KRR Report
Converting FOL Expressions to CNF Form

Divita Jain, Spandan Panda

This report outlines the process we used to convert FOL expressions (as input from a text file) to give the CNF form of the expression as output in a text file.

Example Input: The inputs are in Operator form.
'∀x (p(x) ⊃ q(x))'

Note: Use of underscore ('_') in variable naming might interfere with the function that is renaming variables and is therefore not to be used. For eg. Use x1 instead of x_1.

Step-0: Converting input formula into list format

Firstly, the FOL Expressions taken from a text file is passed through a function called **parse_logical_expression**.
Input: string of FOL expression
Output: list form of that expression

A precedence dictionary is defined with specific values assigned to each operator, indicating their priority level. The lower the value, the lower the precedence, meaning that operators with higher values are processed first.
The function uses a stack-like mechanism (through list operations) to correctly associate operators with their arguments, nesting expressions as necessary based on the logical structure of the input.

Example Conversion:
∀x (p(x) ⊃ ∀y (q(y) ⊃ r(y))) **gives**
['forall', 'x', ['implies', ['p', 'x'], ['forall', 'y', ['implies', ['q', 'y'], ['r', 'y']]]]]

List Format:
Binary operators- are given as [Operator, Operand1, Operand2] format
Unary operators- are given as [Operator, Operand] format
Quantifiers- are given as [Quatifier, Variable, Formula] format
Predicates and functions- are given as [Predicate/Function, Variable1, Variable2, ...] format

Step-1: Standardize the variables

For this, we are using a function called **standardizeVariables** which takes in the list form gotten in Step-0 to standardize variables apart across quantifiers. This ensures renaming variables so that the same symbol does not occur in different quantifiers.

Input: a FOL expression represented in List form (Output of Step-0).
Output: a FOL expression in the same nested list format where all variables have been renamed according to their scope.

The function also uses two optional dictionaries: var_occurrences to track the occurrences of each variable and current_renames to map original variable names to their new standardized names within the current scope.

Example Conversion:
['or', ['forall', 'x', ['P', 'x']], ['exists', 'x', ['Q', 'x']]] **gives**
['or', ['forall', 'x', ['P', 'x']], ['exists', 'x__1', ['Q', 'x__1']]]

The use of double underscore (‘__’) is to prevent any conflicts with the renaming that would be done in the last step.

Step-2: Eliminating the Implication

For this, we are using a function called **parseImplications** that uses 2 helper functions **eliminateImplication** and **eliminateIFF**.
The parseImplications is the main function that processes an entire FOL expression recursively to replace all occurrences of 'implies' and 'iff' with their equivalents using 'and', 'or', and 'not' while the helper functions take in a specific part of the input FOL expression that have ‘implies’ and ‘iff’ respectively.

Input: Output from Step-1 (List form)
Output: Transformed FOL expression in list format after eliminating ‘implies’ and ‘iff’.

Example Conversion:
['or', 'D', ['implies', ['and', 'A', 'B'], 'C']] **gives**
['or', 'D', ['or', ['not', ['and', 'A', 'B']], 'C']]

Step-3: Propagate negation inwards

In order to push the negation sign to the atomic level, we are using a function called **parseNOTs** and a helper function called **propagateNOT.**

The propagateNOT function takes an FOL expression where the main operator is 'not'. The input is expected to be a list where the first element is the string 'not', followed by the expression to negate. This function uses logical equivalences to move the negation inwards:
Negation of 'or' and 'and', double negation and negation of quantifiers applying De Morgan’s Laws at the level of quantifiers.

For composite expressions, the function recursively applies these transformations to each component.

parseNOTs is the main function that processes an entire FOL expression recursively to propagate negations inwards, aiming to achieve a form where negations apply directly to atomic propositions or predicates. It identifies expressions where the main operator is 'not' and applies the propagateNOT function.

Input: Output from Step-2.
Output: List form with 'not' pushed at the atomic levels.

Example Conversion:
['or', ['not', ['or', ['not', ['P', 'x']], ['P', 'y']]], ['P', 'z']] **gives**
['or', ['and', ['P', 'x'], ['not', ['P', 'y']]], ['P', 'z']]

Step-4: Eliminating the Existential Quantifier (Skolemization)

Used a function called **skolemize** that accepts the FOL expression (from Step-3) in the list form. It takes the help of the helper function **replace_variables** which takes in the variable to be replaced and the replacement and replaces all instances of that variable. The output is a transformed FOL expression where existential quantifiers are replaced either by Skolem constants (if no universal quantifiers are in scope), prefixed by *skc* or Skolem functions (if universal quantifiers are in scope), prefixed by *skf*.

For skolemconstants, we will use the format skc0, skc1, skc2, ...
For skolemfunctions, we will use the format skf0, skf1, skf2, ...

$\forall x(\exists y(P(x,y)))$ after skolemization is $\forall x(P(x, skf0(x)))$

Note: Running the same function (without re-running the function def) again on the output of the previous function will give $\forall x(P(x, skf1(x)))$ which is ideal as we will be processing multiple FOL formulas and this prevents the clash of names.

Input: Output from Step-3.
Output: List form with skolemized variables.

Example Conversion:
['forall', 'x', ['exists', 'y', ['P', 'x', ['f', 'y']]]] **gives**
['forall', 'x', ['P', 'x', ['f', 'skf0(x)']]]

Step-5: Removing the Universal Quantifier

The **remove_universal_quantifiers** function is used to simplify a First-Order Logic (FOL) expression by stripping out all universal quantifiers (forall) and collecting the variables they quantify over.

Input: Output from Step-4.
Output: Modified FOL expression without any universal quantifiers, and a set containing all the variables that were quantified universally in the original expression.

Example Conversion:
['forall', 'x', ['and', ['or', 'A', 'B'], ['forall', 'y', ['P', 'x', 'y']]]] **returns**
Modified Formula: ['and', ['or', 'A', 'B'], ['P', 'x', 'y']]
Variables: {'y', 'x'}

Step-6: Distributing union over intersection

We use 3 functions for this step:
isDistributionCandidate: This helper function checks if a given logical expression is a suitable candidate for applying the distribution of or over and. It returns True if the root operator of the expression is ‘or’ and at least one of its sub-expressions (operands) is an ‘and’.

distributeOR: This helper function takes a logical expression that meets the distribution criteria (isDistributeCandidate returns True for the expression) and applies the distribution rule.

- If both operands are ‘and’ expressions, the function enters a double loop to distribute the ‘or’ operation across each pair of sub-expressions from these ‘and’ expressions.
- If only one of the operands is an ‘and’ expression, the function distributes the other operand across all sub-expressions of the ‘and’.

The result is a new expression structured with and at the top level, containing all distributed combinations as its operands.

parseDistribution: This is a recursive function that processes a logical expression to apply the distribution rule wherever possible. It first checks if the current expression is a candidate for distribution using isDistributionCandidate. If it is, distributeOR is applied to transform the expression. The function then recursively processes all sub-expressions to ensure distribution is applied throughout the entire structure of the expression.

Input: Output from Step-5.
Output: Modified List form after distributing ‘or’ over ‘and’.

Example Conversion:
['or', ['and', ['G', 'x', 'y'], ['H', 'z']], ['P', 'z']] **gives**
['and', ['or', ['G', 'x', 'y'], ['P', 'z']], ['or', ['H', 'z'], ['P', 'z']]]

Step-7: Eliminating the conjunctions and getting the Clauses

The **extract_clauses function** and its helper **flatten_or** are designed to process a logical expression in Conjunctive Normal Form (CNF) and extract individual clauses from it.

Input: Output from Step-6.
Output: List of clauses that were connected by 'and', where each clause is a list of literals that were connected by 'or' in the input CNF formula.

Example Conversion:
['and', ['or', ['P', 'x', 'y'], ['Q', 'z']], ['or', ['R', 'x'], ['S', 'y']]] **gives**
[[['P', 'x', 'y'], ['Q', 'z']], [['R', 'x'], ['S', 'y']]]

Step-8: Renaming the variables for unique variables in each clause

The **rename_variables** function takes a list of logical clauses and a list of variable names as input and returns a new list of clauses with variables renamed to ensure uniqueness across the entire set of clauses. Each repeated variable between clauses is renamed to maintain its uniqueness by appending an underscore '_' followed by a count which reflects the number of clauses in which it has been used globally till now minus 1.
For example, if a variable 'x' occurs more than once across different clauses, it is renamed to 'x_1' in the second clause and 'x' in the first clause (no renaming in the first clause).
The function also handles special cases where variables are embedded within Skolem functions and nested functions, ensuring that these embedded variables are uniquely renamed as well.

Input: Output from Step-7 and variables, a list of variable names (Output from Step-5) that appear in the clauses.
Output: List of clauses similar in structure to the input but with all variables renamed to ensure they are unique across clauses.

Example Conversion:
[[['G', 'John']], [['not', ['P', 'x']], ['R', 'x', 'y']], [['not', ['P', 'x']]]] **gives**
[[['G', 'John']], [['not', ['P', 'x']], ['Q', 'y'], ['R', 'x', 'y']], [['not', ['P', 'x_1']]]]

Removing Duplicates

This step uses a function called **remove_duplicates** which removes any duplicates within clauses to give cleaner clauses. It uses two helper functions **predicates_are_equal** and **remove_clause_duplicates** which are representative of themselves.

Equality Axioms

Here, our task is see that if a equality exists in the knowledge base/ FOL expression, then append all the necessary equality axioms so that the query can be resolved using methods like Resolution Refutation. If we find ‘equals’ in our knowledge base, we have to append the 3 equality axioms: Reflexivity, Transitivity and Symmetry to our current KB and in addition to these, for each function and predicate in the knowledge base, we add their respective axiom for substitution.

For this, we will have to first get all the functions and predicates from the list of clauses that we have which we will do using a helper function called **get_preds_func.** The main logic for finding the predicates and functions was that functions would be present inside predicates or another function. But predicates cannot be nested inside predicates or functions. It also returns an **equals_flag** which is True when ‘equals’ is present; False otherwise.

After getting the predicates and functions, they are passed to the main function called **eq_axioms** which appends all the axioms to our existing list of clauses.

Combining steps

The functions from **Steps 1-7** mentioned above are sequentially applied in the wrapper function **parseLogic.**

Reading Input from File

Here, we read the content of a txt file containing various FOL formulas and store it into a list using the function **read_file.**

Converting List Form to Operator Form

The output we got after all the steps is still in list form which is not very readable as we increase the complexity of the expressions, and therefore are converted back to the operator form using the function **logical_expression.**

Main Function

This function essentially combines all the steps along with the supplementary functions like reading inputs, removing duplicates, adding equality axioms etc. to give a final pipeline for the task.
First, the txt file is read and all the expressions are stored in a list. Then for every formula in this list, we convert it into list form and pass it through the wrapper function parseLogic (Steps 0-7). Now, the clauses that we get from each expression at this point are appended to a bigger list. Now, on this list we apply the get_preds_funcs function followed by eq_axioms to add the equality axioms when necessary. And then rename all the variables as necessary (Step-8). Remove any duplicates to get cleaner clauses and convert them back into operator form.

Note: We have included **extract_variables** and **transform_clauses** functions in the end to store the final clauses in an ‘output.txt’ file in the format that is accepted by the parser to get xml format (to ease the process for the subsequent teams). An example usage has also been included in the end of the file.