STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

Report MR 50

SWITCHING AND PROGRAMMING

by

A. van Wijngaarden

Lecture to be held at the Symposium on the Application of Switching Theory in Space Technology

Sunnyvale, California

March 1, 1962

SWITCHING AND PROGRAMMING

1. Introduction

Considerable attention has been paid in switching theory to the analysis and simplification of circuits and systems and to properties of nets. Common in these subjects is the structure of a net of often rather simple components, e.g., relays or diodes in series or parallel. Similar structures are found on another scale in the programs for automatic computers. These programs consist of sequences of statements performing certain operations, which are connected by transfers of control into a complicated network. Executing the statements means moving along the paths of the net, setting on the way conditions, i.e., assigning values to Boolean expressions which determine which controls of transfer shall take place, i.e., which route to take at each node of the net. Just like in the case of switching circuits, seemingly completely different structures may be more or less the same functionally, i.e., with regard of the result, and the problem of simplification rises immediately.

Of course, there is the question which of two equivalent programs is the simplest. There will be little doubt that replacing $x \vee \neg x$ by true means a simplification, but in more complicated cases it depends on the type of logical building bricks assumed to be available which of two logically equivalent forms is the simplest.

One may measure the complexity of a switching circuit in diodes or relays in some prescribed way and has then a norm to compare different forms, although other criteria like equal loading of elements may prevail under certain circumstances. In programs one would normally use criteria of length or duration although other criteria like generality or provision for easier ways of checking might be of importance. Again, some building bricks of more complex structure may be available at the same cost as more elementary bricks, just simply because they have been made. So, if and-circuits with five inputs are available at the same cost of two single diodes, this may change the optimum form of a circuit considerably.

./.

Also, if a procedure is available to compute a function of two arguments, then it may be advantageous to use that procedure even if the second argument is always zero rather than to write a new and simpler procedure for that special case. In the following we shall mainly use the length of the text as criterion for simplicity.

Of course, the structure of a program is much more complicated than that of a binary switching circuit and we cannot do more here than point out some of the obstacles that one encounters rather than to give useful rules to simplify programs. That we nevertheless try to do anything at all is because the subject is getting more and more interesting in view of automatic programming.

As language in which we express the program elements, we choose ALGOL 60 [1]. Due to the generality of its expressions, we can then deal concisely with quite general situations and absolute machine independence is guaranteed. Moreover, we can give a precise meaning to some concepts which are difficult to describe in another way.

2. Simplification of Identities

If we take the length of the text literally as the criterion of simplicity of a program, then we can simplify many programs directly by replacing long identifiers by short ones. This is, though true, so obvious that we shall assume it has been done already, and we can focus on those parts of the text which convey information.

It is natural to start by investigating the simplification of Boolean expressions that may occur in the program. Those expressions may occur as constituents of more complicated Boolean expressions, as right-hand side of an assignment statement, the left part list of which contains only Boolean variables, in an if clause of as actual parameters in a procedure statement or function designator. For instance the relation $\underset{..}{x} = \underset{..}{y}$ might occur in $\underset{..}{x} = \underset{..}{y} \vee \underset{..}{y} > \underset{..}{z}$, $\underset{..}{b} := \underset{..}{x} = \underset{..}{y}$, $\underline{\text{if}}\ \underset{..}{x} = \underset{..}{y}\ \underline{\text{then}}$

./.

u $\underline{\text{else}}$ v, $P(u, v, x = y)$.

As long as these Boolean expressions only contain logical values, logical operators and identifiers of variables of type Boolean, then we can apply without more all the techniques which have been developed in logic or switching theory for their simplification. As soon as they contain relations, however, then the situation demands a more careful investigation.

Let us start with the simplest possible expression of that type, viz., the relation $x = x$ and ask whether this might be simplified by replacing it by $\underline{\text{true}}$. This now only holds true under certain restrictions on the meaning and interpretation of x. First of all, the truth of $x = x$ does not without more imply the truth of $y = y$, since properties associated with the letter x are not necessarily associated with the letter y also.

Hence, let us describe more carefully what we want to know by asking whether an identity, as defined by

<identity> ::= <expression -1> = <expression -1> ,

can be replaced by $\underline{\text{true}}$. Here, as in the ALGOL 60 Report [1] sequences of character in the bracket < > represent metalinguistic variables whose values are sequences of symbols. The extension is that of the numbered metalinguistic variable, viz.,

<<letterstring> - <unsigned integer>>

representing a metalinguistic variable, which occurs in metalinguistic formulae and which is denoted by the same letterstring in the bracket < > but without the - sign and the unsigned integer, under the condition, however, that whenever in a metalinguistic formula the numbered metalinguistic variable occurs more than once, then it stands for the same, although arbitrary value of the corresponding metalinguistic

./.

variable. Hence, examples of an identity are:

x = x, y = y, x + y = x + y, read = read .

Such an identity cannot be replaced by true without more. Indeed, consider the following piece of program:

```
begin  procedure      P(t); string t; <code expressing that
                      the basic symbols of the string without
                      ‘and’ are printed>;
       integer procedure Q(t); string t; <code expressing that
                      the number of basic symbols of the string
                      without ‘and’ is the value of Q>;
       if Q (‘x = x’) > 2 then P (‘x = x’) end .
```

This ought to print: x = x, but if x = x were replaced by true under P then it would print true instead, and if x = x were replaced by true under Q also it would print nothing at all.

Of course, when an expression occurs inside a string, usually not its value but the sequence of its constituting basic symbols is of importance.

Hence, let us describe more carefully what we want to know by asking whether an identity not occurring inside a string can be replaced by true.

This is certainly not the case in general. Indeed if the expressions occurring withing the identity are Boolean expressions or designational expressions, then the identity itself is not an expression, since the only meaningful occurrence of the relational operator = is that in a relation, defined by

```
<relation> ::= <arithmetic expression> <relational operator>
               <arithmetic expression>,
```

./.

which is a special case of a Boolean expression. In the cases mentioned above, the value of the identity is therefore undefined and hence it cannot be replaced by true without more. Let us introduce the concept of a proper identity by

```
<proper identity> ::= <arithmetic expression -1> =
                      <arithmetic expression -1> .
```

Then we can describe more carefully what we want to know by asking whether a proper identity not occurring inside a string can be replaced by true.

This is certainly not the case in general. One has to realize that the expression may contain a function designator, and that in the evaluation of that function designator a goto statement may be executed which defines its successor as a statement outside the procedure body so that the evaluation of the expression remains unfinished forever. For instance, the following piece of program

```
begin   integer k ;
        integer procedure x; if k < 0 then goto end else x := 1;
        procedure print (t); integer t; <code expressing that the
        value of t is printed> ;
        k := -1; if x = x then k := 1;
end:    print (k)   end
```

ought to print: -1, but if x = x were replaced by true it would print 1.

Let us call an expression evaluable if it does not contain any function designator in the body of whose procedure declaration there occurs a goto statement leading out of it. Then we can describe more carefully what we want to know by asking whether a proper identity not occurring inside a string and comparing two identical evaluable expressions, can be replaced by true.

./.

Again, however, this is not the case in general. The arithmetic expression may contain a function designator whose corresponding procedure depends for its operation on values which are altered by its own evaluation. For instance, the following useful procedure is of that type:

```
integer  procedure  altsign 1(b); Boolean b;
         begin  altsign 1  := if b then 1 else -1;
                              b := ¬ b  end .
```

Its value is in turn equal to 1 and to -1 if at least the value of the actual variable corresponding to b, B say, is not changed between two occurrences of its identifier. In this example the variable b is the case of this abnormal behavior, and one might think that by investigating the effect of the procedure upon its formal parameters, one could see whether the function designator is of this class. However, the variable b does not need to appear in the parameter list at all. It may be a non-local variable, as in the following procedure

```
integer  procedure  altsign 2;
         begin altsign 2 := if b then 1 else -1; b := ¬ b  end
```

or it may be an own local variable, as in

```
integer  procedure  altsign 3;
         begin own Boolean b;
               altsign 3 := if b then 1 else -1;
                            b := ¬b  end .
```

In the case, by the way, that one wants a sequence of results to start with 1, one may have the sequence of occurrences of the function designator preceded by respectively

./.

```
B := true    or
b := true    or
if altsign 3 = 1 then altsign 3 .
```

In all these cases one might still investigate what the procedure does by inspecting its body. This would, however, already present difficulty in a case like

```
integer  procedure  altsign 4; <code>
```

in which the body is expressed in some non-ALGOL language, although for its action altsign 4 might be equivalent to altsign 3.

Even in the case that one might be able to interpret the code, there might be a last source of values upon which the value of the function designator depends, viz., some outside source of information. For instance, read might be a function designator, whose value is that of the next number read from a tape, which after reading is advanced to what is then the next number.

An extreme case is provided at last by the function designator random whose value is by definition unpredictable.

It is obvious that if these so-called function designators with side effects occur, our identity cannot be replaced by true in general. It must be emphasized that checking for the absence of side effects is often very difficult and can even often only be done during run time so that perhaps the easiest way to find out whether or not the identity can be replaced by true is to evaluate it. Obviously, this does not help very much.

There is, however, an important class of expression, to be called stable expressions, viz., all those expressions whose value does not depend on the order of evaluation, provided they occur within one basic statement. Examples are numbers, logical values, simple variables,

./.

subscripted variables if the subscript expressions are stable expressions, relations whose arithmetic expressions are stable expressions, and all expressions which can be generated with those as primaries, i.e., therefore, all expressions which can be generated without using a function designator. This subset of stable expressions is easily recognizable in the text. It is, however, unduly limited. We can distinguish in a procedure four categories of parameters, viz., input parameters, output parameters, dummy parameters, and mixed parameters.

Input parameters are those non-local identifiers and the actual parameters corresponding to those formal parameters which appear only in the value list or within the procedure body as variables in the right-hand side expression of assignment statements or in subscript expressions not within a function designator or again as input parameters of function designators and procedure statements.

Output parameters are those non-local identifiers and the actual parameters corresponding to those formal parameters which occur only within the procedure body in the left part list of assignment statements or as output parameters of function designators and procedure statements.

Dummy parameters are those non-local identifiers and the actual parameters corresponding to those formal parameters which either do not appear in the value list and procedure body at all or appear only as designational expressions, procedure identifiers or dummy parameters of function designators and procedure statements.

Mixed parameters are those non-local identifiers and the actual parameters corresponding to those formal parameters which do not belong to one of the three categories mentioned above.

If the function designators in an expression have no output and mixed parameters then the expression is stable. Also if within a basic statement the two sets of the input parameters and the output parameters of itself and all function designators contained within it, and the mixed parameters one by one are disjunct then all expressions

./.

contained within it are stable. Obviously, in this order it becomes more difficult but still possible to check whether the conditions are satisfied.

We can now define more carefully what we want to know by asking whether a proper identity, not occurring inside a string, comparing two identical, evaluable, stable expressions can be replaced by true.

There is still some difficulty in answering yes to this question. If the arithmetic expressions in the identity are of type integer then the answer is yes. If they are of type real, however, it is a question of interpretation. Indeed, the ALGOL 60 Report [1] defines concerning the arithmetic of real quantities:

> "Numbers and variables of type real must be interpreted in the sense of numerical analysis, i.e., as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently."

First of all, one might evaluate the left-hand side expression by means of another hardware representation than the right-hand side, since a large computing system might very well, taking advantage of the evaluability and stability of the expressions, evaluate them simultaneously on different component computers, which parts might work with different precision. But even if this is not the case, one might conceive that the freedom specified above is taken to include the possibility that performing the same computation twice or even calling the same variable twice might yield different results or again that the comparison of two values is an operation involving arithmetic and hence subject to inaccuracy, all possibilities which are not excluded by the wording.

./.

Actually, computation on an analogue computer has these features. One might even write a piece of program like:

```
begin  integer i, j; real x;
                j := 0;
       for i := 1 step 1 until 1000 do
       begin x := random; if x = x then j := j + 1  end;
       for i := 0 step 1 until 1000 - j do <computation> end,
```

where a computation is repeated a number of times depending upon the quality of the computer which is used, which is determined by the program itself! This program would be spoiled completely if $x = x$ were replaced by true.

If digital computation is understood, then one has

(i) if E1 and E2 are results of two identical computations then E1 = E2.

(ii) a variable is a quantity which does not vary unless another value is assigned to it.

After these preliminaries, we can state that a proper identity not occurring inside a string and comparing two identical evaluable stable expressions of type integer or also of type real if digital computation is assumed can be simplified to true.

3. Simplification of Relations

Before we go on we shall assume in order to avoid the same cumbersome wording over and over again that from now on we are dealing with expressions not occurring inside a string, that they are proper, evaluable and stable and that digital arithmetic is understood.

./.

Arithmetic expressions we shall denote by $e1$, $e2$, $e3$.

So far, we have still only dealt with the relation operator $=$ and we widen now our relations by introducing the operators $<$ and $>$. If $e1 = e2$ can then $e1 < e2$ be simplified to false? This cannot be said without more. Actually in a wellknown computer system hold both relations $-0.0 = 0.0$ and $-0.0 < 0.0$. Disregarding those slips we define as proper arithmetic an arithmetic in which the set of real numbers - i.e., numbers of type real or integer - is ordered, so that exactly one of the relations $e1 < e2$, $e1 = e2$, $e1 > e2$ holds. This does not imply that $-0.0 = 0.0$ but it excludes a case as mentioned above. We assume this arithmetic to be proper and postulate the following axioms and definitions:

A1: $e1 = e1$,
A2: $+e1 = e1$,
A3: $(e1) = e1$,
A4: $e1 = e2 \equiv e2 = e1$,
A5: $e1 \neq e2 \equiv \neg e1 = e2$,
A6: $e1 \neq e2 \equiv e1 > e2 \vee e1 < e2$,
A7: $\neg(e1 > e2 \wedge e1 < e2)$,
A8: $e1 > e2 \equiv e2 < e1$,
A9: $e1 \geq e2 \equiv \neg e1 < e2$,
A10: $e1 \leq e2 \equiv \neg e1 > e2$,
A11: $e1 > e2 \wedge e2 > e3 \supset e1 > e3$.

This is as far as one can go without specifying the type of the expressions. One might think, for instance, that

$$e1 = e2 \wedge e2 > e3 \supset e1 > e3$$

might hold. This is certainly not the case. Indeed, there is no objection against an arithmetic in which a relation like $er1 = ei1$ can

./.

hold true. Here the letters r usp i specify the type of the expression to be $\underline{\text{real}}$ resp. $\underline{\text{integer}}$. For instance, $1.00_{10}3 = 1001$ and $1.00_{10}3 = 1002$ could very well hold and since integers are dealt with exactly, certainly $1002 > 1001$. One would, however, find $1.00_{10}3 = 1002 \wedge 1002 > 1001 \supset 1.00_{10}3 > 1001$ against the supposition.

Actually, the formulae with one or two $=$ signs run as follows:

A12: $e1 = er2 \wedge er2 > e3 \supset e1 > e3$,
A13: $e1 = er2 \wedge er2 < e3 \supset e1 < e3$,
A14: $e1 = er2 \wedge er2 = er3 \supset e1 = er3$,
A15: $e1 > ei2 \wedge ei2 = ei3 \supset e1 > ei3$,
A16: $e1 < ei2 \wedge ei2 = ei3 \supset e1 < ei3$;
A17: $e1 = ei2 \wedge ei2 = ei3 \supset e1 = ei3$.

This system enables to simplify relations. For instance, if x, y, z are of type $\underline{\text{real}}$ and i and j of type $\underline{\text{integer}}$ then

$$x < y \wedge (y > x \vee y > z) \equiv x < y ,$$
$$x < y \wedge y < z \wedge z = x \equiv \underline{\text{false}} ,$$

but

$$x = i \wedge x = j \wedge i > j$$

cannot be simplified.

4. Simplification of Relations Containing Arithmetical Operations

The next step consists of introducing actual arithmetic operations. Again, proper arithmetic is supposed to satisfy a set of axioms, some of which are

./.

$$e1 + e2 = e2 + e1 \text{ ,}$$
$$e1 > e2 \supset e3 + e1 \geq e3 + e2 \text{ ,}$$
$$e3 + e1 > e3 + e2 \supset e1 > e2 \text{ .}$$

These three satisfy already to simplify, for instance,

$$\begin{array}{l} \underline{\text{if}}\quad x + y > y + z \ \underline{\text{then}}\ (\underline{\text{if}}\ x > z\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 2) \\ \qquad\qquad\qquad\quad \underline{\text{else}}\ \ \underline{\text{if}}\ x > z\ \underline{\text{then}}\ 3\ \underline{\text{else}}\ 4 \end{array}$$

into

$$\underline{\text{if}}\quad x + y > y + z\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \underline{\text{if}}\ x > z\ \underline{\text{then}}\ 3\ \underline{\text{else}}\ 4 .$$

5. Simplification of Statements

Next, we turn our attention to syntactical units of a higher level than expressions, viz., to statements.

Special attention deserves the separator $:=$ which is actually partly an arithmetic operator. Indeed, there is no reason to assume that the evaluation of expressions of type real is performed with the same precision as in which the results are remembered. Usually the expressions are evaluated in higher precision than that of the variables so that the assignment usually includes a round-off (or chopping in very poor machines).

This means that the sequence

$$x := y \times z \ ;\ i := \underline{\text{if}}\ x = y \times z\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 2$$

cannot be simplified into

$$x := y \times z \ ;\ i := 1 ,$$

on the contrary, $i := 2$ is a much better guess!

However, proper arithmetic is supposed to include that assignment is well defined and idempotent so that when $v1$ and $v2$ are

./.

variables of the same type, the two axioms hold true:

$$\text{A18:}\quad \underset{\cdot\cdot\cdot\cdot}{v1} := \underset{\cdot\cdot\cdot\cdot}{v2} := \underset{\cdot\cdot\cdot\cdot}{e1} \supset \underset{\cdot\cdot\cdot\cdot}{v1} = \underset{\cdot\cdot\cdot\cdot}{v2},$$

$$\text{A19:}\quad \underset{\cdot\cdot\cdot\cdot}{v1} := \underset{\cdot\cdot\cdot\cdot}{v2} \supset \underset{\cdot\cdot\cdot\cdot}{v1} = \underset{\cdot\cdot\cdot\cdot}{v2}\ .$$

They enable, for instance, to simplify

$$\underset{\cdot\cdot}{x} := \underset{\cdot\cdot}{y} := \underset{\cdot\cdot}{z} \times \underset{\cdot\cdot}{z}\ ;\ \underset{\cdot\cdot}{i} := \underline{\text{if}}\ \underset{\cdot\cdot}{x} = \underset{\cdot\cdot}{y}\ \underline{\text{then}}\ \underset{\cdot\cdot}{1}\ \underline{\text{else}}\ \underset{\cdot\cdot}{2}$$

into

$$\underset{\cdot\cdot}{x} := \underset{\cdot\cdot}{y} := \underset{\cdot\cdot}{z} \times \underset{\cdot\cdot}{z}\ ;\ \underset{\cdot\cdot}{i} := \underset{\cdot\cdot}{1}\ .$$

6. Simplification of Procedures

At last, we turn our attention to the bigger lumps of program, like procedures. Obviously, the possible gain by simplification is here the greatest but it seems hard to point out any substantial recognizable points of attack. Something can be said about passing on parameters from one procedure to the next, which is again a somewhat simply ordered structure, but really important simplification is rather dealt with by mathematical than logical methods. Also, the requirement of absolute equality of results may become rather academic. Also, it is not obvious whether or not it is improper to treat the assignment to the procedure identifier of a function designator inside the procedure body like that to a variable, i.e., including a possible loss of precision. In many classical programming schemes, the function designator would be handled by means of a subroutine which includes round off. On the other side in the ALGOL 60 compiler made by E.W. Dijkstra and J.A. Zonneveld for the ELECTROLOGICA X1 computer [2] the function designator does not include the loss of precision. However, in none of both cases this is logically conditioned, and one may have subroutines which compute a double length result or modern procedures which round off. It makes, however, a

./.

difference. Consider, for instance, the following two procedures for computing the sum for k from a to b of the expression fk :

```
          real procedure sum (k, a, b, fk) ; value b;
                         integer k, a, b; real fk;
              begin      real s;
                         s := 0; k := a;
                    L:   if k ≤ b then begin s := s + fk; k := k + 1;
                         goto L end;
                         sum := s      end
```

and

```
          real procedure sum (k, a, b, fk); value a, b;
                         integer k, a, b; real fk;
                         if a ≤ b then begin k := a; sum := fk + sum
                         (k, a + 1, b, fk) end
                         else sum := 0 end.
```

Formally, we ought to say that the first procedure cannot be simplified into the second one, since the arithmetical result is not necessarily equal. But since this difference would presumably also be to the advantage of the second procedure, it can hardly be said to be a reasonable objection.

- - - - -

References:

[1] Backus, J.W. a.o.: Report on the Algorithmic Language ALGOL 60.
Dedicated to the memory of William Turanski.
Edited by Peter Naur, Regnecentralen, Copenhagen, 1960.

[2] Dijkstra, E.W.: ALGOL-60 Translation. ALGOL Bulletin Supplement nr. 10. Report MR 35 Computation Department Mathematical Centre, Amsterdam, November 1961.

- - - - -