Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting — gregor.snelting@kit.edu
Prof. Dr. Ralf Reussner — reussner@kit.edu

Programmierparadigmen – WS 2021/22

https://pp.ipd.kit.edu/lehre/WS202122/paradigmen/uebung

Blatt 1: Rekursive Funktionen in Haskell

Abgabe: 29.10.2021, 14:00
Besprechung: 1.11. – 2.11.2021

Reichen Sie Ihre Abgabe bis zum 29.10.2021 um 14:00 in unserer Praktomat-Instanz unter https://praktomat.cs.kit.edu/pp_2021_WS ein.

Hinweis: Lassen Sie für dieses Blatt Typsignaturen bei der Abgabe noch weg, das macht unsere automatischen Tests stabiler. Ab dem nächsten Blatt sind die Typsignaturen üblicherweise vorgegeben.

1 Potenzen, Wurzeln und Primzahlen in Haskell

Geben Sie Ihre Lösungen als Modul `Arithmetik`[1] ab.

1. Schreiben Sie eine rekursive Funktion `pow1`, die die Basis `b` und den Exponent `e` als Parameter nimmt und b^e naiv über die Gleichungen $b^0 = 1$ und $b^{e+1} = b \cdot b^e$ berechnet.

   Beispiellösung:

   ```
   pow1 b e
     | e < 0      = error "Negativer␣Exponent"
     | e == 0     = 1
     | otherwise = b * pow1 b (e - 1)
   ```

2. Wesentlich effizienter als die naive Implementierung ist es, bei jedem Rekursionsschritt den Exponenten zu halbieren und die Basis zu quadrieren: $b^{2e} = (b^2)^e$ bzw. $b^{2e+1} = b \cdot (b^2)^e$. Schreiben Sie eine weitere Funktion `pow2`, die die Potenz auf diese Weise effizienter berechnet. Wie viele Aufrufe braucht `pow2` im Vergleich zu `pow1`?

   Beispiellösung:

   ```
   pow2 b e
     | e < 0           = error "Negativer␣Exponent"
     | e == 0          = 1
     | e `mod` 2 == 0 = pow2 (b * b) (e `div` 2)
     | otherwise       = b * pow2 (b * b) (e `div` 2)
   ```

   Für die Berechnung der n-ten Potenz braucht `pow1` $\Theta(n)$ Aufrufe, während `pow2` nur $\Theta(\log n)$ Aufrufe braucht.

3. Transformieren Sie nun `pow2` in eine endrekursive Version `pow3`. `pow3` soll weiterhin nur zwei Parameter, Basis und Exponent, erwarten, aber mit einer Hilfsfunktion mit Akkumulator die Potenz berechnen. Fügen Sie auch noch eine Überprüfung hinzu, die bei negativen Exponenten mittels `error` einen Fehler mit aussagekräftiger Fehlermeldung auslöst.

   Beispiellösung:

[1] also in einer Datei `Arithmetik.hs` mit erster Zeile `module Arithmetik where`

```
pow3 b e
  | e < 0 = error "Negativer␣Exponent"
  | otherwise = pow3Acc 1 b e
    where
      pow3Acc acc b e
        | e == 0         = acc
        | e `mod` 2 == 0 = pow3Acc acc (b * b) (e `div` 2)
        | e `mod` 2 == 1 = pow3Acc (b * acc) (b * b) (e `div` 2)
```

4. Implementieren Sie nun eine Funktion `root e r`, die die ganzzahlige `e`-te Wurzel von `r` berechnet, d.h. `root e r` errechnet die größte nichtnegative ganze Zahl x, für die $x^{\mathtt{e}} \leq \mathtt{r}$ gilt.

   Verwenden Sie für die Berechnung das Verfahren der Intervallhalbierung: Schreiben Sie eine Hilfsfunktion, die in Parametern die ganzzahligen Grenzen a und b eines Intervalls bekommt, in dem die gesuchte Zahl x liegt: $a \leq x < b$. Ist die Größe des Intervalls 1 (d.h. $b - a = 1$), so ist die untere Grenze a die gesuchte Zahl. Größere Intervalle halbiert man und prüft dann, in welcher Hälfte des Intervalls die gesuchte Zahl liegt. Die Suche wird dann rekursiv mit dieser Hälfte fortgesetzt. Achten Sie darauf, die Invariante $a \leq x < b$ auch bei den rekursiven Aufrufen zu erhalten.

   Versehen Sie wie schon `pow3` die `root`-Funktion mit Überprüfungen der Vorbedingungen Ihrer Parameter. Überlegen Sie genau, für welche Werte von `e, r` die Berechnung möglich ist und achten Sie auf eine korrekte Behandlung der Randfälle!

   Beispiellösung:

```
root e r
  | e < 1     = error "Nicht-positiver␣Wurzelexponent"
  | r < 0     = error "Negativer␣Radikant"
  | otherwise = searchRoot 0 (r + 1)
  where
    searchRoot low high
      | low + 1 == high  = low
      | pow3 half e <= r = searchRoot half high
      | otherwise        = searchRoot low half
      where half = (low + high) `div` 2
```

5. Schreiben Sie eine Funktion `isPrime`, die eine natürliche Zahl auf ihre Primzahleigenschaft testet. Für Eingabe n soll dazu getestet werden, ob n durch eine Zahl zwischen 2 und $\sqrt{n}$ teilbar ist.

   Beispiellösung:

```
isPrime n
  | n < 2     = error "Zu␣kleine␣Zahl␣fuer␣Primzahltest"
  | otherwise = nHasNoDivisorGreaterThan 2
    where
      upto = root 2 n
      nHasNoDivisorGreaterThan k
        | k > upto       = True
        | n `mod` k == 0 = False
        | otherwise      = nHasNoDivisorGreaterThan (k + 1)
```

   Beispiellösung:

```
isPrime n
  | n < 2     = error "Zu␣kleine␣Zahl␣fuer␣Primzahltest"
  | otherwise = null $ filter (\k -> n `mod` k == 0) [2..root 2 n]
```

Hinweis: Die Funktionen `div` und `mod` sind Divisions- bzw. Modulo-Operator in Haskell.

2 Sortieren

Erzeugen Sie ein Modul `Sort`[2] und implementieren Sie darin Insertionsort. Gehen Sie dazu wie folgt vor:

1. Schreiben Sie eine Funktion `insert`, die eine Ganzzahl in eine aufsteigend sortierte Liste von Ganzzahlen an der richtigen Stelle einfügt.
2. Verwenden Sie `insert` nun für die Funktion `insertSort`, die eine Liste von ganzen Ganzzahlen nimmt und diese mit InsertionSort sortiert.

Beispiellösung:

```
insert x []       = [x]
insert x (y : ys)
    | x <= y      = x : y : ys
    | otherwise   = y : insert x ys

insertSort [] = []
insertSort (x : xs) = insert x (insertSort xs)
```

Beispiellösung:

```
insertSort = foldr insert []
```

Erweitern Sie Ihr Modul nun um eine Mergesort-Implementierung:

3. Schreiben Sie eine Funktion `merge`, die zwei sortierte Listen von Ganzzahlen zu einer sortierten Liste zusammenführt.
4. Verwenden Sie `merge` (und die aus der Vorlesung bekannte Funktion `length`) nun für die Funktion `mergeSort`, die eine Liste von Ganzzahlen entgegennimmt und diese mit Mergesort sortiert.

Beispiellösung:

```
merge xs []   = xs
merge [] ys   = ys
merge (x : xs) (y : ys)
  | x <= y    = x : merge xs (y : ys)
  | otherwise = y : merge (x : xs) ys

mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = merge
  (mergeSort (take (length xs `div` 2) xs))
  (mergeSort (drop (length xs `div` 2) xs))
```

[2] also eine Datei namens `Sort.hs` mit erster Zeile **`module`** `Sort` **`where`**

B-Seite: Tiefgründige Typen

Geben Sie Ihre Lösung in Freitextform ab.

Die Aufgabenreihe der "B-Seiten" versucht, die Rolle einer klassischen Saalübung zu füllen. Sie vermittelt primär Denkweisen und zeigt Zusammenhänge zwischen Kapiteln auf. Bearbeiten Sie diese Aufgaben also bitte gewissenhaft, auch wenn es keine klassischen Rechenübungen sind.

In Haskell verraten die Typen von Ausdrücken viel über die Werte, die sie annehmen können. Beispielsweise wissen wir mit `x :: Int`[3], dass `x` nicht die Werte `"foo"` oder `[1,2,3]` annehmen kann, wohl aber `42`.

Es ist mitunter gar nicht so einfach, zu einem gegebenen Typen einen Bewohner zu finden!

Diese Aufgabe erwartet ein ungefähres Verständnis von Typsignaturen, das wir hier kurz wiederholen:

- Eine Folge von Funktionstypen wie `Int -> String -> Bool` steht gemäß Currying für eine Funktion mit mehreren Parametern. Am rechten Ende eines solchen Typs steht der Rückgabetyp (`Bool`), davor sind die Parameter (`Int`, `String`). Parameter können selbst wieder Funktionstyp haben, müssen dann aber entsprechend geklammert sein.
- In Typen stehen kleine Buchstaben (`a`, `b`) für Typvariablen. Solche Typvariablen können an einer konkreten Verwendungsstelle mit beliebigen Typen ersetzt werden, eine Funktion des Typs `a -> [a]` kann also z.B. als `Int -> [Int]` fungieren, indem man einfach ein Argument vom Typ `Int` übergibt.
- Um solche allgemeinen Funktionen zu schreiben, muss man nichts besonderes tun: der Compiler inferiert selbstständig immer den allgemeinsten Typ, so dass für die Funktion `f x y = [x]` der Typ `a -> b -> [a]` inferiert wird.
- Um noch ein beliebtes Missverständnis aufzuklären: Für einen *Term* `e` ist `[e]` der Term der einelementigen Liste, für einen *Typ* `a` ist dagegen `[a]` der Typ aller Listen (beliebiger Länge!) mit Elementen von Typ `a`. Für `e :: a` gilt insbesondere `[e] :: [a]`, aber auch `[e, e] :: [a]`. `[a, a]` dagegen ist kein valider Typ.

1. Geben Sie je eine Haskell-Definition `f` für folgende Werte an:
   (a) Die wohldefinierte[4] Funktion des Typs `a -> a`
   (b) Die wohldefinierte Funktion des Typs `(a -> b) -> a -> b`
   (c) Die wohldefinierte Funktion des Typs `(a -> b) -> (a -> b)`
   (d) Die wohldefinierte Funktion des Typs `a -> b -> a`
   (e) Alle wohldefinierten, semantisch unterschiedlichen Funktionen des Typs `a -> a -> a`
   (f) Die leere Liste vom Typ `[a]`
   (g) Eine einelementige Liste vom Typ `[Int]`
   (h) Die leere Liste vom Typ `[[a]]`
   (i) Die Liste des Typs `[[a]]`, welche *nur die leere Liste* enthält

   Beispiellösung:

[3] Das ist eine Typsignatur, `::` liest man als „hat Typ"
[4] „Terminiert auf allen Eingaben"

(a) `f x = x`

(b) `f x = x`, Alternativ `f g x = g x`

(c) `f x = x`

(d) `f x y = x`

(e) `f x y = x`, `g x y = y`

(f) `f = []`

(g) `f = [42]`

(h) Wie man die leere Liste schreibt ist unabhängig davon, wie ihre (nicht vorhandenen) Elemente aussehen könnten: `f = []`

(i) `f = [[]]`

2. Warum gibt es keinen Wert, der für alle Typen `a` eine einelementige Liste vom Typ `[a]` darstellt?

   Beispiellösung: Die einelementige Liste müsste als einziges Element einen Wert enthalten, der *alle* Haskell-Typen bewohnt. Solch ein Wert existiert nicht in Haskell[5].

   In Java wäre z.B. `null` ein Wert, der alle Referenztypen (also auch generische Typen) bewohnt.

3. Warum geht genau dies jedoch für eine einelementige Liste mit Typen `[[a]]`?

   Beispiellösung: Weil man keinen Wert von Typ `a` angeben muss, um `[[]] :: [[a]]` zu schreiben.

4. Geben Sie den Term vom Typ `[[[[[[[[[[a]]]]]]]]]]` mit der kürzesten Schreibweise an. Falls Sie jetzt Klammern zählen wollen, denken Sie nochmal über 1. (f), (h) und (i) nach.

   Beispiellösung: `[]`

[5] Nichttermination und Variationen wie `undefined` fassen wir nicht als Wert auf.