Ludwig-Maximilians-Universität München
Institut für Informatik
Thomas Prokosch/Prof. Dr. François Bry

29. Juli 2020

Programmierung und Modellierung in Haskell, SoSe 20

Semestralprüfung

Organisatorische Hinweise:

1) **Prüfungsheft erstellen**

   Die Aufgaben müssen mit einem Texteditor in dem zur Verfügung gestellten Prüfungsheft, der Textdatei namens `pruefungsheft.txt`, ausgearbeitet werden.

   Die Aufgabenstellung in dieser Textdatei soll dabei nicht verändert werden.

   Nur das Prüfungsheft als reine Textdatei wird korrigiert. Sonstige Dokumente (wie Notizen auf dem Aufgabenheft, Bilddateien, oder Umwandlungen des Prüfungsheftes als Word- oder PDF-Dokument) werden nicht korrigiert.

2) **Prüfungsheft ausfüllen**

   Tragen Sie zuerst in Ihr Prüfungsheft Ihren Vornamen, Nachnamen und Ihre Matrikelnummer ein. Nur Prüfungshefte, die entsprechend personalisiert sind, werden korrigiert und benotet.

   Ersetzen Sie die Lücken `____` im Prüfungsheft durch Ihre Antworten.

3) **Prüfungsheft abgeben**

   Laden Sie das von Ihnen ausgefüllte Prüfungsheft vor dem Ende der Prüfung um 14 Uhr unter Verwendung Ihres Kontos auf Uni2work `https://uni2work.ifi.lmu.de/` hoch.

   Sollte das Hochladen auf Uni2work unmöglich sein, so laden Sie das Prüfungsheft unter dem Namen `nachname-vorname-matrikelnummer.txt` und unter Benutzung Ihres CIP-Kontos auf den Server `https://mtls1.ifi.lmu.de` hoch.

4) **Eidesstattliche Erklärung unterschreiben und abgeben**

   Korrigiert und benotet werden die Prüfungshefte der Studierenden, die die Erklärung an Eides statt (die Vorlage befindet sich auf Uni2work `https://uni2work.ifi.lmu.de/` neben den Prüfungsunterlagen) bis zum 30. Juli um 18 Uhr auf Uni2work unterzeichnet als PDF-Datei hochgeladen haben.

5) **Entwerten**

   Sie können Ihr Prüfungsheft entwerten, sodass Sie keine Note bekommen, indem Sie in der entsprechend markierten Zeile Ihres Prüfungsheftes den folgenden Satz schreiben: `Ich beantrage, dass mein Prüfungsheft "entwertet", d. h. nicht korrigiert und nicht benotet, wird.`

6) **Hilfsmittel**

   Hilfsmittel sind nicht nötig, können sogar hinderlich sein und sollten nicht verwendet werden. Es ist jedoch sinnvoll, einen Haskell-Editor und einen Haskell-Übersetzer (wie den Glasgow Haskell Compiler ghc) zu verwenden.

Viel Erfolg!

Aufgabe 1 Typsignaturen (1+1+1 Punkte)

a) Geben Sie in Haskell-Notation einen Lambda-Ausdruck an, der den Typ

```
(b -> c) -> (a -> b) -> a -> c
```

hat. Benutzen Sie nicht den Operator `(.)` zur Funktionskomposition.

```
\f g x -> _________ (_________ _________)
```

b) Geben Sie die Haskell-Typsignatur einer beliebigen Funktion nullter Ordnung namens `k` an. Verwenden Sie dabei keinen anderen Typ als **`Int`**.

```
k :: _________
```

c) Geben Sie die Haskell-Typsignatur einer beliebigen Funktion c in curried Form an, deren uncurried Form ein Argument-Paar vom Typ `(a, b)` und als Ergebnis einen Wert vom Typ `a` hat.

```
c :: _________ -> a
```

Aufgabe 2 Rekursion (1+1+1 Punkte)

Das Produkt der ersten n ($n \geq 1$) natürlichen Zahlen kann wie folgt definiert werden:

$$\prod_{i=1}^{n} i = \begin{cases} 1 & \text{wenn } n \leq 1 \\ n \times \prod_{i=1}^{n-1} i & \text{wenn } n \geq 2 \end{cases}$$

a) Geben Sie eine rekursive, aber nicht endrekursive Haskell-Funktion `produkt :: Integral a => a -> a` an, die unmittelbar dieser Definition entspricht.

```
produkt :: Integral a => a -> a

produkt n | _________ = 1

produkt n | _________ = _________ * _________
```

b) Geben Sie eine endrekursive Haskell-Funktion `produkt'` an, die sich wie `produkt` verhält.

```
produkt' :: Integral a => a -> a

produkt' n = prod _________ _________

  where prod n akk = if n == 1 then _________

                     else prod (_________) _________ * _________
```

c) Verwenden Sie lediglich Lambda-Ausdrücke in Haskell-Notation, um eine Berechnung namens `produkt''` zu realisieren, die sich wie `produkt` und `produkt'` verhält.

```
produkt'' :: Integral a => a -> a

produkt'' = \n -> let prod = _________ akk -> if n == 1

                       then _________

                       else prod (n-1) _________ * n

                  in prod _________ _________
```

Aufgabe 3 Auswertung (1+1+1 Punkte)

Gegeben seien die folgenden Haskell-Definitionen:

```
qsum = \a b -> a*a + b*b
wurzel = \n -> if n<0 then 0 else sqrt n
x = \n -> wurzel (qsum 3 4)
y = \n -> wurzel (-4)
y (x 0)
```

a) Geben Sie die Auswertung des Ausdrucks `y (x 0)` in applikativer Reihenfolge entsprechend den obigen Definitionen an.

Erinnerung: Ein Schritt der Auswertung ist die Ersetzung eines Symbols durch dessen Definition oder die Anwendung von Parametern.

Kein Schritt darf fehlen. Geben Sie einen Schritt pro Zeile an. Geben Sie die Umgebung nicht an.

```
0. y (x 0)
1. _________
2. _________
3. _________
```

usw. (Werten Sie vollständig aus!)

b) Geben Sie die Auswertung des Ausdrucks `y (x 0)` in normaler Reihenfolge entsprechend den obigen Definitionen an.

Erinnerung: Ein Schritt der Auswertung ist die Ersetzung eines Symbols durch dessen Definition oder die Anwendung von Parametern.

Kein Schritt darf fehlen. Geben Sie einen Schritt pro Zeile an. Geben Sie die Umgebung nicht an.

```
0. y (x 0)
1. _________
2. _________
3. _________
```

usw. (Werten Sie vollständig aus!)

c) Geben Sie die verzögerte Auswertung des Ausdrucks `(x 0)` entsprechend den obigen Definitionen an.

Erinnerung: Ein Schritt der Auswertung ist die Ersetzung eines Symbols durch dessen Definition oder die Anwendung von Parametern.

Kein Schritt darf fehlen. Geben Sie einen Schritt pro Zeile an. Geben Sie die Umgebung nicht an. Machen Sie die verschiedenen Auswertungsebenen durch Einrückungen ersichtlich.

```
0. x 0
1. _________
2. _________
3. _________
```

usw. (Werten Sie vollständig aus!)

Aufgabe 4 Binärbäume (1+2 Punkte)

Gegeben sei die folgende Definition des polymorphen Datentyps `BB`:

```
data BB a = L | K (BB a) a (BB a)
```

wobei `BB` für Binärbaum, `L` für leerer Baum und `K` für Knoten steht.

a) Formulieren Sie einen Haskell-Wert `w` dieses Typs `BB`, sodass der Prefix-Tiefendurchlauf des Baumes folgende Werteliste ergibt: `["/=","*","2","3","5"]`. Die Listenelemente sind bezüglich ihrer Stelligkeit als Haskell-Syntaxelemente zu verstehen, d.h. die Zeichenkette `"/="` ist als Ungleich-Operator zu verstehen und hat damit die Stelligkeit 2.

```
w = K (K (K L "_________" L) "_________" (K L "_________" L))
      "_________" (K _________ "5" L)
```

b) Geben Sie die Werteliste `l` für den Infix-Tiefendurchlauf an, und werten Sie den damit repräsentierten Ausdruck nach Haskell-Regeln aus.

```
l = ["_________", "_________", "3", "_________", "_________"]
```

Auswertung: _____

Aufgabe 5 Polymorphie (1+1+1 Punkte)

Gegeben sei die folgende (zu Aufgabe 4 identische) Definition des polymorphen Datentyps `BB`:

```
data BB a = L | K (BB a) a (BB a)
```

wobei `BB` für Binärbaum, `L` für leerer Baum und `K` für Knoten steht.

a) Realisieren Sie eine Haskell-Funktion mit der folgenden Typsignatur, die den Baum vom Typ `BB` von rechts nach links mit dem angegebenen zweistelligen Operator auf einen Wert reduziert. Dabei soll die Infix-Notation verwendet werden, das heisst der Lambda-Ausdruck `(\x xs -> concat ["(", show x, xs, ")"])` und der Baum `K (K L 2 L) 1 (K (K L 4 L) 3 L)` angewendet auf `foldrBB` soll den Wert `"(2(1(4(3))))"` ergeben.

```
foldrBB :: (a -> b -> b) -> b -> BB a -> b

foldrBB f b  _________          = b

foldrBB f b (_________ l a r) =

   _________ f (f _________ (foldrBB _________ _________ _________)) _________
```

b) Geben Sie die notwendigen Haskell-Definitionen, um aus dem Typ `BB` unter Verwendung der Funktion `foldrBB` einen Typ der Typklasse `Foldable` zu schaffen. In jeder Lücke darf nur ein Bezeichner wie zum Beispiel `BB`, `Foldable`, `foldrBB` usw. stehen.

```
instance _________ _________ where

  _________ = _________
```

c) Vervollständigen Sie das untenstehende Haskell-Codefragment, welches die Summe der im Binärbaum `b = K (K L 2 L) 1 (K (K L 4 L) 3 L)` enthaltenen Zahlen berechnet.

```
summiere :: Foldable t => t Int -> Int

summiere = _________ (+) 0

sumBB :: Int
sumBB = let b = K (K L 2 L) 1 (K (K L 4 L) 3 L) in summiere b
```

Aufgabe 6 Datentypen (1+1+1 Punkte)

Eine einfach verkettete Liste wird repräsentiert durch eine Menge von Knoten, die jeweils ein Element von einem nicht näher spezifizierten Typ `a` speichern sowie einen weiteren Knoten der Liste. Eine Liste oder ein Teil davon kann auch leer sein, und wird dann durch einen Sentinel (spezieller Knoten ohne Datenelemente) repräsentiert.

a) Definieren Sie in Haskell einen rekursiven Datentyp `LinkedList` mit den Konstruktoren `ListElem` und `ListSentinel`. Verwenden Sie dafür keine eingebauten Datentypen wie Liste, Tupel oder ähnliches.

```
data LinkedList _________ = ListElem _________ (_________  a)
                          | _________;
```

b) Definieren Sie eine Haskell-Funktion `listToLL :: [a] -> LinkedList a`, die eine Liste von Elementen des Typs `a` in die Datenstruktur `LinkedList` überführt.

```
listToLL :: [a] -> LinkedList a

listToLL _________ = ListSentinel

listToLL (_________:xs) = _________ x (_________ _________)
```

c) Erstellen Sie eine Haskell-Funktion

```
showLL :: Show a => LinkedList a -> String
```

mit der der Inhalt einer LinkedList ausgegeben werden kann. Zwischen den Elementen der LinkedList wird als Trennelement die Zeichenkette `">>>"` (ohne Anführungszeichen) eingefügt. Achten Sie darauf, dass bei der Ausgabe die Trennelemente nur zwischen den Elementen vorkommen und nicht vor bzw. nach den Randelementen auftreten. Beispielsweise ist das Ergebnis des Funktionsaufrufes `showLL (listToLL [1..8])` die Zeichenkette `"1>>>2>>>3>>>4>>>5>>>6>>>7>>>8"`.

```
showLL :: Show a => LinkedList a -> String

showLL _________ = ""

showLL (ListElem _________ _________) = show a

showLL (ListElem _________ _________) = concat [_________ a, ">>>", _________ rest]
```

Aufgabe 7 Typklassen (1+1+1 Punkte)

Gegeben sei die folgende Typsignatur.

```
reverse :: (Foldable t, Applicative t, Monoid (t a)) => t a -> t a
reverse ls = ...
```

a) Vervollständigen Sie die Haskell-Funktion **reverse**, die die Werte im faltbaren, applikativen Funktor `t` umkehrt. Zum Beispiel ist das Ergebnis von **reverse** `[1, 2, 3]` im Funktor „Liste“ der Wert `[3, 2, 1]`.

Nehmen Sie keine weiteren Funktionen an außer jenen, die durch die angegebenen Typklassen definiert sind. Beachten Sie, dass die Funktion für beliebige Typen `t` der angegebenen Signatur und nicht nur für Listen funktionieren soll und dass die Funktion für unendliche Wertemengen nicht definiert ist. Verwenden Sie lediglich explizite Parameter.

```
reverse :: (Foldable t, Applicative t, Monoid (t a)) => t a -> t a
reverse ls = _________ (\l r -> r `_________` _________ l) _________ ls
```

b) Unter der zusätzlichen Verwendung der Funktionen `(.)` und **flip** formulieren Sie die obenstehende Funktion als Haskell-Funktion höherer Ordnung ohne explizite Parameter. Verwenden Sie auch keine expliziten Parameter in Lambda-Ausdrücken.

```
reverse :: (Foldable t, Applicative t, Monoid (t a)) => t a -> t a
reverse = foldr (flip _________ . _________) _________
```

c) Kann die obige Funktion gleichermaßen mittels **foldr** und **foldl** umgesetzt werden?

☐ Ja ☐ Nein

Aufgabe 8 Monaden (1+1+1 Punkte)

Gegeben sei das folgende Haskell-Codefragment.

```
data Expr = Num Double
          | Add Expr Expr
          | Sub Expr Expr deriving Show

eval :: Expr -> Maybe Double
eval (Num a) = if a<0 then Nothing else Just a
eval (Add a b) = case eval a of
  Nothing -> Nothing
  Just a' -> case eval b of
    Nothing -> Nothing
    Just b' -> Just (a' + b')
```

a) Formulieren Sie diesen Haskell-Code mit monadischen `do`-Blöcken. Nennen Sie diese Funktion `evalDo`.

```
evalDo :: Expr -> Maybe Double
evalDo (Num a) = if a<0 then Nothing else return a

evalDo (Add a b) = _________

  _________ <- _________ _________

  _________ <- _________ _________
  return (a'+b')
```

b) Formulieren Sie obigen Haskell-Code als monadische Ausdrücke unter Verwendung der Funktionen `return` und bind `(>>=)` anstelle der `do`-Notation. Nennen Sie diese Funktion `evalBd`.

```
evalBd :: Expr -> Maybe Double
evalBd (Num a)   = if a<0 then Nothing else return a

evalBd (Add a b) = _________ a >>= _________ _________ -> _________ _________

                                >>= \b' -> _________ (a' + _________)
```

c) Formulieren Sie eine Haskell-Regel mit `return` und bind `(>>=)`, die es ermöglicht, zwei Ausdrücke voneinander zu subtrahieren. Dabei soll der entstehende Ausdruck nie kleiner als null werden. So ergibt zum Beispiel `evalBd (Num 7 `Sub` Num 5)` den Wert `Just 2.0`, die Auswertung von `evalBd (Num 2 `Sub` Num 3)` jedoch `Nothing`.

```
evalBd :: Expr -> Maybe Double

evalBd (Sub a b) = _________ _________

                   >>= \a' -> _________ _________
                   >>= \b' -> if a'-b'<0

                              then _________

                              else return (_________ _________ _________)
```