ROUTINE OF THIS ALGORITHM.
  $ IT INVOKES THE FOLLOWING PROCEDURES,

  $ SCAN-SHARES - THE FIRST SCAN THROUGH THE CODE TO LOCATE
  $ DEFINITIONS AND USES OF THE SHARE BITS.

  $ FIND-COPIES - ANALYZING EACH POTENTIALLY DESTRUCTIVE USE
  $ FOR THE REQUIRED COPY ACTIONS.

      SCAN-SHARES();

      FIND-COPIES();

      RETURN;
      END COPY-OPTIMIZE;


      DEFINE SCAN-SHARES;
  $ FIRST SCAN THROUGH THE CODE TO LOCATE DEFINITIONS AND USES OF
  $ THE SHARE BITS.

      COPYFLAG := NL;
  $ COPYFLAG IS A MAP ON OCCURENCES, INDICATING WHAT COPY ACTION
  $ SHOULD BE TAKEN AT EACH OCCURENCE. THE RANGE OF COPYFLAG
  $ CONTAINS THE FOLLOWING POSSIBLE VALUES,

  $ COPY-NO - NO COPY IS REQUIRED.

  $ COPY-PRE - COPY BEFORE THE INSTRUCTION.

  $ COPY-TEST - COPY BEFORE THE INSTRUCTION IF THE CORRESPONDING
  $ SHARE BIT IS ON, OTHERWISE DO NOT COPY.

  $ COPY-SET - SET THE SHARE BIT.

      SHAREBIT := NL;
  $ SHAREBIT IS A MAP ON OCCURENCES, HAVING THE VALUE 0 IF
  $ THE CORRESPONDING VARIABLE GETS A NEW UNSHARED VALUE, 1
  $ IF THE VALUE OF THE VARIABLE BECOMES SHARED AT THIS
  $ INSTRUCTION, 2 IF THIS IS A SHARE BIT TRANSFER BY AN ASSIGNMENT,
  $ AND UNDEFINED OTHERWISE.

      SHAREBITUSE := NL;
  $ SHAREBITUSE IS THE SET OF ALL POTENTIALLY DESTRUCTIVE USES.
```

```
      (∀B → BLOCKS, CODE(B,I))
$ ITERATE OVER THE CODE
          OV := GET¬OVAR(I);
$ OV IS THE OVARIABLE
          CASE OPCODE(I) OF

          (→ OPS¬ASSIGN) :
$ IN AN ASSIGNMENT, CHECK IF THE IVARIABLE IS DEAD
              IV := GET¬IVARS(I)(1) $ GET THE IVARIABLE
              IF FFROM≤IV≥ /= NL THEN
$ IF THE IVARIABLE IS LIVE, THEN BOTH IT AND THE OVARIABLE
$ SHARE THEIR VALUES.
                  SHAREBIT(OV) := 1;
                  SHAREBIT(IV) := 1;
              ELSE SHAREBIT(OV) := 2;
$ INDICATING A SHARE BIT TRANSFER (I.E. THE SHARE BIT OF OV
$ SHOULD BE SET IFF THE SHARE BIT OF IV IS SET AT THIS POINT).
              END IF;

          (→ OPS¬RETRIEVE + OPS¬EXTRACTS) :
$ IN A RETRIEVAL OR AN EXTRACTION, SET THE SHARE BIT OF
$ THE OVARIABLE.
              SHAREBIT(OV) := 1;

          (→ OPS¬NEWVAL) :
$ IN A VALUE CREATING INSTRUCTION, TURN OFF THE SHARE BIT
$ OF THE OVARIABLE.
              SHAREBIT(OV) := 0;

          (→ OPS¬DESTRUCT) :
$ IN A POTENTIALLY DESTRUCTIVE USE, THE OVARIABLE IS NEVER
$ SHARED, HAVING A NEW VALUE. IF THE DESTRUCTIVE USE IS LOCAL
$ IN NATURE (NAMELY, A MODIFICATION OF A GROSS OBJECT BY INSERTING
$ OR DELETING AN ELEMENT) THEN THIS ELEMENT VALUE BECOMES SHARED.
$ IN ANY CASE, THE SHARE BIT OF THE GROSS OBJECT IS ≠USED≠.
              SHAREBIT(OV) := 0;
              SHAREBITUSE WITH LARGE¬OBJ(I);
              IF OPCODE(I) IN OPS¬LOCAL THEN
                  SHAREBIT(SMALL¬OBJ(I)) = 1;;

          END CASE;
      END ∀;

$ FIND FOR ALL OCCURENCES WHOSE SHARE BIT IS 2 WHETHER THEY SHOULD
$ BE TREATED AS A SETTING OF THE SHARE BIT, OR A DROP, OR BOTH.
      PREVSETS := PREVDROPS := NL;
$ PREVSETS IS THE SET OF ALL PAIRS [OIV,RCS], WHERE OIV
$ IS AN OVARIABLE IN A SHARE BIT TRANSFER ASSIGNMENT, AND
$ RCS IS A RC¬STRING LEADING FROM A PREVIOUS SETTING OF THE
$ SHARE BIT OF THE TRANSFERRED VALUE TO THIS ASSIGNMENT.
$ PREVDROP IS THE SAME, EXCEPT FOR A DROP INSTEAD OF A SETTING.

      (∨ OIV → DOM(SHAREBIT) ↑ SHAREBIT(OIV) = 2)
          SEEN := ≤ OIV ≥;
          WORK := ≤ [NULL¬PATH,OIV] ≥;
          (WHILE WORK /= NL)
              [STR,OV] FROM WORK;
              IV := IVARN(OV,2);
              PREVS := ≤ [STR2,OIV1] : [STR1,OIV1] → BFROM(IV) ↑
                  OIV1 NOTIN SEEN AND
                  (STR2 := STR1 CC, STR) /= ERROR¬PATH ≥;
              CHECKED := NL;
              (WHILE PREVS /= NL)
                  [STR2,OIV1] FROM PREVS;
                  CHECKED WITH [STR2,OIV1];
                  CASE SHAREBIT(OIV1) OF

              (OM) :
                  PREVS + ≤ [STR3,OIV2] : [STR0,OIV2] → BFROM(OIV1)
                          ↑ (STR3 := STR0 CC, STR2) /= ERROR¬PATH ≥
                        - CHECKED;
              (0) :
                  PREVDROPS WITH [OIV,STR2];
              (1) :
                  PREVSETS WITH [OIV,STR2];
              (2) :
                  SEEN WITH OIV1;
                  WORK WITH [STR2,OIV1];

                  END CASE;
              END WHILE;
          END WHILE;
      END ∨;

      RETURN;
      END SCAN¬SHARES;


      DEFINE FIND¬COPIES;
$ MAIN ROUTINE - ITERATE OVER THE USES OF THE SHARE BITS.

      (∨IV→SHAREBITUSE)

$ WE NOW COMPUTE INTSEQ = THE TUPLE OF ALL THE INTERVALS
$ CONTAINING IV.
          INTSEQ := [B : (FOR B := INTOV(BLOCKOF(INSTNO(IV)));
                              WHILE B /= OM DOING B := INTOV(B))];
```

```
        INTZERO := 0;
$ INTZERO IS THE SMALLEST INTERVAL CONTAINING IV SUCH THAT
$ ALL ZERO SETTINGS (NEW VALUE CREATIONS) TO THE SHARE BIT OF
$ THE VALUE OF IV, DONE PRIOR TO IV, ARE IN THIS INTERVAL.

        INTONE := #INTSEQ + 1;
$ INTONE IS THE LARGEST INTERVAL CONTAINING IV SUCH THAT ALL
$ SETTINGS OF THE SHARE BIT OF THE VALUE OF IV WERE DONE
$ OUTSIDE OF THIS INTERVAL.

        PREVS := BFROM(IV) LESS [NULL-PATH,IV];
        PREV-BITSETS := NL;
$ PREV-BITSETS IS THE SET OF ALL PREVIOUS OCCURENCES OF THIS
$ VARIABLE, WHICH SET THE SHARE BIT. WE KEEP THEM TO INSERT IN
$ THEM EXPLICITLY THE COPY-SET ACTION, SHOULD A COPY-TEST ACTION
$ BE REQUIRED AT IV.
        CHECKED := NL;

        (WHILE PREVS /= NL)
            [STR,OIV] FROM PREVS;
            CHECKED WITH [STR,OIV];

            CASE SHAREBIT(OIV) OF

        (OM) :
            PREVS + ≤ [STR1,OIV1] : [STR0,OIV1] ∈ BFROM(OIV) |
                    (STR1 := STR0 CC, STR) /= ERROR-PATH ≥
                  - CHECKED;

        (0) :
            INTZERO := INTMIN(OIV,STR,INTZERO);
$ INTMIN RETURNS THE SMALLEST INDEX I IN INTSEQ SUCH THAT
$ I ≥ INTZERO AND OIV IN INTSEQ(I).

        (1) :
            INTONE := INTMAX(OIV,STR,INTONE);
            PREV-BITSETS WITH OIV;
$ INTMAX RETURNS THE LARGEST INDEX I IN INTSEQ SUCH THAT
$ I ≤ INTONE AND OIV NOTIN INTSEQ(I).

        (2) :
$ A PREVIOUS TRANSFER ASSIGNMENT, CHECK IF THERE IS A RC-STRING
$ LEADING TO THIS ASSIGNMENT FROM A SETTING (DROP) OF THE SHARE BIT,
$ WHICH IS COMPATIBLE WITH THE CURRENT RC-STRING. IF SO, REGARD
$ THIS ASSIGNMENT AS A SETTING (DROP) OF THE SHARE BIT. NOTE THAT
$ THIS ASSIGNMENT MAY BE REGARDED AS BOTH A SETTING AND A DROP
```

```
            IF (∃ STR1 → PREVSETS≤OIV≥ †
                    STR1 CC, STR /= ERROR¬PATH) THEN
                INTONE := INTMAX(OIV,STR,INTONE);
                PREV¬BITSETS WITH OIV;
            ELSEIF (∃ STR1 → PREVDROPS≤OIV≥ †
                    STR1 CC, STR /= ERROR¬PATH) THEN
                INTZERO := INTMIN(OIV,STR,INTZERO);
            END IF;

        END CASE;
    END WHILE;

$ NOW DETERMINE, ACCORDING TO INTZERO AND INTONE, THE COPY ACTION
$ TO BE DONE AT IV.

    IF INTONE > ↓INTSEQ THEN
$ NO PRIOR SETTING OF THE SHARE BIT, SO
        COPYFLAG(IV) := COPY¬NO;

    ELSEIF INTONE = 0 THEN
$ NO COPY MOTION IS POSSIBLE, CHECK IF CONDITIONAL OR UNCONDITIONAL
$ COPY IS REQUIRED.
        IF INTZERO /= 0 THEN
$ PRIOR NEW VALUE CREATION, A CONDITIONAL COPY
            COPYFLAG(IV) := COPY¬TEST;
$ AT THIS POINT, PRIOR SETTINGS OF THE SHARE BIT WERE INDEED
$ NECESSARY, SET THEM EXPLICITLY,
            (∀OIV → PREV¬BITSETS)
                COPYFLAG(OIV) := COPY¬SET;;

        ELSE COPYFLAG(IV) := COPY¬PRE; $ UNCONDITIONAL COPY
        END IF;

    ELSE  $ IN THIS CASE COPY MOTION IS POSSIBLE
        COPYFLAG(IV) := COPY¬NO;
        VI := OI¬NAME(IV);
        I := INS¬TARG(INTSEQ(INTONE),O1¬ASN,[VI,VI]);
$ THIS ROUTINE CHECKS WHETHER AN IDENTICAL COPY ASSIGNMENT
$ ALREADY EXISTS IN THE TARGET BLOCK OF THIS INTERVAL. IF NOT,
$ IT INSERTS THAT ASSIGNMENT INTO THIS BLOCK, AND RETURNS
$ THE NEW INSTRUCTION IDENTIFIER, OTHERWISE IT RETURNS OM.
        IF I /= OM THEN
            IF INTZERO > INTONE THEN
$ A NEW VALUE CREATION PRIOR TO THIS INTERVAL HEAD,
$ A CONDITIONAL COPY.
                COPYFLAG([I,2]) := COPY¬TEST;
$ AT THIS POINT, PRIOR SETTINGS OF THE SHARE BIT WERE INDEED
$ NECESSARY, SET THEM EXPLICITLY,
                (∨OIV → PREV¬BITSETS)
                    COPYFLAG(OIV) := COPY¬SET;;

            ELSE  $ ONLY SHARE BIT SETTINGS PRIOR TO THIS POINT.
                COPYFLAG([I,2]) := COPY¬PRE;
            END IF;
        END IF;
    END IF;

    END ∨IV;

    RETURN;
    END FIND¬COPIES;



    DEFINEF INTMAX(OI,RCS,IND);
$ THIS FUNCTION COMPUTES THE LARGEST INDEX I IN INTSEQ,
$ A TUPLE CONTAINING ALL INTERVALS CONTAINING A CERTAIN
$ OCCURENCE, SUCH THAT I ≤ IND  AND [OI,RCS] NOTIN INTSEQ(I).

    BEGIN

    IF IND = 0 THEN RETURN IND;
$ THERE ARE ALREADY ≠BAD≠ OCCURENCES INSIDE THE FIRST
$ ORDER INTERVAL CONTAINING THIS OCCURENCE, NOTHING TO DO.

    ELSEIF RCS /= NULL¬PATH THEN
$ THE OCCURENCE IS IN ANOTHER PROCEDURE.
        IF RCS(↓RCS)(1) = RC¬CALL THEN
$ OI REACHES THIS PROCEDURE THROUGH ITS ENTRY.
            IF IND > ↓INTSEQ THEN RETURN ↓INTSEQ;
            ELSE RETURN IND;
            END IF;
        ELSE INT := BLOCKOF(RCS(↓RCS)(2));
$ OTHERWISE, OI REACHES THIS PROCEDURE THROUGH A RETURN TO
$ SOME CALLING INSTRUCTION, PROCEED AS IF OI OCCURED AT THIS
$ CALLING INSTRUCTION.
        END IF;
    ELSE INT := BLOCKOF(INSTNO(OI));
$ INT IS THE BLOCK CONTAINING OI.
    END IF;

$ NOW ITERATE THROUGH THE DERIVED SEQUENCE AND FIND THE FIRST
$ INTERVAL IN INTSEQ CONTAINING OI, IF IT IS OF ORDER I > IND
$ THEN RETURN IND, OTHERWISE RETURN I-1.
```

```
      (∨ I := 1 ,.. ↓INTSEQ)
           INT := INTOV(INT);
           IF INT = INTSEQ(I) THEN RETURN I - 1;;
           IF I = IND THEN RETURN IND;;
      END ∨;

  $ NO SUCH INTERVAL. THE GRAPH MUST BE IRREDUCIBLE AND
  $ IND MUST BE > ↓INTSEQ. RETURN THE LARGEST INDEX IN INTSEQ.
      RETURN ↓INTSEQ;

      END;

      END INTMAX;



      DEFINEF INTMIN(OI,RCS,IND);
  $ THIS FUNCTION COMPUTES THE SMALLEST INDEX I IN INTSEQ,
  $ A TUPLE CONTAINING ALL INTERVALS CONTAINING A CERTAIN
  $ OCCURENCE, SUCH THAT I ≥ IND AND [OI,RCS] IN INTSEQ(I).
  $ THE FLOW IS SIMILAR TO THE PREVIOUS ROUTINE, INTMAX.

      BEGIN

      IF IND > ↓INTSEQ THEN RETURN IND;
  $ NO WAY TO FIND A LARGER INTERVAL THAN THE CURRENT ONE.

      ELSEIF RCS /= NULL¬PATH THEN
           IF RCS(↓RCS)(1) = RC¬CALL THEN RETURN ↓INTSEQ + 1;
           ELSE INT := BLOCKOF(RCS(↓RCS)(2));
           END IF;
      ELSE INT := BLOCKOF(INSTNO(OI));
      END IF;

  $ ITERATE OVER THE DERIVED SEQUENCE AS BEFORE. RETURN
  $ THE MAXIMUM OF IND AND THE INDEX OF THE FIRST INTERVAL IN
  $ INTSEQ CONTAINING OI.
      (∨ I := 1 ,.. ↓INTSEQ)
           INT := INTOV(INT);
           IF INT := INTSEQ(I) THEN
                IF I > IND THEN RETURN I;
                ELSE RETURN IND;
                END IF;
           END IF;
      END ∨;
```

```
$ NO SUCH INTERVAL FOUND, RETURN AN INDICATION THAT
$ OI IS OUT OF THE LARGEST INTERVAL IN INTSEQ.
    RETURN #INTSEQ + 1;
    END;

    END INTMIN;


    END COPYOP;
```

EXAMPLES
---------

1. CONSIDER THE FOLLOWING SETL CODE FRAGMENT.

```
    L1        A = B;
    L2        (∀X ∈ S)
    L3            A WITH X;;
```

SUPPOSE THAT B IS LIVE AFTER INSTRUCTION L1, OR THAT THE VALUE OF B IS SHARED BEFORE EXECUTING L1. THEN OUR ALGORITHM WILL MOVE THE COPY THAT WILL BE OTHERWISE REQUIRED BEFORE THE DESTRUCTIVE USE OF A AT L3, OUT OF THE LOOP TO THE POINT JUST BEFORE L2. THIS OPTIMIZATION IS ALMOST IDENTICAL TO THE ≠COPY ON ASSIGNMENT≠ OPTIMIZATION, SUGGESTED BY R. DEWAR IN NL. 164, WHICH IS TO COPY JUST AFTER THE ASSIGNMENT AT L1. OUR OPTIMIZATION IS EVEN SUPERIOR TO THE OTHER ONE, SINCE IT INSERTS THE COPY OPERATION AT THE POINT OF MINIMAL LOOP-NESTING LEVEL ON THE PATH FROM L1 TO L3, WHEREAS THE COPY-ON-ASSIGNMENT OPTIMIZATION MIGHT CREATE UNNECESSARY COPY OPERATIONS IF, FOR EXAMPLE, L1 IS CONTAINED IN ANOTHER LOOP.

MOREOVER, IF B IS DEAD JUST AFTER L1 (AS WILL BE THE CASE IF L1 IS ACTUALLY A VALUE TRANSFER FROM A FORMAL WRITE PARAMETER TO AN ACTUAL ARGUMENT), THEN OUR COPY OPTIMIZATION DEPENDS ON THE WAY IN WHICH B WAS CREATED. IF ALL PREVIOUS CREATIONS OF B THAT REACH L1 WERE NEW VALUE CREATIONS, THEN NO COPY WILL BE INSERTED. IF SOME OF THESE CREATIONS YIELD NEW VALUES AND

SOME YIELD SHARED VALUES, THEN WE SHALL INSERT A CONDITIONAL COPY JUST BEFORE L2, INVOLVING A TEST OF THE SHARE BIT OF A. IF ALL OF THESE CREATIONS YIELD SHARED VALUES, THEN AN UNCONDITIONAL COPY WILL BE INSERTED THERE. FINALLY, THE INSTRUCTION AT L1, WHICH POTENTIALLY HAS TO SET THE SHARE BIT OF A, WILL DO SO ONLY IN THE CASE WHERE A CONDITIONAL COPY OF A HAS BEEN INSERTED.

2. AS FAR AS COPY OPTIMIZATION WITH RESPECT TO LOOPS IS CONSIDERED, OUR ALGORITHM WILL DO BETTER THAN THE COPY-ON-ASSIGNMENT ONE. HOWEVER, THERE ARE SOME CASES OF STRAIGHT-LINE CODE COPY OPTIMIZATIONS WHICH OUR ALGORITHM WILL NOT CATCH, WHEREAS THE OTHER ALGORITHM WILL. HOWEVER, WE BELIEVE THAT THESE CASES ARE QUITE RARE. BESIDES, TO DETECT THESE CASES REQUIRES A FULL VALUE-FLOW INFORMATION, WHICH IS NOT NEEDED FOR OUR ALGORITHM.

ONE SUCH EXAMPLE IS AS FOLLOWS.

```
    L1       B := A;
    L2       C := A;


    L3       (∀ [X,Y] → S)
                 B WITH X;
                 C WITH Y;;
```

SUPPOSE THAT A HAS BEEN CREATED BEFORE L1 AS A NEW VALUE, AND THAT IT IS DEAD AFTER L2. OUR ALGORITHM WILL INSERT COPY OPERATIONS OF B AND C BEFORE L3, BUT ACTUALLY ONLY ONE OF THEM SHOULD BE COPIED.