# Project 2 report

Samy Shehata 22-3798
Omar Mahmoud 22-1534
Karim Tarek 22-0466
November 30, 2014

## 1 Unification

### 1.1 Overview

Unification is the process of determining if two first order logic expressions are equivalent. This entails finding a most general unifier that enforces the least number of bindings for variables to constants necessary to make the two expressions equivalent. By applying a substitution of variables with constants according to the MGU, we reach a common instance of the two expressions.

### 1.2 Representation

For unification, FOL expressions should be represented using lists. A list represents a function or a predicate. First element of the list is the function symbol, followed by its arguments. Arguments can either be lists (functions) or atoms. Atoms are either variables or constants, both represented with their own data types. A variable or a constant is defined as a *struct* with field *sym*.
Ex:

$$P(a, y, f(y)) -> (list\ (make-predicate)\ \#\backslash P\ \#\backslash y\ (list\ (make-predicate)\ \#\backslash f\ \#\backslash y))$$

### 1.3 Implementation

FUNCTION UNIFY
**Input**

- E1: first expression listified
- E2: second expression listified
- visual: boolean

**Output**

- mu: A list of bindings ( term / variable )

Tries to unify $E1$ and $E2$ and returns the MGU if successful. Returns FAIL on failure. If $viusal$ is true, the solution is traced step by step.

FUNCTION UNIFY1

- E1: first expression listified
- E2: second expression listified
- mu: A list of bindings term / variable
- visual: boolean

**Output**

- mu: A list of bindings ( term / variable )

Recursive helper function to Unify, carries out the bulk of unification. Takes the two expressions and mu which is the MGU so far. Unifies the first two elements in $E1$ and $E2$, updates the MGU and then continues on the rests of the two expressions. This function follows the algorithm described in class.

FUNCTION UNIFY-VAR

- x: an FOL variable
- e: an FOL term to be unified with x
- mu: A list of bindings term / variable
- visual: boolean

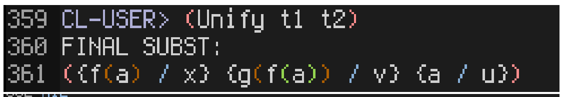
**Output**

- mu: A list of bindings ( term / variable )

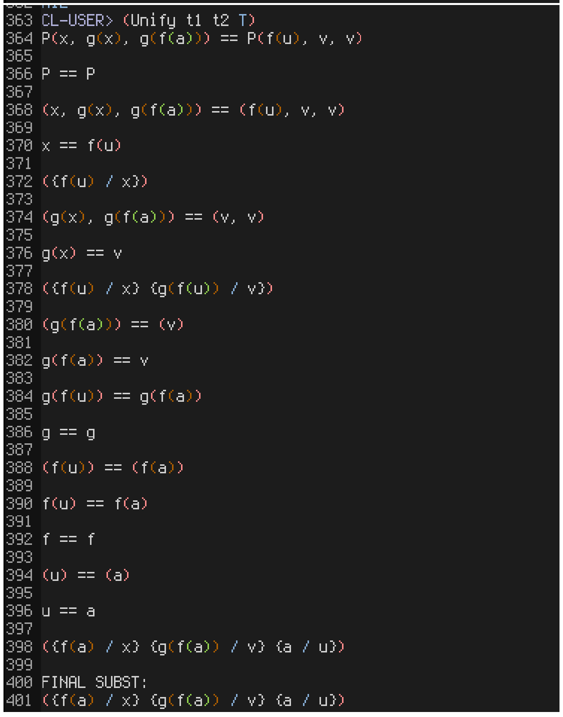
Helper function to $unify1$, called when a variable is encountered in one of the expressions. Takes the variable x, the corresponding term $e$ from the other expression and mu (the MGU so far). If $x$ is not bound according to mu the $x$ is bound to $e$ and mu is updated, otherwise $e$ is unified with the binding of $x$. unify-var fails if $x$ needs to be bound to a term that also contains $x$.

Smaller helper functions are documented in the code.

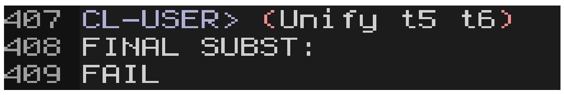
## 1.4 Sample Runs



```
CL-USER> (Unify t1 t2)
FINAL SUBST:
({f(a) / x} {g(f(a)) / v} {a / u})
```



```
CL-USER> (Unify t1 t2 T)
P(x, g(x), g(f(a))) == P(f(u), v, v)

P == P

(x, g(x), g(f(a))) == (f(u), v, v)

x == f(u)

({f(u) / x})

(g(x), g(f(a))) == (v, v)

g(x) == v

({f(u) / x} {g(f(u)) / v})

(g(f(a))) == (v)

g(f(a)) == v

g(f(u)) == g(f(a))

g == g

(f(u)) == (f(a))

f(u) == f(a)

f == f

(u) == (a)

u == a

({f(a) / x} {g(f(a)) / v} {a / u})

FINAL SUBST:
({f(a) / x} {g(f(a)) / v} {a / u})
```



```
CL-USER> (Unify t5 t6)
FINAL SUBST:
FAIL
```

## 1.5 How To Run

Loading the runme.lisp file will compile the code and show a test run with the examples from the project description (please ignore compilation style warnings). To load the file in Top-level lisp REPL:

```
(load ''runme.lisp'')
```

At this point, the example inputs are defined in variables $t1$ - $t6$ and can be used as:

```
(Unify t1 t2)
```

for normal mode and

```
(Unify t1 t2 T)
```

for trace mode. In trace mode, you can press (Enter key) to advance to the next step.

# 2 Clause Form

## 2.1 Overview

Clause form is a representation of first order logic statements as a conjunction of clauses, where each clause is a disjunction of literals.

## 2.2 Representation

For ClauseForm, FOL expressions are represented as lists, where the first element of the list is an operator and the rest of the list are the operands. Each operator is represented as with their own data type using structs. The defined operators are (land, lor, limpl, leq, forall, there-exists and lnot) for (logical and, or, implication, equivelance, for all, there exists and not). Literals are also represented as a list with struct $latom$ as the first element. The struct holds the function symbol and the literal arguments form the rest of the list. Ex:

$$\forall x[P(x)] = (list\ (make-forall\ :sym\ x)\ (list\ (make-latom\ :sym\ P)\ x))$$

## 2.3 Implementation

Function ClauseFrom

- expr: FOL expression listified
- visual: boolean

**Output**

- expr: FOL expression listified

Takes FOL expression and applies the clause form steps then returns the clause form expression. If visual is true, the solution is traced step by step.

Function remove-eql

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively looks for expressions of the form $(list\ leq\ a\ b)$ and changes them to $(list\ land\ (list\ limpl\ a\ b)\ (list\ li$

Function remove-impl

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively looks for expressions of the form $(list\ limpl\ a\ b)$ and changes them to $(list\ lor\ (list\ not\ a)\ b)$

FUNCTION PUSH-NOT

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively looks for expressions of the form $(list\ not\ a)$ and applies the not operation on $a$ recursively until a literal is hit. The not operations changes ∀ to ∃ and the reverse, $(list\ land\ a\ b)$ to $(list\ lor\ (list\ lnot\ a)(list\ lnot\ b)$. The reverse is done to ors.

FUNCTION STANDARISE-APART

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively looks for expressions with quantifiers, the symbol of the quantifiers is added to a list of used variables. If found again in another quantifier, all variables within the scope of the quantifier are renamed.

FUNCTION SKOLEMIZE-APART

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively looks for expressions with ∃, the variable within the scope of the quantifier are renamed to skolem variables.

FUNCTION DISCARDED-APART

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively looks for expressions of the form $(list\ (forall) a)$, and changes them to $a$.

FUNCTION FLATTEN-APART

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively changes all nested expressions to a top *and* expression, with all operands being *or* expressions of literals.

FUNCTION FLATTEN-APART

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Recursively changes all nested expressions to a top *and* expression, with all operands being *or* expressions of literals.

FUNCTION CLAUSES

- expr: FOL expression listified

**Output**

- expr: FOL expression listified

Takes a cnf expression and returns a list of lists where the inner lists are the disjunctions and the outer list is the conjunction,

- expr: FOL expression listified

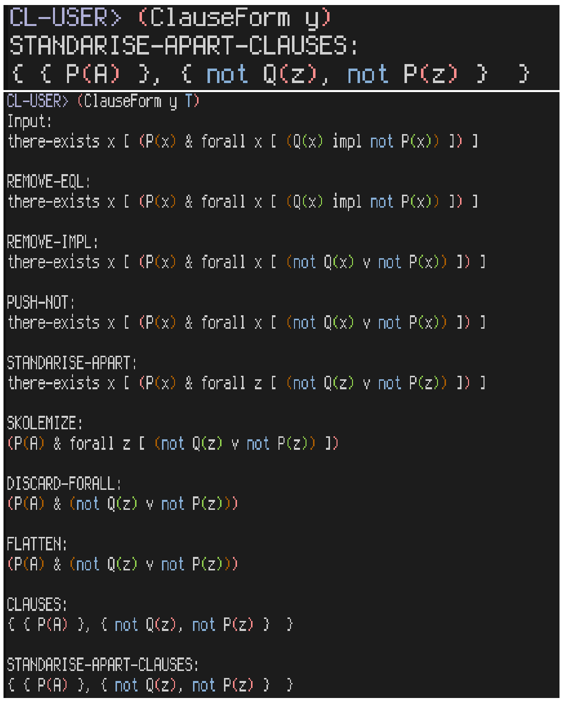
**Output**

- expr: FOL expression listified

Takes a list of lists and renames the variables within them such that no variable is shared among two lists.

Smaller helper functions are documented in code.

## 2.4 Sample Runs



```
CL-USER> (ClauseForm y)
STANDARISE-APART-CLAUSES:
{ { P(A) }, { not Q(z), not P(z) }  }
```

```
CL-USER> (ClauseForm y T)
Input:
there-exists x [ (P(x) & forall x [ (Q(x) impl not P(x)) ]) ]

REMOVE-EQL:
there-exists x [ (P(x) & forall x [ (Q(x) impl not P(x)) ]) ]

REMOVE-IMPL:
there-exists x [ (P(x) & forall x [ (not Q(x) v not P(x)) ]) ]

PUSH-NOT:
there-exists x [ (P(x) & forall x [ (not Q(x) v not P(x)) ]) ]

STANDARISE-APART:
there-exists x [ (P(x) & forall z [ (not Q(z) v not P(z)) ]) ]

SKOLEMIZE:
(P(A) & forall z [ (not Q(z) v not P(z)) ])

DISCARD-FORALL:
(P(A) & (not Q(z) v not P(z)))

FLATTEN:
(P(A) & (not Q(z) v not P(z)))

CLAUSES:
{ { P(A) }, { not Q(z), not P(z) }  }

STANDARISE-APART-CLAUSES:
{ { P(A) }, { not Q(z), not P(z) }  }
```

## 2.5 How To Run

Loading runme.lisp also compiles the code for clause form and creates two variables $y$ and $z$ for example inputs from the project description.

```
(load ‘‘runme.lisp’’)
```

Then clause form can be called as

```
(ClauseFrom y)
```

for normal mode, and

```
(ClauseForm y T)
```

for trace mode. In trace mode you can press (Enter key) to advance to the next step in the solution.