Deskriptive Programmierung

List Comprehensions

Arithmetische Sequenzen

- nützliche Notationsform für Listen von Zahlen:

arithmetische Sequenzen

- abkürzende Schreibweise für Listen von Zahlen mit identischer Schrittweite:

```
>   [ 1 .. 10 ]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Höhere Schrittweite als 1 wird durch Angabe eines zweiten Elements festgelegt:

```
>  [ 1, 3 .. 10 ]
[1, 3, 5, 7, 9]
```

- alternative Definition der Fakultätsfunktion (ohne explizite Rekursion):

```
fac n  =  prod  [ 1 .. n ]
```

List comprehensions (1)

- mächtiges und elegantes Sprachkonzept in Haskell:

list comprehension

engl. von „comprehensive“: umfassend

- Vorbild: implizite Mengennotation der Mathematik (Menge aller x, so dass ...), z.B.

$$\{\ x^2 \mid x \in \{1, \ldots, 100\} \wedge (x \bmod 2) = 0\ \}$$

- in Haskell analoges Konzept für Listen:

```
[ x^2 | x ← [1 .. 100] , (x `mod` 2) == 0 ]
```

List comprehensions (2)

- Eine *list comprehension* besteht im Prinzip aus drei „Zutaten“:

$$[\ \underbrace{x\text{\textasciicircum}2}_{\text{Body}}\ |\ \underbrace{x \leftarrow [1\ ..\ 100]}_{\text{Generator}}\ ,\ \underbrace{(x\ \text{`mod`}\ 2) == 0}_{\text{Guard}}\ ]$$

- Der Body für die Listenelemente ist ein Ausdruck, der in der Regel mindestens eine Variable enthält, deren mögliche Werte durch den Generator erzeugt werden.

- Der Generator ist ein Ausdruck der Form Variable $\leftarrow$ Liste, der die Variable sukzessive an alle Elemente der Liste bindet (in der Listenreihenfolge).

- Der Guard ist ein Boolescher Ausdruck, der die generierten Werte auf diejenigen beschränkt, für die der Ausdruck den Wert True liefert.

- Zusätzlich möglich: lokale Definitionen mittels let.

List comprehensions (3)

- Teile sind optional, z.B.:

```
[ x^2 | x ← [1 .. 10] ]
```

- Eine list comprehension darf auch mehrere Variablen mit mehreren Generatoren enthalten, z.B.:

```
>  [ (x, y) | x ← [ 1, 2, 3 ], y ← [1 .. x] ]
[ (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) ]
```

- Jede (nicht aus Kontext bekannte) Variable braucht einen Generator:

```
[ (x * y) | x ← [ 1, 2, 3 ], y ← [ 1, 2, 3 ] ]
```

aber auch

```
[ x ++ y | (x, y) ← [ ("a", "b"), ("c", "d") ] ]
```

List comprehensions (4)

- Reihenfolge der Generatoren beeinflusst Ausgabereihenfolge:

```
> [ (x, y) | x ← [ 1, 2, 3 ], y ← [ 4, 5 ] ]
[ (1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5) ]
```

vs.

```
> [ (x, y) | y ← [ 4, 5 ], x ← [ 1, 2, 3 ] ]
[ (1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5) ]
```

(wie verschachtelte Schleifen)

List comprehensions (5)

- „Spätere“ Generatoren können von „früheren“ abhängen, z.B.:

```
> [ (x, y) | x ← [ 1, 2, 3 ],  y ← [1 .. x] ]
[ (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) ]
```

- Insbesondere kann eine per Generator gebundene Variable selbst als „Erzeugerliste“ dienen:

```
concat :: [[Int]] → [Int]
concat xss  =  [ x | xs ← xss, x ← xs ]
```

```
> concat [ [ 1, 2, 3 ], [ 4, 5 ], [ 6 ], [ ] ]
[ 1, 2, 3, 4, 5, 6 ]
```

List comprehensions (6)

- Guards können (auch) <u>nur von früheren</u> Generatoren abhängen, z.B.:

```
> [ x | x ← [1 .. 10], even x ]
[ 2, 4, 6, 8, 10 ]
```

- Noch ein Beispiel:

```
factors :: Int → [Int]
factors n  =  [ x | x ← [1 .. n], n `mod` x == 0 ]
```

```
> factors 15
[ 1, 3, 5, 15 ]
```

List comprehensions (7)

- Kann man folgende Funktion mit list comprehensions realisieren?

```
zip :: [Int] → [Int] → [(Int, Int)]
zip (x : xs) (y : ys)        =  (x, y) : zip xs ys
zip xs       ys              =  [ ]
```

```
> zip [1 .. 3] [10 .. 15]
[ (1, 10), (2, 11), (3, 12) ]
```

- Nur mit „parallel list comprehensions“:

```
zip :: [Int] → [Int] → [(Int, Int)]
zip xs ys  = [ (x, y) | x ← xs | y ← ys ]
```

weder in Haskell 98,
noch in Haskell 2010,
aber als GHC extension

Unendliche Listen

- Es gibt in Haskell sogar abkürzende Notationen für unendliche Listen.

```
[ 1, 3 .. ]    steht für    [ 1, 3, 5, 7, 9, . . . . . . . ]
```

- Zum Beispiel:

```
naturals,evens,odds :: [Integer]
naturals   =   [ 1 .. ]
evens      =   [ 2, 4 .. ]
odds       =   [ 1, 3 .. ]
```

- Damit lassen sich unendliche Folgen als Listen darstellen, z.B.:

```
squares   =   [  n^2  |  n ← naturals ]
facs      =   [ fac n  |  n ← naturals ]
primes    =   2 : [ n  |  n ← odds, factors n == [1, n] ]
```

„Endliches Arbeiten“ mit unendlichen Listen

- Eingabe eines Ausdrucks, der eine unendliche Liste bezeichnet, führt zu einer nicht-terminierenden Ausgabe (muss „per Hand“ unterbrochen werden!)

- Praktikabel ist aber das Arbeiten mit endlichen Teillisten von unendlichen Listen, z.B.:

```
> take 5 primes
[2, 3, 5, 7, 11]
```

```
> primes !! 5
13
```

- Dass so etwas möglich ist, ist nicht selbstverständlich, sondern liegt daran, dass Haskell mit einer speziellen Auswertungsstrategie arbeitet, die den Wert eines Ausdrucks nur dann berechnet, wenn er unbedingt nötig ist („lazy evaluation“).

- Dieser Ausdruck bezeichnet zwar „intuitiv“ eine endliche Liste, die Berechnung terminiert aber nicht:

Warum ?

```
> [ x | x ← squares, x < 100 ]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
```

Varianten zur Primzahlgenerierung

- Statt:

```
odds        = [ 1, 3 .. ]
factors n   = [ x | x ← [1 .. n], n `mod` x == 0 ]
primes      = 2 : [ n | n ← odds, factors n == [ 1, n ] ]
```

- Zum Beispiel:

```
primes     = 2 : [ n | n ← [ 3, 5 .. ], isPrime n ]
isPrime n  = and [ n `mod` t > 0 | t ← candidates primes ]
  where candidates (p : ps) | p * p > n  = [ ]
                            | otherwise  = p : candidates ps
```

- Oder auch:

```
primes        = sieve [ 2 .. ]
sieve (p:xs)  = p : sieve [ x | x ← xs, x `mod` p > 0 ]
```

Deskriptive Programmierung

Die Rolle (bzw. Arten) von Rekursion

Pattern Matching + Rekursion vs. List Comprehensions

Wir hatten gesehen:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i - 1)
```

```
> sumsquare 4
30
```

Aber auch möglich:

```
sumsquare :: Int → Int
sumsquare n  =  sum [ i * i  |  i ← [0 .. n] ]
```

```
> sumsquare 4
30
```

Welche Form ist nun „besser“?

Lässt sich nicht so ohne Weiteres beantworten. Was wären Kriterien?
Vielleicht:

- Effizienz
- Lesbarkeit
- „Beweisbarkeit“

Fakt: Auch sum, [0 .. n], … sind letztlich selbst rekursiv definiert.

Die Rolle von Rekursion, bzw. zwei Arten von Rekursion

Strukturelle Rekursion:

```
sum :: [Int] → Int
sum [ ]       =  0
sum (x:xs)  =  x + sum xs
```

Letztlich auch „strukturell“ oder zumindest einfach induktiv:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i - 1)
```

Allgemeine Rekursion:

```
quersumme :: Int → Int
quersumme n  | n < 10       =  n
             | otherwise   =  let (d, m) = n `divMod` 10  in  m + quersumme d
```

Auch: ack, …, Quicksort, …

Vergleich allgemeine und strukturelle Rekursion

- Allgemeine Rekursion ist deutlich flexibler!
  - algorithmische Prinzipien wie „Divide and Conquer“ direkt umsetzbar
  - manche Funktionen sind beweisbar nicht durch strukturelle Rekursion realisierbar

- Strukturelle Rekursion:
  - liefert ein sehr nützliches „Rezept“ zur Definition von Funktionen
  - garantiert Termination (auf endlichen Strukturen)
  - ermöglicht sehr direkt Beweise per Induktion
  - lässt sich als wiederverwendbares Programmschema „verpacken“

Deskriptive Programmierung

Typen in Haskell

Typen

- Ein wichtiges Grundkonzept von Haskell, bisher eher beiläufig betrachtet:

  > Jeder Ausdruck und jede Funktion hat einen Typ.

- Notation für Typzuordnung: doppelter Doppelpunkt

  z.B.: `1 :: Int`

- Grundlage: vordefinierte Basistypen für Konstanten

  - diverse numerische Typen, z.B. Integer, Rational, Float, Double
  - Buchstaben: Char
  - Wahrheitswerte: Bool

- zusätzlich: diverse Typkonstruktoren für komplexe Typen

Typisierung, „type checking“, „type inference“

- Jeder Ausdruck hat genau einen Typ, der noch vor der Laufzeit bestimmbar ist:

  Haskell ist eine stark und statisch getypte Sprache.

- Funktionsdefinition und -anwendungen werden auf Typkonsistenz geprüft:

  Typprüfung (engl.: „type checking“)

- Haskell bietet darüber hinaus Typherleitung (engl.: „type inference“) , d.h., Typen müssen nicht unbedingt explizit angegeben werden.

- Es gibt kein (implizites oder explizites) Casting zwischen Typen.

Besonderheiten zur Typisierung von Zahlen

- Es wurden bereits verschiedene Zahlentypen erwähnt: Int, Integer, Float (und es gibt noch eine ganze Reihe weiterer, zum Beispiel Rational).

- Zahlenliterale können je nach Kontext verschiedenen Typ haben (zum Beispiel, 3 :: Int, 3 :: Integer, 3 :: Float, 3.0 :: Float, 3.5 :: Float, 3.5 :: Double).

- Für allgemeine Ausdrücke gibt es überladene Konversionsfunktionen, zum Beispiel:
  - fromIntegral :: Int → Integer, fromIntegral :: Integer → Int, fromIntegral :: Int → Rational, fromIntegral :: Integer → Float, …
  - truncate :: Float → Int, truncate :: Double → Int, truncate :: Float → Integer, …, round :: …, ceiling :: …, floor :: …

- Konversionen sind nicht nötig in zum Beispiel 3 + 4.5 oder in:

```
f x = 2 * x + 3.5
g y = f 4 / y
```

,

aber zum Beispiel in:

```
f :: Int → Float
f x = 2 * (fromIntegral x) + 3.5
```

oder in:

```
f x = 2 * x + 3.5
g y = f (fromIntegral (length "abcd")) / y
```

Deskriptive Programmierung

Algebraische Datentypen

Deklaration von (algebraischen) Datentypen

- Ein wesentlicher Aspekt typischer Haskell-Programme ist die Definition problemspezifischer Datentypen (statt alles aus Listen zu bauen o.ä.).

- Dazu dienen in erster Linie Datentypdeklarationen:

```
data  Color  =  Red | Green | Blue | White | Black
```

Typkonstruktor

Datenkonstruktoren

- Konstruktoren in Haskell (Daten- wie Typkonstruktoren) beginnen grundsätzlich mit Großbuchstaben (Ausnahme: bestimmte symbolische Formen wie bei Listen).

- Der hier neu definierte Typ Color ist ein Aufzählungstyp, der aus genau den fünf aufgeführten Elementen besteht.

Deklaration von (algebraischen) Datentypen

- Selbst deklarierte Datentypen:

```
data  Color  =   Red  |  Green  |  Blue  |  White  |  Black
```

… können beliebig als Komponenten in anderen Typen auftreten, etwa [(Color, Int)] mit Werten z.B. [ ], [(Red, -5)] und [(Red, -5), (Blue, 2), (Red, 0)].

- Berechnung mittels Pattern-Matching möglich:

```
primary_col :: Color → Bool
primary_col Red     =  True
primary_col Green   =  True
primary_col Blue    =  True
primary_col _       =  False
```

Selbstdefinierte strukturierte Typen

- Man kann auch eigene strukturierte Typen deklarieren, indem man einen Datenkonstruktor mit Parametern einsetzt:

```
data  Point  =   Pt  ( Integer, Integer )
```

Typ-
konstruktor

Daten-
konstruktor

Typ gekapselter Daten

- Mit einem solchen selbstdefinierten Datenkonstruktor lassen sich dann strukturierte Werte eines selbstdefinierten Typs konstruieren:

```
Pt (1,2) :: Point
```

- Es ist zulässig, für die Bezeichnung (irgend-) eines Typs denselben Konstruktor zu verwenden wie zur Konstruktion von Datenelementen (etwa hier beide Male Pt).

Selbstdefinierte strukturierte Typen

- Ein etwas komplexeres Beispiel:

```
data Date  = Date Int Int Int
data Time = Hour Int
data Connection  =  Flight String Date Time Time |
                    Train Date Time Time
```

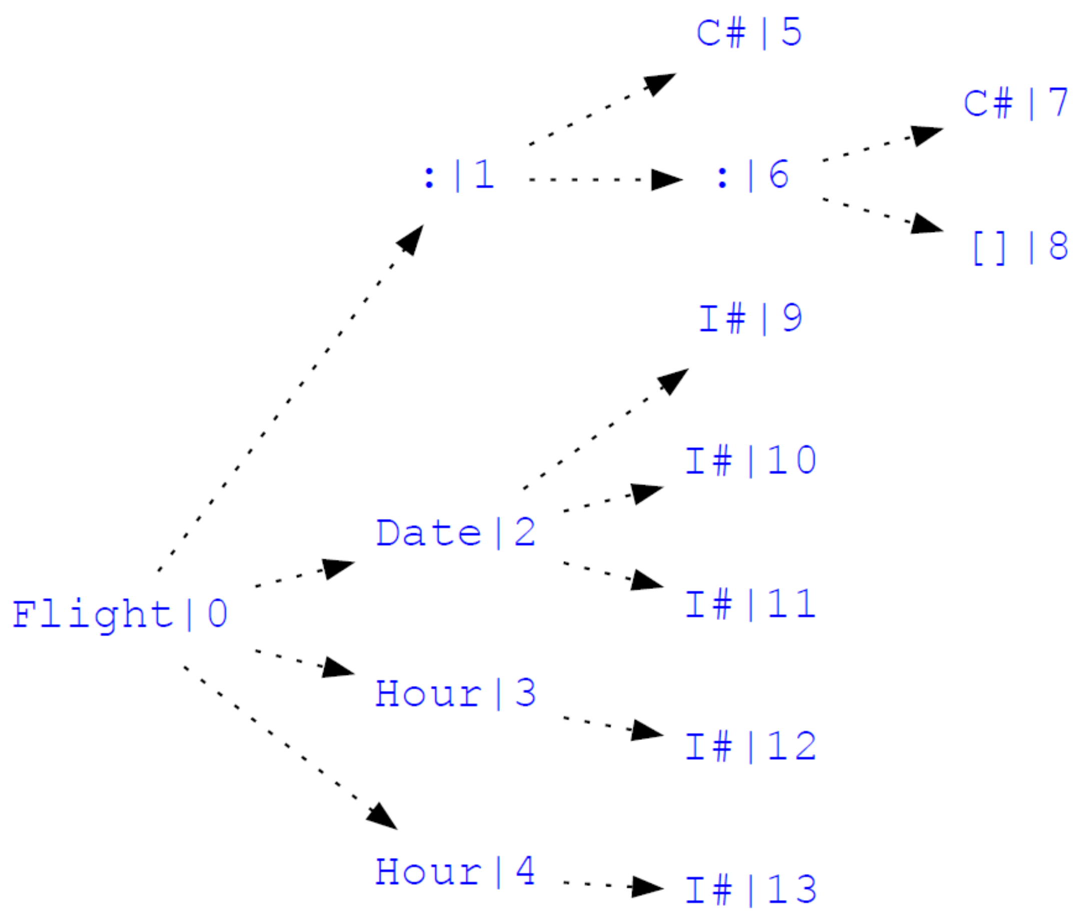
- mögliche Werte für Connection:
  - Train (Date 20 04 2011) (Hour 14) (Hour 11)
  - Flight "LH" (Date 20 04 2011) (Hour 16) (Hour 30)
  - …

- Berechnung mittels Pattern-Matching:

```
travelTime :: Connection → Int
travelTime (Flight _ _ (Hour d) (Hour a)) = a-d+2
travelTime (Train _ (Hour d) (Hour a))     = a-d+1
```

Selbstdefinierte strukturierte Typen

- interne Repräsentation für: Flight "LH" (Date 20 04 2011) (Hour 16) (Hour 30)



Datenkonstruktoren als spezielle Funktionen

Für:

```
data Date        =  Date Int Int Int
data Time        =  Hour Int
data Connection  =  Flight String Date Time Time |
                    Train Date Time Time
```

erhalten wir:

```
> :t Date
Date :: Int → Int → Int → Date
> :t Hour
Hour :: Int → Time
> :t Flight
Flight :: String → Date → Time → Time → Connection
> :t Train
Train :: Date → Time → Time → Connection
```

Rekursive Datentypen

- Wie Funktionsdefinitionen können auch Datentypdeklarationen rekursiv sein.

- Vielleicht das einfachste Beispiel:

```
data Nat  =  Zero | Succ Nat
```

- Werte von Typ Nat:

  Zero, Succ Zero, Succ (Succ Zero), …

- Berechnungen darauf mittels Pattern-Matching:

```
add :: Nat → Nat → Nat
add Zero       m = m
add (Succ n)  m = Succ (add n m)
```

Rekursive Datentypen

- Ein etwas komplexeres Beispiel (so ähnlich schon gesehen):

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr
```

- Mögliche Werte:

  Lit 42 , Add (Lit 2) (Lit 7) , Mul (Lit 3) (Add (Lit 4) (Lit 0)) , …

- Ein „Mini-Interpreter“:

$$
\begin{aligned}
&\text{eval} :: \text{Expr} \rightarrow \text{Int}\\
&\text{eval}\ \ (\text{Lit n}) &&= \text{n}\\
&\text{eval}\ \ (\text{Add } e_1\ e_2) &&= \text{eval } e_1 + \text{eval } e_2\\
&\text{eval}\ \ (\text{Mul } e_1\ e_2) &&= \text{eval } e_1 * \text{eval } e_2
\end{aligned}
$$

- Oder auch allgemeine Binärbäume:

```
data Tree  =  Leaf Int | Node Tree Int Tree
```

- mit wie folgt getypten Datenkonstruktoren:

```
> :t Leaf
Leaf :: Int → Tree
> :t Node
Node :: Tree → Int → Tree → Tree
```

- und (zu definierenden) Funktionen für „Flattening“, Prefix-Traversal, Postfix-Traversal, …

Simultan-rekursive Datentypen

- Schließlich, ein etwas künstliches Beispiel:

```
data T1  =  A T2 | E
data T2  =  B T1
```

- Mögliche Werte für T1:

E , A (B E) , A (B (A (B E))) , A (B (A (B (A (B E))))) , …

- Mögliche Werte für T2:

B E , B (A (B E)) , B (A (B (A (B E)))) , …

- Berechnung:

```
as :: T1 → Int
as (A t) = 1 + as' t
as E     = 0

as' :: T2 → Int
as' (B t) = as t
```