Principles of Data- and Knowledge-based Systems

Torsten Schaub
University of Potsdam
torsten@cs.uni-potsdam.de

Logic: Overview

Outline

Introduction, Motivation (1)

Important goals of mathematical logic are

- to formalize the notion of a statement about a certain domain of discourse (logical formula),
- to precisely define the notions of logical implication and proof,
- to find ways to mechanically check whether a statement is logically implied by given statements.

Introduction, Motivation (2)

Mathematical logic is applied in databases I

- In general, the purpose of both, mathematical logic and databases, is to
  - formalize knowledge,
  - work with this knowledge (process it).
- For instance, in order to talk about a domain of discourse, symbols are needed.
  - In logic, these are defined in a signature.
  - In databases, they are defined in a DB schema.

Introduction, Motivation (2)

Mathematical logic is applied in databases II

- In order to formalize logical implication, mathematical logic had to study possible interpretations of the symbols,

  i.e. possible situations in the domain of discourse about which the logical formulas make statements.
- Database states also describe possible situations in a certain part of the real world.
- Basically, logical interpretations and DB states are the same (at least in the "model-theoretic view").

Introduction, Motivation (3)

Mathematical logic is applied in databases III

- SQL queries are quite similar to formulas in mathematical logic, and there are theoretical query languages that are simply a version of logic.
- The idea is that
  - a query is a logical formula with placeholders ("free variables"),
  - the database system then determines values for these placeholders that make the formula true in the given database state.

Introduction, Motivation (4)

Why it makes sense to learn mathematical logic I

- Logical formulas are simpler than SQL, and can easily be formally studied.
- Important concepts of database queries can already be learned in this simpler, purer environment.
- Experience has shown that students often make logical errors in SQL queries.

Introduction, Motivation (5)

Why it makes sense to learn mathematical logic II

- SQL changes, and becomes more and more complicated (standards: 1986, 1989, 1992, 1999, 2003).
- There are new data models (e.g., XML) with new query languages, and faster changes than SQL.
- At least some part of this course should still be valid and useful in 30 years.

History of the Field (1)

~322 BC	Syllogisms [Aristoteles]
~300 BC	Axioms of Geometry [Euklid]
~1700	Plan of Mathematical Logic [Leibniz]
1847	“Algebra of Logic” [Boole]
1879	“Begriffsschrift” (Early Logical Formulas) [Frege]
~1900	More natural formula syntax [Peano]
1910/13	Principia Mathematica (Collection of formal proofs) [Whitehead/Russell]
1930	Completeness Theorem [Gödel/Herbrand]
1936	Undecidability [Church/Turing]

History of the Field (2)

1960	First Theorem Prover [Gilmore/Davis/Putnam]
1963	Resolution-Method for Theorem proving [Robinson]
$\sim$1969	Question Answering Systems [Green et.al.]
1970	Linear Resolution [Loveland/Luckham]
1970	Relational Data Model [Codd]
$\sim$1973	Prolog [Colmerauer, Roussel, et.al.] (Started as Theorem Prover for Natural Language Understanding) (Compare with: Fortran 1954, Lisp 1962, Pascal 1970, Ada 1979)
1977	Conference "Logic and Databases" [Gallaire, Minker]

Outline

Alphabet (1)

Definition

- Let $ALPH$ be some infinite, but enumerable set, the elements which are called symbols.
- $ALPH$ must contain at least the logical symbols, i.e. $LOG \subseteq ALPH$, where

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

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

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).
- In theory, the exact symbols are not important.

Alphabet (3)

- If the special logical symbols are not available, use:

Symbol	Alternative	Another	Name
$\top$	`true`	`T`	
$\bot$	`false`	`F`	
$\neg$	`not`	$\sim$	Negation
$\wedge$	`and`	`&`	Conjunction
$\vee$	`or`	|	Disjunction
$\leftarrow$	`if`	`<-`	
$\rightarrow$	`then`	`->`	
$\leftrightarrow$	`iff`	`<->`	
$\exists$	`exists`	`E`	Existential Quantifier
$\forall$	`forall`	`A`	Universal Quantifier

Signatures (1)

Definition

- A signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ consists of:
  - A non-empty and finite set $\mathcal{S}$, the elements which are called sorts (data type names).
  - For each $\alpha = s_1 \dots s_n \in \mathcal{S}^*$, a finite set (of predicate symbols) $\mathcal{P}_\alpha \subseteq \mathit{ALPH} \setminus (\mathit{LOG} \cup \mathit{VARS})$.
  - For each $\alpha \in \mathcal{S}^*$ and $s \in \mathcal{S}$, a set (of function symbols) $\mathcal{F}_{\alpha,s} \subseteq \mathit{ALPH} \setminus (\mathit{LOG} \cup \mathit{VARS})$.
- For each $\alpha \in \mathcal{S}^*$ and $s_1, s_2 \in \mathcal{S}$, $s_1 \neq s_2$, it must hold that $\mathcal{F}_{\alpha,s_1} \cap \mathcal{F}_{\alpha,s_2} = \emptyset$.

Signatures (2)

- A sort is a data type name, e.g. `int`, `string`, `person`.
- A predicate is something that can be true or false for given input values, e.g. <, `substring_of`, `female`.
- If $p \in \mathcal{P}_\alpha$, then $\alpha = s_1, \ldots, s_n$ are called the argument sorts of p.
- For example:
  - $< \in \mathcal{P}_{\texttt{int int}}$, also written as `<(int, int)`.
  - `female` $\in \mathcal{P}_{\texttt{person}}$, also written as `female(person)`.

Signatures (3)

- The number of argument sorts (length of α) is called the arity of a predicate symbol, e.g.:
  - $<$ is a predicate symbol of arity 2.
  - `female` is a predicate symbol of arity 1.
- Predicates of arity 0 are called propositional constants, or simply propositions. E.g.:
  - `the_sun_is_shining`,
  - `i_am_working`.
- The symbol ϵ is used to denote the empty sequence. The set $\mathcal{P}_\epsilon$ contains the propositional constants.

Signatures (4)

- The same symbol p can be element of several $\mathcal{P}_\alpha$ (overloaded predicate), e.g.
  - $< \in \mathcal{P}_{\texttt{int int}}$.
  - $< \in \mathcal{P}_{\texttt{string string}}$ (lexicographic order).
- This means that there are actually two different predicates that have the same name.

Signatures (5)

- A function is something that returns a value for given input values, e.g. `+`, `age`, `first_name`.
- A function symbol in $\mathcal{F}_{\alpha,s}$ has argument sorts α and result sort s, e.g.
  - $+ \in \mathcal{F}_{\mathtt{int}\,\mathtt{int},\,\mathtt{int}}$, also written as `+(int, int): int`.
  - `age` $\in \mathcal{F}_{\mathtt{person},\,\mathtt{int}}$, also written as `age(person): int`.

Signatures (6)

- A function with 0 arguments is called a constant.
- Examples of constants:
  - $1 \in \mathcal{F}_{\epsilon,\mathtt{int}}$, also written as `1: int`.
  - $\mathtt{'Ann'} \in \mathcal{F}_{\epsilon,\mathtt{string}}$, also written as `'Ann': string`.
- For data types (e.g., `int`, `string`), it is usual that every possible value can be denoted by a constant.

Signatures (7)

- A signature specifies the application-specific symbols that are used to talk about the domain of discourse (a part of the real world that is to be modeled in the database).
- The above definition is for a multi-sorted (typed) logic.
  One can also use an unsorted logic.

Signatures (8)
Example

- $\mathcal{S} = \{\texttt{person}, \texttt{string}\}$.
- $\mathcal{F}$ consists of
  - constants of sort $\texttt{person}$, e.g. $\texttt{arno}$, $\texttt{birgit}$, $\texttt{chris}$.
  - infinitely many constants of sort $\texttt{string}$, e.g. $''$, $'\texttt{a}'$, $'\texttt{b}'$, $\dots$, $'\texttt{Arno}'$, $\dots$
  - function symbols $\texttt{first\_name}(\texttt{person}) : \texttt{string}$ and $\texttt{last\_name}(\texttt{person}) : \texttt{string}$.
- $\mathcal{P}$ consists of
  - a predicate $\texttt{married\_to}(\texttt{person}, \texttt{person})$.
  - predicates $\texttt{male}(\texttt{person})$ and $\texttt{female}(\texttt{person})$.

Signatures (9)
Definition

- A signature $\Sigma' = (\mathcal{S}', \mathcal{P}', \mathcal{F}')$ is an extension of a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ iff
  - $\mathcal{S} \subseteq \mathcal{S}'$,
  - for every $\alpha \in \mathcal{S}^*$: $\mathcal{P}_\alpha \subseteq \mathcal{P}'_\alpha$,
  - for every $\alpha \in \mathcal{S}^*$ and $s \in \mathcal{S}$: $\mathcal{F}_{\alpha,s} \subseteq \mathcal{F}'_{\alpha,s}$.
- I.e. an extension of Σ' adds new symbols to Σ.

Interpretations (1)

Definition

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

Interpretations (2)

Note

- Empty domains cause certain problems, therefore it is usual to exclude them.
- But in databases, domains can be empty (e.g. a set of persons when the database was just created).

Interpretations (3)

- The relation $\mathcal{I}[p]$ is also called the extension of p (in $\mathcal{I}$).
- Formally, predicate and relation are not the same, but isomorphic notions.
- For instance, $\texttt{married\_to}(\texttt{X}, \texttt{Y})$ is true in $\mathcal{I}$ if and only if $(\texttt{X}, \texttt{Y}) \in \mathcal{I}[\texttt{married\_to}]$.
- Another Example: $(3, 5) \in \mathcal{I}[<]$ means simply $3 < 5$.

Interpretations (4)

Example interpretation for signature on Slide 79

- $\mathcal{I}[\texttt{person}]$ is the set of Arno, Birgit, and Chris.
- $\mathcal{I}[\texttt{string}]$ is the set of all strings, e.g. $'\text{a}'$.
- $\mathcal{I}[\texttt{arno}]$ is Arno.
- For the string constants, $\mathcal{I}$ is the identity mapping.
- $\mathcal{I}[\texttt{first\_name}]$ maps e.g. Arno to $'\texttt{Arno}'$.
- $\mathcal{I}[\texttt{last\_name}]$ maps all three persons to $'\texttt{Schmidt}'$.
- $\mathcal{I}[\texttt{married\_to}] = \{(\text{Birgit}, \text{Chris}),\ (\text{Chris}, \text{Birgit})\}$.
- $\mathcal{I}[\texttt{male}] = \{(\text{Arno}),\ (\text{Chris})\}$, $\mathcal{I}[\texttt{female}] = \{(\text{Birgit})\}$.

Relational 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 a signature Σ) and their meaning (in an interpretation $\mathcal{I}$).
- For every value $d \in \mathcal{I}[s]$, there is at least one constant c with $\mathcal{I}[c] = d$.

- The DB schema in the relational model then adds further predicate symbols (relation symbols).
- The DB state interprets these by finite relations.

Relational Databases (2)

Example

- In a relational database for storing homework results, there might be three predicates/relations:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`

- Here, we treat the "domain calculus" version of the relational model.

Outline

Variable Declaration (1)

- Definition
  - Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be a signature.
  - A variable declaration for Σ is a partial mapping $\nu\colon \mathit{VARS} \to \mathcal{S}$
- 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)

Example

- A variable declaration simply defines which variables are available and what are their sorts, e.g.
  $\nu = \{\mathtt{SID}/\mathtt{int},\ \mathtt{Points}/\mathtt{int},\ \mathtt{E}/\mathtt{exercise}\}$.
- Of course, each variable must have a unique sort.

Variable Declaration (3)

- Definition
  - Let ν be a variable declaration, $X \in \mathit{VARS}$, 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)

- Terms are syntactic constructs that can be evaluated to a value (a number, a string, an exercise).
- There are three kinds of terms:
  - constants, e.g. `1`, `'abc'`, `arno`,
  - variables, e.g. `X`,
  - composed terms, consisting of a function symbol applied to argument terms, e.g. `last_name(arno)`.
- In programming languages, terms are also called expressions.

Terms (2)

Definition

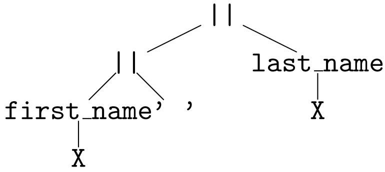
- Let $\Sigma{=}(\mathcal{S},\mathcal{P},\mathcal{F})$ be a signature and ν a variable declaration for Σ.
- The set $TE_{\Sigma,\nu}(s)$ of terms of sort s is recursively defined as follows:
  - Every variable $V \in VARS$ with $\nu(V) = s$ is a term of sort s.
  - Every constant $c \in \mathcal{F}_{\epsilon,s}$ is a term of sort s.
  - If t_1 is a term of sort s_1, ..., t_n is a term of sort s_n, and $f \in \mathcal{F}_{\alpha,s}$ with $\alpha = s_1 \dots s_n$, $n \geq 1$, then $f(t_1, \dots, t_n)$ is a term of sort s.
  - Nothing else is a term of sort s.
- Each term can be constructed by a finite number of applications of the above rules.
- Let $TE_{\Sigma,\nu} := \bigcup_{s\in\mathcal{S}} TE_{\Sigma,\nu}(s)$ be the set of all terms.

Terms (3)

- Certain functions are also written as infix operators, e.g. $\mathtt{X}+\mathtt{1}$ instead of the official notation $+(\mathtt{X},\mathtt{1})$.
- Functions of arity 1 can be written in dot-notation, e.g. "`X.first_name`" instead of "`first_name(X)`".
- Such "syntactic sugar" is useful in practice, but not important for the theory of logic.
- In the following, the above abbreviations are used.

Terms (4)

- Terms can be visualized as operator trees
  ("||" is in SQL the function for string concatenation):



Terms (5)

Exercise

- Which of the following are legal terms (given the signature on slide 79 and a variable declaration ν with $\nu(\mathtt{X}) = \mathtt{string}$)?
  - ☐ `arno`
  - ☐ `first_name`
  - ☐ `first_name(X)`
  - ☐ `firstname(arno, birgit)`
  - ☐ `married_to(birgit, chris)`
  - ☐ `X`

Atomic Formulas (1)

- Formulas are syntactic expressions that can be evaluated to a truth value (true or false), e.g.

$$1 \leq X \wedge X \leq 10.$$

- Atomic formulas are the basic building blocks of such formulas (comparisons etc.).
- Atomic formulas can have the following forms:
  - A predicate symbol applied to terms, e.g. `married_to(birgit, X)`.
  - An equation, e.g. `X = chris`.
  - The logical constants $\top$ (true) and $\bot$ (false).

Atomic Formulas (2)

Definition

- Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be a signature and ν a variable declaration for Σ .
- An atomic formula is an expression of one of the following forms:
  - $p(t_1, \ldots, t_n)$ with $p \in \mathcal{P}_\alpha$, $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$, and $t_i \in TE_{\Sigma,\nu}(s_i)$ for $i = 1 \ldots n$.
  - $t_1 = t_2$ with $t_1, t_2 \in TE_{\Sigma,\nu}(s)$, $s \in \mathcal{S}$.
  - $\top$ and $\bot$.
- Let $AT_{\Sigma,\nu}$ be the set of atomic formulas for Σ,ν.

Atomic Formulas (3)

Remarks

- For some predicates, one traditionally uses infix notation, e.g. $X > 1$ instead of $>(X, 1)$.
- For propositional constants, the parentheses can be skipped, e.g. one can write p instead of $p()$.
- Of course, it would be possible to treat "$=$" as a normal predicate, and some authors do that.

Formulas (1)

Definition

- Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ be a signature and ν a variable declaration for Σ.
- 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 (Σ, ν)-formulas, so are $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
  - $(\forall s\, X \colon F)$ and $(\exists s\, X \colon F)$ are in $FO_{\Sigma,\nu}$ if $s \in \mathcal{S}$, $X \in VARS$, and F is a $(\,\Sigma,\ \nu\langle X/s\rangle\,)$-formula.
  - Nothing else is a (Σ, ν)-formula.

Formulas (2)

- The intuitive meaning of the formulas is as follows:
  - $p(t_1 \dots t_n)$: The predicate p is true for the values of the terms $t_1, \dots, t_n$.
  - $\neg F$: "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".
  - $\forall\, s\, X : F$: "for all X (of sort s), F is true".
  - $\exists\, s\, X : F$: "there is an X (of sort s) such that F".

Formulas (3)

- Above, many parentheses are used in order to ensure that formulas have a unique syntactic structure.
- 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. $F_1 \wedge F_2 \wedge F_3$.

Formulas (4)

Abbreviations for Quantifiers

- When there is only one possible sort of a quantified variable, one can leave it out, i.e. write $\forall X : F$ instead of $\forall\, s\, X : F$ (and the same for $\exists$).
- If one quantifier immediately follows another quantifier, one can leave out the colon.
- Instead of a sequence of quantifiers of the same type, e.g. $\forall X_1 \ldots \forall X_n : F$, one can write $\forall X_1 \ldots X_n : F$.

Formulas (5)

- Abbreviation for Inequality
  - $t_1 \neq t_2$ can be used as an abbreviation for $\neg(t_1 = t_2)$.
- Note
  - Some people say "formulae" instead of "formulas".
- Exercise
  - Given a signature with $\leq \in \mathcal{P}_{\mathsf{int}\ \mathsf{int}}$ and $1, 10 \in \mathcal{F}_{\epsilon,\mathsf{int}}$, and a variable declaration with $\nu(X) = \mathtt{int}$.
  - Is $1 \leq X \leq 10$ a syntactically correct formula?

Formulas (6)

Exercise

- Which of the following are syntactically correct formulas (given the signature on Slide 79)?
  - ☐ $\forall\, \mathtt{X}, \mathtt{Y}\colon \mathtt{married\_to}(\mathtt{X}, \mathtt{Y}) \rightarrow \mathtt{married\_to}(\mathtt{Y}, \mathtt{X})$
  - ☐ $\forall\, \mathtt{person}\ \mathtt{P}\colon\ \vee\ \mathtt{male}(\mathtt{P}) \vee \mathtt{female}(\mathtt{P})$
  - ☐ $\forall\, \mathtt{person}\ \mathtt{P}\colon\ \mathtt{arno} \vee \mathtt{birgit} \vee \mathtt{chris}$
  - ☐ $\mathtt{male}(\mathtt{chris})$
  - ☐ $\forall\, \mathtt{string}\ \mathtt{X}\colon\ \exists\, \mathtt{person}\ \mathtt{X}\colon\ \mathtt{married\_to}(\mathtt{birgit}, \mathtt{X})$
  - ☐ $\mathtt{married\_to}(\mathtt{birgit}, \mathtt{chris}) \wedge \vee\, \mathtt{married\_to}(\mathtt{chris}, \mathtt{birgit})$

Closed Formulas

- Definition
  - Let Σ be a signature.
  - A closed formula (for Σ) is a (Σ,ν)-formula for the empty variable declaration ν.
- Exercise
  - Which of the following are closed formulas?
    - ☐ $\mathtt{female}(\mathtt{X}) \wedge \exists\,\mathtt{X}\colon \mathtt{married\_to}(\mathtt{chris}, \mathtt{X})$
    - ☐ $\mathtt{female}(\mathtt{birgit}) \wedge \mathtt{married\_to}(\mathtt{chris}, \mathtt{birgit})$
    - ☐ $\exists\,\mathtt{X}\colon \mathtt{married\_to}(\mathtt{X}, \mathtt{Y})$

Variables in a Term

Definition

- The function *vars* computes the set of variables that occur in a given term t.
  - If t is a constant c: $vars(t) := \emptyset$.
  - If t is a variable V: $vars(t) := \{V\}$.
  - If t has the form $f(t_1 \ldots t_n)$: $vars(t) := \bigcup_{i=1}^{n} vars(t_i)$.

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 $p(t_1 \ldots t_n)$ or $t_1 = t_2$:
    $free(F) := \bigcup_{i=1}^{n} vars(t_i)$.
  - If F is $\top$ or $\bot$: $free(F) := \emptyset$.
  - If F has the form $(\neg G)$: $free(F) := free(G)$.
  - If F has the form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.:
    $free(F) := free(G_1) \cup free(G_2)$.
  - If F has the form $(\forall\, s\, X\colon G)$ or $(\exists\, s\, X\colon G)$: $free(F) := free(G) \setminus \{X\}$.

Variable Assignment (1)

- Definition
  - A variable assignment $\mathcal{A}$ for $\mathcal{I}$ and ν is a partial mapping $\nu\colon \mathit{VARS} \to \bigcup_{s\in\mathcal{S}} \mathcal{I}[s]$.
  - It maps every variable V, for which ν is defined, to a value from $\mathcal{I}[s]$, where $s := \nu(V)$.
- Remark
  - I.e. a variable assignment for $\mathcal{I}$ and ν defines values from $\mathcal{I}$ for the variables that are declared in ν.

Variable Assignment (2)

Example

- Consider the following variable declaration ν:
  $\nu = \{\mathtt{X}/\mathtt{string}, \mathtt{Y}/\mathtt{person}\}$.
- One possible variable assignment is
  $\mathcal{A} = \{\mathtt{X}/\mathit{abc}, \mathtt{Y}/\mathit{Chris}\}$.

Variable Assignment (3)

- Definition
  - $\mathcal{A}\langle X/d\rangle$ denotes a variable assignment $\mathcal{A}'$ that agrees with $\mathcal{A}$ except that $\mathcal{A}'(X) = d$.
- Example
  - Given the variable declaration on the last slide, $\mathcal{A}\langle \mathtt{Y}/\text{Birgit}\rangle$ is: $\mathcal{A}\langle \mathtt{Y}/\text{Birgit}\rangle = \{\mathtt{X}/abc, \mathtt{Y}/\text{Birgit}\}$.

Value of a Term

Definition

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

$$\langle \mathcal{I}, \mathcal{A} \rangle[t] := \mathcal{I}[f, s_1 \dots s_n]\big(\langle \mathcal{I}, \mathcal{A} \rangle[t_1], \dots, \langle \mathcal{I}, \mathcal{A} \rangle[t_n]\big).$$

Truth of a Formula (1)

Definition

- Let Σ be a signature, ν a variable declaration for Σ, $\mathcal{I}$ a Σ-interpretation, and $\mathcal{A}$ a variable assignment for $(\mathcal{I}, \nu)$.
- The truth value $\langle\mathcal{I}, \mathcal{A}\rangle[F] \in \{\mathsf{f}, \mathsf{t}\}$ of a formula F in $(\mathcal{I}, \mathcal{A})$ is defined as follows (f means false, t true):
  - If F is an atomic formula $p(t_1 \dots t_\mathrm{n})$ with terms t_i of sort s_i:

$$\langle\mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} \mathsf{t} & \text{if } \big(\langle\mathcal{I}, \mathcal{A}\rangle[t_1], \dots, \langle\mathcal{I}, \mathcal{A}\rangle[t_n]\big) \in \mathcal{I}[p, s_1 \dots s_n] \\ \mathsf{f} & \text{otherwise.} \end{cases}$$

  - (continued on next three slides)

Truth of a Formula (2)

Definition, continued

- Truth value of a formula, continued:
  - If F is an atomic formula $t_1 = t_2$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle[t_1] = \langle \mathcal{I}, \mathcal{A} \rangle[t_2] \\ \mathrm{f} & \text{else.} \end{cases}$$

  - If F is $\top$: $\langle \mathcal{I}, \mathcal{A} \rangle[F] := \mathrm{t}$.
  - If F is $\bot$: $\langle \mathcal{I}, \mathcal{A} \rangle[F] := \mathrm{f}$.
  - If F is of the from $(\neg G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle[G] = 0 \\ \mathrm{f} & \text{else.} \end{cases}$$

Truth of a Formula (3)

Definition, continued

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

G_1	G_2	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
f	f	f	f	t	t	t
f	t	f	t	f	t	f
t	f	f	t	t	f	f
t	t	t	t	t	t	t

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

Truth of a Formula (4)

Definition, continued

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

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A}\langle X/d \rangle\rangle[G] = \mathrm{t} \\ & \quad \text{for all } d \in \mathcal{I}[s] \\ \mathrm{f} & \text{otherwise.} \end{cases}$$

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

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathrm{t} & \text{if } \langle \mathcal{I}, \mathcal{A}\langle X/d \rangle\rangle[G] = \mathrm{t} \\ & \quad \text{for at least one } d \in \mathcal{I}[s] \\ \mathrm{f} & \text{otherwise.} \end{cases}$$

Model (1)
Definition

- If $\langle \mathcal{I}, \mathcal{A} \rangle [F] = \mathrm{t}$, one also writes $\langle \mathcal{I}, \mathcal{A} \rangle \models F$.
- 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 $\langle \mathcal{I}, \mathcal{A} \rangle [F] = \mathrm{t}$ for all variable declarations $\mathcal{A}$.
- If $\mathcal{I} \models F$, one says that $\mathcal{I}$ satisfies F.
- 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 there is an interpretation $\mathcal{I}$ and a variable assignment $\mathcal{A}$ such that $(\mathcal{I}, \mathcal{A}) \models F$ (it has a model).

  Otherwise it is called inconsistent.
- A (Σ, ν)-formula F is called a tautology iff for all Σ-interpretations $\mathcal{I}$ and (Σ, ν)-variable assignments $\mathcal{A}$, we have $(\mathcal{I}, \mathcal{A}) \models F$.

Model (3)
Exercise

- Consider the interpretation on Slide 84:
  - $\mathcal{I}[\texttt{person}] = \{\text{Arno}, \text{Birgit}, \text{Chris}\}$.
  - $\mathcal{I}[\texttt{married\_to}] = \{(\text{Birgit}, \text{Chris}),\ (\text{Chris}, \text{Birgit})\}$.
  - $\mathcal{I}[\texttt{male}] = \{(\text{Arno}),\ (\text{Chris})\}$,
    $\mathcal{I}[\texttt{female}] = \{(\text{Birgit})\}$.
- Which of the following formulas are true in $\mathcal{I}$?
  - ☐ $\forall\,\texttt{person X}: \texttt{male}(\texttt{X}) \leftrightarrow \neg\texttt{female}(\texttt{X})$
  - ☐ $\forall\,\texttt{person X}: \texttt{male}(\texttt{X}) \vee \neg\texttt{male}(\texttt{X})$
  - ☐ $\exists\,\texttt{person X}: \texttt{female}(\texttt{X}) \wedge \neg\exists\,\texttt{person Y}: \texttt{married\_to}(\texttt{X},\texttt{Y})$
  - ☐ $\exists\,\texttt{person X}, \texttt{person Y}, \texttt{person Z}:\ \texttt{X} = \texttt{Y} \wedge \texttt{Y} = \texttt{Z} \wedge \texttt{X} \neq \texttt{Z}$

Outline

Databases and Logic (1)

Data values

- The DBMS defines a datatype signature $\Sigma_{\mathcal{D}}$ together with an interpretation $\mathcal{I}_{\mathcal{D}}$, to stipulate the following:
  - For each data type (sort), name and a (non-empty) domain of admissible values.
  - Names of constants (e.g. `123`, `'abc'`) interpreted by a corresponding elements in their data type domain.
  - Names of functions on data types (e.g. `+`, `strlen`) together with their domain and range sorts, interpreted by corresponding functions on domain and range.
  - Names of predicates on data types (e.g. `<`, `odd`), interpreted by corresponding relations on domain and range.

Databases and Logic (2)

- Two formal query languages for the relational model:
  - tuple calculus (with variables for whole tuples)
  - domain calculus (with variables for data values)
- Both are rooted in mathematical logic and portray formal perspectives on relational models.
- Both are equivalent in expressive power but use different logical constructs.
- We consider tuple and domain calculus as restricted variants of first order logic.
- The relational model is formally embedded in first order logic.
- This embedding determines the required restrictions on signatures and interpretations.

Relational Databases (1)

- In relational databases, data is stored as tables, e.g.

Student		
SID	FirstName	LastName
101	Lisa	Weiss
102	Michael	Schmidt
103	Daniel	Sommer
104	Iris	Meier

- Rows are often seen formally as "tuples".

Relational Databases (2)

- In logic, we can formally define the access to table rows in two different ways:
  - Domain Calculus (DC)
    A table with n rows corresponds to an n-ary predicate:
    $p(t_1, \dots, t_n)$ is true iff

t_1	$\cdots$	t_n

    is a row in the table.
  - Tuple Calculus (TC)
    A table with n rows corresponds to a sort with n (access) functions that map to the values of the columns.

Relational Databases (3)

Example

- DC would use a predicate `student`:
  - `student(101, 'Lisa', 'Weiss')` would be true
  - `student(200, 'Martin', 'Mueller')` false
- TC would use a sort `student` accompanied by (access) functions `sid`, `first_name`, `last_name`.
  - For an `X` of sort `student`, we then have that: `sid(X)=101`, `first_name(X)='Lisa'`, and `last_name(X)='Weiss'`.

Relational Databases (1)

Domain Calculus

- 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 a signature Σ) and their meaning (in an interpretation $\mathcal{I}$).
- For every value $d \in \mathcal{I}[s]$, there is at least one constant c with $\mathcal{I}[c] = d$.

Relational Databases (2)

Domain Calculus

- The DB schema in the relational model then adds further predicate symbols (relation symbols).
- The DB state interprets these by finite relations.

- Thus, the main restrictions of the relational model are:
  - No new sorts (types),
  - No new function symbols and constants,
  - New predicate symbols can only be interpreted by finite relations.

Relational Databases (3)

Domain Calculus, Example

- In a relational database for storing homework results, there might be three predicates/relations:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`

Relational Databases (4)

Domain Calculus

Student		
SID	FirstName	LastName
101	Lisa	Weiss
102	Michael	Schmidt
103	Daniel	Sommer
104	Iris	Meier

Result		
SID	ENO	Points
101	1	10
101	2	8
102	1	9
102	2	9
103	1	5

Exercise	
ENO	MaxPt
1	10
2	10

Relational Database (1)

Tuple Calculus

- In TC, a DBMS also defines a set of data types (with constants, functions, predicates).
- The DB schema adds the following:
  - sorts, one per relation (table)
  - unary functions, each mapping from a sort to a data type, one function per column

Relational Databases (2)

Tuple Calculus, Example

- E.g, in the exercise-results DB there is sort `student` with the functions
  - `sid(student): int`
  - `first_name(student): string`
  - `last_name(student): string`

Relational Databases (3)

Tuple Calculus

- E.g. $\mathcal{I}[\texttt{student}]$ contains tuple

$$t = (101, "Lisa", "Weiss")$$

- Then, we have $\mathcal{I}[\texttt{sid}](t) = 101$.
- As usual, these new sorts are also finite (possibly empty) sets.

Formulas in Databases

- The DBMS defines a signature $\Sigma_{\mathcal{D}}$ and an interpretation $\mathcal{I}_{\mathcal{D}}$ for the built-in data types (`string`, `int`, ...).
- Then the database schema extends $\Sigma_{\mathcal{D}}$ to the signature Σ of all symbols that can be used in, e.g., queries.
- A database state is then an interpretation $\mathcal{I}$ for the extended signature Σ.
- Formulas are used in databases as:
  - Integrity constraints
  - Queries
  - Definitions of derived symbols (views).

Integrity Constraints (1)

- Not all interpretations are reasonable DB states.
- For instance, in the old world, a person could only be male or female, but not both.
  Therefore, the following two formulas must be satisfied:
  - $\forall\, \texttt{person}\, \texttt{X}:\ \texttt{male}(\texttt{X}) \lor \texttt{female}(\texttt{X})$
  - $\forall\, \texttt{person}\, \texttt{X}:\ \neg\, \texttt{male}(\texttt{X}) \lor \neg\, \texttt{female}(\texttt{X})$
- These are examples of integrity constraints.

Integrity Constraints (2)

- An integrity constraint is a closed formula.
- A set of integrity constraints is specified as part of the database schema.
- A database state (an interpretation) is called valid iff it satisfies all integrity constraints.

Integrity Constraints (3)

Keys I

- Objects are often identified by unique data values (numbers, names).
- For example, there should never be two different objects of type `student` with the same `sid` (in TC):

$$\forall\, \texttt{student}\ \texttt{X},\ \texttt{student}\ \texttt{Y}\colon\ \texttt{sid}(\texttt{X}) = \texttt{sid}(\texttt{Y}) \ \rightarrow\ \texttt{X} = \texttt{Y}$$

- Alternative, equivalent formulation:

$$\neg\,\exists\, \texttt{student}\ \texttt{X},\ \texttt{student}\ \texttt{Y}\colon\ \texttt{sid}(\texttt{X}) = \texttt{sid}(\texttt{Y}) \ \wedge\ \texttt{X} \neq \texttt{Y}$$

Integrity Constraints (4)

Keys II

- In the relational schema (in DC on Slide 86) a predicate of arity 3 is used to store the student data.
- The first argument (SID) uniquely identifies the values of the other arguments (first name, last name):

$$\begin{array}{l}\forall\, \texttt{int ID},\ \texttt{string F1},\ \texttt{string F2},\ \texttt{string L1},\ \texttt{string L2}: \\ \quad \texttt{student}(\texttt{ID}, \texttt{F1}, \texttt{L1}) \land \texttt{student}(\texttt{ID}, \texttt{F2}, \texttt{L2}) \ \rightarrow \\ \qquad \texttt{F1} = \texttt{F2} \ \land\ \texttt{L1} = \texttt{L2}\end{array}$$

- Since keys are so common, each data model has a special notation for them (one does not actually have to write such formulas).

Queries (1)

Domain Calculus

- In DC, a query is an expression of the form

  $$\{s_1\, X_1, \ldots, s_n\, X_n \mid F\},$$

  where F is a formula for the given DB signature Σ and the variable declaration $\{X_1/s_1, \ldots, X_n/s_n\}$.
- The query asks for all variable assignments $\mathcal{A}$ for the result variables $X_1, \ldots, X_n$ that make the formula F true in the given database state $\mathcal{I}$.

Queries (2)

Domain Calculus, Examples I

- Consider the schema on Slide 86:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`
- Who got at least 8 points for Homework 1?

$$\{\texttt{string FName}, \texttt{string LName} \mid \exists\, \texttt{int SID}, \texttt{int P}: \\ \texttt{student}(\texttt{SID}, \texttt{FName}, \texttt{LName}) \wedge \\ \texttt{result}(\texttt{SID}, 1, \texttt{P}) \wedge \texttt{P} \geq 8\}$$

Queries (1)

Domain Calculus, Examples I

- The formulas `student(S, FirstName, LastName)` and `result(S, 1, P)` correspond to the table lines:

Student		
SID	FirstName	LastName
S	FirstName	LastName

Result		
SID	ENO	Points
S	1	P

- By the same variable `S` the entries are "joined" in the two tables. They must refer to the same student.

Queries (3)

Domain Calculus, Examples II

- Print all results for Ann Smith:

$$\{\texttt{int ENO}, \texttt{int Points} \mid \exists\, \texttt{int SID}: \\ \quad \texttt{student}(\texttt{SID}, '\texttt{Ann}', '\texttt{Smith}') \wedge \\ \quad \texttt{result}(\texttt{SID}, \texttt{ENO}, \texttt{Points})\}$$

- Who has not yet submitted Exercise 2?

$$\{\texttt{string FName}, \texttt{string LName} \mid \\ \quad \exists\, \texttt{int SID}: \texttt{student}(\texttt{SID}, \texttt{FName}, \texttt{LName}) \wedge \\ \qquad \neg \exists\, \texttt{int P}: \texttt{result}(\texttt{SID}, 2, \texttt{P})\}$$

Queries (1)
Tuple Calculus

- In TC, a query is an expression of the form

  $$\{t_1, \ldots, t_k \; [s_1 \, X_1, \ldots, s_n \, X_n] \mid F\},$$

  where F is a formula and the t_i are terms for the given DB signature Σ and the variable declaration $\{X_1/s_1, \ldots, X_n/s_n\}$.
- The DBMS will print the values $\langle \mathcal{I}, \mathcal{A} \rangle [t_i]$ of the terms t_i for every variable assignments $\mathcal{A}$ for the result variables $X_1, \ldots, X_n$ such that $\langle \mathcal{I}, \mathcal{A} \rangle \models F$.

Queries (2)
Tuple Calculus, Example

- Consider the schema on Slide 130 in TC:
  - Sort `student` with access functions for rows:
    `sid(student): int`,
    `first_name(student): string`,
    `last_name(student): string`.
  - Sort `result` with functions `sid`, `eno`, `points`.
  - Sort `exercise` with functions `eno`, `maxpt`.

Queries (3)

Tuple Calculus

- Who has at least 8 points for homework 1?

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \exists\, \texttt{result R}:\ \texttt{R.eno} = \texttt{1} \wedge \texttt{R.sid} = \texttt{S.sid} \wedge \texttt{R.points} \geq \texttt{8}\}$$

- Variables run in tuple calculus over table rows (tuples).
- Equations are typically used to link table rows.

Queries (4)

Tuple Calculus

- We could have formulated the question with a variable for the task itself (who has at least 8 points for homework 1):

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \\ \exists\, \texttt{result R}, \texttt{exercise E}: \\ \texttt{E.eno} = 1 \land \texttt{R.eno} = \texttt{E.eno} \land \\ \texttt{R.sid} = \texttt{S.sid} \land \texttt{R.points} \geq 8\}$$

- This is logically equivalent.

Queries (5)

Tuple Calculus, Example II

- Who hasn't submitted exercise 2 yet?

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \neg\exists\,\texttt{result R}:\ \texttt{R.sid} = \texttt{S.sid} \wedge \texttt{R.eno} = 2\}$$

- Other possible solution:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \forall\,\texttt{result R}:\ \texttt{R.sid} = \texttt{S.sid} \rightarrow \texttt{R.eno} \neq 2\}$$

- Another solution:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \forall\,\texttt{result R}:\ \texttt{R.eno} = 2 \rightarrow \texttt{R.sid} \neq \texttt{S.sid}\}$$

Queries (6)

Tuple Calculus

- The tuple calculus is very close to SQL.
  E.g. who has ≥ 8 points for homework 1?

$$
\begin{aligned}
\{&\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S},\ \texttt{result R}] \mid \\
&\quad \texttt{R.eno} = 1 \land \\
&\quad \texttt{R.sid} = \texttt{S.sid} \land \texttt{R.points} \geq 8\}
\end{aligned}
$$

- Same query in SQL:

```
SELECT S.FirstName, S.LastName
FROM   Student S, Result R
WHERE  R.ENO = 1
AND    R.SID = S.SID
AND    R.Points >= 8
```

Queries (7)
Tuple Calculus

- Variant with explicit existential quantifier:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \exists\, \texttt{result R}:\ \texttt{R.eno} = 1 \wedge \texttt{R.sid} = \texttt{S.sid} \wedge \texttt{R.points} \geq 8\}$$

- A subquery corresponds to this in SQL:

```
SELECT S.FirstName, S.LastName
FROM   Student S
WHERE  EXISTS (SELECT *
               FROM   Result R
               WHERE  R.ENO = 1
               AND    R.SID = S.SID
               AND    R.Points >= 8)
```

Boolean Queries

- A Boolean query is a closed formula F.
- The system prints "yes" if $\mathcal{I} \models F$ and "no" otherwise.

Outline

Implication

Definition/Notation

- A formula or set of formulas Φ (logically) implies a formula or set of formulas G iff every model $\langle \mathcal{I}, \mathcal{A} \rangle$ of Φ is also a model of G.
- In this case we write $\Phi \vdash G$.

Equivalence (1)

Definition

- Two (sets of) (Σ, ν)-formulas F_1 and F_2 are (logically) equivalent iff for every Σ-interpretation $\mathcal{I}$ and every $(\mathcal{I}, \nu)$-variable assignment $\mathcal{A}$

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

- In this case we write $F_1 \equiv F_2$.

Equivalence (2)

- F_1 and F_2 are equivalent iff $F_1 \vdash F_2$ and $F_2 \vdash F_1$.
- "Equivalence" of formulas is an equivalence relation, i.e. it is reflexive, symmetric, and transitive.
- Suppose that G_1 results from G_2 by replacing a subformula F_1 by F_2 and let $F_1 \equiv F_2$.
  Then $G_1 \equiv G_2$.
- If $F \vdash G$, then $F \wedge G \equiv F$.

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.

Some Equivalences (4)

- Removing Negation:
  - $\neg(t_1 < t_2) \equiv t_1 \geq t_2$
  - $\neg(t_1 \leq t_2) \equiv t_1 > t_2$
  - $\neg(t_1 = t_2) \equiv t_1 \neq t_2$
  - $\neg(t_1 \neq t_2) \equiv t_1 = t_2$
  - $\neg(t_1 \geq t_2) \equiv t_1 < t_2$
  - $\neg(t_1 > t_2) \equiv t_1 \leq t_2$

Some Equivalences (5)

- Law of the excluded middle:
  - $F \vee \neg F \quad \equiv \quad \top \quad$ (always true)
  - $F \wedge \neg F \quad \equiv \quad \bot \quad$ (always false)
- Simplifications of formulas with logical constants $\top$ (true) and $\bot$ (false):
  - $F \wedge \top \equiv F \quad F \wedge \bot \equiv \bot$
  - $F \vee \top \equiv \top \quad F \vee \bot \equiv F$
  - $\neg\top \equiv \bot \quad \neg\bot \equiv \top$

Some Equivalences (6)

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

Some Equivalences (7)

- Moving quantifiers: If $X \notin \mathit{free}(F)$:
  - $\forall s\, X\colon\ (F \vee G) \equiv F \vee (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \wedge G) \equiv F \wedge (\exists s\, X\colon\ G)$

  If in addition $\mathcal{I}[s]$ cannot be empty:
  - $\forall s\, X\colon\ (F \wedge G) \equiv F \wedge (\forall s\, X\colon\ G)$
  - $\exists s\, X\colon\ (F \vee G) \equiv F \vee (\exists s\, X\colon\ G)$
- Removing unnecessary quantifiers: If $X \notin \mathit{free}(F)$ and $\mathcal{I}[s]$ cannot be empty:
  - $\forall s\, X\colon\ F \equiv F$
  - $\exists s\, X\colon\ F \equiv F$

Some Equivalences (8)

- Exchanging quantifiers: If $X \neq Y$:
  - $\forall s_1 X\colon (\forall s_2 Y\colon F) \equiv \forall s_2 Y\colon (\forall s_1 X\colon F)$
  - $\exists s_1 X\colon (\exists s_2 Y\colon F) \equiv \exists s_2 Y\colon (\exists s_1 X\colon F)$
- Renaming bound variables: If $Y \notin \mathit{free}(F)$ and F' results from F by replacing every free occurrence of X in F by Y:
  - $\forall s X\colon F \equiv \forall s Y\colon F'$
  - $\exists s X\colon F \equiv \exists s Y\colon F'$

Some Equivalences (9)

- Equality is an equivalence relation:
  - $t = t \equiv \top$ (reflexivity)
  - $t_1 = t_2 \equiv t_2 = t_1$ (symmetry)
  - $t_1 = t_2 \wedge t_2 = t_3 \equiv t_1 = t_2 \wedge t_2 = t_3 \wedge t_1 = t_3$ (transitivity)
- Compatibility to function and predicate symbols:
  - $f(t_1, \dots, t_n) = t \wedge t_i = t'_i \quad \equiv$
    $f(t_1, \dots, t_{i-1}, t'_i, t_{i+1}, \dots, t_n) = t \wedge t_i = t'_i$
  - $p(t_1, \dots, t_n) \wedge t_i = t'_i \quad \equiv$
    $p(t_1, \dots, t_{i-1}, t'_i, t_{i+1}, \dots, t_n) \wedge t_i = t'_i$

Normal Forms (1)

Definition

- A formula F is in Prenex Normal Form iff it is closed and has the form

$$\Theta_1\, s_1\, X_1 \;\ldots\; \Theta_n\, s_n\, X_n : \quad G$$

  where $\Theta_1, \ldots, \Theta_n \in \{\forall, \exists\}$ and G is quantifier-free.

- A formula F is in Disjunctive Normal Form iff it is in Prenex Normal Form, and G has the form

$$(G_{1,1} \wedge \cdots \wedge G_{1,k_1}) \;\vee \cdots \vee\; (G_{n,1} \wedge \cdots \wedge G_{n,k_n}),$$

  where each $G_{i,j}$ is an atomic formula or a negated atomic formula.

Normal Forms (2)

- Conjunctive Normal Form is like disjunctive normal form, but G must have the form

$$(G_{1,1} \vee \cdots \vee G_{1,k_1}) \wedge \cdots \wedge (G_{n,1} \vee \cdots \vee G_{n,k_n}).$$

- Under the assumption of non-empty domains, every formula can be equivalently translated into prenex normal form, disjunctive normal form, and conjunctive normal form.

Outline

Motivation

- Functions are often only partially defined e.g.
  - division by 0,
  - square root of a negative number,
  - integer overflow.
- Often, table columns, i.e., attributes of objects are missing values e.g. not every customer
  - has a fax machine or
  - discloses his birthday.
- Hence, partial function are relevant in real-life.

Interpretation

- Formally, a function symbol $f(s_1, \ldots, s_n)$: s is interpreted as function

  $$\mathcal{I}[f]\colon\ \mathcal{I}[s_1] \times \cdots \times \mathcal{I}[s_n] \ \rightarrow\ \mathcal{I}[s] \cup \{\mathit{null}\},$$

  where *null* is a designated value (different from all elements in $\mathcal{I}[s]$).
- For term evaluation, *null*-values are propagated in a "bottom-up"-fashion: if a function has argument "*null*", it returns "*null*".

Example (1)

- Suppose we also record the semester of students which might not always be known:

Student			
SID	FirstName	LastName	Semester
101	Lisa	Weiss	3
102	Michael	Schmidt	5
103	Daniel	Sommer	
104	Iris	Meier	3

- Consider the following query:

$$\{\mathtt{S.first\_name},\ \mathtt{S.last\_name}\quad [\mathtt{student\ S}] \mid \mathtt{S.semester} \leq 3\}$$

Example (2)

- With SQL semantics, this query would not return Daniel Sommer.
- This is also the case, when querying students in later semesters:

$$\begin{array}{ll}\{\texttt{S.first\_name}, \texttt{S.last\_name} & [\texttt{student S}] \mid \\ & \texttt{S.semester} > 3\}\end{array}$$

- This is (also in SQL) equivalent to query:

$$\begin{array}{ll}\{\texttt{S.first\_name}, \texttt{S.last\_name} & [\texttt{student S}] \mid \\ & \neg(\texttt{S.semester} \leq 3)\}\end{array}$$

Example (3)

- Daniel Sommer would also be omitted by the answer of this query:

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}]\ |\ \texttt{S.semester} \leq 3\ \vee\ \neg(\texttt{S.semester} \leq 3)\}$$

- This violates the law of the excluded middle.
- A two-valued logic with truth-values “true” and “false” does not suffice in this situation.
- A third truth-value, “undefined” (or “null”), is required.

Truth of a Formula (1)

- If F is an atomic formula of form $p(t_1, \ldots, t_n)$ or $t_1 = t_2$, and one of its argument terms t_i evaluates to *null*, then F evaluates to the third truth value u.
- If F is of form $\neg G$, then its truth value is depends on the truth value of G as follows:

G	$\neg G$
f	t
u	u
t	f

Truth of a Formula (2)

- Logical binary connectives are evaluated as follows:

G_1	G_2	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
f	f	f	f	t	t	t
f	u	f	u	u	t	u
f	t	f	t	f	t	f
u	f	f	u	t	u	u
u	u	u	u	u	u	u
u	t	u	t	u	t	u
t	f	f	t	t	f	f
t	u	u	t	t	u	u
t	t	t	t	t	t	t

Truth of a Formula (3)

- The principle is simple: the truth value u is passed on, if the value of the formula is not already determined by the other input value.
- E.g. is $\mathsf{u} \wedge \mathsf{f} = \mathsf{f}$, because it does not matter whether the left input value is t or f.
- In other words: A partial condition, which evaluates to u, should not affect the overall truth value as much as possible.

Truth of a Formula (3)

- An existential statement $\exists\, s\, X\colon G$ is true under $\langle \mathcal{I}, \mathcal{A} \rangle$, iff there exists a value in $d \in \mathcal{I}[s]$ such that

$$\langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle [G] = \mathrm{t}.$$

- Otherwise, false in SQL semantics.
- *null* must not be substituted for X: $\mathit{null} \notin \mathcal{I}[s]$.

Truth of a Formula (4)

- Accordingly, a universal statement $\forall\, s\, X\colon\, G$ is true under $\langle \mathcal{I}, \mathcal{A} \rangle$ iff for each $d \in \mathcal{I}[s]$:
  - $\langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathrm{t}$ or
  - $\langle \mathcal{I}, \mathcal{A}\langle X/d \rangle \rangle[G] = \mathrm{u}$.
- Such a statement is only false if there exists an variable assignment $\mathcal{A}'$ such that $\langle \mathcal{I}, \mathcal{A}' \rangle[G] = \mathrm{f}$, where $\mathcal{A}'$ only differs from $\mathcal{A}$ by the value assigned to X.
- Hence, it holds: $\forall\, s\, X\colon\, G \quad \equiv \quad \neg\exists\, s\, X\, \neg G$.

Equivalences

- Note that some equivalences from two-valued logic do not apply:
  - $t = t$ is not a tautology: if $t = \mathit{null}$, the equation evaluates to u.
  - $F \vee \neg F$ (law of excluded middle).
  - Equivalences with quantifiers that require $\mathcal{I}[s]$ to be not empty.

Check for Null

- To check whether a term evaluates to *null*, one needs another form of atomic formulas.
- t `is null` is true (t) under $\langle \mathcal{I}, \mathcal{A} \rangle$ iff $\langle \mathcal{I}, \mathcal{A} \rangle[t] = \mathit{null}$ and false (f), otherwise.
- For better readability, we may also write t `is not null` instead of $\neg(t$ `is null`$)$.

Total Functions

- Since all functions are partial in this context, one has to explicitly enforce by integrity constraints that a distinct function is total.
- e.g. the last name of students is mandatory

  $\forall$ `student S: S.last_name is not null.`

- Since this is very common, data models usually provide a shorthand, e.g. in SQL we can declare a table row as "`NOT NULL`".

Outline

Summary

- Signature and formulas
- Interpretations and models
- Database signature
  - Datatype signature
  - Domain calculus
  - Tuple calculus
- Integrity constraints (and keys)
- Queries
- Equivalence
- Incomplete information

Bibliography

- The following list of references is compiled from the open source bibliography available at

  `https://github.com/krr-up/bibliography`

- Feel free to submit corrections via pull requests !

[1] S. Abiteboul, R. Hull, and V. Vianu.
Foundations of Databases.
Addison-Wesley, 1995.

[2] C. Aggarwal, editor.
Data Streams — Models and Algorithms, volume 31 of *Advances in Database Systems*.
Springer-Verlag, 2007.

[3] K. Apt, H. Blair, and A. Walker.
Towards a theory of declarative knowledge.
In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann Publishers, 1987.

[4] M. Arenas, L. Bertossi, and J. Chomicki.
Consistent query answers in inconsistent databases.

In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'99)*, pages 68–79. ACM Press, 1999.

[5] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors.
The Description Logic Handbook: Theory, Implementation, and Applications.
Cambridge University Press, 2003.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom.
Models and issues in data stream systems.
In L. Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 1–16. ACM Press, 2002.

[7] C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.
Cambridge University Press, 2003.

[8] S. Ceri, G. Gottlob, and L. Tanca.
Logic Programming and Databases.
Springer-Verlag, 1990.

[9] R. Elmasri and S. Navathe.
Fundamentals of database systems.
Addison-Wesley, 1994.

[10] R. Fagin, J. Ullman, and M. Vardi.
On the semantics of updates in databases. preliminary report.
In *Proceedings of the Second ACM Conference SIGACT-SIGMOD*, pages 352–365, 1983.

[11] H. Gallaire, J. Minker, and J. Nicolas.
Logic and databases: A deductive approach.
Computing Surveys, 16(2):153–185, 1984.

[12] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele.
Potassco User Guide.

University of Potsdam, 2 edition, 2015.

[13] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Answer Set Solving in Practice.
Synthesis Lectures on Artificial Intelligence and Machine Learning.
Morgan and Claypool Publishers, 2012.

[14] M. Gelfond and Y. Kahl.
Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.
Cambridge University Press, 2014.

[15] M. Gelfond and V. Lifschitz.
Classical negation in logic programs and disjunctive databases.
New Generation Computing, 9:365–385, 1991.

[16] H. Katsuno and A. Mendelzon.
On the difference between updating a knowledge database and revising it.

In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 387–394. Morgan Kaufmann Publishers, 1991.

[17] V. Lifschitz.
Closed-world databases and circumscription.
Artificial Intelligence, 27:229–235, 1985.

[18] V. Lifschitz.
Nonmonotonic databases and epistemic queries.
In J. Myopoulos and R. Reiter, editors, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 381–386. Morgan Kaufmann Publishers, 1991.

[19] V. Lifschitz.
Introduction to answer set programming.
Unpublished draft, 2004.

[20] V. Lifschitz, F. van Harmelen, and B. Porter, editors.

Handbook of Knowledge Representation.
Elsevier Science, 2008.

[21] L. Liu and M. Özsu, editors.
Encyclopedia of Database Systems.
Springer-Verlag, 2009.

[22] V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.
Artifical Intelligence. Springer-Verlag, 1993.

[23] J. Minker, editor.
Foundations of Deductive Databases and Logic Programming.
Morgan Kaufmann Publishers, 1988.

[24] R. Reiter.
On closed world data bases.
In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, New York, 1978.

[25] R. Reiter.

Towards a logical reconstruction of relational database theory.
In M. Brodie, J. Myopoulos, and J. Schmidt, editors, *On conceptual modeling: Perspectives from Artificial Intelligence, Datbases and Programming Languages*, pages 191–233. Springer-Verlag, 1984.

[26] R. Reiter.
On asking what a database knows.
In J. Lloyd, editor, *Computational Logic*, pages 96–113. Springer-Verlag, 1990.

[27] J. Ullman.
Principles of Database Systems.
Computer Science Press, Rockville MD, 1982.

[28] J. Ullman.
Principles of Database and Knowledge-Base Systems.
Computer Science Press, 1988.