Typen

Typen

Wie Java ist Haskell statisch typisiert. In gültigen Programmen

- hat jeder Ausdruck einen Typ
- werten Ausdrücke immer zu Werten ihres Typs aus

Schreibweise: *e* `::` *t* falls Ausdruck *e* von Typ *t*

Funktionstypen: *s* `->` *t* ist der Typ von Funktionen von *s* nach *t*

```
'a' :: Char
not :: Bool -> Bool
3 :: Integer
(\x -> x*x) :: Double -> Double
```

```
True :: Bool
isDigit :: Char -> Bool
3 :: Double
```

Untypisierbare Ausdrücke erzeugen Übersetzerfehler:

```
3 + 'a'
not (True > 3)
length 5
\x -> if (x > 3) then x * x else 'a'
```

Typen

Typ von Variablen und Funktionen bzw. Methodensignatur

Java: manuelle Deklaration notwendig

```
boolean isDigit(char c);
char c = 'a';
```

Haskell: automatisch inferiert (fast immer),
manuelle Deklaration optional

Typabfrage in `ghci` per Befehl `:t`

```
*Main> :t isDigit
isDigit :: Char -> Bool
```

```
*Main> :t isDigit 'a'
isDigit 'a' :: Bool
```

```
*Main> :t let c = 'a' in c
let c = 'a' in c :: Char
```

vordefinierte Typen

Basis-Typen:

`Int`	ganze Zahlen, mindestens $\left[-2^{29}, 2^{29}-1\right]$
`Integer`	ganze Zahlen beliebiger Größe
`Float`	Fließkommazahlen einfacher Präzision
`Double`	Fließkommazahlen doppelter Präzision
`Bool`	Wahrheitswerte
`Char`	Unicode-Zeichen

strukturierte Typen: Listen, Tupel, Brüche, Summen-Typen, ...

weitere Typen: In den Haskell Hierarchical Libraries

Polymorphe Typen

Listen verschiedener Typen:

```
[True,False,False]            ::  [Bool]
['a','b','c']                 ::  [Char]
[['a','b'],[]]                ::  [[Char]]
[isDigit, (\c -> (c=='a'))]   ::  [Char -> Bool]
```

Listen-Typ ist polymorph:

- `[t]` ist der Typ von Listen mit Elementen vom Typ `t` (für alle Typen `t`)
- Typvariable `t` steht auch für Nicht-Basistypen

Sichtweisen:

- Typvariablen parametrisieren polymorphe Typen
- Typkonstruktoren wie `[ ]` erzeugen neue Typen aus bestehenden

Funktionstypen

Funktionstyp ist polymorph:

`s -> t` ist der Typ aller Funktionen von `s` nach `t` (für alle Typen `s,t`)

Typen mehrstelliger Funktionen:

```
f x y = x * y          f :: Integer -> Integer -> Integer
g a b x = a * x + b    g :: Integer -> Integer -> Integer -> Integer
app left right = ...   app :: [t] -> [t] -> [t]
```

Typen eingebauter Operatoren:

```
(+)  :: Integer -> Integer -> Integer
(&&) :: Bool -> Bool -> Bool
(<=) :: Integer -> Integer -> Bool
(:)  ::  t -> [t] -> [t]
```

Infixnotation

Operatoren: `3+4, 4<=7, ...`

Binäre Funktionen: `l `app` r` Backquote-Notation

Infixdeklarationen: **`infix`** *n s*

- Bindungsstärke *n* (Vergleich: + 6, * 7)
- Operatorsymbol *s*

Toleranter Vergleichssoperator: für Gleitkommazahlen

```
infix 4 =:=
a =:= b = abs (a-b) < 0.001
```

Präfixnotation	Infixnotation
`(+) 3 4`	`3+4`
`(<=) 4 7`	`4<=7`
`app left right`	`left `app` right`
`mod` `n m`	`m `**`mod`**` n`
`(=:=) 0.9999 1`	`0.9999 =:= 1`

Tupel

Tupel-Typen:

```
(3,True) :: (Integer,Bool)
(not,7) :: (Bool -> Bool,Integer)
```

Tupel-Konstruktor:

```
(,) :: s -> t -> (s, t)
```

Tupel-Destruktoren:

```
fst :: (s, t) -> s        fst (3,True) ⇒ 3
snd :: (s, t) -> t        snd (3,True) ⇒ True
```

Funktionen auf Tupeln:

```
f :: (Double,Double) -> Double  ⇔   f :: (Double,Double) -> Double
f y = (fst y) * (snd y)             f (a, x) = a * x
```

- Funktion mit einem Argument vom Typ `(Double,Double)`
- Definition: Destruktoren oder Pattern-Matching

Vergleich: Typen in Java

Haskell	Java
parametrischer Polymorphismus	**Generics**
`[t]`	`LinkedList<T>`
`[1,2]::[Int]`	`LinkedList<Integer> l;` `l.add(1); l.add(2);`
Funktionstypen	**Signaturen**
`foo :: Int -> Int`	`Integer foo(Integer x);`
`foo :: s -> t -> t`	`<S,T> T foo(S s,T t);`
`foo :: (s -> t) -> s -> t`	—

nur in Java: Vererbung ermöglicht inhomogene Listen

~~`['a',True]`~~	`LinkedList<Object> l;` `l.add('a'); l.add(true);`

Typinferenz

Errechnen der Typen durch den Compiler, statt Deklaration.

Vorteile:

- kompakte Programme
- trotzdem typsicher

Beispielprogramm: Konvertierung von Zeichenketten nach Zahlen

```
digitToInt c
   |  (isDigit  c) = (ord c) - (ord '0')

digitsToInt s = fromRev (rev s)
     where  fromRev []     =  0
            fromRev (d:ds) = 10 * (fromRev ds) + (digitToInt d)
```

Typinferenz: auch von Funktionstypen

```
*Main> :t ord
ord :: Char -> Int

*Main> :t digitToInt
digitToInt :: Char -> Int
```

```
*Main> :t digitsToInt
digitsToInt :: [Char] -> Int
```

Inferenzalgorithmus: im Theorie-Teil

Typdeklaration

Trotz Typinferenz:

- Manuelle Typdeklarationen möglich

```
isSpace :: Char -> Bool
isSpace x = (x == ' ')
```

Typdeklarationen

- Erhöhen die Lesbarkeit des Quelltextes
- Helfen bei der Lokalisierung von Programmierfehlern

$\Longrightarrow$ sind dringend zu empfehlen!

Typsynonyme

Typsynonyme: neuer Name, kein neuer Typ

- Zeichenketten: `type String = [Char]`
- Erhöhen Lesbarkeit von Typannotationen

Was machen folgende Funktionen?

```
passed :: [Double] -> Bool
passed a = ...

graduates :: [([Char], [Double])] -> [[Char]]
graduates e = ...
```

Typsynonyme

Typsynonyme: neuer Name, kein neuer Typ

- Zeichenketten: `type String = [Char]`
- Erhöhen Lesbarkeit von Typannotationen

Was machen folgende Funktionen?

```
passed :: Assessment -> Bool
passed a = ...

graduates :: Examination -> [Student]
graduates e = ...

type Student    = String
type Assessment = [Double]
type Submission = (Student,Assessment)
type Examination = [Submission]
```

Typen bei der Fehlersuche

Typen helfen bei Fehlersuche:

```
isDigit c = isIn c "0123456789"
```

- Korrekte Funktionsdefinition
- Macht aber nicht, was sie soll:

```
*Main> isDigit '3'
  Couldn't match expected type `[[Char]]'
  against inferred type `Char'
```

???

Typen bei der Fehlersuche

Typen helfen bei Fehlersuche:

```
isDigit :: Char -> Bool
isDigit c = isIn c "0123456789"
```

- Beabsichtigten Typ der Funktion deklarieren
- Fehler schon bei der Funktionsdefinition:

```
Couldn't match expected type `[t]' against inferred type `Char'
In the first argument of `isIn', namely `c'
```

Typen bei der Fehlersuche

Typen helfen bei Fehlersuche:

```
isDigit :: Char -> Bool
isDigit c = isIn c "0123456789"
```

- Beabsichtigten Typ der Funktion deklarieren
- Fehler schon bei der Funktionsdefinition:

```
Couldn't match expected type `[t]' against inferred type `Char'
In the first argument of `isIn', namely `c'
```

- `isIn` erwartet die Liste zuerst, dann erst das Zeichen!

Korrekte Definition:

```
isDigit :: Char -> Bool
isDigit c = isIn "0123456789" c
```

Mengen

Datenstruktur: Menge besteht aus

- Typ `Set t = ...` sowie Funktionen zum
- Iterieren (Folds), Einfügen, Löschen, Mengenvergleich, ...

Einfachste Implementierung: Listen

`[4,2,1]`	$\cong$	$\{1,2,4\}$	`[]`	$\cong$	$\emptyset$
`[1,2,4]`	$\cong$	$\{1,2,4\}$	`[1,2,1,4]`	$\cong$	???

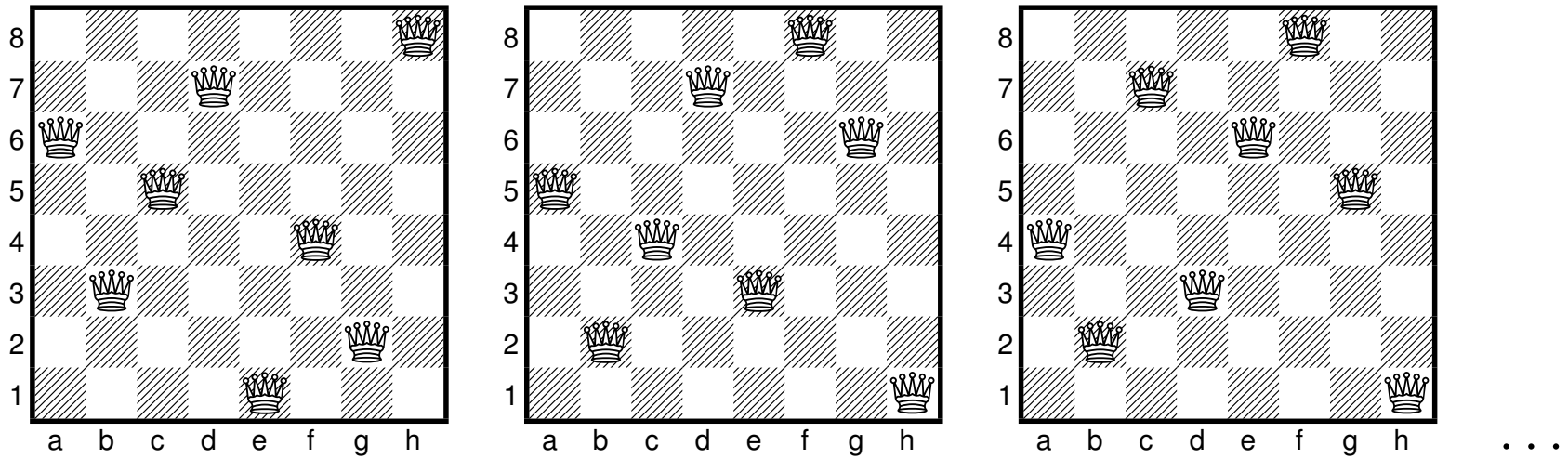
```
type Set t = [t]

insert x s = if (isIn s x) then s else x:s

delete x []     = []
delete x (y:ys) = if (x==y) then ys else delete x ys

fold = foldr
```

- Listengleichheit `s1 == s2` nicht geeignet als Mengengleichheit
- Doppelte Einträge erlauben? $\Rightarrow$ Multi-Mengen
- Weitere Mengenoperationen: per `fold`

Beispiel funktionaler Programmierung

Backtracking

8-Damen Problem Platziere 8 Damen, sodass keine eine andere bedroht



Lösungverfahren: Backtracking-Suche

- Beginne mit zulässiger Anfangskonfiguration – leeres Brett
- Konstruiere und durchsuche Baum von zulässigen Folgekonfigurationen – per Platzierung weiterer Damen, ohne vorhandene zu bedrohen
- Bis Lösungskonfiguration gefunden – 8 Damen sind auf dem Brett

Backtracking in Haskell

Umsetzung: Kombination dreier Funktionen mittels `map` und `filter`

- `successors :: Conf -> [Conf]`
  Bestimmt alle möglichen Folgekonfigurationen
- `legal :: Conf -> Bool`
  Prüft, ob Konfiguration zulässig ist
- `solution :: Conf -> Bool`
  Prüft, ob (zulässige) Konfiguration Lösung ist

Funktion `backtrack` berechnet Liste aller Lösungen:

```
backtrack :: Conf -> [Conf]
backtrack conf =
   if (solution conf) then [conf]
   else flatten (map backtrack (filter legal (successors conf)))
```

Lazyness!

Backtracking: Liste aller Lösungen

```
backtrack :: Conf -> [Conf]
backtrack conf =
   if (solution conf) then [conf]
   else flatten (map backtrack (filter legal (successors conf)))

solutions = backtrack initial
```

- Nur eine Loesung benötigt: `head solutions`
- Lazyness: Backtracking nur solange, bis erste Lösung gefunden

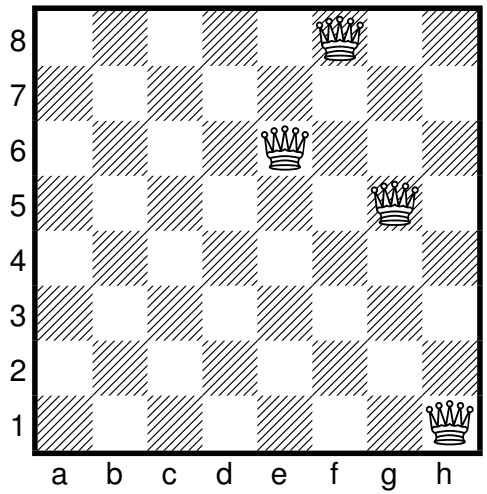
Damenproblem in Haskell

Konfigurationen: Liste von Zeilenpositionen

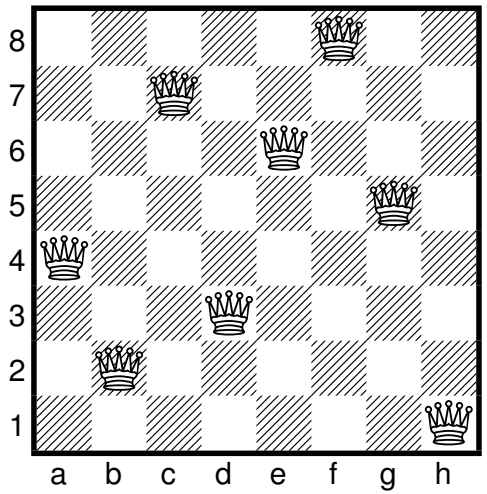
- Maximal eine Dame in jeder Spalte
- Konfiguration: Liste der Zeilenposition der ersten $n \leq 8$ Damen

$\Rightarrow$ viele illegale Stellungen nicht einmal darstellbar!

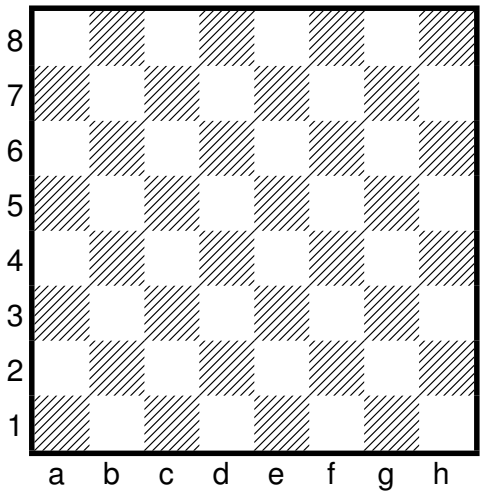
```
type Conf = [Integer]
```



[6,8,5,1]



[4,2,7,3,6,8,5,1]



[]

Damenproblem in Haskell

Folgekonfigurationen: Platziere weitere Dame in beliebiger Zeile

```
successors :: Conf -> [Conf]
successors board = map (:board) [1..8]
```

Beispiel-Auswertung

```
successors [6,8,5,1]  ⇒+  [[1,6,8,5,1],[2,6,8,5,1],...,[8,6,8,5,1]]
```

```
backtrack :: Conf -> [Conf]
backtrack conf =
   if (solution conf) then [conf]
   else flatten (map backtrack (filter legal (successors conf)))
```

Damenproblem in Haskell

Legale Konfigurationen: Neue Dame bedroht keine bestehenden

- Nutze aus: bestehende Damen zulässig

```
threatens :: Int -> Int -> Conf -> Bool
threatens diag col []             = False
threatens diag col (row:rest)     =
   (row==col+diag) || (row==col-diag) ||
   (row==col) || threatens (diag+1) col rest

legal :: Conf -> Bool
legal [] = True
legal (row:rest) = (not (threatens 1 row rest))
```

Beispiel-Auswertung

```
 filter legal (successors [6,8,5,1])   ⇒+   [[3,6,8,5,1],[4,6,8,5,1]]
backtrack :: Conf -> [Conf]
backtrack conf =
   if (solution conf) then [conf]
   else flatten (map backtrack (filter legal (successors conf)))
```

Damenproblem in Haskell

Lösungs-Konfigurationen: 8 Damen auf dem Brett

```
solution :: Conf -> Bool
solution board = (length board) == 8
```

Start-Konfiguration: leeres Brett

```
queensSolutions :: [Conf]
queensSolutions =  backtrack []
```

Lösungsanzeige:

```
take 2 queensSolutions   ⇒+   [[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1]]
length queensSolutions   ⇒+   92
```

```
backtrack :: Conf -> [Conf]
backtrack conf =
   if (solution conf) then [conf]
   else flatten (map backtrack (filter legal (successors conf)))
```