The Generator
Vol 1. No 1.
March MMIV.

Unification in Propositional Calculus.

Kazimir Majorinc

Propositional calculus is perhaps the simplest and the best known example of a formal theory[1]. The formulas of propositional calculus are either (1) *propositional variables* (for example A, B, C, ...) or (2) constructed from other formulas using unary (~) or binary connectives (|, &, >, ...). If infix notation for binary connectives is used, these formulas are similar to the usual algebraic formulas: for example (~A), (A>B), (A>(~(A>B))) are formulas, while A~B>B is not.

Somehow, surprisingly, a standard set of axioms for propositional calculus has not been established; one can hardly find two books that use identical sets of axioms.

There is more agreement about rules of inference. Two of the most frequently used rules are (1) *substitution*: if F is a theorem, X_1, ..., X_n are some of the variables occurring in F and G_1, ..., G_n are formulas without occurrences of X_1, ..., X_n then the formula F' obtained from F by simultaneous substitution of all occurrences of X_1, ..., X_n with G_1, ..., G_n respectively is also the theorem and (2) *modus ponens:* if formulas $(F>G)$ and F are theorems, then G is also theorem. Additional rules of inference are not necessary. Axioms are, by definition, also theorems of propositional calculus.

A logician usually defines propositional calculus syntactically, because syntax is finite and even - visible, and as such it raises less doubts than any semantics[2]. However, the usual intention is to finally add semantics to the defined syntax. Typically, variables are interpreted as statements of the natural language (including mathematical extensions) and connectives ~, |, & and > as logical operators "not", "or", "and" and "implies" respectively. With proper choices of axioms and inference rules, propositional calculus is *complete*, i.e. all *tautologies* (i.e. statements that are true no matter which natural language statements are represented by variables) and only tautologies are theorems of propositional calculus.

Although the tautology concept might seem trivial and useless, it is not. For example, if we know that $(F_1>F_2)$ is a tautology, we also know that F_2 is true whenever F_1 is true; certainly, such a conclusion is not trivial for every possible interpretation of F_1 and F_2 in natural language.

Definition of the propositional calculus is constructive; in principle, one can make a program that derives all theorems of propositional calculus. However, after half a century of research, computers have only established a marginal role in the development of the mathematical knowledge. Furthermore, difficulties in designing programs that match human capabilities in games such as chess, or especially *go* are not encouraging as it is probable that mathematics is more complicated than these games.

---

[1] Any introductory text in mathematical logic will contain an extensive survey of important results, for example any edition of **E. Mendelson,** Introduction to Mathematical Logic.

[2] For example, some logicians do not accept that double negation implies affirmation. They, however, find formula as ((~(~A))>A) acceptable if it is defined as string of characters, without meaning.

Theorems, as found in books are both *true* and *interesting* mathematical statements. In attempts to automate their derivation[1], two general approaches are used. In the first approach, one starts from an interesting statement and tries to determine whether it is true. In the second approach, one starts from statements known to be true and tries to find interesting consequences. These two approaches are called *automated theorem proving* and *automated theorem finding*[2]. (far less researched with exception of the works of **S. N. Vassilyev**[3].)

In practice, the problem with automated theorem finding is always the same: naive algorithms for deriving tautologies generate many obviously trivial, weak or other less than interesting theorems. If interesting theorems are derived at all, it is not known how they could be identified and isolated from myriads of others also generated in the process.

For example, substitution can be applied to any formula and infinitely many formulas can be derived from it. Unfortunately, all these derived formulas are longer and weaker than the premise. These are weaker in the usual mathematical sense that can be easily recognized but is hard to formalize.

Modus ponens is different: the consequence is shorter and stronger than the longer one of the two premises. Unfortunately, modus ponens can rarely be applied; almost certainly it cannot be applied on two randomly chosen theorems of propositional calculus.

This difference suggests that integration of these two rules in some combined rule can both reduce combinatorial explosion caused by substitution and increase the frequency of *successful* application of modus ponens in the process of the development of propositional formulas. One possible *combined rule* is (3) for two theorems F and $(G>H)$, if there are substitutions s and t such that $s(F)=t(G)$ then $t(H)$ is also a theorem.

The combined rule is not trivial any more. The essential part of the problem is determining whether for given formulas F and G there exist substitutions s and t such that $s(F)=t(G)$. That problem also occurs in other contexts and the commonly used name for it is *unification*[4] of formulas.

An algorithm for unification of two propositional formulas is easily implemented in a Unicon program of about hundred and fifty lines.

```
link strings
link sets

$define NL "\n"
$define LINE repl("=", 20)
$define TRUE 1
$define FALSE 0
```

---

[1] Good survey article is **M. Beeson,** THE MECHANIZATION OF MATHEMATICS in **C. Teuscher,** (ed.) ***Alan Turing****: Life and Legacy of a Great Thinker,* Springer-Verlag, Berlin, **2003**.

[2] **L. Wos,** THE PROBLEM OF AUTOMATED THEOREM FINDING. *Journal of Automated Reasoning,* Vol. **10**(1), **1993**, pp. 137-8.

[3] Probably the best review of his work is **S. N. Vassilyev,** MACHINE SYNTHESIS OF MATHEMATICAL THEOREMS, *Journal of Logic Programming*, Vol 9, **1990**, pp. 235-66.

[4] Extensive review can be found in **F. Baader**, **W. Snyder**, UNIFICATION THEORY, Chapter 8, pp. 439-526 in **A. Robinson**, **A. Voronkov** (ed.), *Handbook of Automated Deduction,* Elsevier/MIT Press, **2001**.

```
$define xxxxxx1 (
$define init_to :=:temp_init_to):=(if \temp_init_to then temp_init_to else
$define xxxxxx2 )

procedure main()
   every k:=1 to 4 do
   {F:= [  ["(A>((B>(C>B))>D))","((a>(b>c))>((a>b)>(a>c)))"],
        ["(A>(~A))","((~B)>B)"],
        ["(A>(~B))","(B>(~A))"],
        ["(A>(~B))","(B>(~A))"]
       ][k]
    Fca:=[[TRUE, TRUE], [TRUE, TRUE], [TRUE, FALSE], [TRUE, TRUE]][k]
    write(LINE, NL, "Unification of: ")
    every i:=idx(F) do write(F[i], ", changes allowed: ", Fca[i])
    write("Unification succeeded: ", unified( F, Fca ).formula)
   }
 end
```

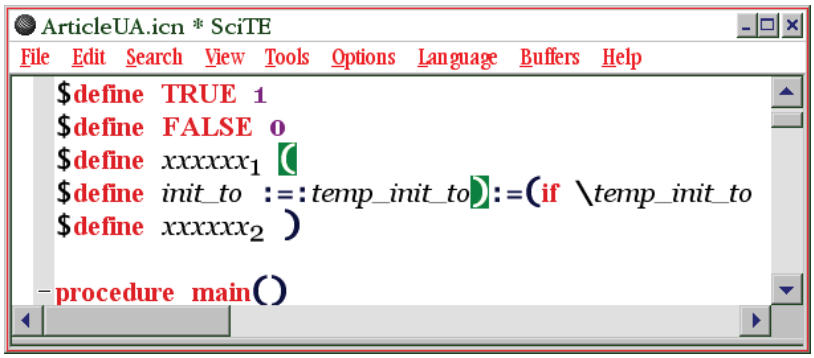
The program links to standard libraries "strings" and "sets." A few simple macros are defined. Only the infix macro operator *init_to* deserves some comment. The expression ((*x init_to* expr)) is equivalent to longer expressions like {/*x*:=expr; *x*} or {**if** /*x* **then** *x*:=expr **else** *x*} that allow initialization of variables in the same place they are used in loops. The macro *init_to* is usually slower than initialization outside of the loop. In some expressions, like **every** ((*x init_to* 0))+:=1 **to** *n*, one can replace ((*x init_to* 0)) with simple (*x*:=0). Also, one execution of the macro in some procedure must be completed before another execution is started. Hence, nested expressions like ((*x init_to* ((*y init_to* 0)))) do not work correctly.[1] However, we believe that replacement of the frequently occurring idioms or patterns in the programs with simple, non-redundant syntactical constructs reveals the logic of the programming itself, so the price appears to be acceptable.[2]

The seemingly strange macros *xxxxxx1* and *xxxxxx2* have only one role: to balance the parenthesis left open by *init_to* and prevent errors in text editors with integrated parentheses matching; the definition of the macro *init_to* uses parentheses in an unusual way; for an excellent example see **N. Hodgons's** SciTE[3].

[1] It seems to be one of the most frequent problems with macros..

[2] **C. Evans** implemented a more powerful macro system and special syntax (x :$ expr) for {/x:=expr; x} in his private build of Unicon.

[3] <http://www.scintilla.org/SciTE.html>.



For testing and demonstration purposes, four pairs of formulas are stored in the list F and passed as an argument to the procedure *unified*. That procedure returns a record consisting of (1) the formula resulting from unification and (2) a list of all performed substitutions.

The procedure *unified* accepts another argument, *Fca*, a list of Boolean values. For clarity of intention the macros FALSE and TRUE are used respectively. In this implementation of *unified*, only formulas $F[i]$ such that $Fca[i]$=TRUE can be changed. If unification succeeds and $Fca[i]$ was FALSE, then $F[i]$ can be obtained from $F[3\text{-}i]$ by substitution. Less formally, $F[i]$ is a special and weaker case of $F[3\text{-}i]$.

Some procedures used in the program can be useful in a more general context. They are copied from other programs or generalized and extracted elements of the early working versions of this program.

```
procedure is_true(B)
  if B==TRUE then return TRUE
  end

procedure card(predicate, X)
  every ((result init_to 0))+:=( predicate(!X) & 1 )
  return result
  end

procedure card_nulls(X)
  every ((result init_to 0)) +:= ((/!X) & 1 )
  return result
  end

procedure card_columns(LL)
  every ((result init_to *?LL)) <:= *!LL
  return result
  end

procedure idx(L)
  suspend 1 to *L
  end
```

```
procedure jdx(LL)
  suspend 1 to card_columns(LL)
  end

procedure column(LL, j)
  if /j then suspend column(LL, jdx(LL))
       else  {  every L:=!LL do
                  put( ((C init_to [])) , if j <= *L then L[j] else &null)
                  return C
                }
  end


procedure projection(XX, index)
  if  type(XX)=="list"
    then {  result:=[];
              every X:=!XX do put(result, X[index])
              return result
            }
  end

procedure is_simple_type(x)
  if type(x)==("real"|"integer"|"string") then return x
  end

procedure generalized_application(p, L)
  every put(result:=[], p(!L))
  return result
  end

procedure equal_by_value(X)
  if not different_by_value(X) then return X
  end


procedure different_by_value(L)
  S:=set(L)
  if member(S, "&equal") & member(S, "&different")
     then error("Ambivalent different/equal_by_value.")
  if member(S, "&equal") then fail
  if member(S, "&different") then return L
    case card_nulls(L) of { 1: return L; 2: fail }
    return case card(is_simple_type, L) of
    {  1: L
       2: if L[1] ~=== L[2] then L else &fail
       0: if different_by_value(column(L)) then L else &fail
    }
  end
```

The predicates *is_true* and *is_false* allow convenient combination of Boolean and success/ failure program control flow.

The generator *jdx*(*LL*) accepts a list of lists as an argument; if understood as two-dimensional array, *jdx* generates indexes of its columns. The procedure *column*(*LL*, *j*) returns *j*-th column of such an array, i.e. list [*LL*[1][*j*], ..., *LL*[**LL*][*j*]]; if the second argument is omitted, it generates all columns of *LL*. Expressions symmetrical to *jdx*(*LL*), *column*(*LL*, *j*) and *column*(*LL*) are 1 **to** **L*, *LL*[*j*] and !*LL* respectively. Syntactical symmetry can be achieved by implementation of procedures *idx*(*LL*)[1] and *row*(*LL*, *j*).

The procedure *projection*(*X*, *index*) is a generalization of the procedure *column*(*LL*, *j*); it accepts a list of tables as an argument and *index* can be any key in the table. Further generalization can be useful.

The procedure *generalized_application*(*p*, *L*) returns the list [*p*(*L*[1]), ..., *p*(*L*[**L*])]. It is similar to **R. Griswold's** *apply* in the Icon Program Library, file "`apply.icn`". Further generalization can be useful.

A few procedures with names containing the prefix *card*[2] count elements of the structures satisfying given criteria.

Unicon's built in operator === and its negation ~=== compare equality of the two structures "*by reference.*" Although there are few similarities with set-theoretical equality, === does not satisfy the axiom of extensionality[3]. For example, {1, 2}={1, 2} is true in set theory, while its Unicon equivalent **set**([1, 2]) === **set**([1, 2]) does not necessarily succeed[4].

Design and implementation of a relation more similar to set theory equality has been addressed in the past[5].

The procedures *different_by_value* and *equal_by_value* presented here are more limited than **J. P. de Ruiter's** procedure. However, they have one useful additional property. Pseudo- keywords "`&equal`" and "`&different`", are defined as *equal_by_value* and *different_by_value* to any value. Comparison between "`&equal`" and "`&different`" is not defined and will result, in a runtime error if attempted.

```
procedure is_variable(F)
  if find(F, &letters) then return F
  end

record character_index_level_type(character, index, level)

procedure character_index_level(F)
  suspend character_index_level_type(
```

---

[1] The function **key** is equivalent to *idx*.

[2] The name of the procedure is inspired by the set-theoretical concept of the cardinal number.

[3] Sets are uniquely defined by their members, i.e. $(\forall x)(\forall y)(((\forall z)(\, z{\in}x \leftrightarrow z{\in}y)) \leftrightarrow (x{=}y))$

[4] Actually, **set**([1, 2]) === **set**([1, 2]) never succeeds in Unicon.

[5] **R. Griswold's** procedure *equiv* (`equiv.icn`, I.P.L.) and **J. P. de Ruiter's** procedure *same_value* (`mset.icn`, I.P.L.) should be mentioned.

```
    cF:=!F,
    ((i init_to 0)) +:= 1,
    ((lev init_to 0)) +:= case cF of { "(": 1; ")": -1; default: 0}
    )
  end

procedure main_connective(F)
  return equal_by_value([ character_index_level(F), ["~"|">", "&equal", 1]])[1]
  end


procedure analysed_formula(F)
  T:=table()
  if is_variable(F)
    then T["variable"]:=F
    else{m:=main_connective(F)
        T["connective"]:=m.character
        T["left"]:=F[2:m.index]
        T["right"]:=F[m.index + 1: -1]
       }
  return T
end
```

The predicate *is_variable* allows all lowercase and uppercase letters as propositional variables.

The procedure *character_index_level*(*F*) generates records containing successive individual characters of the formula *F*, the position index of the character in the formula and the number of opened and unclosed parentheses before that position. Note that **suspend**, aside from its primary role, resumes all generators like **every**.

Perhaps the most elegant procedure in the whole program, *main_connective*(*F*) returns a connective ("~" or ">") enclosed in exactly one pair of parenthesis in the formula *F* and its position in that formula.

The procedure *analysed_formula* accepts a formula as an argument and returns a table containing the main connective and both the left and right subformulas of a given formula. If the main connective is unary, i.e. "~", the left subformula is by the definition empty string.

Finally, we approach the most specific parts of the program.

```
record substitution(variable, formula)

procedure forced_substitution(F, Fca)
  if is_variable(!F)
    then if i:=idx(F) & is_variable(F[i]) & is_true(Fca[i])
      then return substitution(F[i], F[3-i] )
      else {write("No substitution: formulas differ in variable "||
                  "but substitution is not allowed.")
            fail
           }

  AF:=generalized_application(analysed_formula, F)
```

```
  "~==" !( D:=projection(AF, j:=![“connective”, “left”, “right”]))
  if j==“connective”
   then write(“No substitution: different main connectives.”)
   else return forced_substitution(D, Fca)
  end

procedure substitute(F, Fca, s)
  every is_true(Fca[i:=idx(F)])
     do F[i]:=replace(F[i], s.variable, s.formula)
  if find(s.variable, !F)
   then write(“Substitution failed: ”, s.variable, “ cannot be eliminated.”)
   else return s
  end

record unified_type(formula, substitution)

procedure unified(F, Fca)
  while different_by_value( F ) do
   if not(s:=forced_substitution(F, Fca) &
        write( LINE, NL, “Substitution ”, s.formula,
              “ for ”, s.variable, “ suggested.”
             ) &
        substitute(F, Fca, s) &
        write(“Substitution succeeded.”, NL, F[1], NL, F[2]) &
        put( ((applied_substitutions init_to [])), s)
       )
   then fail
  return unified_type(?F, applied_substitutions)
  end
```

The procedure *unified* contains a loop that is repeated as long as formulas $F[1]$ and $F[2]$ are different. In the loop two elementary operations are performed, (1) searching for substitutions that need to be performed and (2) performing the substitutions. If any of these two fail, unification also fails. Those two operations are delegated to the procedures *forced_substitution* and *substitute*.

The prefix “forced” in *forced_substitution* suggests that a found substitution has to be applied; otherwise, it would be impossible to unify two formulas. The *forced_substitution* first searches for the difference between two formulas, translating them into the form of a tree 'on the fly' and then tries to match these trees. There are a few different cases, dependent on the difference between formulas $F[1]$ and $F[2]$.

In the simplest case exactly one of the formulas is a propositional variable; let us denote it with $F[i]$. If changing $F[i]$ is allowed then substitution of $F[3\text{-}i]$ for $F[i]$ is necessary for unification. If changes to the formula $F[i]$ are not allowed, then $F[1]$ and $F[2]$ cannot be unified.

If both formulas are variables, then either of the substitutions $F[1]$ for $F[2]$ or $F[2]$ for $F[1]$ can be chosen.

If neither one of the formulas in F is variable and they differ in the main connective then no substitution can unify them.

Finally, if both formulas in F are complex, (i.e. not variables) and have the same main connectives and differ in at least one of the corresponding subformulas then further searching is performed recursively.

Once found, substitution can be performed easily. The procedure **replace** from "`strings.icn`" in the Icon Program Library can be used for formulas in the form of the string.

Under some circumstances substitution fails, i.e. when a substituted variable still occurs in some part of the formula F. This can happen if (1) the formula to be substituted for a variable contains the same variable[1]; for example, if (∼B) is substituted for B; or (2) when a substituted variable occurs in a formula where changes are not allowed. If substitution fails, again, unification of the formulas is impossible.

After the formulas are unified it does not matter which one is returned as result of the unification; so a random choice is returned. Output produced by the program is relatively readable.

```
==========================
Unification of:
(A>((B>(C>B))>D)), changes allowed: 1
((a>(b>c))>((a>b)>(a>c))), changes allowed: 1
==========================
Substitution (a>(b>c)) for A suggested.
Substitution succeeded.
((a>(b>c))>((B>(C>B))>D))
((a>(b>c))>((a>b)>(a>c)))
==========================
Substitution a for B suggested.
Substitution succeeded.
((a>(b>c))>((a>(C>a))>D))
((a>(b>c))>((a>b)>(a>c)))
==========================
Substitution (C>a) for b suggested.
Substitution succeeded.
((a>((C>a)>c))>((a>(C>a))>D))
((a>((C>a)>c))>((a>(C>a))>(a>c)))
==========================
Substitution (a>c) for D suggested.
Substitution succeeded.
((a>((C>a)>c))>((a>(C>a))>(a>c)))
((a>((C>a)>c))>((a>(C>a))>(a>c)))
Unification succeeded: ((a>((C>a)>c))>((a>(C>a))>(a>c)))
==========================
Unification of:
(A>(∼A)), changes allowed: 1
((∼B)>B), changes allowed: 1
==========================
```

[1] The occur-check test is frequently discussed in the context of Prolog. Most implementations do not perform occur-check.

Substitution (∼B) for A suggested.
Substitution succeeded.
((∼B)>(∼(∼B)))
((∼B)>B)
==========
Substitution (∼(∼B)) for B suggested.
Substitution failed: B cannot be eliminated.
==========
Unification of:
(A>(∼B)), changes allowed: 1
(B>(∼A)), changes allowed: 0
==========
Substitution B for A suggested.
Substitution failed: A cannot be eliminated.
==========
Unification of:
(A>(∼B)), changes allowed: 1
(B>(∼A)), changes allowed: 1
==========
Substitution B for A suggested.
Substitution succeeded.
(B>(∼B))
(B>(∼B))
Unification succeeded: (B>(∼B))

For some pairs of formulas, for example (B>(C>(D>((a>a)>((b>b)>((c>c)>d))))))) and ((A>A)>((B>B)>((C>C)>(b>(c>(d>D)))))), unification requires exponential running time.

==========
(B>(C>(D>((a>a)>((b>b)>((c>c)>d))))))), changes allowed:
1
((A>A)>((B>B)>((C>C)>(b>(c>(d>D)))))), changes
allowed: 1
==========
Substitution (A>A) for B suggested.
Substitution succeeded.
((A>A)>(C>(D>((a>a)>((b>b)>((c>c)>d))))))
((A>A)>(((A>A)>(A>A))>((C>C)>(b>(c>(d>D))))))
==========
Substitution ((A>A)>(A>A)) for C suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>(D>((a>a)>((b>b)>((c>c)>d)))
)))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>(b>(c>(d>D))))))
==========
Substitution (((A>A)>(A>A))>((A>A)>(A>A))) for D
suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A

>A)))>((a>a)>((b>b)>((c>c)>d))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>(b>(c>(d>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
==========
Substitution (a>a) for b suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((c>c)>d))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(c>(d>(((A>A)>(A>A))>((A>A)>(A>A)))))
))))
==========
Substitution ((a>a)>(a>a)) for c suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((((a>a)>(a>a))>((a>a)>(
a>a)))>d))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>(d>(((A>A)>(A>A))>((A
>A)>(A>A)))))))))
==========
Substitution (((a>a)>(a>a))>((a>a)>(a>a))) for d suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((((a>a)>(a>a))>((a>a)>(
a>a)))>(((a>a)>(a>a))>((a>a)>(a>a)))))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((((a>a)>(a>a))>((a>a)>(
a>a)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
==========
Substitution A for a suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>
A)>(A>A)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>
A)>(A>A)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
Unification succeeded:
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>
A)>(A>A)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))

The resulting formula is exponentially longer than the input of the program. Hence, improvement of the algorithm is not possible without redefinition of the propositional calculus. This important negative result is, however, not completely surprising. Similar inefficiencies are observed in the related fields of propositional calculus, and relative improvements are achieved through introduction

of the equality in language or equivalent use of alternative data structures[1]. That idea is, also, fruitfully applied on the unification problem.[2]

---

[1] The most important examples are described in **G. S. Tseitin,** On the Complexity of Derivation in Propositional Calculus, in *Studies in Constructive Mathematics and Mathematical Logic,* Part 2. Consultant Bureau, New York **1968**, pp. 115-25. and **S. A. Cook** and **R. A. Rechkow,** The Relative Efficiency of Propositional Proof Systems. *Journal of Symbolic Logic* 44 (**1979**), pp. 36-50. We addressed similar problem in **K Majorinc,** Extension Rule for Non-Clausal Propositional Calculus, *Fundamenta Informaticae,* Vol 31, No 2, August **1997**, pp. 107-16.

[2] Few quadratic and linear time algorithms for unification in more general sense are reported. Perhaps the best known one is described by **A. Martelli** and **U Montanari** in An Efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems* **4**(2), **1982,** pp. 258-82.