LU-17: INFERENCE IN FIRST ORDER LOGIC

LU Objectives	
To explain differed inference mechanisms	
To study first order resolution technique	
LU Outcomes	CO : 3
Apply inference rules	
Implement automated theorem provers using resolution mechanisms	

Outline

- Reducing first-order inference to propositional inference
- Unification
- Generalized Modus Ponens

Universal instantiation (UI)

- Every instantiation of a universally quantified sentence is entailed by it:
-

$$\frac{\forall v\ \alpha}{\mathrm{Subst}(\{v/g\}, \alpha)}$$

for any variable *v* and ground term *g*

- E.g., $\forall x\ King(x) \land Greedy(x) \Rightarrow Evil(x)$ yields:
-
- To write out the inference rule formally, we use the notion of **substitutions**

  $King(John) \land Greedy(John) \Rightarrow Evil(John)$

  $King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard)$

  $King(Father(John)) \land Greedy(Father(John)) \Rightarrow Evil(Father(John))$

- For example, the three sentences given earlier are obtained with the substitutions {x/John}, {x/Richard }, and {x/Father (John)}.

Existential instantiation (EI)

- For any sentence α, variable v the variable is replaced by constant symbol k that does not appear elsewhere in the knowledge base:
-

$$\frac{\exists v\ \alpha}{\mathrm{Subst}(\{v/k\}, \alpha)}$$

- E.g., $\exists x\ Crown(x) \wedge OnHead(x,John)$ we can infer

$$Crown(C_1) \wedge OnHead(C_1,John)$$

provided C_1 is a new constant symbol, called a Skolem constant

Reduction to propositional inference

Suppose the KB contains just the following:

$\forall$x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)
King(John)
Greedy(John)
Brother(Richard,John)

- Instantiating the universal sentence in all possible ways, we have:
  King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
  King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)
  King(John)
  Greedy(John)
  Brother(Richard,John)

- The new KB is propositionalized: proposition symbols are
-

King(John), Greedy(John), Evil(John), King(Richard), etc.

Reduction contd.

- Every FOL KB can be propositionalized so as to preserve entailment
-
- (A ground sentence is entailed by new KB iff entailed by original KB)
-
- Idea: propositionalize KB and query, apply resolution, return result
-
- Problem: with function symbols, there are infinitely many ground terms,
  - e.g., *Father*(*Father*(*Father*(*John*)))
  -

A first-order inference rule

- E.g:
  $\forall x$ King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)
  King(John)
  $\forall y$ Greedy(y)
  Brother(Richard,John)

- We would still like to be able to conclude that Evil(John), because we know that John is a king (given) and John is greedy (because everyone is greedy).
- We have to find a substitution both for the variables in the implication sentence and for the variables in the sentences that are in the knowledge base.
- By applying the substitution {x/John, y/John} to the implication premises King(x) and Greedy(x) and the knowledge-base sentences King(John) and Greedy(y) will make them identical. Thus, we can infer the conclusion of the implication.

- This inference process can be captured as a single inference rule is called **Generalized Modus Ponens**

A first-order inference rule

- For atomic sentences pi', pi , and q, where there is a substitution θ such that SUBST(θ, pi')= SUBST(θ, pi), for all i,

$$\frac{p_1{}',\ \ p_2{}',\ \ \ldots,\ \ p_n{}',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\mathrm{SUBST}(\theta, q)}\ .$$

- There are n+1 premises to this rule: the n atomic sentences pi' and the one implication.
- The conclusion is the result of applying the substitution θ to the consequent q.

A first-order inference rule

- Ex:

  p1' is King(John) p1 is King(x)

  p2' is Greedy(y) p2 is Greedy(x)

  θ is {x/John, y/John} q is Evil(x)

  SUBST(θ, q) is Evil(John)
- This Generalized Modus Ponens is a sound inference rule.
- For any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ,

  $$p \models \text{SUBST}(\theta, p)$$

- Thus, from p1 ', . . . , pn' we can infer

  $\text{SUBST}(\theta, p1') \wedge \ldots \wedge \text{SUBST}(\theta, pn')$

  and from the implication $p1 \wedge \ldots \wedge pn \Rightarrow q$ we can infer

  $\text{SUBST}(\theta, p1) \wedge \ldots \wedge \text{SUBST}(\theta, pn) \Rightarrow \text{SUBST}(\theta, q)$

A first-order inference rule

- θ in Generalized Modus Ponens is defined so that

  SUBST(θ, pi’)= SUBST(θ, pi), for all i;
- Therefore the first of these two sentences matches the premise of the second exactly. Hence, SUBST(θ, q) follows by Modus Ponens.
- Generalized Modus Ponens is a **lifted version of Modus Ponen.** It raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.

Problems with propositionalization

- Propositionalization seems to generate lots of irrelevant sentences.

- E.g., from:
-
-

$\forall$x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)
King(John)
$\forall$y Greedy(y)
Brother(Richard,John)

- it seems obvious that *Evil*(*John*), but propositionalization produces lots of facts such as *Greedy*(*Richard*) that are irrelevant
-

- With *p* *k*-ary predicates and *n* constants, there are $p \cdot n^k$ instantiations.
-

Unification

- Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification**
- Unification and is a key component of all first-order inference algorithms.
- The UNIFY algorithm takes two sentences and returns a **unifier for** them if one exists:

  UNIFY(p, q)=θ where SUBST(θ, p)= SUBST(θ, q) .

Unification

- Ex: We have a query
  AskVars(Knows(John, x))
- This query means "whom does John know?". Answers to this query can be obtained by finding all sentences in the knowledge base that unify with Knows(John, x).
- The results of unification with four different sentences that are in the knowledge base:
  - UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}
  - UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}
  - UNIFY(Knows(John, x), Knows(y, Mother (y))) = {y/John, x/Mother (John)}
  - UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail .
- The last unification fails because x cannot take on the values John and Elizabeth at the same time.

Unification

- WKT that Knows(x, Elizabeth) means "Everyone knows Elizabeth,"
- So we *should be able to infer that John knows Elizabeth.*
- The problem arises only because the two sentences happen to use the same variable name, x.
- The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes.
- For example, we can rename x in Knows(x, Elizabeth) to x17 (a new variable name) without changing its meaning. Now the unification becomes

  UNIFY(Knows(John, x), Knows(x17, Elizabeth)) =

  {x/Elizabeth, x17/John} .

Unification

- There is one more complication.
- Generally UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier.
- For example, UNIFY(Knows(John, x), Knows(y, z)) could return {y/John, x/z} or {y/John, x/John, z/John}.
- The first unifier gives Knows(John, z) as the result of unification, whereas the second gives Knows(John, John).
- The first unifier is *more general than the second, because it places fewer restrictions on the values of the variables.*
- *F*or every unifiable pair of expressions, there is a single **Most General Unifier (MGU)** that is unique up to renaming and substitution of variables.

The unification algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
   inputs: x, a variable, constant, list, or compound
           y, a variable, constant, list, or compound
           θ, the substitution built up so far

   if θ = failure then return failure
   else if x = y then return θ
   else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
   else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
   else if COMPOUND?(x) and COMPOUND?(y) then
       return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
   else if LIST?(x) and LIST?(y) then
       return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
   else return failure
```

The unification algorithm

function UNIFY-VAR(var, x, θ) **returns** a substitution
  inputs: var, a variable
    x, any expression
    θ, the substitution built up so far

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Unification Algorithm

- The process of Unification is simple:
  - recursively explore the two expressions simultaneously “side by side,” building up a unifier along the way,
  - It fails if two corresponding points in the structures do not match.
- There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term;
- if it does, the match fails because no consistent unifier can be constructed.
- For example, S(x) can’t unify with S(S(x)).
- This is called **occur check ,** it makes the complexity of the entire algorithm quadratic in the size of the expressions being unified

Storage and retrieval

- Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions
  - STORE(s) stores a sentence s into the knowledge base
  - FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base.
- The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works.

Storage and retrieval

- FETCH can be more efficient by ensuring that unifications are attempted only with sentences that have *some chance of unifying.*
- *For example, there is no point in trying* to unify Knows(John, x) with Brother (Richard, John). We can avoid such unifications by **indexing the facts in the knowledge base.**
- **Predicate indexing** puts all the Knows facts in one bucket and all the Brother facts in another. The buckets can be stored in a hash table for efficient access.
- Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Not suitable for a predicate has many clauses.

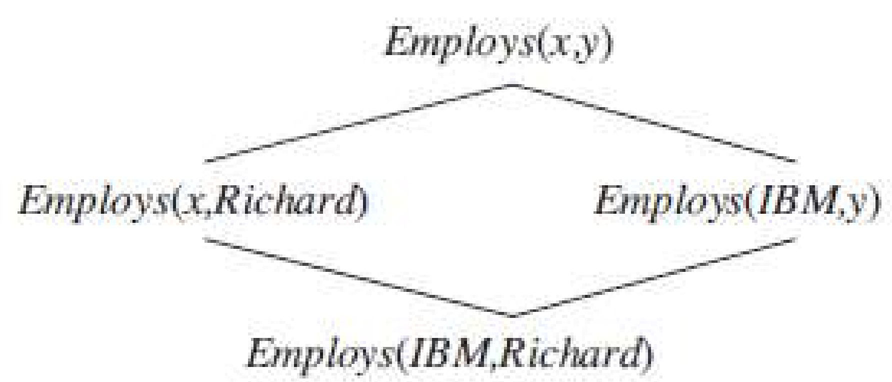
Storage and retrieval

- Ex: A predicate

  Employs(x, y)
- would have a very large bucket with perhaps millions of employers and tens of millions of employees.
- For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key
- For other queries, such as Employs(IBM , y), we would need to have indexed the facts by combining the predicate with the first argument.
- So facts can be stored under multiple index keys, so that they can be instantly accessible to various queries that they might unify with.

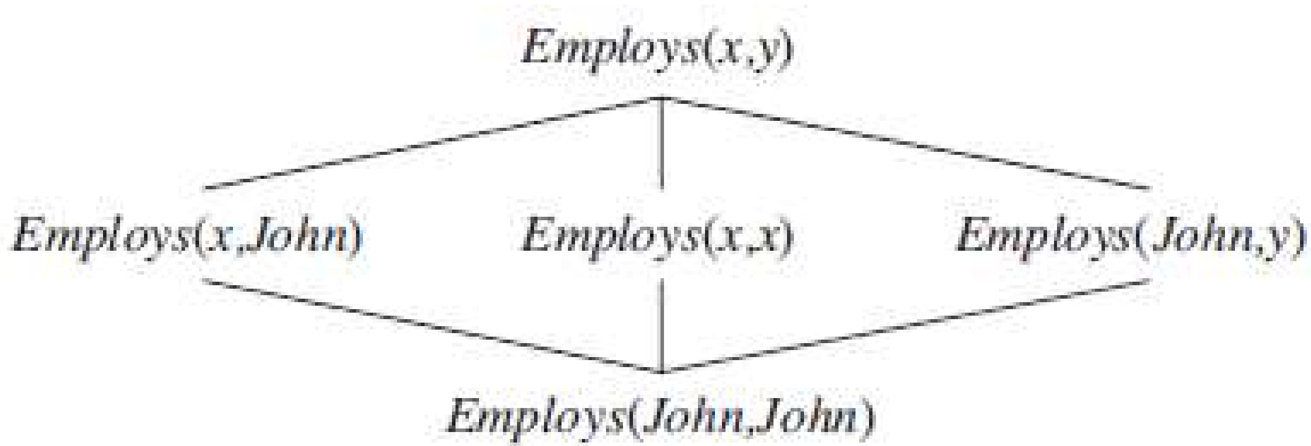
Storage and retrieval

- For the fact Employs(IBM ,Richard), the queries are
  - Employs(IBM ,Richard) Does IBM employ Richard?
  - Employs(x,Richard) Who employs Richard?
  - Employs(IBM , y) Whom does IBM employ?
  - Employs(x, y) Who employs whom?
- These queries form a **subsumption lattice**



Storage and retrieval

- For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the “highest” common descendant of any two nodes is the result of applying their most general unifier.
- A sentence with repeated constants has a slightly different lattice,



Storage and retrieval

- Works very well whenever the lattice contains a small number of nodes.
- For a predicate with n arguments, however, the lattice contains $O(2^n)$ nodes.
- If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored.
- This can be solved by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked.