STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

DR 34

The Wang Algorithm for the Propositional Calculus

The Wang Algorithm for the propositional Calculus.

The Wang Algorithm is a method of deciding whether or not a formula in the propositional calculus is a theorem. The reader will need to know something about the propositional calculus in order to understand this discussion.

We use the five logical constants $\sim$ (not), $\wedge$ (and), V (or), $\supset$ (implication), $\equiv$ (equivalent), with their usual interpretations.

A propositional letter P, Q, R, M or N, et cetera, is a formula (and an "atomic formula"). If ϕ, ψ are formulae- then $\sim \phi$, $\phi \wedge \psi$, ϕ V ψ, $\phi \supset \psi$, $\phi \equiv \psi$ are formulae. If π, ρ are strings of formulae (each, in particular, might be an empty string or a single formula) and ϕ is a formula, then π, ϕ, ρ is a string and $\pi \rightarrow \rho$ is a sequent which, intuitively speaking, is true if and only if either some formula in the string π (the "antecedent") is false or some formula in the string ρ (the "consequent") is true, i.e., the conjuction of all formulae in the antecedent implies the disjunction of all formulae in the consequent.

There are eleven rules of derivation. An initial rule states that a sequent with only atomic formulae (proposition letters) is a theorem if and only if a same formula occurs on both sides of the arrow. There are two rules for each of the five truth functions- one introducing it into the antecedent, one introducing it into the consequent. One need only reflect on the intuitive meaning of the truth functions and the arrow sign to be convinced that these rules are indeed correct. In Wang, Hao "Toward Mechanical Mathematics", IBM. I. Res. Develop., Vol. 4, No. 1. Januari 1960 is a proof given of their completeness, i.e., all intuitively valid sequents are provable, and of their consistency, i.e., all provable sequents are intuitively valid.

P1. Initial rule: if λ, ζ are strings of atomic formulae, then $\lambda \rightarrow \zeta$ is a theorem if some atomic formula occurs on both sides of the arrow.

In the ten rules listed below, λ and ζ are always strings (possibly empty) of atomic formulae. As a proof procedure in the usual sense, each proof begins with a finite set of cases P1 and continues with successive consequences obtained by the other rules.

The rules are so designed that given any sequent, we can find the first logical connective, and apply the appropriate rule to eliminate it, thereby resulting in one or two premises which, taken together, are equivalent to the conclusion. This process can be repeated until we reach a finite set of sequents with atomic formulae only. Each connective-free sequent can then be tested for being a theorem or not, by the initial rule. If all of them are theorems, then the original sequent is a theorem and we obtain a proof; otherwise we get a ccunterexample and a disproof. Some simple samples will make this clear.

For example, given any theorem of "Principia," we can automatically prefix an arrow to it and apply the rules to look for a proof. When the main connective is ⊃, it is simpler, though not necessary, to replace the main connective by an arrow and proceed. For example:

*2.45. : ∿ (PVQ).⊃. ∿ P,

*4.21. : ∿ P Λ ∿ Q.⊃. P ≡ Q

can be rewritten and proved as follows:

*2.45. ∿ (PVQ) → ∿ P

(1) → ∿ P, PVQ

(2) P → PVQ

(3) P → P, Q

VALID

*4.21. → P Λ ∿ Q.⊃. P ≡ Q

(1) ∿ P Λ ∿ Q → P ≡ Q

(2) ∿ P, ∿ Q → P ≡ Q

(3) ∿ Q → P ≡ Q,P

(4) → P ≡ Q,P,Q

(5) P → Q,P,Q

VALID

(5) Q → P,P,Q

VALID

P2a. Rule → ∿: If ϕ, $\zeta \rightarrow \lambda, \rho$, then $\zeta \rightarrow \lambda$, ∿ ϕ, ρ.

P2b. Rule ∿ →: If λ, $\rho \rightarrow \pi, \phi$, then λ, ∿ ϕ, $\rho \rightarrow \pi$.

P3a. Rule → Λ: If $\zeta \to \lambda, \phi, \rho$ and $\zeta \to \lambda, \psi, \rho$, then $\zeta \to \lambda, \phi \Lambda \psi, \rho$.

P3b. Rule Λ →: If $\lambda, \phi, \psi, \rho \to \pi$, then $\lambda, \phi \Lambda \psi, \rho \to \pi$.

P4a. Rule → V: If $\zeta \to \lambda, \phi, \psi, \rho$, then $\zeta \to \lambda, \phi V \psi, \rho$.

P4b. Rule V →: If $\lambda, \phi, \rho \to \pi$ and $\lambda, \psi, \rho \to \pi$, then $\lambda, \phi V \psi, \rho \to \pi$.

P5a. Rule → ⊃: If $\zeta, \phi \to \lambda, \psi, \rho$, then $\zeta \to \lambda, \phi \supset \psi, \rho$.

P5b. Rule ⊃ →: If $\lambda, \psi, \rho \to \pi$ and $\lambda, \rho \to \pi, \phi$ then $\lambda, \phi \supset \psi, \rho \to \pi$.

P6a. Rule → ≡: If $\phi, \zeta \to \lambda, \rho$ and $\psi, \zeta \to \lambda, \phi, \rho$, then $\zeta \to \lambda, \phi \equiv \psi, \rho$.

P6b. Rule ≡ →: If $\phi, \psi, \lambda, \rho \to \pi$ and $\lambda, \rho \to \pi, \phi, \psi$, then $\lambda, \phi \equiv \psi, \rho \to \pi$."

(2) The LISP Program. We define a function theorem{s}whose value is truth or falsity according to whether the sequent s is theorem.

The sequent

s: $\phi_1, \ldots, \phi_n \to \psi_1, \ldots, \psi_m$

is represented by the S-expression

s^*:(ARROW, $(\phi_1^*, \ldots, \phi_n^*)$, $(\psi_1^*, \ldots, \psi_m^*)$)

where in case the ellipsis ... denotes missing terms, and where ϕ^* denotes the S-expression for ϕ.

Propositional formulae are represented as follows:

1. For "atomic formulae" (Wang's terminology) we use "atomic symbols" (LISP terminology).

2. The following table gives our "Cambridge Polish" way of representing propositional formulae with given main connectives.

1. $\sim \phi$	becomes	(NOT ϕ^*)
2. $\phi \Lambda \psi$	becomes	(AND $\phi^* \psi^*$)
3. $\phi V \psi$	becomes	(OR $\phi^* \psi^*$)
4. $\phi \supset \psi$	becomes	(IMPLIES $\phi^* \psi^*$)
5. $\phi \equiv \psi$	becomes	(EQUIV $\phi^* \psi^*$)

Thus the sequent

$\sim P \Lambda \sim Q \to P \equiv Q$, RVS

is represented by

(ARROW ((AND (NOT P) (NOT Q))) ((EQUIV P Q) (OR R S)))

The S-function theorem {s} is given in terms of auxiliary functions as follows:

```
theorem{s} = thl {NIL; NIL; cadr{s}; caddr{s}}

thl{al; a2; a; c} = {null{a}→th2{al; a2; NIL; NIL; c}; T→
                     member{car{a};c}V{atom{car{a}} →
                     thl{{member{car{a}; al}→al; T→ cons{car{
                     a};al}}; a2; cdr{a}; c}; T →thl{al;{
                     member{car{a}; a2} →a2; T→ cons{
                     car{a}; a2}} ; cdr{a}; c}}}

th2{al;a2;c1;c2;c} = {null{c}→th{al;a2;c1;c2};atom{
                     car{c}}→ th2{al;a2;{member{car{
                     c}; c1}→c1;T→ cons{car{c};c1}};
                     c2;cdr{c}};T→ th2{al;a2;c1;{
                     member{car{c};c2}→ c2;T→ cons{
                     car{c};c2}};cdr{c}}}

th{al;a2;c1;c2}= {null{a2}→~null{c2}Λthr{car{c2};
                 al;a2;c1;cdr{c2}};T→ thl{car{a2};
                 al;cdr{a2};c1;c2}}
```

th is the main predicate through which all the recursions take place. theorem, thl and th2 break up and sort the information in the sequent for the benefit of th. The four arguments pf th are:

a1: atomic formulae on left side of arrow

a2: other formulae on left side of arrow

c1: atomic formulae on right side of arrow

c2: other formulae on right side of arrow

The atomic formulae are kept separate from the others to make faster the detection of the occurence of formula on both sides of the arrow and the finding of the next formula to reduce. Each use of th represents one reduction according to one of the 10 rules. The formula to be reduced is chosen from the left side of the arrow if possible. According to whether the formula to be reduced is on the left or right we use thl or thr.

We have

```
th&{u;a1;a2;c1;c2} = {
    car{u} = NOT→ th1r{cadr{u};a1;a2;c1;c2};
    car{u} = AND → th2&{cdr{u} ;a1;a2;c1;c2} ;
    car{u} = OR → th1&{cadr{u} ;a1;a2;c1;c2}∧ th1&{
              caddr{u};a1;a2;c1;c2};
    car{u} = IMPLIES → th1&{caddr{u} ; a1;a2;c1;c2}∧ th1r{
              cadr{u};a1;a2;c1;c2};
    car{u} = EQUIV→ th2&{cdr{u} ;a1;a2;c1;c2}∧th2r{
              cdr{u} ; a1;a2;c1;c2};
    T → error{list{THL;u;a1;a2;c1;c2}}}

thr{u;a1;a2;c1;c2} = {
    car{u} = NOT → th1&{cadr{u};a1;a2;c1;c2};
    car{u} = AND → th1r{cadr{u};a1;a2;c1;c2}∧th1r{
              caddr{u};a1;a2;c1;c2};
    car{u} = OR → th2r{cdr{u}; a1;a2;c1;c2}
    car{u} = IMPLIES → th1 1{cadr{u};caddr{u};a1;a2;c1;c2};
    car{u} = EQUIV → th1 1{cadr{u}; caddr{u};a1;a2;c1;c2}∧
              th1 1{caddr{u}; cadr{u};a1;a2;c1;c2};
    T → error{THR;u;a1;a2;c1;c2}}}
```

The functions th1&, th1r, th2&, th2r, :th1 1 distribute the parts of the reduced formula to the approciate places in the reduced sequent.

These functions are

```
th1&{v;a1;a2;c1;c2} = {atom{v}→ member{v;c1}V
    th{cons{v;a1}; a2;c1;c2};T → member{v;c2}V
    th{a1;cons{v;a2};c1;c2}}

th1r{v;a1;a2;c1;c2} = {atom{v} → member{v;a1} V
    th{a1;a2;cons{v;c1};c2} ;T → member{v;a2}V
    th{a1;a2;c1;cons{v;c2 }}}
```

```
th2ℓ{v;a1;a2;c1;c2} = {atom{car{v}} → {member{car{v};c1}V
    th1ℓcadr{v};cons{car{v};a1} ;a2;c1;c2 ;T → member{
        car{v};c2}V
    th1ℓ{cadr{v};a1;cons{car{v};a2};c1;c2}}

th2r{v;a1;a2;c1;c2} = {atom{car{v}} → member{car{v;a1}V
    th1r{cadr{v};a1;a2;cons{car{v};c1};c2};T → member{
        car{v};a2}V
    th1r{cadr{v};a1;a2;c1;cons{car{v};c2}}}

th1 1{v1;v2a1;a2;c1;c2}  = {atom{v1} → member{v1;c1}V
    th1r{v2;cons{v1;a1};a2;c1;c2};T → member{v1;c2}V
    th1r{v2;a1;cons{v1;a2};c1;c2}}
```

Finally the function member is defined by

```
member{x;u} = ∿ null{u}Λ{equal{x;car{u}}Vmember{x;cdr{u}}}
```

Transcription rules from LISP to ALGOL:

1. A conditional expression has the following form:

$\{p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots; p_n \rightarrow e_n\}$

where each p_i is an expression whose value may be truth or falsity and each e_i is any expression.

The meaning of a conditional expression is:

If p_1 is true then the value of e_1 is the value of the entire expression.

If p_1 is false then if p_2 is true the value of e_2 is the value of the entire expression.

The p_i are searched from left to right until the first true one is found Then the corresponding e_i is selected.

If none of the p_i are true, then the value of the entire expression is undefined.

A conditional expression as above has in the LISP-ALGOL system the following transcription:

$\underline{\text{if}}\ p_1\ \underline{\text{then}}\ e_1\ \underline{\text{else if}}\ p_2\ \underline{\text{then}}\ e_2\ \underline{\text{else}} \cdots \underline{\text{if}}\ p_{n-1}\ \underline{\text{then}}\ e_{n-1}\ \underline{\text{else}}\ e_n$

Consequently the system does not accept conditional expressions with all the p_i false. Side effects of the last predicate p_n are ignored.

2. $p \wedge q = \{p \rightarrow q; T \rightarrow F\}$

So in ALGOL: if p then q else false

3. $p \vee q = \{p \rightarrow T; T \rightarrow q\}$

In ALGOL: if p then true else q

4. $\sim p = \{p \rightarrow F; T \rightarrow T\}$

```
begin   comment LISP-programma Hao-Wang in ALGOL, van de Laarschot;
        integer NOT, AND, OR, IMPLIES, EQUIV, NIL, a, b;

        Boolean procedure null(x); value x; integer x;
                null:= if atom(x) then eq(x,NIL) else false;

Boolean procedure equal(x,y); value x, y; integer x, y;
        equal:= if atom(x) ∧ atom(y) then eq(x,y) else
                if ¬ atom(x) ∧ ¬ atom(y) then
                equal(car(x),car(y)) ∧ equal(cdr(x),cdr(y))
                else false;

Boolean procedure member(x,y); value x, y; integer x,y;
                member:= if null(y) then false else if
                        equal(x,car(y)) then true else
                        member(x,cdr(y));

Boolean procedure th1(u,a1,a2,c1,c2);
        value u,a1,a2,c1,c2; integer u,a1,a2,c1,c2;
        th1:= if eq(car(u),NOT) then
                th1r(car(cdr(u)),a1,a2,c1,c2) else if
        eq(car(u),AND) then th21(cdr(u),a1,a2,c1,c2)
        else if eq(car(u),OR) then
```

```
( if th11(car(cdr(u)),a1,a2,c1,c2) then
  th11(car(cdr(cdr(u))),a1,a2,c1,c2)  else
 false) else if eq(car(u),IMPLIES) then
              (if  th11(car(cdr(cdr(u))),a1,a2,c1,c2) then
                th1r(car(cdr(u)),a1,a2,c1,c2) else false)
             else if th21(cdr(u),a1,a2,c1,c2) then
              th2r(cdr(u),a1,a2,c1,c2) else  false;

Boolean procedure thr(u,a1,a2,c1,c2);
        value u,a1,a2,c1,c2; integer u,a1,a2,c1,c2;
        thr:= if eq(car(u),NOT) then
              th11 (car(cdr(u)),a1,a2,c1,c2) else if
              eq(car(u),AND) then
              (if th1r(car(cdr(u)),a1,a2,c1,c2) then
               th1r(car(cdr(cdr(u))),a1,a2,c1,c2) else false)
              else if eq(car(u),OR) then
              th2r(cdr(u),a1,a2,c1,c2) else if
              eq(car(u),IMPLIES) then
              th11(car(cdr(u)),car(cdr(cdr(u))),a1,a2,c1,c2)
              else if
              th11(car(cdr(u)),car(cdr(cdr(u))),a1,a2,c1,c2)
              then
              th11(car(cdr(cdr(u))),car(cdr(u)),a1,a2,c1,c2)
              else false;
```

```
Boolean procedure th11(v1,v2,a1,a2,c1,c2);
        value v1,v2,a1,a2,c1;c2; integer v1,v2,a1,a2,c1,c2;
        th11:= if atom(v1) then
               ( if member(v1,c1) then true else
                  th1r(v2,cons(v1,a1),a2,c1,c2))
               else if member(v1,c2) then true else
               th1r(v2,a1,cons(v1,a2),c1,c2);

Boolean procedure th2r(v,a1,a2,c1,c2);
        value v,a1,a2,c1,c2; integer v,a1,a2,c1,c2;
        th2r:= if atom(car(v)) then
               (if member (car(v),a1) then true else
                th1r(car(cdr(v)),a1,a2,cons(car(v),c1),c2))
               else if member(car(v),a2) then true else
               th1r(car(cdr(v)),a1,a2,c1,cons(car(v),c2));

Boolean procedure th21(v,a1,a2,c1,c2);
        value v,a1,a2,c1,c2; integer v,a1,a2,c1,c2;
        th21:= if atom(car(v)) then
               ( if member(car(v),c1) then true else
                 th11(car(cdr(v)),cons(car(v),a1),a2,c1,c2))
               else if member(car(v),c2) then true else
               th11(car(cdr(v)),a1,cons(car(v),a2),c1,c2);
```

```
Boolean procedure th1r(v,a1,a2,c1,c2);
        value v,a1,a2,c1,c2; integer v,a1,a2,c1,c2;
        th1r:= if atom(v) then
                ( if member(v,a1)  then true else
                  th(a1,a2,cons(v,c1),c2)) else
                  if member(v,a2) then true else
                  th(a1,a2,c1,cons(v,c2));

Boolean procedure th11(v,a1,a2,c1,c2);
        value v,a1,a2,c1,c2; integer v,a1,a2,c1,c2;
        th11:= if atom(v) then
                ( if member(v,c1) then true else
                  th(cons(v,a1),a2,c1,c2)) else
                  if member(v,c2) then true else
                  th(a1,cons(v,a2),c1,c2);

Boolean procedure th(a1,a2,c1,c2);
        value a1,a2,c1,c2; integer a1,a2,c1,c2;
        th:= if null(a2) then ( if ¬ null(c2) then
                thr(car(c2),a1,a2,c1,cdr(c2))
        else false) else th1(car(a2),a1,cdr(a2),c1,c2);
```

```
Boolean procedure th1(a1,a2,a,c);
        value a1,a2,a,c; integer a1,a2,a,c;
        th1:= if null(a) then th2(a1,a2,NIL,NIL,c) else if
              member(car(a),c) then true else if
              atom(car(a)) then th1(if member(car(a),a1)
              then a1 else cons(car(a),a1),a2,cdr(a),c)
              else  th1(a1, if member(car(a),a2) then a2
              else cons(car(a),a2),cdr(a),c);

Boolean procedure th2(a1,a2,c1,c2,c);
        value a1,a2,c1,c2,c; integer a1,a2,c1,c2,c;
        th2:= if null(c) then th(a1,a2,c1,c2) else if
              atom(car(c)) then
              th2 (a1,a2,if member(car(c),c1) then c1 else
                   cons(car(c),c1),c2,cdr(c)) else
              th2(a1,a2,c1,if member(car(c),c2) then c2 else
                  cons(car(c),c2),cdr(c));

startlisp(XEEN(32767)); call(NOT,⟨¬⟩); call(AND,⟨∧⟩);
call(OR,⟨∨⟩); call(IMPLIES,⟨⊃⟩); call(EQUIV,⟨≡⟩);
call(NIL,⟨⟩); readstring(a);  readstring(b);NLCR;
        if th1(NIL,NIL,a,b) then PRINTTEXT(⟨true⟩) else
        PRINTTEXT(⟨false⟩)
end
```

Some examples of input and output:

inputstrings		output
antecedent	consequent	
⊰P⊱'	⊰⊰¬'P⊱⊱'	false
⊰P⊱'	⊰⊰∧'P'P⊱⊱'	true
⊰P⊱'	⊰⊰∨'P'Q⊱⊱'	true
⊰⊰∨'A'⊰¬'B⊱⊱⊱'	⊰⊰⎽'⊰∧'P'Q⊱'⊰⎽'P'Q⊱⊱⊱'	true
⊰⊰∨'P'Q⊱⊱'	⊰⊰∧'P'Q⊱⊱'	false
⊰⊰∧'⊰¬'P⊱'⊰¬'Q⊱⊱⊱'	⊰⊰⎽'P'Q⊱'⊰∨'R'S⊱⊱'	true
⊰⊰P⊱'⊰Q⊱⊱'	⊰⊰Q⊱'⊰P⊱⊱'	true
⊰P⊱'	⊰⊰∨'R'S⊱⊱'	false
⊰⊰∧'P'Q⊱⊱'	⊰⊰∨'P'Q⊱⊱'	true
⊰A⊱'	⊰⊰∨'P'⊰¬'P⊱⊱⊱'	true
⊰⊰∨'P'Q⊱⊱'	⊰⊰∨'Q'P⊱⊱'	true
⊰A⊱'	⊰⊰⎽'⊰⎽'P'⊰⎽'Q'R⊱⊱'⊰⎽'Q'⊰⎽'P'R⊱⊱⊱'	true
⊰⊰⎽'P'⊰⎽'Q'R⊱⊱⊱'	⊰⊰⎽'Q'⊰⎽'P'R⊱⊱⊱'	true
⊰A⊱'	⊰⊰∨'⊰¬'P⊱'P⊱⊱'	true
⊰A⊱'	⊰⊰⎽'⊰⎽'P'Q⊱'⊰⎽'⊰¬'Q⊱'⊰¬'P⊱⊱⊱'	true
⊰⊰⎽'P'Q⊱⊱'	⊰⊰⎽'⊰¬'P⊱'⊰¬'Q⊱⊱⊱'	false
⊰⊰¬'⊰∧'P'Q⊱⊱⊱'	⊰⊰⎽'P'⊰¬'Q⊱⊱⊱'	true
⊰⊰∨'P'Q⊱⊱'	⊰⊰⎽'⊰¬'Q⊱'P⊱⊱'	true
⊰A⊱'	⊰⊰⎽'⊰∨'P'Q⊱'⊰⎽'⊰¬'Q⊱'P⊱⊱⊱'	true
⊰⊰∧'⊰¬'P⊱'⊰¬'Q⊱⊱⊱'	⊰⊰∨'R'S⊱'⊰⎽'P'Q⊱⊱'	true
⊰⊰∨'P'Q⊱⊱'	⊰⊰∧'P'⊰¬'Q⊱⊱⊱'	false

```
⊬⊮⊼'⊬⊐'P⊁'Q⊁⊁'                    false

      ⊬P⊁'                          true

⊬⊬⊐'⊬⊐'⊮⊼'P'⊬⊐'Q⊁⊁'R⊁'⊬⊐'P'R⊁⊁'  true

⊬⊬⊐'⊮⊼'⊬⊐'P'Q⊁'⊮⊼'P'R⊁⊁'⊮⊼'Q'R⊁⊁'  true

            ⊬⊮⊼'Q'R⊁⊁'              true

⊬⊬⊐'⊬⊐'P'⊮⊼'Q'⊬⊐'Q⊁⊁'⊬⊐'P'R⊁⊁'  true

⊬⊬⊐'⊬⊐'⊮⊼'P'Q⊁⊁'⊬⊐'P'⊬⊐'Q⊁⊁'  true

⊬⊮⊼'P'Q⊁⊁'                        true

⊬⊬⊼'P'Q⊁⊁'⊮⊼'P'⊬⊐'Q⊁⊁'⊮⊼'P⊁'Q⊁⊁'  true

⊬⊮⊼'P'Q⊁⊁'                        true

⊬⊬⊐'⊬⊐'P⊁'P⊁⊁'                    true

⊬⊬⊐'⊬⊐'P'R⊁⊁'                     true
```

```
⊬⊮⊼'P'Q⊁⊁'

⊬⊬⊐'⊬⊐'P⊁'P⊁⊁'

⊬⊬⊐'⊬⊐'P'⊮⊼'Q⊁⊁'

⊬A⊁'

⊬⊬⊐'⊬⊐'P'R⊁⊁'

⊬A⊁'

⊬A⊁'

⊬⊬⊼'P'⊮⊼'Q⊁⊁'⊬⊐'P'⊬⊐'Q⊁⊁'⊮⊼'P⊁'Q⊁⊁'

⊬⊮⊼'P'Q⊁⊁'

⊬⊬⊐'⊬⊐'P⊁'P⊁⊁'

⊬A⊁'

⊬⊬⊐'⊬⊐'P'⊮⊼'Q⊁⊁'
```

P. v.d. Laarschot.