Einführung in Datenbanken

Kapitel 6: Logische Grundlagen für Datentypen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2020/21

http://www.informatik.uni-halle.de/~brass/db20/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Die These akzeptieren/verteidigen (oder diskutieren), dass man Datentypen formal definieren kann.

  Dieses Kapitel gibt den formalen Rahmen dafür, nicht einzelne Definitionen konkreter Datentypen. Datenbanken benötigen Datentypen als Grundlage.

- Die Komponenten einer Signatur benennen.

  Wenn Sie eine extra Komponente für Konstanten vorsehen wollen, können Sie das tun, obwohl das in dieser Vorlesung nicht so gemacht wird.
  Sie können auch Argument- und Ergebnis-Typen anders formalisieren, z.B. ohne überladene Funktionen. Es sollte allerdings eine Definition für mehrsortige Logik sein (d.h. mit Datentypen). Die Konzepte/Begriffe sind wichtig, nicht die formalen Details der Definition (zumindest hier).

- Angeben, worauf eine Interpretation die Komponenten einer Signatur abbildet.

Inhalt

Logik: Motivation (1)

- Der Kern von SQL kann als leichte syntaktische Variante der Prädikatenlogik erster Stufe gesehen werden.

  Vermutlich haben viele Teilnehmer dieser Vorlesung zumindest ein wenig Logik in früheren Vorlesungen gehabt.

- Allgemein geht es in mathematischer Logik wie in Datenbanken darum,
  - Wissen zu formalisieren, und
  - mit diesem Wissen zu arbeiten.
- Ein Zugang zu SQL kann es sein, es als Spezialfall bzw. praktische Anwendung der Logik zu erklären.

  Mir ist bewusst, das verschiedene Teilnehmer der Vorlesung verschiedene Zugänge benötigen. Z.B. kann man eine Sprache auch über viele Beispiele und die Syntaxdiagramme lernen.

Logik: Motivation (2)

- Vorteile der Logik:
  - Die Konzepte der Logik sind präzise mathematisch definiert.
  - In SQL gibt es häufig viele syntaktische Varianten, das Gleiche auszudrücken.
    Logik erlaubt die Konzentration auf das Wesentliche.
  - Logik kann ein allgemeiner Rahmen sein, um auch andere Datenmodelle und Datenbank-Sprachen zu verstehen.
  - Wenn zwei `WHERE`-Bedingungen logisch äquivalent sind, spielt es keine große Rolle, welche man wählt.
  - Begriffe aus der Logik wie Inkonsistenz („immer falsch“) sind nützlich, um Fehler in SQL-Anfragen zu verstehen.
- In der Klausur wird Logik fast nur in SQL-Syntax verlangt.

  Klassische logische Formeln vielleicht als Integritätsbedingungen.

Inhalt

Alphabet (1)

Definition:

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

  Die Elemente von *ALPH* sind die Wortsymbole der lexikalischen Syntax, z.B. ist jeder Bezeichner oder jede Zahlkonstante ein Element von *ALPH*.

- *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. $VARS \cap LOG = \emptyset$).

  Die Menge der Variablen muss unendlich sein, damit es immer noch eine weitere Variable gibt, zusätzlich zu denen, die in einer gegebenen Formel vorkommen.

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. Damit die Definitionen leicht auf SQL übertragbar sind, müssen auch die Namen von parametrisierte Datentypen wie „`NUMERIC(2)`“ als ein Element von *ALPH* gesehen werden.
  - Operator-Symbolen wie `+`, `<`,
  - Datentyp-Literalen (Konstanten) wie `123`, `'abc'`.

Alphabet (3)

- Man beachte, dass Wörter wie „STUDENTEN“ 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 7 (Beispiele für Alternativen siehe Folie 10).

Alphabet (4)

Mögliche Alternativen für logische Symbole:

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

In SQL wird der Existenzquantor mit einer Unteranfrage ausgedrückt:

`EXISTS(SELECT * FROM ... WHERE ...)`.

Worte über einem Alphabet

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

  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

  $$\forall\, \texttt{INT}\, \texttt{X}\colon\ \neg(\texttt{X} < \texttt{X})$$

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

  Es wird hier eine mehrsortige Logik verwendet, also eine Logik mit Datentypen. Deswegen ist hier angegeben, dass die Variable `X` über den ganzen Zahlen läuft. In SQL laufen Variablen über den Zeilen einer Tabelle also nur einem endlichen Bereich (siehe Kapitel 8).

Inhalt

Signaturen (1)

Definition:

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

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

Signaturen (2)

- Sorten sind Datentyp-Namen, z.B. `INT`, `NUMERIC(2)`.

  Später auch für Zeilen von Tabellen (Tupel-Typen), z.B. `STUDENTEN`.

- Prädikate liefern für gegebene Eingabewerte wahr oder falsch, z.B. ist $<$ ein Prädikat.

  SQL hat auch ein Prädikat `LIKE` für einfache Mustervergleiche und `SIMILAR TO` für den Mustervergleich mit regulären Ausdrücken.

- Ist $p \in \mathcal{P}_\alpha$, und $\alpha = s_1 \ldots s_n$, dann werden $s_1, \ldots, s_n$ die Argumentsorten von p genannt.

  s_1 ist der Typ des ersten Arguments, s_2 der des zweiten, usw.

- Die Anzahl der Argumentsorten (Länge von α) nennt man Stelligkeit des Prädikatsymbols, z.B. ist $<$ ein Prädikatsymbol der Stelligkeit 2.

Signaturen (3)

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

  Die Möglichkeit überladener Prädikate ist in der Theorie 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 und sind z.B. auch in Programmiersprachen üblich.

Signaturen (4)

- Eine Funktion liefert für gegebene Eingabewerte einen Ausgabewert. Beispiele: `+`, `-`, `*`, `/`, `SIN`, `SQRT`.

  Es sei hier angenommen, dass Funktionen für alle Eingabewerte definiert sind. Das muss nicht sein, z.B. ist 5/0 ist nicht definiert. Man könnte das mit einer mehrwertigen Logik behandeln (wahr, falsch, Fehler).

- Ein Funktionssymbol in $\mathcal{F}_{\alpha,s}$ hat die Argumentsorten α und die Ergebnissorte s, z.B. $+ \in \mathcal{F}_{\texttt{INT}\,\texttt{INT},\,\texttt{INT}}$.

- Informatiker würden eher schreiben:

  `+(INT, INT): INT`.

  Die exakte Notation ist nicht besonders wichtig. Das Konzept eines Funktionssymbols, von Argumentsorten (oder „Datentypen der Argumente“) und Ergebnissorte schon. Wenn man (wie hier) das Überladen braucht, bietet sich die mathematische Formalisierung als indizierte Menge an. Für die meisten α, s ist $\mathcal{F}_{\alpha,s}$ allerdings leer.

Signaturen (5)

- Eine Funktion ohne Argumente heißt Konstante.

  Das ist nur ein technischer Trick, um eine weitere Komponente der Signatur zu vermeiden.

- Beispiele für Konstanten:

  - $1 \in \mathcal{F}_{\epsilon,\texttt{INT}}$ `1: INT`.
  - `'Lisa'` $\in \mathcal{F}_{\epsilon,\texttt{string}}$ `'Lisa': string`.

- Bei Datentypen (z.B. `INT`, `VARCHAR(10)`) kann normalerweise jeder mögliche Wert durch eine Konstante bezeichnet werden.

  Im Allgemeinen sind die Menge der Werte und die der Konstanten dagegen verschieden. Fließkommazahlen haben einige Spezial-/Fehlerwerte (z.B. NaN: „not a number“), die oft nicht durch Konstanten bezeichnet werden können.

Signaturen (6)

- Natürlich kann man eine unendliche Konstantenmenge nicht durch Aufzählung definieren.
- Mathematisch ist das kein Problem, $\mathcal{F}_{\epsilon,\texttt{INT}}$ 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.): $\mathcal{F}_{\epsilon,\texttt{INT}}$ ist durch ein Programm definiert.
- Die letztendlich in SQL verwendete Signatur hat zwei Teile:
  - Die vom DBMS vorgegebene Signatur der Datentypen.
  - Anwendungsspezifische Symbole, die über das DB-Schema eingeführt werden (eine Sorte für die Tupel jeder Tabelle).

Signaturen (7)

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.

  Die Signatur, die letztendlich für Anfragen und Integritätsbedingungen verwendet wird, ist eine Erweiterung der durch das DBMS vorgebenen Datentyp-Signatur.

Inhalt

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 wird teilweise $\mathcal{I}(p)$ statt $\mathcal{I}(p, \alpha)$ geschrieben, wenn α für die gegebene Signatur Σ eindeutig ist (d.h. p nicht überladen ist).
  - eine Funktion $\mathcal{I}(f, \alpha)\colon\ \mathcal{I}(s_1) \times \cdots \times \mathcal{I}(s_n) \to \mathcal{I}(s)$ 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 Studenten (Tupel der Tabelle `STUDENTEN`), wenn die Datenbank gerade erst erstellt wurde. Auch bei SQL kann es Überraschungen geben, wenn eine Tupelvariable über einer leeren Relation deklariert ist.

- Die Wertebereiche der Datentypen wie `INT` sind nicht leer.
- 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 $\times$.
- Z.B. ist $\mathtt{X} < \mathtt{Y}$ genau dann wahr in $\mathcal{I}$, wenn $(\mathtt{X}, \mathtt{Y}) \in \mathcal{I}[<]$.

  Im Folgenden werden die Wörter „Prädikatsymbol“ und „Relationssymbol“ austauschbar verwendet.

Inhalt

Datenbanken und Logik (1)

- Das DBMS definiert eine Datentyp-Signatur $\Sigma_{\mathcal{D}}$ und eine zugehörige Interpretation $\mathcal{I}_{\mathcal{D}}$, die Folgendes festlegen:
  - 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.
  - Namen für Datentyp-Funktionen (wie `+`, `SIN`) mit Argument- und Ergebnissorten, interpretiert durch entsprechende Funktionen auf den Wertebereichen.
  - Namen für Datentyp-Prädikate (wie `<`, `LIKE`), interpretiert durch entsprechende Relationen auf den Wertebereichen.

    Statt Relation alternativ auch Funktion, die einen Wahrheitswert liefert.

Datenbanken und Logik (2)

- Ein Datenmodell (wie z.B. das relationale Modell) wird normalerweise (so wie in Kapitel 3) 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.
- Es ist hilfreich, die durch ein DB-Schema gegebene Signatur zu definieren, und die Teilmenge der Interpretationen, die als DB-Zustände zulässig sind.

Datenbanken und Logik (3)

- 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).
- Logik ist auch für andere Datenmodelle nützlich (z.B. ER-Modell).