A GOOD PROGRAM = A STRUCTURED PROGRAM + OPTIMIZATION COMMANDS

Takeshi CHUSHO
Systems Development Laboratory, Hitachi, Ltd.
Ohzenji, Tama-ku, Kawasaki, Japan

An interactive optimization system is a tool for applying structured programming to a field with severe object efficiency requirements. A system is described in which optimization of a structured Pascal program is performed with a combination of primitive commands for flexibility and avoidance of a large catalogue of transformation rules. The system automatically verifies all optimization commands so as to eliminate retesting, and makes it possible to use a structured program instead of the optimized program at maintenance time. The command verification of each command is proved by commands on the basis of the hierarchy of command construction. An experiment is described which confirms the effectiveness of our system.

1. INTRODUCTION

For the past few years, considerable effort has been spent on improving software productivity, reliability, and maintainability. However, the greatest attention has been focused on "structured programming." The meaning of structured programming has not been clearly defined but the essence of this approach is to produce a program with high readability. There are well-known techniques for structured programming such as structured coding[1] and top-down development by stepwise refinement[2,3].

Modularization techniques[4,5] and abstraction techniques[6] have also been investigated.

On the other hand, adequate attention has not been focused on the main disadvantage of structured programming, i.e., the degradation of object efficiency. This is because software productivity has been considered more important than object efficiency in large computer systems. In minicomputer and microcomputer systems, however, this is not necessarily true because the main memory capacity is limited and efficient memory use is very important. Therefore, when structured programming is applied to these systems, software must be developed with the following two conflicting criteria:

(1) to produce well-structured programs, and
(2) to produce efficient programs.

The simplest solution to the above-mentioned problem is to separate optimizing from structuring. This methodology is called two-stage programming in this paper. It expands Dijkstra's principle[2] of "one decision at a time" into "one decision at a time with one criterion," and makes programming easier.

At the second stage, there are three ways to optimize a structured program: manual, interactive, and automatic. The following problems result from manipulation by a programmer.

(1) The optimized program must be tested again for assurance of correctness of optimization.
(2) Maintenance of the optimized program is difficult because of low readability so that the merits of structured programming are lost.

On the other hand, automatic manipulation by a translator or preprocessor limits the scope of optimization to rather simple cases because the cost of higher level optimization is extremely high compared with its effect.

An interactive system, however, can exclude these defects of the other two approaches if the system optimizes a program according to commands entered by a programmer, proves correctness of these commands, and preserves an original well-structured program.

In previous studies, Knuth[7] proposed the concept of an interactive program-manipulation system. Standish et al.[8,9] refined this concept with examples introducing general enabling conditions such as commutativity, freedom from side-effects, and invariance. Lowman[10] intended to improve a program by source-to-source transformation. Although his main purpose is high level optimization in the compilation process, the interactive approach is also mentioned. These approaches, however, need large transformation catalogues in which users must find a suitable transformation, and may cause degradation of reliability at the point where a programmer is expected to verify program equivalence. Arsaet[11] makes such a catalogue of more primitive transformations such as syntactic and local semantic transformations.

The author and his colleague have also studied an interactive optimization system[12] in connection with the development of the structured programming language SPL[13]. Our conclusions on the interactive optimization system are:

(1) Proof of correctness of all optimization commands is necessary in order to keep the merits of a structured program during its lifetime.
(2) The function of each command should be primitive and each optimization should be performed with a combination of several primitive commands, so that the system may be flexible enough to perform various kinds of optimization and avoid a large catalogue of transformation rules.

Our original system was designed as a conversational restructuring, optimizing, and parti-
tional system (CROPs) [12] for SPL. This system, however, seemed to be too large for implementation. Therefore, at the experimental implementation stage, Pascal was selected instead of SPL. The remainder of this paper describes the interactive optimization system for structured Pascal programs, named CROPs/Pascal.

2. INTERACTIVE PROGRAMMING SYSTEM

Two-stage programming is composed of the following steps:

(1) Initial programming
(2) Restructuring
(3) Testing and debugging
(4) Performance evaluation
(5) Profile analysis
(6) Optimizing

The first three steps are in the structuring stage. The last three steps are in the optimizing stage and are repeated until the requirements of object efficiency are satisfied.

The testing and debugging step is not necessary after optimization because of automatic proof of correctness of optimization commands. Since the well-structured program (SP) validated at (3) and a sequence of optimization commands (C) applied at (6) are preserved, the current version of an optimized program (OP) is in the relation OP = C(SP) and can be produced at any time by applying C to SP. This feature is very useful especially for maintenance.

3. CORRECTNESS PROOF OF OPTIMIZATION COMMANDS

3.1 Construction of commands

Commands of CROPs/Pascal are divided into the following four classes:

(1) Control commands for system management
(2) OPT commands for optimization or optimization preprocessing
(3) TEST commands for testing and debugging or for verification of the OPT commands
(4) EDIT commands for line/character manipulation or for restructuring

At the optimization step, only control and OPT commands are available for users.

3.2 Classification of OPT commands

In an interactive optimization system, extensibility of a command set is necessary because addition of new commands may be required. Therefore, OPT commands are processed hierarchically by using other OPT, TEST and EDIT commands in CROPs/Pascal. The proof of correctness of these commands, in particular, is performed by using other OPT and TEST commands. OPT commands are classified on the basis of proof methods as follows:

Type I: using only TEST commands
Type II: using only other OPT and TEST commands
Type III: using only other OPT commands
Type IV: using individual techniques
Type V: proof unnecessary

3.3 Verification of OPT commands

(1) Preparation
First, TEST commands and operators which are used in this section are introduced. The TEST commands are:

(1) MOD: getting a set of variables whose values are modified in the specified range or lexically successive statements
(2) USE: getting a set of variables whose values are used in the specified range
(3) LIVE: getting a set of statements in or after whose execution the specified variable may be referred to without modification
(4) LOCAL: verifying independence of two specified parts
(5) COMPARE: verifying semantic equivalence of two specified parts

Operators are:

(1) Sep: getting a set of continuous ranges from a set of statements
(2) Head: getting the upper boundary of the specified continuous range
(3) Tail: getting the lower boundary of the specified continuous range
(4) Pred: getting the predecessor of the specified statement
(5) Succ: getting the successor of the specified statement

In the remainder of this section, A/B means A or B and (n1, n2) means a range from n1 to n2.

The statement number n is composed of a line number k and a statement number m in the line, namely, km. In the case m = 1, however, #1 is omitted.

(ii) Type I

(1) RENAME vold, vnew, n1-n2
This command is used for preprocessing of optimization and replaces the name of the variable vold in (n1, n2) by a new name vnew.

This is verified in the following three cases:

(a) If vnew has not been declared, it will be correct due to the following process:

vnew := vold is generated at every entry point to (n1, n2) if the following condition is true:

[n1\in LIVE(vold)] \land [\forall \ r, vnew \in Sep\cdot LIVE(vold)].

(b) vold := vnew is generated at every exit point from (n1, n2) if the following condition is true:

[n2\in LIVE(vold)] \land [\forall \ r, vnew \in Sep\cdot LIVE(vold)].

(b1) If vnew has already been declared in the block including (n1, n2),
(b2) and otherwise, the same as (a).

(c) If vnew has already been declared outside the block including (n1, n2),

(c1) the condition of rejection is

vnew \notin (MOD(this block) \cup USE(this block)),
(c2) and otherwise, the same as (a).

In the cases of (a) and (c), the variable declaration of vnew is generated in the block including (n1, n2).

(2) MOVE n1-n2, n3
This command is also used for preprocessing of optimization and puts (n1, n2) after n3. This is correct if the following LOCAL command is true:

(a) LOCAL n1-n2, Succ(n3)\land Pred(n1) (if n1<n3)
(b) LOCAL n1-n2, Succ(n2)\land Pred(n1) (if n2<n3)
(3) DELETE LOOP(n)
This command deletes a for loop while leaving
the loop body. The loop body is enclosed with
an if statement if it may not be executed. The
condition of acceptance is
\[ \text{if statements } \land \text{body} \land \neg (r < \text{body}) \land \neg (r \cap \text{body} \neq q) \land \neg \text{Sep-LIVE}(v), \neg \text{MOD}(\text{body}) \land \text{USE} \}\}

(4) MERGE LOOP(n1, n2)
This command merges two for loops which begin
at n1 and n2 respectively and whose loop
controls are the same. It is verified by using
LOCAL body1, body2, condition.
The condition is \(i < j(\text{up})\) or \(i > j(\text{down})\) when i and j are loop variables.

(5) ROTATE LOOP(n1), n2-n3, FORTH
This command changes the first (n2,n3) of
the loop body with the last half, while copying
the first half which may be enclosed with an if
statement in front of the loop. The system must
verify whether the extra execution of the first
half is invalid. That is, the condition of acceptance is
\[ \{r \cap (n2,n3) = b\} \land \text{Tail}(r) < \text{body}, \forall r < \text{S-LIVE}(v), \forall r < \text{MOD}(n2,n3) \}

(iii) Type II

(1) MOVE IF(n1), n2-n3, n4-n5, FORTH/BACK
When (n2,n3) in a then phrase of an if
statement is the same as (n4,n5) in the else
phrase, this command moves these common parts
before/after the if statement and unifies them.
This is verified as follows:
(a) If the common part is not at the top/bottom in the then or else phrase, it is
moved there by using the MOVE command
mentioned in (ii).
(b) Next, COMPARE is used to verify whether
(n2,n3) is same as (n4,n5).

(2) REPLACE n1-n2, p
This command replaces (n1,n2) by the procedure
statement of p, and this operation is verified as
follows:
(a) (n1,n2) is temporarily replaced by p.
(b) This p is inline-substituted by EXPAND.
(c) (n1,n2) in the original program is com-
pared with the expanded part by COMPARE.

(iv) Type III

(1) SPLIT LOOP(n1), n2
A for loop beginning at n1 is split into two
for loops from n2, and this is verified as
follows:
(a) The loop is temporarily split.
(b) It is correct if MERGE is applied to these split loops and accepted.

(2) ROTATE LOOP(n1), n2-n3, BACK
The function of this command is inverse to the
ROTATE command with the option FORTH. It is
correct if the latter command is accepted after the
former command is temporarily applied.

(3) MOVE LOOP(n1), n2-n3
(n2,n3) in a for loop beginning at n1 is moved
in front of this loop for extraction of the
loop invariant. This is not primitive and
composed of the following three commands:

(a) MOVE n2-n3,n1 (if (n2,n3) is not at the
top of the loop body.)
(b) SPLIT LOOP(n1), Succ(n3)
(c) DELETE LOOP(n1)

(v) Type IV

(1) DELETE n
At some stages of applying a sequence of OPT
commands, invalid statements, such as an
assignment statement in which the left-hand
side is not referenced later, are sometimes
generated. This command excludes them.

(vi) Type V

(1) EXPAND LOOP(n), FIRST/LAST
The first/last repetition of a for loop
beginning at n is divided from the loop.
If the first/last repetition may not be
executed, the expanded part is enclosed by an
if statement with executable condition.

(2) EXPAND p(n)
A procedure statement or a function designator of
p at n is inline-substituted while replacing
formal parameters by actual parameters in
accordance with the following rules:
(a) A variable parameter directly replaces
references to the corresponding formal
parameter.
(b) A value parameter, in principle, is
assigned to a temporary variable, and
then, this variable replaces references to
the corresponding formal parameter.
In most cases, however, the temporary
variable is not necessary and the system
may automatically optimize.
(c) In the case of a function, a new variable
name is introduced instead of p.

(3) EXECUTE C/S n1-n2, v
References to the variable v in (n1,n2) are
replaced by its symbolic value. When the option
C is specified, the replaced values are limited
to constants.

(4) REDUCE n

4. SAMPLE PROGRAMMING

An experiment was conducted with sample
programming for the purpose of confirming the
effect of two-stage programming.

4.1 Structured programming

A program for calculation of the following
expressions was selected as a sample. This
program was necessary for research of a paging
algorithm.
\[
Q_{ij} = \begin{cases} 
1 & j = 1, \\
0 & \text{else} \end{cases}, \\
Q_{ij} = \begin{cases} 
\frac{\sum_{k=1}^{2|x|} L_k Q_{i-1,k}}{j} & j > 1, \text{or } j = 1, \text{and } 2|x| > j, \\
\frac{\sum_{k=1}^{2|x|} L_k Q_{i-1,k}}{j} & j = 1 \text{ and } 2|x| \leq j \leq j_{max}, \\
\frac{\sum_{j=1}^{j_{max}} L_j Q_{ij}}{j} & j \leq j_{max}, \\
\frac{\sum_{j=1}^{j_{max}} L_j Q_{ij}}{j} & j > j_{max}, \\
\end{cases}
\]
L_{i} is the reference probability to the j-th layer of an LRU stack. Q_{ij} is the existence probability in the j-th layer of the stack at the i-th time. N_{i} is the reference probability to a particular page at the i-th time. For practical use of this program, it was necessary for i to be greater than 10000 and for jmax to be 1024.

The initial program was developed with emphasis on high readability. That is, the above-mentioned expressions were transformed to a program with clarity of correspondence as shown in Fig. 1. This program, however, could not be executed because memory capacity was insufficient for the array variable Q. Therefore, IMAX was temporarily reduced from 10000 to 40. At that time, cpu time for execution of this program was about 20 minutes on a middle class computer HITAC M-160U. Consequently, optimization became necessary.

4.2 Optimization

(1) Space reduction

In order to make the program executable, the dimension of Q must be reduced from two to one. That is, instead of referring to Q after value assignment of all elements, each element of Q should be referenced directly after its value assignment. Therefore, first, the for loop referring to Q was unified into the other for loop defining Q, using the four commands shown:

```
800   procedure GETN;
900   var Q:array[1..IMAX,1..JMAX] of real;
1000  I,J:integer;
1100  function SUMLOQ(I:integer):real;
1200  var SUM:real;
1300  K:integer;
1400  begin
1500    SUM:=0.0;
1600    for K:=1 to JMAX do
1700      SUM:=SUM+L[X]*Q[I,K];
1800    SUML:=SUM;
1900  end;
2000  function SUML(TOP,BOTTOM:integer):real;
2100  var SUM:real;
2200  K:integer;
2300  begin
2400    SUM:=0.0;
2500    for K:=TOP to BOTTOM do
2600      SUM:=SUM+L[K];
2700    SUML:=SUM;
2800  end;
2900  begin
3000    Q[I,1]:=1.0;
3100    for J:=2 to JMAX do
3200      Q[I,J]:=0.0;
3300    for I:=2 to IMAX do
3400      begin
3500        Q[I,1]:=SUMLOQ(I-1);
3600        for J:=2 to JMAX do
3700          Q[I,J]:=Q[I-1,J]*Q[I-1,J-1];
3800        begin
3900          for I:=1 to IMAX do
4000            N[I]:=SUMLOQ(I);
4100        end;
4200    end;
```

Fig. 1. A well-structured program.

in Fig. 2(a) as follows:

(1) EXPAND makes the repetition numbers of two loops same.
(2) MOVE moves the expanded part in front of the first loop in order to make two loops adjacent.
(3) MERGE unifies two loops.

Next, the function SUMLQ referring to Q was inline-substituted for simplification of control flow, using five commands in Fig. 2(b) as follows:

(1) EXPAND replaces every function designator by its body. A new variable SUMLQ1 is introduced instead of SUMLQ.
(2) EXECUTE replaces references to the variable SUMLQ1 by references to the variable SUM.
(3) The next three DELETES exclude the assignment statements of SUMLQ1:=SUM because SUMLQ1 is never referenced.

Finally, the dimension of Q was reduced using twelve commands in Fig. 2(c) as follows:

(1) New array variables Q1 and PREQ1 with one dimension are introduced and replace Q using RENAME twice. These commands generate the following four assignment statements mentioned in 3.3(ii):

```
3402 Q[I,1]:=Q[I,*];
3404 PREQ1[I,*]:=Q[I-1,*];
3852 Q[I-1,*]:=PREQ1[*];
3860 Q[I,1]:=Q[*];
```

using "*" which represents all elements

EXPAND LOOP(4000), FIRST
MOVE 3950, 3200
MERGE LOOP(3300, 4000)

(a) Mergence of two loops.

EXPAND SUMLQ
EXECUTE S SUMLQ1
DELETE 3240
DELETE 3440
DELETE 3840

(b) Inline-substitution of a function.

RENAME Q[I,*,] Q[*,*], 3410-3850
RENAME Q[I-1,*], PREQ1[*], 3410-3850
DELETE 3402
ROTATE LOOP(3300), 3049, FORTH
EXECUTE S 3860-3870, Q[I,*]
DELETE 3860
DELETE 3852
RENAME Q1[*], PREQ1[*], 3000-3230
DELETE 2910
EXECUTE S 3252-3260, Q[1,*]
DELETE 3260
DELETE 3252

(c) Reduction of dimension of an array variable.

EXPAND SUMLQ
RENAME SUML1, SUML1[J,*], 3600-3800
RENAME SUML2, SUML2[J,*], 3600-3800
SPLIT LOOP(3600), 3700
MOVE LOOP(3300), 3600-3685

(d) Extraction of a loop invariant.

Fig. 2. A sequence of optimization commands.
of possible values corresponding to \( \# \) for convenience sake.

(2) The statement at 3402 is deleted because QI is not referenced.

(3) The statement at 3404 is moved to the bottom of the loop by \textsc{rotate}, dividing the case \texttt{I=2}. Consequently, the following two statements are generated:

\begin{verbatim}
3660 PREQJ[1] := QI[*,*];
3870 PREQJ[1] := QI[*,*];
\end{verbatim}

(4) \textsc{execute} replaces the right-hand side at 3870 by the right-hand side at 3860.

(5) Consequently, the statements at 3852 and 3860 can be deleted because Q is not referenced.

Thus, all QI[*,*]s and Q[1-1,*]s in the loop are excluded. QI[1,*]s are also excluded in the same way, using the other commands in Fig 2(c).

As a result of this optimization, the storage capacity for Q was reduced to be small and constant, whereas previously it was proportional to the maximum value of I.

\begin{verbatim}
800 procedure GETN;
900 var QI, PREQJ: array[1..JMAK] of real;
1000 I, J: integer;
1200 SUM1: real;
1300 K: integer;
2000 SUM1: real;
2100 SUMLJ1, SUMLJ2: array[1..JMAK] of real;
2900 begin
3000 PREQJ[1] := 1.0;
3100 for J := 2 to JMAK do
3200 PREQJ[J] := 0.0;
3210 SUM1 := 0.0;
3220 for K := 1 to JMAK do
3230 SUM1 := SUM1 + L[K] * PREQJ[K];
3250 N[1] := SUM1;
3254 for J := 2 to JMAK do
3258 begin
3262 SUM1 := 0.0;
3266 for K := 1 to J-1 do
3270 SUM1 := SUM1 + L[K];
3274 SUMLJ1 := SUM1;
3278 SUM1 := 0.0;
3282 for K := J to JMAK do
3286 SUM1 := SUM1 + L[K];
3290 SUMLJ2 := SUM1;
3294 end;
3300 for I := 2 to IMAX do
3400 begin
3410 SUM1 := 0.0;
3420 for K := 1 to JMAK do
3430 SUM1 := SUM1 + L[K] * PREQJ[K];
3500 QIJ[1] := SUM1;
3590 for J := 2 to JMAK do
3600 QIJ[J] := SUMLJ1 + L[J] * PREQJ[J];
3602 SUM1 := SUM1 + QIJ[J];
3800 N[J] := SUM1;
3870 for I := 2 to JMAK do
3880 PREQJ[I] := QIJ[J];
3900 end;
4200 end;
\end{verbatim}

Fig. 3. The optimized program.

(ii) Time reduction

For time reduction of this program, extraction of a loop invariant from the loop seemed to be effective because this optimization implies reduction of the number of nesting loops. That is, the summation of elements of L calculated by the function \textsc{suml} was a loop invariant with respect to the outmost loop beginning at 3300. The optimization was performed by commands shown in Fig. 2(d) as follows:

(1) The function designators of \textsc{suml} are inline-substituted.

(2) Calculation results are stored in the array variables instead of simple variables.

(3) \textsc{split} separates the invariant part from the loop body.

(4) The first split loop is moved in front of the outmost loop.

As a result of this optimization, the cpu time was reduced to 4.6\%, that is, from 2'1' 5" to 58" in \textsc{imax}=40. The final program is shown in Fig. 3.

4.3 Discussion of results

(i) Effects

This experiment confirmed the following effects of two-stage programming. The program could be produced easily and quickly at the first stage since we concentrated only on program structure. Moreover, there were no logical errors because the program directly corresponded to the function. At the second stage, the program efficiency was satisfactorily improved though the effect of optimizations depended on each program feature.

(ii) Auxiliary commands

In this experiment, twenty commands were used for space reduction and five commands for time reduction. Some readers may feel that the number of commands applied is a bit large for the optimization performed. These commands, however, are all primitive and most of them are auxiliary to optimization. As mentioned in the introduction, this is why the set of optimization commands is kept small for practical use. This method may be one solution to Loveman's question as to whether there is a complete set of transformations.

(iii) Optimization of algorithms

Some optimizations with respect to modification of algorithms were not performed in this experiment. For example, the following equation must be utilized in summation of \( L \).

\[
\sum_{k=1}^{k=\text{max}} L_k = 1.
\]

Although the program calculates \( \sum_{k=1}^{k=\text{max}} L_k \) and \( \sum_{k=1}^{k=\text{max}} L_k \) individually, the latter summation is gotten from the following expression using the former summation:

\[
\sum_{k=1}^{k=\text{max}} L_k = 1 - \sum_{k=1}^{k=\text{max}} L_k.
\]

Such optimization is difficult for our system because the system cannot automatically prove its correctness. It reminds us of Dijkstra's
conjecture[14] that often an efficient program could be viewed as the successful exploitation of a mathematical theorem. Further study is needed to solve this problem without changing our approach.

5. IMPLEMENTATION

5.1 Description language

CROS/Pascal is written in Pascal itself. One of the reasons for this is portability and the other reason is that the system can be applied to itself. This system is executed on the TSS system of HITMAC M-160II.

5.2 State-of-the-art for implementation

There are few new techniques required for implementation. A lot of hints with respect to flow analysis of this system were given from Hetch's paper on FORTRAN-S [15]. Bard's experience on [16] Pascal and others are helpful. As for symbolic execution, this system performs it in a simplified way. That is, the specified variable in the specified range is replaced by its symbolic value.

5.3 Extensibility

In such a system, extensibility is vital because addition of new commands is necessarily required. Therefore, optimization commands are hierarchically constructed of other commands as mentioned in 3.2. Moreover, the system is produced as a highly modularized program so that modification for addition of new commands may be localized.

6. CONCLUSIONS

An interactive optimization system for structured Pascal programs is described in this paper. The main features of our system are:

(1) All optimization commands are automatically verified in order to keep merits of a well-structured program during its lifetime and to make retesting after optimization unnecessary.

(2) Each optimization is performed with a combination of primitive commands so that the system may be flexible enough to perform various kinds of optimization and avoid a large catalogue of transformation rules.

Other features related to the implementation are as follows:

(3) Optimization commands are processed hierarchically by using other optimization, verification and edition commands in order to make addition of new commands easy.

(4) The verification methods for each optimization command are classified into five types on the basis of a hierarchy of command construction.

An experiment confirmed effectiveness of the system and suggested further study for algorithmic optimization with automatic proof of program equivalence.

ACKNOWLEDGEMENT

The author is indebted to T. Watanabe, T. Hayashi and other colleagues for their invaluable technical assistance.

REFERENCES