

Programmierparadigmen

Tutorium: Listen in Haskell

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

LEHRSTUHL PROGRAMMIERPARADIGMEN

```

prime :: Integer -> Bool
prime n = (n >= 2) && not (any (divides n) [2 .. n-1]) /* thread_mutex_init(&mutexsum, NULL);
  where divides n m = n `mod` m == 0 */ Create threads to perform the dotproduct */
queens :: Conf -> [Conf]
queens board =
  if (solution board) then [board]
  else flatten (map damen (filter legal (succes_board NUMTHRDS; i++)) { */
primes :: [Integer]
primes = sieve [2..] /* Each thread works on a different set of primes */
  where sieve [] = []
        sieve (p : xs) = p : sieve [x | x <- xs, x `mod` p > 0] /* The offset is specified by 'i'. The data for each thread is index[i], x : τ ⊢ 2 : int | ∅ */
qsort :: [Integer] -> [Integer] /* pthread_create(&callThd[i], &attr, [lambda x. 2 : τ → int | ∅] */
qsort [] = []
qsort (p:ps) = qsort [x | x <- ps, x <= p] /* */
  ++ p: (qsort [x | x <- ps, x > p], thread_attr_destroy(&attr)); */

bal :: RedBlackTree t -> RedBlackTree t /* Wait on the other threads */
bal (Node Black (Node Red (Node Red a b)) y c) = z; d, i < NUMTHRDS; i++) {
  (Node Red (Node Black a x b)) y (Node Black); thread_join(callThd[i], &status);
}

/* After joining, print out the results and cleanup

```

$$\frac{\Gamma(f) = \forall\tau. \tau \rightarrow \text{int} \quad \frac{}{\Gamma \vdash f : \text{int} \rightarrow \text{int}}}{\boxed{\Gamma, f : \forall\tau. \tau \rightarrow \text{int}}}$$

$\boxed{1} \vdash \text{let } f = \lambda x. 2 \text{ in } f(f \text{ true})$

Teil I

Listen in Haskell

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. `(:)`)

$$\begin{aligned} \text{app } [] &= r \\ \text{app } (x:xs) &= x : (\text{app } xs \ r) \end{aligned}$$

allgemeinere Rekursionsschemen: komplexere Konstruktoren-*Pattern*

- Beliebige Schachtelung von (Listen-)Konstruktoren

$$\begin{aligned} \text{odds } [] &= [] \\ \text{odds } [x] &= [x] \\ \text{odds } (x:y:xs) &= x : \text{odds } xs \end{aligned}$$

$$\begin{aligned} \text{odds } [] &= [] \\ \text{odds } (x:[]) &= [x] \\ \text{odds } (x:y:xs) &= x : \text{odds } xs \end{aligned}$$

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());
    }
}

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; }
}
```

- 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 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 *Teilliste*

\Rightarrow Index in *Gesamtliste* unbekannt!

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

Zellen-Zugriff: nicht beliebig per Index, nur nacheinander

```
static int passed(int[] points) {                               passed []      = 0
    int p = 0;                                                 passed (x:xs) = ???
    for(int i=5; i<points.length ; i++) {
        p+=points[i];
    }
    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 Teilliste

⇒ 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 „dropen“

```
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 Teilliste

⇒ Index manuell mitführen

```
static int total(Iterable<Integer> points){           total points = (wSumFrom 0 points)
    int t = 0; int i=0;                                where wSumFrom i []      = 0
    for(Integer x : points){                          wSumFrom i (x:xs) = i*x +
        t+=i*x;                                         wSumFrom (i+1) xs
        i++;
    }
    return t;
}
```

Zellen-Zugriff: vorhergehende Elemente „droppen“

```
static int passed(Iterable<Integer> points){           passed points = sum (drop 5 points)
    int p = 0; int i=0;                                where drop 0 l      = l
    Iterator<Integer> iter = points.iterator();         drop i []      = []
    while(iter.hasNext() && i++ < 5) iter.next();       drop i (x:xs) = drop (i-1) xs
    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 Teilliste

⇒ Listenelemente mit Index annotieren

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

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

Zellen-Zugriff: vorhergehende Elemente „dropen“

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

```
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)
```

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

Endrekursion und while

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

```
int fak(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n     = n - 1;
    }
    return acc;
}
```

Funktionsargumente

In Schleife veränderter Zustand

- Endrekursive Funktion \simeq while-Schleife
- Akkumulator \simeq lokale Hilfsvariable
- Funktionsargumente \simeq in while-Schleife veränderter Zustand

Endrekursion und while

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

```
int fak(int n) {
  int acc = 1;
  while (n > 0) {
    acc *= n;
    n     = n - 1;
  }
  return acc;
}
```



- Endrekursive Funktion \simeq while-Schleife
- Akkumulator \simeq lokale Hilfsvariable
- Funktionsargumente \simeq in while-Schleife veränderter Zustand

Endrekursion und while

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

```
int fak(int n) {
  int acc = 1;
  while (n > 0) {
    acc *= n;
    n     = n - 1;
  }
  return acc;
}
```

Abbruchbedingung

- Endrekursive Funktion \simeq while-Schleife
- Akkumulator \simeq lokale Hilfsvariable
- Funktionsargumente \simeq in while-Schleife veränderter Zustand

Endrekursion und while

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

```
int fak(int n) {
    int acc = 1;
    while (n > 0) {
        acc *= n;
        n     = n - 1;
    }
    return acc;
}
```

Rückgabewert

- Endrekursive Funktion \simeq while-Schleife
- Akkumulator \simeq lokale Hilfsvariable
- Funktionsargumente \simeq in while-Schleife veränderter Zustand

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;
```

Zusammenfassung:

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