Media Engineering and Technology Faculty
German University in Cairo

Unification and Clause form Algorithms

Project Two

Name:	Hossam Amer
	Mohamed Gad
	Taher Galal
ID:	13-6070
	13-7516
	13-4426
Tutorial:	T-14
Submission Date:	15 December, 2011

1.1 Unification

The process that was followed to perform the unification process is very similar to the process that was discussed in class. There will be a check, if the substitution is set through out the process, to nothing then this will return that there is no substitution. On the other hand, we will check if both expressions are equal then we will return the substitution that was reached till this point. Otherwise, if the first expression that is an input to the unification process is a variable we will try to unify this variable with the other expression. Also the same check is performed for the other expression if the first expression was no a variable. If both expressions are not variables we will try to unify the first part of the expression and also we will call the unification of the rest of the variable. Moreover if any of the expressions is not an expression or any is empty i will return that there is no unification. Otherwise if they are both sequences and have the same length i try to unify them and recursively try the other elements.Lastly if none of the above cases then i will return that there is no unification that can be performed.

1.1.1 Unification Methods

Next we will describe more the process that is performed in our unification method.

unify(expression, expression, μ)

This method is the root of the unification process. This method does the process that was described above in the unification method.

is_variable(expression)

This method is used to check if the expression is a variable this process is done through the following. It will first check if the value does not have arguments and at the same time the operator is a variable symbol. If all of the above conditions are fulfilled then this is specified as a variable otherwise it is not a variable.

is_var_symbol(String)

This will perform a check to see if the element that is entering is a variable symbol or not. it is taken that any lower case letter is a variable symbol.

unify_var(variable,expression,μ)

This method will try to get a substitution that will for the variable to be the same as the expression but first some conditions need to be satisfied. I need to check if the variable is already in μ then we will try to unify the rest this substitution with the rest of the expression that we had at first. Otherwise, we will need to check to see if the variable already occurred in μ then this will specify that I need to return that there is no unification can occur because we will reach a loop. If it is none of the above cases then we will add this variable to μ.

occur_check(variable, expression)

This method will check to see if this variable already exists in the expression. The first check will be to see if the variable is equal to the expression which will mean that the expression is a variable. If it is the case then this means that the variables are the same implying that the variable occurs. If not the case then a check will be performed to see if the operands of the variable are the same as those of the expression or the rest of the expression parts. If not the case and the expression is a sequence then the check is performed for every element in the sequence.

extend(μ, variable, value)

This method will add the the substitution that is performed on the variable. and then this element is added to μ. Also a check is performed to make sure that any changes needed to be performed the the μ is performed and that the elements are consistent.

subst(expression,dictionary)

This method will start by replacing all occurrences of the dictionary in the expression.

1.2 Clausal Form

To be able to perform the clausal form first we will need to perform Conjunctive normal form(CNF). the process t perform the clausal normal form consists of 8 steps that need to be performed in a certain order these are the following as discussed in class.

1. Remove equivalence: this is done by adding to imply expressions instead of the equivalence.
2. Remove implication: the implication will also be removed by using the implication elimination rule. It specifies that any implication can be changed to and $\vee$ and a $\neg$ to the element that follows the implications.

3. Move Not Inwards: This step, as its name implies, pushes any not inwards into the expression.
4. Variable Renaming: In this process we will start to rename the variables of different quantifiers. This is done to avoid two quantifiers bind to the same variable.
5. Skolemize: Remove the $\exists$ and at the same time add a function in terms of the outer $\forall$ variable instead of the dropped $\exists$.
6. $\forall$ Removal: the $\forall$ is just dropped this is possible because we already renamed the variables so a conflict does not occur.
7. Distribute the $\wedge$ over the $\vee$.
8. Flatten: Remove non-needed brackets.

These are the steps that will be followed to make the expression into CNF. Last but not least we will make Clausal Form a list of lists.

to_clause_form(expression, trace)

This method will be used to convert the expression to Clause form (CF). The given trace will specify if the user wants to see the process of creating the CF. Then the user will change to clausal form where it is a list of lists. The big lists signify the expressions between $\vee$ operators. However, the small ones signify expressions grouped with the $\wedge$ operator.

to_cnf(expression,trace)

This method will create the CNF and this method will call the steps from different methods.

eliminate_equivalence(expression)

This method will perform the equivalence removal. First of all, it checks whether the current expression is a symbol. If it is a symbol and the operator is not $\forall$ neither $\exists$ then it finally returns the expression. If it is not a atom we will keep going through the expression until and check if it is and equivalence it will be removed and replaced with two implications of each part.

eliminate_implications(expression)

This method will perform the removal of the implication. We will check if it is an atom (or a symbol) we will perform no change. Otherwise, we will check if it is an implication and change every implication will be changed to an $\vee$ and the negation of the implied expression.

move_not_inwards(expression)

This method will perform a push to the not inwards as much as possible. There is a check on every part of the expression if it is a ∀ then it will be changed to a ∃. On the other hand, if it is an ∃ then it will be changed to ∀ followed by a not and vice versa. If it is a not then we will change it to a positive literal. The ∧ is changed to an ∨ and vice versa.

standardize_apart(expression,dictionary)

This method will perform renaming of variables. It will start to check for all the ∀ and the ∃ and start renaming variables that are the same to different names to make sure that no two variables have the same name.

skolemize(expression,qun=[],dict=)

TODOOOOOOOOOOXXXXXXXXXXXXXX

eliminate_for_All (s)

This method will simply remove the ∀. A check is done to see if it is a ∀ it is dropped and the rest of the expression is returned. It maps the method on all elements inside the expression to remove all existent ∀ operators.

distribute_and_over_or(expression)

This method will perform steps 7 and 8 at the same step. The ∨ is distributed if there is an expression were it can be pushed in. Also extra brackets are removed.

NaryExpr(operator, expressions)

This method or this kind of expression is really important to distribute the any kind of operator over the expression. It is used in our project to distribute the ∧, ∨, and ¬ operators onto the expression. It promotes nested instances of the same op up to the top level.

disjuncts(expression)

This will change expressions between disjunctions into elements in the same list. If it is an ∧ it will be placed in a large list. This method will be used for clausal form.

addClause(expression)

This method will add everything to a list and return it.

disjunction_clause(expression)

This method will call the disjunct method repeatedly with the different elements of the expression to change them into lists as needed by the clausal form.

1.3 References

The sources listed below helped us to implement our project:

1. Lectures and university materials.
2. http://aima.cs.berkeley.edu/python/logic.html