Technical Report CS74010-R

A COMPUTER IMPLEMENTATION OF THE TRANSFORMATIONS OF FORMULAS INTO PRENEX NORMAL FORM

Clinton E. Provenza

Department of Computer Science

College of Arts and Sciences

Virginia Polytechnic Institute & State University

Blacksburg, Virginia 24061

The program transforming formulas into prenex normal form along with this paper describing, evaluating, and explaining the interests of such a project constitute an independent study for C.S. 497, accomplished during the Spring quarter of 1974 under Dr. Ruth Manor, Department of Computer Science, VPI & SU.

June, 1974

INTRODUCTION

The purpose of this paper is to explain a computerized process whereby any well-formed formula (wff) of first order predicate calculus can be moved to its prenex normal form (PNF). The main features of the program demonstrate some of the interesting capabilities of the WATFIV compiler in utilizing Markov Algorithms. The first section will describe the steps needed to move a wff to its PNF with respect to our data specifications. Section II will present a short description of the theory of Markov Algorithms and their application to the PNF process. The techniques used in writing the three major subroutines to program the PNF process comprise Section III. The topics of Section IV include acheivements of the program as completed in Fortran with respect to execution time, storage, and generality of data as well as their corresponding restrictions. Section V discusses the interest in such a project and its possible applications. An appendix contains the resulting program with examples.

Section I

In this section the PNF of a wff will be presented. Let us first define a wff of first order predicate calculus and then describe the process by which we move this formula to its PNF. The elementary parts of each wff include predicate letters (A through W less E and O), variables (X, Y, or Z followed by a two-digit subscript 01-99 inclusive), the material implication symbol (>), the negation symbol ($\neg$), left and right matching parentheses, the period (.), and the existential quantifier (&) or universal quantifier (@) with one variable immediately following, enclosed in parentheses. Thus there are twenty-one possible predicate letters and 297 possible variables.

Let us now define a wff recursively as follows:

---An atomic formula - a parenthesized expression consisting of a predicate letter followed by one or more variables separated by commas - is a wff.
eg. (FX01,Y29,X46)

---If M and N are any wffs then so are $\neg$ M and (M>N). In the latter case, M is the hypothesis and N is the consequent.

---If M is a wff then so is any expression consisting of a quantifier followed by an atomic formula. eg. (@X01)(FX01,Y19)

For all input strings, the first symbol must be a left parenthesis corresponding to a right parenthesis at the end of the string. The last symbol in all strings is the end-of-string marker ($). In order that the string not be over-burdened with parentheses, the period replaces the left parenthesis and eliminates the need for a right parenthesis in all cases where more than one quantifier precedes a wff. The period must immediately follow the second and following quantifiers thus, ((@X01)((&X02)((&Z98)(FX01>GX02)))) is replaced by ((@X01)(&X02).(&Z98).(FX01>GX02))

The scope of a quantifier is the wff immediately following the quantifier. The scope of a variable in a quantifier expression consists of all occurrences of that variable in the scope of the quantifier. A quantifier is initially placed only in case it is at the beginning of the wff or preceded only by other quantifiers. A variable occurs free in a formula if that occurrence is outside the scope of all quantifiers over that variable. A quantifier over a given variable is non-vacuous in case that this variable has at least one free occurrence in the scope of the quantifier. In the contrary case it is called vacuous. A wff A is in its PNF only when all quantifiers are initially placed and each is non-vacuous thereby having the form:

$$Q_1Q_2Q_3\ldots Q_nM$$

where M is called the matrix of A and is a quantifier free wff and each Q_i is either $(@a_i)$ or $(\&a_i)$, $(i=1,2,3,\ldots,n)$ and where $a_1,a_2,a_3\ldots,a_n$ are variables which are all different and which have at least one free occurrence in M. Note that M may coincide with A in the case where n=0, else the n quantifiers are called the prefix of A.

One and only one of the following reduction steps is applicable to a wff A in case there is a quantifier not initially placed in A:[1]

(i) If in canonical order there is an occurrence of a wf part $\neg(@a)C$ not initially placed and where C does not begin with $\neg$, then this wf part is replaced by $(\&a)\neg C$. Otherwise the two adjacent negation symbols cancel leaving $(\&a)C$.

(ii) If there is an occurrence of a wf part $\neg(\&a)C$ not initially placed, then it is replaced by $(@a)\neg C$ assuming C does not start with $\neg$, else we treat it as in (i).

(iii) If there is an occurrence of a quantifier not initially placed as in the wf part $((@a)C>D)$, then so long as there are no free occurrences of a in D, this wf part is replaced by $(\&a)(C>D)$. If there is an occurrence of a in D, then all such occurrences are substituted to a

completely new variable not occurring in A. In this step and in steps (iv), (v), and (vi) we do not want to bind a free variable by moving a quantifier.

(iv) If there is an occurrence of a quantifier not initially placed as in the wf part ((&a)C>D) and a is not free in D, then this wf part is replaced by (@a)(C>D).

(v) If there is an occurrence of a quantifier not initially placed as in the wf part (C>(@a)D) and a is not free in C, then this wf part is replaced by (@a)(C>D).

(vi) If there is an occurrence of a quantifier not initially placed as in the wf part (C>(&a)D) and a is not free in C, then this wf part is replaced by (&a)(C>D).

We now implement these reduction steps utilizing the Markov Algorithms of the next section.

Section II

A Markov Algorithm[2] (MA) is a finite set of productions $P_1, P_2, P_3, ..., P_n$ used to make canonical substitutions in an input string to accomplish a desired end. For each MA there is a specified alphabet, set of markers, and set of variables whereby each set is disjoint from the others. Let $\rightarrow$ and . be symbols not found in the alphabet, variables, or markers, then we define $\beta \rightarrow \beta'$ to be a simple production and $\beta \rightarrow .\beta'$ to be a terminal production and β to be the antecedent and β' to be the consequent. A production P_i (i=1,2,3,...,n) is applicable to a given string σ in the case that β is a substring of σ. If P_i is applicable then where the symbols of the antecedent are from the alphabet or markers β must exactly match a substring σ_1. If the symbols of the antecedent include variables, then for every variable of β, that variable is temporarily assigned an alphabet character in σ_1. Note that variables do not range over markers.

This variable assignment is then used in the transformation when the substring is replaced by β'.

eg. Let X, Y, and Z be variables, 1 and 2 be markers, and A and B be alphabet characters, then the following independent productions are applicable to the string: B1AA2

Antecedent		Consequent	Result
1AA2	→	1A2A	B1A2A
XX2	→	X2X	B1A2A
X1YZ	→	ZXY1	ABA12

The following productions are not applicable to the string: B1AA2

Antecedent		Consequent	Reason
X1X	→	2XX	The first X of the antecedent is assigned to B, therefore it cannot be assigned to A.
XYY	→	2YY	X cannot be assigned to a marker.
XX2	→	2XY	Y is not assigned to an alphabet character. This must be done in the intecedent.

The sequence of productions $P_1, P_2, \ldots, P_n$ is applied to a string σ in the following manner:

1. Set j = 1.
2. If Pj is applicable, go to 4.
3. Set j = j + 1. If $j \leq n$ go to 2. Else algorithm is blocked.
4. Apply Pj to σ to obtain σ', which may or may not be different from σ. Set $\sigma = \sigma'$. If Pj was simple go to 1. Else algorithm terminates.

In the last section, a wff was defined. In order that all wffs so defined will be processed correctly by the following MA's, we define an

alphabet: ABCDFGHIJKLMNPQRSTUVWXYZ0123456789

with markers: $()\neg$,&@>.

and variables: *?+/

each of which may range over any characters in the alphabet. The symbol (Λ) refers to the empty string.

Recalling that steps (i) and (ii) of the PNF reduction process serve to move all negation symbols to the right of each quantifier and to delete all double occurrences of this symbol, we define a MA to do this for us:

P1: ¬ ¬ →Λ
P2: ¬(@?*+) →(&?*+)¬
P3: ¬(&?*+) →(@?*+)¬

Steps (iii) and (iv) will remove each quantifier in a wf part to the outside of that wf part. The process by which variables are kept from becoming bound will be discussed in the next section. At this point we can expect that all necessary substitutions have been completed. Since the final result will leave all quantifiers to the left of the string (i.e., initially placed) we can devise another MA to delete these quantifiers after first duplicating the wff so that these quantifiers are not "lost". Such a MA would have two productions, namely:

P1: (@?*+)→Λ
P2: (&?*+)→Λ

The resulting string after the application of MA #2 is now the matrix.

Referring to the duplicate string, it remains to move all quantifiers outside the wff; however, an intermediate step requires us to move each quantifier outside of "its" wf part. To accomplish steps (iii) and (iv) is relatively simple: we can construct two productions to do this, namely:

P_: ((@?*+)→(&?*+)(
P_: ((&?*+)→(@?*+)(

But to perform steps (v) and (vi), we cannot use a similar pair of productions. Because atomic formulas are parenthesized we have to differentiate between them and wf part parentheses. To avoid this confusion and to shorten the wff that must be examined each time a production is checked for application, we construct a MA to delete atomic formulas and also negation symbols. Since an atomic formula may consist of any number of variables, we first delete the predicate letter and its variable, then all the variables following it, where

each one starts with a comma. Finally, a left and right parentheseis set standing alone is deleted as well as the negation symbol. The third MA:

P1: +/*?→Λ
P2: ,+/*→Λ
P3: () →Λ
P4: ¬ →Λ

Steps (v) and (vi) do not change the sense of the quantifier when it is moved out of its wf part. Since the implication symbols have not been deleted, we will change the sense of each quantifier coming from a consequent once as it crosses the implication symbol, and again as it crosses the left parenthesis. Thus, there is no net effect. Now we have stated the necessary requirements for the fourth and final MA:

P1: (>) →Λ
P2: ((@?*+)→(&?*+)(
P3: ((&?*+)→(@?*+)(
P4: >(@?*+)→(&?*+)>
P5: >(&?*+)→(@?*+)>

By using the result of MA #1, we apply MA #3 and MA #4 to obtain a prefix of the input wff.

Section III

With the basic ideas for quantifier extraction in front of us, it remains to develop a subroutine to make the necessary variable substitutions, to solidify the MA's into a subroutine that yields a separate prefix and matrix, and to provide a third subroutine to remove vacuous quantifiers from the prefix.

The first subroutine, FREE, will replace all free variables that may become bound as the quantifiers are moved "out front" with completely different variables from those occurring in our wff A. Ambiguous quantifiers (i.e., those quantifiers in the scope of another quantifier where both have the same operator variable, eg. (@X01)((&X01)(FX01)>(GX02))) will also be substituted along with all occurrences of the same operator variable in their scope. A counter KEY, initially set to zero, is incremented by one when a left parenthesis is

encountered and decremented by one when a right parenthesis is found. This variable is responsible for determining the scope of each quantifier in the wff. A predicate letter, a comma, or a quantifier indicates a variable is to follow. Any other symbol encountered will cause the pointer variable PTR to be incremented, thus indicating the position of a new symbol in the array STRING which contains our wff A.

Whenever a variable is encountered, the first dimension of a two-dimensional array STACK is searched to see if this variable has occurred previously. If not, it is put on top of the STACK. If the variable occurs in an atomic formula, the integer 5300 is placed in the second dimension. If it is a quantifier variable, the value of KEY is placed in the second dimension. We call this value in the second dimension the status of the variable. Whenever the value of KEY changes, each variable in the STACK is scanned to see if the scope is reached. If so, the status is set to 5000. The scope is determined when the value of KEY matches the variable's status, indicating the corresponding right parenthesis has been reached. Should the variable encountered already be entered in the STACK, the following table illustrates all possible cases:[3]

Status of variable in STACK	ENCOUNTERED VARIABLE	
	w/o quantifier	with quantifier
<5000	increment PTR to new variable	substitute all variables in the scope of the quantifier encountered to a new variable, increment PTR
=5000	substitute variable encountered to a new variable, add new index to status, increment PTR	
>5000 and ≠5300	subtract 5000 from status to obtain an index and search for this new index in STACK	
=5300	increment PTR to new variable	substitute variable encountered to a new variable, add new index to status, increment PTR

The index is a unique value associated with each variable in the following manner:

If the variable is X, its subscript is the index value.
If the variable is Y, its index is 99 + its subscript.
If the variable is Z, its index is 198 + its subscript.

The second major subroutine needed, PULL, coordinates the MA's used in the quantifier extraction process. Subroutine SEARCH checks to see if a production applies, if it does, subroutine APPLY performs the transformation.[4] After execution of MA #1, the string containing our wff A is duplicated into another array called MATRIX. This array is then passed to the second MA where the quantifiers are dropped forming the matrix, while the STRING goes on to MA #3. Before the STRING is passed for processing by MA #4, the outer parentheses are removed if the input wff contained initially placed quantifiers. These outer parentheses were only needed in subroutine FREE where KEY would have assumed the value zero before the end of the wff was reached. Here they are removed so that MA #4 will not move any initially placed quantifiers across the left parenthesis thereby changing their sense. Finally, PULL blanks out the ends of STRING and MATRIX.

The third and final subroutine, VACUUM, cleans up the prefix by eliminating those quantifiers not having a free occurrence of their operator variable in the matrix. This includes the ambiguous quantifiers as described above. The index of all variables appearing in the matrix are stored in an array. The STRING, which contains our preliminary prefix is scanned and each quantifier is added to a STACK if it has an occurrence of its operator variable in the matrix. This is easily determined by reference to the array containing the matrix variables. All entries but the last one in STACK are then compared to the newly added operator variable to see if there is a previous occurrence. If two quantifiers

have the same operator variable it is the rightmost quantifier that binds any occurrences of its operator variable. Should there be any previous occurrences, those indices are set to 500, indicating they will not be used in the final prefix. When each index is put on the STACK, it is assumed that its corresponding variable was with a universal quantifier. If this is not the case, the additive inverse is put on the STACK.

We now have all the information needed to create the final prefix. An array PREFIX, is mentally considered to consist of blocks, each of which has six character positions to be filled. In actuality it is a continuous character *1 array. The first and last cells of each "block" are set to the left and right parenthesis, respectively. The second cell (if the index value is less than 500) is set to the universal quantifier if the index is positive or the existential quantifier in case the index is negative. Imitating the modulo process, a variable and subscript is obtained from the index. The variable is placed in the third cell with its subscript occupying the fourth and fifth cells. Lastly, VACUUM concatenates the PREFIX with the MATRIX into the STRING array and passes the wff now in its PNF back to the main program.

Section IV

The program described in the last section was written in Fortran IV for a WATFIV compiler. Many extension features were utilized so this program is by no means in American National Standard Fortran. All the extension features could be deleted, reducing the program to ANS Fortran, by adding additional coding to to program around these "difficulties". Indeed, the procedure may have been more efficiently implemented in a string oriented language such as SNOBOL4, but the author was unfamiliar with its capabilities.

At present, the program can handle wffs of 120 characters. In order to increase this limit, the matching dimensions of four arrays must be increased.

(PREFIX, MATRIX, STRING, and HOLD) Since these arrays are of character*1 type, indicating four characters are stored per work; an increase to 1000 characters per wff would only require 1 K-bytes of additional storage. (where K = 1000 bytes) The object code core requirement is 16.88 K-bytes for 380 statements. Halfword integers were used implicitly throughout the program since this permits integer values to range up to roughly 65,000. The array area in core utilizes 3.36 K-bytes. Variables used in several subroutines where placed in common. Compilation time averages 0.70 seconds with an average length wff of sixty characters requiring about 4.5 seconds to move it to its PNF. The simplest atomic formula (FX01) is processed in 0.21 seconds, execution time. Surprisingly, all debugging and program case testing was completed on an IBM System 370 auto-batcher. This instantaneous processing unit allows 10 seconds compilation-execution time with 28^{+} K-bytes maximum storage.

The very nature of wff composition would prohibit variables to occur more frequently than one variable every five character symbols. Therefore with 297 different variables available, this limit would easily accomodate a wff of nearly 1500 characters. For this reason, letting subscripts range from 0-9 inclusive would offer 30 different variables - quite sufficient for any practical application. Should particularly long wffs need to be processed by this method, assuming a time limitation is imposed then the ten statement main program could be segmented so the variable substitution is performed on the first run, the quantifier extraction on the second, and the vacuous quantifiers removed on the third. Since subroutine PULL is the longest with respect to execution time, even that subroutine could be easily segmented to first produce the matrix, then to prepare the wff for quantifier extraction, and a third run could determine the

the preliminary prefix. As a last resort, the wff could be broken down manually where each wf part is processed separately, then consolidated and processed in segments as above.

Although the rules governing wff construction were stated in the first section, some aspects warrant repitition here to point out generalties as well as restrictions. The number of variables following a predicate letter is not limited. Also the number of quantifiers preceding a wf part or atomic formula is not limited so long as the period convention is held in compliance. Negation symbols may appear in any number outside of atomic formulas and quantifer expressions; however, when placing negation symbols between quantifiers, the symbol may not be interjected on the left of the period - it must follow the period on the right. Parentheses, which are most important in determining all quantifier scopes and nesting, are used in four cases:

(1) They must enclose quantifier expressions.
(2) They must enclose atomic formulas.
(3) They must be used to surround two wf parts separated by the implication symbol.
(4) They must surround the entire wff only in case the wff has quantifiers that are initially placed.

Imbedded blanks are not permitted anywhere in the wff before the end-of-string symbol. Lastly, more than one input wff can be processed per run.

Section V

This last section discusses the relative merit of such a project. We investigate the possibilities of extending the process described here to Automatic theorem proving. Although the decision problem of the pure functional calculus of first order is known to be unsolvable (i.e., Given an arbitrary wff, it is impossible to ascertain mechanically whether or not this wff is a theorem)., there exists a special class of wffs where the decision problem IS solvable.[5]

The simpler of these will be listed here:[6]

(i) Wffs having a PNF such that in the prefix, no existential quantifier precedes any universal quantifier.

(ii) Wffs in PNF that have only universal quantifiers in the prefix.

(iii) Wffs having a PNF in which the matrix satisfies the condition of being a disjunction of elementary parts and negations of elementary parts or equivalent by laws of propostional calculus to such a disjunction. (Elementary parts are analogous to atomic formulas).

(iv) Wffs having a PNF with a prefix of the form $(@a_1)(@a_2)\ldots(@a_m)(\&b)(@a_1)(@c_2)\ldots(@c_k)$.

(v) Wffs having a PNF with a prefix of the form $(@a_1)(@a_2)\ldots(@a_m)(\&b_1)(\&b_2)\ldots(\&b_n)(@c_1)(@c_2)\ldots(@c_k)$, and a matrix in which every elementary part that contains any of the variables $b_1,b_2,\ldots,b_n$ contains either all of the variables $b_1,b_2,\ldots,b_n$ or at least one of $c_1,c_2,\ldots,c_k$.

(vii) Wffs having a PNF with a prefix terminating in $(@c_1)(@c_2)\ldots(@c_k)$ and a matrix in which every elementary part that contains any of the variables occurring in the prefix contains at least one of the variables $c_1,c_2,\ldots,c_k$.

In all of the above cases, the PNF of the wff is needed. The first case has been proven by finding the associated formula of propositional calculus matching the one given and then by testing that formula or some disjunction of it to see whether or not it is a tautology. All of the above cases have corresponding proofs that yield after a finite number of steps, an answer as to whether or not a given wff is a theorem.

A second use of the PNF process is involved with obtaining the Skolem normal form of a given wff W. A wff of first order predicate calculus is said

to be in its Skolem normal form when it is in its PNF with no free individual variables and has a prefix of the form:

$$(\&a_1)(\&a_2)\ldots(\&a_m)(@b_1)(@b_2)\ldots(@b_n)$$

where $m \geq 1$ and $n \geq 0$. In order to obtain the Skolem normal form, we start with the wff in its PNF and continue with the reduction steps.[7]

We now turn to the resolution principle[8] which is applicable to any wff of first order predicate calculus after first moving it to its Skolem normal form. This process is guaranteed to give an answer after a finite number of steps only if the formula in question is indeed a theorem. If it is not a theorem, it may run forever. Thus, in programming the resolution principle and testing a wff for theoremhood, if at a given time no answer has been reached, it may be due to the fact that more time is needed or it will run forever and no additional time will bring us the answer.

Such an implementation of the cases listed above could be incorporated into a much larger project aimed at a partial solution to the decision problem. However, this is outside the scope of this paper and should be viewed as a possible continuation of this project.

FOOTNOTES

[1]Adapted from Introduction to Mathematical Logic, vol. 1, Alonzo Church, pp. 209-211, with slight modifications.

[2]Description condensed from Data Structures: Theory and Practice, Alfs Berztiss, pp. 153-159.

[3]Those interested in the finer details are referred to the comments and coding of subroutine FREE of the program.

[4]For a detailed explanation of the implementation process for MA's see the comments and coding of these subroutines in the program.

[5]See Alonzo Church in The Journal of Symbolic Logic, vol. 1 (1936), pp. 40-41, 101-102. Also Hilbert and Bernays, Grundlagen Der Mathematik, vol. 2, Supplement II.

[6]A detailed explanation can be found in Introduction to Mathematical Logic, (see footnote 1) pp. 246-257.

[7]These reduction steps on pp. 224-227 of Introduction to Mathematical Logic, (see footnote 1).

[8]See Chang and Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973, Chapter 5.