# Structuring Control in Fortran

Arthur Sedgwick
University of Toronto*
Department of Computer Science
Toronto Canada M5S 1A7

This note suggests yet another view as to what compound statements should be introduced into languages such as Fortran. In this proposal, selection and refinement are the key constructs and a radically simple approach to repetition due to Hehner [1976] is presented.

A major goal in defining the control constructs is to keep the semantics as *simple* as possible. We would like to encourage the programmer to invent informal or formal assertions describing the state of the computation at various points in the program. The assertions reduce the dynamic relationship between variables and values to *invariant* expressions. Unfortunately, many existing control constructs have sufficiently complicated semantics to discourage the programmer from thinking this way.

The semantics of the statement S determine for any predicate R the weakest precondition such that the execution of S will establish R. This weakest precondition might be denoted S : R following the suggestion of Hehner [1976]. Then the semantics of executing $S_1$ followed by $S_2$ might be denoted

$$(S_1 \ \ S_2) : R = S_1 : (S_2 : R) = S_1 : S_2 : R.$$

That this composition seems so trivial is a credit to the notation. Of course, the weakest precondition

$$X = E \ : \ R,$$

for the assignment X = E to establish R, would be R with all free occurrences of X in R replaced by the expression E. (Note that the notation $R_E^X$ is not very convenient when composition is considered. The unabbreviated notation X = E : R is more convenient.) Dijkstra [1976] gives many examples of the "calculus" of preconditions.

With this point of view for semantics we now introduce the control constructs. The syntax is informal with lower-case letters for non-terminals, and is just a suggestion. The only new character is the double-quote symbol which is available in all modern character sets. (If necessary, it could be replaced by another character as in Hull [1976].) The predicates $p_i$ are logical expressions. The semicolon is used in some of the examples as a statement separator merely to compress the examples.

---

*This research was supported by the National Research Council of Canada.

Control Constructs for Fortran

```
⟨selection⟩ ::=
      (CASE p1:                  |   (IF p1:
          statements1            |       statements1
       CASE p2:                  |    IF p2:
          statements2            |       statements2
          ⋮                      |       ⋮
       CASE pn:                  |    IF pn:
          statementsn            |       statementsn
       CASE)                     |    IF)

⟨refinement⟩ ::=
      (RE "what"                 |   (DO "what"
          statements             |       statements
       RE)                       |    DO)

⟨call for refinement⟩ ::=   "what"

⟨counted loop⟩ ::=          (DO i = e to e:  statements  DO)
```

The key concepts are selection and refinement. There are two forms of selection. The semantics of the CASE statement are that

$$p_i \Rightarrow \text{statements}_i : R, \quad 1 \le i \le n,$$
$$\underline{\text{and}} \quad \underline{\text{not}}(p_1 \ \underline{\text{or}} \ \cdots \ \underline{\text{or}} \ p_n) \Rightarrow R.$$

i.e. this is the weakest precondition for the CASE statement to establish R. Note that the order of the alternatives is irrelevant (except, perhaps, that the most likely alternatives might appear first for reasons of efficiency only). The semantics of the IF statement are the same as those of the CASE statement if there is just one alternative, and otherwise are equivalent to

```
(CASE p1:
      statements1
 CASE.NOT.p1:
     (IF p2:statements2 ··· IF pn:statementsn  IF)
 CASE)
```

i.e. the IF statement is equivalent to a nest of CASE statements with the keyword IF without parenthesis meaning "elseif". Thus IF is associated with ordered alternatives and CASE is not.

Of course, we should invent some syntax for "else". One possibility, which has been adopted for this paper, is to omit $p_n$ when it means "none of the above" (not ($p_1$ or ··· or $p_{n-1}$)). We could also adopt some special syntax for situations where the CASE statement reduces to a "computed go to".

Refinement is the key concept of this paper. The character string "what" describes, perhaps informally, *what* needs to be done and calls for the statements in the corresponding refinement to be executed. The statements determine *how* the goal of the refinement is to be achieved. For example,

"Y = SIGN(X)"

might call for the following statements (i.e. be refined as)

```
(RE "Y = SIGN(X)"
  (CASE X.GT.0:  Y =  1
   CASE X.LT.0:  Y = -1
   CASE X.EQ.0:  Y =  0
   CASE)
 RE)
```

The keyword RE could be pronounced "refine". It might also be interpreted as "regarding". Hull [1976] uses the word "where".

Note that the semantics of refinement come "for free":

"what" : R = **statements** : R

where the statements are those in the corresponding refinement. In other words, the semantics of refinement are those of inserting the corresponding statements at the point of call.

Through refinement a procedure can be presented in a top-down fashion without the overhead of subprograms and arguments. Normally, the refinements would appear just before the END of the program unit; however, they may appear at the point of call via refinement "in situ". The keyword DO calls for refinement and brackets the corresponding statements. It corresponds to a labelled DO-END group in PL/I and is intended for situations where it may be inappropriate to place the refinement elsewhere. As we shall see, the keyword DO will usually be associated with repetition, however, formally it signals refinement in place. The familiar counted DO loop (whose semantics are too distracting to contemplate here) turns out to be an important special case.

Since refinements have no arguments, their implementation is straightforward. The translator could insert the statements at the point of call or arrange for branching to and from the definition. Something equivalent to the GOSUB of BASIC would be ideal for situations where a refinement definition is called from several points. The advantage of leaving it to the translator to insert the refinements, rather than insert them ourselves as we are forced to do at present, is that the intermediate structure of the program is not lost.

Indefinite Repetition Expressed Recursively

The principal point of this paper is that selection and refinement are sufficient. Looping constructs such as DO WHILE( ) and its variants and "exit" are neither necessary nor desirable. The alternative is to express iteration *recursively while still implementing it as at present.* For example

```
(DO"SET B = GCD(A,B)"
    R = MOD(A,B)
   (IF  R.NE.0:
        A=B
        B=R
       "SET B = GCD(A,B)"   IF)   DO)
```

Here the recursive call "SET B = GCD(A,B)" forces repetition until R=0. Note that a translator would have no difficulty in recognizing that the recursive call was the "last action" of the refinement and therefore should be implemented merely by a branch back to the start. (Unfortunately, most systems do not look for this optimization which depends on there being no arguments to the refinement.)

As another example, consider Dijkstra's do-od guarded command construct which repeats until all the guards (predicates) are false.

```
(RE  "LOOP-DIJKSTRA DO-OD"
  (CASE p1: statements1; "LOOP..."
    .
    .
    .
   CASE pn: statementsn; "LOOP..."
   CASE)
 RE)
```

In this implementation the recursive calls have been abbreviated to "LOOP...". This form of abbreviation (Hull and Bedet [1976]) is especially convenient when the character strings "what" are long sentences.

As in the previous example (and in all other repetitive situations) the recursive calls are all "last action" and should be implemented merely as branches without any stacking activity.

There are many advantages to expressing (without implementation overhead) repetition recursively: advantages in terms of power, proof techniques and programmer psychology. The technique is more powerful than the popular WHILE loop and its variants. An exit from any alternative is achieved merely by failing to recurse. This eliminates many flags and extra tests that would be necessary using DO WHILE or do-od.

The proof techniques are simpler in that no special loop cutting techniques (Hantler and King [1976] p. 342) are required. If the programmer is conditioned to verifying the precondition of a refinement at the point of call this suffices for the recursive calls as well. Termination is assured by the standard approach of requiring that some positive integer-valued function be reduced at each call.

The psychological advantages accrue from encouraging the programmer to think in terms of arriving at the same *state of the computation* instead of in terms of the flow of control and branching backward. It is possible that a keyword such as REPEAT could be used to signal last-action recursive repetition; however, this would detract from the psychological advantages.

The following is an example which students found to be awkward using a WHILE loop. The problem is to find KEY in a linked list of records with DATA and LINK fields. The list begins with a dummy HEADER record and ends with a 0 link.

```
PREV = HEADER
(DO"DELETE KEY FROM LIST OR PRINT MESSAGE"
   P = LINK(PREV)
  (IF  P.EQ.0:          "PRINT MESSAGE"
   IF  KEY.EQ.DATA(P):  LINK(PREV) = LINK(P);  CALL FREE(P)
   IF:                  PREV = P;  "DELETE..."   IF)
DO)
```

Note the implicit exit from the loop when P=0 or DATA(P)=KEY. Since the ordering of the alternatives is significant this solution avoids the problem of DATA(P) being undefined when P=0.

Once a programmer has begun expressing his loops recursively he may wish to use non-repetitive recursion. For example,

```
(RE "FIND MATCHING RIGHT PARENTHESIS"
   "GET NEXT SYMBOL"
   (IF SYMBOL.NE.')':
      (IF SYMBOL.EQ.'(':  "FIND MATCHING RIGHT PARENTHESIS" IF)
     "FIND..."
    IF)
RE)
```

The second recursive call is last-action and may be implemented merely as a branch. Aside from this, the refinement is called once internally (non last-action recursion) and at least once externally. If a GOSUB type of implementation is used for multiple calls to a refinement then there need be no distinction between internal and external calls. Of course a single internal call (without arguments) may always be implemented using an integer to count the number of unsatisfied internal calls (or in this case, unmatched left parentheses) and a programmer could write the refinement this way in the first place.

It may seem incongruous to be using recursion in Fortran and non last-action recursion could be banned; however, the extra effort to implement it would be small and worthwhile even for non-recursive calls.

One other point in connection with recursion should be made. Suppose refinement A calls B, and B directly or indirectly calls A. This indirect recursion, whether last-action or not, is dangerous. A may be obviously correct, assuming that B is, and similarly for B. However, the combination may fail. For an example see Sale [1975]. The solution is to require either A or B to appear inside the other using refinement "in situ". In this way issues such as termination of a loop will always be associated with some refinement rather than be a global property of the program.

In conclusion , the main point is that selection and refinement are sufficient for structuring control, and last-action recursion is a more powerful and natural way to express repetition than alternatives such as WHILE loops and exit constructs.

## References

Dijkstra, E.W. [1976] *A Discipline of Programming*, Prentice-Hall, New Jersey.

Hantler, S.L. and King, J.C. [1976] An introduction to proving the correctness of programs, ACM Computing Surveys 8(3), September 1976.

Hehner, E.C.R. [1976] do considered od: A Contribution to the Programming Calculus, CSRG-75, University of Toronto, November 1976.

Hull, T.E. and Bedet, R. [1976] Fortran-S User's Guide, Department of Computer Science, University of Toronto.

Sale, A.H.J. [1975] Basic principles of well-structured code, Department of Information Science No. R75-1, University of Tasmania, April 1975.