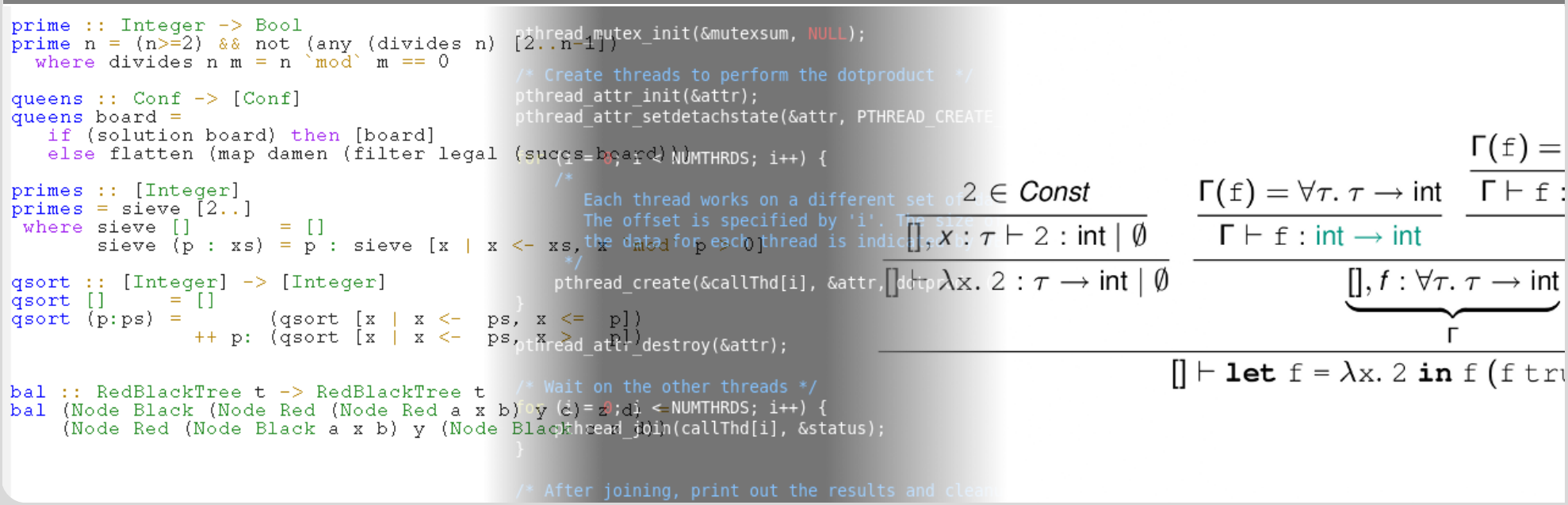
Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: Erste Schritte in Haskell

Prof. Dr.-Ing. Gregor Snelting | WS 2012/13

LEHRSTUHL PROGRAMMIERPARADIGMEN



Größter gemeinsamer Teiler per *remainder*

Sei $r > 0$ der Rest der Division a/b zweier natürlicher Zahlen $a, b > 0$.
Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, r*

- Definiere zweistellige Haskell-Funktion `gcdr` zur Berechnung des größten gemeinsamen Teilers zweier Zahlen größer 0

Größter gemeinsamer Teiler per *remainder*

Sei $r > 0$ der Rest der Division a/b zweier natürlicher Zahlen $a, b > 0$.
Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, r*

- Definiere zweistellige Haskell-Funktion `gcdr` zur Berechnung des größten gemeinsamen Teilers zweier Zahlen größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, r)$,
außerdem: $gcd\,(a, b) = b$ falls *b teilt a*

Größter gemeinsamer Teiler per *remainder*

Sei $r > 0$ der Rest der Division a/b zweier natürlicher Zahlen $a, b > 0$.
Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, r*

- Definiere zweistellige Haskell-Funktion `gcdr` zur Berechnung des größten gemeinsamen Teilers zweier Zahlen größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, r)$,
außerdem: $gcd\,(a, b) = b$ falls *b teilt a*

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

Größter gemeinsamer Teiler per *subtraction*

Sei $a > b > 0$. Dann gilt für $x \in \mathbb{N}$:

$$x \textit{ ist gemeinsamer Teiler von } a, b \text{ genau dann wenn } x \textit{ ist gemeinsamer Teiler von } b, (a - b)$$

Sei $b > a > 0$. Dann gilt für $x \in \mathbb{N}$:

$$x \textit{ ist gemeinsamer Teiler von } a, b \text{ genau dann wenn } x \textit{ ist gemeinsamer Teiler von } a, (b - a)$$

- Definiere zweistellige Haskell-Funktion `gcds` zur Berechnung des größten gemeinsamen Teilers zweier `Zahlen` größer 0

Größter gemeinsamer Teiler per *subtraction*

Sei $a > b > 0$. Dann gilt für $x \in \mathbb{N}$:

x *ist gemeinsamer Teiler von* a, b genau dann wenn x *ist gemeinsamer Teiler von* $b, (a - b)$

Sei $b > a > 0$. Dann gilt für $x \in \mathbb{N}$:

x *ist gemeinsamer Teiler von* a, b genau dann wenn x *ist gemeinsamer Teiler von* $a, (b - a)$

- Definiere zweistellige Haskell-Funktion `gcds` zur Berechnung des größten gemeinsamen Teilers zweier `Zahlen` größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, a - b)$ bzw.
$gcd\,(a, b) = gcd\,(a, b - a)$, außerdem: $gcd\,(a, a) = a$

Größter gemeinsamer Teiler per *subtraction*

Sei $a > b > 0$. Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von b, (a − b)*

Sei $b > a > 0$. Dann gilt für $x \in \mathbb{N}$:

x ist gemeinsamer Teiler von a, b genau dann wenn *x ist gemeinsamer Teiler von a, (b − a)*

- Definiere zweistellige Haskell-Funktion `gcds` zur Berechnung des größten gemeinsamen Teilers zweier `Zahlen` größer 0

Lösungsidee: $gcd\,(a, b) = gcd\,(b, a - b)$ bzw. $gcd\,(a, b) = gcd\,(a, b - a)$, außerdem: $gcd\,(a, a) = a$

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

Terminierung

Größter gemeinsamer Teiler: rekursive mathematische Gleichungen zur Funktion waren *eindeutig* erfüllbar – nicht immer der Fall!

Als Berechnungsvorschift : nicht zwingend terminierend!

Gleichung	Erfüllbar	Eindeutig ($n \in \mathbb{N}$)	Auswertung
$f(n) = \begin{cases} 1 & n = 0 \\ f(n-1) \cdot n & \textit{sonst} \end{cases}$			$f(3) \Rightarrow$
$f(n) = \begin{cases} 0 & n = 0 \\ f(n) + 1 & \textit{sonst} \end{cases}$			$f(1) \Rightarrow$
$f(n) = \begin{cases} 0 & n = 0 \\ f(n+1) & \textit{sonst} \end{cases}$			$f(1) \Rightarrow$
$f(x,y) = \begin{cases} 0 & x = 0 \\ f(x-1, f(x,y)) & \textit{sonst} \end{cases}$			$f(1,42) \Rightarrow$

Terminierung

Größter gemeinsamer Teiler: rekursive mathematische Gleichungen zur Funktion waren *eindeutig* erfüllbar – nicht immer der Fall!

Als Berechnungsvorschift : nicht zwingend terminierend!

Gleichung	Erfüllbar	Eindeutig ($n \in \mathbb{N}$)	Auswertung
$f(n) = \begin{cases} 1 & n = 0 \\ f(n-1) \cdot n & \textit{sonst} \end{cases}$			$f(3) \Rightarrow \ldots \Rightarrow 6$
$f(n) = \begin{cases} 0 & n = 0 \\ f(n) + 1 & \textit{sonst} \end{cases}$			$f(1) \Rightarrow 1 + f(1) \Rightarrow \ldots$
$f(n) = \begin{cases} 0 & n = 0 \\ f(n+1) & \textit{sonst} \end{cases}$			$f(1) \Rightarrow f(2) \Rightarrow \ldots$
$f(x,y) = \begin{cases} 0 & x = 0 \\ f(x-1, f(x,y)) & \textit{sonst} \end{cases}$			$f(1,42) \Rightarrow f(0, f(1,42)) \Rightarrow f(0, f(0, f(1,42))) \Rightarrow \ldots$ *oder* $f(1,42) \Rightarrow f(0, f(1,42)) \Rightarrow 0$

Terminierung

Größter gemeinsamer Teiler: rekursive mathematische Gleichungen zur Funktion waren *eindeutig* erfüllbar – nicht immer der Fall!

Als Berechnungsvorschift : nicht zwingend terminierend!

Gleichung		Erfüllbar	Eindeutig ($n \in \mathbb{N}$)	Auswertung
$f(n) = \begin{cases} 1 & n = 0 \\ f(n-1) \cdot n & \textit{sonst} \end{cases}$	$= n!$	✔	✔	$f(3) \Rightarrow \ldots \Rightarrow 6$
$f(n) = \begin{cases} 0 & n = 0 \\ f(n) + 1 & \textit{sonst} \end{cases}$		✖	✖	$f(1) \Rightarrow 1 + f(1) \Rightarrow \ldots$
$f(n) = \begin{cases} 0 & n = 0 \\ f(n+1) & \textit{sonst} \end{cases}$	$f_k(n) = \begin{cases} 0 & n = 0 \\ k & \textit{sonst} \end{cases}$	✔	✖	$f(1) \Rightarrow f(2) \Rightarrow \ldots$
$f(x,y) = \begin{cases} 0 & x = 0 \\ f(x-1, f(x,y)) & \textit{sonst} \end{cases}$	$= 0$	✔	✔	$f(1,42) \Rightarrow f(0, f(1,42)) \Rightarrow f(0, f(0, f(1,42))) \Rightarrow \ldots$ *oder* $f(1,42) \Rightarrow f(0, f(1,42)) \Rightarrow 0$

Terminierung

Hinreichend: f terminiert für Eingaben $x \in S$, falls

- In jedem rekursiven Aufruf $f\ y$ ist $y \in S$, und "kleiner" als x:
$$x <_f y$$
- Es gibt in S keine bezüglich $<_f$ unendlich-oft absteigenden Ketten
$$\ldots <_f x_3 <_f x_2 <_f x_1$$

$x, y \in S$: Erfüllung/Erhaltung von Terminierungs-*Invariante*

Fakultätsfunktion: Argument wird kleiner

```
fak n   = if (n==0) then 1 else n * fak (n-1)
```

Terminierung

Hinreichend: f terminiert für Eingaben $x \in S$, falls

- In jedem rekursiven Aufruf $f\ y$ ist $y \in S$, und "kleiner" als x:
$$x <_f y$$
- Es gibt in S keine bezüglich $<_f$ unendlich-oft absteigenden Ketten
$$\ldots <_f x_3 <_f x_2 <_f x_1$$

$x, y \in S$: Erfüllung/Erhaltung von Terminierungs-*Invariante*

Fakultätsfunktion: Argument wird kleiner

```
fak n   = if (n==0) then 1 else n * fak (n-1)
```

$$S = \mathbb{N}, \quad n <_f m :\Leftrightarrow n < m$$

- $n \in \mathbb{N}, \neg n = 0 \quad \Rightarrow \quad n - 1 \in \mathbb{N} \wedge n - 1 <_f n$
- keine Ketten $\ldots < n_3 < n_2 < n_1$ in $\mathbb{N}$

Terminierung

Hinreichend: f terminiert für Eingaben $x \in S$, falls

- In jedem rekursiven Aufruf $f\ y$ ist $y \in S$, und "kleiner" als x:
$$x <_f y$$
- Es gibt in S keine bezüglich $<_f$ unendlich-oft absteigenden Ketten
$$\ldots <_f x_3 <_f x_2 <_f x_1$$

$x, y \in S$: Erfüllung/Erhaltung von Terminierungs-*Invariante*

mit Akkumulator: *erstes* Argument wird kleiner

```
fak n a = if (n==0) then a else fak (n-1) (n*a)
```

Terminierung

Hinreichend: f terminiert für Eingaben $x \in S$, falls

- In jedem rekursiven Aufruf $f\ y$ ist $y \in S$, und "kleiner" als x:
$$x <_f y$$
- Es gibt in S keine bezüglich $<_f$ unendlich-oft absteigenden Ketten
$$\ldots <_f x_3 <_f x_2 <_f x_1$$

$x, y \in S$: Erfüllung/Erhaltung von Terminierungs-*Invariante*

mit Akkumulator: *erstes* Argument wird kleiner

```
fak n a = if (n==0) then a else fak (n-1) (n*a)
```

$$S = \mathbb{N} \times \mathbb{Z}, \quad (n,a) <_f (m,b) \ :\Leftrightarrow\ n < m$$

- $(n,a) \in \mathbb{N}\times\mathbb{Z}, \neg n = 0 \quad \Rightarrow \quad (n-1, n\cdot a) \in \mathbb{N}\times\mathbb{Z} \wedge (n-1, n\cdot a) <_f (n,a)$
- keine Ketten $\ldots < n_3 < n_2 < n_1$ in $\mathbb{N}$

$\Rightarrow$ keine Ketten $\ldots <_f (n_3,a_3) <_f (n_2,a_2) <_f (n_1,a_1)$ in $\mathbb{N} \times \mathbb{Z}$

Terminierung *gcd*

Terminierung von `gcdr` Was wird kleiner?

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

$S =$
$(a, b) <_f (a', b') :\Leftrightarrow$

Terminierung von `gcds` Was wird kleiner?

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

$S =$
$(a, b) <_f (a', b') :\Leftrightarrow$

Terminierung *gcd*

Terminierung von `gcdr` zweites Argument wird kleiner

- denn: $r < b$

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

$$S = \\ (a, b) <_f (a', b') \ :\Leftrightarrow$$

Terminierung von `gcds` Was wird kleiner?

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

$$S = \\ (a, b) <_f (a', b') \ :\Leftrightarrow$$

Terminierung *gcd*

Terminierung von `gcdr` zweites Argument wird kleiner

- denn: $r < b$

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

$$S = \mathbb{N}_{>0} \times \mathbb{N}_{>0}$$
$$(a, b) <_f (a', b') \;:\Leftrightarrow\; b < b'$$

Terminierung von `gcds` Was wird kleiner?

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

$$S =$$
$$(a, b) <_f (a', b') \;:\Leftrightarrow$$

Terminierung *gcd*

Terminierung von `gcdr` zweites Argument wird kleiner

- denn: $r < b$

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

$$S = \mathbb{N}_{>0} \times \mathbb{N}_{>0}$$
$$(a, b) <_f (a', b') \;:\Leftrightarrow\; b < b'$$

Terminierung von `gcds` ein Argument wird kleiner, anderes bleibt gleich!

- beachte: $a - b \in \mathbb{N}_{>0}$ bzw. $b - a \in \mathbb{N}_{>0}$

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

$$S =$$
$$(a, b) <_f (a', b') \;:\Leftrightarrow$$

Terminierung *gcd*

Terminierung von `gcdr` zweites Argument wird kleiner

- denn: $r < b$

```
gcdr a b
  | r > 0      = gcdr b r
  | otherwise = b
 where r = a `mod` b
```

$$S = \mathbb{N}_{>0} \times \mathbb{N}_{>0}$$
$$(a, b) <_f (a', b') \;:\Leftrightarrow\; b < b'$$

Terminierung von `gcds` ein Argument wird kleiner, anderes bleibt gleich!

- beachte: $a - b \in \mathbb{N}_{>0}$ bzw. $b - a \in \mathbb{N}_{>0}$

```
gcds a b
  | a>b      = gcds (a-b) b
  | a<b      = gcds a (b-a)
  | a==b     = a
```

$$S = \mathbb{N}_{>0} \times \mathbb{N}_{>0}$$
$$(a, b) <_f (a', b') \;:\Leftrightarrow\; a + b < a' + b'$$

Ackermann-Funktion

Ackermann-Funktion: Terminierung??

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Spezialfall: $m = 2$: `ack 2 n = 2*n + 3  > (n+1)`

- innerer Aufruf `ack m (n-1)`: zweites Argument wird kleiner, erstes Argument bleibt gleich, aber:
- äußerer Aufruf `ack (m-1) (ack m (n-1))`: erstes Argument wird kleiner, aber zweites wächst!

Ackermann-Funktion

Ackermann-Funktion: Terminierung??

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Spezialfall: $m = 2$: `ack 2 n = 2*n + 3  > (n+1)`

- innerer Aufruf `ack m (n-1)`: zweites Argument wird kleiner, erstes Argument bleibt gleich, aber:
- äußerer Aufruf `ack (m-1) (ack m (n-1))`: erstes Argument wird kleiner, aber zweites wächst!
- wenn erstes Argument nicht kleiner wird, dann aber zweites!

Ackermann-Funktion

Ackermann-Funktion: Terminierung??

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Spezialfall: $m = 2$: `ack 2 n = 2*n + 3` $>$ `(n+1)`

- innerer Aufruf `ack m (n-1)`: zweites Argument wird kleiner, erstes Argument bleibt gleich, aber:
- äußerer Aufruf `ack (m-1) (ack m (n-1))`: erstes Argument wird kleiner, aber zweites wächst!
- wenn erstes Argument nicht kleiner wird, dann aber zweites!

Lexikographische Ordnung:

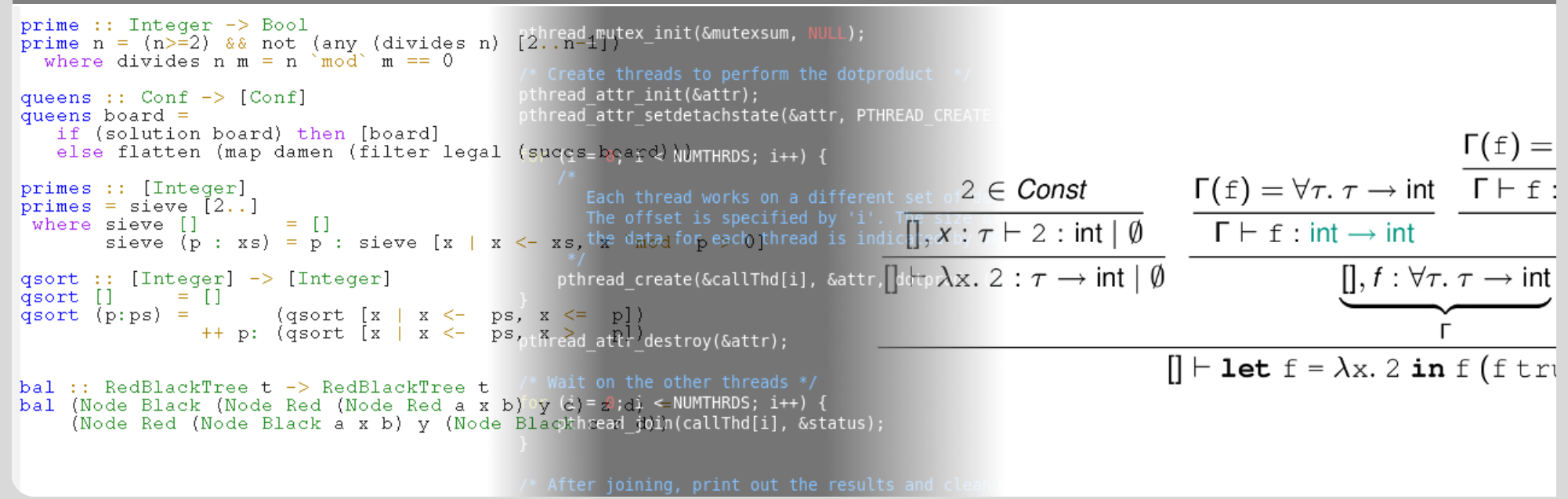
$$(m, n) <_{\mathtt{ack}} (m', n') \;:\Leftrightarrow\; m < m' \vee (m = m' \wedge n < n')$$
$$S = \mathbb{N} \times \mathbb{N}$$

Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: Listen in Haskell

Prof. Dr.-Ing. Gregor Snelting | WS 2012/13

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil I

Listen in Haskell

Listen

Listen

Ein Liste ist entweder

1. die leere Liste `[]`, oder
2. eine Liste `(x:xs)`, konstruiert aus Restliste `xs` und Listenkopf `x`

Primitive Rekursion: Rekursion über diese Struktur von Listen

- Genau eine Gleichung pro Listen-Konstruktor (`[]` bzw. `(:)`)

```
app []     r = r
app (x:xs) r = x:(app xs r)
```

allgemeinere Rekursionsschemen: komplexere Konstruktoren-*Pattern*

- Beliebige Schachtelung von (Listen-)Konstruktoren

```
odds []       = []
odds [x]      = [x]
odds (x:y:xs) = x : odds xs
```

```
odds []       = []
odds (x:[])   = [x]
odds (x:y:xs) = x : odds xs
```

Haskell-Style Listen in Java

```
package list;

public abstract class List {
  List() {}
  public abstract boolean isNull();
  public abstract int head();
  public abstract List tail();
}


public final class ListFunctions {
  public static int length(List l) {
    if (l.isNull())  return 0;
    else return 1 + length(l.tail());
  }
}
```

- Instanzen von `Null` oder `Cons` (keine weitere Unterklassen)
- jeweils genau ein Konstruktor
- Erzeugen von Listen: verschachtelte Konstruktoren

```
new Cons(1,new Cons(2, new Cons(3,new Null())));
```

- Listen sind *immutable*

```
public final class Null extends List {
  public Null() {}
  public int head() {
    throw new RuntimeException();
  }
  public List tail() {
    throw new RuntimeException();
  }
  public boolean isNull() { return true; }
}
```

```
public final class Cons extends List {
  private final int x;
  private final List xs;

  public Cons(int x, List xs) {
    assert xs!=null;
    this.x = x; this.xs=xs;
  }
  public int  head() { return x; }
  public List tail() { return xs; }
  public boolean isNull() { return false; }
}
```

Haskell-Listen $\neq$ Arrays

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Index in *Gesamt*liste unbekannt!

```
static int total(int[] points){
  int t = 0;
  for(int i=0; i<points.length ; i++){
    t+=i*points[i];
  }
  return t;
}
```

```
total []     = 0
total (x:xs) = ???
```

Zellen-Zugriff: nicht beliebig per Index, nur nacheinander

```
static int passed(int[] points){
  int p = 0;
  for(int i=5; i<points.length ; i++){
    p+=points[i];
  }
  return p;
}
```

```
passed []     = 0
passed (x:xs) = ???
```

Haskell-Listen $\simeq$ Iteratoren

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Index manuell mitführen

```
static int total(Iterable<Integer> points){
  int t = 0; int i=0;
  for(Integer x : points){
    t+=i*x;
    i++;
  }
  return t;
}
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){
  int p = 0; int i=0;
  Iterator<Integer> iter = points.iterator();
  while(iter.hasNext() && i++ < 5) iter.next();
  while(iter.hasNext()) p+=iter.next();
  return p;
}
```

Haskell-Listen $\simeq$ Iteratoren

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Index manuell mitführen

```
static int total(Iterable<Integer> points){
  int t = 0; int i=0;
  for(Integer x : points){
    t+=i*x;
    i++;
  }
  return t;
}
```

```
total points = (wSumFrom 0 points)
 where wSumFrom i []     = 0
       wSumFrom i (x:xs) = i*x +
                           wSumFrom (i+1) xs
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){
  int p = 0; int i=0;
  Iterator<Integer> iter = points.iterator();
  while(iter.hasNext() && i++ < 5) iter.next();
  while(iter.hasNext()) p+=iter.next();
  return p;
}
```

```
passed  points = sum (drop 5 points)
  where drop 0 l      = l
        drop i []     = []
        drop i (x:xs) = drop (i-1) xs
```

Haskell-Listen $\simeq$ Iteratoren

Gegeben: Histogramm von Klausurpunkten, z.b. `[3,0,4,5,8,12,2]`

- 3 mal 0 Punkte, 0 mal 1 Punkt, . . .
- Berechne: Gesamtpunkte, Zahl bestandener Prüfungen (Punkte ≥ 5)

Zellen-Zugriff: immer nur auf vorderstes Element von *Teil*liste

$\Rightarrow$Listenelemente mit Index annotieren

```
static int total(Iterable<Integer> points){
  int t = 0; int i=0;
  for(Integer x : points){
    t+=i*x;
    i++;
  }
  return t;
}
```

```
total points =
  sum (zipWith (*) [0..] points)

total points =
  sum [i*x | (i,x) <- zip [0..] points]
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){
  int p = 0; int i=0;
  Iterator<Integer> iter = points.iterator();
  while(iter.hasNext() && i++ < 5) iter.next();
  while(iter.hasNext()) p+=iter.next();
  return p;
}
```

```
passed  points = sum (drop 5 points)
  where drop 0 l      = l
        drop i []     = []
        drop i (x:xs) = drop (i-1) xs
```

Teil II

Endrekursion

Programm Beschleunigen

Gegeben: Fakultätsfunktion

```
fak n
    | n == 0     = 1
    | otherwise = n * fak (n - 1)
```

Ziel: Effiziente Berechnung.

Programm Beschleunigen

Gegeben: Fakultätsfunktion

```
fak n
    | n == 0     = 1
    | otherwise = n * fak (n - 1)
```

Ziel: Effiziente Berechnung.

Vorlesung: Endrekursive Variante ist schneller.

```
fak n = fakAcc n 1
    where fakAcc n acc
        | n == 0     = acc
        | otherwise = fakAcc (n-1) (n*acc)
```

Endrekursion – Warum?

Ohne Akkumulator

```
Stack s = new Stack();

int fak(int n) {
    while (n > 0) {
        push(n);
        n = n - 1;
    }

    int res = 1;

    while (!s.empty()) {
        res *= s.pop();
    }

    return res;
}
```

Mit Akkumulator

```
int fak(int n) {
    int acc = 1;

    while (n > 0) {
        acc *= n;
        n    = n - 1;
    }

    return acc;
}
```

Endrekursion – Zusammenfassung

Zusammenfassung:

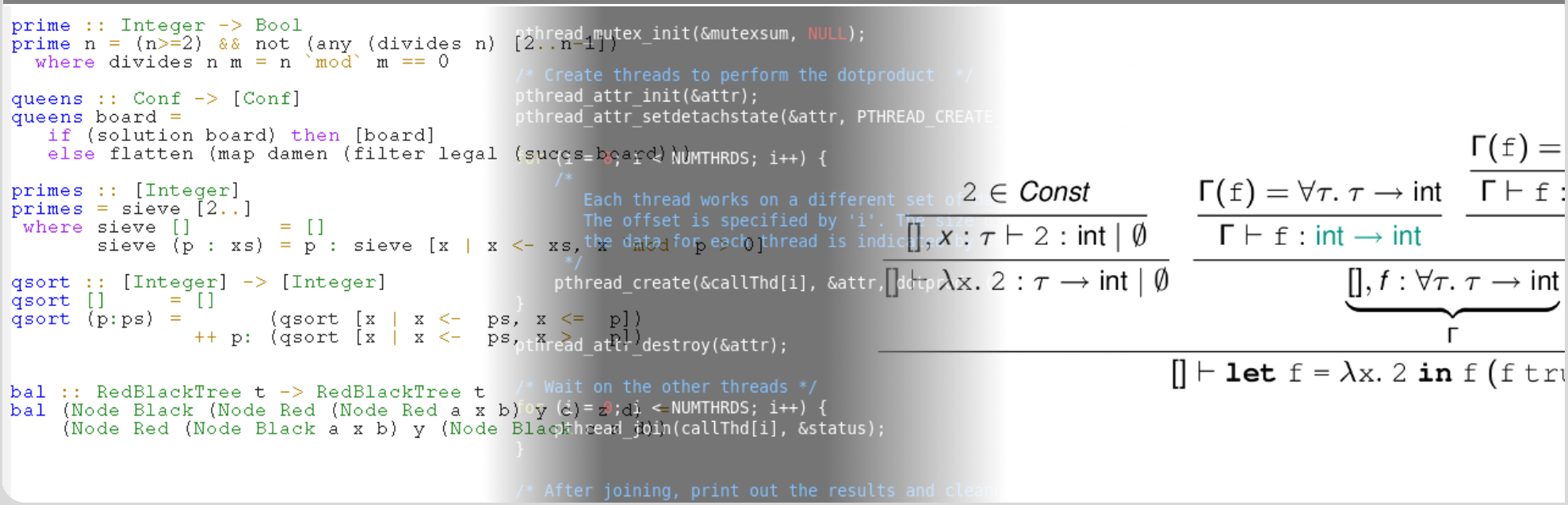
- Endrekursion + Akkumulator $\simeq$ while + lokale Variablen
- Reduzierter Speicherverbrauch

Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: List-Comprehensions, Backtracking

Prof. Dr.-Ing. Gregor Snelting | WS 2012/13

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil I

List-Comprehensions, Backtracking

List-Comprehensions

List-Comprehensions: “lediglich” Schreibweise zur Listengenerierung

- jede List-Comprehension prinzipiell auch als Kombination von **`map`**, **`filter`**, `flatten` darstellbar
- aber üblicherweise viel besser lesbar! zum Beispiel:

```
graduates :: Examination -> [Student]
graduates exam = [student | (student,assesment) <- exam, passed assesment ]
```

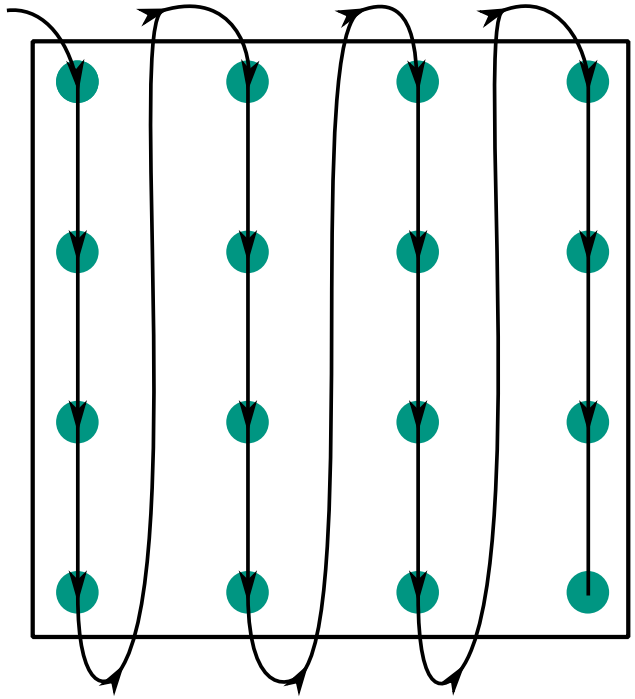
statt

```
graduates exam = map fst (filter (passed . snd) exam)
```

Aufzählung, List-Comprehensions

Menge (unsortiert): $\{(x, y) \mid x \in \{1 \ldots n\} \wedge y \in \{1 \ldots n\}\}$

Liste (spaltenweise):



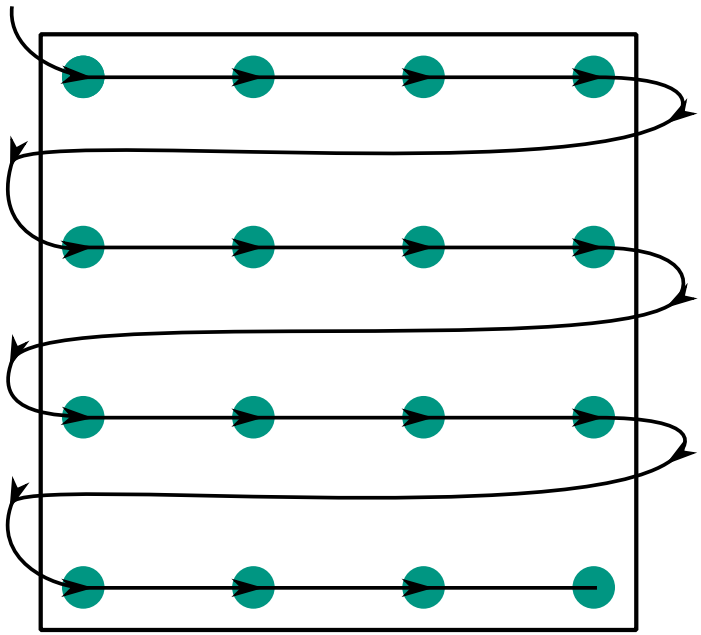
```
pairs     :: Integer -> [(Integer,Integer)]
pairs     n = [(x,y) | x <- [1..n], y <- [1..n]]
```

- Fixiere `x==1`, zähle auf `(1,1),(1,2),(1,3),(1,4)`
- Fixiere `x==2`, zähle auf `(2,1),(2,2),(2,3),(2,4)`
- ...

Aufzählung, List-Comprehensions

Menge (unsortiert): $\{(x, y) \mid x \in \{1 \ldots n\} \wedge y \in \{1 \ldots n\}\}$

Liste (zeilenweise):



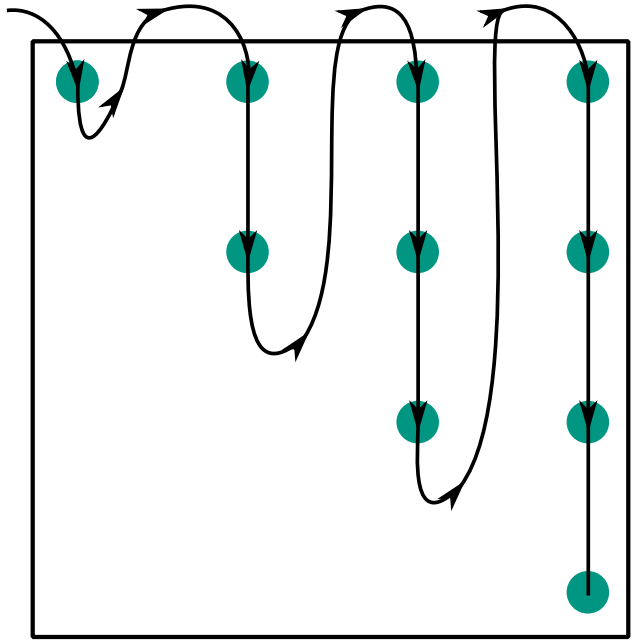
```
pairs     :: Integer -> [(Integer,Integer)]
pairs     n = [(x,y) | y <- [1..n], x <- [1..n]]
```

- Fixiere `y==1`, zähle auf `(1,1),(2,1),(3,1),(4,1)`
- Fixiere `y==2`, zähle auf `(1,2),(2,2),(3,2),(4,2)`
- ...

Aufzählung, List-Comprehensions

Menge (unsortiert): $\{(x, y) \mid x \in \{1 \ldots n\} \wedge y \in \{1 \ldots x\}\}$

Liste (spaltenweise):

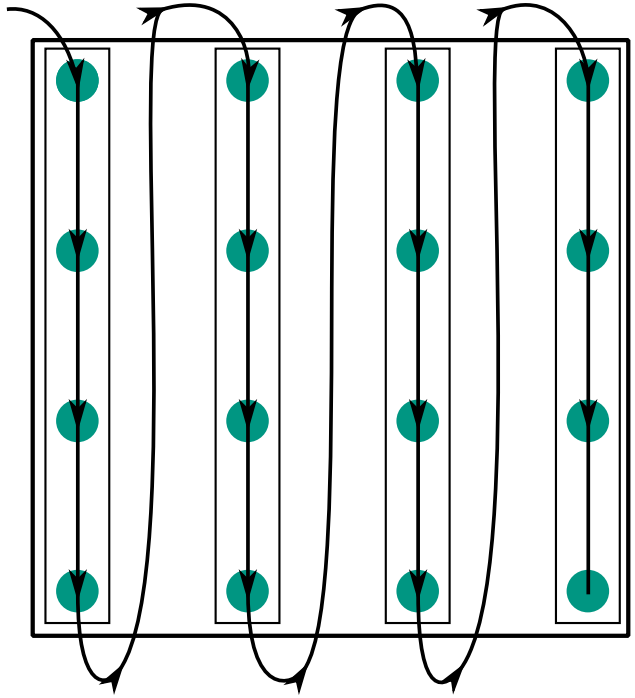


```
triangle :: Integer -> [(Integer,Integer)]
triangle n = [(x,y) | x <- [1..n], y <- [1..x]]
```

- Fixiere `x==1`, zähle auf `(1,1)`
- Fixiere `x==2`, zähle auf `(2,1),(2,2)`
- ...

Aufzählung, List-Comprehensions

`flatten` “Verflache” Listen von Listen



```
flatten :: [[t]] -> [t]
flatten lists = [ x | list <- lists, x <- list ]
```

- Fixiere erste Liste, zähle Elemente auf
- Fixiere zweite Liste, zähle Elemente auf
- ...

Backtracking, List-Comprehensions

Damenproblem: berechne Liste aller von `board` aus erreichbarer Lösungen

```
queens :: Board -> [Board]
queens board =
   if (solution board) then [board]
   else flatten (map queens (filter legal (succs board)))
```

List-Comprehension: macht *backtracking* explizit sichtbar

- Ersetze **`map`**, **`filter`** durch List-Comprehension

```
queens board =
   if   (solution board) then [board]
   else flatten [ queens succ  | succ <- (succs board), legal succ ]
```

- Ersetze `flatten` durch List-Comprehension

```
queens board =
   if   (solution board) then [board]
   else [ sol  | succ <- (succs board), legal succ, sol <- (queens succ) ]
```

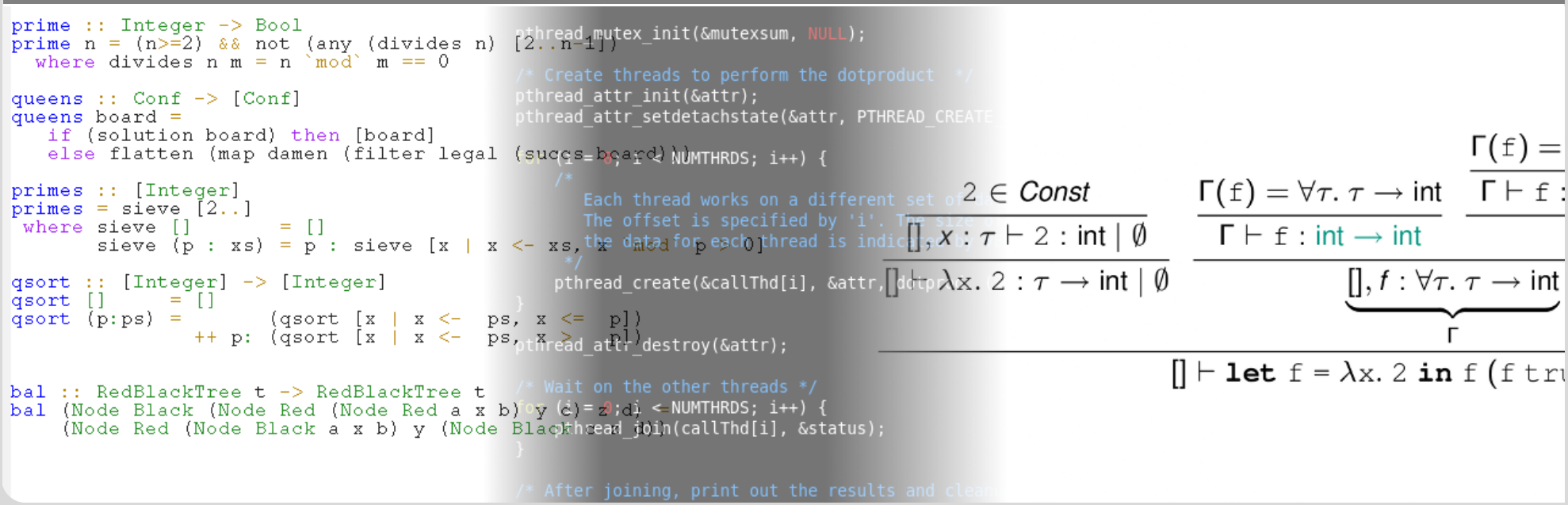
Nimm `sol` in Liste von Lösungen auf, falls `succ` legale Nachfolger von `board`, und `sol` eine von `succ` erreichbare Lösung ist

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 []     = []                    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
-------------------------------------
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 []     = []                        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 von Streams

Karlsruher Institut für Technologie

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	...

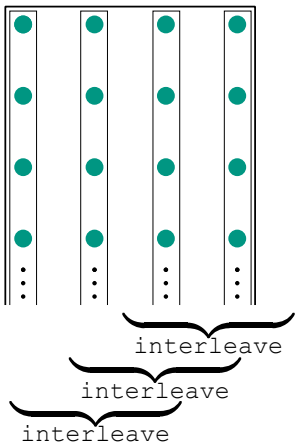
Interleaving von Streams

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

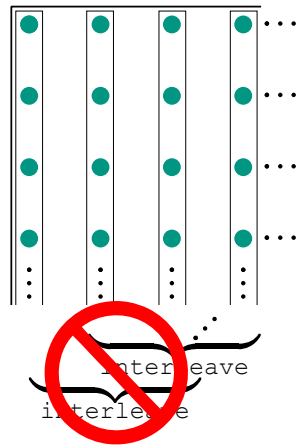
```
interleave :: [t] -> [t] -> [t]
interleave [] ys = ys
interleave xs [] = xs
interleave (x : xs) (y : ys) = x : y : (interleave xs ys)
```

- 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
```

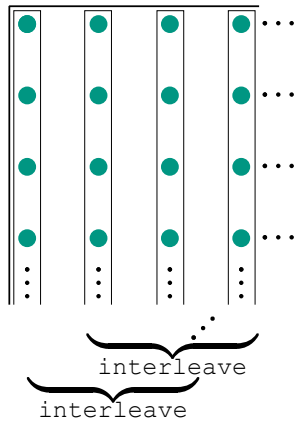
- Abhilfe speziell für `interleave` : alternative Definition

```
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

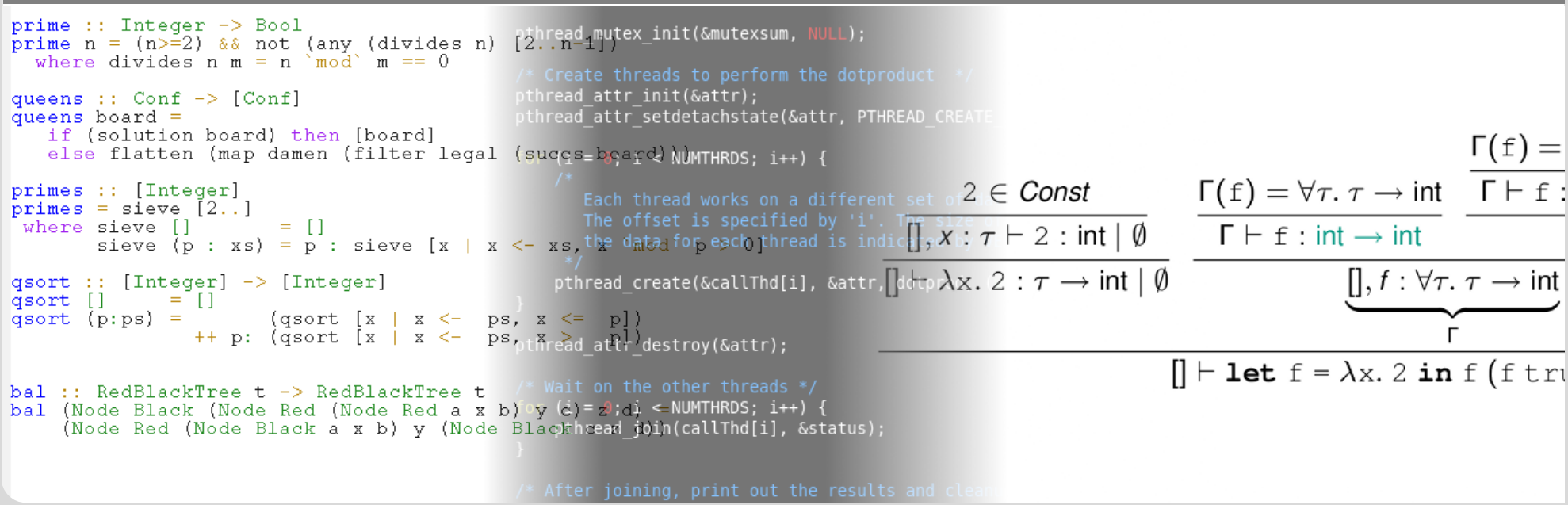


Karlsruher Institut für Technologie

Programmierparadigmen
Tutorium: Lazyness, Streams

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

LEHRSTUHL PROGRAMMIERPARADIGMEN



Listen im λ-Kalkül

rekursive `data`-Typen: darstellbar im λ-Kalkül, z.B. Listen:

$$[1,2,3] \simeq \lambda \mathtt{c}.\ \lambda \mathtt{n}.\ c\ 1\ (c\ 2\ (c\ 3\ \mathtt{n}))$$

- möglicher Typ: $\tau_{\mathtt{list}(\alpha)} = (\alpha \to \beta \to \beta) \to \beta \to \beta$
  α: Elementtyp
- aber: Funktionstypen passen nicht
  $tail = \lambda l\ c\ n.\ l\ (\lambda x\ xs\ i.\ i\ x\ (xs\ c))\ (\lambda f.\ n)\ (\lambda x\ xs.\ xs)$

$\vdash tail : ((\alpha_2 \to (\alpha \to \alpha_3) \to (\alpha_2 \to \alpha_3 \to \alpha_4) \to \alpha_4) \to (\alpha_5 \to \alpha_1) \to (\alpha_6 \to \alpha_7 \to \alpha_7) \to \alpha_8) \to \alpha \to \alpha_1 \to \alpha_8$

- zumindest nicht: $\tau_{\mathtt{list}(\alpha)} \to \tau_{\mathtt{list}(\alpha)}$

Erweitere λ-Kalkül um Listen

- `Cons,Nil,head,tail,null`
- `null` $(\mathtt{Cons}\ t_1\ t_2) \Rightarrow \mathtt{False}$ $\quad \mathtt{null}\ \mathtt{Nil} \Rightarrow \mathtt{True}$
  `head` $(\mathtt{Cons}\ t_1\ t_2) \Rightarrow t_1$ $\quad \mathtt{tail}\ (\mathtt{Cons}\ t_1\ t_2) \Rightarrow t_2$
- `head Nil` $\not\Rightarrow$ (Laufzeitfehler) $\quad$ `head 42` (Typfehler)

Typregeln für Listen

Typ von `Cons`, `Nil`, `head`, `tail`, `null` abhängig von Elementtyp:

- erweiterte Typsyntax

$$\tau = \underbrace{\textit{Basistyp}}_{\text{int, bool}} \mid \underbrace{\textit{Var}}_{\alpha, \alpha_2, \beta, \gamma, \ldots} \mid \tau_1 \underbrace{\rightarrow}_{\text{2-stelliger Typkonstruktor}} \tau_2 \mid \underbrace{\text{list}\,(\tau)}_{\text{1-stelliger Typkonstruktor}}$$

- keine fixen Typen $\tau_{\texttt{Cons}}, \tau_{\texttt{head}}, \ldots \Rightarrow$ neue *Typregeln*

Typregeln für Listen

$$\textsc{Cons}\ \frac{}{\Gamma \vdash \texttt{Cons}\ t_1\ t_2 : \text{list}\,(\tau)} \qquad \textsc{Nil}\ \frac{}{\Gamma \vdash \texttt{Nil} : \text{list}\,(\tau)}$$

$$\textsc{head}\ \frac{}{\Gamma \vdash \texttt{head}\ t : \text{list}\,(\tau)} \qquad \textsc{tail}\ \frac{}{\Gamma \vdash \texttt{tail}\ t : \text{list}\,(\tau)}$$

$$\textsc{null}\ \frac{}{\Gamma \vdash \texttt{null}\ t : \text{bool}}$$

Typregeln für Listen

Typ von `Cons`, `Nil`, `head`, `tail`, `null` abhängig von Elementtyp:

- erweiterte Typsyntax

$$\tau = \underbrace{\textit{Basistyp}}_{\text{int, bool}} \mid \underbrace{\textit{Var}}_{\alpha, \alpha_2, \beta, \gamma, \ldots} \mid \tau_1 \underbrace{\rightarrow}_{2\text{-stelliger Typkonstruktor}} \tau_2 \mid \underbrace{\text{list}\,(\tau)}_{1\text{-stelliger Typkonstruktor}}$$

- keine fixen Typen $\tau_{\texttt{Cons}}, \tau_{\texttt{head}}, \ldots \Rightarrow$ neue *Typregeln*

Typregeln für Listen

$$\textsc{Cons}\ \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \text{list}\,(\tau)}{\Gamma \vdash \texttt{Cons}\ t_1\ t_2 : \text{list}\,(\tau)} \qquad \textsc{Nil}\ \frac{}{\Gamma \vdash \texttt{Nil} : \text{list}\,(\tau)}$$

$$\textsc{Head}\ \frac{\Gamma \vdash t : \text{list}\,(\tau)}{\Gamma \vdash \texttt{head}\ t : \text{list}\,(\tau)} \qquad \textsc{Tail}\ \frac{\Gamma \vdash t : \text{list}\,(\tau)}{\Gamma \vdash \texttt{tail}\ t : \text{list}\,(\tau)}$$

$$\textsc{Null}\ \frac{\Gamma \vdash t : \text{list}\,(\tau)}{\Gamma \vdash \texttt{null}\ t : \text{bool}}$$

Typregeln für Listen

Typ von `Cons,Nil,head,tail,null` abhängig von Elementtyp:

- erweiterte Typsyntax

$$\tau = \underbrace{\textit{Basistyp}}_{\text{int,bool}} \mid \underbrace{\textit{Var}}_{\alpha,\alpha_2,\beta,\gamma,\ldots} \mid \tau_1 \underbrace{\rightarrow}_{\text{2-stelliger Typkonstruktor}} \tau_2 \mid \underbrace{\text{list}\,(\tau)}_{\text{1-stelliger Typkonstruktor}}$$

- keine fixen Typen $\tau_{\texttt{Cons}}, \tau_{\texttt{head}}, \ldots \Rightarrow$ erweiterte Typannahmen

$$\begin{aligned}\Gamma_{\texttt{list}} = \quad & \texttt{Cons} : \forall\alpha.\alpha \rightarrow \text{list}\,(\alpha) \rightarrow \text{list}\,(\alpha),\\ & \texttt{Nil} : \forall\alpha.\text{list}\,(\alpha)\,,\\ & \texttt{head} : \forall\alpha.\text{list}\,(\alpha) \rightarrow \text{list}\,(\alpha),\\ & \texttt{tail} : \forall\alpha.\text{list}\,(\alpha) \rightarrow \text{list}\,(\alpha),\\ & \texttt{null} : \forall\alpha.\text{list}\,(\alpha) \rightarrow \text{bool}\end{aligned}$$

Funktionen in Haskell: polymorphe Definition erweitert Γ um Typschema

```
length :: [a] -> Int
length list = ...
```

$$\Gamma' = \Gamma, \texttt{length} : \forall\alpha.\text{list}\,(\alpha) \rightarrow \text{int}$$

Typinferenz für Listen

Wie zuvor: $[t_1, t_2, \ldots, t_n]$ für $\texttt{Cons}\ t_1\ (\ldots(\texttt{Cons}\ t_n\ \texttt{Nil})\ldots)$

$$\dfrac{\dfrac{\Gamma_{\texttt{list}} \vdash \texttt{\&\&} : \alpha_4 \qquad \dfrac{\Gamma_{\texttt{list}} \vdash \texttt{null} : \alpha_6 \qquad \Gamma_{\texttt{list}} \vdash [] : \alpha_7}{\Gamma_{\texttt{list}} \vdash \texttt{null}\ [] : \alpha_5}}{\Gamma_{\texttt{list}} \vdash \texttt{\&\&}\ (\texttt{null}\ []) : \alpha_3} \qquad \dfrac{\Gamma_{\texttt{list}} \vdash \texttt{null} : \alpha_8 \qquad \Gamma_{\texttt{list}} \vdash [1,2,3] : \alpha_9}{\Gamma_{\texttt{list}} \vdash \texttt{null}\ [1,2,3] : \alpha_4}}{\Gamma_{\texttt{list}} \vdash (\texttt{null}\ []) \texttt{\&\&} (\texttt{null}\ [1,2,3]) : \alpha_1}$$

$$\begin{aligned} C = \quad & \{\alpha_9 = \text{list}\,(\text{int}), \alpha_3 = \alpha_4 \to \alpha_1, \alpha_4 = \alpha_5 \to \alpha_3, \\ & \alpha_4 = \text{bool} \to (\text{bool} \to \text{bool}), \alpha_6 = \alpha_7 \to \alpha_5, \alpha_8 = \alpha_9 \to \alpha_4 \\ & \alpha_6 = ???? \\ & \alpha_8 = ???? \\ & \alpha_7 = ????\} \end{aligned}$$

$$\begin{aligned} \Gamma_{\texttt{list}} = \quad & \ldots \\ & \texttt{null} : \forall\alpha.\text{list}\,(\alpha) \to \text{bool}, \\ & \texttt{Nil} : \forall\alpha.\text{list}\,(\alpha) \\ & \ldots \end{aligned}$$

Typinferenz für Listen

Wie zuvor: $[t_1, t_2, \ldots, t_n]$ für $\texttt{Cons}\ t_1\ (\ldots(\texttt{Cons}\ t_n\ \texttt{Nil})\ldots)$

$$\dfrac{\dfrac{\Gamma_{\texttt{list}} \vdash \texttt{\&\&} : \alpha_4 \qquad \dfrac{\Gamma_{\texttt{list}} \vdash \texttt{null} : \alpha_6 \qquad \Gamma_{\texttt{list}} \vdash [] : \alpha_7}{\Gamma_{\texttt{list}} \vdash \texttt{null}\ [] : \alpha_5}}{\Gamma_{\texttt{list}} \vdash \texttt{\&\&}\ (\texttt{null}\ []) : \alpha_3} \qquad \dfrac{\Gamma_{\texttt{list}} \vdash \texttt{null} : \alpha_8 \qquad \Gamma_{\texttt{list}} \vdash [1,2,3] : \alpha_9}{\Gamma_{\texttt{list}} \vdash \texttt{null}\ [1,2,3] : \alpha_4}}{\Gamma_{\texttt{list}} \vdash (\texttt{null}\ []) \texttt{\&\&} (\texttt{null}\ [1,2,3]) : \alpha_1}$$

$$\begin{aligned} C = \quad & \{\alpha_9 = \text{list}\,(\text{int}), \alpha_3 = \alpha_4 \to \alpha_1, \alpha_4 = \alpha_5 \to \alpha_3, \\ & \alpha_4 = \text{bool} \to (\text{bool} \to \text{bool}), \alpha_6 = \alpha_7 \to \alpha_5, \alpha_8 = \alpha_9 \to \alpha_4 \\ & \alpha_6 = \text{list}\,(\beta_1) \to \text{bool} \\ & \alpha_8 = \text{list}\,(\beta_2) \to \text{bool} \\ & \alpha_7 = \text{list}\,(\beta_3)\} \end{aligned}$$

$$\begin{aligned} \Gamma_{\texttt{list}} = \quad & \ldots \\ & \texttt{null} : \forall\alpha.\text{list}\,(\alpha) \to \text{bool}, \\ & \texttt{Nil} : \forall\alpha.\text{list}\,(\alpha) \\ & \ldots \end{aligned}$$