A language design for concurrent processes

by L. G. TESLER
Computer Consultant
Palo Alto, California
and
H. J. ENEA
Stanford University
Stanford, California

INTRODUCTION

In conventional programming languages, the sequence of execution is specified by rules such as:

1. The statement "GO TO L" is followed by the statement labelled "L" (Branching rule).
2. The last statement in the range of an iteration is followed, under certain conditions, by the first statement in the range (Looping rule).
3. The last statement of a subroutine is followed by the statement immediately after its CALL (Out-of-line code rule).

... (Other rules)

(n) In other cases, each statement is followed by the statement immediately after it (Order rule).

This paper will define a class of general-purpose languages which does not need these rules. The power of these languages is equivalent to that of Algol, PL-1, or LISP. Other languages which do not need these rules appear in the literature.1,2,3

Concurrent processing

The advent of multi-processing systems makes it possible for a computer to execute more than one instruction at a time from the same program without resorting to complicated look-ahead logic. There are many ways in which this capability can be utilized; one way is to find several statements that could be executed simultaneously without conflict, and to delegate their execution to different available processors. This technique is sometimes called "concurrent processing". To employ it there must be a means for determining, during compilation, which statements can be processed concurrently.

Many proposals for programming languages have suggested that more rules like 1, 2, 3, .. , n should be added for explicit indication of concurrence.4,5,6,7 Examples of such rules are:

(n + 1) The statement "FORK M, N, .. " is followed simultaneously by the several statements labeled "M", "N", .. ; the statement "JOIN M, N, .. " terminates the fork.

(n + 2) The range of statements following "DO SIMULTANEOUSLY" can be executed for all values of the iterated variable at once.

The programmer using these facilities must take care that the statements performed simultaneously do not conflict, e.g., do not assign different values to the same variables.8

Other proposals have suggested an analysis of potential conflicts, during compilation, to isolate all concurrence that does not depend on run-time data values ("program concurrence").8,9,10,11,12 This approach is adopted here because its automatic elimination of all possible conflicts guarantees determinacy.

Once it is possible to isolate all program concurrence using implicit information, it is tempting to examine the possibility of determining all program sequence (i.e., non-concurrence) using implicit information. If that were possible, then not only rules n + 1 and n + 2 but also rules 1, 2, 3, .. , n could be eliminated. In the following section, the class of "single assignment" languages will be defined such that all program sequence and program concurrence are determinable during compilation without explicit indication.

Single assignment languages

A program written in a single assignment language has the following essential characteristic:

No variable is assigned values by more than one statement.

The only effect of executing any statement is to as-
sign values to certain variables named in that statement (no side effects).

Every statement is an assignment statement. The variables names in each statement belong to two exclusive groups:

1) Outputs: Those which are assigned values by the statement.

2) Inputs: Those whose values are used by the statement.

The output variables are said to be dependent on the input variables. The abbreviation "AdB" stands for "A is dependent on B". Dependence is:

(a) Transitive: AdB ∧ BdC → AdC.

(b) Antisymmetric: AdB → ¬BdA.

(c) Irreflexive: ¬AdA.

Circular dependence is not allowed; for example, A can't be dependent on B if BdC and CdA. If neither AdB nor BdA, then A and B are independent.

All required program sequence can be determined during compilation by a straightforward tracing of dependence. As the symbol table is built, two-way pointers are constructed between the input and the output variables of each statement. The final symbol table, which is both a data-flow and a program-flow diagram, elucidates the required sequence in the program. Statements that are not found to require sequential execution can be performed concurrently. The rules 1, . . . , n + 2 are replaced by the single rule:

The statement that outputs variable A must be executed before every statement that either inputs A or inputs some B such that BdA.

The order of appearance of statements is immaterial to this analysis; consequently, an incremental compiler can be employed which accepts statements typed in any order. This property is especially useful when adding a statement to a previously written program because neither unforeseen side effects nor mislocation of the statement can occur.

Optimization

One way of optimizing a program is to reduce the amount of redundant computation by combining "common expressions". In a single assignment program there are no side effects; therefore, common expressions throughout the program can be combined during compilation. Another way of optimizing a program is to allocate storage efficiently. For example, in the program:

```
var: x + y;
a: var - w;
b: 2 × var;
c: x - y;
d: -var;
```

storage for the variable "var" need not be reserved until just before any of the statements "a", "b", or "d" demand the value of "var", and may be released as soon as all of "a", "b", and "d" no longer require the value of "var". These requirements can be detected easily during compilation. As a result, storage is never allocated for a variable except when necessary to guarantee the availability of its value for further computation.

Compel

Compel (Compute Parallel), a single assignment language, will be partially defined so that examples of single assignment programs can be given. This language has not been implemented.

Compel programs process three types of quantities: numbers, lists, and functions. A number is a floating-point approximation to a complex number, e.g.,

```
2 + i4.6
-17.3
5
```

A list is an ordered set of zero or more quantities, e.g.,

```
[1,5,9] =

[ factorial, [6,2,7], [] ]

[ 1 by 4 to 9 ]

[ 1 by 4 for 3 ]

[ 1, while preceding < 9 use preceding + 4 ]

[ while index < 3 use 4 × index - 3 ]
```

A function has one argument which may be of any type and is frequently a list ("parameter list"), e.g.,

```
ϕ x (x ↑ 2)
ϕ [a,b] (a × x + b × y)
```

Blocks may be created for local naming—they have nothing to do with storage allocation or with program sequence. The statements within a block may appear in any order, and the blocks within a program may nest and appear in any order. Each block begins with the line:

```
input v1, v2, . . . , vn;
```

where v1, . . . , vn (n ≥ 0) are the names of those variables defined outside the block and used inside the block. Similarly, each block ends with the line:

```
output w1, w2, . . . , wm;
```

where w1, . . . , wm (m ≥ 1) are the names of those variables defined inside the block and used outside. Every statement is of the form:

```
VARIABLES:EXPRESSION:
```

For example, the statement:

```
ar: [1,2,4];
```

assigns to "a" the single quantity "[1,2,4]".

The statement:
is analogous to the Algol statement:

\[
\text{for } a = 1,2,4 \text{ do . . .}
\]

where the range of the do includes all statements which are dependent on "a". On a parallel computer, all values of "a" can be produced concurrently; thus, a variable (in this case "a") can have several values at the same time. The function each splits a list of values so that its elements can be operated on individually.

After a list is split and its elements have been operated on individually the results of the operations are collected back into a list. For example:

\[
a: \text{each } [1,2,4];
b:a \uparrow 2;
c: \text{list of } b;
\]

Statement "a" splits the list "[1,2,4]" into three quantities; statement "b" squares each quantity; statement "c" collects the results into a single list, i.e., "[1,4,16]". Splitting and collecting are analogous to forking and joining, except that splitting operates on the data flow, but forking operates on the program flow.

Every statement that does not assign values to the output variables of its block can be eliminated from that block by systematic substitution. For example, the three statements in the last example can be reduced to:

\[
c: \text{list of } (\text{each } [1,2,4]) \uparrow 2;
\]

This property is the converse of common expression combination.

The two statements:

\[
a: \text{each } [1,2,4];
b:a \uparrow 2 + a \uparrow 3;
\]

where "a" is used in no statement but "b", can be reduced to one statement in another way by use of the following construction:

\[
b:a \uparrow 2 + a \uparrow 3 \text{ with a each } [1,2,4];
\]

Its advantage over the two statements it replaces is that, by omission of the colon after "a", "a" becomes a name local to the statement. A local variable is distinct from all other variables of the same name throughout the program.

In the examples that follow, subscripting is specified by a downward arrow, e.g., \(a_{ij}\) is written:

\[
a \downarrow [i,j]
\]

**Examples**

Two simple Compel examples are given below. For each are given:

(a) a program/data flow diagram displaying concurrency and storage release;
(b) an Algol program using fork, join, and do simultaneously;
(c) a Compel program.

Figure 1 illustrates concurrent execution of statement and optimization of storage.
Compel:

input;
out:(a-e)/d;
a:6;
e:a×b-c;
d:a-b;
b:7;
c:8;
output out;

Figure 2 demonstrates simultaneous assignment of all the elements of a list.

However, the incrementation of “i” is never a major step in a program, but merely one small step in a larger process. In Compel, notation is provided to incorporate such a step into an algorithm.

Example 1: to compute the sum of the elements of list “m” one can write:
sum:=+m;
where “+” is a function which returns the sum of the elements of its argument.

Example 2: in an Algol program, a variable may be assigned values in several statements, some of which increment the variable:
r:=r0;
for i:=1 step 1 until n do
begin
  a [i] :=b×r;
  r:=r+2;
end;
rl:=r;

In Compel each variable is assigned values by only one statement:
r:each [r0 by 2 for n];
a:list of (b×r);
rl:last (list of r);

Conditional statements are not available in Compel; therefore, conditional expressions must be employed to achieve the effect of the Algol statements:

y:=y0;
b:=b0;
t:=t0;
if a < 0 then
  begin
    b:=c+ 1;
y:=a;
  end
else
  begin
    t:=r-a;
y:=v;
  end;

A corresponding Compel program:
b: [b0, if a < 0 then c+1 else preceding] ↓ [2];
t: [t0, if a > 0 then r-a else preceding] ↓ [2];
y: [y0, if a < 0 then a else v] ↓ [2];

The word preceding in the generation of a list denotes the immediately preceding element in the list. If it is necessary to refer to several preceding elements, this can be achieved as in the following example, which generate the first 1000 Fibonacci numbers:

Figure 2 – Simultaneous assignment of all the elements of a list

Algol (extended):

begin
  array t[1:m];
  integer i;
  inarray (t);
  for i:=1 step 1 until m do simultaneously
    y[i]:= t[i] ↑ 2;
  outarray (y);
end

Compel:

input t;
y: list of (each t) ↑ 2;
output y;

Programming methods

When programming in Compel, some traditional techniques cannot be employed and new methods must be substituted. Some of these methods will be discussed below.

To avoid circular dependence, the input variables of a statement must be different from the output variables; therefore, it is impossible to write the equivalent of Algol’s:
i:=i + 1;
Problems which are solved in Fortran or in Algol by repeatedly changing the values of various matrix elements might seem to be difficult to solve in Compel. Therefore, a matrix reduction by Gaussian elimination will be given as an example of an iterative algorithm. The matrix is stored as a list of rows, and the rows are each lists of elements.

The following block defines “gauss”, a function of three parameters: “a”, “b”, and “eps”, where “a” and “b” are matrices:

```
input:
gauss: φ [a, b, eps]:
  if length(a) < 1 then []
  else if last(list of abs pivot)> eps then
    list of pivr
  else undefined;
```

Before iteration begins, scaling is performed. First, “b” is put next to “a” to form the n by n+m matrix “ab”. Every row of “ab” is divided by the largest element (“maxv”) of its “a”-part:

```
iter: each [scale, while abs(pivot)> eps ∧ index <n use
  list of reduce] ;
i: each [1 to n];
n:length(a) ;
```

Every “row” in the matrix, excluding “pivr”, is reduced to “reduce” by subtracting from it that multiple of “pivr” which makes reduce = 0:

```
row: iter ! [each [1 to pivr - 1, pivr + 1 to n - i + 1]]i ;
reduce: list of (each row) - (row ! [1] pivr) x (each pivr) ;
```

In other words “a” and “b” are placed side-by-side, and row operations are performed which reduce the “a”-part to an upper triangular matrix.

If “a” is singular, Gauss returns undefined. The criterion for singularity is that the pivot in some step of the Gaussian elimination becomes 0 ± eps.

The following ten-statement block defines Gauss. Each iteration generates one row, “pivr” of the result matrix. These rows are collected, unless the matrix was singular:

```
output gauss ;
```

On a sufficiently parallel machine, the execution time for this function is proportional to n; on a sequential machine, Gaussian elimination time is roughly proportional to n^3.

**Programmer education**

The language Compel has been taught to several members of Midpeninsula Free University (Menlo Park, California) as part of a course in “Background for Computer Programming”. Practically the entire language was covered in two hours, and the members were able to solve simple problems like those in this paper. Among the reasons that it is easy to learn Compel are:

1. There is no “i = i + 1” paradox to excuse.
2. There is no need to explain “side effects”.
3. Substitution is completely unrestricted.
4. The data types are few, simple, and non-overlapping.
5. Each variable is defined exactly once.
6. Input and output files, considered lists of members, are in the same format as internal quantities, and are accessed in the same manner as
global variables. Therefore, there are no read, write, or format statements to confuse learning.

(7) There is only one kind of statement.

The following concepts, new to a non-programmer, needed to be taught:
(1) Church’s lambda notation (““ in Compel);
(2) input and output to statements and to blocks;
(3) the need to rewrite equations so that the unknowns are on the left side;
(4) split and collect.

In order to teach Fortran, it is necessary to discuss many machine-oriented restrictions, to introduce more new notations, and to concern the student with much detail which is of little relevance to the problems being solved.

SUMMARY

In single assignment languages, no variable is assigned values in more than one statement, and no statement has side effects.

Explicit indication of program sequence and concurrence is not required; these are determined implicitly by tracing dependence during compilation. Among the advantages of this class of languages are: no side effects; immaterial order of statements; common expression combination; compiler detection of all program concurrence; efficient compiler-determined release and management of storage.

The simple examples provide evidence that languages in this class are useful for general-purpose programming. A complete description of Compel and more practical examples are available from the authors.

ACKNOWLEDGMENTS

This research was supported (in part) by Grant MH 06645-07 from the National Institute of Mental Health. We are indebted to Kenneth Mark Colby for his encouragement during the course of this work, to James Lyon for drawing the figures, to William McKeeman for suggesting the Gaussian elimination algorithm, and to the referees for many valuable suggestions.

REFERENCES

1 J McCARTHY P W ABRAHAMS
D J EDWARDS T P HART M I LEVIN
Lisp 1.5 programmers manual
MIT Press Cambridge Massachusetts 1962
2 E D HOMER
An algorithm for selecting and sequencing statements as a basis for a problem-oriented programming system
Proceedings of the 21st National ACM Conference 1964
3 S SCHLESINGER L SASHKIN
POSE: A language for posing problems to a computer
CACM, vol 10 May 1967 pp 279-285
4 M E CONWAY
A multi-processor system design
Proceedings of the 1963 Fall Joint Computer Conference.
5 J A GOSDEN
Explicit parallel processing description and control in programs for multi- and uni-processor computers
Proceedings of the 1966 Fall Joint Computer Conference.
6 J P ANDERSON
Program structures for parallel processing
CACM Vol 8 Dec 1965 pp 786-788
7 N WIRTH
Letter, A note on program structures for parallel processing
CACM, Vol 9 May 1966 p 320
8 J P ANDERSON
Better processing through better architecture
Datamation Vol 13 (August 1967) pp 37-41
9 A J BERNSTEIN
Analysis of programs for parallel processing
10 H W BINGHAM D A FISHER W L SEMON
Detection of implicit computational parallelism from input-output sets
Burroughs Corporation TR-66-4 23 pages December 1966
11 D A FISHER
Program analysis for multiprocessing
Burroughs Corporation TR-67-2 71 pages May 1967
12 H HELLMAN
Parallel processing of algebraic expressions

From the collection of the Computer History Museum (www.computerhistory.org)