Part 2: Basic Notions of Mathematical Logic

References:

- Bergmann/Noll: Mathematische Logik mit Informatik-Anwendungen (in German). Springer, 1977.
- Ebbinghaus/Flum/Thomas: Einführung in die mathematische Logik (in German). Spektrum Akademischer Verlag, 1996.
- Tuschik/Wolter: Mathematische Logik, kurzgefasst (in German). Spektrum Akademischer Verlag, 2002.
- Schöning: Logik für Informatiker (in German). BI Verlag, 1992.
- Chang/Lee: Symbolic Logic and Mechanical Theorem Proving. Academic Press, 1973.
- Fitting: First Order Logic and Automated Theorem Proving. Springer, 1995, 2nd Edition.
- Ulf Nilson, Jan Małuszyński: Logic, Programming, and Prolog (2nd Ed.). 1995, [http://www.ida.liu.se/~ulfni/lpp]

Objectives

After completing this chapter, you should be able to:

- explain the basic notions: signature, interpretation, variable assignment, term, formula, model, consistent, implication.
- 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).

Overview

1. Introduction, Motivation, History
2. Signatures, Interpretations
3. Formulas, Models
4. Formulas in Databases
5. Implication, Equivalence

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 wether a statement is logically implied by given statements.

  As far as possible. It turned out that the task is in general undecidable. Actually, also the notion of decidability was developed by logicians. There was no computer science at that time (there was no computer).

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.

    It should be no surprise that an important part of a database schema specifies a signature.

Introduction, Motivation (3)

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 (4)

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 (5)

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 learnt in this simpler, purer environment.
- Experience has shown that students often make logical errors in SQL queries.

    At least, when they are not specifically taught logic. It will be interesting to see whether the situation improves in this term.

Introduction, Motivation (6)

Why it makes sense to learn mathematical logic II:

- SQL changes, and becomes more and more complicated (standards: 1986, 1989, 1992, 1999, 2003).

  At some point in the future, somebody will propose a drastically simpler language that can do the same. Datalog was already such a proposal, but somehow it was not yet successful. But look at Algol 68 and PL/I, then followed by Pascal and C.

- 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/Russel]
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]
∼1969	Question Answering Systems [Green et.al.]
1970	Linear Resolution [Loveland/Luckham]
1970	Relational Data Model [Codd]
∼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]

Overview

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

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

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

  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.

Alphabet (3)

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

Symbol	Alternative	Alt2	Name
$\top$	true	T	
$\bot$	false	F	
$\neg$	not	~	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 of which are called sorts (data type names).
  - ◇ For each $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$, a finite set (of predicate symbols) $\mathcal{P}_\alpha \subseteq ALPH - (LOG \cup VARS)$.
  - ◇ For each $\alpha \in \mathcal{S}^*$ and $s \in \mathcal{S}$, a set (of function symbols) $\mathcal{F}_{\alpha,s} \subseteq ALPH - (LOG \cup VARS)$.
- 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.

  s_1 is the type of the first argument, s_2 of the second, and so on.

- For example:
  - ◇ $< \in \mathcal{P}_{\texttt{int int}}$, also written as `<(int, int)`.
  - ◇ $\texttt{female} \in \mathcal{P}_{\texttt{person}}$, 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.
- There are predicates of arity 0. They 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.
  - $\texttt{<} \in \mathcal{P}_{\texttt{int int}}$.
  - $\texttt{<} \in \mathcal{P}_{\texttt{string string}}$ (lexicographic order, alphabetically before).
- This means that there are actually two different predicates that have the same name.

  The possibility of overloaded predicates is not very important, one could also use two different names, e.g. `lt_int` and `lt_string`. Overloaded predicates complicate the definitions a bit, therefore some authors exclude them. But they permit more natural formulations.

Signatures (5)

- A function is something that returns a value for given input values, e.g. `+`, `age`, `first_name`.

  It is assumed here that functions are defined for all values of the input types. This is not always true in reality, e.g. `5/0` is undefined, and `telefax_no(peter)` might not exist. SQL uses a three-valued logic to treat null values (statements are not always true or false, they might also be undefined). One must always make a compromise between a modeling reality very exactly and finding a sufficiently simple model.

- 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`.
  - ◇ $\mathtt{age} \in \mathcal{F}_{\mathtt{person},\ \mathtt{int}}$, 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.

  But in general, the set of values and the set of constants are different concepts. For instance, it would be possible that there are no constants of type `person`.

Signatures (7)

- In summary, 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.

  Unsorted means really one-sorted. Then $\mathcal{S}$ is not needed, and $\mathcal{P}$ and $\mathcal{F}$ are simply indexed by the arity. E.g. Prolog uses an unsorted logic. This is also common in textbooks about mathematical logic (the definitions are a bit simpler). Since one can represent sorts as predicates of arity 1, this is no real restriction (although a many-sorted logic treats some formulas as illegal, which are legal in one-sorted logic).

Signatures (8)

Example:

- $\mathcal{S} = \{\texttt{person}, \texttt{string}\}$.
- $\mathcal{F}$ 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): string` and `last_name(person): string`.
- $\mathcal{P}$ consists of
  - ◇ a predicate `married_to(person, person)`.
  - ◇ predicates `male(person)` and `female(person)`.

Signatures (9)

Exercise:

- Define a signature for talking about
  - ◇ Books (with authors, title, ISBN)

    It suffices to treat authors simply as a string. An advanced exercise would be to model a list of strings (for books with several authors).
  - ◇ Book reviews (with reviewer, text, stars).

    Every review is for exactly one book.
  - ◇ "stars" can be none, one, two, three.

Signatures (10)

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 $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$.

    In the following, we will write $\mathcal{I}(p)$ instead of $\mathcal{I}(p, \alpha)$ if α is clear from the given signature Σ (i.e. p is not overloaded).
  - ◇ a function $\mathcal{I}(f, \alpha)$: $\mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n) \to \mathcal{I}(s)$ for every $f \in \mathcal{F}_{\alpha,s}$, $s \in \mathcal{S}$, and $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$.
- In the following, we write $\mathcal{I}[\ldots]$ instead of $\mathcal{I}(\ldots)$.

Interpretations (2)

Note:

- Empty domains cause certain problems, therefore it is usual to exclude them.

  Some equivalences do not hold if domains can be empty. For instance, prenex normal form can only be reached under the assumption that domains are not empty. We will explicitly note where non-empty domains are needed.

- But in databases, domains can be empty (e.g. a set of persons when the database was just created).

  This even causes problems in SQL when tuple variables are declared over empty relations.

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.

  A predicate is a mapping to the set $\{true, false\}$ of boolean values, a relation is a subset of a cartesian product $\times$.
- For instance, $\mathtt{married\_to}(\mathtt{X}, \mathtt{Y})$ is true in $\mathcal{I}$ if and only if $(\mathtt{X}, \mathtt{Y}) \in \mathcal{I}[\mathtt{married\_to}]$.
- Another Example: $(3, 5) \in \mathcal{I}[<]$ means simply $3 < 5$.

  In the following, the words "predicate symbol" and "relation symbol" are used interchangably.

Interpretations (4)

Example (Interpretation for Signature on Slide 2-23):

- $\mathcal{I}[\texttt{person}]$ is the set of Arno, Birgit, and Chris.
- $\mathcal{I}[\texttt{string}]$ is the set of all strings, e.g. `'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 `'Arno'`.
- $\mathcal{I}[\texttt{last\_name}]$ maps all three persons to `'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 $\mathcal{I}[c] = d$.

  I.e. all data values are named by constants. This is also known as the domain closure assumption. It is important e.g. for printing data values that are results of queries. Note that in general there can be several constants mapped to the same data value, e.g. 0, 00, -0.

Relational Databases (2)

- The DB schema in the relational model then adds further predicate symbols (relation symbols).

  These are the formal counterpart of the "tables" mentioned in the introduction.

- 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 (database relations) can be modified by insertions, deletions, and updates. But the database relations must have a finite extension. This is natural because data type predicates are implemented by procedures within the DBMS, whereas the predicates from the database schema are typically implemented as files of records.

Relational Databases (3)

- 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.
- In addition, formulas are required to be "domain independent" or "range restricted" (see below).

  I.e. not arbitrary formulas are permitted. This is necessary in order to ensure that formulas can be evaluated in a given interpretation in finite time (although sorts like `int` are interpreted as infinite sets).

Relational Databases (4)

Example:

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

    Argument names were added to explain the meaning of the arguments: The first one is a unique number ("student ID"), the second is the first name of the student with that ID, the third is the last name. E.g. `student(101, 'Ann', 'Smith')` might be true.
  - `exercise(int ENO, int MaxPoints)`

    E.g. `exercise(1, 10)` means: exercise number 1 is worth 10 points.
  - `result(int SID, int ENO, int Points)`

    E.g. `result(101, 1, 9)` means that Ann Smith (the student with number 101) got 9 points for exercise 1.

Relational Databases (5)

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

  There is also a "tuple calculus" version of the relational model, which is even more similar to SQL. The differences are not essential, e.g. queries can automatically be converted in both directions. However, the definitions of the tuple calculus are a bit more complicated, because variables in tuple calculus range over entire tuples (table rows). Typically, database text books define both versions in separate chapters as two different logical formalisms. However, once one has understood one formalism (such as domain calculus), it is very easy to learn the other one. Furthermore, the above formalism can actually treat the part of tuple calculus that is used in SQL (it is like the entity-relationship model without relationships, see below): The difficult part are tuple variables that are not directly bound to a database relation (SQL forbids this and uses `UNION` instead).

Entity-Relationship Model (1)

- 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 ("attributes"),
  - new predicates between entity types, possibly restricted to arity 2 ("relationships").
  - new functions defined on the same entity types as a relationship, returning a data type ("relationship attributes").

Entity-Relationship Model (2)

- The interpretation of the entity types (in the DB state) must always be finite.

  Thus, also attributes and relationships are finite.

- Relationship attribute functions must yield a fixed dummy value if they are called for a combination of input values for which the corresponding relationship is false.

  Queries should be written in such a way that the exact dummy value is not important for the query result. E.g. if f is an attribute of the relationship p, a formula of the form $p(X,Y) \wedge f(X,Y) = Z$ would have this property. For a really clean treatment of relationship attributes, the logic must be extended, but this seems not worth the effort.

Entity-Relationship Model (3)

Example (Homework Points):

- Sorts: `student` and `exercise`.
- Functions:
  - ◇ `sid(student): int`
  - ◇ `first_name(student): string`
  - ◇ `last_name(student): string`.
  - ◇ `eno(exercise): int`
  - ◇ `maxpoints(exercise): int`
- Predicate: `solved(student, exercise)`.

  Function: `points(student, exercise): int`

Overview

1. Introduction, Motivation, History
2. Signatures, Interpretations
3. Formulas, Models
4. Formulas in Databases
5. Implication, Equivalence

Variable Declaration (1)

Definition:

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

Example:

- A variable declaration simply defines which variables are available and what are their sorts, e.g.

ν	
Variable	Sort
SID	int
Points	int
E	exercise

  Of course, each variable must have a unique sort.

- Variable declarations are also written in the form $\nu = \{\texttt{SID}/\texttt{int},\ \texttt{Points}/\texttt{int},\ \texttt{E}/\texttt{exercise}\}$.

Variable Declaration (3)

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)

- 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)`.

    This can be nested to arbitrary depth.
- In programming languages, terms are also called expressions.

Terms (2)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ 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: (nothing else is a term):
  - ◇ 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).
  - ◇ 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 $\alpha = s_1 \ldots s_n$, $n \geq 1$, then $f(t_1, \ldots, t_n)$ is a term of sort s.

Terms (3)

Definition, continued:

- Each term can be constructed by a finite number of applications of the above rules. Nothing else is a term.

  This remark is formally important because the above rules only positively state what is a term, but they do not state what is not a term. Therefore, the definition must be closed.

Definition:

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

Terms (4)

- Certain functions are also written as infix operators, e.g. `X+1` instead of the official notation $+(\mathtt{X}, \mathtt{1})$.

  If one starts with this, one must also talk about precedence rules and using parentheses as necessary.

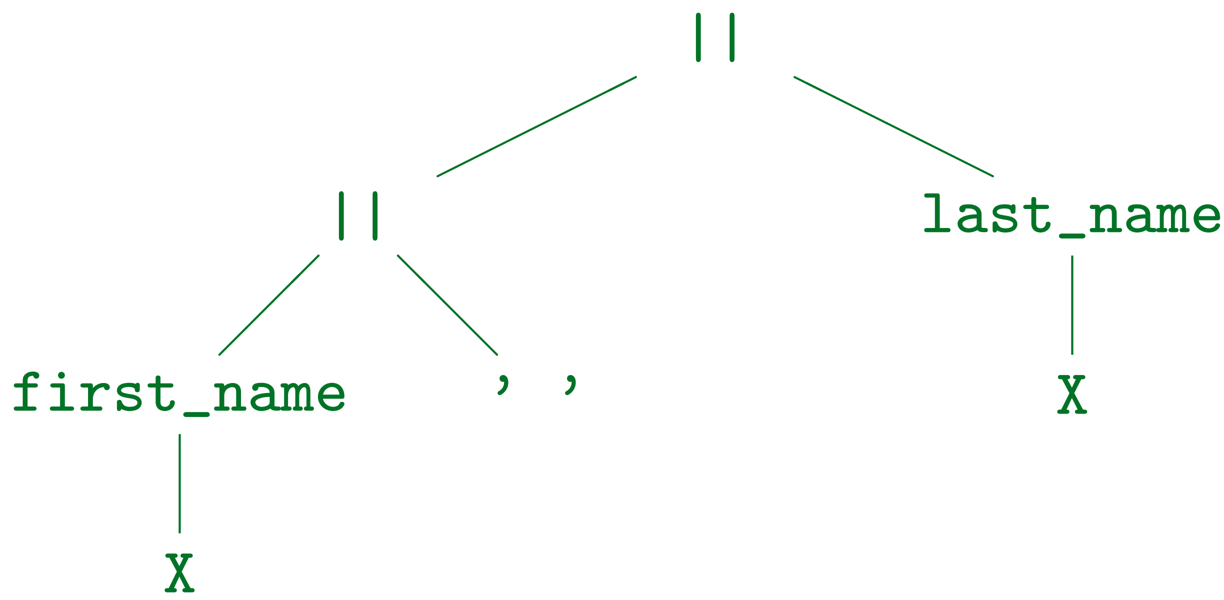
- Functions of arity 1 can be written in dot-notation, e.g. "`X.first_name`" instead of "`first_name(X)`".
- Such "syntactic sugar" is useful in practice, but not important for the theory of logic.

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

- In the following, the above abbreviations are used.

Terms (5)

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



- Exercise: How could this term be written with "||" as infix operator and using the dot-notation?

Terms (6)

Exercise:

- Which of the following are legal terms (given the signature on slide 2-23 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 $\perp$ (false).

Atomic Formulas (2)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ and a variable declaration ν for Σ be given.
- 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 $>(X, 1)$.

  Of course, we will use this common notation in the examples.

- 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.

  However, the above definition ensures that at least the equality is available for all sorts, and below we will make sure it always has the standard interpretation.

Formulas (1)

Definition:

- Let a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ 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 $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
  - ◇ $(\forall\, s\, X\colon F)$ 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 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".
  - $\forall\, s\, X : F$: "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.

  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 (4)

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 (5)

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 $\forall s\, X\colon F$ or $\exists\, s\, X\colon F$ with a level 5 formula F.
- A formula is a level 5 formula.

Formulas (6)

Abbreviations for Quantifiers:

- When there is only one possible sort of a quantified variable, one can leave it out, i.e. write $\forall X\colon F$ instead of $\forall\, s\, X\colon F$ (and the same for $\exists$).

  In most cases, there is indeed only one possible sort for a variable.

- If one quantifier immediately follows another quantifier, one can leave out the colon.

  E.g. write $\forall X\, \exists Y\colon F$ instead of $\forall X\colon \exists Y\colon F$.

- Instead of a sequence of quantifiers of the same type, e.g. $\forall X_1 \ldots \forall X_n\colon F$, one can write $\forall X_1, \ldots, X_n\colon F$.

Formulas (7)

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 $\leq \in \mathcal{P}_{\text{int int}}$ and $1, 10 \in \mathcal{F}_{\epsilon,\text{int}}$, and a variable declaration with $\nu(X) = \mathtt{int}$.
- Is $1 \leq X \leq 10$ a syntactically correct formula?

Formulas (8)

Exercise:

- Which of the following are syntactically correct formulas (given the signature on Slide 2-23)?
  - ☐ $\forall\, \mathtt{X}, \mathtt{Y}\colon \mathtt{married\_to}(\mathtt{X}, \mathtt{Y}) \longrightarrow \mathtt{married\_to}(\mathtt{Y}, \mathtt{X})$
  - ☐ $\forall\, \mathtt{person}\ \mathtt{P}\colon\ \vee\ \mathtt{male}(\mathtt{P}) \vee \mathtt{female}(\mathtt{P})$
  - ☐ $\forall\, \mathtt{person}\ \mathtt{P}\colon\ \mathtt{arno} \vee \mathtt{birgit} \vee \mathtt{chris}$
  - ☐ $\mathtt{male}(\mathtt{chris})$
  - ☐ $\forall\, \mathtt{string}\ \mathtt{X}\colon\ \exists\, \mathtt{person}\ \mathtt{X}\colon\ \mathtt{married\_to}(\mathtt{birgit}, \mathtt{X})$
  - ☐ $\mathtt{married\_to}(\mathtt{birgit}, \mathtt{chris})$
    $\wedge\ \vee\ \mathtt{married\_to}(\mathtt{chris}, \mathtt{birgit})$

Closed Formulas

Definition:

- Let a signature Σ be given.
- A closed formula (for Σ) is a (Σ, ν)-formula for the empty variable declaration ν.

  I.e. the variable declaration that is everywhere undefined.

Exercise:

- Which of the following are closed formulas?
  - ☐ $\mathtt{female(X)} \land \exists\, \mathtt{X}\colon \mathtt{married\_to(chris, X)}$
  - ☐ $\mathtt{female(birgit)} \land \mathtt{married\_to(chris, birgit)}$
  - ☐ $\exists\, \mathtt{X}\colon \mathtt{married\_to(X, Y)}$

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 $(\exists\, s\, X\colon G)$: $free(F) := free(G) - \{X\}$.

Variable Assignment (1)

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.

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:

ν	
Variable	Sort
`X`	`string`
`Y`	`person`

- One possible variable assignment is

$\mathcal{A}$	
Variable	Value
`X`	abc
`Y`	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:

$\mathcal{A}'$	
Variable	Value
X	abc
Y	Birgit

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 $(\mathcal{I}, \nu)$ 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 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 $f(t_1, \ldots, t_n)$, with t_i of sort s_i: $\langle \mathcal{I}, \mathcal{A} \rangle [t] := \mathcal{I}[f, s_1 \ldots s_n](\langle \mathcal{I}, \mathcal{A} \rangle [t_1], \ldots, \langle \mathcal{I}, \mathcal{A} \rangle [t_n])$.

Truth of a Formula (1)

Definition:

- The truth value $\langle \mathcal{I}, \mathcal{A}\rangle[F] \in \{0, 1\}$ of a formula F in $(\mathcal{I}, \mathcal{A})$ is defined as follows (0 means false, 1 true):
  - ◇ If F is an atomic formula $p(t_1, \ldots, t_n)$ with terms t_i of sort s_i:

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} 1 & \text{if } (\langle \mathcal{I}, \mathcal{A}\rangle[t_1], \ldots, \langle \mathcal{I}, \mathcal{A}\rangle[t_n]) \\ & \qquad \in \mathcal{I}[p, s_1 \ldots s_n] \\ 0 & \text{else.} \end{cases}$$

  - ◇ (continued on next three slides . . .)

Truth of a Formula (2)

Definition, continued:

- Truth value of a formula, continued:
  - ◇ If F is an atomic formula $t_1 = t_2$:
  $$\langle \mathcal{I}, \mathcal{A} \rangle [F] := \begin{cases} 1 & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle [t_1] = \langle \mathcal{I}, \mathcal{A} \rangle [t_2] \\ 0 & \text{else.} \end{cases}$$
  - ◇ If F is $\top$: $\langle \mathcal{I}, \mathcal{A} \rangle [F] := 1.$
  - ◇ If F is $\bot$: $\langle \mathcal{I}, \mathcal{A} \rangle [F] := 0.$
  - ◇ If F is of the from $(\neg G)$:
  $$\langle \mathcal{I}, \mathcal{A} \rangle [F] := \begin{cases} 1 & \text{if } \langle \mathcal{I}, \mathcal{A} \rangle [G] = 0 \\ 0 & \text{else.} \end{cases}$$

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	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
0	0	0	0	1	1	1
0	1	0	1	0	1	0
1	0	0	1	1	0	0
1	1	1	1	1	1	1

E.g. if $\langle\mathcal{I},\mathcal{A}\rangle[G_1] = 1$ and $\langle\mathcal{I},\mathcal{A}\rangle[G_2] = 0$ then $\langle\mathcal{I},\mathcal{A}\rangle[(G_1 \wedge G_2)] = 0$.

Truth of a Formula (4)

Definition, continued:

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

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} 1 & \text{if } \langle \mathcal{I}, \mathcal{A}\langle X/d\rangle\rangle[G] = 1 \\ & \quad \text{for all } d \in \mathcal{I}[s] \\ 0 & \text{else.} \end{cases}$$

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

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} 1 & \text{if } \langle \mathcal{I}, \mathcal{A}\langle X/d\rangle\rangle[G] = 1 \\ & \quad \text{for at least one } d \in \mathcal{I}[s] \\ 0 & \text{else.} \end{cases}$$

Model (1)

Definition:

- If $\langle\mathcal{I},\mathcal{A}\rangle[F] = 1$, 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 $\langle\mathcal{I},\mathcal{A}\rangle[F] = 1$ for all variable declarations $\mathcal{A}$.

  I.e. free variables are treated as $\forall$-quantified. Of course, if F is a closed formula, the variable declaration is not important.
- If $\mathcal{I} \models F$, one says that $\mathcal{I}$ satisfies F.

  Or that F is true in $\mathcal{I}$. The same for $\langle\mathcal{I},\mathcal{A}\rangle \models F$.
- 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 it has a model.
- A formula F is called satisfiable iff there is an interpretation $\mathcal{I}$ and a variable declaration $\mathcal{A}$ such that $(\mathcal{I}, \mathcal{A}) \models F$. Otherwise it is called unsatisfiable.

  Sometimes I will say inconsistent when I really mean unsatisfiable.
- A (Σ, ν)-formula F is called a tautology iff for all Σ-interpretations $\mathcal{I}$ and (Σ, ν)-variable assignments $\mathcal{A}$: $(\mathcal{I}, \mathcal{A}) \models F$.

Model (3)

Exercise:

- Consider the interpretation on Slide 2-29:
  - $\mathcal{I}[\mathtt{person}] = \{\text{Arno}, \text{Birgit}, \text{Chris}\}$.
  - $\mathcal{I}[\mathtt{married\_to}] = \{(\text{Birgit}, \text{Chris}),\ (\text{Chris}, \text{Birgit})\}$.
  - $\mathcal{I}[\mathtt{male}] = \{(\text{Arno}),\ (\text{Chris})\}$,
    $\mathcal{I}[\mathtt{female}] = \{(\text{Birgit})\}$.
- Which of the following formulas are true in $\mathcal{I}$?
  - ☐ $\forall\, \mathtt{person}\ \mathtt{X}\colon \mathtt{male}(\mathtt{X}) \leftrightarrow \neg\mathtt{female}(\mathtt{X})$
  - ☐ $\forall\, \mathtt{person}\ \mathtt{X}\colon \mathtt{male}(\mathtt{X}) \vee \neg\mathtt{male}(\mathtt{X})$
  - ☐ $\exists\, \mathtt{person}\ \mathtt{X}\colon \mathtt{female}(\mathtt{X}) \wedge \neg\exists\, \mathtt{person}\ \mathtt{Y}\colon \mathtt{married\_to}(\mathtt{X},\mathtt{Y})$
  - ☐ $\exists\, \mathtt{person}\ \mathtt{X},\ \mathtt{person}\ \mathtt{Y},\ \mathtt{person}\ \mathtt{Z}\colon\ \mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z} \wedge \mathtt{X} \neq \mathtt{Z}$

Overview

1. Introduction, Motivation, History
2. Signatures, Interpretations
3. Formulas, Models
4. Formulas in Databases
5. Implication, Equivalence

Formulas in Databases (1)

- As explained above, 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.

  Every data model imposes certain restrictions for the kinds of new symbols that can be introduced. For instance, in the classical relational model, the database schema can only define new predicate symbols (relation symbols).

Formulas in Databases (2)

- A database state is then an interpretation $\mathcal{I}$ for the extended signature Σ.

  Of course, the system would explicitly store only the interpretation of the symbols of "$\Sigma - \Sigma_{\mathcal{D}}$", because the interpretation of the symbols in $\Sigma_{\mathcal{D}}$ is already built into the DBMS and cannot be changed. Furthermore, not arbitrary interpretations can be used as database states. The exact restrictions depend on the data model, but typically the new symbols must have a finite interpretation.

- Formulas are used in databases as:
  - ◇ Integrity constraints
  - ◇ Queries
  - ◇ Definitions of derived symbols (views).

Integrity Constraints (1)

- Not all interpretations are reasonable DB states.

  The purpose of the database is to model a certain part of the real world. In the real world, certain restrictions exist. Therefore, interpretations that violate these restrictions should be excluded.

- For instance, in the real world, a person can only be male or female, but not both. Therefore, the following two formulas must be satisfied:
  - $\forall\, \texttt{person}\ \texttt{X}\colon\ \texttt{male(X)} \lor \texttt{female(X)}$
  - $\forall\, \texttt{person}\ \texttt{X}\colon\ \neg\,\texttt{male(X)} \lor \neg\,\texttt{female(X)}$
- These are examples of integrity constraints.

Integrity Constraints (2)

- An integrity constraint is a closed formula.

  The data model might not permit arbitrary formulas. Furthermore, the formulas must be domain independent (see below).

- 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.

  In the following, we consider only valid database states. Therefore, we again speak only of "database state".

Integrity Constraints (3)

Keys I:

- Objects are often identified by unique data values (numbers, names).
- For example, consider the signature on Slide 2-37. There should never be two different objects of type `student` with the same `sid`:

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

- Alternative, equivalent formulation:

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

Integrity Constraints (4)

Keys II:

- In the relational schema (Slide 2-33) 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):

$$\forall\, \mathtt{int}\ \mathrm{ID},\ \mathtt{string}\ \mathrm{F1},\ \mathtt{string}\ \mathrm{F2},\ \mathtt{string}\ \mathrm{L1},\ \mathtt{string}\ \mathrm{L2}\colon$$
$$\mathtt{student}(\mathrm{ID}, \mathrm{F1}, \mathrm{L1}) \land \mathtt{student}(\mathrm{ID}, \mathrm{F2}, \mathrm{L2}) \ \rightarrow$$
$$\mathrm{F1} = \mathrm{F2} \ \land\ \mathrm{L1} = \mathrm{L2}$$

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

Integrity Constraints (5)

Exercise:

- Consider the schema on Slide 2-33:
  - `student(int SID, string FName, string LName)`
  - `exercise(int ENO, int MaxPoints)`
  - `result(int SID, int ENO, int Points)`
- Please write the following constraints:
  - The `Points` in `result` are always non-negative.
  - Each exercise number `ENO` in `result` appears also in `exercise` ("no broken links").

    This is an example of a "foreign key constraint".

Queries (1)

- A query (Form A) 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 $\{X_1/s_1, \ldots, X_n/s_n\}$.

  Again, there might be restrictions for the possible formulas F, especially the domain independence (see below).

- 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 $\mathcal{I}$.

  In order to ensure that the variable assignment is printable, the sorts s_i of the result variables typically must be data types.

Queries (2)

Examples I:

- Consider the schema on Slide 2-33:
  - ◇ `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?

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

Queries (3)

Examples II:

- Print all results for Ann Smith:

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

- Who has not yet submitted Exercise 2?

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

Exercise:

- Print students who have 10 points in Exercise 1 and 10 points in Exercise 2.

Queries (4)

- A query (Form B) 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 $\{X_1/s_1, \ldots, X_n/s_n\}$.

- In this case, 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 $\langle \mathcal{I}, \mathcal{A}\rangle \models F$.

  This is especially convenient when the variables X_i range over sorts that are otherwise not printable.

Queries (5)

Example:

- Consider the schema on Slide 2-37:
  - ◇ Sorts `student`, `exercise`.
  - ◇ Functions `first_name(student): string`, ...
  - ◇ Predicate: `solved(student, exercise)`.

    Function: `points(student, exercise): int`
- Who got at least 8 points for Homework 1?

$$\{\texttt{S.first\_name},\ \texttt{S.last\_name}\ [\texttt{student S}] \mid \exists\,\texttt{exercise E}\colon\ \texttt{E.eno} = 1 \wedge \texttt{solved(S, E)} \wedge \texttt{points(S, E)} \geq 8\}$$

Queries (6)

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

Exercise:

- Suppose Form C is not available. Is it possible to simulate it with Form A or Form B?
- Obviously, Form A is a special case of Form B: $\{X_1, \ldots, X_n\ [s_1\ X_1, \ldots, s_n\ X_n] \mid F\}$. Is it conversely possible to simulate Form B with Form A?

Domain Independence (1)

- One cannot use arbitrary formulas as queries. Some formulas would generate an infinite answer:

$$\{\texttt{int SID} \mid \neg\texttt{student(SID, 'Ann', 'Smith')}\}$$

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

$$\exists\, \texttt{int X, int Y, int Z:}\ \ \texttt{X} * \texttt{X} + \texttt{Y} * \texttt{Y} = \texttt{Z} * \texttt{Z}$$

- When a formula is domain-independent, it suffices to consider only finitely many values for each variable. Then the above problems do not occur.

Domain Independence (2)

- A formula is domain independent iff for all possible DB states (interpretations), it suffices to replace variables that range over possibly infinite domains by values that appear in any argument of the DB relations, or as function value of a DB function, or as variable-free 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 $\mathcal{I}$, as value in the database sorts, as value of database functions, or as variable-free 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.

Domain Independence (3)

- Since all database sorts, database relations, and database functions are finite, queries can be evaluated in finite time.
- For instance, the formula

$$\exists\, \mathtt{int}\, X\colon\ X \neq 1$$

is not domain independent: The truth value depends on other integers besides 1, but at least in the empty database state there are no such values.

The exact set of possible values (domain) is sometimes not known, only the set of values that appear in the database is known. Then it is good if the truth value of a formula does not depend on the domain.

Domain Independence (4)

- "Range restriction" is a syntactic constraint on formulas that implies 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 $p(t_1, \ldots, t_n)$ with database relation p, then $posres(F) := \{X \in VARS \mid t_i$ is the variable $X\}$, $negres(F) := \emptyset$. For other atomic formulas, both sets are empty, except when F has the form $X = t$ where t is variable-free or has a database function as outermost function. Then $posres(F) := \{X\}$. 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 $\forall X\colon G$, it holds that $X \in negres(G)$, and for every subformula $\exists X\colon G$, it holds that $X \in posres(G)$.

Domain Independence (5)

- Basically, range striction requires that every variable is bound to a finite set of values.

  E.g. by occurring in a database relation in positive context (unless it is universally quantified, then it must appear in negative context).

- Range restriction is decidable, whereas domain independence is in general undecidable.

  E.g. $\exists X: X \neq 1 \wedge F$ would be domain independent if F is always false. The consistency of formulas (even range-restricted formulas) is undecidable.

  Nevertheless, range-restriction is sufficiently general: For instance, all relational algebra queries can be naturally translated into range-restricted formulas.

Overview

1. Introduction, Motivation, History
2. Signatures, Interpretations
3. Formulas, Models
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 $\Phi \vdash G$.
- 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:

- $\Phi \vdash G$ if and only if $\Phi \cup \{\neg\forall(G)\}$ 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 $F_1 \vdash F_2$ and $F_2 \vdash F_1$.

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 $(\mathcal{I}, \nu)$-variable declaration $\mathcal{A}$:

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

- 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:
  - $\forall\, s\, X\colon F \equiv \neg(\exists\, s\, X\colon (\neg F))$
  - $\exists\, s\, X\colon F \equiv \neg(\forall\, s\, X\colon (\neg F))$
- 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 (5)

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

  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 $X \notin free(F)$ and $\mathcal{I}[s]$ cannot be empty:
  - $\forall s\, X\colon F \;\equiv\; F$
  - $\exists s\, X\colon F \;\equiv\; F$

Some Equivalences (6)

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

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

- 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:
  - $\forall s\, X\colon F \equiv \forall s\, Y\colon F'$
  - $\exists s\, X\colon F \equiv \exists s\, Y\colon F'$

Normal Forms (1)

Definition:

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

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

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)

Remark:

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

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

Theorem:

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