Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
20. Mai 2014

5. Musterlösung zur Vorlesung

Programmierung und Modellierung

A5-1 *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)
```

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 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
```

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 ohne direkte Rekursion, sondern unter Verwendung Funktionen höherer Ordnung aus der Standardbibliothek.

LÖSUNGSVORSCHLAG:

```
all p = and . map p

-- Varianten mit foldr
all1 p = foldr aux True
  where
    aux x acc
      | p x       = acc
      | otherwise = False

all2 p = foldr (\x acc -> p x && acc) True
```

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.

d) Die Funktion `concatMap :: (a -> [b]) -> [a] -> [b]` entspricht der Definition

```
> let concatMap1 f = concat . map f
concatMap1 :: (a -> [b]) -> [a] -> [b]
> concatMap1 (\x -> replicate x x) [1..5]
[1,2,2,3,3,3,4,4,4,4,5,5,5,5,5]
```

Implementieren Sie `concatMap` selbst, so wie es die Standardbibliothek tut: ohne Verwendung der Funktionen `concat` und `map`; dafür mit Verwendung der Funktionen `(++)`, `(.)` und `foldr`.

LÖSUNGSVORSCHLAG:

```
-- | Map a function over a list and concatenate the results.
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f  =  foldr ((++) . f) []
```

A5-2 ***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-Comprehensions), hat er unter Verwendung von `compose` aus Folie 7-25 folgendes dazu implementiert:

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

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

a) Wo liegen der/die Fehler?

b) Wie lautet die richtige Definition im pointfree-Stil?

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, der Typ hier gut passt:

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

A5-3 ***Falten*** Der Datentyp `Ziffer` sei wie folgt definiert:

```
data Ziffer = Null | Eins  | Zwei   | Drei | Vier
            | Fünf | Sechs | Sieben | Acht | Neun
  deriving (Show, Read, Eq, Ord, Enum, Bounded)
```

a) Schreiben Sie eine Funktion `ziffer2integer :: [Ziffer] -> Integer`, welche eine Liste von Ziffern in eine Zahl umwandelt: `ziffer2integer [Eins,Zwei,Neun] = 129` Verwendung Sie dazu mindestens einer der in der Vorlesung vorgestellten Faltungsfunktionen, also `foldl`, `foldr`, etc. Auch Funktionen der Klasse `Enum` könnten nützlich sein.

LÖSUNGSVORSCHLAG:

Wir sollten in dieser Aufgabe die Faltungsfunktionen einsetzen. Dazu werden wir eine Funktion brauchen, welches das vorläufige Zwischenergebnis und das Kopfelement der noch zu bearbeitenden Liste nimmt.

Wenn die Liste leer ist, wird diese Funktion aber gar nicht von `foldr` oder `foldl` aufgerufen, sondern liefert sofort den Startwert zurück. Damit wissen wir wie dieser Startwert aussehen muss, denn die leere Liste entspricht sicherlich der Zahl 0.

Wir können also schon mal direkt hinschreiben:

```
ziffer2integer xs = fold_?_ hilfsfunktion 0 xs
  where
    hilfsfunktion :: Integer -> Ziffer -> Integer
    hilfsfunktion vorläufigesErgebnis nächsteZiffer = _?_
```

Wir wissen noch nicht, ob wir nach links oder recht falten, und noch nicht, wie der Rumpf der Faltungsfunktion aussehen soll – doch deren Typ kennen wir schon, und können deren Grundgerüst direkt anlegen.

Die Faltungsfunktion muss sicherlich irgendwie eine einzelne Ziffer in einen Integer umwandeln. Das ist eine eigene Aufgabe, welche wir besser in einer eigenen Funktion behandeln. Ohne viel Nachdenken können wir dazu programmieren:

```
oneziffer2integer :: Ziffer -> Integer
oneziffer2integer Null = 0
oneziffer2integer Eins = 1
oneziffer2integer Zwei = 2
  ...
```

Das funktioniert auch. Mit Nachdenken können wir jedoch unsere Tippfinger schonen, denn wir sehen das `Ziffer` der Typklasse `Enum` angehört. Diese Typklasse bietet bereits eine Konversion nach `Int`. Mit der Funktion `fromIntegral` können wir `Int` in `Integer` umwandeln. Wir schreiben also den kurzen punktfreien Einzeiler:

```
oneziffer2integer :: Ziffer -> Integer
oneziffer2integer = fromIntegral . fromEnum
```

Zurück zur Faltungsfunktion für unser eigentliches Problem. Wenn wir den Anfang der Ziffern-Liste bereits in einen `Integer` umgewandelt haben, so müssen wir diese Zahl nur noch mit 10 multiplizieren und die aktuelle Ziffer (nach Umwandlung) dazu addieren:

```
hilfsfunktion :: Integer -> Ziffer -> Integer
hilfsfunktion vorläufigesErgebnis nächsteZiffer =
  vorläufigesErgebnis * 10 + (oneziffer2integer nächsteZiffer)
```

Da wir angenommen haben, dass wir den Anfang der Liste schon durchgearbeitet haben, wissen wir nun auch, dass wir nach Links falten wollen.

Dieser Code funktioniert bereits, doch wir können das noch etwas verschönern und punktfrei gestalten:

```
ziffer2integer :: [Ziffer] -> Integer
ziffer2integer = foldl z2i 0
  where
    z2i acc z = 10*acc + oneziffer2integer z

oneziffer2integer :: Ziffer -> Integer
oneziffer2integer = fromIntegral . fromEnum
```

b) Schreiben Sie eine Funktion `integer2ziffer :: Integer -> [Ziffer]`, so dass `integer2ziffer . ziffer2integer` sich für positive Zahlen verhält wie die Funktion `id`. Verwenden Sie dazu folgende rekursive Funktion höherer Ordnung:

```
unfold p h t x | p x = []
               | otherwise = let construct = unfold p h t $ t x
                             in h x : construct
```

LÖSUNGSVORSCHLAG:

Als erstes muss man mal die Funktion `unfold` verstehen, welche *nicht* in der Standardbibliothek drin ist: das erste Argument ist ein Prädikat, was die Abbruchbedingung der rekursion darstellt; das zweite Argument ist eine Funktion, welche als letzte auf jedes Element des Ergebnisses angewandt wird; das dritte Argument ist eine Funktion, die sagt wie in der Rekursion weitergezählt wird; das vierte Argument ist der Startwert. Wie immer hilft auch der Typ dem Verständnis weiter:

```
unfold :: (t -> Bool) -> (t -> a) -> (t -> t) -> t -> [a]
```

Wer will kann auch einfach mal damit herumspielen:

```
> unfold (>7) show (+2) 3
["3","5","7"]
```

Auch hier brauchen wir wieder eine Funktion, welche eine einzelne Ziffer konvertiert, dieses Mal jedoch anders herum. Wir haben wieder zwei Möglichkeiten, welche ich dieses mal unnötigerweise in eine einzelne Definition hinein packe (Warum funktioniert der Code genau so wie hier abgedruckt?):

```
i2z :: Integer -> Ziffer
i2z 0 = Null
i2z 1 = Eins
--     etc.
-- Oder wieder als Einzeiler:
i2z x = toEnum $ fromIntegral  x
```

Die letzte Ziffer einer Zahl erhalten wir mit modulo 10, die letzte Ziffer einer Zahl vergessen wir mit ganzzahlig durch 10 teilen. Das "Weiterschalten" erledigen wir also mit `(\x -> x `div` 10)`. Das Ergebniselement erhalten wir durch die Kombination von Modulo und Konvertierung einer einstelligen Zahl: `(\z -> i2z (z `mod` 10))`.

Jetzt müssen wir noch alles zusammenstecken. Der Startwert ist klar: das ist der gegebene Integer. Der Abbruch ist auch klar: sobald der Integer kleiner oder gleich Null ist. Wir stellen aber fest, dass wir die Ziffern hier in der falschen Reihenfolge berechnen, weshalb wir am Ende das Ergebnis noch einmal mit `reverse` herumdrehen müssen. Auch den Sonderfall eine nicht-positiven Zahl müssen wir noch extra behandeln.

```
integer2ziffer :: Integer -> [Ziffer]
integer2ziffer x
  | x <= 0    = [Null]
  | otherwise = reverse $ unfold (<=0) i2zAux (`div` 10) x
  where
    i2zAux :: Integer -> Ziffer
    i2zAux = toEnum.fromIntegral.(`mod` 10)
```

Kommentar:
Man kann streiten, ob das wirklich gut lesbar ist. Ob man eine Funktion wie `unfold` wirklich im täglichen Gebrauch anwenden möchte, bleibt wohl Geschmackssache – das Lesen von Code und Faltungen können wir aber ganz gut daran üben. Auf die schnelle würde ich das gegebene Problem vielleicht mit `show` und `read` implementieren, was aber sicherlich fürchterlich ineffizient wäre – aber manchmal reicht es ja auch, schnell ein korrektes Programm zu haben, anstatt nach langer Zeit Entwicklungszeit ein schnelles aber möglicherweise fehlerhaftes Programm zu haben.

```
integer2ziffer' :: Integer -> [Ziffer]
integer2ziffer' n = foldr i2z' [] $ show n
  where i2z':: Char -> [Ziffer] -> [Ziffer]
        i2z' z acc = (toEnum $ read $ [z]) : acc
```

H5-1 ***Die Eule*** **(0 Punkte)** (`.hs`-Datei als Lösung abgeben)
Manche Leute sagen, der durch `((.)$(.))` definierte Infix-Operator ist *pointfree*; manche Leute sagen, er ist *pointless*; ich aber sage Euch: Implementiert diesen Operator in herkömmlicher Weise *mit Punkten*!

LÖSUNGSVORSCHLAG:
Wie immer gilt: Wer den Typ versteht, versteht auch was passiert!

```
> :t ((.)$(.))
((.)$(.)) :: (a -> b -> c) -> a -> (a1 -> b) -> a1 -> c
```

Die entfernt an eine Eule erinnernde Kombination von Hintereinanderausführung und Funktionsanwendung nimmt also eine zweistellige Funktion; dann ein Argument, was als erstes Argument der zweistelligen Funktion passt; dann eine einstellige Funktion, deren Ergebnis als zweites Argument der zweistelligen Funktion passt; und zuletzt ein Argument, dass als Parameter der einstelligen Funktion verwendet werden kann. Der Rückgabetyp ist identisch zur zweistelligen Funktion.
Beispiel

```
> ((.)$(.)) (||) False null []
True
```

Die Typsignatur können wir einfach übernehmen. Dann schreiben wir einfach eine Funktion hin, welche 4 Argumente bekommt. Zuerst benennen wir diese 4 Argumente reflexartig ohne Nachzudenken, d.h. wir schreiben `eule a1 a2 a3 a4 = ...`

Jetzt müssen wir noch den Funktionsrumpf ausformulieren. Hier stecken wir die Argumente einfach gemäß Ihrer Typen zusammen. Da die Funktion ja polymorph in allen Argumenten ist, haben wir da gar keine andere Wahl: wir wissen ja zum Beispiel gar nichts über den Typ von `a2`, außer das wir Werte dieses Typ als Argumente für `a1` verwenden können. Dies ist wirklich das einzige, was wir damit machen können! Das ergibt schon `eule a1 a2 a3 a4 = a1 a2 (a3 a4)` und wir sind fertig.

Zur besseren Verständlichkeit benennen wir in unserer Lösung jetzt noch alle gebundenen Variablen um, aber das ist optional und stellt keinen echten Unterschied dar:

```
eule :: (a -> c -> d) -> a -> (b -> c) -> b -> d
eule g y f x = g y (f x)
```

H5-2 *Abstiegsfunktion IV* (2 Punkte) (Abgabeformat: Text oder PDF)
Beweisen Sie, dass folgende Funktionen für alle Eingaben terminiert. Dabei dürfen Sie annehmen, dass die als Argument übergebene Funktion `f` ihrerseits für alle Argumente terminiert.

```
pam :: (a -> b) -> [b] -> [a] -> [b]
pam f xs []     = xs
pam f xs (y:ys) = pam f ((f y):xs) ys
```

Hinweis: Die Aufgabe ist einfacher, als sie auf den ersten Blick erscheinen mag.

LÖSUNGSVORSCHLAG:

Auf) und Def) sind hier wieder trivial. Zur Verständlichkeit begründen wir diese dennoch hier sehr ausführlich:

Auf) Wir müssen zeigen, dass die Funktion für alle Eingaben terminiert, damit können wir A' nicht weiter einschränken. Noch dazu ist die Funktion polymorph, d.h. wir wissen noch nicht einmal, wie diese Eingaben wirklich aussehen. Wählen wir also einen festen, aber beliebigen Typen für den Rest des Beweises. Da die Definition aber Typ-korrekt ist, wissen wir auch das alle rekursiven Aufrufe wieder auf A' erfolgen, d.h. wir brauchen uns gar nicht weiter darum kümmern.

Def) Das Pattern-Matching ist vollständig: Die Argumente 1 und 2 werden immer mit Variablen gematched, was nie fehlschalgen kann. Lediglich Argument 3 wird gematched: einmal mit der leeren Liste und einmal mit einer beliebigen nicht-leeren Liste (beliebig, weil die Teilpatterns wieder Variablen-Pattern sind). Damit sind alle Fälle abgedeckt.

Neben dem rekursiven Aufruf wird noch `f` und `(:)` verwendet. Die Funktion `f` dürfen wir laut Aufgabenstellungen als wohl-definiert annehmen. Als Konstruktor terminiert `(:)` trivialerweise, es werden ja nur die Argumente in einen Listenknoten verpackt.

Abst) Die Funktion reduziert in jedem rekursiven Schritt die Liste im dritten Argument, wählen wir also als Abstiegsfunktion m die Länge des 3. Argument.

Unsere Abstiegsfunktion muss aber alle Funktionsargumente in die natürlichen Zahlen abbilden. Wir wählen also $m(f, l_1, l_2) = |l_2|$, wobei $|l|$ die Länge der Liste l sei. Die Länge einer Liste ist offensichtlich immer eine natürliche Zahl.

Es findet nur ein rekursiver Aufruf statt. Wir rechnen nach:

$$m(f, l, h : t) = |h : t| = 1 + |t| > |t| = m(f, (f\ h) : l, t)$$

Es dabei völlig egal, was $f\ h$ oder $(f\ h) : l$ wirklich ist, da unsere Abstiegsfunktion dieses Argument ohnehin ignoriert. Wichtig ist nur, dass $(f\ h) : l$ ein akzeptables Argument für die zweite Position in einem Aufruf von `pam` ist, aber das haben wir ja bereits in Auf) und Def) überprüft.

Kommentar: Formale Methoden zum Beweis der Korrektheit eines Programms (Abstiegsfunktion, Hoare-Kalkül, etc.) werden in der Praxis leider nur selten eingesetzt. Es ist wohl oft tatsächlich billiger, Tests durchzuführen und mit verärgerten Kunden zu leben oder Schadenersatz zu zahlen, anstatt einen Informatiker zu bezahlen, welcher der die Korrektheit aufwändig mit Papier & Bleistift beweist und dabei ebenfalls Fehler machen könnte.

Auch wenn Sie später vielleicht keine Abstiegsfunktionen explizit für Korrektheitsbeweise hinschreiben, so sollen Ihnen diese Aufgabe vermitteln, Ihren Blick für die wesentlichen Eckpunkte zu schärfen. Wann immer sie eine rekursive Funktion/Methode in irgendeiner Programmiersprache verfassen, sollten Sie gedanklich folgende Punkte prüfen:

a) Werden die Funktionsargumente bei jedem rekursiven Aufruf in irgendeiner Weise kleiner/einfacher?

b) Sind alle auftretenden Fälle abgedeckt?

H5-3 ***Polynome mit Funktionen modellieren*** **(6 Punkte)** (`.hs`-Datei abgeben)
Eine alternative Repräsentation von Polynomen ist gegeben durch folgende Datentypdeklaration:

```
data PolyFun = PolyFun { coeff :: Int -> Double, degree :: Int }
```

Ein Wert des Typs `PolyFun` besteht also aus zwei Komponenten: eine Funktion, welche den Koeffizienten für den angegebenen Exponenten berechnet und einem `Int`-wert ≥ -1, der den Grad des Polynoms angibt. Der Grad eines Polynoms ist der höchste vorkommende Exponent, d.h. für jedes `p :: PolyFun` soll gelten `(coeff p) (degree p) /= 0`. Das Nullpolynom 0 definieren wir mit `nullpoly = PolyFun {coeff= const 0, degree= -1}`

Beispiel: Das Polynom $1.1x^4 + 2.2x + 3.3$ wird modelliert mit

```
myp = PolyFun { coeff=myc, degree=4 }
   where myc :: Int -> Double
         myc 4 = 1.1
         myc 1 = 2.2
```

```
myc 0 = 3.3
myc _ = 0.0
```

a) Schreiben Sie eine Funktion `polyF2L :: PolyFun -> Polynom`, welche ein Polynom des Typs `PolyFun` in die Repräsentation aus Aufgabe H4-3 überführt!

LÖSUNGSVORSCHLAG:

```
polyF2L :: PolyFun -> Polynom
polyF2L p = Polynom [ coeff p n | n <- [0..degree p] ]
```

Wer die `Show`-Instanz aus Aufgabe H4-3d gemacht hat, kann damit auch leicht eine für `PolyFun` schreiben:

```
instance Show PolyFun where
  show p = show $ polyF2L p
```

b) Schreiben Sie eine Funktion `berechnePolyF :: Double -> PolyFun -> Double` welche den Wert eines Polynoms für einen gegebenen x-Wert berechnet:
`berechnenPolyF PolyFun { degree=2, coeff= \x->if x==2 then 3 else 0 } 5 == 75`

Hinweis: Für die volle Punktzahl auf dieser Teilaufgabe müssen Sie `foldl` oder `foldr` verwenden. Keine Punkte erhalten Sie, wenn Sie `polyF2L` verwenden.

LÖSUNGSVORSCHLAG:

```
berechnePolyF :: Double -> PolyFun -> Double
berechnePolyF x (PolyFun { degree=g, coeff=cf}) = foldl xati 0 [0..g]
  where xati acc i = cf i * x^i + acc

berechnePolyF' :: Double -> PolyFun-> Double
berechnePolyF' x (PolyFun { degree=g, coeff=cf}) = sum [cf i * x^i | i <- [0..g]]
```

c) Schreiben Sie eine Funktion `multPolyFun :: PolyFun -> PolyFun -> PolyFun`, welche zwei Polynome in der Repräsentation des Typs `PolyFun` miteinander multipliziert. Achten Sie auf Randfälle! Sie dürfen verwenden, was Sie möchten, aber es kann zu Punktabzug bei grob ineffizienten Programmen kommen.

LÖSUNGSVORSCHLAG:

```
multPolyFun :: PolyFun -> PolyFun -> PolyFun
multPolyFun p1 p2 = PolyFun { coeff=c, degree=g }
  where
    c n = sum [coeff p1 i * coeff p2 j | i<-[0..n], let j = n-i]
    g | (degree p1 == -1) || (degree p2 == -1) = -1
      | otherwise = (degree p1) + (degree p2)
```

Anstatt `[0..n]` könnte man den Generator der List Comprehension auch noch einschränken auf `[(max 0 $ n - degree p2)..(degree p1)]`.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 27.05.2014, 11:00 Uhr mit UniworX abgegeben werden.

Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.