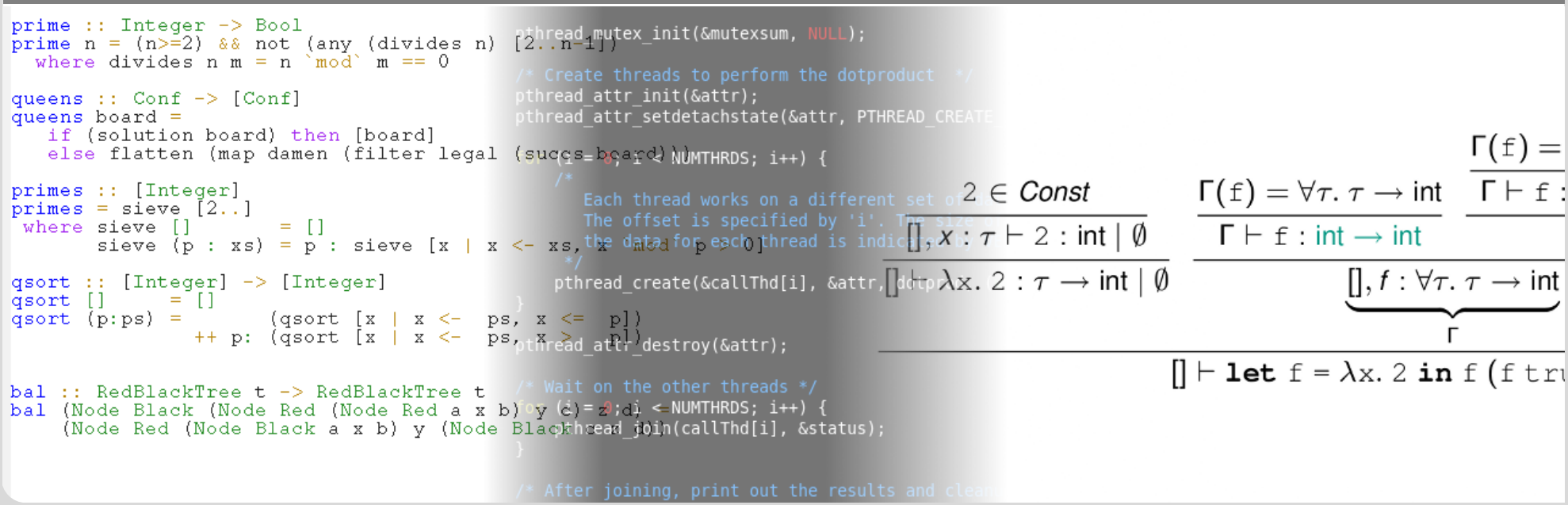
Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: Lazyness, Streams

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

LEHRSTUHL PROGRAMMIERPARADIGMEN



Streams

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
-------------------------------------
nats    0
```

- Streams nutzen *laziness* der Kombinatoren

```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
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

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
-------------------------------------
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

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
-------------------------------------
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

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
-------------------------------------
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

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     ...
-------------------------------------
nats   0    1   2   3   ...
```

- Streams nutzen *laziness* der Kombinatoren

```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
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 von Streams

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,	...
`integers`	0,	1,-1,	2,-2,	3,-3,	4,-4	...

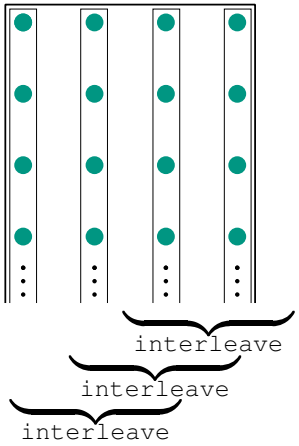
Interleaving von Streams

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 von Streams

Interleaving: Kombinator zum verschmelzen zweier Streams

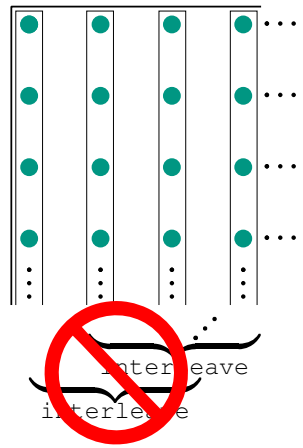
```
interleave :: [t] -> [t] -> [t]
interleave [] ys = ys
interleave xs [] = xs
interleave (x : xs) (y : ys) = x : y : (interleave xs ys)
```

- 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 von Streams

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!

$\Rightarrow$ kein frühzeitiges Auswerten der unendlichen Liste von Streams