Lecture 16: Relevance Lemma and Relational Databases

In the last lecture we saw an introduction to first order logic, discussing both its syntax and semantics. After defining semantics, the reader may have wondered if quantified variables (i.e. those that appear next to the $\forall$ and $\exists$ quantifiers) should be distinguished from the other, non-quantified variables in a formula. This can be helpful to prove some results and so we will start by differentiating them.

1 Review: Distinction between free and bound variables

It is clear that variables used in quantifiers are different from the other variables in that the quantifier prevents the assignment from affecting it.
For example, consider the formula:

$$(\exists x)(\exists x)p(x)$$

This formula will be satisfiable if the following are satisfiable:

- $A, \alpha \models (\exists x)(\exists x)p(x)$ or
- $A, \alpha[x \mapsto a] \models (\exists x)p(x)$ or
- $A, \alpha[x \mapsto a][x \mapsto b] \models p(x)$

That means if there is an $a \in D$ and there is a $b \in D$, we can find a structure A and an assignment α so that $A, \alpha[x \mapsto a][x \mapsto b] \models p(b)$. From the definition of mapping on assignments, this becomes $A, \alpha[x \mapsto b] \models p(b)$. So the outermost quantifier does not affect the formula at all because the inner quantifier somehow makes x "disappear" from the formula.

Let's formalize this. First we define the set of variables that occur in a formula:

Definition 1 *The set of variables in a term t, denoted by $Vars(t)$, can be thought of as a function $Vars : Terms \to 2^{Variables}$, and is defined as:*

1. $Vars(x) = \{x\}$

2. $Vars(f(t_1, \ldots, t_k)) = \bigcup_i Vars(t_i)$

Definition 2 *The set of variables in a formula φ, denoted by $Vars(\varphi)$, can also be thought of as a function $Vars : Terms \to 2^{Variables}$ and equals:*

1. $Vars(P(t_1, \ldots, t_k)) = \bigcup_i Vars(t_i)$
2. $Vars(\neg\varphi) = Vars(\varphi)$
3. $Vars(\theta \wedge \psi) = Vars(\theta) \cup Vars(\psi)$
4. $Vars((\exists x)\varphi) = Vars((\forall x)\varphi) = Vars(\varphi) \cup \{x\}$

We can see from examples that variables occurring with a quantifier are treated differently in the assignment. To distinguish them, we call quantified variables **bound**, and unquantified variables **free**.

Definition 3 *The set of free variables in a formula φ, denoted by $FVars(\varphi)$, can be thought of as a function $FVars : Form \to 2^{Variables}$ and is defined as:*

1. $FVars(P(t_1, \ldots, t_k)) = \bigcup_i Vars(t_i)$
2. $FVars(\neg\varphi) = FVars(\varphi)$
3. $FVars(\theta \circ \psi) = FVars(\theta) \cup FVars(\psi)$
4. $FVars((\exists x)\varphi) = FVars((\forall x)\varphi) = FVars(\varphi) - \{x\}$

Definition 4 *A closed formula or sentence is a formula φ that has no free variables ($FVars(\varphi) = \emptyset$)*

2 The relevance lemma

When comparing different variable assignments and their effect on a given formula φ, those variables that do not occur in the formula or are not free variables in the formula should not affect the fact that a given structure A logically implies the formula or not. This intuitive result is formalized as the *Relevance Lemma*, and can be stated as:

Lemma 1 (Relevance lemma) *Let A be a structure, φ a formula, and α_1, α_2 be variable assignments such that: $\alpha_1\big|_{FVars(\varphi)} = \alpha_2\big|_{FVars(\varphi)}$. Then,*

$$A, \alpha_1 \models \varphi \text{ iff } A, \alpha_2 \models \varphi$$

To prove the relevance lemma, it will be very conveninent to use the following intuitive result, which states that the recursive evaluation of a term under variable assignments that coincide in the term's variables is the same.

Lemma 2 *Let $t \in$ TERM, and let A be a structure and α_1 and α_2 be variable assignments. Then, if $\alpha_1\big|_{Vars(t)} = \alpha_2\big|_{Vars(t)}$ then $\overline{\alpha_1}(t) = \overline{\alpha_2}(t)$.*

Proof: We give a proof by structural induction.

- **Base case**: If t is a variable x then $\alpha_1(x) = \alpha_2(x)$. Notice also that by definition of $\overline{\alpha}$ in this case $\overline{\alpha_1}(t) = \alpha_1(t)$ and the same is true for $\overline{\alpha_2}(t)$. Thus,
$$\overline{\alpha_1}(t) = \alpha_1(t) = \alpha_2(t) = \overline{\alpha_2}(t).$$
and therefore $\overline{\alpha_1}(t) = \overline{\alpha_2}(t)$. Here the first and the third equalities follow from the definition of $\overline{\alpha}$, and the second equality comes from the assumption of the lemma.

- **Inductive case**: The other option for t is to be a function $f^k(t_1, \ldots, t_k)$, where $t_1, \ldots, t_k$ are terms. By definition of $Vars$, $Vars(f^{(k)}(t_1, \ldots, t_k) = \bigcup_{i=1}^{k} Vars(t_i)$ and so $\textsc{Vars}(t_i) \subseteq \textsc{Vars}(t)$. Then by inductive hypothesis we have that $\overline{\alpha_1}(t_i) = \overline{\alpha_2}(t_i)$ for all i, and thus
$$\begin{array}{rcl} \overline{\alpha_1}(f^{(k)}(t_1, \ldots, t_k)) & = & \text{(by definition)} \\ f^A(\overline{\alpha_1}(t_1), \ldots, \overline{\alpha_1}(t_k)) & = & \text{(by I.H.)} \\ f^A(\overline{\alpha_2}(t_1), \ldots, \overline{\alpha_2}(t_k)) & = & \text{(by definition)} \\ \overline{\alpha_2}(f^{(k)}(t_1, \ldots, t_k)) & & \end{array}$$

$\diamondsuit\diamondsuit\diamondsuit$

We are now ready to prove the Relevance Lemma.

Proof:

- **Base case**: Suppose that φ is a k-ary predicate (relation) $P^{(k)}(t_1, \ldots, t_k)$. By definition $FVars(\varphi) = \bigcup_{i=1}^{k} FVars(t_i)$. We now have the following proof sequence:
$$\begin{array}{ll} & A, \alpha_1 \models \varphi \\ \text{iff} & \langle \overline{\alpha_1}(t_1), \ldots, \overline{\alpha_1}(t_k) \rangle \in P^A \text{ (by definition)} \\ \text{iff} & \langle \overline{\alpha_2}(t_1), \ldots, \overline{\alpha_2}(t_k) \rangle \in P^A \text{ (by Lemma 2)} \\ \text{iff} & A, \alpha_2 \models \varphi \text{ (by definition)} \end{array}$$

- **Inductive case**

  1. $\varphi = \neg\psi$. By definition $A, \alpha_1 \models \neg\psi$ iff $A, \alpha_1 \not\models \psi$. Also by definition $FVars(\varphi) = FVars(\psi)$. Thus, by inductive hypothesis we have $A, \alpha_2 \not\models \psi$ which leads to $A, \alpha_2 \models \varphi$.
  2. $\varphi = \psi \wedge \theta$. By definition $A, \alpha_1 \models \varphi$ iff $A, \alpha_1 \models (\psi \wedge \theta)$ iff $A, \alpha_1 \models \psi$ and $A, \alpha_1 \models \theta$. Clearly $FVars(\psi) \subseteq FVars(\varphi)$ and $FVars(\theta) \subseteq FVars(\varphi)$, so by inductive hypothesis $A, \alpha_2 \models \psi$ and $A, \alpha_2 \models \theta$. This is another way of saying $A, \alpha_2 \models \varphi$.

3. $\varphi = (\exists x)\psi$. $A, \alpha_1 \models (\exists x)\psi$ by definition means that there is some $d \in D$ such that $A, \alpha_1[x \mapsto d] \models \psi$. To use the inductive hypothesis we need agreement on the free variables. Observe that $FVars(\varphi) = FVars(\psi) - \{x\}$ and therefore $FVars(\psi) \subseteq FVars(\varphi) \cup \{x\}$.
Since $\alpha_1\big|_{FVars(\varphi)} = \alpha_2\big|_{FVars(\varphi)}$ it must also be true that $\alpha_1[x \mapsto d]\big|_{FVars(\psi)} = \alpha_2[x \mapsto d]\big|_{FVars(\psi)}$ which is the agreement on free variables that we need. We quote once again the inductive hypothesis and conclude that $A, \alpha_2[x \mapsto d] \models \varphi$ which is equivalent to $A, \alpha_2 \models (\exists x)\varphi$.

The proof for the remaining connectives and one quantifier can be derived from those above stated by using DeMorgan's laws. ♢♢♢

Corollary 1 *Let φ ne a sentence, A a structure, and α_1 and α_2 assignments. Then $A, \alpha_1 \models \varphi$ iff $A, \alpha_2 \models \varphi$.*

Thus, for sentences we can treat $\models$ is a binary relation. That is, , if φ is a sentence and A is a structure, then either $A \models \varphi$ or $A \not\models \varphi$.

3 Tower of Abstractions

One of the most fundamental characteristics of any scientific field is the ability to abstract away unnecessary details. A systems programmer rarely needs to worry about the digital circuits on which the operating system runs. A C++ programmer need not know the operating system on which his program will run. This is possible thanks to a concept called "Tower of Abstraction".

a. Analog circuits

b. Digital circuits

c. Microarchitecture

d. Architecture level

e. OS kernel

f. OS

g. Assembly level programming

h. High-level programming languages (Fortran, Cobol)

i. Middleware

Computer science starts at b.

4 Application of First Order Logic to Databases

We need a way for many people to access and manipulate the same data. From this came the concept of a Central Database. An application program would get services from the Database and OS and the Database would get services from the OS. The data in a database needs structure. In the 1960's people came up with the idea of organizing data in a hierarchy, known as the hierarchical model. A Hierarchical database stores its data in a series of records each of which has a set of field values attached to it. It collects all instances of record together as a record type. To create links between these record types, the hierarchical model uses Parent-Child relationships which creates tree-like structures. A more general model called the Network Model for Databases, allows cycles in the hierarchy.

Ted Codd at IBM devised a new type of database structure. He thought it was too complicated to have to follow all of the record links in the Network model to find the data you wanted. He came up with the idea of Tables. He found out that there is a connection between Tables and First Order Logic. A table is a set of rows and a relation is a set of tuples. So the idea of a row in a table is analogous to a tuple in a relation. From this he devised what he called the Relational Model of databases. This consisted of centralized data, with a set $\sigma = R_1 \dots R_k$ of relations. A databases can be viewed as *relational structure* $A = (D, R_1^A \dots R_k^A)$.

He then asserted that a query can be expressed as a formula $\varphi(x_1, \dots, x_k)$. We can view a variable assignment α, where $\alpha(x_i) = a_i$, as a tuple and look at it as $\langle a_1, \dots, a_k \rangle$.

We can now say that $A, \langle a_1, \dots, a_k \rangle \models \varphi(x_1, \dots, x_k)$. Thus, we define $\varphi(x_1 ... x_k)(A) = \{\langle a_1 ... a_k \rangle \mid A, < a_1 ... a_k > \models \varphi\}$. A response to a query will be the set of variable assignments satisfy $\varphi(x_1, \dots, x_k)$.

The two fundamental ideas of Codd are:

1. A table is a relation and a relational database is a set of relations.

2. A query is a formula

Example 1

EmpID	EmpName	EmpSalary
123	Moshe	1,000,000
456	Ted	1,000,000

The above table is the relation. The domain of the relation is $\{EmpIDs\} \cup \{EmpNames\} \cup \{EmpSalaries\}$.

However, $< Moshe, Moshe, Moshe >$ doesn't quite make sense in the above table. It is possible to split the domain D into $D_1 \times D_2 \times D_3$, and have elaborate type systems. In this class we will stay more abstract and ignore types.

Example 2

Suppose we have a database having the following relations

$Employee{-}Dept^{(2)}, Manager{-}Department^{(2)}, Manager{-}Secretary^{(2)}, Employee{-}Salary^{(2)}$ as well as the predicate $>^{(2)}$

Suppose we want to: 'Find all employees whose manager's secretary makes more money than the manager'.

To answer this query we need one free variable x for Employee:

$(\exists y)(\exists z)(\exists u)(\exists s1)(\exists s2)Employee{-}Department(x,y) \wedge Manager{-}Department(z,y) \wedge Manager{-}Secretary(z,u) \wedge Employee{-}Salary(z,s1) \wedge Employee{-}Salary(u,s2) \wedge > (s2,s1)$.

This query will return all the possible variable assignments to our free variables (Employee) that match the constraints in the formula.

What if φ has no free variables, i.e. it is a sentence? Then, $\varphi(A) = \{<> | A, <> \models \varphi\}$. So, if φ is a sentence, then $\varphi(A) = \emptyset$ or $\varphi(A) = \{<>\}$. By convention, $\{<>\}$ is "true", and $\emptyset$ is "false". Therefore, $\varphi(A)$ is either true or false if φ is a sentence. Such a query is referred to as a Boolean query.

These two ideas were fundamental enough for Ted Codd to be awarded the Turing Award. After Codd published this, two projects were started to test its viability: Ingress at UC Berkeley, and SystemR at IBM. Ingress used QUEL as a query language and SystemR used SEQUEL. Both of these were later combined to create SQL.

Typically, the formulas for a database are expressed as statements in SQL. The intent of SQL was that it would be simple enough for anyone to use. Ideally, you shouldn't be required to know First Order Logic in order to access database information.