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 (\(\neg A\)), (A>B), (A>(\(\neg (A>B)\))) 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 ((\(\neg (\neg A)\))>\(A\)) 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, 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 (far less researched with exception of the works of S. N. Vassilyev).

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 of formulas.

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

```unicon
link strings
link sets

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

---


$define \textit{xxxxxx1}$
$define \textit{init-to} := (if \textit{temp_init_to} then \textit{temp_init_to} else$
$define \textit{xxxxxx2}$

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

The program links to standard libraries "strings" and "sets." A few simple macros are defined. Only the infix macro operator \textit{init-to} deserves some comment. The expression ((\textit{x init-to} expr)) is equivalent to longer expressions like \{/\textit{x}:=expr; \textit{x}\} or \{if/\textit{x} then \textit{x}:=expr else \textit{x}\} that allow initialization of variables in the same place they are used in loops. The macro \textit{init-to} is usually slower than initialization outside of the loop. In some expressions, like every ((\textit{x init-to} 0))+:=1 to \textit{n}, one can replace ((\textit{x init-to} 0)) with simple (\textit{x}=0). Also, one execution of the macro in some procedure must be completed before another execution is started. Hence, nested expressions like ((\textit{x init-to} ((\textit{y init-to} 0)))) do not work correctly.\textsuperscript{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.\textsuperscript{2}

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

\textsuperscript{1} It seems to be one of the most frequent problems with macros.
\textsuperscript{2} C. Evans implemented a more powerful macro system and special syntax (x :$ expr) for \{/x:=expr; x\} in his private build of Unicon.
\textsuperscript{3} <http://www.scintilla.org/SciTE.html>. 

15
For testing and demonstration purposes, four pairs of formulas are stored in the list \( F \) and passed as an argument to the procedure \textit{unified}. That procedure returns a record consisting of (1) the formula resulting from unification and (2) a list of all performed substitutions.

The procedure \textit{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 \textit{unified}, only formulas \( F[i] \) such that \( Fca[i] = \text{TRUE} \) can be changed. If unification succeeds and \( Fca[i] \) was FALSE, then \( F[i] \) can be obtained from \( F[3-i] \) by substitution. Less formally, \( F[i] \) is a special and weaker case of \( F[3-i] \).

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.

\begin{verbatim}
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
\end{verbatim}
procedure \( jdx(\textit{LL}) \)
\begin{verbatim}
suspend 1 to \texttt{card\_columns(\textit{LL})}
end
\end{verbatim}

procedure \( \texttt{column(\textit{LL}, j)} \)
\begin{verbatim}
if \( j \) then suspend \( \texttt{column(\textit{LL}, jdx(\textit{LL}))} \)
else \{ every \( \texttt{L} := \textit{\textit{LL}} \) do
put((C \texttt{init\_to []}) , if \( j <= \#\textit{L} \) then \( \textit{L}[j] \) else &null)
return \( C \)
\}
end
\end{verbatim}

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

procedure \( \texttt{is\_simple\_type(x)} \)
\begin{verbatim}
if \( \texttt{type(x)} \) == ("real"|"integer"|"string") then return \( x \)
end
\end{verbatim}

procedure \( \texttt{generalized\_application(p, L)} \)
\begin{verbatim}
every \texttt{put(result:=[], p(L))}
\texttt{return result}
end
\end{verbatim}

procedure \( \texttt{equal\_by\_value(X)} \)
\begin{verbatim}
if not \( \texttt{different\_by\_value(X)} \) then return \( X \)
end
\end{verbatim}

procedure \( \texttt{different\_by\_value(L)} \)
\begin{verbatim}
\( S := \texttt{set(L)} \)
if \( \texttt{member(S, \"equal\") \& \texttt{member(S, \"different\")} \) 
then \texttt{error("Ambivalent different/equal\_by\_value."))}
if \( \texttt{member(S, \"equal\") \) then \texttt{fail}
if \( \texttt{member(S, \"different\") \) then return \( L \)
case \texttt{card\_nulls(L)} of \{ 1: return \( L \); 2: \texttt{fail} \}
\texttt{return case} \texttt{card(is\_simple\_type, L)} of 
\{ 1: \( L \)
\texttt{2: if} \( L[1] \approx \textit{L}[2] \texttt{then} \textit{L} \texttt{else} \texttt{fail}
\texttt{0: if different\_by\_value(column(L)) then} \textit{L} \texttt{else} \texttt{fail}
\}
end
\end{verbatim}
The predicates \texttt{is\_true} and \texttt{is\_false} allow convenient combination of Boolean and success/failure program control flow.

The generator \texttt{jdx(LL)} accepts a list of lists as an argument; if understood as two-dimensional array, \texttt{jdx} generates indexes of its columns. The procedure \texttt{column(LL, j)} returns \texttt{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 \texttt{LL}. Expressions symmetrical to \texttt{jdx(LL)}, \texttt{column(LL, j)} and \texttt{column(LL)} are \texttt{1 to *L}, \texttt{LL[j]} and \texttt{!LL} respectively. Syntactical symmetry can be achieved by implementation of procedures \texttt{idx(LL)}\(^1\) and \texttt{row(LL, j)}.

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

The procedure \texttt{generalized\_application(p, L)} returns the list \([p(L[1]), ..., p(L[*L])]\). It is similar to R. Griswold’s \texttt{apply} in the Icon Program Library, file "apply.icn". Further generalization can be useful.

A few procedures with names containing the prefix \texttt{card}\(^2\) count elements of the structures satisfying given criteria.

Unicon’s built in operator \texttt{===} and its negation \texttt{~===} compare equality of the two structures “by reference.” Although there are few similarities with set-theoretical equality, \texttt{===} does not satisfy the axiom of extensionality\(^3\). For example, \([1, 2] = (1, 2)\) is true in set theory, while its Unicon equivalent \texttt{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 \texttt{different\_by\_value} and \texttt{equal\_by\_value} presented here are more limited than J. P. de Ruiter’s procedure. However, they have one useful additional property. Pseudo-keywords \texttt{&equal} and \texttt{&different}, are defined as \texttt{equal\_by\_value} and \texttt{different\_by\_value} to any value. Comparison between \texttt{&equal} and \texttt{&different} is not defined and will result, in a runtime error if attempted.

1. The function \texttt{key} is equivalent to \texttt{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, \texttt{set([1, 2]) == set([1, 2])} never succeeds in Unicon.
5. R. Griswold’s procedure \texttt{equiv (equiv.icn, I.P.L.)} and J. P. de Ruiter’s procedure \texttt{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])[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-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 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; 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.
(\((a>(b>c))>((B>(C>B))>D))\)
(\((a>b)>(a>c))\)

Substitution \(a\) for \(B\) suggested.
Substitution succeeded.
(\((a>(b>c))>((a>(C>a))>D))\)
(\((a>b)>(a>c))\)

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:
(\(\Lambda>(\neg\Lambda)\)), changes allowed: 1
((\neg 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.

$$(B>(~B))$$

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.

$$\text{(B>(C>(D>((a>a)>(b>b)>((c>c)>d))))}$$, changes allowed: 1

$$(A>(~B))$$, changes allowed: 0

Substitution (A>A) for B suggested.
Substitution succeeded.

$$(A>(~B))$$, changes allowed: 0

Substitution ((A>A)>(A>A)) for C suggested.
Substitution succeeded.

$$(A>(~B))$$, changes allowed: 0

The Generator
Vol 1. No 1.
March MMIV.

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.\(^2\)


\(^2\) 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.