Media Engineering and Technology Faculty
German University in Cairo

Unication and Clause Form

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 Expression

The first order logic expressions are expressed in our implementation Using Expr module. Expression can be either variable, constant, predicate or function. This Expr object consists of 2 main parts: operator and list of arguments. This list of argument can contain either variable, constants, predicates or function expressed as Expressions or a complex expression. operators can be:

1. And: to represent the and in first order logic we will let the user enter an & literal to donate the and
2. Or: to represent the Or in first order logic we will let the user enter | to denote an Or.
3. Not: to represent the Not in first order logic the user will enter $\sim$ to denote and $\sim$
4. $\Rightarrow$(Implication): this is donated by $\gg$ or $\ll$. A$\gg$B means that the A $Rightarrow$B and the opposite is true.
5. $\Leftrightarrow$ (Equivalence): the user can denote the Equivalence by <=> symbol

For quantifier are expressed as special type of predicates

1. $\forall$: is represented by All(variable, Expression). Where the variable is the variable that the $\forall$ is applied to.
2. $\exists$: is represented by Exists(variable, Expression). Where variable is the variable that the $\exists$ is applied to.

Also Expr has some functions that used to know the type of the expression used whether its variable or predicate....

1. is_variable to check whether this is an variable or not.
2. is _var_symbol to check whether the operator this expression is avariable symbol or not
3. is_symbol to check whether the expression operator is predicate symbol or logical operator or not
4. expr that takes string input to create Expr object

Using the above expressions one can perform first order logic expressions. The user we do it as follows, expr(), and inside the bracket we can add any of the above expressions. A sample expression could be:

expr ('All(x, P(x) | Q(x)) >> Exists(y, R(x, y))')

which is equivalent to:

$\forall x$ ((P(x) | Q(x)) $\Rightarrow \exists y$(R(x, y)))

1.2 Unification

It is a process of trying to find a substitution for two first order logic expressions such that the expressions after applying the substitution are identical. It is used in a process of adding extra expressions to the AI's knowledge base using forward chaining and backward chaining.

1.2.1 Unification Implementation

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 not 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 expression. Moreover if any of the expressions is not an expression or empty we will return that there is no unification. Otherwise if they are both sequences and have the same length we try to unify them and recursively try the other elements.Lastly if none of the above cases then we will return that there is no unification that can be performed.

1.2.2 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.

unify_var(variable,expression,μ)

This method will try to get a substitution that will allow the variable to be the same as the expression but first some conditions need to be satisfied. We need to check if the variable is already in μ then we will try to unify the rest of this substitution with the rest of the expression that we had at first. Otherwise, we will need to check to see if the variable is not generated after applying μ on the expression, then will be specified that we need to return that there is no unification that 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. This method is used to check that the variable is not part of any of it's substitution.

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 fix all old substitution

subst(expression,dictionary)

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

1.3 Clausal Form

Clause form is another way to illustrate conjunctive normal form. Conjunctive normal form is achieved by applying a certain number or rules to convert any first order logic expression into conjunction of disjunction of expressions. After obtaining the CNF, it is easily converted into clausal from. The Clausal Form is used for applying resolution and proofing first order logic.

1.4 Clausal Form Implementation

To be able to perform the clausal form first we will need to convert into Conjunctive normal form(CNF). To convert into the clausal normal form consists of 8 steps that need to be performed in a sequential order listed below:

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 $\forall$ then it will be changed to a $\exists$. On the other hand, if it is an $\exists$ then it will be changed to $\forall$ followed by a not and vice versa. If it is a not then we will change it to a positive literal. The $\wedge$ is changed to an $\vee$ and vice versa.

standardize_apart(expression,dictionary)

This method will perform renaming of variables. It will start to check for all the $\forall$ and the $\exists$ 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=)

This method should remove the existenitial quantifiers and substitute its variable with the proper new variable or function from the outer for all quantifier. This is a recursive function. It starts from the most outer operation and as stated before Forall and there exist are represented as predicates with two parts first is the variable and second is the evaluated expression. So, while executing whenever a forall is met its variable is stored to be used as function argument subsitituting following there exist variable. When a Exists is met its variable is stored in dictionary with its value the value is either a new function fn with which its arguments are the collected list of variables of upper foralls. If the list of variables is empty that means the existential quantifier is outer quantifier so a new variable is added to as value to the exist variable. Exists is removed and its expression is returned after substituting each variable from the dictionary.

eliminate_for_All (s)

This method will simply remove the $\forall$. A check is done to see if it is a $\forall$ 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 $\forall$ operators.

distribute_and_over_or(expression)

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

For further documentation, consult the code.

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 $\wedge$, $\vee$, and $\neg$ operators onto the expression. It promotes nested instances of the same op up to the top level.

For further documentation, consult the code.

disjuncts_to_clauses(expression)

This method will convert all | expressions to a list and place the disjunctions in a list.

cnf_to_clauses(expression)

This method will check for $\wedge$ and apply disjuncts_to_clauses on every conjunction.

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.

clause_form_standardize_apart

This method will start to rename the variables inside the clauses. It is performed through a check to see if the variable is already in the dictionary then it is removed other wise it is kept the same. The method will go through all clauses performing this action.

1.5 Running The Code

To run the code you have to run FOL.py file. To test:

1. FOL file you will find testing part, set trace to True to use trace mode.
2. Add new expressions by x= expr('P(x)'), y=expr('P(y)')
3. To test the unify call unify unify(x,y) and clause form to_clause_form(x)
4. To use $\forall x$ use All(x, exprission) and $\exists y$ use Exists(y, expression)

Also, user can run the code through terminal by using :

python FOL.py 'unify' 'expr1' 'expr2' (1 or 0 for trace)

python FOL.py 'toClause' 'expr1' 'expr2' (1 or 0 for trace)

1.6 Running Results

Unification

Expression	μ
P(x,g(x),g(f(a))), P (f (u), v, v)	x: f(a), v: g(f(a)), u: a
P(a,y,f(y)), P(z,z,u)	a: z, y: z, u: f(z)
f(x,g(x),x), f(g(u),g(g(z)),z)	None

Clause Form

Expression	Clausal form
$\exists x$[P(x) $\wedge$ $\forall x$[Q(x) $\Rightarrow$ $\neg$P(x)]]	[[P(z1)], [P(s1), Q(s1)]]
$\forall x$[P(x) $\Leftrightarrow$ (Q(x) $\wedge$ $\exists y$[Q(y) $\wedge$ R(y,x)])]	[[Q(s2), P(s2)], [Q(f1(x1)), P(x1)], [R(f1(x2), x2), P(x2)], [P(x3), Q(x3), Q(s3), R(s3, x3)]]

1.7 Refrences

1. Torsten Hahmann, lecture: "Skolemization, Most General Uniers,First-Order Resolution", (http://www.cs.toronto.edu/ sheila/384/w11/Lectures/csc384w11-KR-tutorial.pdf)

2. AIMIA library by Berkeley uni (http://aima.cs.berkeley.edu/python/logic.html)

3. Lecture notes and materials