

Übung zur Vorlesung
„Einsatz und Realisierung von Datenbanksystemen“
im Sommersemester 2007

Richard Kuntschke (richard.kuntschke@in.tum.de)

Lösungen zu Blatt 6

Aufgabe 1

Die nachfolgend gezeigten Beispiele sind in Oracle-Syntax angegeben. Für jeden Objekttyp werden repräsentativ ausgewählte Methodendefinitionen gezeigt – die vollständige Definition der Funktionalität sei den Lesern überlassen.

Zuerst definieren wir den Objekttyp für Punkte: *PunkteTyp* definiert drei Methoden, nämlich *verschieben*, *rotieren* und *skalieren*. Der lesende Zugriff auf die Objektinstanz **self** ist in Oracle stets möglich. Greift man aber auch schreibend darauf zu, so muss dies explizit durch **self in out** *<Objekttyp>* angegeben werden.

```
create or replace type PunkteTyp as object(
  x float,
  y float,
  z float,
  member function verschieben (self in out PunkteTyp,
    delta in PunkteTyp) return number,
  member function rotieren (self in out PunkteTyp,
    winkel in PunkteTyp) return number,
  member function skalieren (self in out PunkteTyp,
    faktor in float) return number
);

create or replace type body PunkteTyp as
  member function verschieben (self in out PunkteTyp,
    delta PunkteTyp) return number is
  begin
    x := x + delta.x;
    y := y + delta.y;
    z := z + delta.z;
    return ODCIConst.Success;
  end verschieben;

  member function rotieren (self in out PunkteTyp,
    winkel PunkteTyp) return number is
  begin
    -- winkel ist 3-dimensional angegeben, da eine
    -- Rotation in drei Ebenen möglich ist
```

```

    return ODCIConst.Success;
end rotieren;

member function skalieren (self in out PunkteTyp,
    faktor float) return number is
begin
    -- Punkte werden in unserem Beispiel nicht skaliert
    return ODCIConst.Success;
end skalieren;
end;

```

Als Nächstes definieren wir den Objekttyp *KantenTyp*. Eine Kante ist durch einen Start- und einen Endpunkt beschrieben. Da Punkte diese Rolle von unterschiedlichen Kanten übernehmen können, sind sie somit nicht existenzabhängig von einer speziellen Kante. Aus diesem Grund sind die Attribute für Start- und Endpunkt als Referenzen auf *PunkteTyp*-Instanzen realisiert.

```

create or replace type KantenTyp as object(
    KantenID number,
    beginn ref PunkteTyp,
    ende ref PunkteTyp,
    member function Laenge return float
);

create or replace type body KantenTyp as
member function Laenge return float is
    von PunkteTyp;
    nach PunkteTyp;
    dx float;
    dy float;
    dz float;
begin
    UTL_REF.SELECT_OBJECT(self.beginn,von);
    UTL_REF.SELECT_OBJECT(self.ende,nach);
    dx := nach.x - von.x;
    dy := nach.y - von.y;
    dz := nach.z - von.z;
    return SQRT( dx*dx + dy*dy + dz*dz ) ;
end Laenge;
end;

```

Flächen werden durch einen umschreibenden Kantenzug repräsentiert. Wie zuvor gilt auch in diesem Fall, dass Kanten nicht eindeutig einer Fläche zugeordnet sind, d.h. es besteht keine existentielle Abhängigkeit. Aus diesem Grund erzeugen wir zuerst einen eigenen *KantenRefListenTyp*, der als „Sammelbehälter“ für Referenzen auf die Kanten, die eine Fläche umschreiben, dient.

```

create or replace type KantenRefListenTyp as table of ref KantenTyp;

create or replace type FlaechenTyp as object(

```

```

FlaechenID number,
Kanten KantenRefListenTyp,
member function Umfang return float,
member function Volumen return float );

```

```

create or replace type body FlaechenTyp as
member function Umfang return float is
umfang float := 0.0;
i integer := 0;
kante KantenTyp;
begin
for i in 1..self.Kanten.count loop
UTL_REF.SELECT_OBJECT( Kanten(i), kante);
umfang := umfang + kante.Laenge;
end loop;
return umfang;
end Umfang;

member function Volumen return float is
begin
-- hier Volumenberechnung einfügen
return 0.0;
end Volumen;
end;

```

Damit ist es uns nun möglich, den Objekttyp für Polyeder, *PolyederTyp*, zu definieren. Da Flächen Polyedern exklusiv zugeordnet sind (siehe Abbildung 1), enthält *FlaechenListenTyp* keine Referenzen auf Instanzen von Flächen, sondern die Instanzen selbst.

```

create or replace type FlaechenListenTyp as table of FlaechenTyp;

```

```

create or replace type PolyederTyp as object(
PolyID number,
Flaechen FlaechenListenTyp,
member function Gewicht return float,
member function Volumen return float,
member function verschieben (self in out PolyederTyp,
delta in PunkteTyp) return number,
member function rotieren (self in out PolyederTyp,
winkel in PunkteTyp) return number,
member function skalieren (self in out PolyederTyp,
faktor in float) return number
);

```

```

create or replace type body PolyederTyp as
member function Gewicht return float is
begin
-- Gewichtsberechnung einfügen

```

```

    return 0.0;
end Gewicht;

member function Volumen return float is
begin
    -- Volumenberechnung einfügen
    return 0.0;
end Volumen;

member function verschieben (self in out PolyederTyp, delta in PunkteTyp)
return number is
begin
    -- über alle Flächen iterieren und diese verschieben
    return ODCIConst.Success;
end verschieben;

member function rotieren (self in out PolyederTyp, winkel in PunkteTyp)
return number is
begin
    -- über alle Flächen iterieren und diese rotieren;
    return ODCIConst.Success;
end rotieren;

member function skalieren (self in out PolyederTyp, faktor in float)
return number is
begin
    -- über alle Flächen iterieren und diese skalieren;
    return ODCIConst.Success;
end skalieren;
end;

```

Nachdem wir nun alle erforderlichen Objekttypen definiert haben, erstellen wir im abschließenden Schritt das zugehörige objektrelationale Schema. Eine Besonderheit ist hier vielleicht die Definition der *PolyederTab*-Tabelle. Jeder Polyeder enthält eine Liste von Flächen, was durch die **nested table FlaechenListeTab** realisiert wird. In dieser werden aber Flächen-Objekte, und nicht nur Referenzen auf solche gespeichert. Deshalb ist es nötig, eine weitere **nested table** für die Kanten von Flächen-Instanzen zu definieren.

```

create table PunkteTab of PunkteTyp;
create table KantenTab of KantenTyp (KantenID primary key);

create table FlaechenTab of FlaechenTyp
(FlaechenID primary key)
nested table Kanten store as KantenListeTab;

create table PolyederTab of PolyederTyp (PolyID primary key)
nested table Flaechen store as FlaechenListeTab (
    nested table Kanten store as KantenListe2Tab
);

```

Die Tabelle *FlaechenTab* wird nur für einzelne Flächen benötigt, die nicht Teil eines Polyeders sind. Für die Schemadefinition der Begrenzungsflächendarstellung ist sie streng genommen nicht notwendig.

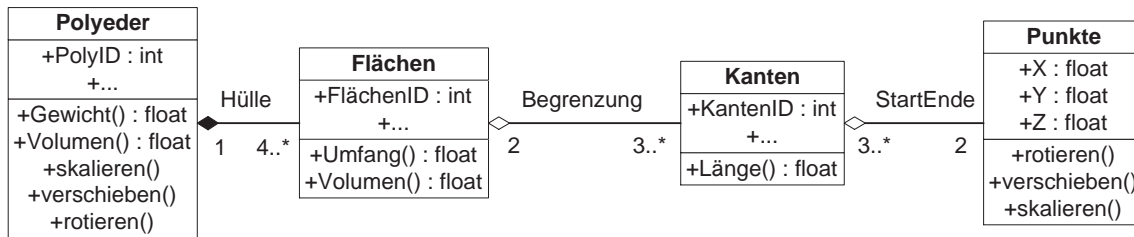


Abbildung 1: Objektorientierte Modellierung von Polyedern

Aufgabe 2

Die in der ersten Regel angegebene Konstante kürzen wir mit c ab, also

$$p(c, X, Y) \quad :- \quad r(X, Y).$$

$$p(X, Y, X) \quad :- \quad r(Y, X).$$

Als Nächstes formulieren wir beide Regeln derart um, dass sie jeweils den Kopf $p(U, V, W)$ haben.

$$p(U, V, W) \quad :- \quad r(X, Y), U = c, V = X, W = Y.$$

$$p(U, V, W) \quad :- \quad r(Y, X), U = X, V = Y, W = X.$$

Wir bedienen uns also des einfachen Tricks, neue Variablen einzuführen, um Konstanten und gleiche Variablenbezeichner im Kopf zu vermeiden. Indem wir die alten Variablen X und Y durch die neuen, d.h. U , V und W ersetzen, erhalten wir

$$p(U, V, W) \quad :- \quad r(V, W), U = c.$$

$$p(U, V, W) \quad :- \quad r(V, U), W = U.$$

In der ersten Regel haben wir also X durch V und Y durch W ersetzt. In der zweiten entsprechend X durch U und Y durch V .

Aufgabe 3

Gegeben sei das sichere Datalog-Programm p , welches wir durch das in Aufgabe 2 beispielhaft beschriebene Verfahren in ein Programm p' überführen. Dabei setzen wir voraus, dass, falls eine atomare Formel (Subgoal) $X = Y$ in p enthalten ist, wir die Variable Y nicht durch X substituieren und so keine Variablen eliminieren. Da p sicher ist, sind alle in p vorkommenden Variablen eingeschränkt. Da sich die für die Transformation von p nach p' neu eingeführten Variablen alle auf Variablen in p beziehen, folgt daraus, dass auch die Variablen in p' eingeschränkt sind. Damit ist auch p' sicher.

Aufgabe 4

Programm zur naiven Auswertung Die naive Auswertung ist ähnlich zur ursprünglichen naiven Auswertung:

```

A := {};
repeat
  A' := A;
  A := Vs(V, N); /*erste Regel*/
  A := A ∪ ΠV,N(A'(V, M) ⋈ A'(M, N)); /*zweite Regel*/
until A' = A
output A;

```

Semi-naive Auswertung Der vorgegebene Algorithmus zur semi-naiven Auswertung liefert nicht alle Tupel an Vorlesungspaaren, die aufeinander aufbauen.

Wendet man den Algorithmus auf die in Abbildung 2 dargestellte Beispielausprägung an, so ergeben sich in den jeweiligen Schritten folgende Ergebnistupel:

| Schritt | ΔA |
|-----------------|--|
| Initialisierung | (sieben Tupel aus Vs , d.h. voraussetzen) [5001, 5041], [5001, 5043], [5043, 5052], [5041, 5052], [5001, 5049], [5041, 5216], [5052, 5259] |
| 1. Iteration | (Pfade der Länge 2) [5001, 5216], [5001, 5052], [5041, 5259], [5043, 5259] |
| 2. Iteration | \emptyset hier bricht die Auswertung also zu früh ab, das Tupel [5001, 5259] wird nicht gefunden |

Der Fehler liegt in der Zeile $\Delta A := \Delta A \cup \Pi_{V,N}(\Delta A1'(V, M) \bowtie \Delta A2'(M, N))$; des ursprünglichen Programms.

Das korrekte Programm lautet:

```

A := {}; A1 := A; A2 := A; ΔVs := {};
ΔA := Vs(V, N) ∪ ΠV,N(A1(V, M) ⋈ A2(M, N));
A := ΔA;
repeat
  ΔA' := ΔA;
  ΔA1' := ΔA'; ΔA2' := ΔA';
  ΔA := ΔVs(V, N);
  ΔA := ΔA ∪ ΠV,N(ΔA1'(V, M) ⋈ A(M, N))
    ∪ ΠV,N(A(V, M) ⋈ ΔA2'(M, N));
  ΔA := ΔA - A;
  A := A ∪ ΔA;
until ΔA = ∅

```

Es ist also entscheidend, dass immer nur für eine einzige Subgoal-Relation das Delta aus dem vorhergehenden Auswertungsschritt eingesetzt wird. Die abgeleitete Information früherer Schritte geht ansonsten eventuell verloren!

| Professoren | | | | Studenten | | |
|-------------|------------|------|------|-----------|--------------|----------|
| PersNr | Name | Rang | Raum | MatrNr | Name | Semester |
| 2125 | Sokrates | C4 | 226 | 24002 | Xenokrates | 18 |
| 2126 | Russel | C4 | 232 | 25403 | Jonas | 12 |
| 2127 | Kopernikus | C3 | 310 | 26120 | Fichte | 10 |
| 2133 | Popper | C3 | 52 | 26830 | Aristoxenos | 8 |
| 2134 | Augustinus | C3 | 309 | 27550 | Schopenhauer | 6 |
| 2136 | Curie | C4 | 36 | 28106 | Carnap | 3 |
| 2137 | Kant | C4 | 7 | 29120 | Theophrastos | 2 |
| | | | | 29555 | Feuerbach | 2 |

| Vorlesungen | | | | voraussetzen | |
|-------------|----------------------|-----|------------|--------------|------------|
| VorlNr | Titel | SWS | gelesenVon | Vorgänger | Nachfolger |
| 5001 | Grundzüge | 4 | 2137 | 5001 | 5041 |
| 5041 | Ethik | 4 | 2125 | 5001 | 5043 |
| 5043 | Erkenntnistheorie | 3 | 2126 | 5001 | 5049 |
| 5049 | Mäeutik | 2 | 2125 | 5041 | 5216 |
| 4052 | Logik | 4 | 2125 | 5043 | 5052 |
| 5052 | Wissenschaftstheorie | 3 | 2126 | 5041 | 5052 |
| 5216 | Bioethik | 2 | 2126 | 5052 | 5259 |
| 5259 | Der Wiener Kreis | 2 | 2133 | | |
| 5022 | Glaube und Wissen | 2 | 2134 | | |
| 4630 | Die 3 Kritiken | 4 | 2137 | | |

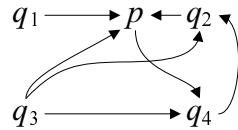
| hören | | Assistenten | | | |
|--------|--------|-------------|--------------|--------------------|------|
| MatrNr | VorlNr | PersNr | Name | Fachgebiet | Boss |
| 26120 | 5001 | 3002 | Platon | Ideenlehre | 2125 |
| 27550 | 5001 | 3003 | Aristoteles | Syllogistik | 2125 |
| 27550 | 4052 | 3004 | Wittgenstein | Sprachtheorie | 2126 |
| 28106 | 5041 | 3005 | Rhetikus | Planetenbewegung | 2127 |
| 28106 | 5052 | 3006 | Newton | Keplersche Gesetze | 2127 |
| 28106 | 5216 | 3007 | Spinoza | Gott und Natur | 2134 |
| 28106 | 5259 | | | | |
| 29120 | 5001 | | | | |
| 29120 | 5041 | | | | |
| 29120 | 5049 | | | | |
| 29555 | 5022 | | | | |
| 25403 | 5022 | | | | |
| 29555 | 5001 | | | | |

| prüfen | | | |
|--------|--------|--------|------|
| MatrNr | VorlNr | PersNr | Note |
| 28106 | 5001 | 2126 | 1 |
| 25403 | 5041 | 2125 | 2 |
| 27550 | 4630 | 2137 | 2 |

Abbildung 2: Beispielausprägung der Universitäts-Datenbank

Aufgabe 5

Das angegebene Datalog-Programm ist *nicht stratifiziert*. In der ersten Regel tritt das Subgoal q_2 negiert auf. Während der Auswertung von p liegt q_2 allerdings noch nicht vollständig materialisiert vor, da q_2 transitiv von p abhängt. Dies zeigt sich in folgendem Abhängigkeitsgraphen, in dem ein Zyklus auftritt, der p , q_4 und q_2 umfasst.



Das Programm ist *sicher*, da der Aufgabenstellung zufolge alle angegebenen Subgoals p , q_1 , q_2 , q_3 , q_4 IDB- oder EDB-Prädikate sind. Da IDB- und EDB-Prädikate endlich sind, gilt damit die Behauptung. Für den Nachweis, dass ein Datalog-Programm sicher ist, dürfen negierte Subgoals nicht betrachtet werden. In q_2 tritt jedoch keine freie Variable auf, die nicht auch in einem nicht-negierten Subgoal auftritt. Im Speziellen heißt dies, dass die Variablen X und Z eingeschränkt sind.