Teil 3: Mathematische Logik mit Datenbank-Anwendungen

Literatur:

- Bergmann/Noll: Mathematische Logik mit Informatik-Anwendungen. Springer, 1977.
- Ebbinghaus/Flum/Thomas: Einführung in die mathematische Logik. Spektrum Akademischer Verlag, 1996.
- Tuschik/Wolter: Mathematische Logik, kurzgefasst. Spektrum Akademischer Verlag, 2002.
- Schöning: Logik für Informatiker. BI Verlag, 1992.
- Chang/Lee: Symbolic Logic and Mechanical Theorem Proving. Academic Press, 1973.
- Fitting: First Order Logic and Automated Theorem Proving. Springer, 1995, 2. Auflage.
- Ulf Nilson, Jan Małuszyński: Logic, Programming, and Prolog (2. Auflage). 1995, `http://www.ida.liu.se/{\char126}ulfni/lpp`

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- grundlegende Begriffe erklären: Signatur, Interpretation, Variablenbelegung, Term, Formel, Modell, Konsistenz, Implikation.
- Integritätsbedingungen und Anfragen als Formeln aufschreiben.
- gebräuchliche Äquivalenzen anwenden, um logische Formeln zu transformieren.
- prüfen, ob eine Formel in einer Interpretation erfüllt ist, ggf. passende Variablenbelegungen finden.

Inhalt

Einführung, Motivation (1)

Wichtige Ziele mathematischer Logik sind:

- den Begriff einer Aussage über gewisse "Miniwelten" zu formalisieren (logische Formel),
- die Begriffe der logischen Implikation und des Beweises präzise zu definieren,
- Algorithmen zu finden, um zu testen, ob eine Aussage von anderen logisch impliziert wird.

  So weit das möglich ist. Es hat sich herausgestellt, dass diese Aufgabe im allgemeinen unentscheidbar ist. Auch der Begriff der Entscheidbarkeit, heute ein Kernbegriff der Informatik, wurde von Logikern entwickelt (damals gab es noch keine Computer und keine Informatik).

Einführung, Motivation (2)

Anwendung mathematischer Logik in Datenbanken I:

- Allgemein ist das Ziel von mathematischer Logik und von Datenbanken
  - Wissen zu formalisieren,
  - mit diesem Wissen zu arbeiten.
- Zum Beispiel benötigt man Symbole, um über eine Miniwelt sprechen zu können.
  - In der Logik sind diese in Signaturen definiert.
  - In Datenbanken sind sie im DB-Schema definiert.

    Es ist daher keine Überraschung, dass ein Datenbankschema in der Hauptsache eine Signatur definiert.

Einführung, Motivation (3)

Anwendung mathematischer Logik in Datenbanken II:

- Um logische Implikationen zu formalisieren, muss die mathematische Logik mögliche Interpretationen der Symbole untersuchen,
- d.h. mögliche Situationen der Miniwelt, über die Aussagen in logischen Formeln gemacht werden.
- Der DB-Zustand beschreibt auch mögliche Situationen eines gewissen Teils der realen Welt.
- Im Grunde sind logische Interpretation und DB-Zustand dasselbe (aus "modelltheoretischer Sicht").

Einführung, Motivation (4)

Anwendung mathematischer Logik in Datenbanken III:

- SQL-Anfragen sind Formeln der mathematischen Logik sehr ähnlich. Es gibt theoretische Anfragesprachen, die nur eine Variante der Logik sind.
- Die Idee ist, dass
  - eine Anfrage eine logische Formel mit Platzhaltern ("freien Variablen") ist,
  - und das Datenbanksystem dann Werte für diese Platzhalter findet, so dass die Formel im gegebenen DB-Zustand erfüllt wird.

Einführung, Motivation (5)

Warum sollte man mathematische Logik lernen I:

- Logische Formeln sind einfacher als SQL, und können leicht formal untersucht werden.
- Wichtige Konzepte von DB-Anfragen können schon in dieser einfachen Umgebung erlernt werden.
- Die Erfahrung hat gezeigt, dass Studierende oft logische Fehler in SQL-Anfragen machen.

    Z.B. inkonsistente Anfragen (Anfragen, die unabhängig vom DB-Zustand nie eine Antwort liefern können).

Einführung, Motivation (6)

Warum sollte man mathematische Logik lernen II:

- SQL verändert sich und wird zunehmend komplexer (Standards: 1986, 1989, 1992, 1999, 2003, 2008).

  Irgendwann wird vermutlich jemand eine wesentlich einfachere Sprache vorschlagen, die das gleiche kann. Datalog war schon ein solcher Vorschlag, hat sich aber bisher noch nicht durchsetzen können. Aber man denke an Algol 68 und PL/I, gefolgt von Pascal and C.

- Es werden neue Datenmodelle vorgeschlagen (z.B. XML), mit noch schnelleren Änderungen als SQL.
- Zumindest ein Teil dieser Vorlesung sollte auch in 30 Jahren noch gültig und nützlich sein.

Geschichte des Gebietes (1)

~322 v.Chr.	Syllogismen [Aristoteles]
~300 v.Chr.	Axiome der Geometrie [Euklid]
~1700	Plan der mathematischen Logik [Leibniz]
1847	"Algebra der Logik" [Boole]
1879	"Begriffsschrift" (frühe logische Formeln) [Frege]
~1900	natürlichere Formelsyntax [Peano]
1910/13	Principia Mathematica (Sammlung formaler Beweise) [Whitehead/Russel]
1930	Vollständigkeitssatz [Gödel/Herbrand]
1936	Unentscheidbarkeit [Church/Turing]

Geschichte des Gebietes (2)

1960	Erster Theorembeweiser [Gilmore/Davis/Putnam]
1963	Resolutionsmethode (Resolventenmethode) zum automatischen Beweisen [Robinson]
~1969	Frage-Antwort-Systeme [Green et.al.]
1970	Lineare Resolution [Loveland/Luckham]
1970	Relationales Datenmodell [Codd]
~1973	Prolog [Colmerauer, Roussel, et.al.] (begann als Theorembeweiser für das Verstehen natürl. Sprache) (Vgl.: Fortran 1954, Lisp 1962, Pascal 1970, Ada 1979)
1977	Konferenz "Logik und Datenbanken" [Gallaire, Minker]

Inhalt

Alphabet (1)

Definition:

- Sei $ALPH$ eine unendliche, aber abzählbare Menge von Elementen, die Symbole genannt werden.

  Formeln sind Wörter über $ALPH$, d.h. Folgen von Symbolen.

- $ALPH$ muss zumindest die logischen Symbole enthalten, d.h. $LOG \subseteq ALPH$, wobei

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

- Zusätzlich muss $ALPH$ eine unendliche Teilmenge $VARS \subseteq ALPH$ enthalten, die Menge der Variablen. Diese ist disjunkt zu LOG (d.h. $VARS \cap LOG = \emptyset$).

  Einige Autoren betrachten Variablen als logische Symbole.

Alphabet (2)

- Z.B. kann das Alphabet bestehen aus
  - ◇ den speziellen logischen Symbolen LOG,
  - ◇ Bezeichnern: Folgen von Buchstaben, Ziffern, "_",

    In der Praxis sind Variablen oft Bezeichner. In Logik-Lehrbüchern wird üblicherweise angenommen, dass die Variablen disjunkt zu den Bezeichnern sind, die für Prädikate und Funktionen verwendet werden. Das vereinfacht die Definitionen, aber die Beispiele sehen dann komisch aus, wenn z.B. alle Nicht-Variablen klein geschrieben werden müssen, oder man nur x, y, z (mit Index) als Variablen verwenden kann.
  - ◇ Operator-Symbolen wie `+`, `<`,
  - ◇ Datentyp-Literalen (Konstanten) wie `123`, `'abc'`.

Alphabet (3)

- Man beachte, dass Wörter wie "`Student`" als Symbole angesehen werden (Elemente des Alphabets).

  Wenn man ein Alphabet aus 26 Buchstaben gewöhnt ist, erscheint es zunächst komisch, dass das Alphabet hier unendlich ist. Es ist aber bei Programmiersprachen auch üblich, die einzelnen Zeichen zunächst zu Wortsymbolen (Token) zusammenzufassen (durch die lexikalische Analyse). Die eigentliche Grammatik der Programmiersprache beschreibt dann, wie Programme als Folgen solcher Wortsymbole zu bilden sind (gewissermaßen "Sätze"). Nichts anderes geschieht hier.

- In der Theorie sind die exakten Symbole unwichtig.

  Auch die logischen Symbole müssen nicht unbedingt so aussehen wie auf Folie 3-13 (Beispiele für Alternativen siehe Folie 3-17). Es ist nur wichtig, dass es für jedes logische Zeichen eine Entsprechung gibt. Es muß auch immer eine noch unbenutze Variable geben.

Alphabet (4)

- Für eine Menge A ist A^* die Menge der endlichen Folgen $a_1 \ldots a_n$ von Elemententen $a_i \in A$.

  Man nennt A^* auch die Menge der Worte über dem Alphabet A.

- ϵ ist die leere Folge (Folge der Länge 0).

  Man nennt ϵ auch das leere Wort.

- Formeln sind Elemente von $ALPH^*$, z.B. ist

  $$\forall\, \texttt{Person X: schlaeft(X)} \rightarrow \neg\, \texttt{arbeitet(X)}$$

  eine Formel bestehend aus den Symbolen $\forall$, `Person`, `X`, u.s.w.

Alphabet (5)

Mögliche Alternativen für logische Symbole:

Symbol	Alternative	Alt2	Name
$\top$	`true`	`T`	
$\bot$	`false`	`F`	
$\neg$	`not`	`~`	Negation
$\wedge$	`and`	`&`	Konjunktion
$\vee$	`or`	`|`	Disjunktion
$\leftarrow$	`if`	`<-`	
$\rightarrow$	`then`	`->`	
$\leftrightarrow$	`iff`	`<->`	
$\exists$	`exists`	`E`	Existenzquantor
$\forall$	`forall`	`A`	Allquantor

Signaturen (1)

Definition:

- Eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ enthält:
  - ◇ Eine nichtleere, endliche Menge $\mathcal{S}$. Die Elemente heißen Sorten (Datentypnamen).
  - ◇ Für jedes $\alpha = s_1 \dots s_n \in \mathcal{S}^*$, eine endliche Menge (Prädikatssymbole) $\mathcal{P}_\alpha \subseteq ALPH - LOG$.
  - ◇ Für jedes $\alpha \in \mathcal{S}^*$ und $s \in \mathcal{S}$, eine Menge (von Funktionssymbolen) $\mathcal{F}_{\alpha,s} \subseteq ALPH - LOG$.

    Für jedes $\alpha \in \mathcal{S}^*$ und $s_1, s_2 \in \mathcal{S}$, $s_1 \neq s_2$ muss gelten, dass $\mathcal{F}_{\alpha,s_1} \cap \mathcal{F}_{\alpha,s_2} = \emptyset$ (Beschränkung des Überladens von Funktionssymbolen: Name und Argumentsorten bestimmen Ergebnissorte). Außerdem muss $VARS - \cup_{s \in \mathcal{S}} \mathcal{F}_{\epsilon,s}$ noch unendlich sein (ausreichend Variablen nach Auflösung von Namenskonflikten mit Konstanten).

Signaturen (2)

- Sorten sind Datentypnamen, z.B. `string`, `Person`.
- Prädikate liefern für gegebene Eingabewerte wahr oder falsch, z.B. `<`, `substring`, `ungerade`, `verheiratet`.
- Ist $p \in \mathcal{P}_\alpha$, und $\alpha = s_1 \dots s_n$, dann werden $s_1, \dots, s_n$ die Argumentsorten von p genannt.

  s_1 ist der Typ des ersten Arguments, s_2 der des zweiten, usw.
- Beispiele:
  - ◇ `ungerade` $\in \mathcal{P}_{\texttt{int}}$
  - ◇ `verheiratet` $\in \mathcal{P}_{\texttt{Person Person}}$

Signaturen (3)

- Wenn man die Menge $\mathcal{P}$ definiert, wird man i.a. wohl nicht die Index-Notation verwenden, sondern die Prädikate z.B. in folgender Syntax deklarieren:

```
ungerade(int).
verheiratet(Person, Person).
```

- Man kann auch Argumentnamen einführen und die beabsichtigte Bedeutung des Prädikates in einem Kommentar erklären:

```
verheiratet(Person X, Person Y).
    /* X ist mit Y verheiratet. */
```

Fomal sind die Argumentnamen auch nur Kommentare.

Signaturen (4)

- Die Anzahl der Argumentsorten (Länge von α) nennt man Stelligkeit des Prädikatsymbols, z.B.:
  - `ungerade` ist ein Prädikatsymbol der Stelligkeit 1.
  - `verheiratet` hat die Stelligkeit 2.
- Prädikate der Stelligkeit 0 nennt man aussagenlogische Konstanten/Symbole. Z.B.:
  - `die_sonne_scheint`,
  - `ich_arbeite_gerade`.
- Die Menge $\mathcal{P}_\epsilon$ enthält alle aussagenlogischen Konstanten (ϵ ist das leere Wort).

Signaturen (5)

- Das gleiche Symbol p kann Element verschiedener $\mathcal{P}_\alpha$ sein (überladenes Prädikat), z.B.
  - `<` $\in \mathcal{P}_{\texttt{int int}}$
  - `<` $\in \mathcal{P}_{\texttt{string string}}$ (lexikographische/alphabetische Ordnung)
- Es kann also mehrere verschiedene Prädikate mit gleichem Namen geben.

  Die Möglichkeit überladener Prädikate ist nicht wichtig. Man kann stattdessen auch verschiedene Namen verwenden, etwa `lt_int` und `lt_string`. Überladene Prädikate komplizieren die Definition und sind in der mathematischen Logik normalerweise ausgeschlossen. Sie erlauben aber natürlichere Formulierungen.

Signaturen (6)

- Eine Funktion liefert für gegebene Eingabewerte einen Ausgabewert. Beispiele: `+`, `Alter`, `Vorname`.

  Es sei hier noch angenommen, dass Funktionen für alle Eingabewerte definiert sind. Das muss nicht sein, z.B. könnte `telefax_nr(peter)` nicht existieren und `5/0` ist nicht definiert. Im letzten Abschnitt dieses Kapitels wird eine dreiwertige Logik zur Behandlung von Nullwerten (undefinierten Werten) eingeführt: Aussagen können dann wahr, falsch oder undefiniert sein.
- Ein Funktionssymbol in $\mathcal{F}_{\alpha,s}$ hat die Argumentsorten α und die Ergebnissorte s, z.B.
  - `+` $\in \mathcal{F}_{\texttt{int int, int}}$ `+(int, int): int`.
  - `Alter` $\in \mathcal{F}_{\texttt{Person, int}}$ `Alter(Person): int`.

Signaturen (7)

- Eine Funktion ohne Argumente heißt Konstante.
- Beispiele für Konstanten:
  - `1` $\in \mathcal{F}_{\epsilon,\texttt{int}}$ `1: int`.
  - `'Birgit'` $\in \mathcal{F}_{\epsilon,\texttt{string}}$ `'Birgit': string`.
- Bei Datentypen (z.B. `int`, `string`) kann üblicherweise jeder mögliche Wert durch eine Konstante bezeichnet werden.

  Im Allgemeinen sind die Menge der Werte und die der Konstanten dagegen verschieden. Zum Beispiel wäre es möglich, dass es keine Konstanten vom Typ `Person` gibt, wohl aber Objekte dieses Typs.

Signaturen (8)

- Natürlich kann man eine unendliche Konstantenmenge nicht durch Aufzählung definieren.
- Mathematisch ist das kein Problem, $\mathcal{F}_{\epsilon,\mathtt{int}}$ ist eben die Menge der ganzen Zahlen in Dezimalnotation.
- Praktisch ist das auch kein Problem, da die Datentypen bereits in das DBMS eingebaut sind (s.u.): $\mathcal{F}_{\epsilon,\mathtt{int}}$ ist durch ein Programm definiert.
- Es kann aber Aufzählungstypen geben:

```
Januar:  Monat. /* Konstante der Sorte Monat */
Februar: Monat. /* u.s.w. */
```


Signaturen (9)

- Zusammengefasst legt eine Signatur anwendungsspezifische Symbole fest, die verwendet werden, um über die entsprechende Miniwelt zu sprechen.
- Hier: mehrsortige (typisierte) Logik.
  Alternative: unsortierte/einsortige Logik.

  Dann wird $\mathcal{S}$ nicht benötigt, und $\mathcal{P}$ und $\mathcal{F}$ haben als Index nur die Stelligkeit. Z.B. Prolog verwendet eine unsortierte Logik. Dies ist auch in Lehrbüchern über mathematische Logik gebräuchlich (die Definitionen sind dann etwas einfacher). Da man Sorten durch Prädikate der Stelligkeit 1 simulieren kann, ist eine einsortige Logik keine echte Einschränkung. Allerdings sind Formeln mit Typfehlern in einer mehrsortierten Logik syntaktisch falsch, während die entsprechende Formel in einer einsortigen Logik legal, aber logisch falsch (inkonsistent) ist.

Signaturen (10)

Beispiel:

- $\mathcal{S} = \{\mathtt{Person}, \mathtt{string}\}$.
- $\mathcal{F}$ besteht aus
  - ◇ Konstanten `arno`, `birgit`, `chris` der Sorte `Person`.
  - ◇ unendlich vielen Konstanten der Sorte `string`, z.B. `''`, `'a'`, `'b'`, . . . , `'Arno'`, . . .
  - ◇ Funktionssymbolen `Vorname(Person): string` und `Nachname(Person): string`.
- $\mathcal{P}$ besteht aus
  - ◇ dem Prädikat `verheiratet(Person, Person)`.
  - ◇ den Prädikaten `Mann(Person)` und `Frau(Person)`.

Signaturen (11)

- Ein System zur logischen Wissensrepräsentation hat normalerweise Datentypen wie `string` vordefiniert.
- Man könnte die obige Signatur dann in folgender (fiktiven) Syntax definieren:

```
SORTS   Person.
CONSTS  arno, birgit, chris: Person.
FUNS    Vorname(Person): string.
        Nachname(Person): string.
PREDS   verheiratet(Person, Person).
        Mann(Person).
        Frau(Person).
```

Signaturen (12)

Übung:

- Definieren Sie eine Signatur über
  - ◇ Bücher (mit Autoren, Titel, ISBN)

    Es reicht aus, die Liste der Autoren eines Buches als einen String zu behandeln. Fortgeschrittene Übung: Behandeln Sie Bücher mit mehreren Autoren, indem Sie Listen von Strings modellieren.
  - ◇ Buchbesprechungen (mit Kritiker, Text, Sternen).

    Jede Besprechung ist für genau ein Buch.
  - ◇ "Sterne" können keiner, einer, zwei oder drei sein.

Signaturen (13)

Definition:

- Eine Signatur $\Sigma' = (\mathcal{S}', \mathcal{P}', \mathcal{F}')$ ist eine Erweiterung der Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$, falls
  - ◇ $\mathcal{S} \subseteq \mathcal{S}'$,
  - ◇ für jedes $\alpha \in \mathcal{S}^*$:
    $\mathcal{P}_\alpha \subseteq \mathcal{P}'_\alpha$,
  - ◇ für jedes $\alpha \in \mathcal{S}^*$ und $s \in \mathcal{S}$:
    $\mathcal{F}_{\alpha,s} \subseteq \mathcal{F}'_{\alpha,s}$.
- D.h. eine Erweiterung von Σ fügt nur neue Symbole zu Σ hinzu.

Interpretationen (1)

Definition:

- Sei die Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ gegeben.
- Eine Σ-Interpretation $\mathcal{I}$ definiert:
  - ◇ eine Menge $\mathcal{I}(s)$ für jedes $s \in \mathcal{S}$ (Wertebereich),
  - ◇ Eine Relation $\mathcal{I}(p, \alpha) \subseteq \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n)$ für jedes $p \in \mathcal{P}_\alpha$ und $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$.

    Im Folgenden schreiben wir $\mathcal{I}(p)$ an Stelle von $\mathcal{I}(p, \alpha)$ wenn α für die gegebene Signatur Σ klar ist (d.h. p nicht überladen ist).
  - ◇ eine Funktion $\mathcal{I}(f, \alpha)$: $\mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n) \to \mathcal{I}(s)$ für jedes $f \in \mathcal{F}_{\alpha,s}$, $s \in \mathcal{S}$ und $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$.
- Im Folgenden schreiben wir $\mathcal{I}[\ldots]$ anstatt $\mathcal{I}(\ldots)$.

Interpretationen (2)

Beachte:

- Leere Wertebereiche verursachen Probleme, deshalb werden sie normalerweise ausgeschlossen.

  Einige Äquivalenzen gelten nicht, wenn die Wertebereiche leer sein können. Zum Beispiel kann die Pränex-Normalform nur unter der Annahme erreicht werden, dass die Wertebereiche nicht leer sind.
- In Datenbanken kann dies aber vorkommen.

  Z.B. Menge von Personen, wenn die Datenbank gerade erst erstellt wurde. Auch bei SQL kann es Überraschungen geben, wenn eine Tupelvariable über einer leeren Relation deklariert ist.
- Statt Wertebereich sagt man auch Individuenbereich, Universum oder Domäne (engl. Domain).

Interpretationen (3)

- Die Relation $\mathcal{I}[p]$ wird auch die Extension von p genannt (in $\mathcal{I}$).
- Formal gesehen sind Prädikat und Relation nicht gleiche, aber isomorphe Begriffe.

  Ein Prädikat ist eine Abbildung auf die Menge $\{true, false\}$ boolescher Werte. Eine Relation ist eine Teilmenge des kartesischen Produkts $\times$.
- Zum Beispiel ist `verheiratet(X,Y)` genau dann wahr in $\mathcal{I}$, wenn $(\mathtt{X},\mathtt{Y}) \in \mathcal{I}[\mathtt{verheiratet}]$.
- Weiteres Beispiel: $(3,5) \in \mathcal{I}[<]$ bedeutet $3 < 5$.

  Im Folgenden werden die Wörter "Prädikatsymbol" und "Relationssymbol" austauschbar verwendet.

Interpretationen (4)

Beispiel (Interpretation für Signatur auf Folie 3-27):

- $\mathcal{I}[\mathtt{Person}]$ ist die Menge mit Arno, Birgit und Chris.
- $\mathcal{I}[\mathtt{string}]$ ist die Menge aller Strings, z.B. `'a'`.
- $\mathcal{I}[\mathtt{arno}]$ ist Arno.
- Für Stringkonstanten c ist $\mathcal{I}[c] = c$.
- $\mathcal{I}[\mathtt{Vorname}]$ bildet z.B. Arno auf `'Arno'` ab.
- $\mathcal{I}[\mathtt{Nachname}]$ liefert für alle drei Personen `'Schmidt'`.
- $\mathcal{I}[\mathtt{verheiratet}] = \{(\text{Birgit}, \text{Chris}), (\text{Chris}, \text{Birgit})\}$.
- $\mathcal{I}[\mathtt{Mann}] = \{(\text{Arno}), (\text{Chris})\}$, $\mathcal{I}[\mathtt{Frau}] = \{(\text{Birgit})\}$.

Interpretationen (5)

Beispiel (Forts.):

- Man kann endliche Wertebereiche, Funktionen, und Relationen (Prädikate) auch als Tabellen darstellen:

Person
Arno
Birgit
Chris

Vorname	
Arno	'Arno'
Birgit	'Birgit'
Chris	'Chris'

Nachname	
Arno	'Schmidt'
Birgit	'Schmidt'
Chris	'Schmidt'

verheiratet	
Birgit	Chris
Chris	Birgit

Mann
Arno
Chris

Frau
Birgit

Datenbanken und Logik (1)

Datenwerte:

- Das DBMS definiert eine Datentyp-Signatur $\Sigma_{\mathcal{D}}$ und eine zugehörige Interpretation $\mathcal{I}_{\mathcal{D}}$, wodurch folgendes festgelegt wird (Forts. auf nächster Folie):
  - ◇ Namen für die Datentypen (Sorten) und jeweils einen zugehörigen (nichtleeren) Wertebereich.
  - ◇ Für jede Sorte Namen für Datentyp-Konstanten (wie `123`, `'abc'`), interpretiert durch Elemente des entsprechenden Wertebereichs.

    Oben wurden Konstanten als nullstellige Funktionen formalisiert, das ist nur ein technischer Trick, um in der Signatur eine Komponente einzusparen.

Datenbanken und Logik (2)

- Festlegung durch $\Sigma_{\mathcal{D}}$ und $\mathcal{I}_{\mathcal{D}}$, Forts.:
  - ◇ Namen für Datentyp-Funktionen (wie `+`, `strlen`) mit Argument- und Ergebnissorten, interpretiert durch entsprechende Funktionen auf den Wertebereichen.
  - ◇ Namen für Datentyp-Prädikate (wie `<`, `odd`), interpretiert durch entsprechende Relationen auf den Wertebereichen.

    Anstelle einer Relation kann man auch eine Funktion verwenden, die einen Wahrheitswert liefert ($\{\mathbf{w}, \mathbf{f}\}$).

Datenbanken und Logik (3)

- Ein Datenmodell (wie z.B. das relationale Modell) wird normalerweise (so wie in Kapitel 2) ohne direkte Zuhilfenahme der Logik definiert:
  - ◇ Natürlich nimmt man Datentypen (Signatur und Interpretation) als gegeben an,
  - ◇ aber dann werden DB-Schemata und Zustände durch Anwendung üblicher mathematischer Formalismen (wie Mengen, Funktionen) definiert.
  - ◇ Spätestens bei der Anfragesprache werden aber viele Konstruktionen der mathematischen Logik dupliziert.

Datenbanken und Logik (4)

- Jedes DB-Lehrbuch auf Uni-Niveau enthält zwei formale Anfragesprachen für das relationale Modell:
  - ◇ Tupelkalkül (mit Variablen für ganze Tupel) und
  - ◇ Bereichskalkül (mit Variablen für Datenwerte).
- Beide basieren sehr stark auf der mathematischen Logik (es sind eigentlich Spezialfälle).
- Wenn man aber z.B. nur den Tupelkalkül definiert (nicht allgemeine Prädikatenlogik), muss man anschließend bei der Definition des Bereichskalküls vieles neu definieren (ähnlich, aber nicht identisch).

Datenbanken und Logik (5)

- In dieser Vorlesung werden Tupelkalkül und Bereichskalkül als Spezialfälle allgemeiner Prädikatenlogik eingeführt.
- Dies wird erreicht, indem die Datenmodelle (nicht nur das relationale, sondern auch das ER-Modell) in die Prädikatenlogik eingebettet werden.
- Eine Einbettung eines Datenmodells bestimmt Einschränkungen für Signaturen und Interpretationen.

  DB-Schemata legen Signaturen fest, aber man erhält so nicht beliebige Signaturen. DB-Zustände entsprechen Interpretationen, aber umgekehrt ist nicht jede Interpretation Bild eines DB-Zustandes.

Datenbanken und Logik (6)

- Es gibt oft mehrere Möglichkeiten, wie die Konstrukte eines Datenmodells auf Konstrukte der Logik abgebildet werden können (siehe Tupelkalkül und Bereichskalkül).
- Man kann auch, anstatt erst das Datenmodell zu definieren, und dann die Abbildung in die Logik, gleich entsprechende Einschränkungen auf Signaturen und Interpretationen studieren.

  Man muss dann allerdings erklären, dass z.B. Tupelkalkül und Bereichskalkül nicht wirklich verschiedene Datenmodelle sind, sondern auf den gleichen Konzepten/Ideen beruhen und man Abbildungen von Signaturen und Interpretationen in beiden Richtungen definieren kann.

Relationale Datenbanken (1)

- In relationalen Datenbanken werden die Daten in Tabellen abgespeichert, z.B.

Student

SID	Vorname	Nachname
101	Lisa	Weiss
102	Michael	Schmidt
103	Daniel	Sommer
104	Iris	Meier

- Statt Zeilen spricht man formal auch von "Tupeln".

  Bei einer Tabelle mit drei Spalten wie im Beispiel entsprechen die Tabellenzeilen Tripeln, z.B. (101, 'Lisa', 'Weiss'). Bei vier Spalten: Quadrupel, bei fünf Spalten: Quintupel, u.s.w.

Relationale Datenbanken (2)

- In der Logik kann man den Zugriff auf Tabellenzeilen auf zwei verschiedene Arten formalisieren:
  - ◇ Bereichskalkül (BK): Eine Tabelle mit n Spalten entspricht einem n-stelligen Prädikat:
    $p(t_1, \ldots, t_n)$ ist wahr gdw.

t_1	$\cdots$	t_n

    eine Zeile der Tabelle ist.
  - ◇ Tupelkalkül (TK): Eine Tabelle mit n Spalten entspricht einer Sorte mit n Zugriffsfunktionen, die die Werte der Spalten liefern.

    Alternativ statt Sorte auch einstelliges Prädikat, siehe unten.

Relationale Datenbanken (3)

- Im Beispiel würde der Bereichskalkül ein Prädikat `Student` einführen.
  - ◇ `Student(101, 'Lisa', 'Weiss')` wäre wahr.
  - ◇ `Student(200, 'Martin', 'Mueller')` wäre falsch.

    Im Bereichskalkül laufen Variablen über Datentypen (`int`, `string`).
- Der Tupelkalkül würde eine Sorte `Student` einführen, sowie Zugriffsfunktionen `SID`, `Vorname`, `Nachname`.
  - ◇ Für ein `X` der Sorte `Student` gilt dann: `SID(X) = 101`, `Vorname(X) = 'Lisa'`, und `Nachname(X) = 'Weiss'`.

    Im Tupelkalkül laufen Variablen über ganzen Tupeln.

Relationale DBen: BK (1)

- Ein DBMS definiert eine Menge von Datentypen (z.B. Strings, Zahlen) mit Konstanten, Datentypfunktionen (z.B. $+$) und Prädikaten (z.B. $<$).
- Für diese definiert das DBMS Namen (in der Signatur Σ) und Bedeutung (in der Interpretation $\mathcal{I}$).
- Für jeden Wert $d \in \mathcal{I}[s]$ gibt es mindestens eine Konstante c mit $\mathcal{I}[c] = d$.

  D.h. alle Datenwerte sind durch Konstanten benannt. Das wird auch Bereichsabschlußannahme genannt und ist z.B. zur Ausgabe von Datenwerten im Anfrageergebnis wichtig. Im Allg. können verschiedene Konstanten den gleichen Datenwert bezeichnen, z.B. `0`, `00`, `-0`.

Relationale DBen: BK (2)

- Das DB-Schema im relationalen Modell (BK) fügt dann Prädikatsymbole (Relationssymbole) hinzu.

  Diese sind die formale Entsprechung der “Tabellen”.

- Der DB-Zustand interpretiert diese durch endliche Relationen.

  Während die Interpretation der Datentypen festgeschrieben und in das DBMS eingebaut ist, kann die Interpretation der zusätzlichen Prädikatsymbole (DB-Relationen) durch Einfügen, Löschen und Updates verändert werden. Dafür müssen die DB-Relationen aber eine endliche Extension haben. Datentypprädikate sind durch Prozeduren im DBMS implementiert, während die Prädikate des DB-Schemas durch Dateien auf der Platte implementiert werden.

Relationale DBen: BK (3)

- Die wesentlichen Einschränkungen des relationalen Modells (Variante Bereichskalkül) sind also:
  - ◇ Keine neuen Sorten (Typen),
  - ◇ Keine neuen Funktionssymbole und Konstanten,
  - ◇ Neue Prädikatsymbole können nur durch endliche Relationen interpretiert werden.
- Zusätzlich müssen Formeln “bereichsunabhängig” oder “bereichsbeschränkt” sein (siehe unten).

  Diese Einschränkung stellt sicher, dass die erlaubten Formeln in einer gegebenen Interpretation in endlicher Zeit auswertet werden können (obwohl z.B. `int` als unendliche Menge interpretiert wird).

Relationale DBen: BK (4)

Beispiel:

- In einer relationalen DB zur Speicherung von Hausaufgabenpunkten könnte es drei Prädikate geben:
  - ◇ `Student(int SID, string Vorname, string Nachname)`

    Argumentnamen erklären die Bedeutung der Argumente (s.o.). Das erste Argument ist eine eindeutige Nummer, das zweite der Vorname des Studenten mit dieser Nummer und das dritte der Nachname. Z.B. könnte `Student(101, 'Lisa', 'Weiss')` wahr sein.
  - ◇ `Aufgabe(int ANR, int MaxPt)`

    Z.B. bedeutet `Aufgabe(1, 10)`: Für Übung 1 gibt es 10 Punkte.
  - ◇ `Bewertung(int SID, int ANR, int Punkte)`

    Z.B. bedeutet `Bewertung(101, 1, 10)`, dass Lisa Weiss (die Studentin mit Nummer 101) 10 Punkte für Übung 1 bekommen hat.

Relationale DBen: BK (5)

Student

SID	Vorname	Nachname
101	Lisa	Weiss
102	Michael	Schmidt
103	Daniel	Sommer
104	Iris	Meier

Bewertung

SID	ANR	Punkte
101	1	10
101	2	8
102	1	9
102	2	9
103	1	5

Aufgabe

ANR	MaxPt
1	10
2	10

Relationale DBen: TK (1)

- Wie üblich, definiert das DBMS auch beim Tupelkalkül eine Menge von Datentypen (mit Konstanten, Funktionen, Prädikaten).

  In diesem Punkt gibt es zunächst keinen Unterschied zum Bereichskalkül. Es kommen aber noch Record-Typen hinzu (siehe unten).

- Das DB-Schema fügt dann Folgendes hinzu:
  - Sorten (eine für jede Relation/Tabelle).
  - einstellige Funktionen, jeweils von einer der neuen Sorten in einen Datentyp (für jede Spalte).

    Dies sind Zugriffsfunktionen für die Attribute/Komponenten der Zeilen/Tupel/Records.

Relationale DBen: TK (2)

- In der Punkte-DB gibt es z.B. die Sorte `Student` mit den Funktionen
  - `SID(Student): int`
  - `Vorname(Student): string`
  - `Nachname(Student): string`

    Die Werte der Sorte `Student` sind die Tabellenzeilen. Jede der Funktionen liefert den Wert der entsprechenden Spalte für die als Argument gegebene Tabellenzeile. Diese Zugriffsfunktionen tun also nichts anderes, als jeweils eine Komponente des Records/Tupels auszuwählen.

- Beispiel für Überladung: `SID(Bewertung): int`.

Relationale DBen: TK (3)

- Z.B. enthält $\mathcal{I}[\texttt{Student}]$ das Tupel

  $$t = (101, \text{'Lisa'}, \text{'Weiss'})$$

- Dann ist $\mathcal{I}[\texttt{SID}](t) = 101$.
- Selbstverständlich müssen die neuen Sorten als endliche Mengen interpretiert werden (ggf. auch leer).

  Man fordert auch, daß es keine zwei verschiedenen Tupel geben kann, die in den Werten aller Zugriffsfunktionen übereinstimmen. Dies ist für die Äquivalenz zum Bereichskalkül wichtig, da dort ein Prädikat nicht zweimal wahr sein kann. In der Praxis (SQL) könnte es tatsächlich solche Tupel geben, die in allen Komponenten übereinstimmen. Fast immer werden aber Schlüssel für eine Relation/Tabelle definiert, und dann kann dieser Fall wieder nicht auftreten.

Relationale DBen: TK (4)

- Man braucht auch Tupel, die nicht an Tabellen gebunden sind (z.B. für Vereinigung zweier Spalten).
- Formal führt das DBMS hierzu unendlich viele Sorten mit Namen wie "<a: string, b: int>" ein, die
  - fest als das kartesische Produkt der entsprechenden Datentypen interpretiert werden, und
  - auf denen dann Zugriffsfunktionen für die Komponenten definiert sind, im Beispiel a und b.

    Dies ist einfach eine Erweiterung der Datentypen: Es gibt jetzt eben auch Record-Typen.

Relationale DBen: TK (5)

- Tatsächlich braucht man diese allgemeinen Record-Typen nur auf äußerster Ebene (zur Ausgabe).

  SQL vermeidet das und braucht dann aber die Möglichkeit, mehrere Anfragen mit UNION zu verbinden.

- Wenn man die Record-Typen akzeptiert, kann man die Relationen auch als Prädikate der Stelligkeit 1 auf dem entsprechenden Record-Typ verstehen.

  Theoretisch schöner: Nur eine Formalisierung von Tabellenzeilen. Dieser Ansatz ist aber in der praktischen Anwendung in Formeln (Anfragen) etwas mühsamer, so daß man eine Syntax, die "Relationen als Sorten" entspricht, dann doch als Abkürzung einführt.

Entity-Relationship-Modell (1)

- Im Entity-Relationship-Modell kann das DB-Schema folgendes einführen (bei gegebenen Datentypen):
  - neue Sorten ("Entity-Typen"),
  - neue Funktionen der Stelligkeit 1 von Entity-Typen zu Datentypen ("Attribute"),
  - neue Prädikate zwischen Entity-Typen, evtl. beschränkt auf Stelligkeit 2 ("Relationships").
  - neue Funktionen, die auf gleichen Entity-Typen wie Relationships definiert sind, aber einen Datentyp zurückgeben ("Relationship-Attribute").

Entity-Relationship-Modell (2)

- Die Interpretation von Entity-Typen (im Datenbankzustand) muss immer endlich sein.

  Damit sind auch Attribute und Relationships endlich.

- Funktionen für Relationship-Attribute müssen bei Eingabewerten, für die das Relationship falsch ist, einen festen Dummy-Wert als Ausgabe haben.

  Anfragen sollten so geschrieben sein, dass der genaue Dummy-Wert für das Anfrageergebnis unwichtig ist. Z.B. wenn f ein Attribut des Relationships p ist, würde eine Formel der Form $p(X,Y) \wedge f(X,Y) = Z$ diese Eigenschaft haben. Tatsächlich sollte f eine partielle Funktion sein: Dies wird unten im Abschnitt über dreiwertige Logik behandelt.

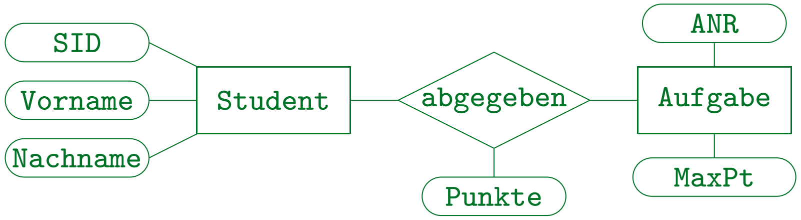
Entity-Relationship-Modell (3)

Beispiel (Hausaufgabenpunkte):

- Sorten: `Student` und `Aufgabe`.
- Funktionen:
  - ◇ `SID(Student): int`
  - ◇ `Vorname(Student): string`
  - ◇ `Nachname(Student): string`.
  - ◇ `ANR(Aufgabe): int`
  - ◇ `MaxPt(Aufgabe): int`
- Prädikat: `hat_abgegeben(Student, Aufgabe)`.
  Funktion: `Punkte(Student, Aufgabe): int`

Entity-Relationship-Modell (4)

- Man kann eine Signatur im ER-Modell gut durch ein Diagramm veranschaulichen (siehe Kapitel 8):



- Rechtecke stehen für Sorten (“Entity-Typen”).
- Ovale für Funktionen (“Attribute”).
- Rauten für Prädikate (“Relationships”).

Datenbank-Entwurf

- Aufgabe des DB-Entwurfs ist es, ein DB-Schema für eine gegebene Anwendung zu entwickeln.
- In der Logik bedeutet das, eine Signatur Σ zu entwerfen, so daß die abzuspeichernde Information in den Σ-Interpretationen stehen kann.

  Genauer muß man nur den anwendungsspezifischen Teil der Signatur entwerfen: Die Datentypen sind durch das DBMS schon vordefiniert.
- Jedes Datenmodell legt bestimmte Einschränkungen für Signaturen und Interpretationen fest.

Inhalt

Variablendeklaration (1)

Definition:

- Sei die Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ gegeben.
- Eine Variablendeklaration für Σ ist eine partielle Abbildung $\nu: \mathit{VARS} \to \mathcal{S}$, so dass
  - ◇ der Definitionsbereich von ν ist endlich, und

    Das ist keine Einschränkung, weil jede Formel nur endlich viele Variablen enthält.
  - ◇ für $c \in \mathcal{F}_\epsilon \cap \mathit{VARS}$ ist $\nu(c)$ undefiniert.

    Wenn ein Bezeichner schon als Konstante verwendet wird, kann er nicht gleichzeitig als Variable benutzt werden, weil sonst Vorkommen des Bezeichners in Formeln mehrdeutig wären.

Variablendeklaration (2)

Beispiel:

- Variablendeklarationen definieren, welche Variablen verwendet werden können, und was ihre Sorten sind:

ν	
Variable	Sorte
SID	int
Punkte	int
A	Aufgabe

  Jede Variable muss eine eindeutige Sorte haben.
- Variablendeklarationen werden auch in der Form $\nu = \{\texttt{SID}/\texttt{int},\ \texttt{Punkte}/\texttt{int},\ \texttt{A}/\texttt{Aufgabe}\}$ geschrieben.

Variablendeklaration (3)

Bemerkung:

- Die Variablendeklaration ist nicht Teil der Signatur, da sie lokal durch Quantoren geändert wird (s.u.).

  Die Signatur ist für die gesamte Anwendung fest, die Variablendeklaration ändert sich schon innerhalb einer der Formel.
- Natürlich wären auch andere Formalisierungen anstatt der partiellen Abbildung möglich.

  Die Abbildung bot sich an, da die semantische Entsprechung, die Variablenbelegung (siehe Folie 3-88), ganz natürlich eine Abbildung ist. Eine partielle Abbildung wurde gewählt, damit man die Variablendeklaration endlich aufschreiben kann.

Variablendeklaration (4)

Definition:

- Sei ν eine Variablendeklaration, $X \in \mathit{VARS}$, und $s \in \mathcal{S}$.
- Dann schreiben wir $\nu\langle X/s\rangle$ für die lokal modifizierte Variablendeklaration ν' mit

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

Bemerkung:

- Beides ist möglich: ν kann für X schon definiert sein, oder an dieser Stelle bisher undefiniert sein.

Terme (1)

- Terme sind syntaktische Konstrukte, die zu einem Wert ausgewertet werden können (z.B. zu einer Zahl, einer Zeichenkette, oder einer Person).
- Es gibt drei Arten von Termen:
  - Konstanten, z.B. `1`, `'abc'`, `arno`,
  - Variablen, z.B. `X`,
  - zusammengesetzte Terme, bestehend aus Funktionssymbolen angewandt auf Argumentterme, z.B. `Nachname(arno)`.

Terme (2)

- Zusammengesetzte Terme können beliebig tief geschachtelt sein, z.B.

```
mult(div(Punkte(S,A), MaxPt(A)), 100)
```

- Oder mit Infix-Operatoren für Multiplikation und Division (nur andere Syntax für Funktionsaufrufe):

```
(Punkte(S,A)/MaxPt(A)) * 100)
```

- Terme sollten aus Programmiersprachen bekannt sein. Dort sagt man Ausdruck oder Wertausdruck (engl. expression) statt Term.

Terme (3)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Die Menge $TE_{\Sigma,\nu}(s)$ der Terme der Sorte s ist folgendermaßen rekursiv definiert:
  - Jede Variable $V \in VARS$ mit $\nu(V) = s$ ist ein Term der Sorte s (dafür muss ν definiert sein für V).
  - Jede Konstante $c \in \mathcal{F}_{\epsilon,s}$ ist ein Term der Sorte s.
  - Wenn t_1 ein Term der Sorte s_1 ist, …, t_n ein Term der Sorte s_n, und $f \in \mathcal{F}_{\alpha,s}$ mit $\alpha = s_1 \ldots s_n$, $n \geq 1$, dann ist $f(t_1, \ldots, t_n)$ ein Term der Sorte s.

Terme (4)

Definition, fortgesetzt:

- Jeder Term kann durch endlich häufige Anwendung obiger Regeln konstruiert werden. Nichts anderes ist ein Term.

  Diese Bemerkung ist formal wichtig, da die obigen Regeln nur festlegen, was ein Term ist, und nicht, was kein Term ist. Dazu muss die Definition abgeschlossen werden. (Selbst wenn man die obigen Regeln als Gleichungssystem auffasst, müßten unendliche Baumstrukturen als Lösungen ausgeschlossen werden.)

Definition:

- $TE_{\Sigma,\nu} := \bigcup_{s \in \mathcal{S}} TE_{\Sigma,\nu}(s)$ sei die Menge aller Terme.

Terme (5)

- Wie oben schon gezeigt, werden einige Funktionen üblicherweise als Infix-Operatoren zwischen ihre Argumente (Operanden) geschrieben, also z.B. `X+1` statt der "offiziellen" Notation `+(X, 1)`.

  Wenn man damit anfängt, muss man auch Rangfolgen (Prioritäten) der Operatoren definieren, und explizite Klammerung erlauben. Die formalen Definitionen werden dadurch komplizierter.

- Aufrufe von Funktionen der Stelligkeit 1 kann man auch in Punktnotation (objektorientiert) schreiben, z.B. "`X.Nachname`" für "`Nachname(X)`".

Terme (6)

- "Syntaktischer Zucker" wie Infix- und Punktnotation ist in der Praxis sinnvoll, aber für die Theorie der Logik nicht wichtig.

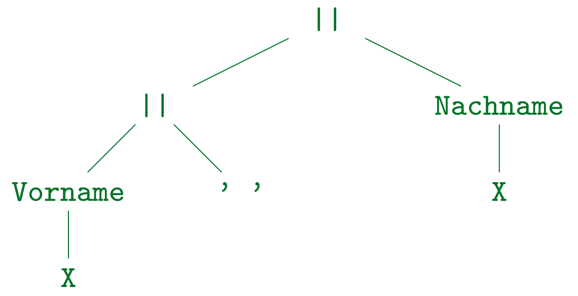
  In Programmiersprachen gibt es manchmal Unterschiede zwischen der "konkreten Syntax" und der "abstrakten Syntax" (Syntaxbaum). Die abstrakte Syntax läßt viele Details weg und beschreibt eher die internen Datenstrukturen des Compilers.

- Im Folgenden nutzen wir die obigen Abkürzungen in praktischen Beispielen, aber die formalen Definitionen behandeln nur die Standardnotation.

  Die Übersetzung von Abkürzungen in Standardnotation sollte offensichtlich sein.

Terme (7)

- Terme kann man in Operatorbäumen visualisieren ("`||`" bezeichnet in SQL die Stringkonkatenation):



- Übung: Wie kann man diesen Term mit "`||`" als Infixoperator und mit Punktnotation schreiben?

Terme (8)

Übung:

- Welche der folgenden Ausdrücke sind korrekte Terme (bezüglich der Signatur auf Folie 3-27 und einer Variablendeklaration ν mit $\nu(\mathtt{X}) = \mathtt{string}$)?

  ☐ `arno`

  ☐ `Vorname`

  ☐ `Vorname(X)`

  ☐ `Vorname(arno, birgit)`

  ☐ `verheiratet(birgit, chris)`

  ☐ `X`

Atomare Formeln (1)

- Formeln sind syntaktische Ausdrücke, die zu einem Wahrheitswert ausgewertet werden können, z.B

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

- Atomare Formeln sind die grundlegenden Bestandteile zur Bildung dieser Formeln (Vergleiche etc.).
- Atomare Formeln können folgende Formen haben:
  - ◇ Ein Prädikatsymbol, angewandt auf Terme, z.B. `verheiratet(birgit, X)` oder `die_sonne_scheint`.
  - ◇ Eine Gleichung, z.B. `X = chris`.
  - ◇ Die logischen Konstanten $\top$ (wahr), $\bot$ (falsch).

Atomare Formeln (2)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Eine atomare Formel ist ein Ausdruck der Form:
  - ◇ $p(t_1, \ldots, t_n)$ mit $p \in \mathcal{P}_\alpha$, $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$, $n > 0$ und $t_i \in TE_{\Sigma,\nu}(s_i)$ für $i = 1, \ldots, n$.
  - ◇ p mit $p \in \mathcal{P}_\epsilon$,
  - ◇ $t_1 = t_2$ mit $t_1, t_2 \in TE_{\Sigma,\nu}(s)$, $s \in \mathcal{S}$.
  - ◇ $\top$ oder $\bot$.
- $AT_{\Sigma,\nu}$ sei die Menge der atomaren Formeln für Σ, ν.

Atomare Formeln (3)

Bemerkungen:

- Für einige Prädikate verwendet man üblicherweise Infixnotation, z.B. $X > 1$ statt $>(X, 1)$.

  In der Praxis (und in den folgenden Beispielen) wird diese übersichtlichere Notation verwendet. In den formalen Definitionen wird hier aber die Standard-Notation vorausgesetzt.

- Es ist möglich, "$=$" als normales Prädikat zu behandeln, wie es einige Autoren auch tun.

  Die obige Definition garantiert, daß zumindest die Gleichheit für alle Sorten verfügbar ist. Die Definition des Wahrheitswertes einer Formel (s.u.) stellt sicher, daß es auch immer die Standardinterpretation hat. Wenn man "$=$" als normales Prädikat behandelt, braucht man entsprechende Axiome, und muß Äquivalenzklassen von Werten bilden.

Formeln (1)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Die Mengen $FO_{\Sigma,\nu}$ der (Σ, ν)-Formeln sind folgendermaßen rekursiv definiert:
  - ◇ Jede atomare Formel $F \in AT_{\Sigma,\nu}$ ist eine Formel.
  - ◇ Wenn F und G Formeln sind, so auch $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
  - ◇ $(\forall\, s\, X\colon F)$ und $(\exists\, s\, X\colon F)$ sind in $FO_{\Sigma,\nu}$ falls $s \in \mathcal{S}$, $X \in VARS$, und F eine $(\Sigma,\ \nu\langle X/s\rangle)$-Formel ist.
  - ◇ Nichts anderes ist eine Formel.

Formeln (2)

- Die intuitive Bedeutung der Formeln ist wie folgt:
  - $\neg F$: "nicht F" (F ist falsch).
  - $F \wedge G$: "F und G" (beide sind wahr).
  - $F \vee G$: "F oder G" (eine oder beide sind wahr).
  - $F \leftarrow G$: "F wenn G" (ist G wahr, so auch F)
  - $F \rightarrow G$: "wenn F, dann G"
  - $F \leftrightarrow G$: "F genau dann, wenn G".
  - $\forall\, s\, X\colon F$: "für alle X (der Sorte s) gilt F".
  - $\exists\, s\, X\colon F$: "es gibt ein X (aus s), so daß F gilt".

Formeln (3)

- Bisher wurden viele Klammern gesetzt, um eine eindeutige syntaktische Struktur zu sichern.

  Für die formale Definition ist das eine einfache Lösung, aber für Formeln in realen Anwendungen wird diese Syntax unpraktisch.

- Regeln zur Klammersetzung:
  - Die äußersten Klammern sind nie notwendig.
  - $\neg$ bindet am stärksten, dann $\wedge$, dann $\vee$, dann $\leftarrow$, $\rightarrow$, $\leftrightarrow$ (gleiche Stärke), und als letztes $\forall$, $\exists$.
  - Da $\wedge$ und $\vee$ assoziativ sind, werden z.B. für $F_1 \wedge F_2 \wedge F_3$ keine Klammern benötigt.

    Beachte, dass $\rightarrow$ und $\leftarrow$ nicht assoziativ sind.

Formeln (4)

Formale Behandlung der Bindungsstärke:

- Eine Formel der Stufe 0 (Formel-0) ist eine atomare Formel oder eine in (...) eingeschlossene Formel-5.

  Die Stufe einer Formel entspricht der Bindungsstärke des äußersten Operators (kleinste Nummer bedeutet höchste Bindungsstärke). Man kann jedoch eine Formel-i wie eine Formel-j verwenden mit $j > i$. In entgegengesetzter Richtung werden Klammern benötigt.

- Eine Formel-1 ist eine Formel-0 oder eine Formel der Form $\neg F$, wobei F eine Formel-1 ist.
- Eine Formel-2 ist eine Formel-1 oder eine Formel der Form $F_1 \wedge F_2$, wobei F_1 eine Formel-2 ist, und F_2 eine Formel-1 (implizite Klammerung von links).

Formeln (5)

Formale Behandlung der Bindungsstärke, fortgesetzt:

- Eine Formel-3 ist eine Formel-2 oder eine Formel der Form $F_1 \vee F_2$ mit einer Formel-3 F_1 und einer Formel-2 F_2.
- Eine Formel-4 ist eine Formel-3 oder eine Formel der Form $F_1 \leftarrow F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$, wobei F_1 und F_2 Formeln der Stufe 3 sind.
- Eine Formel-5 ist eine Formel-4 oder eine Formel der Form $\forall s\ X\colon F$, $\exists\, s\ X\colon F$ mit einer Formel-5 F.
- Eine Formel ist eine Formel der Stufe 5 (Formel-5).

Formeln (6)

Abkürzungen für Quantoren:

- Wenn es nur eine mögliche Sorte für eine quantifizierte Variable gibt, kann man sie weglassen, d.h. $\forall X\colon F$ statt $\forall\, s\, X\colon F$ schreiben (entsprechend für $\exists$).

  Oft ist der Typ der Variablen durch ihre Verwendung eindeutig festgelegt (z.B. Argument eines Prädikates mit eindeutiger Argumentsorte).

- Wenn ein Quantor direkt auf einen anderen Quantor folgt, kann man den Doppelpunkt weglassen.

  Z.B. $\forall X\, \exists Y\colon F$ statt $\forall X\colon \exists Y\colon F$.

- Statt einer Sequenz von Quantoren gleichen Typs, z.B. $\forall X_1 \ldots \forall X_n\colon F$, schreibt man $\forall X_1, \ldots, X_n\colon F$.

Formeln (7)

Abkürzung für Ungleichheit:

- $t_1 \neq t_2$ kann als Abkürzung für $\neg(t_1 = t_2)$ verwendet werden.

Übung:

- Es sei Folgendes gegeben:
  - ◇ Eine Signatur Σ mit $\leq\ \in \mathcal{P}_{\text{int int}}$ und $1, 10 \in \mathcal{F}_{\epsilon,\text{int}}$
  - ◇ Eine Variablendeklaration ν mit $\nu(X) = \texttt{int}$.
- Ist $1 \leq X \leq 10$ eine syntaktisch korrekte Formel?

Formeln (8)

Übung:

- Welche der folgenden Formeln sind syntaktisch korrekt (bezüglich der Signatur von Folie 3-27)?
  - ☐ $\forall\, \texttt{X}, \texttt{Y}\colon \texttt{verheiratet}(\texttt{X}, \texttt{Y}) \rightarrow \texttt{verheiratet}(\texttt{Y}, \texttt{X})$
  - ☐ $\forall\, \texttt{Person P}\colon\ \vee\ \texttt{Mann}(\texttt{P}) \vee \texttt{Frau}(\texttt{P})$
  - ☐ $\forall\, \texttt{Person P}\colon\ \texttt{arno} \vee \texttt{birgit} \vee \texttt{chris}$
  - ☐ $\texttt{Mann}(\texttt{chris})$
  - ☐ $\forall\, \texttt{string X}\colon\ \exists\, \texttt{Person X}\colon\ \texttt{verheiratet}(\texttt{birgit}, \texttt{X})$
  - ☐ $\texttt{verheiratet}(\texttt{birgit}, \texttt{chris})$ $\wedge\ \vee\ \texttt{verheiratet}(\texttt{chris}, \texttt{birgit})$

Geschlossenene Formeln (1)

Definition:

- Sei eine Signatur Σ gegeben.
- Eine geschlossene Formel (für Σ) ist eine (Σ, ν)-Formel für die leere Variablendeklaration ν.

  D.h. die Variablendeklaration, die überall undefiniert ist.

Übung:

- Welche der folgenden Formeln sind geschlossen?
  - ☐ $\texttt{Frau}(\texttt{X}) \wedge \exists\, \texttt{X}\colon \texttt{verheiratet}(\texttt{chris}, \texttt{X})$
  - ☐ $\texttt{Frau}(\texttt{birgit}) \wedge \texttt{verheiratet}(\texttt{chris}, \texttt{birgit})$
  - ☐ $\exists\, \texttt{X}\colon \texttt{verheiratet}(\texttt{X}, \texttt{Y})$

Geschlossenene Formeln (2)

Bemerkung:

- Um festzulegen, ob eine Formel wahr oder falsch ist, braucht man außer einer Interpretation auch Werte für die Variablen, die nicht durch Quantoren gebunden sind (freie Variablen).
- Bei geschlossenen Formeln reicht die Interpretation.

  Alle in der Formel verwendeten Variablen sind in der Formel selbst durch einen Quantor eingeführt. Es gibt keine Variablen, die schon von außen kommen (gewissermaßen als Parameter).

Variablen in einem Term

Definition:

- Die Funktion $vars$ berechnet die Menge der Variablen, die in einem gegebenem Term t auftreten.
  - ◇ Wenn t eine Konstante c ist:
    $$vars(t) := \emptyset.$$
  - ◇ Wenn t eine Variable V ist:
    $$vars(t) := \{V\}.$$
  - ◇ Wenn t die Form $f(t_1, \ldots, t_n)$ hat:
    $$vars(t) := \bigcup_{i=1}^{n} vars(t_i).$$

Freie Variablen einer Formel

Definition:

- $free(F)$ ist die Menge der freien Variablen in F:
  - ◇ Ist F atomare Formel $p(t_1, \ldots, t_n)$ oder $t_1 = t_2$:
    $$free(F) := \bigcup_{i=1}^{n} vars(t_i).$$
  - ◇ Ist F logische Konstante $\top$ oder $\bot$: $free(F) := \emptyset$.
  - ◇ Wenn F die Form $(\neg G)$ hat: $free(F) := free(G)$.
  - ◇ Wenn F die Form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc. hat: $free(F) := free(G_1) \cup free(G_2)$.
  - ◇ Wenn F die Form $(\forall s\, X\colon G)$ oder $(\exists s\, X\colon G)$ hat: $free(F) := free(G) - \{X\}$.

Variablenbelegung (1)

Definition:

- Eine Variablenbelegung $\mathcal{A}$ für $\mathcal{I}$ und ν ist eine partielle Abbildung von $VARS$ auf $\bigcup_{s \in \mathcal{S}} \mathcal{I}[s]$.
- Sie definiert für jede Variable V, für die ν definiert ist, einen Wert aus $\mathcal{I}[s]$, wobei $s := \nu(V)$.

Bemerkung:

- D.h. eine Variablenbelegung legt für alle Variablen, die in ν deklariert sind, Werte aus $\mathcal{I}$ fest (jeweils passender Sorte).

Variablenbelegung (2)

Beispiel:

- Es sei folgende Variablendeklaration ν gegeben:

ν	
Variable	Sorte
X	string
Y	Person

- Eine mögliche Variablenbelegung ist

$\mathcal{A}$	
Variable	Wert
X	abc
Y	Chris

Variablenbelegung (3)

Definition:

- $\mathcal{A}\langle X/d\rangle$ sei die Variablenbelegung $\mathcal{A}'$, die bis auf $\mathcal{A}'(X) = d$ mit $\mathcal{A}$ übereinstimmt.

Beispiel:

- Wenn $\mathcal{A}$ die Variablenbelegung von der letzten Folie ist, so ist $\mathcal{A}\langle \texttt{Y}/\text{Birgit}\rangle$:

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

Wert eines Terms

Definition:

- Sei eine Signatur Σ, eine Variablendeklaration ν für Σ, eine Σ-Interpretation $\mathcal{I}$, und eine Variablenbelegung $\mathcal{A}$ für $(\mathcal{I}, \nu)$ gegeben.
- Der Wert $\langle \mathcal{I}, \mathcal{A}\rangle[t]$ eines Terms $t \in TE_{\Sigma,\nu}$ ist wie folgt definiert (Rekursion über die Termstruktur):
  - ◇ Ist t eine Konstante c, dann ist $\langle \mathcal{I}, \mathcal{A}\rangle[t] := \mathcal{I}[c]$.
  - ◇ Ist t eine Variable V, dann ist $\langle \mathcal{I}, \mathcal{A}\rangle[t] := \mathcal{A}(V)$.
  - ◇ Hat t die Form $f(t_1, \ldots, t_n)$, mit t_i der Sorte s_i: $\langle \mathcal{I}, \mathcal{A}\rangle[t] := \mathcal{I}[f, s_1 \ldots s_n](\langle \mathcal{I}, \mathcal{A}\rangle[t_1], \ldots, \langle \mathcal{I}, \mathcal{A}\rangle[t_n])$.

Wahrheit einer Formel (1)

Definition:

- Der Wahrheitswert $\langle \mathcal{I}, \mathcal{A}\rangle[F] \in \{\mathbf{f}, \mathbf{w}\}$ einer Formel F in $(\mathcal{I}, \mathcal{A})$ ist definiert als (**f** bedeutet falsch, **w** wahr):
  - ◇ Ist F eine atomare Formel $p(t_1, \ldots, t_n)$ mit den Termen t_i der Sorte s_i:

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} \mathbf{w} & \text{falls } (\langle \mathcal{I}, \mathcal{A}\rangle[t_1], \ldots, \langle \mathcal{I}, \mathcal{A}\rangle[t_n]) \\ & \quad \in \mathcal{I}[p,\, s_1 \ldots s_n] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

  - ◇ (auf den nächsten 3 Folien fortgesetzt ...)

Wahrheit einer Formel (2)

Definition, fortgesetzt:

- Wahrheitswert einer Formel, fortgesetzt:
  - Ist F eine atomare Formel $t_1 = t_2$:

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} \mathbf{w} & \text{falls } \langle \mathcal{I}, \mathcal{A}\rangle[t_1] = \langle \mathcal{I}, \mathcal{A}\rangle[t_2] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

  - Ist F "true" $\top$: $\langle \mathcal{I}, \mathcal{A}\rangle[F] := \mathbf{w}$.
  - Ist F "false" $\bot$: $\langle \mathcal{I}, \mathcal{A}\rangle[F] := \mathbf{f}$.
  - Hat F die Form $(\neg G)$:

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

Wahrheit einer Formel (3)

Definition, fortgesetzt:

- Wahrheitswert einer Formel, fortgesetzt:
  - Hat F die Form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.:

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

Z.B. falls $\langle \mathcal{I}, \mathcal{A}\rangle[G_1] = \mathbf{w}$ und $\langle \mathcal{I}, \mathcal{A}\rangle[G_2] = \mathbf{f}$ dann $\langle \mathcal{I}, \mathcal{A}\rangle[(G_1 \wedge G_2)] = \mathbf{f}$.

Wahrheit einer Formel (4)

Definition, fortgesetzt:

- Wahrheitswert einer Formel, fortgesetzt:
  - Hat F die Form $(\forall\, s\, X\colon G)$:

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} \mathbf{w} & \text{falls } \langle \mathcal{I}, \mathcal{A}\langle X/d\rangle\rangle[G] = \mathbf{w} \\ & \quad \text{für alle } d \in \mathcal{I}[s] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

  - Hat F die Form $(\exists\, s\, X\colon G)$:

$$\langle \mathcal{I}, \mathcal{A}\rangle[F] := \begin{cases} \mathbf{w} & \text{falls } \langle \mathcal{I}, \mathcal{A}\langle X/d\rangle\rangle[G] = \mathbf{w} \\ & \quad \text{für mindestens ein } d \in \mathcal{I}[s] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

Modell (1)

Definition:

- Ist $\langle \mathcal{I}, \mathcal{A}\rangle[F] = \mathbf{w}$, so schreibt man auch $\langle \mathcal{I}, \mathcal{A}\rangle \models F$. Dann heißt $\langle \mathcal{I}, \mathcal{A}\rangle$ ein Modell von F.

  Man sagt dann auch $\langle \mathcal{I}, \mathcal{A}\rangle$ erfüllt F bzw. F ist in $\langle \mathcal{I}, \mathcal{A}\rangle$ wahr.

  Viele Autoren wenden den Beriff "Modell" allerdings nur auf Interpretationen allein an (ohne Variablenbelegung), wie im folgenden Punkt beschrieben.

- Sei F eine (Σ, ν)-Formel. Gilt $\langle \mathcal{I}, \mathcal{A}\rangle[F] = \mathbf{w}$ für alle Variablenbelegungen $\mathcal{A}$ (für $\mathcal{I}$ und ν), so schreibt man $\mathcal{I} \models F$ und nennt $\mathcal{I}$ ein Modell von F.

  D.h. freie Variablen werden als $\forall$-quantifiziert behandelt. Die Variablenbelegung ist aber irrelevant, falls F eine geschlossene Formel ist.

Modell (2)

Definition:

- Eine Formel F heißt konsistent gdw. es $\mathcal{I}$ und $\mathcal{A}$ gibt mit $\langle \mathcal{I}, \mathcal{A} \rangle \models F$ (d.h. wenn sie ein Modell hat).

  Entsprechend für Mengen von Formeln.

  Manche Autoren nennen eine Formel nur dann konsistent, wenn sie in einer Interpretation $\mathcal{I}$ für alle Variablenbelegungen $\mathcal{A}$ wahr ist. Wenn sie nur für mindestens eine Variablenbelegung wahr ist, würde die Formel "erfüllbar" heißen.

- F ist inkonsistent gdw. F ist nicht konsistent.

  D.h. F ist immer falsch, egal welche Interpretation und Variablenbelegung man nimmt. Mit anderen Worten: F hat kein Modell. (Man beachte wieder, daß es in manchen Büchern "unerfüllbar" heißt, und inkonsistent etwas schwächer ist.)

Modell (3)

Definition:

- $\langle \mathcal{I}, \mathcal{A} \rangle \models \Phi$ für eine Menge Φ von Σ-Formeln gdw. $\langle \mathcal{I}, \mathcal{A} \rangle \models F$ für alle $F \in \Phi$.

  Analog wird $\mathcal{I} \models \Phi$, Modell und Konsistenz einer Menge Φ von Formeln definiert: Man fordert die entsprechende Eigenschaft jeweils für alle Formeln $F \in \Phi$.

Definition:

- Eine (Σ, ν)-Formel F nennt man Tautologie gdw. für alle Σ-Interpretationen $\mathcal{I}$ und (Σ, ν)-Variablenbelegungen $\mathcal{A}$ gilt: $(\mathcal{I}, \mathcal{A}) \models F$.

  D.h. Tautologien sind immer wahr.

Modell (4)

Übung:

- Man betrachte die Interpretation auf Folie 3-34:
  - ◇ $\mathcal{I}[\texttt{Person}] = \{\text{Arno}, \text{Birgit}, \text{Chris}\}$.
  - ◇ $\mathcal{I}[\texttt{verheiratet}] = \{(\text{Birgit}, \text{Chris}), (\text{Chris}, \text{Birgit})\}$.
  - ◇ $\mathcal{I}[\texttt{Mann}] = \{(\text{Arno}), (\text{Chris})\}$, $\mathcal{I}[\texttt{Frau}] = \{(\text{Birgit})\}$.
- Welche der folgenden Formeln ist in $\mathcal{I}$ wahr?
  - ☐ $\forall\, \texttt{Person X: Mann(X)} \leftrightarrow \neg\texttt{Frau(X)}$
  - ☐ $\forall\, \texttt{Person X: Mann(X)} \vee \neg\texttt{Mann(X)}$
  - ☐ $\exists\, \texttt{Person X: Frau(X)} \wedge \neg\exists\, \texttt{Person Y: verheiratet(X,Y)}$
  - ☐ $\exists\, \texttt{Person X, Person Y, Person Z: X=Y} \wedge \texttt{Y=Z} \wedge \texttt{X}\neq\texttt{Z}$

Inhalt

Formeln in Datenbanken (1)

- Wie oben erklärt, definiert das DBMS eine Signatur $\Sigma_{\mathcal{D}}$ und eine Interpretation $\mathcal{I}_{\mathcal{D}}$ für die eingebauten Datentypen (`string`, `int`, ...).

- Dann erweitert das DB-Schema diese Signatur $\Sigma_{\mathcal{D}}$ zu der Signatur Σ aller Symbole, die z.B. in Anfragen verwendet werden können.

  Jedes Datenmodell legt gewisse Restriktionen für die Art der neuen Symbole, die eingeführt werden können, fest. Z.B. kann im klassischen relationalen Modell (Bereichskalkül) das Datenbankschema nur neue Prädikatsymbole definieren.

Formeln in Datenbanken (2)

- Der DB-Zustand ist dann eine Interpretation $\mathcal{I}$ für die erweiterte Signatur Σ.

  Das System speichert natürlich nur die Interpretation der Symbole von "$\Sigma - \Sigma_{\mathcal{D}}$" explizit ab, da die Interpretation der Symbole in $\Sigma_{\mathcal{D}}$ schon in das DBMS eingebaut ist und nicht verändert werden kann. Außerdem können nicht beliebige Interpretationen als DB-Zustand verwendet werden. Die genauen Restriktionen hängen vom Datenmodell ab, aber die neuen Symbole müssen eine endliche Interpretation haben.

- Formeln werden in Datenbanken verwendet als:
  - ◇ Integritätsbedingungen (Constraints)
  - ◇ Anfragen (Queries)
  - ◇ Definition abgeleiteter Symbole (Sichten, Views).

Integritätsbedingungen (1)

- Nicht jede Interpretation ist als DB-Zustand zulässig.

  Der Zweck einer DB ist, einen Teil der realen Welt zu modellieren. In der realen Welt existieren gewisse Restriktionen. Dafür sollen Interpretationen, die diese Restriktionen verletzen, ausgeschlossen werden.

- Z.B. muß eine Person in der realen Welt männlich oder weiblich sein, aber nicht beides. Daher sind folgende Formeln in jedem sinnvollen Zustand erfüllt:
  - ◇ $\forall\,\mathtt{Person}\ \mathtt{X}\colon\ \mathtt{Mann(X)} \lor \mathtt{Frau(X)}$
  - ◇ $\forall\,\mathtt{Person}\ \mathtt{X}\colon\ \neg\,\mathtt{Mann(X)} \lor \neg\,\mathtt{Frau(X)}$

- Dies sind Beispiele für Integritätsbedingungen.

Integritätsbedingungen (2)

- Integritätsbedingungen sind geschlossene Formeln.

  Das Datenmodell muss nicht beliebige Formeln zulassen. Eine typische Einschränkung ist die Bereichsunabhängigkeit (siehe unten).

- Eine Menge von Integritätsbedingungen wird als Teil des DB-Schemas spezifiziert.

- Einen DB-Zustand (Interpretation) nennt man zulässig, wenn er alle Integritätsbedingungen erfüllt.

  Im Folgenden betrachten wir nur zulässige DB-Zustände. Dabei sprechen wir dann wieder nur von "DB-Zustand".

Integritätsbedingungen (3)

Schlüssel:

- Objekte werden oft durch eindeutige Datenwerte (Zahlen, Namen) identifiziert.
- Beispiel (Punkte-DB, ER-Variante, Folie 3-57): Es soll keine zwei verschiedene Objekte des Typs `Student` mit der gleichen `SID` geben:

  $\forall\,\texttt{Student X, Student Y: SID(X)} = \texttt{SID(Y)} \rightarrow \texttt{X} = \texttt{Y}$

- Alternative, äquivalente Formulierung:

  $\neg\exists\,\texttt{Student X, Student Y: SID(X)} = \texttt{SID(Y)} \wedge \texttt{X} \neq \texttt{Y}$

Integritätsbedingungen (4)

Schlüssel, Forts.:

- In der relationalen Variante (Bereichskalkül, Folie 3-48) wird ein Prädikat der Stelligkeit 3 verwendet, um die Studentendaten zu speichern.
- Das erste Argument (`SID`) identifiziert die Werte der anderen (`Vorname`, `Nachname`) eindeutig:

  $\forall\,\texttt{int SID, string V1, string V2, string N1, string N2:}$
  $\quad\texttt{Student(SID,V1,N1)} \wedge \texttt{Student(SID,V2,N2)} \rightarrow$
  $\quad\quad\texttt{V1} = \texttt{V2} \wedge \texttt{N1} = \texttt{N2}$

- Da Schlüssel in der Praxis häufig vorkommen, hat jedes Datenmodell eine spezielle Notation dafür.

Integritätsbedingungen (5)

Übung:

- Man betrachte das Schema auf Folie 3-48:
  - ◇ `Student(int SID, string Vorname, string Nachname)`
  - ◇ `Aufgabe(int ANR, int MaxPt)`
  - ◇ `Bewertung(int SID, int ANR, int Punkte)`
- Formulieren Sie folgende Integritätsbedingungen:
  - ◇ Die `Punkte` in `Bewertung` sind stets nichtnegativ.
  - ◇ Für jede `ANR`, die in der Tabelle `Bewertung` vorkommt, gibt es auch einen Eintrag in `Aufgabe`.

    Dies ist ein Beispiel einer "Fremdschlüsselbedingung". Sie entspricht dem Verbot von "broken links" / "Pointern ins Nirvana".

Anfragen: Form A

- Eine Anfrage (Form A) ist ein Ausdruck der Form

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

  wobei F eine Formel bzgl. der gegebenen Signatur Σ und Variablendeklaration $\{X_1/s_1, \ldots, X_n/s_n\}$ ist.

  Auch hier kann es Restriktionen für die möglichen Formeln F geben, vor allem die Bereichsunabhängigkeit (siehe unten).

- Die Anfrage fragt nach allen Variablenbelegungen $\mathcal{A}$ für die Ergebnisvariablen $X_1, \ldots, X_n$, für die die Formel F im gegebenen DB-Zustand $\mathcal{I}$ wahr ist.

  Um sicherzustellen, dass die Variablenbelegungen ausgegeben werden können, sollten die Sorten s_i der Ergebnisvariablen Datentypen sein.

Anfragen: BK (1)

Beispiel I:

- Man betrachte das Schema auf Folie 3-48:
  - ◇ Student(int SID, string Vorname, string Nachname)
  - ◇ Aufgabe(int ANR, int MaxPt)
  - ◇ Bewertung(int SID, int ANR, int Punkte)
- Wer hat mindestens 8 Punkte für Hausaufgabe 1?

$$\{\texttt{string Vorname, string Nachname} \mid \exists\, \texttt{int S, int P:}\\ \quad \texttt{Student(S, Vorname, Nachname)} \land\\ \quad \texttt{Bewertung(S, 1, P)} \land \texttt{P} \geq 8\}$$

Anfragen: BK (2)

- Die Formeln Student(S, Vorname, Nachname) und Bewertung(S, 1, P) entsprechen Tabellenzeilen:

Student		
SID	Vorname	Nachname
S	Vorname	Nachname

Bewertung		
SID	ANR	Punkte
S	1	P

- Durch die gleiche Variable S werden die Einträge in den beiden Tabellen verknüpft (“Join”): Sie müssen sich auf den gleichen Studenten beziehen.

  Eine Variable kann zu einem Zeitpunkt (d.h. in einer Variablenbelegung) ja nur einen Wert haben.

Anfragen: BK (3)

Übung:

- Was halten Sie von dieser Anfrage?

$$\{\texttt{string Vorname, string Nachname} \mid\\ \quad (\exists\, \texttt{int S: Student(S, Vorname, Nachname)}) \land\\ \quad (\exists\, \texttt{int S, int P: Bewertung(S, 1, P)} \land \texttt{P} \geq 8)\}$$

Bemerkung:

- Wenn die atomaren Formeln nicht durch gemeinsame Variablen verknüpft sind, liegt sehr häufig ein Fehler vor.

Anfragen: BK (4)

Beispiel II:

- Geben Sie alle Ergebnisse von Lisa Weiss aus:

$$\{\texttt{int Aufg, int Punkte} \mid \exists\, \texttt{int Stud:}\\ \quad \texttt{Student(Stud, 'Lisa', 'Weiss')} \land\\ \quad \texttt{Bewertung(Stud, Aufg, Punkte)}\}$$

- Entspricht den beiden verknüpften Tabellenzeilen:

Student		
SID	Vorname	Nachname
Stud	'Lisa'	'Weiss'

Bewertung		
SID	ANR	Punkte
Stud	Aufg	Punkte

Beide Beispiele folgen dem gleichen (häufigen) Muster: Alle Variablen, die nicht ausgegeben werden, sind existenz-quantifiziert. Die Ausgabevariablen eigentlich auch: Man sucht ja Belegungen, die die Bedingung wahr machen.

Anfragen: BK (5)

Übung:

- Geben Sie die Studenten aus, die 10 Punkte in Übung 1 und 10 Punkte in Übung 2 haben.

  Geben Sie bitte den Nachnamen und den Vornamen dieser Studierenden aus, nicht nur die Nummer.

- Die relevanten Tabellen/Prädikate sind:

Student		
SID	Vorname	Nachname

Bewertung		
SID	ANR	Punkte

Anfragen: BK (6)

Beispiel III:

- Wer hat Übung 2 bisher noch nicht eingereicht?

  {string VName, string NName |
  ∃ int SID: Student(SID, VName, NName) ∧
  ¬ ∃ int P: Bewertung(SID, 2, P)}

- Hier fordert man die Nicht-Existenz einer Zeile in der Tabelle Bewertung von der angegebenen Form, also für den gleichen Studenten, die Aufgabe 2, und eine beliebige Punktzahl:

Student		
SID	Vorname	Nachname
SID	VName	NName

	Bewertung		
	SID	ANR	Punkte
¬	SID	2	P

Anfragen: BK (7)

Übung:

- Was halten Sie von folgender Variante der obigen Anfrage? Würde sie auch funktionieren?

  {string VName, string NName |
  ∃ int SID, int P: Student(SID, VName, NName) ∧
  ¬Bewertung(SID, 2, P)}

Bemerkung:

- In Prolog kann man nicht benötigte Argumente eines Prädikates mit "_" füllen.

  Dies ist eine Abkürzung für eine neue Variable, die direkt vor der atomaren Formel existenz-quantifiziert ist ("anonyme Variable").

Anfragen: BK (8)

Bemerkung:

- Es gibt oder gab eine graphische Anfragesprache namens QBE ("query by example"), die mit Tabellengerüsten wie oben gezeigt arbeitet.
- Variablen sind hier Zeichenketten, die mit einem Unterstrich "_" beginnen, z.B. _101 ("z.B. 101").

Student		
SID	Vorname	Nachname
_101	Lisa	Weiss

Bewertung		
SID	ANR	Punkte
_101	P.	P.

- P. ist der Befehl, den Tabelleneintrag an dieser Stelle zu drucken.

Anfragen: Form B

- Eine Anfrage (Form B) ist ein Ausdruck der Form

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

  wobei F eine Formel und die t_i Terme für die gegebene DB-Signatur Σ und die Variablendeklaration $\{X_1/s_1, \ldots, X_n/s_n\}$ sind.
- In diesem Fall gibt das DBMS die Werte $\langle \mathcal{I}, \mathcal{A}\rangle[t_i]$ der Terme t_i für jede Belegung $\mathcal{A}$ der Ergebnisvariablen $X_1, \ldots, X_n$ aus, so dass $\langle \mathcal{I}, \mathcal{A}\rangle \models F$.

  Diese Form der Anfrage ist vor allem dann praktisch, wenn die Variablen X_i von Sorten sind, die sonst nicht ausgegeben werden können.

Anfragen: ER (1)

Beispiel:

- Sei das Schema im ER-Modell gegeben (→ 3-57):
  - ◇ Sorten `Student`, `Aufgabe`.
  - ◇ Funktionen `Vorname(Student): string`, ...
  - ◇ Prädikat: `hat_abgegeben(Student, Aufgabe)`.
    Funktion: `Punkte(Student, Aufgabe): int`
- Wer hat mindestens 8 Punkte für Hausaufgabe 1?

$$\{\texttt{S.Vorname}, \texttt{S.Nachname}\ [\texttt{Student S}] \mid \exists\, \texttt{Aufgabe A}: \texttt{A.ANR} = 1 \land \texttt{hat\_abgegeben(S, A)} \land \texttt{Punkte(S, A)} \geq 8\}$$

Anfragen: ER (2)

Bemerkung:

- Im ER-Modell werden Objekte also typischerweise über Relationship-Prädikate verknüpft.

  Obwohl natürlich auch Wertvergleiche möglich sind.

Übung:

- Könnte man die Variable `A` auch direkt im Kopf der Anfrage mit deklarieren?

$$\{\texttt{S.Vorname}, \texttt{S.Nachname}\ [\texttt{Student S}, \texttt{Aufgabe A}] \mid \texttt{A.ANR} = 1 \land \texttt{hat\_abgegeben(S, A)} \land \texttt{Punkte(S, A)} \geq 8\}$$

Anfragen: ER (3)

Beispiel:

- Wer hat Aufgabe 2 noch nicht abgegeben?

$$\{\texttt{S.Vorname}, \texttt{S.Nachname}\ [\texttt{Student S}] \mid \exists\, \texttt{Aufgabe A}: \texttt{A.ANR} = 2 \land \lnot\texttt{hat\_abgegeben(S, A)}\}$$

Übung:

- Was halten Sie von dieser Lösung?

$$\{\texttt{S.Vorname}, \texttt{S.Nachname}\ [\texttt{Student S}] \mid \forall\, \texttt{Aufgabe A}: \texttt{A.ANR} = 2 \to \lnot\texttt{hat\_abgegeben(S, A)}\}$$

- Wie verhalten sich beide Lösungen, wenn es keine Aufgabe 2 in der Datenbank gibt?

Anfragen: TK (1)

Beispiel:

- Sei nun die Signatur für die Hausaufgaben-DB im Relationenmodell/Tupelkalkül betrachtet (→ 3-57):
  - ◇ Sorte `Student` mit Zugriffsfunktionen für Spalten:
    `SID(Student): int,`
    `Vorname(Student): string,`
    `Nachname(Student): string.`
  - ◇ Sorte `Bewertung`, Funktionen `SID`, `ANR`, `Punkte`.
  - ◇ Sorte `Aufgabe`, Funktionen `ANR`, `MaxPt`.

Anfragen: TK (2)

- Wer hat mindestens 8 Punkte für Hausaufgabe 1?

  $\{$`S.Vorname, S.Nachname [Student S]` $\mid$
  $\exists$ `Bewertung B: B.ANR` $= 1 \wedge$
  `B.SID` $=$ `S.SID` $\wedge$ `B.Punkte` $\geq 8\}$

- Variablen laufen im Tupelkalkül über Tabellenzeilen.

  Eine Variablenbelegung weist jeder Variablen eine bestimmte Zeile zu. Im Prinzip werden alle möglichen Variablenbelegungen durchprobiert.

- Es werden typischerweise Gleichungen verwendet, um Tabellenzeilen zu verknüpfen.

  Wobei meist eindeutige Identifikationen, wie hier die Studentennummer, beteiligt sind.

Anfragen: TK (3)

Student

	SID	Vorname	Nachname
S:	101	Lisa	Weiss
		drucken	drucken

Bewertung

	SID	ANR	Punkte
B:	101	1	10
		$= 1$	≥ 8

$=$

- Die Anfrage betrachtet jeweils zwei Tabellenzeilen gleichzeitig: `S` aus `Student` und `B` aus `Bewertung`. Die Zeile aus `Bewertung` muss sich auf den/die Studierende `S` beziehen, also die gleiche `SID` enthalten.

Anfragen: TK (4)

- Man kann sich die Auswertung so vorstellen:

```
foreach S in Student do
    exists := false;
    foreach B in Bewertung do
        if B.ANR = 1 and B.SID = S.SID
                and B.Punkte >= 8 then
            exists := true;
        fi
    od;
    if exists then
        print S.Vorname, S.Nachname;
    fi
od
```

Anfragen: TK (5)

- Man kann auch `B` im Kopf der Anfrage (zusammen mit `S`) deklarieren, die Auswertung entspricht dann:

```
foreach S in Student do
    foreach B in Bewertung do
        if B.ANR = 1 and B.SID = S.SID
                and B.Punkte >= 8 then
            print S.Vorname, S.Nachname;
        fi
    od
od
```

- Im Beispiel macht dies keinen Unterschied, allgemein kann es Duplikate geben.

Anfragen: TK (6)

- In beiden Fällen werden (theoretisch) alle $4*5 = 20$ möglichen Variablenbelegungen für `S` und `B` durchgegangen.
- Die erste erzeugt die Ausgabe "Lisa Weiss":

	Student		
	SID	Vorname	Nachname
S:	101	Lisa	Weiss
	102	Michael	Schmidt
	103	Daniel	Sommer
	104	Iris	Meier

	Bewertung		
	SID	ANR	Punkte
B:	101	1	10
	101	2	8
	102	1	9
	102	2	9
	103	1	5

Anfragen: TK (7)

- Als nächstes wird die folgende Variablenbelegung betrachtet:

	Student		
	SID	Vorname	Nachname
S:	101	Lisa	Weiss
	102	Michael	Schmidt
	103	Daniel	Sommer
	104	Iris	Meier

	Bewertung		
	SID	ANR	Punkte
	101	1	10
B:	101	2	8
	102	1	9
	102	2	9
	103	1	5

- Sie erfüllt aber nicht die Bedingung `B.ANR = 1`, daher gibt es keine Ausgabe.

Anfragen: TK (8)

- Bei der nächsten Variablenbelegung ist `B.SID = S.SID` verletzt (wieder keine Ausgabe):

	Student		
	SID	Vorname	Nachname
S:	101	Lisa	Weiss
	102	Michael	Schmidt
	103	Daniel	Sommer
	104	Iris	Meier

	Bewertung		
	SID	ANR	Punkte
	101	1	10
	101	2	8
B:	102	1	9
	102	2	9
	103	1	5

Ein Optimierer kann natürlich einen Auswertungsalgorithmus wählen, der solche Variablenbelegungen nicht explizit betrachtet. Um die Semantik der Anfrage zu verstehen, reicht aber dieser naive Algorithmus.

Anfragen: TK (9)

- Und so weiter. Die nächste Ausgabe gibt es bei folgender Variablenbelegung:

	Student		
	SID	Vorname	Nachname
	101	Lisa	Weiss
S:	102	Michael	Schmidt
	103	Daniel	Sommer
	104	Iris	Meier

	Bewertung		
	SID	ANR	Punkte
	101	1	10
	101	2	8
B:	102	1	9
	102	2	9
	103	1	5

- In diesem Beispiel gibt es dann keine weiteren Ausgaben mehr.

Anfragen: TK (10)

- Man hätte die Anfrage (wer hat mindestens 8 Punkte für Hausaufgabe 1) auch mit einer Variablen für die Aufgabe selbst formulieren können:

$$\{\texttt{S.Vorname, S.Nachname [Student S]} \mid \exists\,\texttt{Bewertung B, Aufgabe A:}\ \texttt{A.ANR} = 1 \land \texttt{B.ANR} = \texttt{A.ANR} \land \texttt{B.SID} = \texttt{S.SID} \land \texttt{B.Punkte} \geq 8\}$$

- Dies ist logisch äquivalent (unnötige Verkomplizierung: besser vermeiden).

Die Äquivalenz setzt voraus, daß Aufgabe 1, wenn sie in `Bewertung` vorkommt, auch in `Aufgabe` eingetragen sein muß. Dies ist eine wichtige Integritätsbedingung, die im Schema formuliert werden sollte.

Anfragen: TK (11)

Beispiel II:

- Wer hat Übung 2 bisher noch nicht eingereicht?

$$\{\texttt{S.Vorname, S.Nachname [Student S]} \mid \neg\exists\,\texttt{Bewertung B: B.SID} = \texttt{S.SID} \land \texttt{B.ANR} = 2\}$$

- Andere mögliche Lösung:

$$\{\texttt{S.Vorname, S.Nachname [Student S]} \mid \forall\,\texttt{Bewertung B: B.SID} = \texttt{S.SID} \rightarrow \texttt{B.ANR} \neq 2\}$$

- Noch eine Lösung:

$$\{\texttt{S.Vorname, S.Nachname [Student S]} \mid \forall\,\texttt{Bewertung B: B.ANR} = 2 \rightarrow \texttt{B.SID} \neq \texttt{S.SID}\}$$

Anfragen: TK (12)

Übung:

- Formulieren Sie folgende Anfragen im Tupelkalkül:
  - ◇ Geben Sie alle Ergebnisse für Aufgabe 1 aus (drucken Sie jeweils Vorname, Name, Punktzahl).
  - ◇ Wer hat 10 Punkte in Aufgabe 1 oder 2?
  - ◇ Wer hat sowohl in Aufgabe 1, als auch in Aufgabe 2 jeweils 10 Punkte?
- Ist dies eine korrekte Lösung der letzten Aufgabe?

$$\{\texttt{S.Vorname, S.Nachname [Student S, Bewertung B]} \mid \texttt{B.ANR} = 1 \land \texttt{B.ANR} = 2 \land \texttt{S.SID} = \texttt{B.SID} \land \texttt{B.Punkte} = 10\}$$

Anfragen: TK (13)

Bemerkung:

- Im reinen Tupelkalkül sind Variablen nur über Tupeln erlaubt, also über den Sorten für die Tabellen bzw. den Recordtypen.

  Man darf keine Variablen über den Datentypen (`int`, `string`, ...) einführen.

- Oft wird auch ausgeschlossen, daß man die Gleichheit auf Tupelebene verwendet.

  Bei Gleichheitsbedingungen muß dann links und rechts ein Term einer Datensorte stehen (`int`, `string`, ...). Man kann aber natürlich alle Attribute/Spalten der Tupel einzeln vergleichen.

Anfragen: TK (14)

- Der Tupelkalkül ist sehr ähnlich zur DB-Sprache SQL. Z.B. wer hat ≥ 8 Punkte für Hausaufgabe 1?

  $\{\texttt{S.Vorname},\ \texttt{S.Nachname}\ [\texttt{Student S},\ \texttt{Bewertung B}] \mid$
  $\quad \texttt{B.ANR} = 1 \wedge$
  $\quad \texttt{B.SID} = \texttt{S.SID} \wedge \texttt{B.Punkte} \geq 8\}$

- Gleiche Anfrage in SQL:

```
SELECT S.Vorname, S.Nachname
FROM   Student S, Bewertung B
WHERE  B.ANR = 1
AND    B.SID = S.SID
AND    B.Punkte >= 8
```


Anfragen: TK (15)

- Variante mit explizitem Existenzquantor:

  $\{\texttt{S.Vorname},\ \texttt{S.Nachname}\ [\texttt{Student S}] \mid$
  $\quad \exists\, \texttt{Bewertung B:}\ \texttt{B.ANR} = 1 \wedge$
  $\qquad \texttt{B.SID} = \texttt{S.SID} \wedge \texttt{B.Punkte} \geq 8\}$

- Dem entspricht in SQL eine Unteranfrage:

```
SELECT S.Vorname, S.Nachname
FROM   Student S
WHERE  EXISTS (SELECT *
               FROM   Bewertung B
               WHERE  B.ANR = 1
               AND    B.SID = S.SID
               AND    B.Punkte >= 8)
```


Boolsche Anfragen

- Eine Anfrage (Form C) ist eine geschlossene Formel F.
- Das System gibt "ja" aus, falls $\mathcal{I} \models F$ und "nein" sonst.

Übung:

- Angenommen, Form C ist nicht verfügbar. Kann sie mit Form A oder B simuliert werden?
- Offensichtlich ist Form A Spezialfall von Form B: $\{X_1, \ldots, X_n\ [s_1\ X_1, \ldots, s_n\ X_n] \mid F\}$. Ist es umgekehrt auch möglich, Form B auf Form A zurückzuführen?

Bereichsunabhängigkeit (1)

- Man kann nicht beliebige Formeln als Anfragen verwenden. Einige Formeln würden unendliche Antworten generieren:

  $\{\texttt{int SID} \mid \neg\texttt{Student(SID, 'Lisa', 'Weiss')}\}$

- Andere Formeln verlangen, daß man unendlich viele Werte für quantifizierte Variablen testet:

  $\neg\exists\ \texttt{int X, int Y, int Z, int n:}$
  $\texttt{X}^\texttt{n} + \texttt{Y}^\texttt{n} = \texttt{Z}^\texttt{n} \wedge \texttt{n} > 2 \wedge \texttt{X} \neq 0 \wedge \texttt{Y} \neq 0 \wedge \texttt{Z} \neq 0$

- In Datenbanken sollen nur Formeln zugelassen werden, bei denen es ausreicht, endlich viele Werte für jede Variable einzusetzen.

Bereichsunabhängigkeit (2)

- Für einen DB-Zustand (Interpretation) $\mathcal{I}$ und eine Formel F sei der "aktive Bereich" $\mathcal{D}_{\mathcal{I},F}[s] \subseteq \mathcal{I}[s]$ (der Sorte s) die Menge der Werte aus $\mathcal{I}[s]$, die
  - ◇ in DB-Relationen in $\mathcal{I}$,
  - ◇ als Wert einer DB-Sorte oder DB-Funktion,
  - ◇ oder als variablenfreier Term (Konstante) in F auftreten.
- Der aktive Bereich ist endlich, enthält aber für bereichsunabhängige Formeln alle relevanten Werte.

Bereichsunabhängigkeit (3)

- Eine Formel F ist bereichsunabhängig gdw. es für alle möglichen Datenbankzustände $\mathcal{I}$ gilt: Man bekommt die gleiche Antwort wenn man für die Variablen
  - ◇ beliebige Werte aus $\mathcal{I}[s]$ einsetzt, oder
  - ◇ nur Werte aus dem aktiven Bereich $\mathcal{D}_{\mathcal{I},F}[s]$.

  Genauer: (1) F muß falsch sein, wenn ein Wert außerhalb des aktiven Bereichs für eine freie Variable eingesetzt wird. (2) Für jede Teilformel $\exists X\colon G$ muß G falsch sein, wenn X einen Wert außerhalb des aktiven Bereichs annimmt. (3) Für jede Teilformel $\forall X\colon G$ muß G wahr sein, wenn X einen Wert außerhalb des aktiven Bereichs annimmt.

Bereichsunabhängigkeit(4)

- Da der aktive Bereich (im Zustand abgespeicherte Werte) endlich ist, können bereichsunabhängige Anfragen in endlicher Zeit ausgewertet werden.
- Z.B. ist folgende Formel nicht bereichsunabhängig:

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

  Der Wahrheitswert ist abhängig von anderen ganzen Zahlen (`int`-Werten) als 1, aber z.B. im leeren DB-Zustand gibt es solche nicht.

  Die exakte Menge möglicher Werte (Bereich) ist manchmal unbekannt. Es treten nur die Werte auf, die in der DB bekannt sind. Dann ist es günstig, wenn der Wahrheitswert nicht vom Bereich abhängt.

Bereichsunabhängigkeit (5)

- "Bereichsbeschränkung" ist eine syntaktische Bedingung, die Bereichsunabhängigkeit impliziert.
- Man definiert zunächst, welche Variablen in einer Formel in positivem bzw. negiertem Kontext an eine Teilmenge des aktiven Bereichs gebunden sind.

  Z.B. wenn F eine atomare Formel $p(t_1, \dots, t_n)$ mit der DB-Relation p ist, dann ist $posres(F) := \{X \in VARS \mid t_i \text{ ist die Variable } X\}$ und $negres(F) := \emptyset$. Für andere atomare Formeln, sind beide Mengen leer, außer wenn F die Form $X = t$ hat, wobei t variablenfrei ist oder eine DB-Funktion als äußerste Funktion hat. Dann gilt $posres(F) := \{X\}$. Ist F gleich $\neg G$, dann $posres(F) := negres(G)$, $negres(F) := posres(G)$. Hat F die Form $G_1 \wedge G_2$, dann $posres(F) := posres(G_1) \cup posres(G_2)$ und $negres(F) := negres(G_1) \cap negres(G_2)$. Etc.

Bereichsunabhängigkeit (6)

- Eine Formel F ist bereichsbeschränkt gdw.
  - für alle freien Variablen X von F, die nicht über endlichen DB-Sorten laufen, $X \in posres(F)$ gilt,

    Werte von endlichen DB-Sorten gehören zum aktiven Bereich. Daher braucht man für solche Variablen keine Einschränkungen. Die Menge $posres(F)$ ist so definiert, daß F garantiert falsch ist, wenn für ein $X \in posres(F)$ ein Wert außerhalb des aktiven Bereiches eingesetzt wird.
  - für jede Teilformel $\forall s X\colon G$, wobei s keine endliche DB-Sorte ist, $X \in negres(G)$ gilt, und
  - für jede Teilformel $\exists s X\colon G$ gilt, daß $X \in posres(G)$ oder s eine endliche DB-Sorte ist.

Bereichsunabhängigkeit (7)

- Ist eine Formel bereichsbeschränkt, so ist sie auch bereichsunabhängig. Das Umgekehrte gilt nicht.

  Bereichsbeschränkung stellt sicher, daß jede Variable auf einfache Weise an eine endliche Menge von Werten gebunden ist.
- Bereichsunabhängigkeit ist im allgemeinen unentscheidbar (nicht algorithmisch zu testen). Bereichsbeschränkung ist entscheidbar.

  Z.B. $\exists X\colon X \neq 1 \wedge F$ wäre bereichsunabhängig, wenn F stets falsch ist. Die Konsistenz dieser Formel (auch bereichsbeschränkter Formeln) ist unentscheidbar.

  Trotzdem ist die Bereichsbeschränkung ausreichend allgemein: Z.B. können alle Anfragen der relationalen Algebra in bereichsbeschränkte Formeln übersetzt werden.

Inhalt

Implikation

Definition:

- Eine Formel oder Menge von Formeln Φ impliziert (logisch) eine Formel oder Menge von Formeln G, gdw. jedes Modell $\langle \mathcal{I}, \mathcal{A} \rangle$ von Φ auch ein Modell von G ist. In diesem Fall schreibt man $\Phi \vdash G$.

  In der Behandlung von freien Variablen unterscheiden sich die Definitionen verschiedener Autoren. Z.B. würde $X = Y \land Y = Z \ \vdash \ X = Z$ nach obiger Definition gelten. Betrachtet man freie Variablen aber als implizit allquantifiziert, so gilt das nicht.

  Man beachte auch, daß viele Autoren $\Phi \models G$ schreiben. Der Unterschied ist wichtig, wenn man über Axiome und Deduktionsregeln spricht. Dann steht $\Phi \vdash G$ für syntaktische Deduktion, $\Phi \models G$ für durch Modelle definierte Implikation. Ein Deduktionssystem ist korrekt und vollständig, wenn beide Relationen übereinstimmen.

Äquivalenz (1)

Definition:

- Zwei (Σ, ν)-Formeln oder Mengen solcher Formeln F_1 und F_2 heißen (logisch) äquivalent gdw. sie die gleichen Modelle haben, d.h. wenn für jede Σ-Interpretation $\mathcal{I}$ und jede $(\mathcal{I}, \nu)$-Variablenbelegung $\mathcal{A}$ gilt:

  $$\langle \mathcal{I}, \mathcal{A} \rangle \models F_1 \iff \langle \mathcal{I}, \mathcal{A} \rangle \models F_2.$$

  Man schreibt dann: $F_1 \equiv F_2$.

  Wie schon beim Modellbegriff und der logischen Implikation behandeln manche Autoren freie Variablen als implizit allquantifiziert.

  Bei Datenbanken wird bezieht sich "Äquivalenz" häufig auf eine gegebene Menge von Integritätsbedingungen: Dann werden nicht beliebige Σ-Interpretationen $\mathcal{I}$ betrachtet, sondern nur solche, die die Integritätsbedingungen erfüllen.

Äquivalenz (2)

Lemma:

- F_1 und F_2 sind äquivalent gdw. $F_1 \vdash F_2$ und $F_2 \vdash F_1$.
- "Äquivalenz" von Formeln ist eine Äquivalenzrelation, d.h. sie ist reflexiv, symmetrisch und transitiv.

  Reflexiv: $F \equiv F$.
  Symmetrisch: Wenn $F \equiv G$, dann $G \equiv F$.
  Transitiv: Wenn $F_1 \equiv F_2$ und $F_2 \equiv F_3$, dann $F_1 \equiv F_3$.
- Entsteht G_1 aus G_2 durch Ersetzen der Teilformel F_1 durch F_2, und gilt $F_1 \equiv F_2$, so gilt auch $G_1 \equiv G_2$.
- Wenn $F \vdash G$, dann $F \land G \equiv F$.

Einige Äquivalenzen (1)

- Kommutativität (für und, oder, gdw):
  - ◇ $F \land G \equiv G \land F$
  - ◇ $F \lor G \equiv G \lor F$
  - ◇ $F \leftrightarrow G \equiv G \leftrightarrow F$
- Assoziativität (für und, oder, gdw):
  - ◇ $F_1 \land (F_2 \land F_3) \equiv (F_1 \land F_2) \land F_3$
  - ◇ $F_1 \lor (F_2 \lor F_3) \equiv (F_1 \lor F_2) \lor F_3$
  - ◇ $F_1 \leftrightarrow (F_2 \leftrightarrow F_3) \equiv (F_1 \leftrightarrow F_2) \leftrightarrow F_3$

Einige Äquivalenzen (2)

- Distributivgesetz:
  - ◇ $F \wedge (G_1 \vee G_2) \equiv (F \wedge G_1) \vee (F \wedge G_2)$
  - ◇ $F \vee (G_1 \wedge G_2) \equiv (F \vee G_1) \wedge (F \vee G_2)$
- Doppelte Negation:
  - ◇ $\neg(\neg F) \equiv F$
- De Morgan'sche Regeln:
  - ◇ $\neg(F \wedge G) \equiv (\neg F) \vee (\neg G)$.
  - ◇ $\neg(F \vee G) \equiv (\neg F) \wedge (\neg G)$.

Einige Äquivalenzen (3)

- Ersetzung des Implikationsoperators:
  - ◇ $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (F \leftarrow G)$
  - ◇ $F \leftarrow G \equiv G \rightarrow F$
  - ◇ $F \rightarrow G \equiv \neg F \vee G$
  - ◇ $F \leftarrow G \equiv F \vee \neg G$
- Zusammen mit den De Morgan'schen Regeln bedeutet dies, daß z.B. $\{\neg, \vee\}$ ausreichend sind, weil die anderen logischen Junktoren $\{\wedge, \leftarrow, \rightarrow, \leftrightarrow\}$ durch diese ausgedrückt werden können.

  Wir werden sehen, dass auch nur einer der Quantoren benötigt wird.

Einige Äquivalenzen (4)

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

  Zusammen mit dem De'Morganschen Gesetz kann man die Negation bis zu den atomaren Formeln herunterschieben, und dann durch Umdrehen der Vergleichsoperatoren eliminieren.
  Das geht auch mit Quantoren, aber man braucht dann $\exists$ und $\forall$. Da es in SQL nur $\exists$ gibt, kann man dort $\neg$ vor $\exists$ nicht entfernen.

Einige Äquivalenzen (5)

- Prinzip des ausgeschlossenen Dritten:
  - ◇ $F \vee \neg F \equiv \top$ (immer wahr)
  - ◇ $F \wedge \neg F \equiv \bot$ (immer falsch)
- Vereinfachung von Formeln mit den logischen Konstanten $\top$ (wahr) und $\bot$ (falsch):
  - ◇ $F \wedge \top \equiv F$ $\qquad$ $F \wedge \bot \equiv \bot$
  - ◇ $F \vee \top \equiv \top$ $\qquad$ $F \vee \bot \equiv F$
  - ◇ $\neg\top \equiv \bot$ $\qquad$ $\neg\bot \equiv \top$

Einige Äquivalenzen (6)

- Ersetzung von Quantoren:
  - ◇ $\forall\, s\, X\colon F \equiv \neg(\exists\, s\, X\colon (\neg F))$
  - ◇ $\exists\, s\, X\colon F \equiv \neg(\forall\, s\, X\colon (\neg F))$
- Logische Junktoren über Quantoren bewegen:
  - ◇ $\neg(\forall\, s\, X\colon F) \equiv \exists\, s\, X\colon (\neg F)$
  - ◇ $\neg(\exists\, s\, X\colon F) \equiv \forall\, s\, X\colon (\neg F)$
  - ◇ $\forall\, s\, X\colon (F \wedge G) \equiv (\forall\, s\, X\colon F) \wedge (\forall\, s\, X\colon G)$
  - ◇ $\exists\, s\, X\colon (F \vee G) \equiv (\exists\, s\, X\colon F) \vee (\exists\, s\, X\colon G)$

Einige Äquivalenzen (7)

- Quantoren bewegen: Sei $X \notin \mathit{free}(F)$:
  - ◇ $\forall\, s\, X\colon (F \vee G) \equiv F \vee (\forall\, s\, X\colon G)$
  - ◇ $\exists\, s\, X\colon (F \wedge G) \equiv F \wedge (\exists\, s\, X\colon G)$

  Falls zusätzlich $\mathcal{I}[s]$ nicht leer sein kann:
  - ◇ $\forall\, s\, X\colon (F \wedge G) \equiv F \wedge (\forall\, s\, X\colon G)$
  - ◇ $\exists\, s\, X\colon (F \vee G) \equiv F \vee (\exists\, s\, X\colon G)$
- Eliminierung überflüssiger Quantoren:
  Ist $X \notin \mathit{free}(F)$ und kann $\mathcal{I}[s]$ nicht leer sein:
  - ◇ $\forall\, s\, X\colon F \equiv F$
  - ◇ $\exists\, s\, X\colon F \equiv F$

Einige Äquivalenzen (8)

- Vertauschung von Quantoren: Ist $X \not\equiv Y$:
  - ◇ $\forall\, s_1\, X\colon (\forall\, s_2\, Y\colon F) \equiv \forall\, s_2\, Y\colon (\forall\, s_1\, X\colon F)$
  - ◇ $\exists\, s_1\, X\colon (\exists\, s_2\, Y\colon F) \equiv \exists\, s_2\, Y\colon (\exists\, s_1\, X\colon F)$

    Beachte, dass Quantoren verschiedenen Typs ($\forall$ und $\exists$) nicht vertauscht werden können.
- Umbenennung gebundener Variablen:
  Ist $Y \notin \mathit{free}(F)$ und F' entsteht aus F durch Ersetzen jedes freien Vorkommens von X in F durch Y:
  - ◇ $\forall\, s\, X\colon F \equiv \forall\, s\, Y\colon F'$
  - ◇ $\exists\, s\, X\colon F \equiv \exists\, s\, Y\colon F'$

Einige Äquivalenzen (9)

- Gleichheit ist Äquivalenzrelation:
  - ◇ $t = t \equiv \top$ (Reflexivität)
  - ◇ $t_1 = t_2 \equiv t_2 = t_1$ (Symmetrie)
  - ◇ $t_1 = t_2 \wedge t_2 = t_3 \equiv t_1 = t_2 \wedge t_2 = t_3 \wedge t_1 = t_3$ (Transitivität)
- Verträglichkeit mit Funktions-/Prädikatsymbolen:
  - ◇ $f(t_1, \ldots, t_n) = t \wedge t_i = t_i' \equiv$
    $f(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n) = t \wedge t_i = t_i'$
  - ◇ $p(t_1, \ldots, t_n) \wedge t_i = t_i' \equiv$
    $p(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n) \wedge t_i = t_i'$

Normalformen (1)

Definition:

- Eine Formel F ist in Pränex-Normalform gdw. sie geschlossen ist und die folgende Form hat:

$$\Theta_1\, s_1\, X_1\ \dots\ \Theta_n\, s_n\, X_n\colon\ G,$$

wobei $\Theta_1, \dots, \Theta_n \in \{\forall, \exists\}$ und G quantor-frei sind.

- Eine Formel F ist in disjunktiver Normalform gdw. sie in Pränex-Normalform ist, und G die Form hat

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

wobei jedes $G_{i,j}$ eine atomare Formel oder eine negierte atomare Formel ist.

Normalformen (2)

Bemerkung:

- Konjunktive Normalform entspricht disjunktiver NF, aber G hat die folgende Form:

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

Theorem:

- Kann man nichtleere Bereiche $\mathcal{I}[s]$ voraussetzen, so kann jede Formel äquivalent in Pränex-Normalform, disjunktive Normalform, und konjunktive Normalform transformiert werden.

Bedeutung für DBen (1)

- Für die Korrektheit einer Anfrage ist nicht wichtig, welche von mehreren äquivalenten Formulierungen man wählt: Die Antwort ist immer gleich.
- Die Anfragen unterscheiden sich aber eventuell in der Lesbarkeit: Natürlich sollte man eine möglichst einfache Formulierung wählen.

  Es können in der Klausur Punkte für unnötige Verkomplizierungen abgezogen werden, obwohl die Anfrage das richtige Ergebnis liefert.

- Unnötige Verkomplizierungen können auch zu einer weniger effizienten Auswertung führen.

Bedeutung für DBen (2)

- Über die Auswahl zwischen gleich komplizierten Bedingungen wie z.B. $F \wedge G$ und $G \wedge F$ braucht man sich aber keine Gedanken mehr zu machen: Die Anfragen sollten gleich schnell ausgewertet werden.

  Optimierer in heutigen DBMS kennen solche einfachen Äquivalenzen.

- Dagegen werden kompliziertere Äquivalenzen, wie etwa, daß `AUFGNR - 2 = 0` das gleiche wie `AUFGNR = 2` bedeutet, vom Optimierer nicht unbedingt erkannt.

  Während der Optimierer bei `AUFGNR = 2` eventuell einen Index benutzt, um schnell auf die passenden Tabellenzeilen zuzugreifen, wird er bei `AUFGNR - 2 = 0` vermutlich die Tabelle komplett lesen.

Bedeutung für DBen (3)

- In der Sprache SQL gibt es nur $\wedge$, $\vee$, $\neg$ und $\exists$.
- Durch Anwendung der obigen Äquivalenzen kann man ggf. andere Junktoren ($\leftarrow$, $\rightarrow$, $\leftrightarrow$) sowie den Allquantor $\forall$ aus der Anfrage-Bedingung entfernen.

  Diese erweiterten logischen Operatoren sind aber nützlich, weil sie sich manchmal direkt aus der natürlichsprachlichen Formulierung der Anfrage ergeben. Auch bei Integritätsbedingungen sind "wenn-dann" Regeln $F \rightarrow G$ nicht selten: Man muß die Äquivalenz zu $\neg F \vee G$ kennen, um sie systematisch nach SQL übersetzen zu können.

Bedeutung für DBen (4)

Übung:

- Was bedeutet die folgende Anfrage?

  $\{\texttt{S.Vorname, S.Nachname [Student S, Bewertung B]} \mid$
  $\quad \texttt{S.SID} = \texttt{B.SID} \wedge \texttt{B.ANR} = 1 \wedge$
  $\quad \forall\, \texttt{Bewertung X: X.ANR} = 1 \rightarrow$
  $\qquad \texttt{X.Punkte} \leq \texttt{B.Punkte}\}$

- Eliminieren Sie $\forall$ und $\rightarrow$ durch Anwendung der obigen Äquivalenzen.

Bedeutung für DBen (5)

Übung:

- Sind die beiden folgenden Anfragen äquivalent?

  $\{\texttt{S.Vorname, S.Nachname [Student S]} \mid$
  $\quad \exists\, \texttt{Bewertung B1, Bewertung B2:}$
  $\qquad \texttt{B1.SID} = \texttt{S.SID} \wedge \texttt{B2.SID} = \texttt{S.SID} \wedge$
  $\qquad \texttt{B1.ANR} = 1 \wedge \texttt{B2.ANR} = 2 \wedge$
  $\qquad \texttt{B1.Punkte} \leq \texttt{B2.Punkte}\}$

  $\{\texttt{S.Vorname, S.Nachname [Student S]} \mid$
  $\quad \exists\, \texttt{Bewertung X: X.ANR} = 1 \wedge \texttt{X.SID} = \texttt{S.SID} \wedge$
  $\qquad \exists\, \texttt{Bewertung Y: Y.ANR} = 2 \wedge \texttt{Y.SID} = \texttt{X.SID} \wedge$
  $\qquad\quad \neg(\texttt{X.Punkte} > \texttt{Y.Punkte})\}$

Bedeutung für DBen (6)

- Es ist überflüssig (und schlecht) Integritätsbedingungen zu fordern, die von anderen Integritätsbedingungen logisch impliziert werden.
- Beispiel:
  - ◇ $\forall\, \texttt{Student S: S.SID} > 100$
  - ◇ $\forall\, \texttt{Bewertung B: } \exists\, \texttt{Student S: S.SID} = \texttt{B.SID}$
  - ◇ $\forall\, \texttt{Bewertung B: B.SID} > 100$

  Die dritte Bedingung folgt aus den ersten beiden.

Bedeutung für DBen (7)

- Bei Datenbank-Anfragen beziehen sich die Begriffe "äquivalent" und "konsistent" nicht auf beliebige Interpretationen, sondern nur auf Datenbank-Zustände, d.h. Interpretationen, die
  - die gegebene Interpretation $\mathcal{I}_\mathcal{D}$ der Datentypen beinhalten, und

    Nur deshalb ist z.B. `AUFGNR - 2 = 0` äquivalent zu `AUFGNR = 2`. Bei der reinen logischen Äquivalenz wird keine bestimmte Bedeutung des Symbols "-" vorausgesetzt. Entsprechend wäre `1 = 0` rein logisch nicht inkonsistent: Man muß erst wissen, daß `1` und `0` wirklich für verschiedene Werte stehen.
  - die Integritätsbedingungen erfüllen.

Bedeutung für DBen (8)

- Man kann es auch so definieren:
  - Zwei Anfragen heißen äquivalent, wenn sie für jeden Datenbank-Zustand jeweils die gleiche Antwort liefern.
  - Eine Anfrage heißt konsistent, wenn sie in mindestens einem Datenbank-Zustand eine nichtleere Antwortmenge hat.
- Wenn die Anfrage-Bedingungen logisch äquivalent sind, sind die Anfragen natürlich äquivalent.

Inhalt

Motivation

- Funktionen sind nicht immer definiert, z.B.
  - Division durch 0,
  - Wurzel einer negativen Zahl,
  - arithmetischer Überlauf.
- Auch Tabellenspalten bzw. Attribute von Objekten haben öfters keinen Wert: Z.B.
  - hat nicht jeder Kunde ein Fax,
  - und nicht jeder verrät sein Geburtsdatum.
- Daher sind partielle Funktionen praktisch relevant.

Interpretation

- Formal wird ein Funktionssymbol $f(s_1, \ldots, s_n)\colon s$ nun interpretiert als Funktion

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

wobei $null$ ein neuer Wert ist (verschieden von allen Elementen von $\mathcal{I}[s]$).

- Der Null-Wert wird bei der Termauswertung einfach "hochgereicht": Hat eine Funktion "$null$" als Eingabe, liefert sie automatisch "$null$" als Ausgabe.

Z.B. ist $1 + null = null$. Es ist allerdings auch $0 * null = null$, obwohl man darüber diskutieren könnte, ob hier 0 herauskommen sollte.

Beispiel (1)

- Angenommen, es wird auch das Semester der Studenten erfasst, aber nicht alle haben es angegeben:

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

- Es sei nun folgende Anfrage betrachtet:

$$\{\texttt{S.Vorname},\ \texttt{S.Nachname}\ [\texttt{Student S}] \mid \texttt{S.semester} \le 3\}$$

Beispiel (2)

- Zumindest nach der SQL Semantik kommt Daniel Sommer dieser Anfrage nicht heraus.

Eigentlich wäre es richtiger, neben den normalen, sicheren Antworten noch mögliche Antworten auszugeben. In Wirklichkeit gibt es ja ein Semester, in dem Daniel Sommer ist, wir wissen es nur nicht (Existenzaussage). Es ist ein wichtiges Problem von Nullwerten in SQL, daß der gleiche Nullwert in verschiedenen Bedeutungen verwendet wird. Wenn es z.B. eine Spalte für die Mobilfunknummer wäre, könnte es sein, daß Daniel Sommer kein Handy hat. Dann würde die Existenzaussage nicht mehr gelten. Die Intuition in SQL ist, daß Tabellenzeilen mit fehlenden Einträgen in abgefragten Spalten das Anfrageergebnis nicht beeinflussen sollten. Man kann sich den Zugriff so eine Spalte wie einen Fehler vorstellen. Allerdings liefert SQL bei richtigen Fehlern (z.B. Division durch 0) eine Fehlermeldung, und nicht den Nullwert (Problem für Deklarativität/Optimierung).

Beispiel (3)

- Daniel Sommer kommt auch nicht heraus, wenn nach Studenten in höheren Semestern gefragt wird:

$$\{\texttt{S.Vorname},\ \texttt{S.Nachname}\ [\texttt{Student S}] \mid \texttt{S.semester} > 3\}$$

- Diese Anfrage ist (auch in SQL) äquivalent zu folgender Anfrage:

$$\{\texttt{S.Vorname},\ \texttt{S.Nachname}\ [\texttt{Student S}] \mid \neg(\texttt{S.semester} \le 3)\}$$

Beispiel (4)

- Daniel Sommer würde sogar hier nicht ausgedruckt:

  {S.Vorname, S.Nachname [Student S] |
  S.semester $\leq 3 \vee \neg($S.semester $\leq 3)$}

- Damit ist das Prinzip des ausgeschlossenen Dritten verletzt.
- Eine zweiwertige Logik, nur mit den Werten "wahr" und "falsch" kann das nicht mehr leisten.
- Man braucht einen dritten Wahrheitswert "undefiniert" (oder "null").

Wahrheit einer Formel (1)

- Ist F eine atomare Formel $p(t_1, \ldots, t_n)$ oder $t_1 = t_2$, und wird einer der Argumentterme t_i zu $null$ ausgewertet, so liefert die Formel den dritten Wahrheitswert **u**.
- Hat F die Form $\neg G$, so ergibt sich der Wahrheitswert von F aus dem von G nach folgender Tabelle:

G	$\neg G$
f	**w**
u	**u**
w	**f**

Wahrheit einer Formel (2)

- Die zweistelligen logischen Verknüpfungen arbeiten nach folgender Tabelle:

G_1	G_2	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
f	**f**	**f**	**f**	**w**	**w**	**w**
f	**u**	**f**	**u**	**u**	**w**	**u**
f	**w**	**f**	**w**	**f**	**w**	**f**
u	**f**	**f**	**u**	**w**	**u**	**u**
u	**u**	**u**	**u**	**u**	**u**	**u**
u	**w**	**u**	**w**	**u**	**w**	**u**
w	**f**	**f**	**w**	**w**	**f**	**f**
w	**u**	**u**	**w**	**w**	**u**	**u**
w	**w**	**w**	**w**	**w**	**w**	**w**

Wahrheit einer Formel (3)

- Man braucht die obige Tabelle nicht auswendig zu lernen: Das Prinzip ist einfach, daß der Wahrheitswert **u** weitergegeben wird, sofern der Wert der Formel nicht schon durch durch den andern Eingabewert festliegt.
- Z.B. ist $\mathbf{u} \wedge \mathbf{f} = \mathbf{f}$, weil es keine Rolle spielt, ob der linke Eingabewert vielleicht **w** oder **f** ist.
- Mit anderen Worten: Eine Teilbedingung, die zu **u** ausgewertet wird, sollte den Gesamt-Wahrheitswert möglichst nicht beeinflussen.

Wahrheit einer Formel (4)

- Eine Existenzaussage $\exists\, s\; X\colon G$ ist in $\langle \mathcal{I}, \mathcal{A}\rangle$ wahr, wenn es einen Wert $d \in \mathcal{I}[s]$ gibt, so daß

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

- Ansonsten ist sie nach der SQL-Semantik falsch.

  D.h. sie ist niemals undefiniert, auch dann nicht, wenn die quantifizierte Formel für alle getesteten Variablenbelegungen undefiniert ist. Ein Existenzquantor entspricht damit nicht mehr einer großen Disjunktion. Eine Disjunktion von lauter undefinierten Teilbedingungen wäre ja selbst undefiniert.

  Allerdings gibt es in SQL auch ein Konstrukt `= ANY` etc., bei dem die Disjunktionssemantik verwendet wird (etwas verwirrend).

- Der Nullwert wird für X nicht eingesetzt: $null \notin \mathcal{I}[s]$.

Wahrheit einer Formel (5)

- Entsprechend ist eine Allaussage $\forall\, s\; X\colon G$ in $\langle \mathcal{I}, \mathcal{A}\rangle$ wahr gdw. für alle $d \in \mathcal{I}[s]$:
  - ◇ $\langle \mathcal{I}, \mathcal{A}\langle X/d\rangle\rangle[G] = \mathbf{w}$ oder
  - ◇ $\langle \mathcal{I}, \mathcal{A}\langle X/d\rangle\rangle[G] = \mathbf{u}$.
- Sie ist nur dann falsch, wenn es eine Variablenbelegung $\mathcal{A}'$ gibt, die sich nur im Wert für X von $\mathcal{A}$ unterscheidet, für die $\langle \mathcal{I}, \mathcal{A}'\rangle[G] = \mathbf{f}$ ist.
- Somit gilt wieder: $\forall\, s\; X\colon G \;\equiv\; \neg\exists\, s\; X\; \neg G$.

Äquivalenzen

- Man beachte, daß einige aus der zweiwertigen Logik bekannte Sachverhalte hier nicht mehr gelten:
  - ◇ $t = t$ ist keine Tautologie: Falls t den Nullwert liefert, ist der Wert der Gleichung undefiniert (**u**).
  - ◇ $F \vee \neg F$ (Prinzip des ausgeschlossenen Dritten).
  - ◇ Alle Äquivalenzen mit Quantoren, die voraussetzen, daß $\mathcal{I}[s]$ nicht leer ist.

    Z.B. ist $\forall s\, X\colon G$, wobei X in G nicht vorkommt, und $\mathcal{I}[s]$ nicht leer ist, nicht unbedingt äquivalent zu G: Falls G den Wahrheitswert **u** liefert, ist die Allaussage wahr (**w**).

Test auf Null

- Man braucht nun eine weitere Form von atomarer Formel, um testen zu können, ob ein Term den Nullwert liefert.
- t `is null` ist wahr (**w**) in $\langle \mathcal{I}, \mathcal{A}\rangle$ gdw. $\langle \mathcal{I}, \mathcal{A}\rangle[t] = null$, und falsch (**f**) sonst.

  Diese atomare Formel liefert niemals den Wert **u**.

- Um die Lesbarkeit zu erhöhen, erlaubt man t `is not null` als alternative Syntax für $\neg$(t `is null`).

Konstante mit Wert Null

- Man kann auch eine neue Art von Term einführen: "`null`" wird immer zu *null* ausgewertet.

  Der SQL Standard hat keinen solchen Term, aber Oracle schon. An manchen Stellen braucht man die Möglichkeit, einen Nullwert explizit anzugeben. Diese muß der SQL Standard dann gesondert behandeln.

  Ein Term, der beliebigen Typ haben kann, führt zusammen mit überladenen Operatoren zu Mehrdeutigkeiten. Korrekter wäre es also, den Typ immer explizit anzugeben, z.B. in der Syntax "(s) `null`".

- Man beachte, daß "$t =$ `null`" immer den Wahrheitswert **u** liefert, als Test auf den Nullwert also nicht geeignet ist.

  Dies ist ein häufiger Fehler in Oracle. Die Regel ist, daß jeder Vergleich **u** liefert, wenn einer oder beide Operanden ein Nullwert sind.

Totale Funktionen

- Da nun alle Funktionen grundsätzlich partiell sind, muß man explizit mit einer Integritätsbedingung fordern, daß bestimmte Funktionen für alle Eingabewerte definiert sind.
- Z.B. soll der Nachname aller Studierenden bekannt sein:

  $\forall$ `Student S: S.Nachname is not null.`

- Weil diese Art von Integritätsbedingungen so häufig ist, wird man eine Abküzung dafür einführen.

  In SQL fügt man der Deklaration der Tabellenspalte "`NOT NULL`" hinzu.

Relationship-Attribute

- Mit partiellen Funktionen kann man im ER-Modell die Relationship-Attribute besser behandeln.

  Im Beispiel sei das Attribut `Punkte` des Relationships `hat_abgegeben` zwischen den Entity-Typen `Student` und `Aufgabe` betrachtet.

- Die Funktion, die so einem Attribut entspricht, ist genau dann definiert, wenn die Beziehung besteht:

  $\forall$ `Student S, Aufgabe A :`
  `hat_abgegeben(S, A)` $\leftrightarrow$ `Punkte(S, A) is not null.`

- Falls das Attribut Nullwerte erlauben soll:

  $\forall$ `Student S, Aufgabe A :`
  $\neg$`hat_abgegeben(S, A)` $\rightarrow$ `Punkte(S, A) is null.`

Anmerkung: Bereichskalkül

- Die so definierte Logik erlaubt es, Nullwerte im Tupelkalkül (und im ER-Modell) zu behandeln, mit einer Semantik, die SQL entspricht.
- Man kann die Logik auch so definieren, daß auch ein Bereichskalkül mit Nullwerten unterstützt wird.

  Prädikate müssten dann auf den Nullwert als Eingabe auch explizit mit wahr (**w**) oder falsch (**f**) reagieren können. Im Moment liefern sie automatisch "undefiniert" (**u**). Außerdem müssen quantifizierte Variablen auch den Nullwert annehmen. Das will man zwar eigentlich nicht immer, aber da ein Nullwert meistens nicht schadet (normalerweise wird der Wahrheitswert des Quantors nicht beeinflusst), ist es wohl das kleinere Übel im Vergleich zur Alternative, bei jedem Quantor eine explizite Festlegung treffen zu müssen.