Gasser Akila	19-2409	T12
Mohamed El-Menisy	19-5654	T14
Yahia El Gamal	19-1023	T14

Discussion of Unification

Unification is when we have two FOL expressions and we try to unify them into a common instance of a certain form that satisfies both expressions, we try to find a substitution set which when applied to both expressions yields this common instance. Unification is at the heart of first-order reasoning techniques as it provides a systematic way to prove the equivalence of two terms. Unification is used with other components to build sophisticated reasoning systems.

Different unification algorithms appear in the literature but we decided to implement the one discussed in class as it appears to be more straight forward and it behaves better on simple cases compared to other almost linear time algorithms (Martelli-Montanari and Escalada-Ghallab algorithms).

Discussion of ClauseForm

Resolution is the cornerstone of resolution-based reasoning systems, and as resolution requires information for the tell actions to be in the CNF or INF (depending on the system). A proper converter from normal FOL senteces to CNF (or INF) is needed. In this project, a CNF_Converter is implemented.

FOL Representation

We represented the FOL domain by having a super class called “Expression” which other classes inherit from.

The Expression class has a list of terms that hold the FOL terms that are within this expression.

The Quantifier class is a small class representing the Quantifiers. It has only one attribute which is ‘kind’ and it is either a ‘A’ (for all) or a ‘E’ (there exists).

The Term class is an abstract class that inherits from Expression, it is used for the following classes to inherit from.

The Predicate class inherits from Term class and it consists of a name and a list of terms representing the terms that this predicate holds. This Predicate class represents both Predicate symbols and function symbols for simplicity. It implements comparable interface in order to be able to equate it to other predicates/functions depending on the equivalence of both their names and their terms.

The Variable class inherits from Term class and is used to represent variables and it consists of the name of that variable. It also implements the Comparable interface in order to compare different variables according to their names.

The Constant class inherits from Term class and is used to represent constants and it consists of the name of that constant. It implements the Comparable interface in order to compare different constants according to their names.

Sentence class models how sentences are represented in FOL. Its instances consist of two main parts, a string “type” and a hash “vars”. Type is one of {"atomic", "equiv", "neg", "op", "quant"}. And the keys needed for vars depend on the type like the following

[predicate]	if type is atomic
[term1, term2]	if type is equiv
[sentence]	if type is neg
[sentence1, sentence2, op]	if type is ‘op’ where op is in [‘v’, ‘^’, ‘=>’, ‘<=>’]
[quant, variable, sentence]	if type is ‘quant’

The Sentence class doesn’t offer many functionality except the implementation of the printing.

Examples of forming a sentences

¬P(x)

```
p_x = Predicate.new('P', [Variable.new('x')])
Sentence.new('neg', {sentence: p_x.to_sentence})
```

```
∀x P(x) => Q(x)
x = Variable.new('x')
p_x = Predicate.new('P', [x]).to_sentence
q_x = Predicate.new('Q', [x]).to_sentence
impl = Sentence.new('op' {op: '=>', sentence1: p_x, sentence2: q_x})
fa = Sentence.new('quant', {quant: Quantifier.new('A'), variable: x,
sentence: impl})
```

where predicate.to_sentence is a helper method equivalent to

```
Sentence.new('atomic', {predicate: Predicate.new(name, terms)}
```

The ClauseForm module contains all functionalities needed to to convert a Sentence (as described later) to a of set of clauses in CNF including the top_level clause_form.

The Unifier class handles the unification process through its unify method which takes two terms and returns a list of substitution tuples.

Discussion of Unify implementation

The unify implementation is very straight forward and it follows the algorithm discussed in class with some minor modifications that allow it to suit our data structures and representation of FOL expressions.

The unification procedure is encapsulated in the Unifier class which consists of the following functions.

unify term1, term2
This is the method that gets called for unification of the two terms (term1 and term2). It calls the method unify1 with the parameters term1 and term2 and an empty list which will later on hold the tuples of substitution.

unify1 term1, term2, theta
This method implements the main logic of the unification procedure. It runs through different checks before either calling on unify_var or doing a recursive call on itself.

term1 and term2 represent the terms to be unified while theta represents the array of substitution tuples.

if theta is false, return false, meaning that we failed to unify in a previous step and therefore we failed to unify overall.

if term1 == term2, meaning that we succeeded in unifying them and hence we return the array of substitution tuples theta.

if term1 or term2 is a variable, return unify_var on that variable and the other term along with theta which will try to unify that variable with the other term and would either succeed or return failure.

if term1 or term2 are atoms, return false meaning that we failed to unify. This is because we can't unify atomic terms that reached this step since they are not variables and they are not equal.

if term1 and term2 do not have the same length "i.e same number of arguements" return false meaning that we failed to unify. This is because we can't unify predicates or functions that do not have the same number of arguments.

we then call a recursive call to unify1 with rest_term1, rest_term2, (unify1 head1, head2, theta) with rest_term1 and rest_term2 are the rest of these terms after the head, and head1 and head2 are the heads of term1 and term2. Meaning that we try to unify the heads and the rest of both term1 and term2.

The heads are either the names "in case of predicates" or the first elements of the terms array "in case of arrays", and the rest is either the terms "in case of predicates" or the rest of the array besides the first term "in case of arrays". This check has been made since the terms that reach this part of the code are either predicates or arrays of terms.

The second main component of the unification procedure is the unify_var method. It takes a variable x, a term e, and an array of substitution tuples theta. it first tries to find a substitution in theta having the variable x, if it finds it and it's not equal to x, we then call unify1 on that substitution variable, term e and theta. We then update the term e by applying the substitutions to it, afterwards, we check if x occurs deeply in e and if it does we return false (we don't want x to be substituted with f(x) not to get infinite substitution rules). If we pass all the we return theta with the new substitution found appended to it.

This on it's own does not guarantee deep substitutions. So after the unify1 returns a value, if this value is not false "i.e unification succeeded", we run the method anchor on this array of substitution tuples that checks for deeply nested variables and apply their substitutions if they exist, we keep repeating this until no more substitutions can be made meaning that we reached our MGU substitution set with all its variables assigned correctly.

Different tests were written to validate the correctness of the algorithm, these tests include the examples given in the project description along with other examples that test the deeply nested variable cases like x and g(f(x)) which should fail and other that should succeed.

How to run Unifier

We first construct our terms and then create a new instance of the Unifier class which takes a boolean representing the trace value "whether we should pretty print or not".
Here is an example of constructing the terms and calling the unify method, the example is based on the second example given in the project description.

```
#example 2

puts "Example 2"

a = Constant.new 'a'

y = Variable.new 'y'

z = Variable.new 'z'

u = Variable.new 'u'

f = Predicate.new 'f', [y]

p1 = Predicate.new 'P', [a, y, f]

p2 = Predicate.new 'P', [z, z, u]
```

```
unifier = Unifier.new true              # true enables trace mode, false disables it.

puts unifier.unify p1, p2
```

Discussion of CNF_Converter
The CNF_Converter module contains all functionalities needed to to convert a Sentence (as described later) to a of set of clauses in CNF. The top level method is CNF_Converter.clause_form(sentence, trace=false). Accompanied with it there are 9 main methods (along with other helper methods). Each one represent a step of the presented in class (except for one method 'build_clauses' that represent steps 8,9,10 (will discuss that in later sections)).

Each method of the first 8 methods takes a sentence and returns a new sentence after making the computation on the old_sentence (without modifying the old sentence). Names of the methods represents the corresponding step. The only exception is the last method which manipulates the clauses list in-place.

'eliminate_equiv(sentence): Sentence'

'eliminate_impl(sentence): Sentence'

'push_neg_inwards(sentence): Sentence'

'standardize_apart(sentence): Sentence'

'skolemize(sentence): Sentence'
skolem functions are represented by sk[0-10]

'discard_for_all(sentence): Sentence'

'translate_to_CNF(sentence): Sentence'

'build_clauses(sentence): lists of lists where the elements are atomic sentences (predicates) Conjunction of disjunctions ' (for step 8-10)

'standardize_clauses!(clauses): Nothing {changes in-place}'

standardize_clauses! is the only exception as it takes an array of clauses and changes it in place (this made implementation simpler).

A small number of helper methods (like make_a_new_skolem and make_a_new_name) are added to CNF_Converter module.

The code of the Converter was tested with Locs ratio of roughly 1:1

How to run
Build a sentence as described under Sentences section under FOL Representation. Then call

CNF_Converter(sentence, trace). That's all.
Example, ∃x[P(x) ∧ ∀x[Q(x) ⇒ ¬P(x)]]

```
  x = V.new('x')
  p_x = P.new('P', [x]).to_sentence
  neg_p_x = S.new('neg', {sentence: p_x} )
  q_x = P.new('Q', [x]).to_sentence

  # Q(x) ⇒ ¬P(x)
  sen3 = S.new('op', {op: '=>', sentence1: q_x, sentence2: neg_p_x})

  # ∀x[Q(x) ⇒ ¬P(x)]
  sen2 = S.new('quant' , {quant: Q.new('A'), variable: x, sentence:
sen3})

  # P(x) ∧ ∀x[Q(x) ⇒ ¬P(x)]
  sen1 = S.new('op', {op: '^', sentence1: p_x, sentence2: sen2})

  # ∃x[P(x) ∧ ∀x[Q(x) ⇒ ¬P(x)]]
  sentence = S.new('quant', {quant: Q.new('E'), variable: x,
sentence: sen1})
CNF_Converter.clause_form(sentence, true)
```

Note on pretty_printing, to avoid ambiguity, we surrounded all sentences with parenthesis (even if they are negated). This was crucial in the first steps of resolution. But it lead to many redundant parentheses by the end of the resolution. That's why we added pretty_s to sentence that prints sentences without many of the parenthesis (yet might be ambiguous if some resolution steps were not done first).

Note on the examples, in the test file, there are two methods namely (make_first_example, and make_second_example) they return the first and second examples.

Note on tests, you need to have ruby (the language) to run the project and rspec (a ruby gem) to run the tests. Running the tests is basically rspec test_file.rb

Unification sample run with pretty print

==========

Example 1

P(x, g(x), g(f(a)))

P(f(u), v, v)

****

"Unifying P(x, g(x), g(f(a))) and P(f(u), v, v)"

"Substitutions so far: []"

****

"Unifying P and P"

"Substitutions so far: []"

****

"Unifying [x, g(x), g(f(a))] and [f(u), v, v]"

"Substitutions so far: []"

****

"Unifying x and f(u)"

"Substitutions so far: []"

****

"Unifying [g(x), g(f(a))] and [v, v]"

"Substitutions so far: [[x, f(u)]]"

****

"Unifying g(x) and v"

"Substitutions so far: [[x, f(u)]]"

****

"Unifying [g(f(a))] and [v]"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"

****

"Unifying g(f(a)) and v"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"

****

"Unifying g(f(u)) and g(f(a))"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"

****

"Unifying g and g"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"

****

"Unifying [f(u)] and [f(a)]"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"

****

"Unifying f(u) and f(a)"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"

****

"Unifying f and f"

"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"
****
"Unifying [u] and [a]"
"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"
****
"Unifying u and a"
"Substitutions so far: [[x, f(u)], [v, g(f(u))]]"
****
"Unifying [] and []"
"Substitutions so far: [[x, f(u)], [v, g(f(u))], [u, a]]"
****
"Unifying [] and []"
"Substitutions so far: [[x, f(u)], [v, g(f(u))], [u, a]]"
****
"Unifying [] and []"
"Substitutions so far: [[x, f(u)], [v, g(f(u))], [u, a]]"
[[x, f(a)], [v, g(f(a))], [u, a]]
==========

Example 2
P(a, y, f(y))
P(z, z, u)
****
"Unifying P(a, y, f(y)) and P(z, z, u)"
"Substitutions so far: []"
****
"Unifying P and P"
"Substitutions so far: []"
****
"Unifying [a, y, f(y)] and [z, z, u]"
"Substitutions so far: []"
****
"Unifying a and z"
"Substitutions so far: []"
****
"Unifying [y, f(y)] and [z, u]"
"Substitutions so far: [[z, a]]"
****
"Unifying y and z"
"Substitutions so far: [[z, a]]"
****
"Unifying [f(y)] and [u]"
"Substitutions so far: [[z, a], [y, z]]"

```
****************************************
"Unifying f(y) and u"
"Substitutions so far: [[z, a], [y, z]]"
****************************************
"Unifying [] and []"
"Substitutions so far: [[z, a], [y, z], [u, f(z)]]"
[[z, a], [y, a], [u, f(a)]]
==================================================


Example 3
P(a, y, f(z))
P(z, z, u)
****************************************
"Unifying f(x, g(x), x) and f(g(u), g(g(z)), z)"
"Substitutions so far: []"
****************************************
"Unifying f and f"
"Substitutions so far: []"
****************************************
"Unifying [x, g(x), x] and [g(u), g(g(z)), z]"
"Substitutions so far: []"
****************************************
"Unifying x and g(u)"
"Substitutions so far: []"
****************************************
"Unifying [g(x), x] and [g(g(z)), z]"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying g(x) and g(g(z))"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying g and g"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying [x] and [g(z)]"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying x and g(z)"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying g(u) and g(z)"
"Substitutions so far: [[x, g(u)]]"
```

```
****************************************
"Unifying g and g"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying [u] and [z]"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying u and z"
"Substitutions so far: [[x, g(u)]]"
****************************************
"Unifying [] and []"
"Substitutions so far: [[x, g(u)], [u, z]]"
****************************************
"Unifying [] and []"
"Substitutions so far: [[x, g(u)], [u, z]]"
****************************************
"Unifying [x] and [z]"
"Substitutions so far: [[x, g(u)], [u, z]]"
****************************************
"Unifying x and z"
"Substitutions so far: [[x, g(u)], [u, z]]"
****************************************
"Unifying g(u) and z"
"Substitutions so far: [[x, g(u)], [u, z]]"
****************************************
"Unifying [] and []"
"Substitutions so far: false"
false
```

CNF_Convter.clause_form sample run with pretty print

Sentece is ∃x[(P(x)) ^ (∀x[(Q(x)) => (¬(P(x)))])]
---------- Step 1 Eliminate <=> ----------
∃x[(P(x)) ^ (∀x[(Q(x)) => (¬(P(x)))])]
---------- Step 2 Eliminate => ----------
∃x[(P(x)) ^ (∀x[(¬(Q(x))) v (¬(P(x)))])]
---------- Step 3 push ¬ inwards ----------
∃x[(P(x)) ^ (∀x[(¬(Q(x))) v (¬(P(x)))])]
---------- Step 4 standardize apart ----------
∃x[(P(x)) ^ (∀z[(¬(Q(z))) v (¬(P(z)))])]
---------- Step 5 skolemize ----------
(P(sk())) ^ (∀z[(¬(Q(z))) v (¬(P(z)))])
---------- Step 6 discard ∀ ----------
(P(sk())) ^ ((¬(Q(z))) v (¬(P(z))))
---------- Step 7 Translate into CNF ----------
(P(sk())) ^ ((¬(Q(z))) v (¬(P(z))))
---------- Step 8-10 get clauses ----------
%%%%%%%%%%%%
{P(sk())},
{¬Q(z), ¬P(z)},
%%%%%%%%%%%%
---------- Step 11 standardize clauses ----------
%%%%%%%%%%%%
{P(sk())},
{¬Q(z), ¬P(z)},
%%%%%%%%%%%%

Example 2
Sentence is ∀x[(P(x)) <=> ((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))]))]
---------- Step 1 Eliminate <=> ----------
∀x[((P(x)) => ((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))]))) ^ (((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))])) => (P(x)))]
---------- Step 2 Eliminate => ----------
∀x[((¬(P(x))) v ((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))]))) ^ ((¬((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))]))) v (P(x)))]
---------- Step 3 push ¬ inwards ----------
∀x[((¬(P(x))) v ((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))]))) ^ (((¬(Q(x))) v (∀y[(¬(Q(y))) v (¬(R(y, x)))])) v (P(x)))]
---------- Step 4 standardize apart ----------
∀x[((¬(P(x))) v ((Q(x)) ^ (∃y[(Q(y)) ^ (R(y, x))]))) ^ (((¬(Q(x))) v (∀z[(¬(Q(z))) v (¬(R(z, x)))])) v (P(x)))]
---------- Step 5 skolemize ----------
∀x[((¬(P(x))) v ((Q(x)) ^ ((Q(sk(x))) ^ (R(sk(x), x))))) ^ (((¬(Q(x))) v (∀z[(¬(Q(z))) v (¬(R(z, x)))])) v (P(x)))]
---------- Step 6 discard ∀ ----------
((¬(P(x))) v ((Q(x)) ^ ((Q(sk(x))) ^ (R(sk(x), x))))) ^ (((¬(Q(x))) v ((¬(Q(z))) v (¬(R(z, x))))) v (P(x)))
---------- Step 7 Translate into CNF ----------
((¬(P(x))) v (Q(x))) ^ (((¬(P(x))) v (Q(sk(x)))) ^ ((¬(P(x))) v (R(sk(x), x)))) ^ ((¬(Q(x))) v ((¬(Q(z))) v (¬(R(z, x))))) v (P(x))
---------- Step 8-10 get clauses ----------
%%%%%%%%%%%%
{¬P(x), Q(x)},
{¬P(x), Q(sk(x))},
{¬P(x), R(sk(x), x)},
{¬Q(x), ¬Q(z), ¬R(z, x), P(x)},
%%%%%%%%%%%%
---------- Step 11 standardize clauses ----------
%%%%%%%%%%%%
{¬P(x), Q(x)},
{¬P(z), Q(sk(z))},
{¬P(y), R(sk(y), y)},
{¬Q(w), ¬Q(v), ¬R(v, w), P(w)},
%%%%%%%%%%%%