Dr. Steffen Jost
Leah Neukirchen

Ludwig-Maximilians-Universität München
Institut für Informatik
23. Mai 2018

Lösungsvorschlag zur 06. Übung zur Vorlesung

Programmierung und Modellierung

Hinweis: Aufgrund des Feiertages entfallen die Übungen am Mittwoch, Donnerstag & Freitag (30.5.–1.6.18). Das Übungsblatt erscheint ab 4.6.18 immer am Dienstag.

A6-1 *Listenverarbeitung höherer Ordnung*

a) Ersetzen Sie die List-Comprehension in der folgenden Definition durch den Einsatz der Funktionen `map` und `filter` aus der Standardbibliothek:

```
foo1 f p xs = [f x | x <- xs, x >= 0, p x]
```

LÖSUNGSVORSCHLAG:

```
foo2 f p xs = map f (filter p (filter (>=0) xs))

foo3 f p xs = map f (filter (\x-> p x && x >=0) xs)

foo4 f p = map f . filter p . filter (>=0)
```

Hinweis: Alle drei Versionen sind nahezu gleich effizient mit GHC, wenn alle Optimierungen eingeschaltet sind (Stichwort: “Fusion”).

b) Die Funktion `dropWhile :: (a -> Bool) -> [a] -> [a]` aus der Standardbibliothek entfernt so lange Element vom Anfang einer Liste, so lange diese das gegebene Prädikat erfüllen. Implementieren Sie diese Funktion unter den Namen `myDropWhile` selbst mit Rekursion, ohne die Verwendung von Bibliotheksfunktionen.
Beispiel: `dropWhile (<4) [1,3,4,5,3,1] == [4,5,3,1]`

LÖSUNGSVORSCHLAG:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] =  []
dropWhile p xs@(x:xs')
  | p x        =  dropWhile p xs'
  | otherwise  =  xs
```

Wer `@`-Patterns nicht kann, schaut entweder auf Folie 2.32 nach oder schreibt alternativ:

```
dropWhile2 :: (a -> Bool) -> [a] -> [a]
dropWhile2 p (x:xs) | p x = dropWhile2 p xs
dropWhile2 _ l = l
```

In der Standardbibliothek findet man auch oft Definitionen im folgenden Stil, was den Vorteil hat, dass bei der Rekursion ein Argument entfällt, was dann geringfügig effizienter ist:

```
dropWhile3 :: (a -> Bool) -> [a] -> [a]
dropWhile3 p = dW3aux
  where dW3Aux (x:xs) | p x = dW3Aux xs
        dW3Aux    xs        = xs
```

c) Die Funktion `all :: (a -> Bool) -> [a] -> Bool` aus der Standardbibliothek gibt nur dann `True` zurück, wenn alle Element der Liste das übergebene Prädikat erfüllen. Implementieren Sie die Funktion unter dem Namen `myAll` selbst ohne direkte Rekursion, sondern unter Verwendung Funktionen höherer Ordnung aus der Standardbibliothek.

LÖSUNGSVORSCHLAG:

```
all1 p xs = foldr aux True xs
  where
    aux x acc
      | p x       = acc
      | otherwise = False

-- Etwas punktfreier ohne xs geht es hier auch:
all2 p = foldr aux True
  where
    aux x acc
      | p x       = acc
      | otherwise = False

-- Mit anonymer Funktion geht es etwas kompakter:
all3 p = foldr (\x acc -> p x && acc) True

-- Wer sich in der Standardbibliothek gut auskennt,
-- schreibt ganz Punkt-frei mit Punkt-Operator einfach:
all4 p = and . map p
-- oder auch mit dropWhile
all5 p = null . dropWhile p
```

Man könnte auch analog das endrekursive `foldl` verwenden. Allerdings ist `foldr` hier oft effizienter, da die Berechnung ja sofort abbricht, so bald ein Listenelement das Prädikat `p` nicht erfüllt. Für `foldl` gilt das zwar prinzipiell auch, aber `foldl` beginnt am Ende der Liste und muss diese dadurch trotzdem komplett durchlaufen.

A6-2 ***Funktionen höherer Ordnung***

a) Implementieren Sie folgende Funktionen, analog zu (`un`)-`curry` und (`un`)-`curry3`:

```
curry4   :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
uncurry4 :: (a -> b -> c -> d -> e) -> (a, b, c, d) -> e
```

Hinweis: Die Typsignaturen lassen bei der Implementation einer totalen Funktion hier schon keine Wahl mehr zu, wenn man auf Schummeleien wie `undefined`, `error` oder endlose Rekursion verzichtet.

LÖSUNGSVORSCHLAG:

```
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
curry4   f  a b c d = f (a,b,c,d)

uncurry4 :: (a -> b -> c -> d -> e) -> (a, b, c, d) -> e
uncurry4 f (a,b,c,d) = f  a b c d
```

b) Diskutieren Sie anhand des Typs den Unterschied zwischen folgenden drei Funktionen:

i) `uncurry3 foldr`

ii) `uncurry (uncurry foldr)`

iii) `(uncurry . uncurry) foldr`

LÖSUNGSVORSCHLAG:

```
uncurry3 foldr            :: ( a -> c -> c, c , [a]) -> c
uncurry (uncurry foldr)   :: ((a -> c -> c, c), [a]) -> c
(uncurry . uncurry) foldr :: ((a -> c -> c, c), [a]) -> c
```

b)i und b)ii unterscheiden sich lediglich in der Gruppierung der Argumente: einmal haben wir die Struktur `(a,b,c)` und einmal `((a,b),c)`. In der Mathematik unterscheidet man oft nicht zwischen diesen beiden Produkten. In einer Programmiersprache kann man eine Unterscheidung aber nur schwerlich vermeiden, schon allein wegen der Unterschiedlichen Kodierung im Speicher des Computers, welche je nach Verwendung notwendig ist.

Die Funktionen b)ii und b)iii sind dagegen tatsächlich identisch. Es ist ja generell egal, ob man zuerst zwei Funktion komponiert und dann auf ein Argument anwendet, oder ob man die eine Funktion zuerst auf das Argument anwendet und die andere Funktion danach auf das Ergebnis.

Achtung: Die Klammern in der Definition b)iii sind wichtig, ansonsten kommt eine recht verrückte Funktion dabei heraus.

A6-3 ***Compose*** Alois Dimpfelmoser möchte eine Funktion programmieren, welche die Summe der Quadrate aller geraden Zahlen aus einer Liste berechnet. Weil Alois der pointfree-Stil so gut gefällt (komischerweise sogar besser als List-Comprehension), hat er unter Verwendung von `compose` aus Folie 6.26 folgendes dazu implementiert:

```
geradequadratsumme  =  compose [sum, map (^2), filter even]
```

Leider mag GHC diese Definition nicht! Helfen Sie dem armen Alois!

Wo liegen der/die Fehler? Wie lautet die richtige Definition im pointfree-Stil?

Hinweis: Die Verwendung von `compose` könnte hier der falsche Ansatz sein.

LÖSUNGSVORSCHLAG:

Alle Werte einer List müssen immer den gleichen Typ haben, prüfen wir also mal die Typen nach:

```
> :t sum
sum :: Num a => [a] -> a
> :t map (^2)
map (^2) :: Num b => [b] -> [b]
> :t filter even
filter even :: Integral a => [a] -> [a]
```

(Natürlich brauchen wir hier eigentlich GHCI nicht dazu, weil wir die Typen dieser grundlegenden Funktionen ohnehin im Kopf haben!)

Die Typen von `map (^2)` und `filter even` sind kompatibel: beides sind einstellige Funktionen, welche eine Liste als Argument nehmen und eine Liste gleichen Typs zurückgeben. Der Typ der Listenelemente muss sowohl in der Typklasse `Num` als auch `Integral` liegen, aber das ist kein Problem, da `Integral` ja ohnehin eine Unterklasse von `Num` ist.

Der Typ von `sum` passt aber nicht dazu, weil als Ergebnis eine Zahl und keine Liste zurückgeben wird.

Schauen wir uns nun den Typ von `compose` an: `[a -> a] -> a -> a`. Auch dieser Typ verträgt sich mit der Liste `[map (^2), filter even]`, denn wir können den Typ spezalisieren zu `Integral b => [[b] -> [b]] -> [b] -> [b]`. Somit haben wir `compose [map (^2), filter even] :: Integral b => [b] -> [b]`. Jetzt müssen wir nur noch sie Summierung durchführen, z.B:

```
geradequadratsumme xs = sum (compose [map (^2), filter even] xs)
```

Das wäre aber nicht im pointfree-Stil (da `xs` hier der Punkt ist). Also benutzen wir die Hintereinanderausführung von Funktionen, deren Typ hier gut passt:

```
geradequadratsumme :: [Int] -> Int
geradequadratsumme  = sum . compose [map (^2), filter even]
geradequadratsumme' = sum . map (^2) . filter even  -- direkt ohne compose
```

Ende der Lösungsvorschläge für die Präsenzaufgaben.
Lösungsvorschläge für die Hausaufgaben folgen nach Ende der Abgabezeit.

H6-1 ***I need a Dollar*** (2 Punkte; Abgabe: `H6-1.txt` oder `H6-1.pdf`)
Ein klammer Kleptomane hat alle `$` geklaut! Fügen Sie in die nachfolgenden Haskell-Ausdrücke wieder `$` ein, so dass jeder Ausdruck zu `42` auswertet!

Hinweis: Es ist ausschließlich `$` einzufügen; sonst nichts, auch keine Klammern! Die Infix-Funktion `($)` wurde auf Folie 6.29 besprochen. Wem unklar ist, wie die Aufgabe anzugehen ist, kann die ersten beiden Teilaufgaben auch zuerst durch Einfügen von runden Klammern lösen, und diese dann erst anschließend wieder durch `$` ersetzen; bei den letzten beiden Teilaufgaben geht das aber nicht mehr — warum? (Unterschied zwischen den ersten beiden Teilaufgaben und den letzten beiden Teilaufgaben mit 1–3 Sätzen erklären.)

a) `gcd 210 28 * 2 * 3`

b) `sum filter odd [2..8] ++ [6..12]`

c) `(3)(*14)`

d) `(foldr) (6) (6) [(-),(*),(-),(+)]`

H6-2 ***Falten*** (2 Punkte; Datei `H6-2.hs` als Lösung abgeben)
Definieren Sie die Funktionen `myLength :: [a] -> Int` und `myReverse :: [a] -> [a]` ohne direkte Rekursion, sondern nur unter Verwendung von `foldl`. Die Funktionen sollen so funktionieren wie `length` und `reverse` aus der Standardbibliothek.
Zur Erinnerung: `foldl :: (b -> a -> b) -> b -> [a] -> b`

H6-3 ***Monoid für MyList*** (2 Punkte; Datei `H6-3.hs` als Lösung abgeben)
Implementieren Sie die Instanzdeklaration `Monoid (MyList a)` für den Datentyp aus H4-3 und A5-1:

```
data MyList a = Leer | Element a (MyList a)     deriving Show
```

Achten Sie darauf, dass Ihre Instanzdeklaration alle geforderten Gesetze für `Monoid` erfüllt!

H6-4 ***Einige Funktor*** (2 Punkte; Datei `H6-4.hs` als Lösung abgeben)
Implementieren Sie die Instanzdeklaration `Functor Einige` für folgenden Datentypen:

```
data Einige a = Keins | Eins a | Zwei a a | Drei a a a
  deriving (Eq, Ord, Show)
```

Hinweise:

- Wir überlegen uns zuerst, welchen konkreten Typ die geforderte Funktion `fmap` dabei hat: `fmap :: (a -> b) -> (Einige a) -> (Einige b)`
- In der Instanzdeklaration ist zu beachten, dass `Functor` als Argument einen Typkonstruktor mit Kind `* -> *` erwartet. (Was heisst das hier?)
- Der Datentyp `Einige` kann als Erweiterung von `Maybe` aufgefasst werden, d.h. wir können uns an dessen Implementation in Foliensatz 7 gut orientieren.

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 2.6.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 3 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ss-2018/promo/`).