NAME SPLITTING IS ONE OF THE FINAL PHASES OF THE SETL OPTIMIZER. IT IS PERFORMED AFTER THE TYPE-FINDING AND AUTOMATIZED DATA STRUCTURE SELECTION PHASES HAVE COMPUTED A MAP #OI-REPR, DEFINED ON VARIABLE OCCURRENCES, FOR EACH OCCURRENCE VO, #OI-REPR(VO) IS A REPR FOR VO THAT HAS BEEN SELECTED BY THESE PHASES, CONJOINED WITH USER-SUPPLIED REPR INFORMATION.

HOWEVER, THE MAP #OI-REPR IS NOT OF DIRECT USE TO THE CODE GENERATOR, WHICH DEALS WITH SYMBOL TABLE ENTRIES RATHER THAN WITH OCCURRENCES.

OF COURSE, IF ALL OCCURRENCES OF THE SAME VARIABLE HAVE THE SAME REPR, THEN WE CAN TRANSMIT THIS REPR TO THE CODE GENERATOR. UNFORTUNATELY, THERE ARE MANY COMMON CASES (MAINLY, BUT NOT EXCLUSIVELY, IN NON-REPR'D PROGRAMS) WHERE THIS IS NOT THE CASE.

FOR EXAMPLE, CONSIDER

EXAMPLE 1.

READ X;
Y := X + 1;

OR EVEN:

EXAMPLE 2.

(1) READ X;
(2) IF COID THEN
(3)   (~)
(4)   Y := X + 1;
(5)   END ~;
(6) ELSE
(7)   (~)
(8)   Z := X + #1/*;
(9)   END ~;
(10) END IF;

EVEN IF WE FULLY REPR EXAMPLE 2, THE ONLY REPR THAT X CAN HAVE IS TYPE GENERAL, SO THAT THE ADDITIONS AT LINES (4) AND (8) WILL BE SLURRED DUE TO TYPE CHECKS AND CONVERSIONS, MOREOVER, IT WILL BE IMPOSSIBLE TO EMIT IN-LINE CODE FOR THESE ADDITIONS, IT IS THEREFORE OF INTEREST TO NOTE THAT BY ANALYZING OCCURRENCES, RATHER THAN VARIABLES, TYPE-FINDING WILL REVEAL THAT TYPE(X| := X AT LINE 1) := GENERAL, TYPE(X4) := INTEGER AND
TYPE(XB) := CHARACTERS, WE CAN THEN MAKE USE OF THIS INFORMATION
BY SPLITTING THE VARIABLE X INTO THREE DIFFERENT SYMBOL TABLE
ENTRIES, XA, XB, XC, HAVING THE REPS GENERAL, INTEGER AND
CHARACTERS RESPECTIVELY, IN THIS WAY WE SEPARATE THE NECESSARY
TYPE CHECKS AND CONVERSIONS FROM THE ACTUAL ADD INSTRUCTIONS
(4) AND (8), AND THUS OPEN UP THE POSSIBILITY OF CREATING
EFFICIENT CODE FOR THEM, WHEN THIS IS DONE, CONVERSIONS MUST
BE MADE EXPLICIT IN THE CODE, WE CAN TRY TO INSERT THESE CONVERSIONS
AT OPTIMAL PLACES BY MOVING THEM OUT OF LOOPS, IF POSSIBLE.

LET US VISUALIZE THESE CONVERSIONS AS ASSIGNMENTS OF ONE SPLIT
VARIABLE TO ANOTHER. THE SECOND EXAMPLE WOULD THEREBY BE TRANSFORMED INTO THE FOLLOWING CODE:

EXAMPLE 2A,

READ XA;
IF COID THEN
   XB := XA;
   (v)
   Y := XB + 1
END v;
ELSE
   XC := XA;
   (v)
   Z := XC + ?1X
END v;
END IF;

WE CALL THE TRANSFORMATION FROM (2) TO (2A) NAME-SPLITTING.

ANOTHER AND MORE IMPORTANT CASE IN WHICH NAME SPLITTING IS REQUIRED
IS THE INSERTION OF #LOCATE# INSTRUCTIONS WHICH PUT BASED ELEMENTS
INTO THEIR BASES. CONSIDER THE FOLLOWING EXAMPLE:

EXAMPLE 3,

(1) (v)
(2) X := X + 1;
(3) END v;
(4) $ WITH X;

SUPPOSE THAT S4 HAS THE REPR SET(8) (CHOSEN AUTOMATICALLY OR
MANUALLY), IN THIS CASE IT IS DISADVANTAGEOUS TO REPR BOTH X2
AND X4 AS #8, FOR ONLY THE LAST CREATED VALUE OF X2 HAS ACTUALLY
TO BE INSERTED INTO THE BASE 8, WE EXPECT THE AUTOMATIC DATA
STRUCTURE SELECTION PHASE TO COME UP WITH 01-REPR(X2) := INTEGER,
01-REPR(X4) := 8. THE HEURISTIC NAME SPLITTING SHOULD THEREFORE
AIM TO TRANSFORM THIS CODE INTO:
EXAMPLE 3A,

\[
\begin{align*}
& \text{let } X := X + 1; \\
& \text{END } X; \\
& XB := XA; \\
& S \text{ with } XB;
\end{align*}
\]

WHERE XA, XB ARE SPLIT SYMBOL TABLE ENTRIES FOR X, HAVING THE
REPRS INTEGER AND -B RESPECTIVELY, AND THE ASSIGNMENT XB := XA
ACTUALLY SIGNIFIES A #LOCATE# OF THE VALUE OF XA IN B. SUCH
A #LOCATE# COMPUTES A BASE POINTER FOR XA, INSERTING IT INTO B
IF NECESSARY, AND ASSIGNS THIS POINTER TO XB.

IN THE FOREGOING EXAMPLE, NO MOTION OF LOCATES IS NEEDED. HOWEVER,
IN THE FOLLOWING EXAMPLE

EXAMPLE 4,

\[
\begin{align*}
& \text{let } X := X + 1; \\
& \text{END } X; \\
& S \text{ with } X;
\end{align*}
\]

CODE MOTION IS PROBABLY ADVANTAGEOUS, SINCE IT WILL TRANSFORM
THIS CODE INTO

EXAMPLE 4A,

\[
\begin{align*}
& \text{let } XA := XA + 1; \\
& XB := XA; \\
& \text{END } XB; \\
& S \text{ with } XB;
\end{align*}
\]

WITH SIMILAR XA AND XB,

HAVING CONVINCED OURSELVES THAT A NAME-SPLITTING MECHANISM IS
NECESSARY, LET US NOW DESCRIBE A NAME SPLITTING ALGORITHM AND
THE CONVERSION-MOTION ALGORITHM IT USES IN DETAIL.

DEFINITION: LET V BE A PROGRAM VARIABLE, AND LET R BE A REPR OF
SOME OF ITS OCCURRENCES, WE DEFINE A SPLIT VARIABLE OF V WITH THE
REPR R, AS A PAIR \([V, R]\). LET #SPLIT-NAME# BE THE MAP SENDING
EACH OCCURRENCE VO INTO THE PAIR \([OI-NAME(VO), OI-REPR(VO)]\), AND
LET #EQUIP# BE THE EQUIVALENCE RELATION INDUCED BY THIS MAP AS
A QUOTIENT MAP.

FOR EVERY SPLIT VARIABLE OF V WE INTRODUCE A NEW SYMBOL TABLE
ENTRY VA, WITH THE UNDERSTANDING THAT IF V IS NOT REALLY SPLIT,
THEN VA WILL BE THE ORIGINAL ENTRY OF V. AFTER SPLITTING, EACH
VARIABLE OCCURRENCE WILL THEN BE REGARDED AS AN OCCURRENCE OF
AN APPROPRIATE SPLIT VARIABLE, AND THE CODE WILL BE MODIFIED TO SHOW THESE SPLIT VARIABLES RATHER THAN THE VARIABLES ORIGINALLY OCCURRING.

ONE IMPORTANT OBSERVATION IS THAT TWO SPLIT VARIABLES OF THE SAME ORIGINAL VARIABLE ARE NEVER LIVE SIMULTANEOUSLY, AND SO THEY CAN SHARE STORAGE. THIS FACT WILL BE USED BY THE CODE GENERATOR, AND ALSO BY THE NAME-SPLITTING ALGORITHM ITSELF.

THE NAME SPLITTING ALGORITHM CONSISTS OF THE FOLLOWING STEPS:

STEP 1) PERFORMS ACTUAL NAME SPLITTING AND COLLECTS ALL LINKS BETWEEN OCCURRENCES OF A SINGLE VARIABLE WHICH HAVE BEEN SPLIT.

STEP 2) INSERTS CONVERSIONS AND CHECKS INTO THE CODE.

STEP 1) IS PERFORMED IN A STRAIGHTFORWARD WAY, BY ITERATING OVER BFROM, WHENEVER WE ENCOUNTER A LINK BETWEEN TWO OCCURRENCES WITH DIFFERENT UI-REPR VALUES, WE AUGMENT A WORKPILE OF SUCH LINKS, AND ADD NEW ENTRIES TO THE SYMBOL TABLE, IF NECESSARY. THE VARIABLE NAMES APPEARING IN OCCURRENCES ARE REPLACED BY THE NAMES OF THE CORRESPONDING SPLIT VARIABLES.

STEP 2) IS MORE COMPLICATED, AND, LIKE ANY OTHER CODE NOTION ALGORITHM, RAISES PROBLEMS OF SAFETY AND PROFITABILITY, AS WELL AS A FEW ADDITIONAL MORE SPECIFIC PROBLEMS. THE APPROACH DESCRIBED BELOW AIDS TO ENSURE A MODERATE LEVEL OF PROFITABILITY, BUT MAY NOT PRODUCE OPTIMAL CODES IN SEVERAL EXTREME CASES, HOWEVER, IT IS RATHER SIMPLE, AND WILL GENERALLY PRODUCE QUITE ACCEPTABLE CODE.

LET VO BE A VARIABLE OCCURRENCE, FOR WHICH THERE EXISTS V01 \rightarrow BFROMSV02 SUCH THAT NOT (V01 =E REPR V01). THE SAFEST PLACE TO INSERT A CONVERSION/TEST OF THE FORM OF VO IS JUST BEFORE THE INSTRUCTION CONTAINING VO, INDEED, THE type Finder AND THE AUTOMATIC DATA STRUCTURE SELECTION PHASES FUNCTION IN SUCH A WAY AS TO ENSURE THAT THE INSTRUCTION ORIGINALLY CONTAINING VO, BEFORE NAME-SPLITTING, IS EQUIVALENT TO THE CONVERSION/TEST FOLLOWED BY THE SAME INSTRUCTION BUT WITH THE SPLIT VARIABLE REPLACING THE ORIGINAL VARIABLE.

CONVERSION/TEST INSTRUCTIONS ARE REPRESENTED IN THE FINAL CODE WE ENVISAGE AS ASSIGNMENTS OF ONE SPLIT VARIABLE TO ANOTHER, SINCE ALL THE MEMBERS OF A GROUP OF VARIABLES SPLIT FROM A SINGLE ORIGINAL VARIABLE SHARE STORAGE, SUCH AN ASSIGNMENT IS SIMPLY A CONVERSION OF THE VALUE SPECIFIED AT THIS COMMON LOCATION (OR PERHAPS JUST A TEST THAT THIS VALUE HAS A DESIRED FORM), AND A REPLACEMENT OF THE OLD VALUE SPECIFIER WITH THE NEW ONE.

HOWEVER, VO MAY HAVE TWO OR MORE OCCURRENCES, V01, V02 \rightarrow BFROMSV02, SUCH THAT V01, V02 ARE TO BE REPLACED BY DIFFERENT SPLIT
VARIABLES, so that if we insert the conversion just before \( v_0 \),
it's variable must have a form dominating those of \( v_{o1} \) and \( v_{o2} \)
(in the type-lattice sense). In this case, we generate a new
dummy split variable of the variable of \( v_0 \), having this dominating
form, and make it the name of the variable of the conversion.
This operation is easily seen to work properly, since all these
split variables share storage.

In some cases we also have another alternative: namely, to push
the conversion upward towards \( v_{o1} \) and \( v_{o2} \), continuing to move
it upward until the conversion itself can be split into two
conversions having as variables the split variables of \( v_{o1} \) and
\( v_{o2} \) respectively. This is shown in the following example:

**Example 5.**

```plaintext
L2    F := [1,2];
L1    PRINT F(1);
```

Here \( F \) has three split variables, \( F_{a} \) (map), \( F_{b} \) (tuple) and
\( F_{c} \) (general). The first approach sketched above will transform
this into

**Example 5A.**

```plaintext
L2    F_{a} := [1,2];
L1    F_{b} := [1,2];
L1    F_{c} := F_{d};
      PRINT F_{c}(1);
```

Where \( F_{d} \) is a split variable of \( F \) having general type
(in this particular example, \( F_{d} \) is identical with \( F_{b} \)). The
second approach sketched above will transform the code into

**Example 5B.**

```plaintext
L2    F_{a} := [1,2];
L2    F_{c} := F_{a};
L1    F_{b} := [1,2];
L1    F_{c} := F_{b};
L1    PRINT F_{c}(1);
```
HOWEVER, THERE ARE CASES WHERE THE SECOND ALTERNATIVE WILL FAIL,
AS IN THE FOLLOWING EXAMPLE:

EXAMPLE 6,

\[
\begin{align*}
F &:= \leq [1,2] \\
& \text{GO TO L1;}
\end{align*}
\]

L2

\[
F := [1,2];
\]

L1

\[
\begin{align*}
\text{IF \, \text{COND}, \quad} & F \cdot \leq [3,4] \\
\text{ELSE,} & F \cdot + [3,4];
\end{align*}
\]

END IF;

HERE F HAS ONLY TWO SPLIT VARIABLES \texttt{FA(MAP)} AND \texttt{FB(TUPLE)},
but, as can be checked, there is no place in the code in which we can insert an explicit conversion of one such split variable
into another, without causing a possible abort, which might not
have occurred in the original program.

THUS, OUR ALGORITHM WILL HAVE TO MAKE USE OF THE FIRST ALTERNATIVE
FOR AT LEAST CERTAIN CASES, AND WE PROPOSE TO DROP THE SECOND
ALTERNATIVE ALTOGETHER, FOR THE FOLLOWING REASONS:

A) SITUATIONS SUCH AS THOSE SHOWN IN EXAMPLE 5, ARE RARE. IN
MOST CASES, VO WILL BE LINKED ONLY TO ONE SPLIT VARIABLE, AND
SO INSERTED CONVERSIONS WILL HAVE A SPECIFIC VARIABLE ANYWAY;

B) EVEN WHEN SITUATIONS SUCH AS THAT SHOWN IN EXAMPLE 5, DO
HAPPEN, THE GAIN FROM HAVING A SPECIFIC VARIABLE IN INSERTED
CONVERSIONS (NOTE THAT CONVERSIONS FROM TYPE GENERAL ARE
SLOWER, SINCE THEY INVOLVE A BRANCH ON THE ACTUAL FORM OF
THE VARIABLE) DOES NOT JUSTIFY THE SIGNIFICANT INCREASE IN
THE COMPLEXITY OF THE ALGORITHM REQUIRED TO IMPLEMENT THE
SECOND APPROACH.

HOWEVER, CERTAIN QUITE COMMON CASES DO CALL FOR SOMETHING LIKE
OUR SECOND APPROACH. THESE ARE CASES IN WHICH ALL OCCURRENCES OF
A VARIABLE WITHIN A LOOP HAVE THE SAME SPLIT VARIABLE, BUT SOME
OF THEM ARE ALSO LINKED TO OCCURRENCES OUTSIDE THE LOOP, HAVING
A DIFFERENT SPLIT VARIABLE. IF IN THESE CASES WE LEAVE CONVERSIONS
INSIDE A LOOP, THEY MAY HAVE TO BE CONVERSIONS FROM TYPE
GENERAL, AND IN ANY CASE THEY WILL IMPLY REDUNDANT TESTS THAT
A VARIABLE IS IN THE CORRECT FORM, FROM THE SECOND ITERATION
ONWARD, THUS, THE POSSIBILITY OF MAKING THE VARIABLE OF AN
INSERTED CONVERSION MORE SPECIFIC WHILE MOVING CONVERSIONS OUT OF
LOOPS WILL BE TAKEN INTO CONSIDERATION IN OUR ALGORITHM.
AS WITH ANY KIND OF CODE MOTION, MOVING A CONVERSION OUT OF A LOOP IS IN GENERAL NOT SAFE, FOR THE MODIFIED PROGRAM MAY ABORT IN CERTAIN SITUATIONS IN WHICH THE ORIGINAL PROGRAM WOULD NOT (E.G. THE LOOP MAY BE BYPASSED).

EVEN IF WE FOLLOW (AND INDEED WE WILL) THE APPROACH OF THE STANDARD CODE MOTION PHASE OF THE OPTIMIZER (SEE NL, 197), I.E. ASSUME THAT CODE MOTION WILL BE PERFORMED ONLY ON PROGRAMS THAT WILL RUN IN A SPECIAL EXECUTION MODE, IN WHICH OPERATIONS WITH ILLEGAL ARGUMENTS DO NOT CAUSE A PROGRAM ABORT, BUT PRODUCE AN ERROR VALUE, WE STILL FACE A SAFETY PROBLEM, CHARACTERISTIC OF ASSIGNMENTS; NAMELY - IF WE MOVE AN ASSIGNMENT OUT OF A LOOP AND IT IS ILLEGAL, THEN EVEN IF THE PROGRAM DOES NOT ABORT, ASSIGNING THE ERROR VALUE TO THE VARIABLE WILL KILL THE PREVIOUS VALUE SPECIFIER OF THAT VARIABLE, WHICH WOULD BE RETAINED, HAD THE ASSIGNMENT BEEN LEFT IN THE LOOP, AND THE LOOP NEVER Executed. FOR EXAMPLE:

EXAMPLE 7,

```
READ V;
LENV := 0;
IF TYPE V = TUPLE THEN LENV := +V; END IF;

(v I := 1 ... LENV)
X := V(I);
END v;

IF LENV = 0 THEN X := V; END IF;
```

ASSUMING THAT THE INPUT V IS EITHER AN INTEGER OR A TUPLE OF INTEGERS, THE ABOVE CODE WILL NOT ABORT, AND AT ITS END, X WILL BE ASSIGNED AN INTEGER VALUE, SUPPOSE THAT THE TYPE INFORMATION SUGGESTS THAT WE SPLIT V INTO THREE SPLIT VARIABLES, VA(GENERAL), VB(TUPLE) AND VC(INTEGER), THE CODE CAN THEN BE SAFELY TRANSFORMED INTO THE FOLLOWING CODE:

```
READ VA;
LENV := 0;
IF TYPE VA = TUPLE THEN LENV := +VA; END IF;

(v I := 1 ... LENV)
VB := VA;
X := VB(I);
END v;

IF LENV = 0 THEN VC := VA; X := VC; END IF;
```

NOTE THAT THE ASSIGNMENT VB := VA; CAN NOT BE MOVED OUT OF THE LOOP, EVEN IN THE SPECIAL EXECUTION MODE DESCRIBED ABOVE, FOR IF WE MOVE IT OUT, AND V HAPPENS TO BE AN INTEGER, THEN THIS
CONVERSION WILL RESULT IN AN ERROR VALUE, STORED AT THE COMMON LOCATION OF ALL THE SPLIT VARIABLES OF V, SO THAT THE INTEGER VALUE OF V IS DESTROYED, AND X WILL BE ASSIGNED AN ERROR VALUE, INSTEAD OF THE INPUT INTEGER VALUE OF V.

NOTE, HOWEVER, THAT NOT EVERY CONVERSION CAN FAIL. A CONVERSION IS UNCONDITIONALLY SAFE, IF THE FORM OF ITS VARIABLE IS MORE SPECIFIC THAN, OR EQUIVALENT TO, THE FORM OF ITS VARIABLE. FOR EXAMPLE, CONVERSION TO GENERAL IS ALWAYS SAFE; CONVERSION FROM INTEGER TO AN ELEMENT OF B, WHERE B IS A BASE OF INTEGERS, IS ALWAYS SAFE, AS WELL AS THE INVERSE CONVERSION.

THIS SUGGESTS THAT WE EXTEND THE TYPE ORDER TO A PSYDUS-ORDER RELATION (REFLEXIVE, TRANSITIVE, BUT NOT NECESSARILY ANTI-SYMMETRIC) \$ \leq \text{defined on forms in the following way: let } \$ \text{type-of} \$

\text{denote the function that computes the type of a given form, in a recursive manner, replacing } \$ \text{element-of-base} \$

\text{descriptors by the mode of the corresponding cases, then } \text{form}_1 \text{ le } \text{form}_2 \text{ iff } \text{type-of(form}_2 \text{) dominates type-of(form}_1 \text{) in the standard type lattice, thus, it can be easily seen that a conversion } \text{va := vb} \text{ is always safe iff } \text{form(vb)} \text{ le } \text{form(va)}.\$

IN VIEW OF THE OBSERVATIONS MADE IN THE PREVIOUS PARAGRAPHS, WE SHALL USE THE FOLLOWING RATHER SIMPLE, THOUGH SOMEWHAT WEAK, CRITERION TO DETERMINE WHETHER A CONVERSION CAN BE MOVED OUT OF A LOOP:

LET VO BE A VARIABLE OCCURRENCE, LINKED BY BFROM TO OTHER SPLIT VARIABLES, IF THE FOLLOWING CONDITION IS SATISFIED,

\((*)\) \ \text{v}_01 \rightarrow \text{BFROM} \leq \text{v}_02 \rightarrow \left( \text{v}_1\text{-REPR}(\text{v}_01) \text{ LE } \text{v}_1\text{-REPR}(\text{v}_0) \right) \\
\text{or} \left( \text{v}_02 \rightarrow \text{BFROM} \leq \text{v}_01 \rightarrow \left( \text{v}_1\text{-REPR}(\text{v}_02) \text{ LE } \text{v}_1\text{-REPR}(\text{v}_0) \right) \right)

THEN A CONVERSION TO SPLIT-NAME(VO), INITIALLY PLACED JUST BEFORE THE INSTRUCTION CONTAINING VO, CAN BE MOVED OUT OF ITS LOOP(S), OTHERWISE, IT MUST REMAIN AT ITS INITIAL LOCATION.

THE HEURISTIC BASIS OF THIS APPROACH MAY BE STATED AS FOLLOWS: GIVEN THAT WE DO NOT HAVE ANY MORE DETAILED DATA-FLOW INFORMATION, WE MUST ASSUME THAT IF A LOOP CONTAINING VO IS NOT EXECUTED, THEN THERE MAY BE A PATH FROM SOME VO1 \rightarrow BFROM \leq VO2 TO ANOTHER VO2 \rightarrow BFROM \leq VO12, WHICH PASSES THROUGH THE TARGET BLOCK OF THIS LOOP (INTERVAL).

THE TARGET BLOCK OF AN INTERVAL IS A SPECIAL BASIC BLOCK, CREATED BY THE OPTIMIZER, WITH THE PROPERTY THAT IT IS THE ONLY BASIC BLOCK OUTSIDE THIS INTERVAL THAT IS A PREDECESSOR OF THE INTERVAL HEAD. WE CREATE THIS BLOCK SO THAT CODE MOVED OUT OF THIS INTERVAL CAN BE INSERTED INTO IT), AND IF WE MOVE A CONVERSION TO THE FORM OF VO OUT OF THAT INTERVAL, INTO ITS BASIC BLOCK, THIS CONVERSION CAN CUT THE ABOVE PATH FROM VO1 TO VO2, THUS, TO BE SURE THAT NO HARM WILL BE DONE, WE MUST BE SURE THAT IF THIS CONVERSION FAILS, THEN THE CONVERSION TO SPLIT-NAME(VO2), PLACED JUST BEFORE VO2, WOULD HAVE FAILED ALSO.

CONDITION \((*)\) IS PRECISELY EQUIVALENT TO THAT ASSERTION.
ONCE HAVING GIVEN THE PRECEDING SOLUTION TO THE SAFETY PROBLEM, WE USE A RATHER LIBERAL CRITERION FOR PROFITABILITY, AND ASSUME THAT IT IS ALWAYS PROFITABLE TO MOVE A CONVERSION FROM THE LOOP-PART OF AN INTERVAL TO ITS TARGET BLOCK.

NOTE THAT THE CRITERION FOR PROFITABILITY SET BY THE STANDARD CODE MOTION PHASE OF THE OPTIMIZER IS STRICTER, AND DEMANDS THAT A COMPUTATION MUST BE UNCONDITIONALLY EXECUTED INSIDE AN INTERVAL, PROVIDED THAT THE LOOP OF THAT INTERVAL IS EXECUTED AT LEAST ONCE. NOTE ALSO THAT OUR CODE MOTION CRITERIA REFLECT OUR RELUCTANCE TO RE-PERFORM A FULL SCALE DATA-FLOW ANALYSIS, WHICH WOULD ALLOW US TO SHARPEN THE CRITERIA OF MOTION, AND IMPROVE THE LOCATION-OPTIMIZATION OF CONVERSIONS, WE MIGHT BE WILLING TO PERFORM SUCH AN ANALYSIS IF FUTURE EXPERIMENTATION WITH SETL OPTIMIZATION REVEALS SIGNIFICANT INEFFECTIVENESS IN THE MOTION OF CONVERSIONS, BUT PRESENTLY IT SEEMS LIKELY THAT OUR ALGORITHM WILL ACHIEVE GOOD RESULTS IN MOST CASES.

THERE IS YET ONE MORE PROBLEM THAT THE NOTION OF CONVERSIONS RAISES, IN THE GENERAL CASE, IT IS RATHER DIFFICULT TO DETERMINE A PROPER REPR FOR AN IVARIABLE OF AN INSERTED CONVERSION OPERATION, LET VO BE SOME VARIABLE OCCURRENCE OF A VARIABLE V WHICH IS LINKED BY BFROM TO OTHER OCCURRENCES OF V HAVING SPLIT VARIABLES WHICH ARE DIFFERENT FROM SPLIT-NAME(VO), SO THAT A CONVERSION TO THE FORM OF VO OUGHT TO BE INSERTED BEFORE VO IS USED, SUPPOSE THAT OUR ALGORITHM HAS DETERMINED TO MOVE THAT CONVERSION TO THE TARGET BLOCK D OF SOME INTERVAL INT CONTAINING VO, IN COMPLIANCE WITH ALL THE ABOVE CRITERIA OF CONVERSION-MOTION. THEN THE IVARIABLE OF THAT CONVERSION WILL HAVE AN OBVIOUS FORM IFF ALL OCCURRENCES IN BFROM<VO> THAT CAN REACH B HAVE THE SAME FORM, OUR PROBLEM IS TO DETERMINE, WITHOUT A FULL DATA FLOW ANALYSIS, WHICH OCCURRENCES IN BFROM<VO> CAN REACH B, IF VO1 = BFROM<VO> IS NOT CONTAINED IN INT, THEN IT MUST CERTAINLY REACH B, BUT IF VO1 IS INSIDE INT, (AND SINCE VO IS IN THE LOOP-PART OF INT THERE WILL BE AT LEAST ONE SUCH OCCURRENCE), THEN THERE IS NO SIMPLE CRITERION TO DETERMINE WHETHER VO1 CAN ALSO REACH VO THROUGH B. HOWEVER, THIS OCCURRENCE WILL BE EREPR TO VO, WHEREAS OTHER OCCURRENCES IN BFROM<VO> WILL NOT BE, AND SO, IF WE CAN NOT IGNORE THESE INSIDE LINKS IN DETERMINING THE REPR OF THE IVARIABLE OF THE CONVERSION, WE WILL ALWAYS HAVE TO REPR IT AS A GENERAL TYPE, WHICH IS CERTAINLY UNDESIRABLE.

IT IS RATHER DIFFICULT TO FIND A GENERAL NECESSARY AND SUFFICIENT CONDITION TO DETERMINE THE ABSENCE OF SUCH INSIDE LINKS, THE CONDITION THAT OUR ALGORITHM WILL USE (CONDITION (**), SEE BELOW) IS SOMEWHAT PESSIMISTIC, AND IS ONLY SUFFICIENT, BUT IT GIVES AN ADEQUATE ANSWER IN MOST CASES, THOUGH IT MAY CHOOSE, IN SOME RATHER RARE CASES, A GENERAL IVARIABLE FOR A CONVERSION UNNECESSARILY.
LET US NOW SKETCH THE CONVERSION INSERTION PHASE OF THE NAME-SPLITTING ALGORITHM:

THE FIRST PHASE OF THE ALGORITHM WILL HAVE COMPUTED A WORKPILE OF OCCURRENCES THAT ARE LINKED BY BFROM TO DIFFERENT VARIABLES OF THE SAME SPLIT GROUP.

(VOICE TO WORKPILE)

IF VO DOES NOT SATISFY THE SAFETY CONDITION (*) THEN

IF ALL VO1 - BFROM ≤ VO2 HAVE THE SAME FORM THEN

VI := SPLIT-NAME(ARITH BFROM ≤ VO2);

ELSE

VI := (OI-NAMESPACE(VO), GENERAL);

END IF;

INSERT BEFORE THE INSTRUCTION OF VO THE CONVERSION SPLIT-NAME(VO) := VI

ELSE

COMPUTE INTSEQ, THE SEQUENCE OF INTERVALS CONTAINING VO, STARTING AT ITS BASIC BLOCK.

FIND THE LARGEST INDEX J SUCH THAT INT := INTSEQ(J) DOES NOT CONTAIN ANY OCCURRENCE IN THE SET ≤ VO1 - BFROM ≤ VO2 + NOT (VO1 EREPR VO2) AND INTSEQ(J-1) IS IN THE LOOP-PART OF INT (THAT IS, HEAD(INT) CAN BE REACHED FROM INTSEQ(J-1) ALONG A PATH WHOLLY CONTAINED IN INT).

(INT IS COMPUTED IN TWO STEPS: FIRST, FIND THE LARGEST INDEX K SUCH THAT INTSEQ(K) DOES NOT CONTAIN ANY OCCURRENCE IN THE ABOVE SET, THEN, FIND THE LARGEST INDEX J <= K, SUCH THAT INTSEQ(J-1) IS IN THE LOOP-PART OF INTSEQ(J). SET INT := INTSEQ(J).)

IF ALL VO1 - BFROM ≤ VO2 + INT DOES NOT CONTAIN VO1 HAVE THE SAME FORM, AND THE FOLLOWING CONDITION IS SATISFIED,

(**) THERE EXISTS M <= INTSEQ SUCH THAT ALL OCCURRENCES VO1 - BFROM ≤ VO2 THAT ARE OUTSIDE INT, ARE CONTAINED IN INTSEQ(M) BUT NOT IN INTSEQ(M-1)

THEN

VI := SPLIT-NAME(ARITH (THE ABOVE SET));

ELSE

VI := (OI-NAMESPACE(VO), GENERAL);

END IF;
INSERT AT THE END OF THE TARGET BLOCK OF INT THE
CONVERSION

\[ \text{SPLIT-NAME(VO) := VI;} \]

UNLESS THERE
IS ALREADY SUCH A CONVERSION IN THAT BLOCK, IN WHICH
CASE BOTH CONVERSIONS ARE MERGED INTO ONE.

END IF;
END

LET US FIRST JUSTIFY THE USE OF CONDITION (***) IN OUR ALGORITHM.
WE ASSUME THAT THE ANALYZED PROGRAM HAS THE PROPERTY THAT ALL
THE VARIABLES ARE INITIALIZED BEFORE THEY ARE EVER USED (LOCAL
VARIABLES ARE INITIALIZED AT THE ENTRY TO THEIR PROCEDURE, AND
GLOBAL VARIABLES AT THE ENTRY OF THE MAIN PROGRAM), THUS, ANY
(STATIC) EXECUTION PATH LEADING FROM THE PROGRAM ENTRY TO A USE
OF A VARIABLE, MUST CONTAIN A DEFINITION OF THAT VARIABLE.

PROPOSITION: UNDER THE ABOVE ASSUMPTION, LET VO ≠ WORKPILE,
AND LET INTSEQ, INT BE AS COMPUTED IN THE ABOVE ALGORITHM FOR VO;
IF CONDITION (**) HOLDS FOR VO, THEN THERE CANNOT EXIST
VO2 ≠ BFROMSV02 WHICH IS INSIDE INT AND REACHES THE TARGET
BLOCK OF INT.

PROOF: LET V BE THE VARIABLE OF VO, OBSERVE THAT THERE EXISTS
A V-FREE PATH LEADING FROM HEAD(INTSEQ(M-1)) TO VO (OBVIOUS),
BUT THERE DOES NOT EXIST A V-FREE PATH LEADING FROM HEAD(INTSEQ(M))
TO VO, FOR OTHERWISE, BY OUR ASSUMPTION, THERE SHOULD BE AN
OCURRENCE IN BFROMSV02 OUTSIDE INTSEQ(M), CONTRARY TO CONDITION
(**).

SUPPOSE THAT SUCH VO2 EXISTS, IT FOLLOWS THAT ANY V-FREE PATH
FROM VO2 TO TB(INT) (THE TARGET BLOCK OF INT) IS CONTAINED IN
INTSEQ(M-1). HENCE, THERE EXISTS SOME K ≤ M-1 SUCH THAT THIS
PATH IS WHOLLY CONTAINED IN INTSEQ(K), BUT NOT WHOLLY CONTAINED
IN INTSEQ(K-1). INTSEQ(K) STRICTLY CONTAINS INT, BECAUSE TB(INT)
IS CONTAINED IN INTSEQ(K) AND IS NOT CONTAINED IN INT. THIS
IMPLIES THAT INTSEQ(K-1), WHICH CONTAINS INT, IS IN THE LOOP-
PART OF INTSEQ(K). ALSO, BY CONDITION (**) AND THE DEFINITION
OF INT, INTSEQ(K) EXCLUDES ALL OCCURRENCES VO1 ≠ BFROMSV02 THAT
ARE NOT EGHEREPR TO VO. THE LAST TWO OBSERVATIONS IMPLY THAT THE
INDEX J CONSTRUCTED BY OUR ALGORITHM FOR VO MUST BE ≥ K, SO
THAT INTSEQ(K) IS CONTAINED IN, OR EQUAL TO INT, WHICH IS A
CONTRADICTION.

Q. E. D.

REMARK: THIS CONDITION IS EVIDENTLY SATISFIED IF ONLY ONE
OCURRENCE IN BFROMSV02 IS NOT EGHEREPR TO VO AND IS OUTSIDE SOME
INTERVAL CONTAINING VO AND ALL OTHER OCCURRENCES IN BFROMSV02.
THIS WILL HAPPEN IN THE OVERTWELMING MAJORITY OF CASES, EVEN
WHEN THIS IS NOT THE CASE, SUCH OCCURRENCES ARE MORE LIKELY TO OCCUR
IN THE SAME LOOP DEPTHEST LEVEL AND NEAR EACH OTHER, SO THAT
CONDITION (**) IS VERY LIKELY TO HOLD.
LET US NOW SHOW THAT THE CONVERSION INSERTION ALGORITHM DOES ELIMINATE ALL LINKS BETWEEN DIFFERENT SPLIT VARIABLES.

THEOREM: AT THE END OF THE CONVERSION INSERTION ALGORITHM, IF WE REPLACE THE VARIABLE NAMES BY THEIR ORIGINAL, UNSPLIT NAMES, AND RE-COMPUTE THE BFROM MAP, THEN, FOR EACH OCCURRENCE VO, AND EACH VO1 BFROMVO2, VO1 EOREPR VO, UNLESS VO IS OF TYPE GENERAL.


LEMMA A: IF THERE IS A PATH LEADING FROM SOME CONVJ TO ANOTHER OCCURRENCE VO OF THE VARIABLE VJ, FREE OF OTHER (ORIGINAL) OCCURRENCES OF THAT VARIABLE, THEN THERE EXISTS VOP INSIDE INTJ, VOP BFROMVO2 BFROMVOJ AND VOP EOREPR VOJ.

PROOF: INTJ /= OM, FOR IF VOJ DOES NOT SATISFY (*), THEN NO SUCH PATH CAN EXIST. FIGURE (1) BELOW ILLUSTRATES THE SITUATION

**Figure (1)**

LEMMA B: IF CONVJ INSIDE INTJ, THEN CONVJ CAN REACH ANY VARIABLE OF INTJ.

**Figure (2)**

PROOF: CONVJ IS INSIDE INTJ, AND CONVJ CAN REACH ANY VARIABLE OF INTJ.
HEAD(INTJ)), AND LET P2 BE A VJ-FREE PATH FROM HEAD(INTJ) TO VO. LET P := P1 + P2. P IS NOT VJ-FREE, FOR VOJ LIES ON IT. LET VOP BE THE LAST OCCURRENCE OF VJ ON P (BEFORE VO), SINCE P2 IS VJ-FREE, VOP IS INSIDE INTJ, AND THE TERMINAL SUBPATH OF P, VOP ..., HEAD(INTJ) ..., VO IS VJ-FREE, HENCE VOP ✺ BFROMSVOJ ✺ BFROMSVOJ, AND BY THE DEFINITION OF INTJ VOP ✺ EOREPR ✺ VOJ.

Q. E. D.

WE NOW PROCEED WITH THE PROOF OF THE THEOREM, IN THE FOLLOWING STEPS

(1) FOR EACH K := 1 ... N, NEW-BFROMSVOK ✺ CONTAINS ONLY OCCURRENCES EOREPR TO VOK, INDEED, IF INTK = OM, THEN NEW-BFROMSVOK ✺ = OVK, AND THIS OCCURRENCE IS EOREPR TO VOK BY DEFINITION. OTHERWISE, BY THE DEFINITION OF INTK, EACH OCCURRENCE IN NEW-BFROMSVOK ✺ MUST BE EITHER OVCK, OR AN ORIGINAL OCCURRENCE OF VK INSIDE INTK, EOREPR TO VOK, OR AN VARIABLE UVCJ OF ANOTHER CONVERSION, INSERTED INSIDE INTK. THUS, IT IS SUFFICIENT TO SHOW THAT NEW-BFROMSVOK ✺ CANNOT HAVE ANY MEMBER OF THE FORM OVCK. INDEED, SUPPOSE THAT THERE EXISTS A CONVERSION CONVJ SUCH THAT VJ = VK, CONVJ INSIDE INTK AND THERE IS A VK-FREE PATH FROM CONVJ TO VOK, AS ILLUSTRATED IN FIGURE (2) BELOW

CONVK
↓
HEAD(INTK)

↓
CONVJ

↓
HEAD(INTJ)

←
VOJ

↓
VOK

FIGURE(2)

(WITH NO LOSS OF GENERALITY WE MAY ASSUME THAT NO OTHER CONVERSION OF VK APPEARS ALONG THIS PATH.) WE HAVE TO CONSIDER TWO CASES: EITHER VOK NOT EOREPR ✺ VOJ, OR ELSE THESE OCCURRENCES ARE EOREPR. FIGURE (2) ASSUMES THAT CONVJ, AND CONSEQUENTLY ALL INTJ, ARE STRICTLY INSIDE INTK, HOWEVER, IT IS ALSO POSSIBLE THAT INTJ = INTK AND CONVJ SUCCEEDS CONVK IN THE SAME TARGET BLOCK.
ASSUME FIRST THAT VOK NOT EOREPR VOJ, THEN IN THE CONFIGURATION
OF FIGURE (2), IT FOLLOWS FROM LEAMA A THAT THERE EXISTS
VO = BFROMSVOK2, VO INSIDE INTJ AND VO EOREPR VOJ. HENCE,
VO INSIDE INTJ AND IS NOT EOREPR TO VOK, CONTRADICTING
THE DEFINITION OF INTJ, A SIMILAR ARGUMENT SHOWS THAT THE
OTHER CONFIGURATION MENTIONED ABOVE IS ALSO IMPOSSIBLE IN
THIS CASE.

NOW, ASSUME THAT VOK EOREPR VOJ, WE CLAM THAT IN THIS CASE
INTJ = INTK. INDEED, IF NOT, THEN OBSERVE FROM FIGURE (2)
THAT INTJ MUST BE IN THE LOOP PART OF INTK. HENCE, BY THE DEFINITION
OF INTJ, THERE MUST EXIST VO = BFROMSVOK2, VO INSIDE INTK
BUT OUTSIDE INTJ, AND VO NOT EOREPR VOJ (FOR IF NO SUCH OCCURRENCE
EXISTS, THEN THE FACT THAT INTJ IS IN THE LOOP-PART OF INTK
IMPLIEDS THAT A LARGER INTERVAL THAN INTJ COULD HAVE BEEN CHOSEN
IN THE J-TH ITERATION OF OUR ALGORITHM). IT ALSO FOLLOWS FROM
FIGURE (2) THAT VO = BFROMSVOK2, BECAUSE VO CAN REACH VOK
ALONG THE CONCATENATION OF THE VK-FREE PATH LEADING FROM VO TO
HEAD(INTJ) WITH THE TERMINAL SUBPATH OF THE VK-FREE PATH LINKING
CONVJ WITH VO, WHICH STARTS AT HEAD(INTJ). BUT VO IS OBVIOUSLY
NOT EOREPR TO VOK. THIS CONTRADICTS THE DEFINITION OF INTK, AND SO
INTJ = INTK. BUT IN THIS CASE, SINCE VOJ EOREPR VOK, THE ALGORITHM
WOULD HAVE MERGED CONVK WITH CONVJ, THUS THE ASSERTION IS PROVED.

(2) LET VO BE AN ORIGINAL OCCURRENCE OF SOME VARIABLE V, NOT
PLACED IN WORKPILE. WE CLAIM THAT IF CONVJ IS ANY CONVERSION OF
V, FROM WHICH, AT THE END OF THE ALGORITHM, THERE MAY EXIST A
V-FREE PATH TO VO, THEN THE OUTPUT VARIABLE OVCJ OF CONVJ IS
TO VO. INDEED, IT FOLLOWS FROM LEAMA A THAT THERE EXISTS
VO = BFROMSVOK2 SUCH THAT VOP EOREPR VOJ. HENCE, EITHER OVCJ
EOREPR VOJ, OR ELSE VOJ IS NOT EOREPR TO VO, AND HENCE VOP IS NOT
EOREPR TO VO, THIS VO MUST INITIALLY HAVE BELONG TO THE
WORKPILE, THIS IS A CONTRADICTION, AND IT FOLLOWS THAT
NEW-BFROMSVOK2 CONTAINS ONLY OCCURRENCES EOREPR TO VO.

(3) IT NOW REMAINS TO PROVE THE THEOREM FOR THE VARIABLES
OF THE CONVERSIONS. LET IVCK BE SUCH AN VARIABLE. IF IT IS
OF TYPE GENERAL, THEN THERE IS NOTHING TO PROVE. OTHERWISE,
BY THE DEFINITION OF IVCK, EACH OCCURRENCE IN NEW-BFROMSVIVCK2
IS EITHER EOREPR TO IVCK, OR ELSE IS AN VARIABLE OF SOME
OTHER CONVERSION. THE SECOND CASE IS ILLUSTRATED AS FOLLOWS:

```
CONVJ
   ↓
HEAD(INTJ) ←
   ↓
VOJ
   ↓
HEAD(INTK)
   ↓
VOK
```

**FIGURE (3)**
(Here no relationship of inclusion between the two intervals is implied, note that Figure (3) excludes the case where ConvK and ConvJ are in the same target hluck, for this case has been shown, in step (1), to be impossible.)

Since Vok can be reached along a Vk-free path from ConvK, it follows from Lemma A that there exists Vop + Bfrom[Vok], which reaches Vok through ConvK, such that Vop inside IntJ and Vop Erepr Voj. Hence, Ivck Erepr Vop Erepr Voj Erepr Ivck, so that new-Bfrom[Ivck] indeed contains only occurrences Erepr to Ivck, and this concludes the proof of the theorem.

Q. E. D.

REMARKS:

(1) Our algorithm is intra-procedural in nature. The Bfrom map, however, is inter-procedural, indicating for each link the RC-path(s) through ConvK, such that this link is materialized, we shall interpret Bfrom as an intra-procedural map, in the same way as we did in copy optimization, as follows:

Assume that RC-paths are compacted, so that complete calls are deleted from them. Let Vo be a variable occurrence and let [P, V01] = Bfrom[V02]. Then [P, V01] is interpreted as V01, if P = NULL-path, else, if P terminates at a call point, [P, V01] is interpreted as a dummy occurrence just after the entry to the currently analyzed routine, and if P terminates at a return point, [P, V01] is interpreted as a dummy occurrence just after the corresponding calling instruction.

(2) Note that, though we base our algorithm on interval analysis, the routine flow graph need not be reducible (compare with copy optimization, NL.195). For example, when checking condition (**), we shall make use of a routine that computes the index of the smallest interval in IntSeq, containing Vo and some V01 = Bfrom[V02]. If no such interval exists, the routine will return +IntSeq + 1 (this will not happen, however, in reducible flow graphs).

(3) Our algorithm selects the general type for an IVariable of a conversion linked to more than one split variable, this is done in order to simplify the description of the algorithm, but is not always the best choice. For example, if such an IVariable Ivck is linked to two occurrences, having the reprs set(int), set(char), then a better choice would have been to repr Ivck as set(general). This will make the conversion somewhat faster.

In this alternative approach, the form of each such IVariable Ivck is computed as a disjunction of the reprs of all occurrences.
TO WHICH IVCK IS LINKED, SUCH A DISJUNCTION MUST SATISFY THE
CONDITION THAT NO CONVERSION WILL BE REQUIRED FROM ANY OF THE
FORMS OF THE LINKED OCCURRENCES TO THE MORE GENERAL FORM OF
IVCK (THUS THE DISJUNCTION OF +B AND INT MUST BE GENERAL, EVEN
IF B IS A BASE OF INTEGERS). THE ABOVE THEOREM NOW READS AS
FOLLOWS:

THEOREM C1 UNDER THE SAME HYPOTHESIS AS IN THEOREM B, FOR EACH
VARIABLE OCCURRENCE VO AND EACH VO1 \ implies \ FROMSVO2, VO EOREPR VO1
IF VO IS NOT AN I-VARIABLE OF AN INSERTED CONVERSION; IF IT IS,
THEN THE FORM OF VO INCLUDES THAT OF VO1, IN THE SENSE THAT
THE ASSIGNMENT \ implies \ SPLIT-NAME(VO) := SPLIT-NAME(VO1); \ is a NO-OP.

THE PROOF OF THEOREM C GOES IN MUCH THE SAME WAY AS THE PROOF
OF THEOREM B, WITH A SLIGHT MODIFICATION OF STEP (3).