Typsynonyme

- Typsynonyme vergeben neue Namen für schon existierende Typen:

$$\text{type String} \;=\; [\text{Char}]$$

  – im Unterschied zu data keine Konstruktoren, keine Alternativen; außerdem wirklich nur ein neuer Name, kein neuer Typ

  – können verschachtelt sein:

$$\begin{aligned} &\text{type Pos} && = \; (\text{Int}, \text{Int}) \\ &\text{type Trans} && = \; \text{Pos} \rightarrow \text{Pos} \end{aligned}$$

  aber nicht rekursiv!

Typisomorphe

- mit newtype Erzeugung einer unterscheidbaren Kopie eines vorhandenen Typs:

```
newtype Rat = Rat (Int, Int)
```

  – genau ein Datenkonstruktor mit genau einem Argument, keine Alternativen; wirklich ein neuer Typ!

  – typischer Anwendungsfall:

```
newtype Rat = Rat (Int, Int)
newtype Pos = Pos (Int, Int)
```

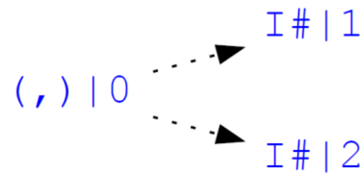
  – Rekursion erlaubt:

```
newtype InfList = Cons (Int, InfList)
```

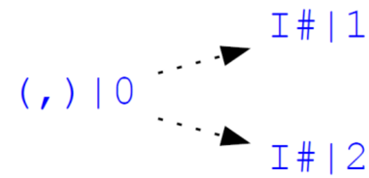
  – per Anschein nur ein Spezialfall von data, tatsächlich aber Unterschiede hinsichtlich Effizienz und Termination

Typsynonyme vs. Typisomorphe vs. „data“

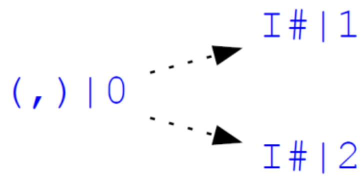
```
type Pos = (Int, Int)
p = (3, 4)
```



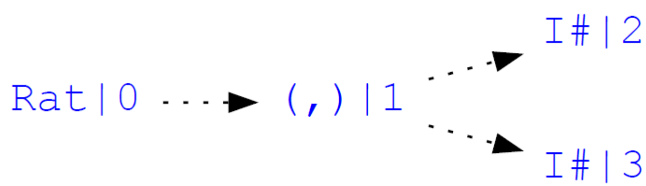

```
newtype Pos = Pos (Int, Int)
p = Pos (3, 4)
```



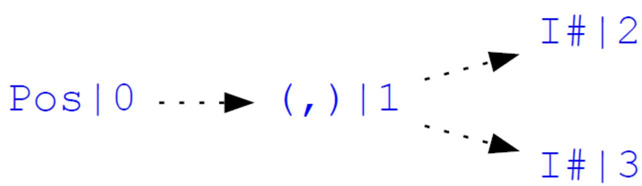

```
newtype Rat = Rat (Int, Int)
r = Rat (3, 4)
```




```
data Rat = Rat (Int, Int)
r = Rat (3, 4)
```

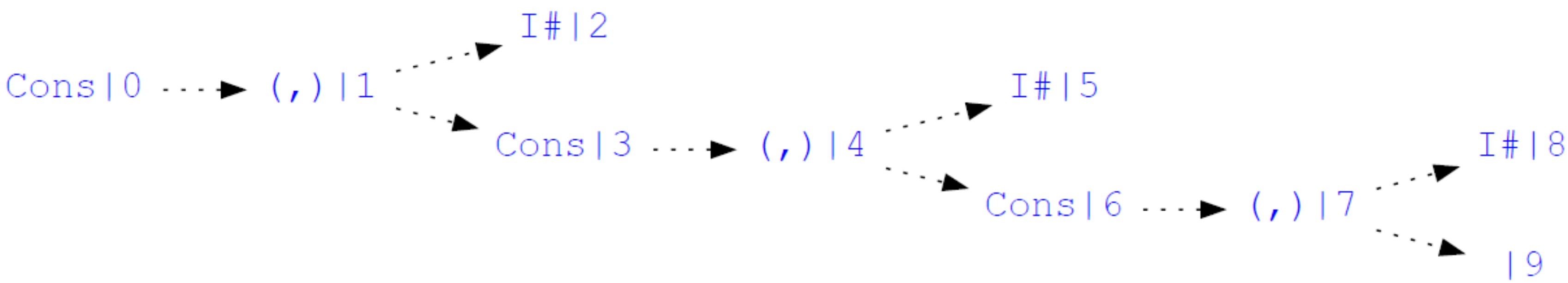



```
data Pos = Pos (Int, Int)
p = Pos (3, 4)
```

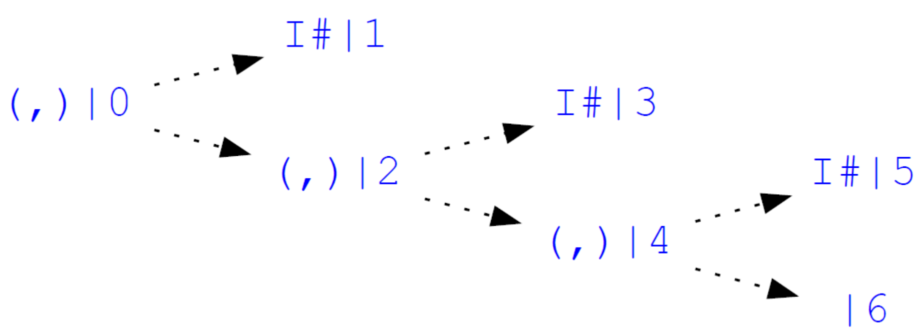


Typsynonyme vs. Typisomorphe vs. „data“

data InfList = Cons (Int, InfList)
xs = Cons (1, Cons (2, Cons (3, xs)))



newtype InfList = Cons (Int, InfList)
xs = Cons (1, Cons (2, Cons (3, xs)))



~~type InfList = (Int, InfList)~~
~~xs = (1, (2, (3, xs)))~~

Deskriptive Programmierung

Parametrische Polymorphie

Parametrisch polymorphe Funktionen

- Viele schon gesehene/vorhandene Listenoperatoren sind für Listen aus beliebigen Elementtypen gedacht, z.B.:

```
length [ ]       =  0
length (x : xs)  =  length xs + 1
```

```
> length [ 1, 2 ]
2
```

```
> length [ [ ], [ 'a', 'b', 'c' ] ]
2
```

- Wie für Standardfunktionen hat man natürlich auch für selbstdefinierte Funktionen gern solche Flexibilität:

```
concatenation  [ ]       ys   =   ys
concatenation  (x : xs)  ys   =   x : concatenation xs ys
```

Parametrisch polymorphe Funktionen

- Statt mehrerer Varianten:

$$
\begin{array}{l}
\text{concatenation} :: [\text{Int}] \to [\text{Int}] \to [\text{Int}] \\
\text{concatenation}\ \ [\,]\qquad\quad \text{ys} \ = \ \text{ys} \\
\text{concatenation}\ \ (\text{x} : \text{xs})\ \ \text{ys} \ = \ \text{x} : \text{concatenation xs ys}
\end{array}
$$

$$
\begin{array}{l}
\text{concatenation}' \ :: [\text{Bool}] \to [\text{Bool}] \to [\text{Bool}] \\
\text{concatenation}'\ [\,]\qquad\quad \text{ys} \ = \ \text{ys} \\
\text{concatenation}'\ (\text{x} : \text{xs})\ \ \text{ys} \ = \ \text{x} : \text{concatenation}'\ \text{xs ys}
\end{array}
$$

$$
\begin{array}{l}
\text{concatenation}'' \ :: \text{String} \to \text{String} \to \text{String} \\
\text{concatenation}''\ [\,]\qquad\quad \text{ys} \ = \ \text{ys} \\
\text{concatenation}''\ (\text{x} : \text{xs})\ \ \text{ys} \ = \ \text{x} : \text{concatenation}''\ \text{xs ys}
\end{array}
$$

- nur eine Definition:

$$
\begin{array}{l}
\text{concatenation} :: [\text{a}] \to [\text{a}] \to [\text{a}] \\
\text{concatenation}\ \ [\,]\qquad\quad \text{ys} \ = \ \text{ys} \\
\text{concatenation}\ \ (\text{x} : \text{xs})\ \ \text{ys} \ = \ \text{x} : \text{concatenation xs ys}
\end{array}
$$

Typvariablen und parametrisierte Typen

- Um polymorphen Funktionen einen Typ zuordnen zu können, werden Variablen verwendet, die als Platzhalter für beliebige Typen stehen:

Typvariablen

- Mit Typvariablen können für polymorphe Funktionen parametrisierte Typen gebildet werden:

```
length :: [a] → Int
length [ ]         =  0
length (x : xs)  =  length xs + 1
```

- Ist auch der Resultattyp mittels einer Typvariable beschrieben, dann bestimmt natürlich der Typ der aktuellen Parameter den Typ des Resultats:

```
> :t last
last :: [a] → a
```

```
> :t last [ True, False ]
last [ True, False ] :: Bool
```

Sichere Verwendung polymorpher Funktionen

```
concatenation :: [a] → [a] → [a]
concatenation [ ]      ys = ys
concatenation (x : xs) ys = x : concatenation xs ys
```

```
> concatenation [ True ] [ False, True, False ]
[True, False, True, False]

> concatenation "abc" "def"
"abcdef"

> concatenation "abc" [True]
   Couldn't match 'Char' against 'Bool'
      Expected type: Char
      Inferred type: Bool
   In the list element: True
   In the second argument of 'concatenation', namely '[True]'
```

Weitere Beispiele

```
drop :: Int → [Int] → [Int]
drop 0       xs       =  xs
drop n       [ ]      =  [ ]
drop (n + 1) (x : xs) =  drop n xs
```

```
drop :: Int → [a] → [a]
drop 0       xs       =  xs
drop n       [ ]      =  [ ]
drop (n + 1) (x : xs) =  drop n xs
```

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

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

```
fst :: (a, b) → a

head :: [a] → a

take :: Int → [a] → [a]

id :: a → a
```

Sichere Verwendung polymorpher Funktionen

$$\begin{array}{lcl}
\text{zip} :: [a] \to [b] \to [(a, b)] & & \\
\text{zip}\ (x : xs)\ (y : ys) & = & (x, y) : \text{zip}\ xs\ ys \\
\text{zip}\ xs \quad\quad\ ys & = & [\ ]
\end{array}$$

```
> zip "abc" [ True, False, True ]
[('a', True), ('b', False), ('c', True)]

> :t "abc"
"abc" :: [Char]

> :t [ True, False, True ]
[True, False, True] :: [Bool]

> :t [ ('a', True), ('b', False), ('c', True) ]
[('a', True), ('b', False), ('c', True)] :: [(Char, Bool)]
```

Polymorphe Datentypen

- Abstraktion möglich von:

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

- zu:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

- mit wie folgt getypten Datenkonstruktoren:

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

Polymorphe Datentypen

- Mögliche Werte für:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

sind etwa: Leaf 3 :: Tree Int
Node (Leaf 'a') 'b' (Leaf 'c') :: Tree Char

aber nicht: Node (Leaf 'a') 3 (Leaf 'c')

- Beispielfunktion:

```
height :: Tree a → Int
height (Leaf _)         =  0
height (Node t₁ _ t₂)  =  1 + max (height t₁)  (height t₂)
```

Polymorphe Typsynonyme und Typisomorphe

- Abstraktion genauso möglich für type und newtype:

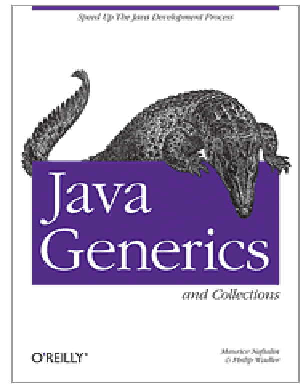
```
type PairList a b = [(a, b)]
```

```
newtype InfList a = Cons (a, InfList a)
```

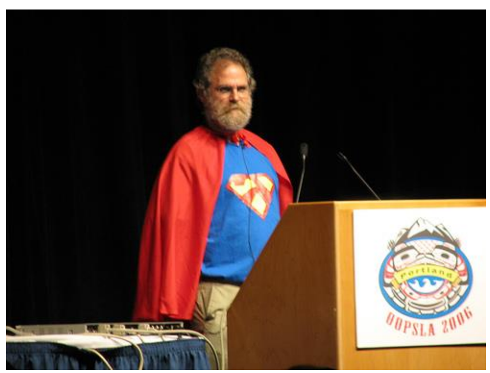
Parametrische Polymorphie in anderen Programmiersprachen

- Nun mögen Sie meinen:
  „Hmm, diese Art Polymorphie kann Java (oder …) auch.“

- Ja, das stimmt, und zu verdanken ist das zu gutem Teil der Arbeit von Phil Wadler:



also known as LAMBDA MAN:



Exkurs: Semantische Konsequenzen parametrischer Polymorphie

- Typen mit Typvariablen implizieren (in einer „puren“ Sprache wie Haskell) nicht-triviale Einschränkungen des Verhaltens der entsprechenden Funktionen.

- Als einfaches Experiment, überlegen Sie sich mal jeweils drei Funktionen folgender Typen:
  - $a \to [a]$
  - $Int \to a \to [a]$
  - $(a, b) \to a$
  - $(a, a) \to a$
  - a
  - $a \to a$
  - $[a] \to a$
  - $[a] \to Int$
  - $[a] \to [a]$

- Jenseits dieser Art „Charakterisierungen“ auch praktischer angelegte Konsequenzen, zum Beispiel, für jede <u>beliebige</u> Funktion $fun :: [a] \to [a]$ gilt:

$$\boxed{fun\ [\ g\ x \mid x \leftarrow xs\ ] \ = \ [\ g\ y \mid y \leftarrow fun\ xs\ ]}$$

Deskriptive Programmierung

Ad-hoc Polymorphie

Vordefinierte Typklassen, insbesondere automatisches „deriving“

- Das freizügige Einführen immer neuer Typen mag zunächst unattraktiv erscheinen, da man jeweils auch bestimmte Funktionalität (neu)-implementieren müsste (für Ein- und Ausgabe, für „Rechnen“ auf Aufzählungstypen, …).

- Diese Sorgen erübrigen sich jedoch durch Mechanismen für generische Funktionalität, zum Beispiel:

```
data Color  =  Red | Green | Blue | White | Black  deriving (Enum, Bounded)

allColors  =  [minBound .. maxBound] :: [Color]
```

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr  deriving (Read, Show, Eq)
```

…

- Funktioniert für data und für newtype. (… während für type nicht sinnvoll – Warum?)

Vordefinierte Typklassen, Instanzdefinitionen von Hand

Am einfachsten erklärt durch Beispiele:

```
data Color  =  Red | Green | Blue | White | Black  deriving (Enum, Bounded)

instance Show Color where
  show Red     =  "rot"
  show Green  =  "gruen"
  …
```

```
newtype Rat  =  Rat (Int, Int)

instance Show Rat where
  show (Rat (n, m))  =  show n ++ " / " ++ show m
```

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

instance Show Expr where
  show (Lit n)          =  "Lit "  ++  show n  ++  "; "
  show (Add e1 e2)  =  show e1  ++  show e2  ++  "Add; "
  show (Mul e1 e2)  =  show e1  ++  show e2  ++  "Mul; "
```

Natürlich dürfen auch beliebige andere Funktionen aufgerufen werden, nicht nur die gerade definierte (auf anderem oder dem selben Typ).

Zusammenspiel mit parametrischer Polymorphie

- Wir hatten Typvariablen benutzt, um auszudrücken, dass eine bestimmte Funktionalität etwa nicht vom Typ der Elemente einer Liste abhängt:

```
length :: [a] → Int
length [ ]        =  0
length (x : xs)  =  length xs + 1
```

- Wie ist das nun zum Beispiel mit show?

- Sicher wollen wir nicht etwa schreiben:

```
instance Show [Int] where
  show [ ]        =  "[ ]"
  show (i : is)  =  … show i … show is …

instance Show [Color] where
  show [ ]         =  "[ ]"
  show (c : cs)  =  … show c … show cs …
```

Zusammenspiel mit parametrischer Polymorphie

- Parametrisierung über den Elementtyp, aber mit explizitem Constraint:

```
instance Show a ⇒ Show [a] where
  show [ ]        =  "[ ]"
  show (x : xs)  =  … show x … show xs …
```

- Ein solcher Constraint kann auch Abhängigkeit zu einer <u>anderen</u> Typklasse ausdrücken:

```
instance Show a ⇒ Eq a where
  x == y   =   show x == show y
```

- Und auf ganz natürliche Weise können Constraints auch in Typsignaturen „normaler“ Funktionen auftauchen:

```
elem :: Eq a ⇒ a → [a] → Bool
elem x [ ]         =   False
elem x (y : ys)  =   x == y  ||  elem  x  ys
```