

# Funktionale Datenbanken

Henning Thielemann

2010-02-05



1 Übersicht

2 Monade

3 Mehr Lösungen

4 Ausblick



# 1 Übersicht

2 Monade

3 Mehr Lösungen

4 Ausblick



# Probleme in SQL

- Spezialsprache: erlaubt keine vollständigen Anwendungen
- rekursive Anfragen
- Behandlung von NULL-Werten



# Probleme in deduktiven Datenbanken

- Negation
- Gruppierung und Aggregation
- Änderungen in Tabellen
- Grenzen der Optimierung: Halteproblem



# Funktionale Programmierung

- Prinzip: drücke alles mit Funktionen aus
- erlaubt Programmbeweise  
und darauf aufbauend Optimierungen
- kann mit oben genannten Problemen umgehen
- Allerdings: Datenbankoptimierungen fehlen (noch)



# Datenbanken haben viele funktionale Elemente

- relationale Algebra
- SQL: JOIN, UNION, INTERSECT, EXCEPT, Unterabfragen
- Datenbank-Optimierer



# Verschiedene Paradigmen in SQL: JOIN

```
SELECT E.ENAME, D.DNAME  
FROM   EMP E, DEPT D  
WHERE  E.DEPTNO = D.DEPTNO
```

```
SELECT ENAME, DNAME  
FROM   EMP NATURAL JOIN DEPT
```



# Verschiedene Paradigmen in SQL: UNION

```
SELECT ENAME  
FROM EMP  
WHERE JOB = 'CLERK' OR SAL <= 1000
```

```
SELECT ENAME FROM EMP WHERE JOB = 'CLERK'  
UNION  
SELECT ENAME FROM EMP WHERE SAL <= 1000
```



# Intuition

- Mensch denkt in Beispielen
- elementorientierte Abfragen einfacher zu verstehen als Tabellentransformationen

deshalb:

- SELECT-Anfrageschablone in SQL
- Regeln in deduktiven Datenbanken
- Monade in funktionaler Programmierung



## Denken in Beispielen

```
SQL      SELECT E.ENAME
         FROM   EMP E, DEPT D
         WHERE  E.DEPTNO = D.DEPTNO
            AND D.DNAME = 'RESEARCH'
```

```
Deduktiv researcher(Name) :-
          emp(_,Name,_,_,_,DeptNo),
          dept(DeptNo,"RESEARCH",_).
```

```
do-Notation do e <- emp
            d <- dept
            guard (deptno e == deptno d &&
                  dname d == "RESEARCH")
            return (ename e)
```



1 Übersicht

2 **Monade**

3 Mehr Lösungen

4 Ausblick



# Übersetzung SQL → do-Notation

```
do -- FROM EMP E, DEPT D
  e <- emp
  d <- dept
  {- WHERE E.DEPTNO = D.DEPTNO
      AND D.DNAME = 'RESEARCH' -}
  guard (deptno e == deptno d &&
        dname d == "RESEARCH")
  -- SELECT E.ENAME
  return (ename e)
```



# do-Notation

- an imperative Sprachen angelehnte Notation
- erlaubt für alle Monade
- „Überladen des Semikolons“
- nur syntaktischer Zucker
- Übersetzung in Funktionsaufrufe einfach



# do-Notation → Monad-Kombinatoren

```
do e <- emp
  d <- dept
  guard (deptno e == deptno d &&
         dname d == "RESEARCH")
  return (ename e)
```



# do-Notation → Monad-Kombinatoren

```
emp >>= \e ->
do d <- dept
  guard (deptno e == deptno d &&
         dname d == "RESEARCH")
  return (ename e)
```



## do-Notation → Monad-Kombinatoren

```
emp >>= \e ->
dept >>= \d ->
do guard (deptno e == deptno d &&
          dname d == "RESEARCH")
   return (ename e)
```



## do-Notation → Monad-Kombinatoren

```
emp >>= \e ->  
dept >>= \d ->  
guard (deptno e == deptno d &&  
       dname d == "RESEARCH") >>  
do return (ename e)
```



## do-Notation → Monad-Kombinatoren

```
emp >>= \e ->  
dept >>= \d ->  
guard (deptno e == deptno d &&  
       dname d == "RESEARCH") >>  
return (ename e)
```



# Nichtdeterministische Verbindung

```
(>>=) :: List a -> (a -> List b) -> List b  
x >>= y = concat (map y x)
```

```
Prelude> ["Miller", "Smith", "Clark"] >>=  
  \name -> [name ++ " Sr.", name ++ " Jr."]
```

```
["Miller Sr.", "Miller Jr.", "Smith Sr.",  
"Smith Jr.", "Clark Sr.", "Clark Jr."]
```



# Leere Listen

```
guard :: Bool -> List ()  
guard b = if b then [()] else []
```

```
Prelude> ["Miller", "Smith", "Clark"] >>=  
  \name -> Monad.guard (length name < 6)
```

```
[(), ()]
```

Übung: Schreiben Sie eine Funktion `duplicate` mit Signatur `guard :: Int -> List ()` welche eine Liste mit einer vorgegebenen Anzahl Elemente erzeugt.

Tip: `replicate`



# Einelementige Listen

```
return :: a -> List a  
return a = [a]
```

```
Prelude> ["Miller", "Smith", "Clark"] >>=  
  (\name -> Monad.guard (length name < 6) >>  
    return name)
```

```
["Smith", "Clark"]
```



# Monad-Kombinatoren und Gesetze

Methoden für Monad  $m$

`return` ::  $a \rightarrow m\ a$

`(>>=)` ::  $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

Der Infix-Operator „`>>=`“ heißt „bind“.

`(>>>)` ::  $m\ a \rightarrow m\ b \rightarrow m\ b$

$x \gg y = x \gg= \_ \rightarrow y$

Gesetze

Linksneutrales Element    `return a >>= x = x a`

Rechtsneutrales Element    `x >>= return = x`

Assoziativität            `(x >>= y) >>= z`  
                               `= x >>= \a -> (y a >>= z)`



# Gesetze für Listenmonade

Selber überprüfen!



# Monade als Kästen

Bildchen



# Monadgesetze ausgedrückt in do-Notation

$$\begin{array}{l} \text{do } b \leftarrow \text{return } a \\ \quad x \ b \end{array} = \text{do } x \ a$$
$$\begin{array}{l} \text{do } a \leftarrow x \\ \quad \text{return } a \end{array} = \text{do } x$$
$$\begin{array}{l} \text{do } b \leftarrow \\ \quad \text{do } a \leftarrow x \\ \quad \quad y \ a \\ \quad \quad z \ b \end{array} = \begin{array}{l} \text{do } a \leftarrow x \\ \quad \text{do } b \leftarrow y \ a \\ \quad \quad z \ b \end{array}$$


# Alternative Monad-Kombinatoren und Gesetze

Wem die Operationen und Gesetze zu unsymmetrisch erscheinen, kann auch diese nehmen:

```
return :: a -> m a
fmap  :: (a -> b) -> m a -> m b
join  :: m (m a) -> m a
```

Es gilt:  $x \gg= y = \text{join } (\text{fmap } y \ x)$

Gesetze (veranschaulichen Sie sich die Typen der Teilausdrücke!)

Linksneutrales Element  $\text{join } (\text{return } x) = x$

Rechtsneutrales Element  $\text{join } (\text{fmap } \text{return } x) = x$

Assoziativität  $\text{join} = \text{fmap } \text{join}$

Funktoridentität  $\text{fmap } \text{id} = \text{id}$

In Anwendung sind zuerst gezeigte Operationen bequemer.



# Anwendungen

Es gibt sehr viele Dinge, die Monadstruktur besitzen

- Ein-/Ausgabe (IO)
- Zustand (State)
- globaler Kontext (Reader)
- Protokoll (Writer)
- Transaktionen (STM)
- Nichtdeterminismus (List)
- Wahrscheinlichkeitsrechnung
- Ausnahmebehandlung (Maybe)
- Parser (Parsec)



# Warnungen I

- Auch wenn Monade sehr vielseitig sind – manchmal sind Monoide besser geeignet, oder Functors, Pointed Functors, Applicative Functors, Arrows, Categories, ...
- Die do-Notation sagt nichts darüber aus, ob und in welcher Reihenfolge Aktionen ausgeführt werden.
- Wenn Sie denken, dass Sie Monade verstanden haben – Schreiben Sie keine Einleitung in Monade! Es gibt schon zu viele unausgegrenzte.



# Warnungen II

- Widerstehen Sie der Versuchung, Monade nur deshalb zu wählen, weil sie die do-Notation benutzen wollen.
- Monade bergen die Gefahr, auch in funktionaler Programmierung weiter imperativ zu programmieren.  
„Ein guter FORTRAN-Programmierer ist in jeder Sprache ein guter FORTRAN-Programmierer.“



- 1 Übersicht
- 2 Monade
- 3 Mehr Lösungen
- 4 Ausblick



# NULL-Werte

- In normaler Programmiersprache sind alle Typen „NOT NULL“
- In Haskell gibt es  
`data Maybe a = Just a | Nothing`
- Maybe ist ein Monad: Wenn in einem Block von Berechnungen eine einzelne fehlschlägt, dann soll der ganze Block fehlschlagen.



# Negation

- benötigt keine spezielle Behandlung
- Beispiel

```
do e <- emp
  d <- dept
  guard (deptno e == deptno d &&
         dname d /= "RESEARCH")
return (ename e)
```



# Tabellenoperationen

Manche Operationen lassen sich nicht auf der Ebene einzelner Elemente ausdrücken

- Sortierung
- Duplikateentfernung
- Aggregation
- Gruppierung

Vorteile in funktionaler Programmierung:

- Tabellen sind selbst Objekte
- Tabellen mit Untertabellen sind möglich
- Tabellen und einzelne Werte werden unterschieden (Aggregationen)



# Änderung von Tabellen

- Objekte in funktionaler Programmierung nur lesbar
- benötigte Referenzen: Ref
- benötigte Serialisierung, da zu jedem Zeitpunkt die gleiche Referenz auf verschiedene Inhalte verweisen kann:  
Monad Trans



# Änderung von Tabellen

- `create :: List a -> Trans (Ref (List a))`
- `from :: Ref (List a) -> Trans (List a)`
- `insert :: Ref (List a) -> a -> Trans ()`
- `delete :: Ref (List a) -> (a -> Bool) -> Trans ()`
- `update :: Ref (List a) -> (a -> a) -> Trans ()`



- 1 Übersicht
- 2 Monade
- 3 Mehr Lösungen
- 4 **Ausblick**



# Probleme mit Listen für relationale Anfragen

- Nicht effizient
- Reihenfolgeabhängigkeit (keine echten Multimengen)

Lösung: Spezialisierter Typ für Tabellen



# Datenbanken mit Haskell

- HaskellDB: Monad baut SQL-Anfragen zusammen
  - setzt auf JDBC oder HSQL auf
  - Tupel werden nur bei Bedarf ausgeliefert („lazy“)
- JDBC: Kommunikation mit einigen Datenbanksystemen
- HSQL: dito
- Takusen: Iterator-basiert (Alternative zu Bedarfsauswertung)

