

Programmierparadigmen

Tutorium: Lazyness, Streams

Prof. Dr.-Ing. Gregor Snelting | WS 2012/2013

LEHRSTUHL PROGRAMMIERPARADIGMEN

```

prime :: Integer -> Bool
prime n = (n>=2) && not (any (divides n) [2..n-1])
  where divides n m = n `mod` m == 0

queens :: Conf -> [Conf]
queens board =
  if (solution board) then [board]
  else flatten (map damens (filter legal (successors board) NUMTHRS; i++))

primes :: [Integer]
primes = sieve [2..]
  where sieve [] = []
        sieve (p : xs) = p : sieve [x | x <- xs, x > p]

qsort :: [Integer] -> [Integer]
qsort [] = []
qsort (p:ps) = (qsort [x | x <- ps, x <= p])
              ++ p: (qsort [x | x <- ps, x > p])

bal :: RedBlackTree t -> RedBlackTree t
bal (Node Black (Node Red (Node Red a x b) y (Node Black c d)) e) =
  (Node Red (Node Black a x b) y (Node Black c d)) `thread`
  (Node Black e)
  where (Node Black c d) = bal (Node Red (Node Black a x b) y (Node Black c d))
        (Node Black e) = bal (Node Black e)

```

$$\begin{array}{c}
 \Gamma(f) = \\
 \Gamma(f) = \forall \tau. \tau \rightarrow \text{int} \quad \Gamma \vdash f : \\
 \Gamma \vdash f : \text{int} \rightarrow \text{int} \quad \Gamma \vdash f : \forall \tau. \tau \rightarrow \text{int} \\
 \Gamma
 \end{array}$$

$$\Gamma \vdash \text{let } f = \lambda x. 2 \text{ in } f(f \text{ tr})$$

Streams: unendliche Listen, üblicherweise erzeugt durch *Kombination* (auch rekursiv!) unendlicher Listen

```
ones = 1 : ones                nats = 0 : (map (+1) nats)
```

- `zipWith` kombiniert zwei streams

```
nats = 0 : (zipWith (+) ones nats)
```

ones	1	1	1	1	...
nats	0				
<hr/>					
nats	0				

- Streams nutzen *laziness* der Kombinatoren

```
map f []      = []                zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
map f (x:xs) = f x : map f xs    zipWith f l1      l2      = []
```

Zur Berechnung des nächstens Listenelements $f\ x$ bzw. $f\ x\ y$ muss nur das jeweils erste Element der Eingabelisten bestimmt werden! (nicht: alle Elemente der Eingabelisten)

Streams: unendliche Listen, üblicherweise erzeugt durch *Kombination* (auch rekursiv!) unendlicher Listen

```
ones = 1 : ones                nats = 0 : (map (+1) nats)
```

■ zipWith kombiniert zwei streams

```
nats = 0 : (zipWith (+) ones nats)
```

ones	1	1	1	1	...
nats	0	1			
<hr/>					
nats	0	1			

■ Streams nutzen *laziness* der Kombinatoren

```
map f []      = []                zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
map f (x:xs) = f x : map f xs    zipWith f l1    l2      = []
```

Zur Berechnung des nächstens Listenelements $f\ x$ bzw. $f\ x\ y$ muss nur das jeweils erste Element der Eingabelisten bestimmt werden! (nicht: alle Elemente der Eingabelisten)

Streams: unendliche Listen, üblicherweise erzeugt durch *Kombination* (auch rekursiv!) unendlicher Listen

```
ones = 1 : ones                nats = 0 : (map (+1) nats)
```

■ zipWith kombiniert zwei streams

```
nats = 0 : (zipWith (+) ones nats)
```

ones	1	1	1	1	...
nats	0	1	2		
<hr/>					
nats	0	1	2		

■ Streams nutzen *laziness* der Kombinatoren

```
map f []      = []                zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
map f (x:xs) = f x : map f xs    zipWith f l1      l2      = []
```

Zur Berechnung des nächstens Listenelements $f\ x$ bzw. $f\ x\ y$ muss nur das jeweils erste Element der Eingabelisten bestimmt werden! (nicht: alle Elemente der Eingabelisten)

Streams: unendliche Listen, üblicherweise erzeugt durch *Kombination* (auch rekursiv!) unendlicher Listen

```
ones = 1 : ones           nats = 0 : (map (+1) nats)
```

- `zipWith` kombiniert zwei streams

```
nats = 0 : (zipWith (+) ones nats)
```

ones	1	1	1	1	...
nats	0	1	2	3	
<hr/>					
nats	0	1	2	3	

- Streams nutzen *laziness* der Kombinatoren

```
map f []      = []           zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
map f (x:xs) = f x : map f xs zipWith f l1      l2      = []
```

Zur Berechnung des nächstens Listenelements $f\ x$ bzw. $f\ x\ y$ muss nur das jeweils erste Element der Eingabelisten bestimmt werden! (nicht: alle Elemente der Eingabelisten)

Streams: unendliche Listen, üblicherweise erzeugt durch *Kombination* (auch rekursiv!) unendlicher Listen

```
ones = 1 : ones                nats = 0 : (map (+1) nats)
```

- `zipWith` kombiniert zwei streams

```
nats = 0 : (zipWith (+) ones nats)
```

ones	1	1	1	1	...
nats	0	1	2	3	...
<hr/>					
nats	0	1	2	3	...

- Streams nutzen *laziness* der Kombinatoren

```
map f []      = []                zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
map f (x:xs) = f x : map f xs    zipWith f l1      l2      = []
```

Zur Berechnung des nächstens Listenelements $f\ x$ bzw. $f\ x\ y$ muss nur das jeweils erste Element der Eingabelisten bestimmt werden! (nicht: alle Elemente der Eingabelisten)

Interleaving: Kombinator zum verschmelzen zweier Streams

```
interleave :: [t] -> [t] -> [t]
interleave [] ys = ys
interleave xs [] = xs
interleave (x : xs) (y : ys) = x : y : (interleave xs ys)
```

■ Liste aller ganzen Zahlen

```
integers = 0 : (interleave [1..] (map (*(-1)) [1..]))
```

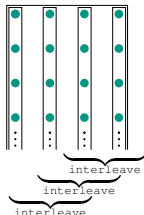
[1..]		1,	2,	3,	4,	...
map (*(-1)) [1..]		-1,	-2,	-3,	-4,	...
<hr/>						
integers	0,	1,-1,	2,-2,	3,-3,	4,-4	...

Interleaving: Kombinator zum verschmelzen zweier Streams

```
interleave :: [t] -> [t] -> [t]
interleave [] ys = ys
interleave xs [] = xs
interleave (x : xs) (y : ys) = x : y : (interleave xs ys)
```

■ Kombination *endlich* vieler Streams

```
interleaveAll :: [[t]] -> [t]
interleaveAll lists = foldr interleave [] lists
```



Interleaving: Kombinator zum verschmelzen zweier Streams

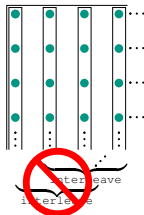
```
interleave :: [t] -> [t] -> [t]
interleave [] ys = ys
interleave xs [] = xs
interleave (x : xs) (y : ys) = x : y : (interleave xs ys)
```

- **nicht:** Kombination *unendlich* vieler Streams

```
interleaveAll :: [[t]] -> [t]
interleaveAll lists = foldr interleave [] lists
```

Pattern-Matching ($y:ys$) zwingt uns

- im äußersten `interleave` festzustellen, ob zweites `interleave` zu `[]` auswertet
- im zweiten `interleave` festzustellen, ob drittes `interleave` zu `[]` auswertet
- ...



Interleaving: Kombinator zum verschmelzen zweier Streams

```
interleave :: [t] -> [t] -> [t]
interleave [] ys = ys
interleave (x:xs) ys = x : interleave ys xs
```

■ Abhilfe **speziell** für `interleave`: alternative Definition

```
interleaveAll :: [[t]] -> [t]
interleaveAll lists = foldr interleave [] lists
```

Neues `interleave` ist

- komplett *lazy* im rechten Argument!
- ⇒ kein frühzeitiges Auswerten der unendlichen Liste von Streams

