A FOUR-VALUED LOGIC

J. Ulisses Ferreira

Trv Pirapora 36 Costa Azul, 41770-220, Salvador, Brazil

ABSTRACT

This short and informal article briefly introduces a four-valued logic.

KEYWORDS

four-valued logic, logic, philosophy of computer science

1. INTRODUCTION

In early 20th century, Kleene, Łukasiewicz and Priest individually proposed their three-valued logics[1], whereas Łukasiewicz and Tarski introduced their propositional calculi into the literature[2], while a considerable number of contributions appeared on many-valued logics.
The paraconsistent logic was simultaneously and independently created by two logicians, namely, the Polish logician Stanisław Jaśkowski and the Brazilian logician Newton da Costa[3].

In 1977, Nuel Belnap (1930-) proposed his logic on four values[4]. His two extra values are *N* (none) and *B* (both) in the four-valued Belnap's logic, and they correspond to *uu* and *ii*, respectively, in both referred logics of the present author, as well as in his four-valued logic introduced in this paper.

In 1988, the present author started his Master research inserting a third value called "unknown" in a programming language[5] that he was designing at that time. The *unknown* constant has been theoretically referred to as *uu* since the end of nineties. Later, a five-valued logic was introduced containing the values in *{tt, ff, uu, ii, kk}*. In 2004, the same logic was published as a journal article[6] and a seven-valued logic was also published in a Conference in San Diego[7], adding the values *{fi, it}* ("false or inconsistent", "inconsistent or true", respectively) for being able to be used together with the same uncertainty model that had been proposed during the present author's Master course in 1990.

As part of the present author's previous contribution, the *kk* value means "knowable", and it is usable when something is not already known, but it is already known that it is consistent. It can either be *true* or *false* but not both. It can be known by God or someone else or some machine, for instance, but it is not already known by the machine which is deductively reasoning, or by the person who is deductively doing, and it may be unknown forever, but at least its consistency is guaranteed beforehand. This is the meaning of the *kk* value, which fits in the referred uncertainty model when a variable thresholds collapse: *False* = *True*, which means that there is nothing strictly between the *False* and the *True* thresholds.

In 2017, the four-valued logic being introduced was briefly presented in [8].

Natarajan Meghanathan et al. (Eds) : NeCoM, SEAS, CMCA, CSITEC - 2017
pp. 71– 84, 2017. © CS & IT-CSCP 2017 DOI : 10.5121/csit.2017.71206

Section 2 introduces the present author's four-valued logic. Section 3 makes some comparisons to other logics. Section 4 shows the program that the present author wrote to discover his logic. Subsection 4.1 presents an example of input data whereas subsection 4.2 shows a C source code. Section 5 presents the properties of the logic that is being introduced, whereas section 5 contains the conclusions.

2. FERREIRA FOUR-VALUED TRUTH TABLES

Table 1. Ferreira four-valued logic truth tables

A	¬A
U	I
F	T
T	F
I	U

∧	U	F	T	I
U	U	F	U	U
F	F	F	F	F
T	U	F	T	I
I	U	F	I	I

∨	U	F	T	I
U	U	U	T	I
F	U	F	T	I
T	T	T	T	T
I	I	I	T	I

→	U	F	T	I
U	T	T	T	T
F	T	T	T	T
T	U	F	T	T
I	F	U	T	T

↔	U	F	T	I
U	T	F	F	F
F	F	T	F	F
T	F	F	T	F
I	F	F	F	T

3. SOME COMPARISONS

In some multi-valued logics, which regard uncertainty such as probability, after some considerations, the numeric values (e.g. in the interval [0,1]) are translated to a few values using words (such as true, false, unknown and inconsistent) in order to be used in sentences and complex expressions containing the well-known logical operators. In the end, one computes and knows the logical value of the whole expression.

In da Costa's paraconsistent logic, *(tt ∨ ff = ii)* as well as *(tt ∧ ff = uu)* which do not look natural. In Belnap's four-valued logic, *(B ∨ N = T)*, where *T* represents the true value. That is to say, in terms of applications, the expression *(ii ∨ uu = tt)* does not look natural either. Such criticisms led to the work on the logic being introduced here.

4. A PROGRAM FOR HELPING DISCOVER THE DESIRED LOGIC

In 2007, having the present author set his own requirements in a form towards constraint programming, and written the corresponding program below, which seems to be correct, he checked whether such a four-valued logic which he desired exists, and its computation resulted in several such logics. He observed that one of the solutions was Belnap's four-valued logic. One of the solutions written by the program computation was chose for being the logic which was the most interesting. In 2011, one could not claim the authorship of that four-valued logic, but the referred truth tables are those introduced in section 2, above.

The computation of the program `a4vlogic.c` makes use of the input data `a4vlogic.dat`, and, before resuming, it generates the output file called `a4vlogic.sol` containing the solutions found if any, and reads and writes an auxiliary file called `a4vlogic.aux`.

The input data contains the truth tables for ∧, ∨, ¬ as well as ↔ operators, respectively, followed by the → partial truth table. Each truth table element (implicitly indexed by a row and a column) contains one of the four values. Originally, the value was not in the same order as shown in section 2, table 1, but in the following order, *(ff, uu, ii, tt)*, and the labels of the tables are not written, as they are implicitly represented by just their contents, except for the output on screen as well as the row labels written to the output file. Wherever one of the four values is read from the input file, it means that the same value is a constraint programmed by the present author. The

only truth table that is not fully filled in is the truth table for →. In subsection 4.1, all entries of the truth table for implication are left for the program, although some constraints could be imposed. In fact, there were a number of experiments, and subsection 4.1 just presents the last one. In every table entry that contains "..", it means that the program role is to try all possibilities for that entry and, thus, to produce all possible and different truth tables. In this way, the program permits the corresponding logician to either impose a value or let the program to try all n possibilities in that entry, where $n = 4$ in the present logic. In terms of efficiency, the program can be improved for eliminating an intermediate number of possibilities if the corresponding logician wishes, in such a way that its computation can be carried out much faster, for some large n. Since 4, 5 and 7 are relatively small numbers for the present C program, and just those numbers have been part of the present author's work, he has not been concerned about making the suggested improvement.

The auxiliary file is important for the program can be easily adapted for another number n of values. The greater this number n is, the longer time the computation takes. If n is sufficient large, it is important that the computation often write the current state in a file, in particular, because the computation was being performed in the present author's personal computer, which can be sometimes switched off. In the present logic, where $n = 4$, the response was as fast as while the present author was slipping in one night. Nonetheless, for greater n, the computation may take even many years for giving the first solution if there is any.

Whenever the program starts running, the calculation of the possible logics continues from the point contained in the auxiliary file. Thus, this file contains the last state only, whereas the output file is meant to concatenate all the solutions that are being found, if any.

While in the loop, for regarding the current truth table as a solution, the program computation checks more than 12 properties, which are written as part of the C code. If all desired properties hold, the program computation appends the current truth table as a new solution, advances one value, and the loop carries on.

4.1 THE CONSTRAINTS

```
ff ff ff ff
ff uu uu uu
ff uu ii ii
ff uu ii tt

ff uu ii tt
uu uu ii tt
ii ii ii tt
tt tt tt tt

tt ii uu ff

tt ff ff ff
ff tt ff ff
ff ff tt ff
ff ff ff tt

.. .. .. ..
.. .. .. ..
.. .. .. ..
.. .. .. ..
```

4.2 THE C CODE

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<string.h>
//#include<sys/time.h>

// NN é o número de linhas ou de colunas da matriz lógica verdade
#define NN 4
#define SN "4"

// M = NN - 1 ?
#define M 3
// IX é NN * NN - 1
#define IX 15
// NN2 = NN * NN
#define NN2 16

// NNM1 = NN + 1
#define NNM1 5

// True, o índice do valor máximo, = M
#define TT 3

//#define BT 8
//#define IT 7
//#define UU 3

typedef char word[3];

// "tt" e ".." devem ser sempre os últimos símbolos,
// e "ff" deve ser o primeiro.
const word v[NNM1] =
{"ff", "uu", "ii", "tt", ".."};
// "ub", "ff", "fb", "fi", "bb", "kk", "ii", "bt", "it", "tt",
"ui", ".."};

#define FALSE 0
#define TRUE 1

typedef int tab[NN2][NN2];

typedef struct {
  int i, j;
} indices;

// índices de 0 a IX
indices hash[NN2];

tab and,or,imp,eqv,impmsk/*,imp2,imp3,imp4*/;

int not[NNM1];
```

```
FILE *in, *sol;

int nn = -1;

void ler(char im, tab t) {
  word w;
  char achou;
  int i, j, k;
  for (i=0; i<NN; i++) {
      for (j=0; j<NN; j++) {
        fscanf(in,"%s ",w);
        printf("w = %s\n",w);
        achou = FALSE;
        for (k = 0; !achou && k<=M+1; k++) {
            if (strcmp(w,v[k])==0) {
              achou = TRUE;
              t[i][j] = k;
              if (k == M+1) {
            if (im) {
                    hash[++nn].i = i; hash[nn].j = j;
                    imp[i][j] = 0;
            }
            else {
                    printf(
"Erro: máscara %s fora da tabela da implicação.\n",w);
                    exit(1);
            }
        }
         }
        }
       if (!achou) {
         printf("Dado incorreto no arquivo de entrada: %s.\n",w);
            exit(1);
        }
      }
  }
  printf("LEU\n");
}

void escreve(FILE *f, tab t) {
  int i,j;
  for (i=0; i<NN; i++) {
      fprintf(f,"%s |",v[i]);
      for (j=0; j<NN; j++) {
        fprintf(f," %s",v[t[i][j]]);
      }
      fprintf(f,"\n");
  }
  fprintf(f,"\n");
}

unsigned long exp(int base, int x) {
  unsigned long n = 1;
```

```
  while (x > 0) {
    n = n * base;
    x--;
  }
  return n;
}

tab A, B, C;

void err(char *e, char n) {
  (*e)++;
  printf("Erro #%d\n",n);
}

char mask(tab imp) {
  int a, b, c;
  char erro = 0;

  a = 0;
  while (erro == 0 && a<=M) {
    // nagação dupla
    if (a != NN && not[a] != NN && eqv[a][not[not[a]]] != TT)
      err(&erro,1);

    // P1. Identity:
     if (imp[a][a] != NN && eqv[imp[a][a]][TT] != TT)
err(&erro,2);

    b=0;
    while (erro == 0 && b<=M) {

      // contrapositiva
      if (imp[a][b] != NN && imp[not[b]][not[a]] != NN &&
          eqv[imp[a][b]][imp[not[b]][not[a]]] != TT)
err(&erro,3);
      // a and b --> a or b
      if (imp[and[a][b]][or[a][b]] != NN &&
          eqv[imp[and[a][b]][or[a][b]]][TT] != TT) err(&erro,4);

       // De Morgan 1
      if (a != NN && b != NN && or[a][b] != NN &&
          not[or[a][b]] != NN && not[a] != NN && not[b] != NN &&
          and[not[a]][not[b]] != NN &&
          eqv[not[or[a][b]]][and[not[a]][not[b]]]!=TT) {
         printf("a=%d b=%d\n");
           err(&erro,5);
      }

      // De Morgan 2
      if (a != NN && b != NN && and[a][b] != NN &&
          not[and[a][b]] != NN && not[a] != NN && not[b] != NN &&
          or[not[a]][not[b]] != NN &&
        eqv[not[and[a][b]]][or[not[a]][not[b]]]!=TT) {
```

```
          printf("a=%d b=%d
%d=%d\n",a,b,not[and[a][b]],or[not[a]][not[b]]);
            err(&erro,6);
      }

      /*
      if (imp[a][b] != imp2[a][b] && imp[a][b] != imp3[a][b]
       && imp[a][b] != imp4[a][b]) erro++;
      */

      // P6. a => (b => a)
      if (a != NN && b != NN && imp[b][a] != NN &&
          imp[a][imp[b][a]] != NN &&
          eqv[imp[a][imp[b][a]]][TT] != TT) err(&erro,7);

      // P7. ((a => b) => a) => a
      if (a != NN && b != NN && imp[a][b] != NN &&
          imp[imp[a][b]][a] != NN &&
          imp[imp[imp[a][b]][a]][a] != NN &&
          eqv[imp[imp[imp[a][b]][a]][a]][TT] != TT) err(&erro,8);

      c = 0;
      while (erro == 0 && c<=M) {
        // P2. (a => (b => c)) => (b => (a => c))
        if (a != NN && b != NN && c != NN && imp[b][c] != NN &&
            imp[a][imp[b][c]] != NN && imp[a][c] != NN &&
            imp[b][imp[a][c]] != NN &&
            imp[imp[a][imp[b][c]]][imp[b][imp[a][c]]] != NN &&
            eqv[imp[imp[a][imp[b][c]]][imp[b][imp[a][c]]]][TT] !=
TT)
              err(&erro,9);
        // P3. (c => a) => ((b => c) => (b => a))
        if (c != NN && a != NN && imp[c][a] != NN &&
            b != NN && imp[b][c] != NN && imp[b][a] != NN &&
            imp[imp[b][c]][imp[b][a]] != NN &&
            imp[imp[c][a]][imp[imp[b][c]][imp[b][a]]] != NN &&
            eqv[imp[imp[c][a]][imp[imp[b][c]][imp[b][a]]]][TT] !=
TT)
              err(&erro,10);
        // P4. (c => a) => ((a => b) => (c => b))
        if (c != NN && a != NN && imp[c][a] != NN &&
            imp[a][b] != NN &&
            imp[c][b] != NN && imp[imp[a][b]][imp[c][b]] != NN &&
            imp[imp[c][a]][imp[imp[a][b]][imp[c][b]]] != NN &&
            eqv[imp[imp[c][a]][imp[imp[a][b]][imp[c][b]]]][TT] !=
TT)
              err(&erro,11);
        // P5. (a => (b => c)) => ((a => b) => (a => c))
        if (a != NN && b != NN && c != NN && imp[b][c] != NN &&
            imp[a][imp[b][c]] != NN &&
            imp[a][b] != NN &&
            imp[a][c] != NN && imp[imp[a][b]][imp[a][c]] != NN &&
            imp[imp[a][imp[b][c]]][imp[imp[a][b]][imp[a][c]]]
```

```
            != NN && eqv[
               imp[imp[a][imp[b][c]]][imp[imp[a][b]][imp[a][c]]]
            ][TT] != TT) err(&erro,12);
        /*
        // associativa and
        if (eqv[and[and[a][b]][c]][and[a][and[b][c]]]!=TT)
erro++;
        // associativa or
        if (eqv[or[or[a][b]][c]][or[a][or[b][c]]]!=TT) erro++;
        // distributiva and-or
        if (eqv[or[and[a][b]][c]][and[or[a][c]][or[b][c]]]!=TT)
          erro++;
        // distributiva or-and
        if (eqv[and[or[a][b]][c]][or[and[a][c]][and[b][c]]]!=TT)
          erro++;
        */
        c++;
      }
      b++;
    }
    a++;
  }
  return erro == 0;
}


char tabok(tab imp) {
  int a, b, c;
  char erro = 0;

  a = 0;
  while (erro == 0 && a<=M) {
    // nagação dupla
    if (eqv[a][not[not[a]]] != TT) erro++;

    // P1. Identity:
    if (eqv[imp[a][a]][TT] != TT) erro++;

    b=0;
    while (erro == 0 && b<=M) {

      // contrapositiva
      if (eqv[imp[a][b]][imp[not[b]][not[a]]] != TT) erro++;
      // a and b --> a or b
      if (eqv[imp[and[a][b]][or[a][b]]][TT] != TT) erro++;

      // De Morgan 1
      if (eqv[not[or[a][b]]][and[not[a]][not[b]]]!=TT) erro++;
      // De Morgan 2
      if (eqv[not[and[a][b]]][or[not[a]][not[b]]]!=TT) erro++;

      /*
      if (imp[a][b] != imp2[a][b] && imp[a][b] != imp3[a][b]
```

```
       && imp[a][b] != imp4[a][b]) erro++;
      */

      // P6. a => (b => a)
      if (eqv[imp[a][imp[b][a]]][TT] != TT) erro++;

      // P7. ((a => b) => a) => a
      if (eqv[imp[imp[imp[a][b]][a]][a]][TT] != TT) erro++;

      c = 0;
      while (erro == 0 && c<=M) {
        // P2. (a => (b => c)) => (b => (a => c))
        if (eqv[
              imp[imp[a][imp[b][c]]][imp[b][imp[a][c]]]
            ][TT] != TT)
          erro++;
        // P3. (c => a) => ((b => c) => (b => a))
        if (eqv[
              imp[imp[c][a]][imp[imp[b][c]][imp[b][a]]]
            ][TT] != TT) erro++;
        // P4. (c => a) => ((a => b) => (c => b))
        if (eqv[
              imp[imp[c][a]][imp[imp[a][b]][imp[c][b]]]
            ][TT] != TT) erro++;
        // P5. (a => (b => c)) => ((a => b) => (a => c))
        if (eqv[
              imp[imp[a][imp[b][c]]][imp[imp[a][b]][imp[a][c]]]
            ][TT] != TT) erro++;

           // transitiva
           if (imp[and[imp[a][b]][imp[b][c]]][imp[a][c]] != TT)
             erro++;
           // associativa and
           if (eqv[and[and[a][b]][c]][and[a][and[b][c]]]!=TT)
             erro++;
           // associativa or
        if (eqv[or[or[a][b]][c]][or[a][or[b][c]]]!=TT)
          erro++;
        // distributiva and-or
        if (eqv[or[and[a][b]][c]][and[or[a][c]][or[b][c]]]!=TT)
          erro++;
        // distributiva or-and
        if (eqv[and[or[a][b]][c]][or[and[a][c]][and[b][c]]]!=TT)
          erro++;
        c++;
      }
      b++;
    }
    a++;
  }
  return erro == 0;
}
```

```
int main(void) {
  char fn[13];
  sprintf(fn,"a%svlogic.dat",SN);
  in=fopen(fn,"rt");
  if (in==NULL) {
     printf("Não pôde abrir arquivo %s.\n",fn);
  }
  else {
    int i, j, k;
    unsigned long ii;
    FILE *aux;
    word w;
    char c, achou, fim;
    ler(FALSE,and);
    ler(FALSE,or);
    for (j=0; j<NN; j++) {
      fscanf(in,"%s ",w);
      achou = FALSE;
      for (k=0; !achou && k<=M; k++) {
        if (strcmp(v[k],w)==0) {
          achou = TRUE;
          not[j] = k;
        }
      }
      if (!achou) {
        printf("Dado incorreto no arquivo de entrada: %s.\n",w);
        exit(1);
      }
    }
    ler(FALSE,eqv);
    ler(TRUE,impmsk);
    /*
    ler(FALSE,imp2);
    ler(FALSE,imp3);
    ler(FALSE,imp4);
    */
    fclose(in);
    printf("Conjunção:\n");
    printf("   | ");
     for (i=0; i<NN; i++) printf("%s ",v[i]);
     printf("\n---+------------------------------\n");
     escreve(stdout,and);

     printf("Disjunção:\n");
     printf("   | ");
     for (i=0; i<NN; i++) printf("%s ",v[i]);
     printf("\n---+------------------------------\n");
     escreve(stdout,or);

     printf("Equivalência:\n");
     printf("   | ");
     for (i=0; i<NN; i++) printf("%s ",v[i]);
     printf("\n---+------------------------------\n");
```

```
      escreve(stdout,eqv);

      printf("Negação:\n");
      printf("   | ");
      for (i=0; i<NN; i++) printf("%s ",v[i]);
      printf("\n---+-----------------------------\n");
      printf("   | ");
      for (j=0; j<NN; j++) printf("%s ",v[not[j]]);
      printf("\n");

      for (i = 0; i < NN; i++) {
        for (j = 0; j < NN; j++)
            imp[i][j] = impmsk[i][j];
      }

      printf("\nImplicação:\n");
      printf("   | ");
      for (i=0; i<NN; i++) printf("%s ",v[i]);
      printf("\n---+-----------------------------\n");
      escreve(stdout,imp);
      printf("\n");

      if (!mask(imp)) {
        printf("A máscara impossibilita qualquer solução!\n");
      }
      else {
        printf(
"exp(10,%d)-1 = %u (talvez muito grande p/ se representar
assim).\n",
                 nn+1,exp(10,nn+1)-1);

        sprintf(fn,"a%svlogic.aux",SN);
        aux = fopen(fn,"rt");
        if (aux == NULL)
            printf("Não pôde abrir arquivo %s.\n",fn);
        else {
            int x = 0, j = 0;
            time_t t; struct tm *tv;
            printf("Buscando soluções a partir da tentativa #");
            k = 0;
            do {
              fscanf(aux,"%c",&c);
              if (c >= '0' && c <= '9') {
                  printf("%c",c);
                  if (k > 0)
                    imp[hash[k].i][hash[k].j] = c - '0';
                  else
                    imp[hash[k].i][hash[k].j] = c - '0' - 1;
                  k++;
              }
            } while (c >= '0' && c <= '9');
            fclose(aux);
            printf("\n");
```

```
            fim = FALSE;
            while (!fim) {
              time(&t);
              tv = localtime(&t);
              if (tv->tm_min % 5== 0 && tv->tm_sec == 0) {
                  sprintf(fn,"a%svlogic.aux",SN);
                  aux = fopen(fn,"wt");
                  for (k = 0; k <= nn; k++)
               fprintf(aux,"%c",'0'+imp[hash[k].i][hash[k].j]);
             fprintf(aux,"\n");
             fclose(aux);
             while (tv->tm_min % 5 == 0 && tv->tm_sec == 0) {
               time(&t);
               tv = localtime(&t);
                  }
             }

              imp[hash[x].i][hash[x].j]++;
              while (x <= nn && imp[hash[x].i][hash[x].j] > TT) {
                  imp[hash[x].i][hash[x].j] = 0;
                  x++;
                  if (x <= nn) imp[hash[x].i][hash[x].j]++;
              }

              /*
              printf("Verificando a seguinte implicação:\n");
              escreve(imp);
              */
              if (x > nn) fim = TRUE; else {
                  x = 0;
                  if (tabok(imp)) {
                    printf("Solução #%d\n",++j);
                    escreve(stdout,imp);
                    printf("\n\n");
                    sprintf(fn,"a%svlogic.sol",SN);
                    sol = fopen(fn,"at");
                    if (sol == NULL) {
                        printf("Não pode abrir o arquivo de
soluções %s.\n",fn);
                        exit(1);
                    }
                    else {
                        fprintf(sol,"Solução #%d\n",j);
                        escreve(sol,imp);
                        fprintf(sol,"\n\n");
                        fclose(sol);
                    }
                  }
              }
          }
        printf("Total, %d soluções.\n",j);
      }
    }
```

```
  }
  return(1);
}
```

5. THE LOGICAL PROPERTIES

Some properties of this four-valued logic are the following:

Identity: $(A = A)$ holds as the $\leftrightarrow$ truth table ensures this principle. $(A \rightarrow A)$ also holds for any A. The truth tables ensure that $\wedge$, $\vee$, $\leftrightarrow$ are commutative and associative:

$(A \leftrightarrow B) \leftrightarrow (B \leftrightarrow A)$.

$(A \wedge B) \leftrightarrow (B \wedge A)$. $(A \vee B) \leftrightarrow (B \vee A)$.

$(A \leftrightarrow (B \leftrightarrow C)) \leftrightarrow ((A \leftrightarrow B) \leftrightarrow C)$.

$(A \wedge (B \wedge C)) \leftrightarrow ((A \wedge B) \wedge C)$. $(A \vee (B \vee C)) \leftrightarrow ((A \vee B) \vee C)$.

The law of excluded third: $(A \vee \neg A)$. It holds for any Boolean value, but results in *ii* for either *uu* or *ii*. The intuitionistic logic also rejects this law. Thus, there is no unexpected result here, since the logic makes use of four values.

Contradiction (or non-contradiction) principle: $\neg(A \wedge \neg A)$. It holds for Boolean values, but results in *ii* for the extra values.

Furthermore, all the following properties hold for any input values:

Double negation: $A \leftrightarrow (\neg \neg A)$.

Contraposition: $(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$.

De Morgan laws: $\neg (A \vee B) \leftrightarrow (\neg A \wedge \neg B)$. $\neg (A \wedge B) \leftrightarrow (\neg A \vee \neg B)$.

Modus ponens: $(A \wedge (A \rightarrow B)) \rightarrow B$.

More properties:

$(A \wedge B) \rightarrow (A \vee B)$.

$A \rightarrow (B \rightarrow A)$.

$(A \rightarrow (B \rightarrow A)) \rightarrow A$.

Commutativity with respect to the implication: $(A \rightarrow (B \rightarrow C)) \rightarrow (B \rightarrow (A \rightarrow C))$.

More properties:

$(C \rightarrow A) \rightarrow ((B \rightarrow C) \rightarrow (B \rightarrow A))$.

$(C \rightarrow A) \rightarrow ((A \rightarrow B) \rightarrow (C \rightarrow B))$.

$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$.

Distributive and-or: $((A \wedge B) \vee C) \leftrightarrow ((A \vee C) \wedge (B \vee C))$.

Distributive or-and: $((A \vee B) \wedge C) \leftrightarrow ((A \wedge C) \vee (B \wedge C))$.

Transitive: $((A \rightarrow B) \wedge (B \rightarrow C)) \rightarrow (A \rightarrow C)$.

6. CONCLUSIONS

The four-valued logic introduced in this paper is more useful than Belnap's and da Costa's ones.

REFERENCES

[1] Łukasiewicz, Jan (1920) On three valued-logic. in Borkowski, L. (editor) Selected Works by Jan Łukasiewicz, (1970) North–Holland, Amsterdam, pp87-88.

[2] Ackermann, Robert (1967) An introduction to many-valued logics. Routledge & Kegan Paul Ltd, London, and Dover Publications Inc, New York.

[3] da Costa, Newton C. A. et al (1999) Lógica Paraconsistente Aplicada. Atlas, São Paulo, Brazil, 214 pages.

[4] Belnap Jr, Nuel, (1975) A useful four-valued logic. In J. Michael Dunn and George Epstein, editors, Proceedings of the Fifth International Symposium on Multiple-Valued Logic, Modern Uses of Multiple-Valued Logic. Indiana University, D. Reidel Publishing Company, pp8-37.

[5] Ferreira, Ulisses (2000) uu for programming languages. ACM SIGPLAN Notices, 35(8): pp20-30.

[6] Ferreira, Ulisses (2004) A five-valued logic and a system. Journal of Computer Science and Technology, 4(3): pp134-140, October.

[7] Ferreira, Ulisses (2004) Uncertainty and a 7-valued logic. In Pradip Peter Dey, Mohammad N. Amin, and Thomas M. Gatton, editors, Proceedings of The 2nd International Conference on Computer Science and its Applications, National University, San Diego, CA, USA, June, pp170-173.

[8] Ferreira, J. Ulisses (2017) A Note on Gödel's Theorem, International Journal of Computer Science and Information Technology, Vol. 9, No. 2, pp69-76.

AUTHOR

The present author studied as a Master student at the Universidade Federal da Paraíba in Campina Grande, Brazil, and as a postgraduate student in the Department of Computer Science at the University of Edinburgh, and did some further research work at Trinity College in Dublin from 1998 until 2001.