Christoph Lüth
Bernd Gersdorf
Christian Maeder
René Wagner
Dennis Walter
Diedrich Wolter
Simon Ulbricht

4. Übungsblatt

Ausgabe: 25.11.10
Abgabe: 06.12.10

4.1 *Fingerübungen mit Funktionen höherer Ordnung* *10 Punkte*

In dieser Übung vertiefen Sie Ihr Verständnis für einige kanonische Funktionen höherer Ordnung, indem Sie diese nachimplementieren bzw. in einfachen, aber typischen Anwendungsfällen einsetzen.

1. (2 Punkte) Stellen Sie folgende *list comprehensions* mittels `map` und `filter` dar:

```
x1 :: [Integer]
x1 = [ x * x | x ← [1..10], even x ]

x2 :: [(Integer, Double)]
x2 = [ (x, sqrt (fromInteger x)) | x ← x1, x > 10, x < 100 ]
```

2. (6 Punkte) Die Vorlage `Blatt04.hs` enthält zwei Funktionen `trim` und `splitIt` zum Entfernen von Leerzeichen am Rand von Strings bzw. zum Zerlegen einer Liste in durch Separatoren getrennte Sequenzen. Zusätzlich wird eine Funktion `nub` importiert, die Duplikate aus einer Liste entfernt. Verwenden Sie diese Funktionen, um eine Funktion `taggify` mit den folgenden Eigenschaften für alle `s, s1, s2, s3 :: String` zu implementieren:

```
map trim (taggify s) == taggify s                                    — (1)
not (elem "" (taggify s))                                            — (2)
not (elem ',' (concat (taggify s)))                                  — (3)
let tags = taggify s in
  null tags || and (zipWith (<) tags (tail tags))                    — (4)
let s2' = trim s2
    s' = s1 ++ s2' ++ s3 in
  null s2' || elem ',' s2' || or (map (sublist s2') (taggify s'))    — (5)
```

   Der Typ dieser Funktion ergibt sich unmittelbar aus obigen Eigenschaften.[1] Typische Tag-Strings `s` sind etwa

   `"semantic web, tool, download"` oder `" pflanzen, zoologie, lexikon ,a,a,"`

   (a) Erläutern Sie in Ihrer Dokumentation die Bedeutung der oben angegebenen Eigenschaften (1) bis (5) und entwerfen Sie Testfälle, die diese Eigenschaften überprüfen.

   (b) Implementieren Sie diese Funktion als Komposition von Funktionen, die jeweils Teilaufgaben erledigen (auch unter Verwendung von `map` und `filter`), dem folgenden Beispiel folgend: (die Aufrufreihenfolge hat hier keinen Bezug zur Lösung):

```
foo x = (f1 . map f2 . filter p3 . f4) x
  where f1 x = ?
        f2 x = ?
        ...
```

3. (2 Punkte) Implementieren Sie eine Funktion

```
until :: (a → Bool) → (a → a) → a → a
```

   die zwei Funktionsargumente `p` und `f` erwartet und `f` wiederholt auf einen gegebenen Eingabewert `x :: a` anwendet, bis `p` für den Endwert zu `True` auswertet. Verwenden Sie diese Funktion, um den niedrigsten Zahlenwert zu ermitteln, dessen `chr`-Interpretation `isLetter` erfüllt. (Hinweis: es geht um den Buchstaben 'A')

[1] Beachten Sie insbesondere die Typgleichheit zwischen `String` und `[Char]`.

4.2 *Alter Falter!* — *10 Punkte*

In dieser Übung soll das Verständnis für die Anwendung von `foldr` bzw. `foldl` vertieft werden. Definieren sie alle der nachfolgend aufgeführten Funktionen mittels `foldr` gemäß dem folgenden Schema:

```
f :: t1 → ... → tN → [u] → v
f x1 ... xN xs =
  let h x r = ? in
  g (foldr h ? xs ⟨ggf. weitere Argumente⟩)
```

Hierbei wird das "Rekursionsergebnis" `r` ggf. von komplexem Typ sein, z.B. ein Tupel `(t1, t2)`. `g` wird in den meisten Fällen `id` sein, also nichts tun. Für jede nachfolgende Aufgabe gibt es 2 Punkte.

1. Zum Warmwerden: Zählen der Vorkommen von Elementen mit einer bestimmten Eigenschaft (`countOccs`); anschließend Verwendung dieser Funktion zum Auffinden des Strings mit den meisten Vorkommen eines Buchstabens (`mostOccs`):

   ```
   countOccs :: (a → Bool) → [a] → Integer
   mostOccs :: Char → [String] → (Integer, String)

   countOccs (\ c → c == 'c') "Cappuccino" == 2
   countOccs (\ c → c == 'e') "Cappuccino" == 0

   mostOccs 'c' ["Cappuccino", "Kaffee Hag", "Nescafe"] == (2, "Cappuccino")
   ```

2. Die Klassiker (aus der Vorlesung bekannt):

   ```
   map :: (a → b) → [a] → [b]
   filter :: (a → Bool) → [a] → [a]
   elem :: Eq a ⇒ a → [a] → Bool
   ```

3. Die Nützlichen: Lesen (mit Default-Wert) und Setzen in einer Assoc-Liste (Liste aus Paaren von Schlüsseln und Werten):

   ```
   aget :: Eq a ⇒ a → b → [(a, b)] → b
   aput :: Eq a ⇒ a → b → [(a, b)] → [(a, b)]
   ```

   Hierbei soll für alle Eingaben `x, y, z` und alle "gültigen" Listen `xs` (s. nächster Punkt) gelten:

   ```
   not (elem (x, y) xs) || aget x z xs == y     — (1)
   elem x (map fst xs) || aget x z xs == z      — (2)
   aput x z (aput x y xs) == aput x z xs        — (3)
   aget x y (aput x z xs) == z                  — (4)
   ```

   Beachten Sie: (1) und (2) spezifizieren das Verhalten bzgl. des Default-Wertes. (3) fordert, dass `aput` keine Duplikate erzeugt, (4) stellt sicher, dass geschriebene Werte anschließend gelesen werden können. Überprüfen Sie diese Eigenschaften mittels geeigneter Testfälle.

4. Die Nebenbedingung: gültige Assoc-Listen enthalten keine doppelten Schlüssel (1. Komponente):

   ```
   noDup :: Eq a ⇒ [(a, b)] → Bool

   noDup [(1, 0), (2, 0)] && not (noDup [(1, 2), (2, 4), (1, 3)])
   ```

   Hinweis: (auch) hier ist `g /= id`!

5. (∗∗) Die harte Nuss: Definieren Sie `foldl` mittels `foldr`:

   ```
   myfoldl :: (a → b → a) → a → [b] → a
   myfoldl f a xs =
     let h x r b = ? in
     foldr h ? xs ?
   ```

Sollten Sie Schwierigkeiten mit der direkten Umsetzung via `foldr` haben, so erstellen Sie zunächst primitiv rekursive Versionen der Funktionen und übertragen diese anschließend. Insbesondere die Definition von `aput` ist wegen der geforderten Vermeidung von Duplikaten nicht ganz so offensichtlich.