

Logic Programming and Deductive Databases

Chapter 2: Basic Notions of Predicate Logic

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2021

<http://www.informatik.uni-halle.de/~brass/lp21/>

Objectives

After completing this chapter, you should be able to:

- explain the basic notions: signature, interpretation, variable assignment, term, formula, model, consistent formula, implication between formulas.
- use some common equivalences to transform logical formulas.
- write formulas for given specifications.
- check whether a formula is true in an interpretation.
- find models of a given formula (if consistent).

Contents

- ① Signature, Interpretation
- ② Formulas (Syntax)
- ③ Models (Semantics)
- ④ Formulas in Databases
- ⑤ Implication, Equivalence

Alphabet (1)

Definition:

- Let $ALPH$ be some infinite, but enumerable set, the elements of which are called symbols.

Formulas will be words over $ALPH$, i.e. sequences of symbols.

- $ALPH$ must contain at least the logical symbols, i.e. $LOG \subseteq ALPH$, where

$$LOG = \{ (,), , , \wedge, \vee, \leftarrow, \rightarrow, \leftrightarrow, \forall, \exists \}.$$

- In addition, $ALPH$ must contain an infinite subset $VARs \subseteq ALPH$, the set of variables. This must be disjoint to LOG (i.e. $VARs \cap LOG = \emptyset$).

Some authors consider variables as logical symbols.

Alphabet (2)

- E.g., the alphabet might consist of
 - the special logical symbols *LOG*,
 - variables starting with an uppercase letter and consisting otherwise of letters, digits, and “_”,
 - identifiers starting with a lowercase letter and consisting otherwise of letters, digits, and “_”.
- Note that words like “*father*” are considered as symbols (elements of the alphabet).

Compare with: lexical scanner vs. context-free parser in a compiler.

- In theory, the exact symbols are not important.

Signatures (1)

Definition:

- A signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \alpha, \rho)$ consists of:
 - A non-empty and finite set \mathcal{S} , the elements of which are called **sorts** (data type names).
 - $\mathcal{P} \subseteq \text{ALPH} \setminus (\text{LOG} \cup \text{VARS})$, the elements are called **predicate symbols**.
 - $\mathcal{F} \subseteq \text{ALPH} \setminus (\text{LOG} \cup \text{VARS} \cup \mathcal{P})$, the elements are called **function symbols**.
 - $\alpha: (\mathcal{P} \cup \mathcal{F}) \rightarrow \mathcal{S}^*$, which defines the **argument sorts** of predicates and functions.
 - $\rho: \mathcal{F} \rightarrow \mathcal{S}$, this defines the **result sort** of functions.

Signatures (2)

- If $\alpha(c) = \epsilon$ for some $c \in \mathcal{F}$ (i.e. c has no arguments), then c is called a **constant**.
- A predicate symbol $p \in \mathcal{P}$ with $\alpha(p) = \epsilon$ is called a **propositional symbol**.
- The length of the argument string (number of arguments) is called the **arity** of the function/predicate.
- The above definition is for a multi-sorted (typed) logic. One can also use an unsorted logic.

Unsorted means really one-sorted. Then \mathcal{S} and ρ are not needed, and α defines the arity, i.e. $\alpha: (\mathcal{P} \cup \mathcal{F}) \rightarrow \mathbb{N}_0$. E.g. Prolog uses an unsorted logic. This is also common in textbooks about mathematical logic.

Signatures (3)

Example:

- $\mathcal{S} = \{\text{person}, \text{string}\}$.
- \mathcal{F} consists of
 - constants of sort `person`, e.g. `arno`, `birgit`, `chris`.
 - infinitely many constants of sort `string`, e.g. `''`, `'a'`, `'b'`, ..., `'Arno'`, ...
 - function symbols `first_name: person \rightarrow string` and `last_name: person \rightarrow string`.
- \mathcal{P} consists of
 - a predicate `married_with: person \times person`.

Interpretations (1)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \alpha, \rho)$ be given.
- A Σ -interpretation \mathcal{I} defines:
 - a non-empty set $\mathcal{I}(s)$ for every $s \in \mathcal{S}$ (domain),
 - a relation $\mathcal{I}(p) \subseteq \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n)$ for every $p \in \mathcal{P}$, where $s_1, \dots, s_n := \alpha(p)$,
 - a function $\mathcal{I}(f): \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n) \rightarrow \mathcal{I}(s)$ for every $f \in \mathcal{F}$, where $s_1, \dots, s_n := \alpha(f)$ and $s := \rho(f)$.
- In the following, we write $\mathcal{I}[\dots]$ instead of $\mathcal{I}(\dots)$.

Interpretations (2)

Example:

- $\mathcal{I}[\text{person}]$ is the set of Arno, Birgit, and Chris.
- $\mathcal{I}[\text{string}]$ is the set of all strings, e.g. 'a'.
- $\mathcal{I}[\text{arno}]$ is Arno.
- For the string constants, \mathcal{I} is the identity mapping.

If one has \-escapes and octal codes as in C, several constants are mapped to the same string. Or consider 0, -0, 00.

- $\mathcal{I}[\text{first_name}]$ maps e.g. Arno to 'Arno'.
- $\mathcal{I}[\text{last_name}]$ maps all three persons to 'Schmidt'.
- $\mathcal{I}[\text{married_with}] = \{(\text{Birgit}, \text{Chris}), (\text{Chris}, \text{Birgit})\}$.

Relation to Databases (1)

- A DBMS defines a set of data types, such as strings and numbers, together with constants, data type functions (e.g. $+$) and predicates (e.g. $<$).
- For these, the DBMS defines names (in the signature) and their meaning (in the interpretation).
- The DB schema in the relational model then adds further predicate symbols (relation symbols).
- The DB state interprets these by finite relations.

Whereas the interpretation of the data types is fixed and built into the DBMS, the interpretation of the additional predicate symbols can be modified by insertions, deletions, and updates. But it must be finite.

Relation to Databases (2)

- In the Entity-Relationship-Model, the DB schema can introduce
 - new sorts (entity types),
 - new functions of arity 1 from entity types to data types (the attributes),
 - new predicates between entity types, possibly restricted to arity 2 (the relationships).
- The interpretation of the entity types (in the DB state) must always be finite.

Thus, also attributes and relationships are finite.

Contents

- 1 Signature, Interpretation
- 2 Formulas (Syntax)**
- 3 Models (Semantics)
- 4 Formulas in Databases
- 5 Implication, Equivalence

Variable Declaration (1)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \alpha, \rho)$ be given.
- A variable declaration for Σ is a partial mapping $\nu: VARS \rightarrow \mathcal{S}$ (defined only for a finite subset of $VARS$).

Remark:

- The variable declaration is not part of the signature because it is locally modified by quantifiers (see below).
- The signature is fixed for the entire application, the variable declaration changes even within a formula.

Variable Declaration (2)

Definition:

- Let ν be a variable declaration, $X \in VARS$, and $s \in \mathcal{S}$.
- Then we write $\nu\langle X/s \rangle$ for the modified variable declaration ν' with

$$\nu'(V) := \begin{cases} s & \text{if } V = X \\ \nu(V) & \text{otherwise.} \end{cases}$$

Remark:

- Both is possible: ν might have been defined before for X or it might be undefined.

Terms (1)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \alpha, \rho)$ and a variable declaration ν for Σ be given.
- The set $TE_{\Sigma, \nu}(s)$ of terms of sort s is recursively defined as follows:
 - Every variable $V \in VARS$ with $\nu(V) = s$ is a term of sort s (this of course requires that ν is defined for V).
 - If t_1 is a term of sort s_1, \dots, t_n is a term of sort s_n , and $f \in \mathcal{F}$ with $\alpha(f) = s_1 \dots s_n$ and $\rho(f) = s$, then $f(t_1, \dots, t_n)$ is a term of sort s .
 - Nothing else is a term.

Terms (2)

- In particular every constant c of sort s is a term of sort s . Formally, one would have to write " $c()$ " for the term, but one simplifies the notation to " c ".
- Certain functions are also written as infix operators, e.g. $X+1$ instead of the official notation $+(X, 1)$.
- Such "syntactic sugar" is not important for the theory of logic.

In programming languages, there is sometimes a distinction between "concrete syntax" and "abstract syntax" (the syntax tree).

- Let $TE_{\Sigma, \nu} := \bigcup_{s \in \mathcal{S}} TE_{\Sigma, \nu}(s)$ be the set of all terms.

Atomic Formulas

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \alpha, \rho)$ and a variable declaration ν for Σ be given.

- An atomic formula is an expression of the form

$$p(t_1, \dots, t_n)$$

where $p \in \mathcal{P}$, $\alpha(p) = s_1 \dots s_n$, and $t_i \in TE_{\Sigma, \nu}(s_i)$ for $i = 1, \dots, n$.

Again, one sometimes uses infix notation in concrete syntax, e.g. $X > 1$.

- Let $AT_{\Sigma, \nu}$ be the set of atomic formulas for Σ, ν .

Formulas (1)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \alpha, \rho)$ and a variable declaration ν for Σ be given.
- The sets $FO_{\Sigma, \nu}$ of (Σ, ν) -formulas are defined recursively as follows:
 - Every atomic formula $F \in AT_{\Sigma, \nu}$ is a formula.
 - If F and G are formulas, so are $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
 - $(\forall X: s F)$ and $(\exists X: s F)$ are in $FO_{\Sigma, \nu}$ if $s \in \mathcal{S}$, $X \in VARS$, and F is a formula for Σ and $\nu\langle X/s \rangle$.
 - Nothing else is a formula.

Formulas (2)

- The intuitive meaning of the formulas is as follows:
 - $\neg F$: “Not F ” (F is false).
 - $F \wedge G$: “ F and G ” (F and G are both true).
 - $F \vee G$: “ F or G ” (at least one of F and G is true).
 - $F \leftarrow G$: “ F if G ” (if G is true, F must be true).
 - $F \rightarrow G$: “if F , then G ”
 - $F \leftrightarrow G$: “ F if and only if G ”.
 - $\forall X: s F$: “for all X (of sort s), F is true”.
 - $\exists X: s F$: “there is an X (of sort s) such that F ”.

Formulas (3)

- A Σ -formula is a (Σ, ν) -formula for any variable declaration ν .

The variable declaration is local to the formula. If one considers a set of Σ -formulas, each formula can use a different variable declaration.

- Variants of predicate logic:
 - One can add atomic formulas “*true*” and “*false*” that are interpreted as true and false, resp.
 - One can add atomic formulas of the form $t_1 = t_2$ and ensure that is is really interpreted as equality.

Formulas (4)

- Above, many parentheses are used in order to ensure that formulas have a unique syntactic structure.

For the formal definition, this is a simple solution, but for writing formulas in practical applications, the syntax becomes clumsy.

- One uses the following rules to save parentheses:
 - The outermost parentheses are never needed.
 - \neg binds strongest, then \wedge , then \vee , then \leftarrow , \rightarrow , \leftrightarrow (same binding strength), and last \forall , \exists .
 - Since \wedge and \vee are associative, no parentheses are required for e.g. $F_1 \wedge F_2 \wedge F_3$.

Note that \rightarrow and \leftarrow are not associative.

Formulas (5)

Formal Treatment of Binding Strengths:

- A level 0 formula is an atomic formula or a level 5 formula enclosed in parentheses.

The level of a formula corresponds to the binding strength of its outermost operator (smallest number means highest binding strength). However, one can use a level i -formula like a level j -formula with $j > i$. In the opposite direction, parentheses are required.

- A level 1 formula is a level 0 formula or a formula of the form $\neg F$ with a level 1 formula F .
- A level 2 formula is a level 1 formula or a formula of the form $F_1 \wedge F_2$ with a level 2 formula F_1 and a level 1 formula F_2 .

Formulas (6)

Formal Treatment of Binding Strengths, Continued:

- A level 3 formula is a level 2 formula or a formula of the form $F_1 \vee F_2$ with a level 3 formula F_1 and a level 2 formula F_2 .
- A level 4 formula is a level 3 formula or a formula of the form $F_1 \leftarrow F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$ with level 3 formulas F_1 and F_2 .
- A level 5 formula is a level 4 formula or a formula of the form $\forall X: s F$ or $\exists X: s F$ with a level 5 formula F .
- A formula is a level 5 formula.

Variables in a Term

Definition:

- The function *vars* computes the set of variables that occur in a given term *t*.
 - If *t* is a variable *V*:

$$\text{vars}(t) := \{V\}.$$

- If *t* has the form $f(t_1, \dots, t_n)$:

$$\text{vars}(t) := \bigcup_{i=1}^n \text{vars}(t_i).$$

Free Variables in a Formula

Definition:

- The function *free* computes the set of free variables (not bound by a quantifier) in a formula F :
 - If F is an atomic formula $p(t_1, \dots, t_n)$:

$$\text{free}(F) := \bigcup_{i=1}^n \text{vars}(t_i).$$

- If F has the form $(\neg G)$:
 $\text{free}(F) := \text{free}(G)$.
- If F has the form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.:
 $\text{free}(F) := \text{free}(G_1) \cup \text{free}(G_2)$.
- If F has the form $(\forall X: s G)$ or $(\exists X: s G)$:
 $\text{free}(F) := \text{free}(G) \setminus \{X\}$.

Closed and Ground Formulas

Definition:

- A formula F is **closed** iff $\text{free}(F) = \emptyset$.

A closed formula may contain variables, but they are all bound by quantifiers.

- A formula or a term is **ground** iff it does not contain any variables (not even quantified ones).

Every ground formula is closed, but the opposite is not in general true.

- A formula is **propositional** iff it contains only predicates without arguments and no quantifiers.

Predicates without arguments are called propositional symbols. A propositional formula is always ground.

Uniqueness of Variable Sorts

- Let F be a (Σ, ν) -formula. If $X \in \text{free}(F)$, the variable X can have only a unique sort, namely $s = \nu(X)$.

I.e. if ν_1 and ν_2 are two variable declarations such that F is a (Σ, ν_i) -formula, and if $X \in \text{free}(F)$, then $\nu_1(X) = \nu_2(X)$. The reason is that function and predicate symbols cannot be overloaded, thus, when the variable appears in an argument, it must have a certain sort. Note that in practice there might be exceptions, e.g. the predicate “=”.

- For every (Σ, ν) -formula F , there is a unique variable declaration $\nu' \subseteq \nu$ that is defined exactly on $\text{free}(F)$ and such that F is a (Σ, ν') -formula.
- Let $\text{vdecl}(F)$ be this minimal variable declaration.

Abbreviations of Quantifiers

- If $X \in \text{free}(F)$, one can leave out the sort in the quantifiers, because it is uniquely determined by F . I.e. one can write $\forall X F$ instead of $\forall X: s F$.

And the same for \exists . If $X \notin \text{free}(F)$, the quantifier is anyway a bit strange. But in this case, if $\mathcal{I}[s]$ can be empty, it would make a difference which sort is chosen, thus this abbreviation cannot be used.

- Instead of a sequence of quantifiers of the same type, e.g. $\forall X_1 \dots \forall X_n F$, one can write $\forall X_1, \dots, X_n F$.
- The universal closure of a formula F , written $\forall(F)$, is $\forall X_1 \dots \forall X_n F$, where $\{X_1, \dots, X_n\} := \text{free}(F)$.

Contents

- 1 Signature, Interpretation
- 2 Formulas (Syntax)
- 3 Models (Semantics)**
- 4 Formulas in Databases
- 5 Implication, Equivalence

Variable Assignment

Definition:

- A variable assignment \mathcal{A} for \mathcal{I} and ν is a partial mapping from $VARS$ to $\bigcup_{s \in \mathcal{S}} \mathcal{I}[s]$.
- It maps every variable V , for which ν is defined, to a value from $\mathcal{I}[s]$, where $s := \nu(V)$.

I.e. a variable assignment for \mathcal{I} and ν defines values from \mathcal{I} for the variables that are declared in ν .

Definition:

- $\mathcal{A}\langle X/d \rangle$ denotes a variable assignment \mathcal{A}' that agrees with \mathcal{A} except that $\mathcal{A}'(X) = d$.

Value of a Term

Definition:

- Let a signature Σ , a variable declaration ν for Σ , a Σ -interpretation \mathcal{I} , and a variable assignment \mathcal{A} for (\mathcal{I}, ν) be given.
- The value $\langle \mathcal{I}, \mathcal{A} \rangle[t]$ of a term $t \in TE_{\Sigma, \nu}$ is defined as follows (recursion over the structure of the term):
 - If t is a variable V , then $\langle \mathcal{I}, \mathcal{A} \rangle[t] := \mathcal{A}(V)$.
 - If t has the form $f(t_1, \dots, t_n)$, then
$$\langle \mathcal{I}, \mathcal{A} \rangle[t] := \mathcal{I}[f](\langle \mathcal{I}, \mathcal{A} \rangle[t_1], \dots, \langle \mathcal{I}, \mathcal{A} \rangle[t_n]).$$

Truth of a Formula (1)

Definition:

- The truth value $\langle \mathcal{I}, \mathcal{A} \rangle[F] \in \{\mathbf{f}, \mathbf{w}\}$ of a formula F in $(\mathcal{I}, \mathcal{A})$ is defined as follows (\mathbf{f} means false, \mathbf{w} true):
 - If F is an atomic formula $p(t_1, \dots, t_n)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathbf{w} & \text{if } (\langle \mathcal{I}, \mathcal{A} \rangle[t_1], \dots, \langle \mathcal{I}, \mathcal{A} \rangle[t_n]) \\ & \in \mathcal{I}[p] \\ \mathbf{f} & \text{else.} \end{cases}$$

- If F is of the form $(\neg G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathbf{w} & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle[G] = \mathbf{f} \\ \mathbf{f} & \text{else.} \end{cases}$$

Truth of a Formula (2)

Definition, continued:

- Truth value of a formula, continued:
 - If F is of the form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.:

G_1	G_2	\wedge	\vee	\leftarrow	\rightarrow	\leftrightarrow
f	f	f	f	w	w	w
f	w	f	w	f	w	f
w	f	f	w	w	f	f
w	w	w	w	w	w	w

E.g. if $\langle \mathcal{I}, \mathcal{A} \rangle[G_1] = \mathbf{w}$ and $\langle \mathcal{I}, \mathcal{A} \rangle[G_2] = \mathbf{f}$
then $\langle \mathcal{I}, \mathcal{A} \rangle[(G_1 \wedge G_2)] = \mathbf{f}$.

Truth of a Formula (3)

Definition, continued:

- Truth value of a formula, continued:
 - If F has the form $(\forall X: s \ G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle [F] := \begin{cases} \mathbf{w} & \text{if } \langle \mathcal{I}, \mathcal{A}(X/d) \rangle [G] = \mathbf{w} \\ & \text{for all } d \in \mathcal{I}[s] \\ \mathbf{f} & \text{else.} \end{cases}$$

- If F has the form $(\exists X: s \ G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle [F] := \begin{cases} \mathbf{w} & \text{if } \langle \mathcal{I}, \mathcal{A}(X/d) \rangle [G] = \mathbf{w} \\ & \text{for at least one } d \in \mathcal{I}[s] \\ \mathbf{f} & \text{else.} \end{cases}$$

Model (1)

Notation:

- If $\langle \mathcal{I}, \mathcal{A} \rangle[F] = \mathbf{w}$, one also writes $(\mathcal{I}, \mathcal{A}) \models F$.

Definition:

- Let F be a (Σ, ν) -formula. A Σ -interpretation \mathcal{I} is a **model** of the formula F (written $\mathcal{I} \models F$) iff $\langle \mathcal{I}, \mathcal{A} \rangle[F] = \mathbf{w}$ for all variable assignments \mathcal{A} .
I.e. free variables are treated as \forall -quantified. Of course, if F is a closed formula, the variable declaration is not important.
- A Σ -interpretation \mathcal{I} is a model of a set Φ of Σ -formulas, written $\mathcal{I} \models \Phi$, iff $\mathcal{I} \models F$ for all $F \in \Phi$.

Model (2)

Definition:

- A formula F or set of formulas Φ is called **consistent** iff it has a model. Otherwise it is called inconsistent.

A model must make the formula true for all variable assignments.

- A formula F is called **satisfiable** iff there is an interpretation \mathcal{I} and a variable assignment \mathcal{A} such that $(\mathcal{I}, \mathcal{A}) \models F$. Otherwise it is called unsatisfiable.
- A (Σ, ν) -formula F is called a **tautology** iff for all Σ -interpretations \mathcal{I} and (Σ, ν) -variable assignments \mathcal{A} : $(\mathcal{I}, \mathcal{A}) \models F$.

Contents

- 1 Signature, Interpretation
- 2 Formulas (Syntax)
- 3 Models (Semantics)
- 4 Formulas in Databases**
- 5 Implication, Equivalence

Formulas in Databases (1)

- Consider a database with relations
 - $\text{EMP}(\underline{\text{EMPNO}}, \text{ENAME}, \text{SAL}, \text{DEPTNO} \rightarrow \text{DEPT})$
 - $\text{DEPT}(\underline{\text{DEPTNO}}, \text{DNAME}, \text{LOC})$
- Formulas can be used as queries: They ask for values for the free variables that make the formula true in the current database state (interpretation).
- E.g. print name and salary of all employees in the research department (X and Y are the free variables):
$$\exists E, D, L \text{ emp}(E, X, Y, D) \wedge \text{dept}(D, \text{'RESEARCH'}, L)$$

Formulas in Databases (2)

- In order to make the free variables, for which values are sought, better visible, domain calculus queries are usually written in the form:

$$\{X, Y \mid \exists E, D, L \text{ emp}(E, X, Y, D) \wedge \text{dept}(D, \text{'RESEARCH'}, L)\}$$

- One cannot use arbitrary formulas as queries. E.g. some formulas would generate an infinite answer:

$$\{D, N \mid \neg \text{dept}(D, N, \text{'NEW YORK'})\}$$

- Other formulas would require that infinitely many values are tried for quantified variables:

$$\exists X \exists Y \exists Z \exists n \ X^n + Y^n = Z^n \wedge n > 2 \wedge X > 0 \wedge Y > 0$$

Formulas in Databases (3)

- A formula is **domain independent** iff for all possible DB states (interpretations), it suffices to replace variables by values that appear in any argument of the DB relations or as ground term in the query.

For a given interpretation \mathcal{I} and formula F , the “active domain” is the set of values that appear in database relations in \mathcal{I} or as ground term (e.g. constant) in F . Domain independence means that (1) F must be false if a value outside this set is inserted for a free variable. (2) For all subformulas $\exists X G$, the formula G must be false if X has a value outside the active domain. (3) For all subformulas $\forall X G$, the formula G must be true if X has a value outside the active domain.

- Since all database relations are finite, queries can be evaluated in finite time.

Formulas in Databases (4)

- “Range restriction” is a syntactic (decidable) constraint on formulas that implies the domain independence (i.e. it is stricter than domain independence).

For every formula, one defines the set of restricted variables in positive context and in negative context.

E.g. if F is an atomic formula $p(t_1, \dots, t_n)$ with database relation p , then $posres(F) := free(F)$ and $negres(F) := \emptyset$.

For other atomic formulas, both sets are empty, except that when F has the form $X = t$ with a ground term t , $posres(F) := \{X\}$.

If F is $\neg G$, then $posres(F) := negres(G)$ and $negres(F) := posres(G)$.

If F has the form $G_1 \wedge G_2$, then $posres(F) := posres(G_1) \cup posres(G_2)$ and $negres(F) := negres(G_1) \cap negres(G_2)$. Etc.

A formula F is range restricted if $free(F) = posres(F)$ and for every subformula $\forall X G$, it holds that $X \in negres(G)$, and for every subformula $\exists X G$, it holds that $X \in posres(G)$.

Formulas in Databases (5)

- **Exercise:** Write these queries in domin calculus:
 - Which employees work in New York or Dallas and earn more than 3000 \$ per month?
 - Which department has no employees?
 - Print for every department the employee(s) with the greatest salary in that department.
- Relations:
 - $\text{EMP}(\underline{\text{EMPNO}}, \text{ENAME}, \text{SAL}, \text{DEPTNO} \rightarrow \text{DEPT})$
 - $\text{DEPT}(\underline{\text{DEPTNO}}, \text{DNAME}, \text{LOC})$

Formulas in Databases (6)

- Closed formulas can appear as boolean queries, but more often they are used as constraints, e.g.
 - DEPTNO is a key of DEPT(DEPTNO, DNAME, LOC):

$$\begin{aligned} \forall D, N_1, L_1, D_2, N_2, L_3 \\ \text{dept}(D, N_1, L_1) \wedge \text{dept}(D, N_2, L_2) \rightarrow \\ N_1 = N_2 \wedge L_1 = L_2 \end{aligned}$$

- DEPTNO in EMP is a foreign key that references DEPT:
$$\forall X, Y, Z, D (\text{emp}(X, Y, Z, D) \rightarrow \exists N, L \text{dept}(D, N, L)).$$

- Exercise: Write a constraint that DEPTNO in DEPT must be greater than 0.

Contents

- 1 Signature, Interpretation
- 2 Formulas (Syntax)
- 3 Models (Semantics)
- 4 Formulas in Databases
- 5 Implication, Equivalence

Implication

Definition/Notation:

- A formula or set of formulas Φ (logically) **implies** a formula or set of formulas G iff every model of Φ is also a model of G . In this case we write $\Phi \models G$.
- Many authors write $\Phi \models G$.

The difference is important if one talks also about axioms and deduction rules. Then $\Phi \vdash G$ is used for syntactic deduction, and $\Phi \models G$ for the implication defined above via models. Correctness and completeness of the deduction system then mean that both relations agree.

Lemma:

- $\Phi \models G$ if and only if $\Phi \cup \{\neg G\}$ is inconsistent.

Equivalence (1)

Definition/Lemma:

- Two Σ -formulas or sets of Σ -formulas F_1 and F_2 are **equivalent** iff they have the same models, i.e. for every Σ -interpretation \mathcal{I} :

$$\mathcal{I} \models F_1 \iff \mathcal{I} \models F_2.$$

- F_1 and F_2 are equivalent iff $F_1 \models F_2$ and $F_2 \models F_1$.

Lemma:

- “Equivalence” of formulas is an equivalence relation, i.e. it is reflexive, symmetric, and transitive.

This also holds for strong equivalence defined on the next page.

Equivalence (2)

Definition/Lemma:

- Two (Σ, ν) -formulas F_1 and F_2 are **strongly equivalent** iff for every Σ -interpretation \mathcal{I} and every (\mathcal{I}, ν) -variable assignment \mathcal{A} :

$$(\mathcal{I}, \mathcal{A}) \models F_1 \iff (\mathcal{I}, \mathcal{A}) \models F_2.$$

- Strong equivalence of F_1 and F_2 is written: $F_1 \equiv F_2$.
- Suppose that G_1 results from G_2 by replacing a subformula F_1 by F_2 and let F_1 and F_2 be strongly equivalent. Then G_1 and G_2 are strongly equivalent.

Some Equivalences (1)

- Commutativity (for and, or, iff):
 - $F \wedge G \equiv G \wedge F$
 - $F \vee G \equiv G \vee F$
 - $F \leftrightarrow G \equiv G \leftrightarrow F$
- Associativity (for and, or, iff):
 - $F_1 \wedge (F_2 \wedge F_3) \equiv (F_1 \wedge F_2) \wedge F_3$
 - $F_1 \vee (F_2 \vee F_3) \equiv (F_1 \vee F_2) \vee F_3$
 - $F_1 \leftrightarrow (F_2 \leftrightarrow F_3) \equiv (F_1 \leftrightarrow F_2) \leftrightarrow F_3$

Some Equivalences (2)

- Distribution Law:

- $F \wedge (G_1 \vee G_2) \equiv (F \wedge G_1) \vee (F \wedge G_2)$

- $F \vee (G_1 \wedge G_2) \equiv (F \vee G_1) \wedge (F \vee G_2)$

- Double Negation:

- $\neg(\neg F) \equiv F$

- De Morgan's Law:

- $\neg(F \wedge G) \equiv (\neg F) \vee (\neg G).$

- $\neg(F \vee G) \equiv (\neg F) \wedge (\neg G).$

Some Equivalences (3)

- Replacements of Implication Operators:
 - $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (F \leftarrow G)$
 - $F \leftarrow G \equiv G \rightarrow F$
 - $F \rightarrow G \equiv \neg F \vee G$
 - $F \leftarrow G \equiv F \vee \neg G$
- Together with De Morgan's Law this means that e.g. $\{\neg, \vee\}$ are sufficient, all other logical junctors $\{\wedge, \leftarrow, \rightarrow, \leftrightarrow\}$ can be expressed with them.

As we will see, also only one of the quantifiers is needed.

Some Equivalences (4)

- Replacements for Quantifiers:
 - $\forall X: s \ F \equiv \neg(\exists X: s \ (\neg F))$
 - $\exists X: s \ F \equiv \neg(\forall X: s \ (\neg F))$
- Moving logical junctors over quantifiers:
 - $\neg(\forall X: s \ F) \equiv \exists X: s \ (\neg F)$
 - $\neg(\exists X: s \ F) \equiv \forall X: s \ (\neg F)$
 - $\forall X: s \ (F \wedge G) \equiv (\forall X: s \ F) \wedge (\forall X: s \ G)$
 - $\exists X: s \ (F \vee G) \equiv (\exists X: s \ F) \vee (\exists X: s \ G)$

Some Equivalences (5)

- Moving quantifiers: If $X \notin \text{free}(F)$:

- $\forall X: s (F \vee G) \equiv F \vee (\forall X: s G)$

- $\exists X: s (F \wedge G) \equiv F \wedge (\exists X: s G)$

If in addition $\mathcal{I}[s]$ cannot be empty:

- $\forall X: s (F \wedge G) \equiv F \wedge (\forall X: s G)$

- $\exists X: s (F \vee G) \equiv F \vee (\exists X: s G)$

- Removing unnecessary quantifiers:

If $X \notin \text{free}(F)$ and $\mathcal{I}[s]$ cannot be empty:

- $\forall X: s F \equiv F$

- $\exists X: s F \equiv F$

Some Equivalences (6)

- Exchanging quantifiers: If $X \neq Y$:
 - $\forall X: s_1 (\forall Y: s_2 F) \equiv \forall Y: s_2 (\forall X: s_1 F)$
 - $\exists X: s_1 (\exists Y: s_2 F) \equiv \exists Y: s_2 (\exists X: s_1 F)$

Note that quantifiers of different type (\forall and \exists) cannot be exchanged.

- Renaming bound variables: If $Y \notin \text{free}(F)$ and F' results from F by replacing every free occurrence of X in F by Y :
 - $\forall X: s F \equiv \forall Y: s F'$
 - $\exists X: s F \equiv \exists Y: s F'$