

Übungen zu „Parallelität in funktionalen Programmiersprachen“, Sommer 2009

Nr. 1, Abgabe der Aufgaben: 28. April 2009 vor der Vorlesung

Hinweise:

- Die Übungsblätter erscheinen in der Vorlesung und sind auch dort wieder abzugeben.
- Programmieraufgaben senden Sie bitte **zusätzlich** per E-Mail, ebenfalls spätestens zum Vorlesungsbeginn.

Haskell Compiler und Interpreter

Wir werden parallele Haskellerweiterungen benutzen, die nur mit dem Glasgow Haskell Compiler (GHC) funktionieren. Der 8-Kern-Rechner *sakania* läuft unter Linux und ist eine gute Testplattform für parallele Programme.

Benutzung von Linux, Netz des Fachbereichs

Auch wenn das HOME-Verzeichnis unter Windows als Laufwerk *U:* sichtbar ist, müssen Sie mit Hilfe einer Konsole (Shell) Programme übersetzen und ausführen. Die Bedienung einer Linux-Konsole mit den gängigsten Kommandos sollten Sie beherrschen oder erlernen.

Windows: Aufruf des *SSH-Client* (Programme-Network) und Angabe eines Rechnernamens. Dies öffnet eine Konsole auf dem angegebenen Rechner. Sie können mit Windows-Programmen in Ihrem HOME arbeiten und die Konsole zum Übersetzen und Ausführen verwenden.

Linux: Sie können einen der Linux-Rechner in den Räumen 04D01 oder 05D08 benutzen. Dort arbeiten Sie unter einer grafischen Benutzeroberfläche und können eine Konsole öffnen (Systemwerkzeuge-Terminal).

Verschiedene GHC-Versionen sind am Fachbereich unter Linux installiert. Zur Benutzung mit *sakania* fügen Sie `/app/lang/functional/bin64` zu Ihrem Suchpfad hinzu. Sie können den Pfad permanent erweitern, je nach Shell

bash: in der Datei `~/.bashrc` durch die Zeile: `export PATH=$PATH:/app/lang/functional/bin64`

tcsh: in der Datei `~/.tcshrc` durch die Zeile: `setenv PATH ${PATH}:/app/lang/functional/bin64`

Mit dem Befehl *finger* angewandt auf Ihren Benutzernamen erfahren Sie, welche Shell benutzt wird.

Übungen in der Übung

Diese Aufgaben dienen zum Auffrischen Ihrer Haskell Kenntnisse während der Übung und werden nicht abgegeben.

I Bedingungen abfangen

Die Haskell Funktion `tail :: [a] → [a]` liefert das letzte Element einer Liste. Definieren sie eine Funktion `safetail :: [a] → [a]` analog zu `tail`, die jedoch die leere Liste auf sich selbst abbildet anstatt einen Fehler auszugeben. Benutzen Sie dazu:

- (a) *pattern matching*
- (b) einen bedingten Ausdruck
- (c) *guards* (`| boolExp = Exp`)

Hinweis: Verwenden Sie die Funktion `null`.

II Rekursion

Definieren Sie rekursiv eine Funktion `merge :: Ord a ⇒ [a] → [a] → [a]`, die zwei sortierte Listen zu einer sortierten Liste zusammenmischt.

III Generisches Traversieren

Definieren Sie für einen Baum vom Typ `Tree a` mit Definition

`data Tree a = Leaf a | Node (Tree a) (Tree a)` eine Funktion

`leafs :: Tree a → [a]`, welche die Blattknoten des Baumes ebenenweise liefert.

Hinweis: Ziehen Sie Hilfsfunktionen von folgendem Typ in betracht.

```
leafs' :: [Tree a] → [a]
--oder
leafs'' :: Tree a → [[a]]
```

IV Funktionen höherer Ordnung

Definieren Sie unter Benutzung von `foldl` eine Funktion `dec2int :: [Int] → Int`, die eine Dezimalzahl in einen Integer konvertiert. Z.B.:

```
> dec2int [2,3,4,5]
2345
```

V Lambda Abstraktionen

Was drückt die Funktion `f` aus und welchen Typ hat `f`?

```
f g = λx → (λy → (λz → (x `g` y) `g` z))
```

Hinweis: Sie können den Typ mit `ghci` überprüfen (`:type f`), versuchen Sie es jedoch zunächst ohne Hilfsmittel.

Aufgaben (zum Abgeben)

1.1 Umstellen von Listen

3 Punkte

Schreiben Sie eine Haskell-Funktion `mytranspose :: [[a]] → [[a]]`, welche eine Liste von Zeilen in eine Liste von Spalten umwandelt (also transponiert):

Kann Ihre Funktion auch dann Ergebnisse liefern, wenn die Listen im Argument unterschiedliche Länge haben?

```
*Main> mytranspose [[1,2],[3,4,5],[],[6,7,8]]
[[1,3,6],[2,4,7],[5,8]]
*Main>
```

1.2 Programmanalyse und Parallelisierbarkeit

9 Punkte

Die `transpose`-Funktion lässt sich etwa im folgenden Programm verwenden:

```
import List(transpose)
import Control.Parallel(par,seq)
main :: IO ()
main = putStr (concat (map find' hidden))
find' x = length y `seq` y
  where y = find x
-- find:
-----

find :: [Char] → String
find word =
  word ++ "␣" ++ (concat dirs) ++ "\n"
  where dirs = map snd (forw ++ back)

  forw = filter (any (contains word) . fst)
        [(r,"right␣"), (d,"down␣"), (dl,"downleft␣"), (ul,"upleft␣")]
  back = filter (any (contains draw) . fst)
        [(r,"left␣"), (d,"up␣"), (dl,"upright␣"), (ul,"downright␣")]
  draw = reverse word

  r = grid
  d = transpose grid
  dl = diagonals grid
  ul = diagonals (reverse grid)

-- diagonals:
-- zipinit:
-----

diagonals :: [[a]] → [[a]]
diagonals [r] = map (:[]) r
diagonals (r:rs) = zipinit r ([]:diagonals rs)

zipinit :: [a] → [[a]] → [[a]]
zipinit [] ys = ys
zipinit (x:xs) (y:ys) = (x : y) : zipinit xs ys

-- contains:
-- prefix:
-- suffixes:
-----

contains :: (Eq a) ⇒ [a] → [a] → Bool
contains xs ys = any (prefix xs) (suffixes ys)

suffixes :: [a] → [[a]]
suffixes [] = []
```

```

suffixes xs = xs : suffixes (tail xs)

prefix :: (Eq a) => [a] -> [a] -> Bool
prefix []      ys      = True
prefix xs     []      = False
prefix (x:xs) (y:ys) = x == y && prefix xs ys
-----
grid = [['Y', 'I', 'O', 'M', 'R', 'E', 'S', 'K', 'S', 'T'],
        ['A', 'E', 'H', 'Y', 'G', 'E', 'H', 'E', 'D', 'W'],
        ['Z', 'F', 'I', 'A', 'C', 'N', 'I', 'T', 'I', 'A'],
        ['N', 'T', 'O', 'C', 'O', 'M', 'V', 'O', 'O', 'R'],
        ['E', 'R', 'D', 'L', 'O', 'C', 'E', 'N', 'S', 'M'],
        ['Z', 'O', 'U', 'R', 'P', 'S', 'R', 'N', 'D', 'A'],
        ['O', 'Y', 'A', 'S', 'M', 'O', 'Y', 'E', 'D', 'L'],
        ['R', 'N', 'D', 'E', 'N', 'L', 'O', 'A', 'I', 'T'],
        ['F', 'I', 'W', 'I', 'N', 'T', 'E', 'R', 'R', 'C'],
        ['F', 'E', 'Z', 'E', 'E', 'R', 'F', 'T', 'F', 'I'],
        ['I', 'I', 'D', 'T', 'P', 'H', 'U', 'B', 'R', 'L'],
        ['C', 'N', 'O', 'H', 'S', 'G', 'E', 'I', 'O', 'N'],
        ['E', 'G', 'M', 'O', 'P', 'S', 'T', 'A', 'S', 'O'],
        ['T', 'G', 'F', 'F', 'C', 'I', 'S', 'H', 'T', 'H'],
        ['O', 'T', 'B', 'C', 'S', 'S', 'N', 'O', 'W', 'I']]

hidden = ["COSY", "SOFT", "WINTER", "SHIVER", "FROZEN", "SNOW",
          "WARM", "HEAT", "COLD", "FREEZE", "FROST", "ICE" ]

```

- (a) Geben Sie an, wozu dieses Programm und die enthaltenen Funktionen dienen. Vervollständigen Sie die Kommentare zu den Hilfsfunktionen.
- (b) Welche Teile sind voneinander unabhängig und könnten also gleichzeitig – parallel – ausgeführt werden?
Geben Sie allgemeine Regeln zum Auffinden von Parallelität in funktionalen Programmen an.