Three-valued logic and computer science

Albert Hoogewijs
(Rijksuniversiteit Gent)

A three-valued logic considers besides the classical truth-values T (true) and F (false), a third value U (undefined). According to the interpretation of this value such as meaningless, intermediate, neutral or indeterminate, one gets different truth-tables for the logical connectives and hence different formalizations for the three-valued logic (see [9] and [10]).

Now we want to consider some practice of this notion in Computer Science and deduce the appropriate interpretation and formalization for the third value in this context.

---

1. Digital networks and switching theory

Switching theory or the logical design of digital networks, deals with the developing of digital systems that are to carry out particular information-processing tasks. Since binary coding provides for an adequate way to handle this process, the whole theory relies on the realization of functions of the form

$$f : \{0,1\}^m \longrightarrow \{0,1\}^n,$$

which can be decomposed into n functions of the form

$$f_i : \{0,1\}^m \longrightarrow \{0,1\}, \quad i = 1,\ldots,n.$$

From the existence of the disjunctive normal form for such functions, they can be represented in the form

$$f(x_1,\ldots,x_m) = \bigvee_{\substack{(a_1,\ldots,a_m)\\ f(a_1,\ldots,a_m)=1}} x_1^{a_1}\ldots x_m^{a_m}$$

where $x^1 := x$ and $x^0 := \overline{x}$ (the complement of x)

Hence digital networks can be composed of elementary *AND*-, *OR*- and *COMPLEMENT*-gates which are the realizations of the classical logical connectives $\wedge$, $\vee$, $\neg$ and where 1 stands for T and 0 for F. Since the disjunctive normal form can easily be obtained from the function-table, as the following example shows, the whole problem seems to be solved.

However in the majority of the cases, this normal form must be simplified in some way to provide an ecomical hardware solution.

1.1. Example

x_1	x_2	x_3	x_4	$f(x_1,x_2,x_3,x_4)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

$$
\begin{aligned}
f(x_1,x_2,x_3,x_4) = {} & \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4 \vee \overline{x}_1\overline{x}_2\overline{x}_3x_4 \vee \overline{x}_1\overline{x}_2x_3\overline{x}_4 \\
\vee {} & \overline{x}_1\overline{x}_2x_3x_4 \vee \overline{x}_1x_2x_3\overline{x}_4 \vee \overline{x}_1x_2x_3x_4 \\
\vee {} & x_1\overline{x}_2\overline{x}_3x_4 \vee x_1\overline{x}_2x_3\overline{x}_4
\end{aligned}
$$

1.2. Minimization methods

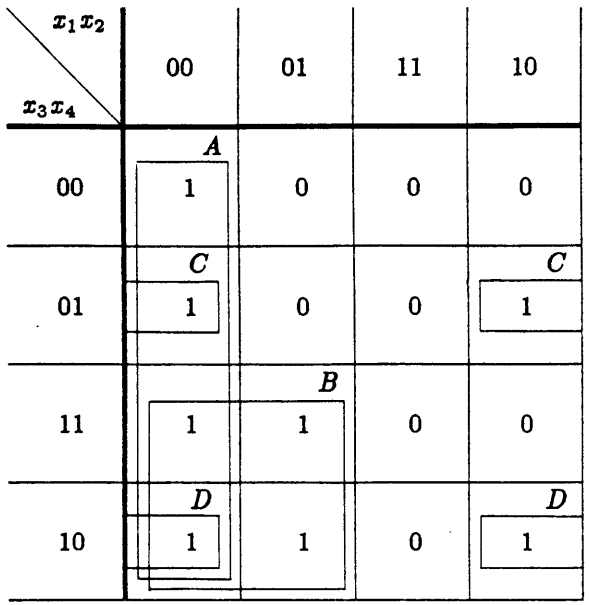
A first way to simplify the canonical form is using the following rule of Boolean Algebra :

$$
AxB \vee A\overline{x}B = AB \qquad \text{(R)}
$$

where A and B are any formulas of the form $x_{i_1} \ldots x_{i_n}$ or 1. This method has been systematized into two procedures one of which we will illustrate on the considered example. For more details we refer the reader to [7] et [2].

The **Karnaugh map method** is a graphical procedure which is easy to use for functions with at most four arguments. The starting point is a picture (called the map of the function), containing 2^n labelled squares (where n is the number of arguments). The labels consist of binary codes and stand at the top and the left side of the picture (as on ordinary maps). They are arranged in such a way that the labels of two adjacent squares (note that the first and the last line [column] are considered to be adjacent) differ only in one variable. Each square contains a binary code according to the value of the function (which has to be represented) applied to the label of the that square.

In this way the map of the fonction f from the example becomes :



The next step in this procedure is combining an even number of adjacent squares containing 1, in order to form rectangles or squares which refer to the terms that can be simplified. For the considered example we get :

$$f(x_1,x_2,x_3,x_4) = \underset{A}{\overline{x}_1\overline{x}_2} \vee \underset{B}{\overline{x}_1x_3} \vee \underset{C}{\overline{x}_2\overline{x}_3x_4} \vee \underset{D}{\overline{x}_2x_3\overline{x}_4}$$

where each of this terms is obtained by encoding the common part of the labels of the squares in the corresponding blocks.

For a larger number of variables there is the **Quine-McCluskey procedure** which consists of a systematic enumerative technique to find out on which terms the reduction rule (R) may apply (See the given references).

1.3. Practical design of a logic circuit

Assume that we have to devise a circuit that will control a coffee-and-tea machine which offers the possibility to add sugar and/or milk. Coding the possibilities we get :

INPUT				OUTPUT
x_1	x_2	x_3	x_4	
0	0	0	0	no output
.	.	0	1	coffee
.	.	1	0	tea
.	0	.	.	without sugar
.	1	.	.	with sugar
0	.	.	.	without milk
1	.	.	.	with milk

Hence we have to consider a 4-bits input.

Since the circuit must control 5 relays $R_1, \ldots, R_5$ for the dosage of the coffee, the tea, the sugar, the milk-powder and the water resp., we have to consider 5 functions $f_i : \{0,1\}^4 \longrightarrow \{0,1\}$.

Now it is clear that one may not ask for coffee and tea in the same cup. Hence the machine must be constructed in such a way that an input ..11 is impossible. This leads to the so-called "*don't care* conditions" in the function-table (see [7], [2]). If we denote them U we get the following function tables :

x_1	x_2	x_3	x_4	f_1	f_2	f_3	f_4	f_5
0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	1
0	0	1	0	0	1	0	0	1
0	0	1	1	*U*	*U*	*U*	*U*	*U*
0	1	0	0	0	0	1	0	1
0	1	0	1	1	0	1	0	1
0	1	1	0	0	1	1	0	1
0	1	1	1	*U*	*U*	*U*	*U*	*U*
1	0	0	0	0	0	0	1	1
1	0	0	1	1	0	0	1	1
1	0	1	0	0	1	0	1	1
1	0	1	1	*U*	*U*	*U*	*U*	*U*
1	1	0	0	0	0	1	1	1
1	1	0	1	1	0	1	1	1
1	1	1	0	0	1	1	1	1
1	1	1	1	*U*	*U*	*U*	*U*	*U*

If we interpret U as "not important", we may take it as 0 and we get for

the function f_3 :

$$\begin{aligned} f_3(x_1, x_2, x_3, x_4) &= \overline{x}_1 x_2 \overline{x}_3 \overline{x}_4 \;\vee\; \overline{x}_1 x_2 \overline{x}_3 x_4 \;\vee\; \overline{x}_1 x_2 x_3 \overline{x}_4 \\ &\vee\; x_1 x_2 \overline{x}_3 \overline{x}_4 \;\vee\; x_1 x_2 \overline{x}_3 x_4 \;\vee\; x_1 x_2 x_3 \overline{x}_4 \end{aligned}$$

Applying the Karnaugh map method we get :

x_3x_4 \ x_1x_2	00	01	11	10
00	0	1	1 B	0
01	0	1	1 A	0
11	U_1	U_2	U_3	U_4
10	0	1	B 1	0

Combining terms as shown on the map gives us :

$$f_3(x_1, x_2, x_3, x_4) = \underset{A}{x_2 \overline{x}_3} \;\vee\; \underset{B}{x_2 \overline{x}_4}.$$

However, looking at the map, we see that if $U_1 = U_4 = 0$ and $U_2 = U_3 = 1$ then we can consider a rectangle of 8 squares. Hence f_3 becomes :

$$f_3(x_1, x_2, x_3, x_4) = x_2.$$

This means that we have considered the U's of the function-table as "not yet defined" with the possibility to become either 0 or 1 in the Karnaugh map.

With this interpretation in mind one gets the following truth-tables for $\neg$, $\wedge$, $\vee$.

$\neg$	
0	1
1	0
U	U

$\wedge$	0	1	U
0	0	0	0
1	0	1	U
U	0	U	U

$\vee$	0	1	U
0	0	1	U
1	1	1	1
U	U	1	U

2. The third value U in programming

2.1. Introduction

Consider the expression $(y = 0 \ OR \ x/y = 5)$ where x and y are integers. Since x/y is not defined in the case that $y = 0$, there is no classical way to use the statement

$$\texttt{if } (y = 0 \ OR \ x/y = 5) \texttt{ then } S_1 \texttt{ else } S_2 .$$

One has to translate this statement into the forms :

$$\begin{array}{l} \texttt{if } \ y = 0 \ \texttt{ then } S_1 \\ \qquad\quad \texttt{else if } \ x/y = 5 \ \texttt{ then } S_1 \\ \qquad\qquad\qquad\quad \texttt{else } S_2 \end{array}$$

In [3], Gries introduces two operators :

CAND : conditional *AND*
COR : conditional *OR*

defined in the following truth tables :

CAND	*T*	*F*	*U*
T	*T*	*F*	*U*
F	*F*	*F*	*F*
U	*U*	*U*	*U*

COR	*T*	*F*	*U*
T	*T*	*T*	*T*
F	*T*	*F*	*U*
U	*U*	*U*	*U*

They can be explained considering the interpretation

$$'b' \ CAND \ 'c' \ := \ \texttt{if } 'b' \texttt{ then } 'c' \texttt{ else } F.$$

A computer is supposed to evaluate that expression from left to right. If the value of *'b'* is *T* he will evaluate *'c'* and acording to this value the result will be *T*, *F*, or *U* (where *U* means abortion of the program). If *'b'* is *F* the whole expression is *F*. If *'b'* is *U* the program aborts, hence the result is *U*.

Similary one gets an interpretation for *COR* as

$$'b' \ COR \ 'c' \ := \ \texttt{if } 'b' \texttt{ then } T \texttt{ else } 'c'.$$

A serious objection against the use of these operators is the fact that they are not commutative.

Gries uses the *CAND* and *COR* in his "Programming language" :

- which more or less looks like Pascal or Algol,
- is much simpler in use,
- offers some facilities in proving programs correct.

However there are still no interpreters nor compilers available to run such programs.

The usefullness of the language may be compared to the use of flowcharts in the older days, which were ment to clarify the program structure. But where flowcharts in most cases were written after the program was finished, to provide for some documentation, this language may be used

- to produce a well-structured program,
- to prove the program correct,
- as a proper base for a well-founded program in most available programming languages.

2.2. Some specifications of Gries' language

2.2.1. A guarded command is an expression of the form

$$B \longrightarrow S,$$

where the "guard" B is a generalized boolean expression (using *CAND* and *COR*) and S some instruction to be executed if B is true. They can be combined to form :

2.2.2. An **alternative command**, which in its general form looks like

$$\begin{array}{ll} \textbf{if} & B_1 \longrightarrow S_1 \\ \square & B_2 \longrightarrow S_2 \\ \square & \quad \ldots \\ \square & B_n \longrightarrow S_n \\ \textbf{fi} & \end{array}$$

where **if** and **fi** mark beginning and end of the command resp. and $\square$ separates the component guarded instructions. On execution the expressions B_i are evaluated, but without any rules on the order of evaluation;

- if a B_i is found to be undefined, abortion occurs,
- if no B_i is true then execution aborts,
- if a B_i is found to be true the corresponding S_i is executed.

2.2.3. The **iterative command** which has the general form

$$\begin{array}{ll} \textbf{do} & B_1 \longrightarrow S_1 \\ \square & B_2 \longrightarrow S_2 \\ \square & \quad \ldots \\ \square & B_n \longrightarrow S_n \\ \textbf{od} & \end{array}$$

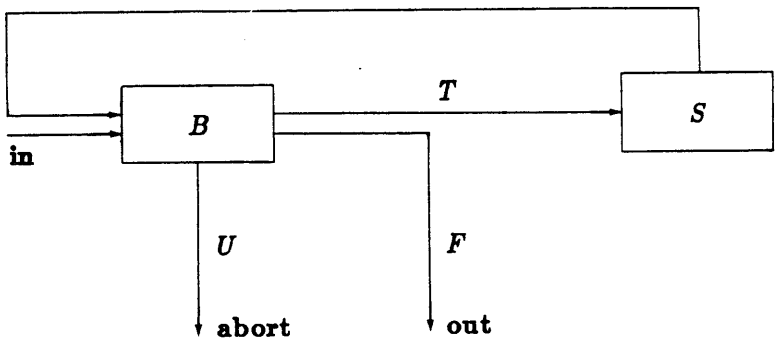
Beginning and end are marked **do** and **od** resp. and $\square$ is used as separator of the components. On execution, a search is started for true guards B_i and the corresponding S_i are executed. Upon termination all the guards are false.

2.3. Example

A simple example may illustrate the use of the latter. Consider the problem of finding the position i of an element x in an array $b[0..n-1]$ if x belongs to the array else if x doesn't belong to $b[0..n-1]$ put $i := n$. The following statement does the job :

do ($i < n$ *CAND NOT* $(x = b[i])$) $\longrightarrow$ $i =: i + 1$ **od**

B S



2.4. Proving programs correct

Gries discusses the problem of proving correctness of loop execution. He introduces 3 predicats :

- the precondition Q, which must be true when execution starts;
- the invariant P, which must be true before, during and after execution of the loop;
- the result assertion R, which must be true after execution of the loop.

It follows that

$$\neg BB \wedge P \Longrightarrow R,$$

where $BB = \bigcup_{i=1}^{n} B_i$.

For the considered example we have :

$$
\begin{array}{lcl}
Q & := & 0 \leq n \\
P & := & 0 \leq i \leq n \ \wedge \ x \notin b[0..n-1] \\
R & := & (0 \leq i \leq n \ \wedge \ x = b[i]) \ \vee \ (i = n \ \wedge \ x \notin b[0..n-1]).
\end{array}
$$

---

3. Recursive procedures and undefinedness

3.1. Introduction

Consider the definition

$$subp(i,j) := [\ \texttt{if}\ i = j\ \texttt{then}\ 0\ \texttt{else}\ subp(i+1,j)+1\ ]$$

which is a recursive procedure that can be defined in PASCAL or ALGOL .... We get

$$subp(i,j) = j - i \quad \text{if } j \geq i$$

but $subp(i,j)$ is not defined if $j < i$, since in this case the procedure does not halt. In order to prove that for all integers i, j

$$i < j \implies subp(i,j) = j - i, \tag{P}$$

one nees a three-valued logic.

Barringer, Cheng and Jones in [1] propose to use the following truth-tables for the connectives $\neg$, $\wedge$ and $\vee$.

$\neg$	
T	T
F	T
U	U

$\wedge$	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

$\vee$	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Note that those table correspond to tables we got in 1.
where $1 := T$ and $0 := F$. Moreover McCarty [8] observes that one get the tables for $\wedge$ and $\vee$ from the tables of *CAND* and *COR* if one introduces the

$$\text{axiom}: \texttt{(if 'b' then 'a' else 'a')} \ = \ \texttt{'a'}.$$

Besides they consider partially defined terms and use the following definitions for equality

$=$	x	y	$\perp$
x	T	F	U
y	F	T	U
$\perp$	U	U	U

$==$	x	y	$\perp$
x	T	F	F
y	F	T	F
$\perp$	F	F	T

where x and y stand for two different terms and $\perp$ denotes an undefined term.

The formalization of those connectives and the quantors $\forall$ and $\exists$ leads to a proof theory which enables Barringer *et al.* to prove some procedures, involving partial definedness, to be correct.

3.2. Example

As an illustration we copy the proof which shows that the property (P) follows from classical properties of the integers, the following rules of the theory :

$$\forall\text{-introduction :} \quad \frac{p(x)}{\forall x.\, p(x)} \qquad (\forall\text{-I})$$

$$\forall\text{-elimination :} \quad \frac{\forall x.\, p(x), s = s}{p(s/x)} \qquad (\forall\text{-E})$$

$$=\text{-substitution} \quad \frac{s_1 = s_2, p}{p(s_2/s_1)} \qquad (=\text{-subs})$$

and the properties for subp which follow from the definition :

$$d_1 : \quad \frac{}{subp(n, n) = 0}$$

$$d_2 : \quad \frac{n_1 \neg n_2, subp(n_1 + 1, n_2) = n_3}{subp(n_1, n_2) + n_3 = 1}$$

Note that the proof is based on natural deduction.

$\vdash \forall i, j.\ j - i \geq 0 \implies subp(i, j) = j - i.$

1.	$\vdash \forall k.\ k \geq 0 \implies subp(j-k, j) = k.$		
	1.1	$\vdash subp(j-0, j) = 0$	d_1
	1.2	$k \geq 0,\ subp(j-k, j) = k \vdash subp(j-(k+1), j) = k+1$	
		1.2.1 $k \geq 0$	prem 1.2
		1.2.2 $k+1 > 0$	integers
		1.2.3 $j-(k+1) \neq j$	integers
		1.2.4 $subp(j-k, j) = k$	prem 1.2
		1.2.5 $j-k = j-(k+1)+1$	integers
		1.2.6 $subp(j-(k+1)+1, j) = k$	=-subs -1,-2
		1.2.7 $subp(j-(k+1), j) = k+1$	d_2 -1,-4
	1.3	$\forall k.\ k \geq 0 \Rightarrow subp(j-k, j) = k$	induct 1.1, 1.2
2.	$\vdash j-1 \geq 0 \Rightarrow subp(j-(j-i), j) = j-i$		$\forall$-E -1
3.	$\vdash j-1 \geq 0 \Rightarrow subp(i, j) = j-i$		integers -1
4.	$\vdash \forall i, j.\ j-i \geq 0 \Rightarrow subp(i, j) = j-i$		$\forall$-I twice -1

In [5] we show that natural deduction implies that selfdenial – and selfassertion – rules cannot hold in the proof theory.

$$\text{(SeDe)}\quad \frac{L,\ A \ \vdash\ \neg A}{L \ \vdash\ \neg A}, \qquad \text{(SeAs)}\quad \frac{L,\ \neg A \ \vdash\ A}{L \ \vdash\ A}$$

It follows that easy classical proofs are much harder to obtain. However if we introduce the symbol Δ as in PPC [4], we still have interesting rules such as

$$\frac{L_1, A \ \vdash\ \neg A \qquad L_2 \ \vdash\ \Delta A}{L_{12} \ \vdash\ \neg A} \qquad \text{and} \qquad \frac{L_1, A \ \vdash\ B \qquad L_2 \ \vdash\ \Delta A}{L_{12}, \neg B \ \vdash\ \neg A}$$

which may simplify at least some of those proofs. For further details we refer to [6].

4. References

[1] BARRINGER H., CHENG J.H., JONES C.B.,
A logic Covering Undefinedness in Program Proofs,
Acta informatica **21** (1984), pp. 251–269.

[2] BOOTH T.,
Digital networks and Computer Systems,
John Wiley, New York, 1971.

[3] GRIES D.,
The science of programming,
Springer Verlag, New York, 1983.

[4] HOOGEWIJS A.,
On a formalization of the Non-definedness Notion,
Zeitschr. f. math. Logik **25** (1979), pp. 213–221.

[5] HOOGEWIJS A.,
The Partial Predicate Calculus PPC and Undefinedness in Computer science,
Logic Colloquium Manchester, 1984.

[6] HOOGEWIJS A.,
Cut-rule in a logic for Program Proofs,
Draft, Gent, 1984.

[7] LEWIN D.,
Logical design of Switching Circuits,
Nelson, London, 1968.

[8] McCARTHY J.,
A basis for a Mathematical theory of Computation,
in Computer Programming and Formal Systems, (ed. P. Braffort and D. Hirschberg), North-Holland, 1967.

[9] RESCHER N.,
Many-valued Logic,
McGraw Hill, New York, 1969.

[10] WOLF R.,
A Survey of Many-valued Logic (1966–1974),
in Modern Uses of Multiple-valued Logic, (ed. J.M. Dunn and G. Epstein), Reidel, 1977, pp. 167–323.

Rijksuniversiteit Gent
Galglaan 2
B-9000 Gent