First Order Logic 1

Prof. Mauro Gaspari
Dipartimento di Informatica Scienze e Ingegneria (DISI)
mauro.gaspari@unibo.it

Improving expressive power

- Propositional logic as several limitations as KR language: atomic formulas are strings without an internal structure.
- Thus, it is not possible to represent relations between components of the atomic formulas:

$$
\begin{aligned}
HCl \wedge NaOH &\rightarrow NaCl \wedge H_2O \\
C \wedge O_2 &\rightarrow CO_2 \\
CO_2 \wedge H_2O &\rightarrow H_2CO_3
\end{aligned}
$$

Limitations of PL

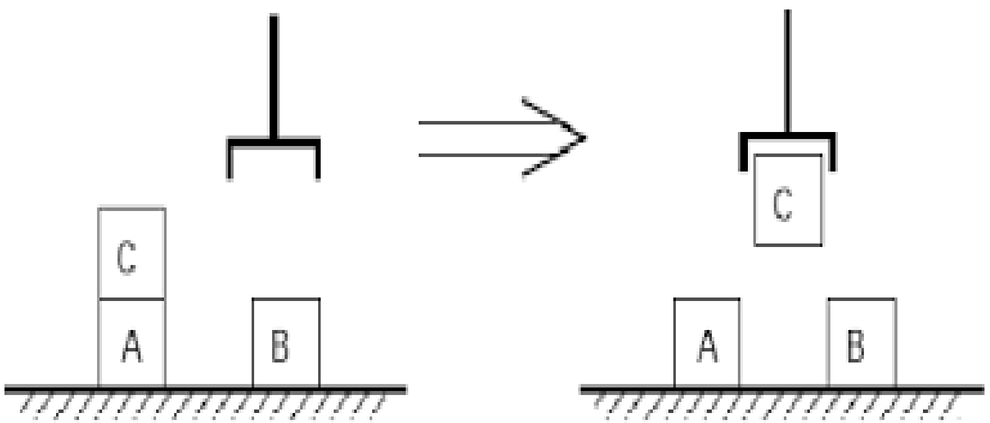
- Representing block world:

ON_A_FLOOR

ON_B_FLOOR

ON_C_A

MOVE_C



Limitation of PL

- We can’t express the fact that when we move block MOVE_C, it is the same block that is on bock A, according to the proposition ON_C_A.
- Using mnemonic names for atomic formulas has no influence on what these formulas represents and on their semantics:
    - MOVE_C ≡ Q
    - ON_C_A ≡ P

Limitations of PL

- PL does not have mechanisms to represent time.
- PL does not have mechanisms to represent space notions.
- It is not possible to represent the structure of objects.
- It is not possible to express general lows that govern relationships between objects of the worls.
- For example, it is not possible to organize objects in categories.

What do we need?

- A more useful language would be one that could refer to **objects** in the world as well as to **propositions** aboult the world.
- We need a language that has names for objects and names for propositions:
    - $ON\_B\_C \rightarrow \neg CLEAR\_C$
    - $on(x,y) \rightarrow \neg clear(y)$

First Order Logic (FOL)

- Also known as **Predicate Calculus.**
- It can be considered a **standard** for knowledge representation.
- **Syntax**:
  - An infinite set of object constants.
  - An infinite set of variables.
  - An infinite set of functional symbols of all arieties.
  - An infinite set of predicates symbols of all arieties.
  - Connectives: ∧,∨,→,¬
  - Quantifier symbols: ∀, ∃

The arity is a non negative integer indicating the number arguments.

Syntax

- **Terms**:
  - A term is a constant or a variable
  - A term is a functional symbol of arity n followed by n terms in parentheses and separated by commas.
- Examples: bob

  x

  room(R3.4, engineering,100)

We use x,y,z...
for variables all the
other symbols represents
constants and numbers.

Syntax

- **wffs**:
  - Atomic formulas: predicare symbols of arity n followed by n Terms in parentheses are wffs.
  - IF $F1$ and $F2$ are wffs THEN $F1 \wedge F2$, $F1 \vee F2$, $F1 \rightarrow F2$, $\neg F1$ are wffs.
  - IF F is a wff that contain variable x THEN $\forall x\ F(x)$, $\exists x\ F(x)$ are wffs.

Example of wffs

- $\forall x\ (\text{professor}(x) \rightarrow \text{brilliant}(x))$
- $\forall x\ (\text{student}(x) \rightarrow \exists y\ \text{supervises}(y, x))$
- $\forall x\ y\ (\text{student}(x) \wedge \text{college}(y) \wedge \text{member}(x, y)$
  $\rightarrow \text{member}(\text{tutor}(x), y))$
- $\forall x\ y\ (\text{banker}(x) \wedge \text{bedder}(y) \rightarrow$ $\text{greather}(\text{income}(x),\text{income}(y)))$
- $\forall P\ [P(0) \wedge \forall k\ (P(k) \rightarrow P(k + 1)) \rightarrow \forall n\ P(n)]$ this is not a FOL formula, why?

Semantics

- Notion of **interpretation**:
  - It is associated to the world we are modelling with our KB.
  - In other words an intepretations depends from the application.
  - A constant or a predicate might have different meaning in different applications.
  - Intuitively, the semantic of a formula is a function of the interpretation of terms and predicates.

Interpretation

- An interpretation of all the objects (constants) in the world.
- An interpretation of functional symbols used in terms.
- An interpretation of predicates that for each predicate establish the set of terms that make it true.
- In other word an interpretation indicates all the atomic formulas that are true.

Interpretation

- Starting fron an interpretation, we can compute the semantic of all the FOL formulas as follows:
- If a formula F includes logic connectives as follows: $F1 \wedge F2$, $F1 \vee F2$, $F1 \rightarrow F2$, $\neg F1$ we use truth table to determine the semantics of F.
- Let F be $\exists x\ F(x)$, F it is true if exists an object obj (constant) in the domain such that F(obj) is true.
- Let F be $\forall x\ F(x)$, F it is true if for all the objects obj (constants) in the domain F(obj) is true.

Definitions and notation

- A **model** for a KB is an interpretation in which each formula in the KB is True.
- A KB is **satisfiable** iff exists an interpretation which is a model for it.
- A KB is **unsatisfiable** (or, contradictory) if it is false in every interpretation.
- F is a **logical conseguence** of a KB, written KB $\models$ F means that whenever KB is True, so is F; in other words, all models of P are also models of Q.
- Two formulas F1, F2 are logically equivalent (F1 $\equiv$ F2) if they have the same semantics for all the Interpretations.

Example

- $P(a) \wedge \neg P(b)$ is satisfiable:
    - Consider the interpretation: I[a] = Paris, I[b] = London, P(Paris) = true
- $\forall x\ y\ (P(x) \wedge \neg P(y))$ is unsatisfiable:
    - because it requires P(x) to be both true and false for all x.
- The formula $(\exists x\ P(x)) \rightarrow P(c)$ is satisfiable.
    - Consider the interpretation I[c] = 0 and P(0) = true
    - However if we modify this interpretation by making I[c] = 1 then the formula no longer holds. Thus it is satisfiable but not valid.
- The formula $(\forall x\ P(x)) \rightarrow (\forall x\ P(\ f\ (x)))$ is valid.
    - Given an interpretation where $\forall x\ P(x)$ holds then P(x) holds for all x in the domain, thus also for f(x).
- The formula $\forall x\ y\ x = y$ is satisfiable but not valid:
    - It is true in every domain that consists of exactly one element.

Reasoning in FOL

- We can effectively say that FOL as a **good expressive power**.
- The inference engines that can be used with FOL are similar to those used with PL.
- However, the consequences of the enhanced expressive power have an important impact on the properties of these engines.

Reasoning in FOL

- Two are the main concerns:
- Inference steps are more complex: we need **unification and substitutions** to perform them.
- The set of all the possible clauses is not finite any more. Thus **termination becomes an issue**.

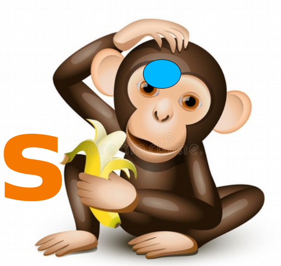
Logic Variables

- Variables used in FOL are called **logic variables.**
- Logic variables **are bound by quantifiers**.
- An occurrence of a variable x in a formula is **bound** if it is contained within a subformula of the form ∀x A or ∃x A.
- An occurrence of a variable is **free** if it is not bound.
- A **closed formula** is one that contains no free variables.
- A **ground formula** is one that contains no variables at all.
- **Bound variables can be renamed** without changing the semantic of a formula.

Examples

- $\forall x \exists y\, P(x, y, z)$: variables x and y are bound while z is free.
- $\forall x \exists y\, loves(x,y)$: is a closed formula.
- $loves(john, mary)$: is a ground formula.
- $(\forall x\, P(x)) \rightarrow (\forall x\, P(f(x)))$: is a closed formula and the second instance of variable x can be renamed to y or to another variable, in other words the second instance of x is a different variable.:
    - $(\forall x\, P(x)) \rightarrow (\forall y\, P(f(y)))$

Inference with variables



- Considering the following knowledge base:

  $\forall x\ person(x) \rightarrow likes(x, sun)$

  $person(john)$

- What can we do?

- Universal elimination: $\dfrac{\forall x\ P(x)}{P(a)}$ where a is a ground term.

Universal elimination

Modus ponens

$$\dfrac{\dfrac{\forall x\ person(x) \rightarrow likes(x, sun)}{person(john) \rightarrow likes(john, sun)} \qquad person(john)}{likes(john, sun)}$$

Unification

- The modus pones can be applied only if we have a fact in the KB that is (or can be made) identical to the right hand side of an implication.
- **Unification** is the operation of finding a common instance of two terms.
- A **substitution** is a finite set of replacements for the variables of one or more terms.
- A substitution θ is a **unifier** of two terms t1 and t2 if $t1\theta = t2\theta$.
- The substitution θ is **more general** than φ if $\varphi = \theta \circ \sigma$ for some substitution σ.
- A substitution θ is a **most general unifier (MGU)** of two terms t1 and t2 if θ unifies t1 and t2 and θ is more general than every other unifier.

Examples

- f(x, b) and f(a, y) unify and have the common instance f(a, b)

  unifier: [x/a,y/b}
- f(x, x) and f(a, b) do not unify.
- p(x) and p(y) unify with unifier [x/a,y/b] and many others.

  The MGU is [x/y]
- g(g(x)) and g(y) unify with MGU [y/g(x)]
- What about g(f(x)) and g(x) ?

FOL in LiSP

- Logic variables: Lisp Symbols ?x.?y,?z
- Constant, predicates, functions: Lisp Symbols
- Terms: (function t1 ... tm)
- Atomic formulas: (predicate t1 … tn)
- (AND F1 F2), (OR F1 F2), (=> F1 F2), (NOT F)
- (FORALL (X) F), (EXISTS (X) F)

Implementing Unification

- (UNIFY term-l term-2 &optional env) is a top level unification function. Returns a substituition or :fail if t1 and t2 do not unify. term-1 and term-2 have the form (p t1 … tn)

```
(defun unify (t1 t2 &optional env)
        (multiple-value-bind (flag new-env)
                                     (unify-r t1 t2 env)
        (if flag new-env :fail)))
```

It returns :fail for failure because NIL also represents the empty env

- (MULTIPLE-VARIABLE-BIND Variables-list BODY) Binds a list of vars to a list of values returned by BODY. No checking is performed.

- (unify-r t1 t2 env) is the real implementation.

Implementing Unification

- A substitution (env) is an associative list having the form:

  ((?var1 . Bind1) … (?varn . Bindn))

```
(defun bind (var thing &optional env)
        (cons (cons var thing) env))
```

- BINDING-OF ---Returns 2 values: <flag, binding>. The flag says whether or not var is bound in env.

```
(defun binding-of (var env)
    (let ((pair (assoc var env :test #'eq)))
        (and pair (values t (cdr pair)))))
```

- Recognize variables:

```
(defun variable? (pattern)
    (and (symbolp pattern)
        (char= (elt (symbol-name pattern) 0) #\?)))
```

- After **&optional** an optional arg follows.
- **assoc** retrieve a pair from an associative list.
- **values** returns two values
- **elt** returns the ith element of a string.

Dereferencing

- This function implements dereferining when a variable is bound to another variable it follows the link. It returns two values flag and binding.

```
(defun lookup (var env)
    (multiple-value-bind (flag binding)  (binding-of var env)
        (when flag
            (cond ((variable? Binding)
                        (multiple-value-bind (flag binding-2)
                                                    (lookup binding env)
                          (if flag   (values t binding-2)
                              (values t binding))))
                      ((ground-termp binding)  (values t binding))
                      (t ;;;it must be a list
                         (values t  (cons (lookup-or-self (car binding) env)
                                                (lookup-or-self (cdr binding) env))))))))
```

- ground-termp returns true if a term is ground.
- lookup-or-self returns a the variable if it is not bound.

Implementi Unification

```
(defun unify-r (term-a term-b &optional env)
    (cond  ((equal term-a term-b) (values t env))
           ;;If they are equal, they match without substitutions
            ((variable? Term-a)  ;;term  is a variable.
             (unify-var term-a term-b env))
           ((variable? term-b)
            (unify-var term-b term-a env))
           ((and (numberp term-a)(numberp term-b))
                  (values (= term-a term-b) env))
           ((or (atom term-a)(atom term-b)) nil)
            ;;If one arg is an atom we fail
           (t (multiple-value-bind (flag new-env)
             (unify-r (car term-a)(car term-b) env)
                 (when flag
                      (unify-r (cdr term-a) (cdr term-b) new-env))))))
```

In all the other cases it calls recursive unification on the first and (if succeeds) on rest of the list.

```
(defun unify-var (var term env)
      (multiple-value-bind (flag-1 binding-1)  (lookup var env)
          (if flag-1 ;;;var is bound
              (if (variable? term) ;;var is bound and term is a var
                 (multiple-value-bind (flag-2 binding-2)  (lookup term env)
                    (if flag-2  ;;both var and term are bound
                          (unify-r binding-1 binding-2 env)
                          ;;var is bound and term is not let's match term
                          ;;to binding-1 unless term = binding-1
                          (if (equal term binding-1)  (values t env)
                             (values t  (bind term binding-1 env)))))
                  ;;term is not a variable and var is bound
                  (unify-r  binding-1 term env ))
              (if (variable? term) ;;var is unbound and term is a variable
                  (multiple-value-bind (flag-2 binding-2) (lookup term env)
                     (if flag-2  ;;term is bound, var is unbound as before
                         (if (equal var binding-2) (values t env)
                             (values t (bind var binding-2 env)))
                         ;;both term and var are unbound
                         (values t  (if (internal-var? term) (bind term var env)
                                        (bind var term env)))))
                         ;;var is unbound, term is not a variable
                  (values t (bind var term env))))))
```

lookup follows binding in env dereferencing

occur check

Implementing unification

- The implementation of unification is a crucial point for FOL engines.
- The occur check test is in general omitted.
- Unification binds logic variables, these binds remain active until the end of the reasoning process and cannot be changed, unless they are undone when backtraking occurs, in the case of failure.
- Efficient (stack based) data structures for representing logic variables and for undoing their bindings during backtracking are needed.
- In some situations unification instructions can be compiled in abstract machine code (WAM) or native code.