Users' Guide to Toolpack/1 Tools for Data Dependency Analysis and Program Transformation

by Wayne R. Cowell

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38
Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

---

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report has been reproduced from the best available copy.

Available from the National Technical Information Service
NTIS Energy Distribution Center
P.O. Box 1300
Oak Ridge, TN 37831

Price: Printed Copy A03
Microfiche A01
Users' Guide to Toolpack/1 Tools
for
Data Dependency Analysis and Program Transformation

by

Wayne R. Cowell

Mathematics and Computer Science Division

September 1988

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.
A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.
Users' Guide to Toolpack/1 Tools
for
Data Dependency Analysis and Program Transformation*

by
Wayne R. Cowell

ABSTRACT

This is a guide to the use of a collection of software tools for data dependency analysis and program transformation. The tools are written in Fortran in the tool-writing environment provided by Toolpack/1 and are easily incorporated into a Toolpack/1 installation. We focus on a Unix environment in which Berkeley Unix C-shell scripts provide the user interface to the tools. In this report, the operation of the tools is described in terms of two scripts named args and trans.
1. Introduction

A collection of Unix shell scripts that serves as a user interface to Toolpack/1 is described in [4]. The software tools accessed by the scripts provide various analysis and transformation capabilities of general applicability to Fortran development, maintenance, and conversion. These include semantic analysis, checking for conformance to the Fortran standard, code structuring, declaration uniformization, precision transformation, and formatting. This guide describes the use of scripts called args and trans, now available in the collection, which serve as user interfaces to a set of tools oriented to program development for advanced-architecture machines. The conceptual background for these tools, in particular a characterization of the notion of data independence, is given in [3]. In that paper we avoided operational detail, referring the reader to this guide. Similarly, we recommend [3] to readers of this guide for a discussion of the conceptual underpinnings of the commands and for examples of their use.

The new analysis tools detect parallelism in DO-loops and are primarily intended to aid in adapting existing programs to execute on shared-memory multiprocessors. The new transformation tools are aimed at eliminating data dependencies, thereby introducing parallelism, and at localizing arithmetic in registers. The latter tools are of interest when adapting programs to execute on high-performance vector machines where memory access is a bottleneck relative to arithmetic operations. The key transformations manipulate blocks of code whose order of execution is immaterial, in the sense that the data in shared memory that is read from or written to by any block is not changed by other blocks. Blocks with this property can, in principle, be executed in parallel. Hence, the same underlying machinery for data dependency analysis is used in the implementation of both the transformation tools and the tools that discover parallelism.

2. Preliminary Steps

A Fortran program that is to be analyzed or transformed by these tools should first be checked by the script pfort [4]. Any error cited by pfort should be corrected and each unsafe reference should be checked to ascertain whether it is potentially dangerous. pfort should be rerun after corrections are made, not only to confirm the validity of the changes, but also because certain types of errors cause pfort to terminate rather than proceed with contaminated data. Correcting one error could enable pfort to detect other errors further downstream in its analysis.

Following pfort, we recommend that the program be transformed by decs [4]. This is not essential for the correct operation of the data dependency tools but makes the program more readable and may make data-entry errors that are syntactically correct, such as misspelled variable names, more noticeable to the user.

Whether or not decs is applied, the Fortran program should be structured using stf [4]. This will, in particular, alleviate certain awkward cases of control flow that could affect the behavior of trans commands sp and da.

If the Fortran program contains external references in a block of code that will undergo data dependency analysis (e.g., a DO-loop to be parallelized), the script args, described in Section 3, should be invoked. The resulting information will be particularly
useful in answering questions posed by certain \texttt{trans} commands about the input/output status of subprogram arguments.

After the preceding steps, the user is ready to invoke the script \texttt{trans} (Section 4) which provides the interface to the data dependency analysis and transformation tools. The first \texttt{trans} command should be the DO-loop regularizer \texttt{rg} (Section 4.7), because the other tools assume that DO-loops are regular. The user may then proceed to apply the commands of \texttt{trans} to pursue strategies such as are outlined in [3].

3. \texttt{args}

Invocation:

\begin{verbatim}
args Fortran_source
\end{verbatim}

where \texttt{Fortran_source} is the name of a file containing one or more program units of “Toolpack/1 Target Fortran 77”, which is standard Fortran 77 extended by the addition of certain features (see [1]). \texttt{args} prepares a tab-free copy of \texttt{Fortran_source} and then, using information about the program units in \texttt{Fortran_source} produced by the Toolpack/1 parser and semantic analyzer, creates a report that is sent to standard output and may be redirected to a file. In response to a prompt, the user specifies whether the report is to be “terse” or “verbose”.

To describe the report, it is helpful to have the notion of the “argument passage graph” associated with the programs in \texttt{Fortran_source}. The nodes of the graph are pairs of the form <\texttt{pu_ent}, \texttt{argument_number}> where \texttt{argument_number} is the ordinal number of a dummy argument in the definition of the program unit or entry point named \texttt{pu_ent}. There are no nodes for program units or entry points that have no arguments. A branch, directed from <\texttt{NAME1}, \texttt{ARGNO1}> to <\texttt{NAME2}, \texttt{ARGNO2}>, exists when \texttt{NAME1} references \texttt{NAME2}, passing argument numbered \texttt{ARGNO1} as the actual variable corresponding to dummy variable \texttt{ARGNO2} of \texttt{NAME2}.

An example program and the verbose version of the \texttt{args} report for that program are found in Appendix A. As seen in the example, an \texttt{args} report contains information about each argument of each program unit or entry point in the file \texttt{Fortran_source}, i.e., about each node in the argument passage graph. The data type of the argument is given; if the data type is \texttt{CHARACTER}, the length is given. For each node, at least one and at most five kinds of additional information are given, each kind indicated by a 3-character flag, as follows:

\texttt{Flag}

\begin{verbatim}
--> Argument passage information. The subgraph of the argument passage graph branching from the node being reported can be constructed from this information.

+++ Information about whether or not the argument is updated in the routine being reported or in a routine to which it is passed. If the update status cannot be
\end{verbatim}
determined in a particular routine (because the argument is passed but not otherwise used), that fact is noted.

== Assertions that the argument occurs in an expression in the routine being reported or in a routine to which it is passed. Occurrence in an expression in a particular routine is not reported if the argument is also passed from that routine to another.

*** Warnings about possibly dangerous constructions involving the argument. These are not necessarily harmful, but the user should check to make sure they are benign. In the example in Appendix A, the report warns that the main program passes an expression as the second argument in the call of subroutine subr1. No harm is done since the argument is not updated. (This construction would be flagged by pfort if the argument were updated.)

!!! Information about whether the argument is an input and/or output variable. This is called the "data dependency input/output status" and is presented in the form {input status}/output status where input status and output status are "y", "n", or "?" indicating, respectively, "yes", "no", or "don't know". For example, "y/n" means that the argument is an input variable (is read from memory) but not an output variable (not written to memory, i.e., not updated).

Obtaining the data dependency input/output status is a central purpose of args because this information may be used to answer questions posed by the tools invoked by trans. A status of "?" means that the information produced by the Toolpack parser and semantic analyzer does not permit an unambiguous determination of the status. (These tools do not perform data flow analysis.) When a question about the status of such an argument arises, the user must examine the program.

The data dependency input/output status for each argument is always included in the report, whether the terse or verbose option is selected. The terse report omits the information flagged by "-->", "+++", "===", and "***", although this information is used internally in the determination of the data dependency input/output status.

4. trans

Invocation:

trans Fortran_source

where Fortran_source is the name of a file containing one or more program units of "Toolpack/7 Target Fortran 77", which is standard Fortran 77 extended by the addition of certain features (see [1]). trans uses a tab-free copy of Fortran_source. trans replaces the script ucs [4], which provided access to forerunners of the transformation tools.
When first invoked, trans enters Mode 1, described as follows:

Unix less is active on a Toolpack-formatted copy of the current version of the program, which we shall call the "subject program". The editor vi may be entered by typing v; less is active on the edited file upon leaving vi.

Mode 1 terminates when less is terminated by typing q. trans then enters Mode 2, described as follows:

The prompt "trans:" is given, and the user responds by typing one of thirteen commands. The command qt (quit) terminates trans. The remaining commands invoke various operations on the subject program as it existed when Mode 1 terminated.

Mode 2 terminates when the operation invoked by the command is finished. trans then re-enters Mode 1. less is active on the new subject program as it exists following the operation invoked by the command.

We used the word "tool" in [3] to denote essentially the same thing that we denote here by "trans command". We will discuss functionality at the level of trans commands, the same level as the discussion of "tools" in [3], but here we do not hide the fact that trans invokes a sequence of "tools" (in the Toolpack/1 sense) for each command issued. The user will be aware that these tools are active because he will receive messages such as "[ISTLP Normal Termination]" or "Tool Termination Status: TERM-FLAG_1". These information messages may be safely ignored if one is not interested in the underlying processes.

There are three error categories to which trans reacts in the following ways:

1. Scanning errors (i.e., errors detected by the Toolpack/1 scanner) in the original subject program. trans gives a diagnostic message and terminates.

2. Scanning errors introduced while editing in Mode 1. The current subject program is saved in a file named _transEDERR; trans gives a diagnostic message and terminates.

3. Errors detected by underlying Toolpack/1 tools other than the scanner. trans gives a diagnostic message and re-enters Mode 1 with the subject program in the state it was in when Mode 1 terminated. Such errors include parsing errors in the original subject program or introduced while editing, errors in trans directives that the user has inserted in the code (see Appendix B), and errors arising from the lengths of strings or table sizes. Most of these errors can be corrected in Mode 1 and the command reissued. A few error messages refer to internal system matters (such as the sizes of tables) that the user should refer to the developers.
The trans command issued in Mode 2 must be one of the following:

**cd** Condense sequences of DO-loops satisfying certain conditions.

**ci** Condense simple block IF sequences.

**da** Produce a report, in the form of comments in the code, of the dependency sets for a specified block of statements.

**de** Re-enter Mode 1 with less active on the current version of the program.

**pd** Perform DO-loop peeling transformations on specified DO-loops.

**qt** Terminate trans.

**rg** Perform regularizing transformations on DO-loops.

**sb1** Perform substitution/elimination transformations in sequences of assignment statements that satisfy certain requirements.

**sb2** Perform substitution/elimination transformations in sequences that can include assignment statements, logical IFs whose objects are assignment statements, block IF, ELSE IF, ELSE, END IF, DO, and CONTINUE statements, and that satisfy certain requirements.

**si** Perform secondary loop index transformations on specified DO-loops.

**sp** Examine specified DO-loops to determine whether they can be split and parallelized. For those that can, insert a report about shared and local variables in the form of a compiler instruction for the Sequent Balance. For those that cannot, report the dependencies.

**ud** Unroll DO-loops satisfying certain conditions to a specified depth.

**xd** Interchange a specified pair of DO-loops.

trans is interactive. In particular, the commands cd, sb1, sb2, si, sp, and xd invoke Toolpack/1 tools that refer certain questions about data dependencies to the user. The questions can always be answered "yes" or "no". The user's response, as indicated in the prompt, is "y" for "yes", "n" or a carriage return for "no", and "p" to request that a copy of the pertinent Fortran be printed and the question repeated. (The handling
of "p" is system dependent and is determined by the installer.) In the operation of these tools, the user is given an opportunity to override dependency findings if option override_deps is enabled in the options file _imropts. (See Appendix C and Section 4.11.) The same tools also obtain the input/output status of arguments in function and subroutine calls by querying the user. Answering these queries is greatly facilitated by using the output from args (see Section 3). The answers are stored in a file named "_.callarg" in the working directory; if needed again, they are extracted from this file. Fortran names and segments of Fortran programs that are displayed when asking questions of the user always appear in upper case.

Currently, the underlying machinery for data dependency analysis treats Fortran statements of the following types: assignment, logical IF, block IF, ELSE IF, ELSE, END IF, CONTINUE, DO, GO TO, and CALL. Generally, this is sufficient to cover the blocks of Fortran code that can benefit from application of the tools.

Side effects involving COMMON and EQUIVALENCE are not checked by the tools during data dependency analysis. However, warnings are issued when a variable in a COMMON or EQUIVALENCE statement appears in code undergoing data dependency analysis and is updated (i.e., appears in an output dependency set).

The trans commands da, pd, sb2, si, sp, and xd require that the portions of the program that they analyze or transform be identified by special comments, called source-embedded directives (or simply directives), that begin with the string "*$XX$", where "XX" is a pair of case-insensitive alphabetic characters. The remainder of the directive comment line, following the second "$", is arbitrary. In the discussions of these tools we will specify which directive shall precede and which shall follow some Fortran construct. By "precede" ("follow") we mean that the block of comments that immediately precedes (follows) the construct shall include the indicated directive, but none of the other directives used by trans commands.

The directives are summarized in Appendix B. Special files used by trans are summarized in Appendix C.

In the sections that follow, each trans command will be discussed and simple examples will be given. We will indicate the usual context in which each command is used, but it must be remembered that the tools are used in concert as part of a strategy for adapting existing Fortran to execute on advanced-architecture machines. The discussion of this strategy in [3] is essential to an understanding of how the tools are used.

You will note that Fortran output from the transformation tools often contains unsimplified expressions, in particular, redundant parentheses. An expression simplifier is under development for use by future versions of these tools.

4.1. cd

The condensing transformation maps a pair of contiguous DO-loops,
with the same DO variable and the same parameters, and whose ranges satisfy certain dependency conditions, into the single DO-loop

\[
\text{DO } 3 \ i = e_1, e_2, e_3 \\
\quad \text{R1} \\
\quad \text{R2} \\
\quad \text{CONTINUE}
\]

Condensing usually follows unrolling (Section 4.12), and the loops are assumed to be regular (Section 4.7).

If two DO-loops have the same DO variable and the same expression as the initial parameter, we say that they are "e1 equivalent". Similarly defined are "e2 equivalent" DO-loops and "e3 equivalent" DO-loops. cd searches for sequences of two or more contiguous DO-loops that are e3 equivalent. If the first two in the sequence are e1 and e2 equivalent, and satisfy the dependency conditions, they are condensed and the remaining DO-loops are treated as a new sequence. If the first two are either not e1 and e2 equivalent or fail to satisfy the dependency conditions, the sequence beginning with the second DO-loop is treated as a new sequence. If the new sequence contains two or more DO-loops, it is treated in the same way as the original sequence, i.e., the first two loops are examined for e1 and e2 equivalence, etc.; if not, the process terminates.

When any pair of DO-loops is condensed, the termination status of the underlying Toolpack/1 condensing tool signals that another pass is to be made. trans then invokes the tool again, taking as input the transformed Fortran from the previous invocation. The tool is re-invoked in this way until no further condensing takes place. To illustrate, ed will transform the sequence of DO-loops

\[
\text{DO } 99997 \ i = 1, m \\
\quad \text{cr(i)} = (\text{cr(i)} + \text{ar(i,j)}*\text{br(1,j)}) - \text{ai(i,j)}*\text{bi(1,j)} \\
\quad \text{ci(i)} = (\text{ci(i)} + \text{ar(i,j)}*\text{bi(1,j)}) + \text{ai(i,j)}*\text{br(1,j)} \\
\quad \text{CONTINUE} \\
\text{DO } 99996 \ i = 1, m \\
\quad \text{cr(i)} = (\text{cr(i)} + \text{ar(i,j+1)}*\text{br(1,j+1)}) - \text{ai(i,j+1)}*\text{bi(1,j+1)} \\
\quad \text{ci(i)} = (\text{ci(i)} + \text{ar(i,j+1)}*\text{bi(1,j+1)}) + \text{ai(i,j+1)}*\text{br(1,j+1)} \\
99996 \quad \text{CONTINUE} \\
\text{DO } 99995 \ i = 1, m \\
\quad \text{cr(i)} = (\text{cr(i)} + \text{ar(i,j+2)}*\text{br(1,j+2)}) - \text{ai(i,j+2)}*\text{bi(1,j+2)} \\
\quad \text{ci(i)} = (\text{ci(i)} + \text{ar(i,j+2)}*\text{bi(1,j+2)}) + \text{ai(i,j+2)}*\text{br(1,j+2)} \\
99995 \quad \text{CONTINUE}
\]
DO 99994 i = 1,m
  cr(i) = (cr(i)+ar(i,j+3)*br(1,j+3)) - ai(i,j+3)*bi(1,j+3)
  ci(i) = (ci(i)+ar(i,j+3)*bi(1,j+3)) + ai(i,j+3)*br(1,j+3)
99994 CONTINUE

into

DO 59999 i = 1,m
  cr(i) = (cr(i)+ar(i,j)*br(1,j)) - ai(i,j)*bi(1,j)
  ci(i) = (ci(i)+ar(i,j)*bi(1,j)) + ai(i,j)*br(1,j)
  cr(i) = (cr(i)+ar(i,j+1)*br(1,j+1)) - ai(i,j+1)*bi(1,j+1)
  ci(i) = (ci(i)+ar(i,j+1)*bi(1,j+1)) + ai(i,j+1)*br(1,j+1)
59999 CONTINUE

DO 59998 i = 1,m
  cr(i) = (cr(i)+ar(i,j+2)*br(1,j+2)) - ai(i,j+2)*bi(1,j+2)
  ci(i) = (ci(i)+ar(i,j+2)*bi(1,j+2)) + ai(i,j+2)*br(1,j+2)
  cr(i) = (cr(i)+ar(i,j+3)*br(1,j+3)) - ai(i,j+3)*bi(1,j+3)
  ci(i) = (ci(i)+ar(i,j+3)*bi(1,j+3)) + ai(i,j+3)*br(1,j+3)
59998 CONTINUE

on the first pass. On the second pass, these two loops will be transformed into

DO 59997 i = 1,m
  cr(i) = (cr(i)+ar(i,j)*br(1,j)) - ai(i,j)*bi(1,j)
  ci(i) = (ci(i)+ar(i,j)*bi(1,j)) + ai(i,j)*br(1,j)
  cr(i) = (cr(i)+ar(i,j+1)*br(1,j+1)) - ai(i,j+1)*bi(1,j+1)
  ci(i) = (ci(i)+ar(i,j+1)*bi(1,j+1)) + ai(i,j+1)*br(1,j+1)
  cr(i) = (cr(i)+ar(i,j+2)*br(1,j+2)) - ai(i,j+2)*bi(1,j+2)
  ci(i) = (ci(i)+ar(i,j+2)*bi(1,j+2)) + ai(i,j+2)*br(1,j+2)
  cr(i) = (cr(i)+ar(i,j+3)*br(1,j+3)) - ai(i,j+3)*bi(1,j+3)
  ci(i) = (ci(i)+ar(i,j+3)*bi(1,j+3)) + ai(i,j+3)*br(1,j+3)
59997 CONTINUE

To illustrate user/tool interaction to resolve possible dependencies, consider the following pair of DO-loops:

DO 99997 i = 1,j - 1
  b(i) = b(i) + a(j,i)*b(j)
99997 CONTINUE
DO 99996 i = 1,j - 1
  b(i) = b(i) + a(j+1,i)*b(j+1)
99996 CONTINUE

cd outputs the message:

The following two DO-loops are candidates for condensation:

DO 99997 I=1,J -1
  B(I)=B(I)+A(J,1)*B(J)
99997 CONTINUE
DO 99996 I=1,J-1
   B(I)=B(I)+A(J+1,I)*B(J+1)
99996 CONTINUE

Can the string

\[ B([1:J-1]) \]

in the output set of a statement in the first DO-loop for one value of the loop variable refer to the same memory location as the string

\[ B(J+1) \]

in the input set of a statement in the second DO-loop for a different value of the loop variable?

The loop variables have been replaced by their ranges in these strings.

\[ y = \text{yes} \]
\[ n \text{ or RETURN } = \text{no} \]
\[ p = \text{Print a copy of the DOs and repeat the question} \]

Distinctiveness of the values of the loop variable in the two sets of strings is automatic in this case because the second set, consisting of the string \( B(J+1) \), is unaffected by the loop variable. As noted in the question and as seen in the first string, the loop variable is replaced by its range. The tool is asking a question that the user can easily answer "no"; he may give his answer by typing either "n" or a carriage return. cd then asks

Can the string

\[ B(J) \]

in the input set of a statement in the first DO-loop for one value of the loop variable refer to the same memory location as the string

\[ B([1:J-1]) \]

in the output set of a statement in the second DO-loop for a different value of the loop variable?

Again the answer is "no". cd resolves any other dependency issues without user interaction. For example, the tool determines that the references to \( B(I) \) on the left sides of the two assignment statements do not cause a dependency, because the dependency conditions for loop condensation permit us to assume that the values of the loop variable in the two loops are different, i.e., \( B(I) \) is referencing a different memory location in the first loop than in the second loop. (The pertinent dependency conditions are derived from Theorem 2 in [2].) cd transforms the two DO-loops into
DO 59999  i = 1,j - 1
    b(i) = b(i) + a(j,i)*b(j)
    b(i) = b(i) + a(j+1,i)*b(j+1)
      59999  CONTINUE

4.2. ci

A simple block IF sequence is a sequence of contiguous block IFs,

    IF (Logic.1) THEN
    Block.1
    END IF
    IF (Logic.2) THEN
    Block.2
    END IF
    .
    .
    .
    IF (Logic.n) THEN
    Block.n
    END IF

where Block.1, Block.2, ..., Block.n are blocks of code limited to assignment statements, DO statements, and CONTINUE statements. Moreover, the only statements that may be labelled are CONTINUE statements that are DO terminators. ci transforms ("condenses") the simple block IF sequence into

    IF (Logic.1 .AND. Logic.2 .AND. ... .AND. Logic.n) THEN
    Block.1
    Block.2
    .
    .
    Block.n
    ELSE
    IF (Logic.1) THEN
    Block.1
    END IF
    IF (Logic.2) THEN
    Block.2
    END IF
    .
    .
    .
IF (Logic.n) THEN
  Block.n
END IF
END IF

Simple block IF sequences generally result from unrolling a DO-loop whose range contains a block IF. It frequently happens that the sequence Block.1, ... Block.n in the first branch above may be further transformed by substitution/elimination (Sections 4.8 and 4.9), loop peeling (Section 4.5), and loop interchange (Section 4.13) to create a sequence of DO-loops for condensation (Section 4.1).

4.3. da

A string of Fortran characters representing a memory reference is said to be in the input set if the execution of the statement containing the string results in a memory location designated by the string being read from. Similarly, a string is in the output set if the execution of the statement containing the string results in a memory location designated by the string being written to.

da exhibits the "raw material" used in data dependency analysis. The command produces a report of the input and output sets, in the form of comments in the code, for each block of Fortran code preceded by the directive "**$BB$" and followed by the directive "**$EB$". To illustrate, the block

```
*BB$
  DO 40 j = 1,m99998,2
    DO 99997 i = 1,j - 1
      b(i) = b(i) + a(j,i)*b(j)
  99997 CONTINUE
    DO 99996 i = 1,j - 1
      b(i) = b(i) + a(j+1,i)*b(j+1)
  99996 CONTINUE
    b(j) = (a(j,j)*b(j)) + a(j+1,j)*b(j+1)
    b(j+1) = a(j+1,j+1)*b(j+1)
  40 CONTINUE
*EB$
```

would, after the application of da, appear as

```
*BB$
  DO 40 j = 1,m99998,2
    DO 99997 i = 1,j - 1
      b(i) = b(i) + a(j,i)*b(j)
  99997 CONTINUE
    DO 99996 i = 1,j - 1
      b(i) = b(i) + a(j+1,i)*b(j+1)
  99996 CONTINUE
    b(j) = (a(j,j)*b(j)) + a(j+1,j)*b(j+1)
    b(j+1) = a(j+1,j+1)*b(j+1)
  40 CONTINUE
```
*** Dependency sets for this block ***

*** Input: A(J+1,I), A(J+1,J), A(J+1,J+1), A(J,I), A(J,J), B(I), B(J),
* B(J+1), I, J, M99998

*** Input entries defined external to block: A(J+1,I), A(J+1,J),
* A(J+1,J+1), A(J,I), A(J,J), B(I), B(J), B(J+1), M99998

*** Output: B(I), B(J), B(J+1), I, J

*** Output scalars defined before used in block: I, J

Entries may, of course, appear in both sets; indeed, in the example, every member of the output set is in the input set.

As seen in the example, two additional sets are exhibited by da. The set labelled "Input entries defined external to block" are strings that were not in the output set when added to the input set. The set labelled "Output scalars defined before used in block" are strings representing scalars that were not in the input set when added to the output set. The tool does not account for the flow of control during execution of the program. Thus, the report is constructed as if all IF conditions were true and every DO-loop traversed at least once.

4.4. de

This command causes a return to Mode 1 with less active on the current version of the program.

4.5. pd

We say that a regular DO-loop (Section 4.7) is peeled from the top if its range (except for the terminating CONTINUE) is replicated immediately before the DO statement using the first value of the DO variable and with the second value specified as e1 in the DO statement itself. Similarly, the DO-loop is peeled from the bottom if its range (except for the terminating CONTINUE) is replicated immediately after the terminating CONTINUE using the final value of the DO variable and with the penultimate value specified as e2 in the DO statement itself.

pd applies the indicated peeling transformation to a DO-loop preceded by the directive ""*$PT$"" (peel from the top) or the directive ""*$PB$"" (peel from the bottom). A message from the tool reminds the user to ensure that the DO-loop is regular and is executed at least once. For example, pd transforms

*$PB$

```
DO 99996 i = 1, j
   b(i) = b(i) + a(j+1,i)*b(j+1)
99996 CONTINUE
```
DO 99996 i = 1, j - (1)
   b(i) = b(i) + a(j+1, i)*b(j+1)
99996 CONTINUE
   b((j)) = b((j)) + a(j+1, (j))*b(j+1)

If the redundant parentheses are a problem, they must, at present, be removed by hand, i.e., while in the editor in Mode 1. As mentioned in Section 4, an expression simplifier is under development.

4.6. qt

Terminate trans. When trans was invoked, a subdirectory named `.TOOLPACK` was created to contain intermediate files used by the underlying tools. This subdirectory is removed by `qt` and is thus invisible to the user during normal operation. When trans terminates, the working directory contains the file `.transLOG`, which lists the commands issued in Mode 2 during the session. In case of abnormal termination, `.TOOLPACK` and certain files left by trans may be removed by invoking the script discard. (See [4].)

4.7. rg

The DO-loop

```fortran
DO 1 i = e1, e2, e3
   <BLOCK>
   1 <terminating statement>
```

is said to be regular [2] if

(1) `<terminating statement>` is a CONTINUE.

(2) There are no transfers to `<terminating statement>` from `<BLOCK>`.

(3) When the parameters are such that the loop is traversed at least once, then \( e2 = e1 + m*e3 \) for some nonnegative integer \( m \).

(4) The value or the DO variable is not used after the satisfaction of the loop.

Any DO-loop can be transformed into a regular DO-loop, but transformations that guarantee the satisfaction of the fourth condition would involve renaming the DO variable, and the resulting changes in the Fortran may inhibit the application of other transformations. We have not provided the means to transform the loop automatically so that (4) is guaranteed. The user should check this condition by inspecting the code. If the value of a DO variable is used after the satisfaction of the loop, transformations such as loop peeling (Section 4.5) and loop unrolling (Section 4.12) could change the semantics of the program.

`rg` transforms every DO-loop in the program to a form in which the first three regularity conditions are satisfied. This command also performs various transformations that facilitate the application of other tools. For example, it converts logical IFs in the
program to block IFs and inserts before each DO statement the directive ‘‘*SPS’’ that flags DO-loops to be processed by sp (Section 4.11). To illustrate, the DO-loop

\[
\begin{align*}
\textbf{DO } 2 & \ i = 10,1,-2 \\
& \text{IF } (i \text{ .EQ. } 2) \ \text{GO TO 2} \\
& a(i) = \text{fun}(i) \\
2 & b(i) = \text{fun}(i+1)
\end{align*}
\]

would be transformed by rg into

*SPS* This is DO LOOP No: 1

\[
\begin{align*}
\text{DO } 20 & \ i = 10,10 + \frac{(1-(10)+(2))}{(-2)} \cdot (-2),-2 \\
\end{align*}
\]

C*** DO-LOOP transformed to REGULAR-DO

C*** LOGICAL-IF transformed to block IF

IF (i .EQ. 2) THEN

GO TO 10

END IF

\[
\begin{align*}
& a(i) = \text{fun}(i) \\
& \text{CONTINUE} \\
10 & \text{CONTINUE} \\
& b(i) = \text{fun}(i+1) \\
& \text{CONTINUE}
\end{align*}
\]

As noted in Section 2, we recommend that rg be the first trans command applied to any code.

4.8. sb1

substitution/elimination is performed by eliminating an assignment statement \(<left>\) = \(<right>\), where \(<left>\) is redefined by a subsequent assignment statement, and substituting \(<right>\) for occurrences of \(<left>\) in statements up to and including the right side of the first redefinition. We require that none of the intervening statements redefine any scalar or array reference that occurs in \(<right>\). Further, we require that if \(<left>\) is an array reference, then any reference to the same array that occurs in the input sets of either the intervening statements or the redefinition statement be either identical to \(<left>\) or else not refer to any elements of the array to which \(<left>\) can refer.

sb1 performs substitution/elimination in sequences consisting entirely of unlabelled assignment statements that do not contain a function reference with an output argument. An output argument is either a scalar whose value is different after the reference, or an array any of whose elements has a different value after the reference. sb1 will, for example, transform

\[
\begin{align*}
a & = b + c \\
d & = a - e \\
a & = e + f \\
\end{align*}
\]

into
\begin{align*}
    d &= (b+c) - c \\
    a &= e + f
\end{align*}

In the sequence
\begin{align*}
    a &= b + c \\
    b &= a - e \\
    a &= e + f
\end{align*}

\texttt{sbl} recognizes that substitution/elimination cannot take place because of the dependency involving \( b \). The tool will then attempt to permute the last statement (the redefinition) upwards in the sequence to immediately before the statement involving the dependency. If it were possible to do this, substitution/elimination would take place with the redefinition statement in its new position. However, in this example, the second and third statements cannot be permuted because of a dependency involving \( a \).

Interaction between \texttt{sbl} and the user to resolve data dependencies can be illustrated by changing \( a \) and \( b \) to arrays in the last example, i.e.,
\begin{align*}
    a(i) &= b(j) + c \\
    b(k) &= a(m) - e \\
    a(i) &= e + f
\end{align*}

\texttt{sbl} outputs the following (we will call this \textit{output 1} for reference below):

\textbf{Subsequence considered for transformation by substitution/elimination:}
\begin{align*}
    1 & \quad A(I)=B(J)+C \\
    2 & \quad B(K)=A(M)-E \\
    3 & \quad A(I)=E+F
\end{align*}

Can the string
\begin{equation*}
    B(J)
\end{equation*}
on the right side of the first statement refer to the same memory location as the string
\begin{equation*}
    B(K)
\end{equation*}
on the left side of statement 2?

If the user answers "yes", \texttt{sbl} responds

\textit{Statement 2 is an assignment to a member of the input dependency set of the first statement.}

An attempt will be made to permute the last statement (the redefinition) to immediately before statement 2.
As it examines whether the permutation can take place, sb1 outputs

For information, the code block is:

\[ B(K) = A(M) - E \]

and the redefinition statement is:

\[ A(I) = E + F \]

Can the string

\[ A(M) \]

in the input set of a statement in the code block refer to the same memory location as the string

\[ A(I) \]

on the left side of the redefinition statement?

If the answer is "no", sb1 makes the permutation followed by substitution/elimination, resulting in

\[ a(i) = e + f \]
\[ b(k) = a(m) - e \]

Return now to output 1 identified above. Suppose the user answered "no", meaning that \( b(j) \) and \( b(k) \) are never references to the same memory location. sb1 then asks

Can the string

\[ A(M) \]

in the input set of one or more statements from Statement 2 onward refer to the same memory location as the string

\[ A(I) \]

on the left side of Statement 1?

If the user answered "yes", substitution/elimination will not occur. (The allowable set of values of \( m \) overlaps, but is not necessarily identical to, the allowable set of values of \( i \); the tool does not know whether the substitution would be correct.) If the user answered "no", substitution/elimination will occur, resulting in

\[ b(k) = a(m) - e \]
\[ a(i) = e + f \]
Whenever a substitution/elimination transformation is performed, the termination status of the underlying Toolpack/l tool signals that another pass is to be made. trans then invokes the tool again, taking as input the transformed Fortran from the previous invocation. The tool is re-invoked in this way until no further substitution/elimination takes place.

sb1 is often invoked after ed to reduce multiple assignments to the same array. Thus the DO-loop

```
DO 59997 i = 1,m
  cr(i) = (cr(i)+ar(i,j)*br(1,j)) - ai(i,j)*bi(1,j)
  ci(i) = (ci(i)+ar(i,j)*bi(1,j)) + ai(i,j)*br(1,j)
  cr(i) = (cr(i)+ar(i,j+1)*br(1,j+1)) - ai(i,j+1)*bi(1,j+1)
  ci(i) = (ci(i)+ar(i,j+1)*bi(1,j+1)) + ai(i,j+1)*br(1,j+1)
  cr(i) = (cr(i)+ar(i,j+2)*br(1,j+2)) - ai(i,j+2)*bi(1,j+2)
  ci(i) = (ci(i)+ar(i,j+2)*bi(1,j+2)) + ai(i,j+2)*br(1,j+2)
  cr(i) = (cr(i)+ar(i,j+3)*br(1,j+3)) - ai(i,j+3)*bi(1,j+3)
  ci(i) = (ci(i)+ar(i,j+3)*bi(1,j+3)) + ai(i,j+3)*br(1,j+3)
59997 CONTINUE
```

in the example of Section 4.1, would be transformed into

```
DO 59997 i = 1,m
  cr(i) = (((((cr(i)+ar(i,j)*br(1,j))-ai(i,j)*bi(1,j))
          & +ar(i,j+1)*br(1,j+1))-ai(i,j+1)*bi(1,j+1))
          & +ar(i,j+2)*br(1,j+2))-ai(i,j+2)*bi(1,j+2))
          & +ar(i,j+3)*br(1,j+3)-ai(i,j+3)*bi(1,j+3)
  ci(i) = (((((ci(i)+ar(i,j)*bi(1,j))+ai(i,j)*br(1,j))
          & +ar(i,j+1)*bi(1,j+1))+ai(i,j+1)*br(1,j+1))
          & +ar(i,j+2)*bi(1,j+2))+ai(i,j+2)*br(1,j+2))
          & +ar(i,j+3)*bi(1,j+3)) + ai(i,j+3)*br(1,j+3)
59997 CONTINUE
```

Several passes of the underlying tool would be made.

In general, when the resulting code is compiled, there will be fewer references to memory per arithmetic operation because more of the intermediate results will be stored in registers. This is an important source of improved performance, especially for high-performance computers.

4.9. sb2

sb2 performs substitution/elimination in sequences that consist entirely of assignment statements, logical IF statements of the form IF (expr) <statement> (where <statement> is an assignment statement), block IF, ELSE IF, ELSE, END IF, DO, and CONTINUE statements. Further, a statement in the sequence may be labelled if and only if it terminates a DO-loop, and no statement may contain a function reference with an output argument. The "original" definition must be an assignment statement, and the redefinition may be made in either a DO statement or an assignment statement that is not the object of a logical IF statement. Finally, unless the user interferes by forcing the
consideration of a statement sequence, as described below, neither the original definition nor the redefinition may be contained in the range of a DO-loop or in a branch of a block IF statement.

The user may force sb2 to consider a sequence that is *entirely* within the range of a DO-loop or a branch of a block IF by preceding the sequence with the directive "**$BG$**" and following the sequence with the directive "**$EN$**". The effect will be to consider the code bracketed by these directives as if it were not in a DO-loop range or branch of a block IF. Within the bracketed block, all the requirements for statement sequences must be met in order for sb2 to perform substitution/elimination. To illustrate, sb2 will treat

\[
\begin{align*}
\text{IF } (x(jx).\text{NE.}0) \text{ THEN} & \\
*BG & \\
\quad y(j) = y(j) + \alpha x(jx) a(j,j) \\
\quad \text{DO } 77776 \quad i = j + 1, n \\
\quad \quad y(i) = y(i) + y(j) x(jx) a(j,i) \\
\quad \quad \text{CONTINUE} \\
\quad \text{DO } 77775 \quad i = 1, j - 1 \\
\quad \quad y(i) = y(i) + y(j) x(jx+incx) a(i, j+1) \\
\quad \quad \text{CONTINUE} \\
\quad y(j) = y(j) + \alpha x(jx+incx) a(j, j+1) & \\
\text{END IF} & \\
\end{align*}
\]

as if the IF and END IF statements were not present. sb2 poses several questions to the user about whether \(y(j)\) could be the same reference as any of the occurrences of \(y(i)\) in the ranges of the DO-loops. In each case, the answer is "no" because the loop variable does not assume the same value as \(j\). The transformed code is

\[
\begin{align*}
\text{IF } (x(jx).\text{NE.}0) \text{ THEN} & \\
*BG & \\
\quad \text{DO } 77776 \quad i = j + 1, n \\
\quad \quad y(i) = y(i) + (y(j) + \alpha x(jx) a(j,j)) x(jx) a(j,i) \\
\quad \quad \text{CONTINUE} \\
\quad \text{DO } 77775 \quad i = 1, j - 1 \\
\quad \quad y(i) = y(i) + (y(j) + \alpha x(jx) a(j,j)) x(jx+incx) a(i, j+1) \\
\quad \quad \text{CONTINUE} \\
\quad \text{DO } 77775 \quad i = j + 1 \\
\quad \quad y(i) = y(i) + (y(j) + \alpha x(jx) a(j,j)) x(jx+incx) a(j, j+1) & \\
\text{END IF} & \\
\end{align*}
\]

The substitution/elimination transformations performed by sb1 (Section 4.8) could be performed by sb2, but the user would be required to force consideration of sequences of assignment statements that appear in the ranges of DO-loops and branches of block IFs. sb1, on the other hand, pays no attention to the context of a sequence of assignment statements and is thus more convenient in some situations.
Consider the DO-loop

```fortran
DO 10 i = e1, e2, e3
  Assign.Block
  iy = iy + (expr)
10 CONTINUE
```

where `iy` is an integer scalar variable, `Assign.Block` is a block of assignment statements that may read from, but do not assign to, `iy`, and `expr` is an expression not containing the loop variable. Further, none of the statements in the range can imply a data type conversion, nor can any function referenced in any of the statements in the range have an output argument. `expr` must be parenthesized unless it is a scalar variable or a constant.

This is a common construction in scientific software. We call `iy` a secondary loop index because its value depends solely on which iteration is being executed. (Presumably, `iy` would have been initialized before the loop was executed.) Formally, `iy` is a dependency that will, in particular, inhibit the transformations performed by `cd` (Section 4.1) and `sp` (Section 4.11). However, certain substitutions permit the assignment to `iy` to be eliminated, thereby eliminating the dependency.

The command `si` will cause each DO-loop preceded by the directive `"*$SI*"` to be checked for satisfaction of the preceding requirements. If the requirements are satisfied, the DO-loop will be transformed to

```fortran
DO 10 i = e1, e2, e3
  <Assign.Block with iy + (expr)*((i-e1)/e3)) substituted for iy>
10 CONTINUE
  iy = iy + (expr)*max(0,(e2-e1+e3)/e3)
```

The underlying Toolpack/1 tool has been provided with a list of "approved" functions, all of whose arguments are input only. If the tool encounters a function not on the list, it will query the user about this requirement. `si` invokes underlying substitution/elimination machinery, so the user will receive information messages that come from the same underlying tool invoked by `sbl` (Section 4.8) and `sb2` (Section 4.9).

As an example, `si` transforms

```
*$SI*
DO 10 i = 1,n
  b(i) = a(i,iy)
  a(i,iy) = a(i,iy) + min(i,iy)
  iy = iy + 5
10 CONTINUE
```
DO 10 i = 1,n
   b(i) = a(i, (iy+5* ((i- (1)))))
   a(i, (iy+5* ((i- (1))))) = a(i, (iy+5* ((i- (1))))) +
      & min(i, (iy+5* ((i- (1)))))
10 CONTINUE
   iy = iy + 5*max(0, ((n)- (1)+1))

4.11. sp

We say that a DO-loop can be split and parallelized if iterations of the loop for
different values of the DO variable can be executed asynchronously on different processors. sp examines whether a DO-loop preceded by the directive "*$SP$" satisfies the
data-independence conditions that permit it to be split and parallelized. (As noted in Section 4.7, one effect of the command rg is to insert this directive before every DO-loop in the program.) Tool analysis is supplemented by interaction with the user.

sp prompts the user for the name of a dependency report file. The user responds
with either "<carriage return>", indicating that no report file is to be generated, or with a Unix file name.

For example, suppose sp is invoked on a program containing the DO-loop

*$SP$
   DO 77776 i = j + 1,n
      jx = i + 1
      y(i) = y(i) + y(j)*x(jx)*a(j,i)
77776 CONTINUE

The user will be asked the question

The loop variable has been replaced by its range in the following question.

For some value of the loop variable, can the corresponding array reference

   Y(J+1:N])

in the output set of one or more statements be to the same
memory location as the array reference

   Y(J)

in the input set of one or more statements for a different value
of the loop variable?

The loop variable does not appear in the second set; its range would replace it if it did. The question is whether J lies in the range [J+1:N]. Clearly, the answer is "no", so the user may respond with "'n" or "<carriage return>". sp inserts comments that precede the DO-loop as follows:

*** Loop can be split and parallelized.
CSDOACROSS SHARE(J,N,Y,X,A), LOCAL(JX)
DO 77776 i = j + 1,n
   jx = i + 1
   y(i) = y(i) + y(j)*x(jx)*a(j,i)
77776 CONTINUE

The comment "C$DOACROSS ..." is an instruction to the Sequent Balance compiler [5] to generate code that assigns different iterations of the DO-loop to different processors. The indicated shared variables cause no conflicts from data dependencies, as determined by sp. The variable jx would cause a dependency if shared, but sp has determined that jx is defined before being used in every iteration of the loop. The instruction to the compiler specifies that each processor is to have a local copy of jx. The code to generate this compiler instruction is contained in one routine in the tool source; the same information from the analysis could be used to generate similar specifications for other compilers. However, independently of any particular architecture, the "C$DOACROSS ..." comment may be regarded as the tool's report to the user about shared and local variables.

Local variables, such as jx in this example, are scalars that are defined before they are used. Normally, if the first definition of the scalar is in the range of a block IF, it will not be considered a local variable (and will be a dependency) because the tool has no way of knowing whether a particular branch of a block IF is executed. On the other hand, the user may realize that the scalar is always defined, for example if there is a definition in every branch of the block IF. An option may be enabled that causes every scalar that is defined before it is used to be accepted as a local variable, even if its definition is in the range of a block IF. This option is enabled by including a line consisting of the string "accept_locals_in_if_blocks" in the options file _imropts. (See Appendix C.)

For the above DO-loop, the dependency report states

The following scalar temporaries are not considered dependencies:

   JX

DO-loop can be split and parallelized.

The user/tool interaction for the following example is described in [3].

*$SP$ This is DO LOOP No: 2
DO 30 j = 1,iendm1
*$SP$ This is DO LOOP No: 3
DO 20 i = 1,iendm2
   work1(i) = ud(i+2,j) + rho(i+2,j) + work2(i)
   work2(i+1) = ud(j,i) - rho(j,i)
   temp = work1(i) - ud(i,j)
   taux(i,j) = taux(i,j) + tau(i) +
              (lambda(i,j)+mu(i,j))*temp
20 CONTINUE
30 CONTINUE

-22-
Neither loop can be split and parallelized. The report gives the reasons:

Report for --------> DO 30 J = ...

The following scalar temporaries are not considered dependencies:

\[ I \]
\[ TEMP \]

There are dependencies involving

\[ WORK1(I) \]
\[ WORK2(I+1) \]

Dependencies - DO-loop cannot be split and parallelized.

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

DO 30 J = ...

Report for --------> DO 20 I = ...

The following scalar temporaries are not considered dependencies:

\[ TEMP \]

There are dependencies involving at least one pair from the set:

\[ WORK2([1:1ENDM2]+1) \]
\[ WORK2([1:1ENDM2]) \]

Dependencies - DO-loop cannot be split and parallelized.

There are situations in which it is correct for the user to override the claim by sp that a dependency exists. An example, given in [3], is

\*SSPS\ This is DO LOOP No: 1
\DO 10 i = 1,n
\i0 = select(i)
\a(i0) = value(i0, a(i0))
\10 CONTINUE

where we assume (and so inform the tool in response to prompts) that the arguments to the functions select and value are input only. select may be chosen so that the loop can be split and parallelized although sp, not having sufficient information, will claim that there is a dependency involving a(i0). sp has an override option that is enabled by including a line consisting of the string "override_deps" in the options file _imropts. (See Appendix C.) When the option is enabled, sp will give the user an opportunity to override every dependency finding, even those determined in response to a query of the user by the tool. (The user need not attempt to deceive the tool; he will be asked if his "yes", in response to the prompt, is to be ignored and treated as "no".) With this option enabled and dependency claims overridden by the user, sp produces
*** Loop can be split and parallelized.
C$DOACROSS SHARE(N,A), LOCAL(10)
*$SP$ This is DO LOOP No: 1
   DO 10 i = 1,n
      i0 = select(i)
      a(i0) = value(i0,a(i0))
   10 CONTINUE

The report file notes the claimed dependencies and the user overrides.

4.12. ud

The strategy for transforming regular DO-loops so that arithmetic is localized in the
registers (i.e., increasing the number of arithmetic operations per global memory refer-
ence) begins with unrolling. Typically, the outer DO of a 2-level nest is unrolled by
replicating its range (except for the terminating CONTINUE) d times (d is called the
unrolling depth), with suitable adjustments in the DO parameters and with occurrences of
the outer DO variable, say j, changed to $j + (r-1)*e3$ in the $r^{th}$ replication, where $e3$ is the
incrementation parameter. Generally, there are values of the DO variable not covered by
the new parameters; the corresponding iterations are performed by a "clean-up" loop.

The replicated ranges include copies of the inner loops of the nest. The challenge is
to apply ci (Section 4.2), pd (Section 4.5), sb2 (Section 4.9), si (Section 4.10), and xd
(Section 4.13) to the unrolled code so that occurrences of inner loops may be condensed
using cd (Section 4.1), followed by sb1 (Section 4.8) to produce code in which more
arithmetic is performed per Fortran statement. (See loop 59997 in the example in Section
4.8.) A complete example is worked through in [3].

The unrolling transformation is treated in detail in [2]. To illustrate, consider the
DO nest

   DO 40 j = 1,n
      DO 20 i = 1,j - 1
         b(i) = b(i) + a(j,i)*b(j)
      20 CONTINUE
   40 CONTINUE

Unrolling to depth 4, using ud, produces

C*** DO-loop unrolled to depth 4 ***
   m99999 = (n - (1)+1)/ (4)
m99998 = 1 + 4* (m99999 - 1)
   DO 40 j = 1,m99998,4
      DO 99997 i = 1,j - 1
         b(i) = b(i) + a(j,i)*b(j)
   99997 CONTINUE
   b(j) = a(j,j)*b(j)
When invoked, ud prompts the user for the unrolling depth. DO-loops are checked in the order of appearance of the DO statements for satisfaction of the following criteria:

1. The terminating statement is a CONTINUE.
2. There is no transfer to the terminating CONTINUE.
3. No inner DO has the same terminating CONTINUE.
4. The range contains at least one statement besides the terminating CONTINUE.
5. Every labelled statement in the range is a CONTINUE.
6. Every statement in the range is an assignment, a DO, an IF (arithmetic, block, or logical), an ELSE IF, an END IF, a GO TO, or a CONTINUE.
7. The DO is at the next-to-last level in a nest (this requirement may be overridden, as explained below).

A DO-loop satisfying all of these criteria is unrolled to the specified depth, the clean-up loop is generated, and the search for eligible DO-loops continues in the original program following the transformed portion.

The transformations performed by rg (Section 4.7) guarantee that criteria 1-3 and 5 will be satisfied. Formally, the code resulting from unrolling a regular DO-loop is equivalent to the original. Note that Fortran transformed by rg is not guaranteed to satisfy the fourth regularity condition, and ud does not check this condition. This is left as a user responsibility.
Criterion 7 is not enforced for any DO-loop whose DO statement is preceded by the directive "**$UD$**". On the other hand, an eligible DO-loop will not be unrolled if the DO statement is preceded by the directive "**$NU$**".

4.13. **xd**

A pair of contiguous regular DO-loops can be interchanged if the ranges of their DO variables are disjoint and if the loops satisfy the same data independence conditions as would guarantee that they would produce equivalent results if executed asynchronously on different processors. There is a strong analogy with the notion of a loop being split and parallelized.

Formally, in view of the regularity assumption, the two DO variables need not be the same. However, they are the same in the case of primary interest, namely, rearranging sequences of inner loops in preparation for condensing. Hence, to simplify the tool, we require the two DO variables to be identical.

The directive "**$XDS$"", preceding a sequence of contiguous DO-loops, directs **xd** to test the conditions for interchange of the first two DO-loops in the sequence and to perform the transformation if they are satisfied. In the following example from [3], we wish to interchange DO-loops 77776 and 77777.

```
DO 77777 i = 1, j - 1
   y(i) = y(i) + alpha*x(jx)*a(i,j)
77777 CONTINUE
CONTINUE
*$XDS$
DO 77776 i = j + 1,n
   y(i) = y(i) + alpha*x(jx)*a(j,i)
77776 CONTINUE
CONTINUE
DO 77775 i = 1, j - 1
   y(i) = y(i) + alpha*x(jx+incx)*a(i,j+1)
77775 CONTINUE
CONTINUE
DO 77774 i = j + 1,n
   y(i) = y(i) + alpha*x(jx+incx)*a(j+1,i)
77774 CONTINUE
```

**xd** will display the loops under consideration and ask

Are there common values assumed by the DO variables, i.e., are the ranges NOT disjoint?

The answer, of course, is "no". **xd** then asks

Can the string

\[ Y([J+1:N]) \]

-26-
in the output set of a statement in the first DO-loop for one value of the loop variable refer to the same memory location as the string 

\[ Y([1:J-1]) \]

in the input set of a statement in the second DO-loop for a different value of the loop variable?

The loop variables have been replaced by their ranges in these strings.

Again the answer is 'no'. Two similar questions follow. The analysis, aided by these responses, shows that the dependency conditions are satisfied. The resulting Fortran is

```
DO 77777 i = 1, j - 1
   y(i) = y(i) + alpha*x(jx)*a(i,j)
77777 CONTINUE
DO 77775 i = 1, j - 1
   y(i) = y(i) + alpha*x(jx+incx)*a(i,j+1)
77775 CONTINUE
DO 77776 i = j + 1, n
   y(i) = y(i) + alpha*x(jx)*a(j,i)
77776 CONTINUE
DO 77774 i = j + 1, n
   y(i) = y(i) + alpha*x(jx+incx)*a(j+1,i)
77774 CONTINUE
```

By removing the extra CONTINUE, then applying cd (Section 4.1) followed by sb1 (Section 4.8), the above sequence is transformed into the following form:

```
DO 59999 i = 1, j - 1
   y(i) = (y(i)+alpha*x(jx)*a(i,j)) +
         alpha*x(jx+incx)*a(i,j+1)
59999 CONTINUE
DO 59998 i = j + 1, n
   y(i) = (y(i)+alpha*x(jx)*a(j,i)) +
         alpha*x(jx+incx)*a(j+1,i)
59998 CONTINUE
```

Generally, this will compile into code that makes fewer references to global memory than the original.

**Acknowledgments**

The influence of Chris Thompson is seen throughout the design and implementation of the data dependency tools. We appreciate the exchange of ideas with those who have used args and trans during their "fine tuning", notably Greg Chisholm, Jim Patterson, Bob Schmitt, and Brian Smith. Many helpful comments on the presentation in this guide were offered by Burt Garbow and Gail Pieper.
References


 Appendix A - args Demonstration Program and Report

PROGRAM demo
C .. Local Scalars ..
REAL x
INTEGER index,m
C ..
C .. Local Arrays ..
REAL a(10)
C ..
C .. External Subroutines ..
EXTERNAL subrl
C ..
C .. Data statements ..
DATA m/10/,a/10*0.0/,x/2.5/
C ..
CALL subrl(m,x+1,a)
WRITE (6,RVF=*) (a(index),index=1,m)
STOP
END
SUBROUTINE subrl(m,x,a)
C .. Scalar Arguments ..
REAL x
INTEGER m
C ..
C .. Array Arguments ..
REAL a(*)
C ..
C .. Local Scalars ..
INTEGER i,ip
C ..
C .. External Functions ..
REAL fun,funl
EXTERNAL fun,funl
C ..
DO 10 i = 1,m,2
  ip = i
  a(i) = x*fun(ip,m)
10 CONTINUE
DO 20 i = 2,m,2
  ip = i
  a(i) = x*funl(ip,m)
20 CONTINUE
END
REAL FUNCTION fun(k,j)
C .. Scalar Arguments ..
INTEGER j,k
C ..
A1
C .. External Subroutines ..
EXTERNAL subr2
C ..
C .. Entry Points ..
REAL fun1
C ..
fun = k**2 + j
ENTRY fun1(k,j)

fun1 = k**2 - j

CALL subr2(j)
END
SUBROUTINE subr2(jm)
C .. Scalar Arguments ..
INTEGER jm
C ..
IF (jm.GT.5) jm = 5
RETURN
END

Following is the verbose version of the report produced by args:

Program Unit DEMO has 0 arguments.

Program Unit SUBR1 has 3 arguments.

Argument 1 (M):

INTEGER Variable

+++ Unknown update status in SUBR1
---> Passed by SUBR1 to FUN as argument number 2 (J)
+++ Unknown update status in FUN
---> Passed by FUN to SUBR2 as argument number 1 (JM)
+++ Updated in SUBR2
=== Appears in an expression in SUBR2
---> Passed by SUBR1 to FUN (ENTRY point) as argument number 2
+++ Unknown update status in FUN1
---> Passed by FUN1 to SUBR2 as argument number 1 (JM)
+++ Updated in SUBR2
=== Appears in an expression in SUBR2

!!! Data dependency input/output status: ?/y
Argument 2 (X):

REAL Variable

+++ Not updated in SUBR1
=== Appears in an expression in SUBR1

*** Actual argument is an expression when SUBR1 is called from DEMO at statement number 7.

!!! Data dependency input/output status: y/n

Argument 3 (A):

REAL Array

+++ Updated in SUBR1

!!! Data dependency input/output status: ?/y

Program Unit FUN has 2 arguments.

Argument 1 (K):

INTEGER Variable

+++ Not updated in FUN
=== Appears in an expression in FUN

!!! Data dependency input/output status: y/n

Argument 2 (J):

INTEGER Variable

+++ Unknown update status in FUN
---> Passed by FUN to SUBR2 as argument number 1 (JM)
+++ Updated in SUBR2
=== Appears in an expression in SUBR2

!!! Data dependency input/output status: ?/y

A3
Program Unit SUBR2 has 1 argument.

**Argument 1 (JM):**

INTEGER Variable

+++ Updated in SUBR2

=== Appears in an expression in SUBR2

!!! Data dependency input/output status: ?/y

Entry Point FUN1 has 2 arguments.

**Argument 1:**

INTEGER Variable

+++ Not updated in FUN1

!!! Data dependency input/output status: ?/n

**Argument 2:**

INTEGER Variable

+++ Unknown update status in FUN1

-&gt; Passed by FUN1 to SUBR2 as argument number 1 (JM)

+++ Updated in SUBR2

=== Appears in an expression in SUBR2

!!! Data dependency input/output status: ?/y
Source-embedded directives are comments that begin with the string "*$XX$", where "XX" is a pair of case-insensitive alphabetic characters. The remainder of the directive comment line, following the second "$", is arbitrary. A directive precedes (follows) a statement if the block of comments that immediately precedes (follows) the statement includes the indicated directive, but none of the other directives used by trans commands.

Following is a summary of the directives and their usage:

*$BB$
Precedes the first statement of a block of code whose dependency sets are to be reported by da (Section 4.3).

*$BG$
Precedes the first statement of a block of code that lies entirely within the range of a DO-loop or a branch of a block IF and that sb2 (Section 4.9) is being forced to consider for substitution/elimination.

*$EB$
Follows the last statement of a block of code whose dependency sets are to be reported by da (Section 4.3).

*$EN$
Follows the last statement of a block of code that lies entirely within the range of a DO-loop or a branch of a block IF and that sb2 (Section 4.9) is being forced to consider for substitution/elimination.

*$NU$
Precedes the DO statement of a DO-loop which ud (Section 4.12) is not to unroll, even if the DO-loop otherwise satisfies the conditions for unrolling.

*$PB$
Precedes the DO statement of a DO-loop which is to be peeled from the bottom by pd (Section 4.5).

*$PT$
Precedes the DO statement of a DO-loop which is to be peeled from the top by pd (Section 4.5).
*$SI$*
Precedes the DO statement of a DO-loop to which the secondary loop index transformation is to be applied by si (Section 4.10).

*$SP$*
Precedes the DO statement of a DO-loop for which split and parallelize analysis is to be performed by sp (Section 4.11).

*$UD$*
Precedes the DO statement of a DO-loop which ud (Section 4.12) is to unroll even if the requirement that the DO be at the next-but-last level in a nest is not satisfied. (The other six conditions for unrolling must still be satisfied.)

*$XD$*
Precedes the DO statement of the first DO in a sequence of contiguous DO-loops that xd (Section 4.13) is to consider for interchange with the second DO.
Appendix C - Special Files Used by `trans`

`.callarg`
Input/output status of arguments in external references. Created by tools using responses to queries. The response to queries is greatly facilitated through the use of the report generated by `args`.

`.imropts`
Options for treatment of certain dependencies. Created by user. Currently there are two options, each enabled by including a line in `.imropts` consisting of a particular string. The strings are "accept_locals_in_if_blocks", which affects the operation of `sp`, and "override_deps", which affects the operation of `cd`, `sb1`, `sb2`, `si`, `sp`, and `xd`.

`.transEDERR`
Contains the current version of the subject program when `trans` terminates abnormally due to scanning errors introduced by editing in Mode 1. Created by `trans`.

`.transLOG`
A list of the commands issued in Mode 2 during a `trans` session. Created by `trans`.

C1
Distribution for ANL-88-17

Internal:

J. M. Beumer (3)
W. R. Cowell (50)
F. Y. Fradin
H. G. Kaper
A. B. Krisciunas
G. W. Pieper (50)

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (3)

External:

DOE-OSTI, for distribution per UC-405 (66)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, Pennsylvania State University
T. L. Brown, University of Illinois, Urbana
P. Concus, Lawrence Berkeley Laboratory
S. Gerhart, Micro Electronics and Computer Technology Corp., Austin, TX
H. B. Keller, California Institute of Technology
J. A. Nohel, University of Wisconsin, Madison
M. J. O'Donnell, University of Chicago
D. Austin, ER-DOE
G. Michael, Lawrence Livermore Laboratory