AN INTERACTIVE VERSION OF THE PL/C COMPILER

Richard Conway
Charles Moore, Jr.
Steven Worona

TR 77-301

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

An Interactive Version of the PL/C Compiler

Richard Conway, Charles Moore, Jr. and Steven Worona
Cornell University

The paper discusses a conceptual model of a terminal as an "internal procedure" in an interactive system for a block-structured language. A specific implementation is described, following this model, for the Cornell PL/I compiler.

PL/CT, an interactive version of PL/C -- the Cornell compiler for PL/I (reference 1) -- has been in use at Cornell since 1974, and available for general distribution since 1975. It is an attractive instructional vehicle, wholly compatible with normal batch PL/C, although it is somewhat limited by its dependence upon either the TSO or CMS supervisory systems. Although the system has been in use for two years no description has been published, and it represents two novel concepts that are worth noting:

1. The concept of the "terminal procedure" is a useful way to regard the user-at-his-terminal during interactive execution of a program.

2. The implementation of PL/CT is based on a <u>compiler</u> rather than an interpreter, and yet still provides flexible control of the program during execution.

This paper discusses these two points. To provide specific context, a summary of the PL/CT terminal language is given as an appendix.

The Terminal Procedure

The terminal in an interactive system plays a rather complicated role with respect to the stored program -- in fact, the terminal actually has three distinct roles:

1. As a program entry device -- a means of entering the lines of the source program. This involves a filing and editing system of some sort. Terminal entries are either <u>commands</u> to this system, or <u>data</u> to be processed by this system.

2. As a data entry device, and output display device during

3. As a controller, and source of immediate commands, during execution of the program.

The user and the terminal shift back and forth between these roles with bewildering rapidity. For example, on the basis of output displayed (role 2), the user interrupts the program and sets a "breakpoint" (role 3). When this break occurs, he lists several lines of source program in the neighborhood of the break (role 1), and then displays and changes several values (role 3). After several such interruptions, he may alter the source program (role 1). Throughout this process he will intermittently receive execution output and have to supply additional input data (role 2). It is, of course, essential that the user always understand what "mode" the terminal is in when it seeks his response, since the same entry could be interpreted quite differently in the various different roles.

The first two of these roles are conceptually quite straightforward; it is the third that is difficult to really understand. While in this role, in some actions the user is essentially an extension of the source program; in others he is supervising the action of the source program. The "language" available in role 3 is similar to the source language of the system, but not identical to it. Some source statements are not available in this mode, but other statements are added which are not valid in the source program. Particularly in a modern block-structured language, questions of scope of access from the terminal are difficult. The terminal language for IBM's checkout interpreter for PL/I is a good example of how complex this all can be (reference 4).

In an attempt to organize this confusion we developed a view of the terminal as a "procedure". The idea is to explain role 3 as far as possible in normal terms of the base language, and then make very precise just where the analogy fails. We will discuss this idea in PL/I terms, but believe it is applicable to any comparable ALGOLic language.

Whenever the terminal is active in role 3, it represents an interruption of the execution of the normal stored program. If we assume that an interruption takes place <u>only at a statement boundary</u> -- that is, between two source language statements -- then imagine the following statements to have been inserted at the point of interruption:

```
CALL TP;
TP: PROCEDURE;
      ...
      END TP;
```

The interruption is, in effect, the execution of the CALL. The role 3 action of the terminal represents the <u>body of the procedure TP</u>. A key point is that TP is an <u>internal</u> procedure, inserted at the point of interrupt. This procedure is somewhat

language that is an extended subset of the source language. The procedure is completely ordinary with respect to the scope rules of the source language -- it has access to essentially the same objects that a normal stored internal procedure would have if it were defined at that point in the program.

Procedure TP is said to be "ephemeral" because it exists only for this one execution. It is not stored, and cannot be re-executed except by explicit re-entry. It is said to be "mobile" because its definition moves to the point of interrupt, and hence acquires the environment that exists at that point. Its internal language is an "extended subset" of the source language because there are some source statements that cannot be used in TP, and some additional statements are allowed in TP that are not present in the parent source language -- for example, those statements concerned with establishing breakpoints.

This model somewhat reduces the necessity of introducing novel statements for the TP language. For example, there is no need to introduce a special "resume" statement (sometimes called "run", or "go"). The normal RETURN statement of the source language clearly calls for termination of the terminal procedure and return to the calling environment. No special explanation is required. Similarly, the GO TO and CALL statements should require no special instructions -- if their use is allowed in the terminal procedure they should accomplish exactly what the same statement would do in a normal internal procedure defined at that point. As a final example, PUT DATA(X) in TP is unambiguous, no matter how many instances of X exist in the program, simply because the normal scope rules apply.

All of these statements, and some others -- in particular, the assignment statement -- are easily understood as "immediate" statements. They are executed as soon as entered, and are not saved for future execution. However, role 3 becomes conceptually much more complex if the TP subset includes constructs such as loops, ON-units or procedure declarations. A DO statement simply cannot be immediately executed in the same sense as, say, an assignment statement. If such constructs are included, the situation is more complex for both the processor and the user. PL/CT has limited the TP subset to the obviously immediate statements of PL/I, partly in the belief that these are the most useful in role 3, and partly because the compiled implementation made other statements painfully difficult. On the other hand IBM's Checker allows a much richer subset -- presumably because the interpretive implementation accommodates this more easily, but possibly also because PL/I has never been particularly appreciative of the diminishing returns and compounding complexity of adding features.

The most important extension of the TP subset is the control of breakpoints -- that is, the ability to establish (and remove) points in the source program at which the terminal procedure is called. PL/CT provides two different mechanisms for this:

"PAUSE AT location" establishes calls at absolute source program locations.

"STEP(n)" establishes calls after every n statements executed, relative to the current point of interrupt.

Both these facilities are fairly standard in interactive systems, and in fact, their genealogy can be traced back to console procedures on some historical computers (the IBM 650, for example). However, in a block-structured language an interesting question arises. If absolute program locations (for PAUSE) are specified by means of normal labels and entry-names, the scope rules of the language severely restrict the utility of the breakpoints. To be unable to establish breakpoints within a certain block until the terminal procedure is itself within that environment would be a serious handicap. Various solutions have been proposed -- all involving some circumvention of the scope rules. We decided that the least offensive of these is to allow statement numbers as alternative arguments for PAUSE, and to consider statement numbers to be "external" objects -- accessible from anywhere in the program.

The only other extension included in PL/CT is the ability to selectively list statements of the source program. (Strictly speaking, this should be considered an action from role 1, rather than role 3.) Since one does not always have at hand a complete and current source listing this facility is essential. In PL/CT this is accomplished by an optional form of PUT. (The IBM Checker introduces the "LIST" statement.)

Invocation of the Terminal Procedure

The terminal procedure model is also useful in explaining exactly when and how the terminal is activated in role 3. We referred earlier to the imaginary insertion of the statement "CALL TP;" at the point of interrupt. Now imagine that the following segment is inserted before each statement in the original source program:

```
IF any TP-condition is true
   THEN CALL TP;
```

There are several kinds of TP-conditions. The simplest are bit flags. There is an ERROR_FLAG which is set by raising the ERROR condition. There is an ATTN_FLAG which is set by raising the "attention" condition. In terms of these flags, the segment inserted before statement s of the source program would be:

```
IF ATTN_FLAG | ERROR_FLAG | ...
   THEN DO;
        ATTN_FLAG = '0'B;
        ERROR_FLAG = '0'B;
        CALL TP;
        END;
```

Now the terminal activation can be explained clearly and precisely in terms of normal PL/I constructs. For example, it is clear that in the event of an execution error, the normal error processing takes place first, and the terminal is not activated until the statement causing the error has been completed. It is clear that attention interrupts and program errors are effective only once in invoking the terminal. It indicates that it would be allowable to exit the procedure TP via a GO TO, since all relevant flags have been cleared.

The action of the STEP feature can be explained in terms of the following external procedure:

```
PLCT_STEP: PROCEDURE;
  /* THIS PROCEDURE IMPLEMENTS THE PL/CT "STEP" COMMAND.       */
  /* THERE ARE TWO STATIC COUNTERS:                            */
  /*    STEP_VALUE = "N", FROM THE MOST RECENT EXECUTION       */
  /*       OF "STEP N".  IF STEP_VALUE<=0, THEN "NOSTEP"       */
  /*       IS IN EFFECT.                                       */
  /*    REMAINDER = THE NUMBER OF STATEMENTS REMAINING         */
  /*       TO BE EXECUTED BEFORE THE CURRENT "STEP" VALUE      */
  /*       IS REACHED.  IF REMAINDER<=0, THEN "NOSTEP" IS      */
  /*       IN EFFECT.                                          */
  /* THERE ARE THREE ENTRY POINTS:                             */
  /*    STEP_STMT:  CALLED BEFORE EACH STATEMENT IN THE        */
  /*       PROGRAM IS EXECUTED.  RETURNS '1'B IF THE           */
  /*       CURRENT "STEP" VALUE IS REACHED BY THAT             */
  /*       STATEMENT.  RETURNS '0'B OTHERWISE.                 */
  /*    STEP:  CALLED WITH PARAMETER N WHEN "STEP N" IS        */
  /*       EXECUTED.                                           */
  /*    NOSTEP:  CALLED WHEN "NOSTEP" IS EXECUTED.             */

  DCL (STEP_VALUE, REMAINDER) STATIC BIN FIXED INIT(0);

STEP_STMT: ENTRY RETURNS(BIT(1));
  REMAINDER = REMAINDER - 1;  /*COUNT UPCOMING STATEMENT*/
  IF REMAINDER ¬= 0
     THEN RETURN('0'B);       /*INDICATE LIMIT NOT REACHED*/
  REMAINDER = STEP_VALUE;     /*RESET REMAINDER TO LIMIT*/
  RETURN('1'B);               /*INDICATE LIMIT REACHED*/

STEP: ENTRY(N);
  DCL N BIN FIXED;
  STEP_VALUE = N;             /*RESET STEP LIMIT AS SPECIFIED*/
  REMAINDER = N;
  RETURN;
```

```
NOSTEP: ENTRY;
  STEP_VALUE = 0;              /*RESET TO INDICATE NOSTEP*/
  REMAINDER = 0;
  RETURN;

  END PLCT_STEP;
```

Adding the STEP condition to the other TP-conditions, the assumed insertion before statement s is:

```
            IF STEP_STMT | ATTN_FLAG | ERROR_FLAG | ...
               THEN DO;
                    ATTN_FLAG = '0'B;
                    ERROR_FLAG = '0'B;
                    CALL TP;
                    END;
```

The breakpoint feature can also be explained in terms of an external procedure. This feature involves three TP-subset statements (see Appendix): "PAUSE AT s" and "NOPAUSE AT s" to set and clear breakpoints, and "IGNORE n" to ignore the first n breakpoints encountered. These statements are explained by PLCT_PAUSE, defined below:

```
PLCT_PAUSE: PROCEDURE;
  /* THIS PROCEDURE IMPLEMENTS THE PL/CT "PAUSE" FEATURE.   */
  /* THERE IS A STATIC BIT(1) ARRAY:                        */
  /*     PAUSEAT, PROVIDING A "PAUSE FLAG" FOR EACH         */
  /*       STATEMENT IN THE SOURCE PROGRAM.                 */
  /*       SIZE OF ARRAY DEPENDS ON PROGRAM LENGTH.         */
  /* THERE ARE TWO STATIC COUNTERS:                         */
  /*     IGNORE_VALUE = THE NUMBER OF PAUSES TO BE IGNORED. */
  /*       IF <= 0, THEN PAUSES ARE NOT BEING IGNORED.      */
  /*     IGNORE_COUNT = THE REMAINING NUMBER OF PAUSES TO   */
  /*       BE IGNORED IN THE CURRENT CYCLE.                 */
  /*       IF < 0, THEN PAUSES ARE NOT BEING IGNORED.       */
  /* THERE ARE FOUR ENTRY POINTS:                           */
  /*     PAUSE_STMT:  CALLED WITH PARAMETER S BEFORE        */
  /*       STATEMENT S IS EXECUTED.  RETURNS '1'B IF THERE  */
  /*       IS AN EFFECTIVE PAUSE FOR THAT STATEMENT.        */
  /*       RETURNS '0'B OTHERWISE.                          */
  /*     PAUSE:  CALLED WITH PARAMETER S WHEN "PAUSE AT S"  */
  /*       IS EXECUTED.                                     */
  /*     NOPAUSE:  CALLED WITH PARAMETER S WHEN             */
  /*       "NOPAUSE AT S" IS EXECUTED.                      */
  /*     IGNORE:  CALLED WITH PARAMETER N WHEN "IGNORE N"   */
  /*       IS EXECUTED.                                     */

  DCL PAUSEAT(----) STATIC BIT(1) INIT((----)('0'B));
  DCL (IGNORE_COUNT, IGNORE_VALUE) STATIC BIN FIXED INIT(0);
  DCL S DEC FIXED;             /* STATEMENT NBR PARAMETER   */

PAUSE_STMT: ENTRY(S) RETURNS(BIT(1));
```

```
  IF PAUSEAT(S) = '0'B
     THEN RETURN('0'B);        /* INDICATE NO PAUSE FOR S   */
  IGNORE_COUNT = IGNORE_COUNT - 1;  /* COUNT PAUSE FOR S   */
  IF IGNORE_COUNT >= 0
     THEN RETURN('0'B);        /* PAUSE IGNORED             */
  IGNORE_COUNT = IGNORE_VALUE;  /* RESET FOR NEXT CYCLE.    */
  RETURN('1'B);                /* INDICATE PAUSE FOR S      */

PAUSE: ENTRY(S);
  PAUSEAT(S) = '1'B;           /* SET PAUSE AT S            */
  RETURN;

NOPAUSE: ENTRY(S);
  PAUSEAT(S) = '0'B;           /* CLEAR PAUSE AT S          */
  RETURN;

IGNORE: ENTRY(N);
  DCL N BIN FIXED;
  IGNORE_VALUE = N;            /* SET NEW N FROM IGNORE N   */
  IGNORE_COUNT = N;
  RETURN;

  END PLCT_PAUSE;
```

Adding the PAUSE condition to the other TP-conditions, the complete form of the assumed insertion before statement s is:

```
    IF STEP_STMT | PAUSE_STMT(s) | ATTN_FLAG | ERROR_FLAG
       THEN DO;
            ATTN_FLAG = '0'B;
            ERROR_FLAG = '0'B;
            CALL TP;
            END;
```

By specifying that this insertion is positioned _after_ the label prefix of a statement, the interaction between breakpoints and GO TOs is completely specified. Furthermore, it is clear that breakpoints are not cleared by invoking the terminal. They are "permanent", regardless of terminal action, until the NOPAUSE statement is used.

These two procedures _completely explain the role of the terminal_ relative to the execution of the stored program. They explain exactly when the terminal is activated in role 3, and why. (We should note that these procedures are _only_ for purposes of explanation, and do not describe the actual method of implementation.)

While the TP-conditions shown above are the only ones that currently exist in PL/CT, the model is clearly provocative. It invites the introduction of more general conditions. For example, suppose the TP-subset included a statement of the form:

```
          RUN UNTIL condition;
```

where "condition" is any arbitrary, user-specified condition. This condition would then become part of the compound condition of the segment inserted before each source statement. The result would be a very general user-defined ON-condition, with the terminal serving as the ON-unit. This would effectively provide a type of "event-driven" programming (reference 2).

Implementation of PL/CT

Part of the incentive to consider PL/CT in the first place was our curiosity to see how much in the way of interactive execution could be provided by a compiler -- that is, without resorting to interpretive execution. Beyond the obvious implementation economy (in that we already had the PL/C compiler), this would ensure complete compatibility with normal batch PL/C, and greatly simplify the task of maintaining the two systems.

Under either the TSO or CMS supervisor the accomplishment of role 1 and role 2 terminal use is quite straightforward. Either system provides an editor than can be used for role 1, and both provide device-assignment facilities that allow the terminal to serve as the SYSIN and SYSPRINT medium. There is a conceptually messy question of how to correctly interleave two streams on the same device, given PL/I's definition of stream I/O (reference 5), but this just involves choosing the least unattractive compromise.

The real question is how to implement role 3, and this turned out to be surprisingly easy, beause of two fortuitous properties of PL/C -- (1) the identity of source language statements is preserved in the PL/C object program, and (2) the symbol table is accessible at run-time. Point (2) made it straightforward to translate references to identifiers in TP-subset commands into the form needed by the PL/C run-time I/O routines.

With regard to point (1), PL/C object code includes, at each source-statement boundary, a literal giving the source number of that statement and an instruction to store that number in a fixed location. (This allows execution-time messages to refer to source statement numbers.) The PL/C code generator had only to be modified to include in each statement preamble a call to a routine that tests the various TP-conditions. In PL/I terms, this is equivalent to preceding each source statement s by:

```
CALL TEST(s);
```

where TEST is an external procedure defined as shown below. TEST in turn depends upon the procedures PLCT_STEP and PLCT_PAUSE described above.

```
TEST: PROCEDURE(S);
   DCL S FIXED DEC;
   IF STEP_STMT | PAUSE_STMT(S) | ATTN_FLAG | ERROR_FLAG
       THEN DO;
           ATTN_FLAG = '0'B;
           ERROR_FLAG = '0'B;
           CALL TP;
           END;
   RETURN;
   END TEST;
```

Since this adds only a single instruction to the code for each source statement, and several instructions to the execution, the penalty in both space and time is modest. It effectively provides the responsiveness and control usually obtained by interpretive execution, without the substantial degradation of performance.

In aggregate, PL/CT consists of a system interface module (different modules for TSO and CMS), the PL/C compiler (with the code generator modified as described above), and an additional module to implement the role 3 terminal activity -- in effect, to support the TP procedure. This latter module is, by definition, interpretive. Each statement entered from the terminal is parsed and immediately executed. Those terminal statements that correspond to normal PL/C statements are executed with exactly the same instruction sequence as would be generated into the object program for the comparable statement.

The overall result is a system that is very simple to implement and maintain, given that the underlying compiler already existed. The system is relatively efficient to use (compared to an interpreter), and absolutely compatible between the batch and interactive versions. Most important, it is conceptually easy to understand just how the terminal relates to the stored program.

References

1. Conway, R. W. and T. R. Wilcox, "Design and Implementation of a Diagnostic Compiler for PL/I", Communications of the ACM, March 1973

2. Morgan, H. L., "An Interrupt Based Organization for Management Information Systems", Communications of the ACM, December 1970

3. PL/CT User's Guide, Release 2/76, Department of Computer Science, Cornell University, 1976

4. PL/I Checkout Compiler: TSO User's Guide, SC33-0033, IBM

5. Worona, S.; "PL/CT, Another Approach to Two Problems in Interactive PL/I", SIGPLAN Notices, Vol. 11, No. 1, January 1976

Appendix

The following describes the TP-subset language provided by the current version of PL/CT. This is copied directly from the PL/CT User's Guide (reference 3), where the TP-subset is called "debug mode".

PL/CT Debug Commands

When the system is in debug mode any of the following commands may be given. Each command is executed immediately; it is not saved, and does not become part of the source program. The format for commands is free-field -- essentially the same as for statements in PL/C, except:

1. Comments are not allowed.
2. Commands may begin in position 1 of the line.
3. Commands cannot be continued onto a second line.

PUT SKIP LIST(variable, ...);
PUT SKIP DATA(variable, ...);

A restricted form of the PL/C PUT statement. The variable specified can be a scalar, an array, a structure or a subscripted variable with a constant subscript. Variables must be accessible at the point of interrupt under normal PL/C scope rules. Neither expressions nor literals can be given.

SKIP is assumed and need not be given.

If neither LIST nor DATA is specified the default output format will be used. If either LIST or DATA is specified, either in a PUT or as a separate command (see below), this sets the default output format. Initially the default is LIST.

This command may be abbreviated as just "PUT variable;" or just as the variable name alone. That is, assuming that LIST is the default output format, "X;" and "PUT X;" are equivalent to PUT SKIP LIST(X);".

LIST;

Set the default output format (for debug commands only) to be LIST.

DATA;

Set the default output format (for debug commands only) to be DATA.

PUT m, n;

m is a statement number from the source listing, and n is an integer. Display n source lines beginning with the line on which statement m started. If n is omitted from the command, 1 is assumed.

n can also be given as a label or entry-name, accessible from the point of interrupt under the normal PL/C scope rules.

Variable = constant;
A restricted form of the PL/C assignment statement. The target variable must be a scalar or a subscripted variable with constant subscript(s). It cannot be a label variable, an array or a structure. Structure elements must be fully-qualified. Multiple left sides and BY NAME assignment are not allowed. The right side can only be an arithmetic or string constant -- neither a variable nor an expression is allowed.

STEP n;
n is an integer. Reset the STEP interval to n, so that PL/CT will re-enter debug mode after execution of n statements of the source program. If n is omitted, 1 is assumed. This STEP interval remains in effect until changed -- it does not just apply to the first RETURN. Note that statements are counted in a manner comparable to PL/C numbering -- that is, END, PROCEDURE, DO are also counted as statements.

NOSTEP;
Reset the STEP interval to the default value: STEP 2^{16}.

PAUSE AT s;
Establish a PAUSE before statement(s) s. s can be given in several forms:

-a statement number, as given on the PL/C source listing

-a label or entry-name, which is accessible at the point of interrupt under normal PL/C scope rules

-an accessible label or entry-name modified by an integer. For example:
```
PAUSE AT ERRORPROC+6;
PAUSE AT TERMLOOP-3;
```

-an inclusive range of statements: "s^1 TO s^2" where s^1 and s^2 are any of the forms listed above. s^2 can also be the word END, implying the last statement of the program. For example:
```
PAUSE AT 14 TO TERM_LOOP;
PAUSE AT EVALPROC+3 TO EVALPROC+14;
PAUSE AT PRINT+6 TO END;
```

-ALL, which means "1 TO END".

The PAUSE command may be abbreviated by giving s (or s^1 TO s^2) alone. That is, if a command consists of any of the valid forms for s, "PAUSE AT s;" is assumed. For example, "36;" is equivalent to "PAUSE AT 36;".

PAUSEs are maintained in a list of fixed length within PL/CT. When this list is full, further PAUSE commands will be rejected. You will have to remove some PAUSEs before new ones can be added.

NOPAUSE AT s;
Remove the PAUSE (if any) before statement(s) s. s is given in the same forms as for the PAUSE command. Note that NOPAUSE can have a range <u>but not a list</u> of arguments. That is, "NOPAUSE AT s^1, s^2;" is <u>not valid</u>. (s2 will be considered a separate command -- an abbreviation of "PAUSE AT s^2;".) Also note that since removing the middle of a PAUSE range actually creates two ranges, it is possible for NOPAUSE to cause overflow of the PAUSE list.

IGNORE n;
n is an integer. During program execution ignore the first n PAUSEs encountered; re-enter debug mode on the n+1st PAUSE. If n is omitted, 2^{16} is assumed. This IGNORE count remains in effect until changed -- it does not just apply to the first RETURN. Initially, the IGNORE count is 0 -- that is, PL/CT will stop on every PAUSE unless you set the PAUSE count to some non-zero value.

NOCHECK;
Suppress the printing of CHECK output, exactly as in PL/C.

CHECK;
Resume the printing of CHECK output, as in PL/C except that no parameters are allowed on the command.

NOFLOW;
Suppress the printing of FLOW output, exactly as in PL/C.

FLOW;
Resume the printing of FLOW output, as in PL/C except that no parameters are allowed on the command.

PUT OFF;
Suppress printing of SYSPRINT output, exactly as in PL/C.

PUT ON;
Resume printing of SYSPRINT output, exactly as in PL/C.

PUT ALL;
Display the current values of all automatic, scalar variables in the blocks active at the point of interrupt, as well as the current values of all static and external scalar variables, exactly as in PL/C.

PUT ARRAY;
Same as PUT ALL but also includes arrays, exactly as in PL/C.

PUT FLOW;
    Display recent FLOW history, exactly as in PL/C.

PUT SNAP;
    Display recent calling history, exactly as in PL/C.

RETURN;
    Leave debug mode and resume execution of the source program. RETURN can be indicated by a null line. That is, after the "DBC:" prompt a carriage return with an empty line is equivalent to a RETURN command.

GO TO label;
    Leave debug mode and resume execution of the source program starting with the statement whose label is given. This label must be accessible from the point of interrupt under the normal PL/C scope rules.

STOP;
    Terminate execution of the PL/CT program, exactly as in PL/C.