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

3. Juli 2021

Programmierung und Modellierung in Haskell, SoSe 21

Probe-Prüfung

Organisatorische Hinweise:

1) **Prüfungsaufgaben ausarbeiten**

   Die Aufgaben müssen auf der Online-Plattform GATE `https://gate.ifi.lmu.de/` ausgearbeitet werden.

   Nur die auf GATE eingetragenen Lösungen werden korrigiert. Sonstige Dokumente (wie Notizen auf dem Prüfungsheft, Bilddateien, oder Umwandlungen des Prüfungsheftes als Word- oder PDF-Dokument) werden nicht korrigiert.

2) **Unterschrift üben und abgeben**

   Die Semestral- und Nachprüfung wird nur von den Studierenden korrigiert und benotet, die eine Eigenständigkeitserklärung abgeben. Dieser Vorgang kann in der Probe-Prüfung mit Hilfe eines Unterschriften-Übungsblattes geübt werden. Die Vorlage dazu befindet sich auf Uni2work `https://uni2work.ifi.lmu.de/` und muss bis spätestens 23:59 Uhr des Probe-Prüfungstages auf Uni2work unterzeichnet als PDF-Datei hochgeladen werden.

3) **Entwerten**

   Sie können Ihr Prüfungsheft entwerten, sodass Sie keine Note bekommen, indem Sie die entsprechend bezeichnete Aufgabe in GATE bearbeiten und Ihre Entscheidung durch Auswahl der Option "Ich beantrage, dass mein Prüfungsheft "entwertet", d. h. nicht korrigiert und nicht benotet, wird." entsprechend kundtun.

4) **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 ->` ___f___ (___g___ ___x___)

b) Geben Sie die Haskell-Typsignatur einer beliebigen einstelligen Funktion `p` mit polymorphem Typ an. Verwenden Sie dabei ausschließlich den polymorphen Typ `a`.

`p ::` ___a -> a___

c) Geben Sie die Haskell-Typsignatur einer beliebigen Funktion `u` in uncurried Form an, deren curried Form die drei Argumente `a`, `b` und `c` und das erste Element als Rückgabewert hat.

`u :: (`___c, b, a___`) -> c`

Aufgabe 2 Rekursion (1+1+1 Punkte)

Das innere Produkt zweier Vektoren $\vec{x}, \vec{y}$ ist definiert als

$$\sum_{i=1}^{n} x_i \cdot y_i.$$

Ein Vektor soll in Haskell durch eine nicht-leere endliche Liste repräsentiert werden.

a) Schreiben Sie eine rekursive, aber nicht endrekursive Haskell-Funktion

```
iprod :: Num a =>[a] -> [a] -> a
```

die das innere Produkt zweier Vektoren ausgibt. Dabei soll bei unterschiedlicher Länge der Vektoren der kürzere Vektor so behandelt werden, als wären seine fehlenden Werte gleich null. Verwenden Sie dafür von den Listenfunktionen nur den Listenkonstruktor `(:)`; `head` und `tail` sind nicht erlaubt.

```
iprod :: Num a => [a] -> [a] -> a

iprod xs [] = 0

iprod [] ys = 0

iprod (x:xs) (y:ys) = x * y + iprod xs ys
```

b) Geben Sie eine endrekursive Haskell-Funktion `iprod'` an, die sich wie `iprod` verhält. Sie dürfen dafür die Listenfunktionen `(:)`, `null`, `head` und `tail` verwenden.

```
iprod' :: Num a => [a] -> [a] -> a

iprod' xs ys = ip xs ys 0
  where ip xs ys akk =

          if null xs || null ys

          then akk

          else ip (tail xs) (tail ys) akk+(head xs)*(head ys)
```

c) Realisieren Sie unter Verwendung der Listenfunktionen `zip` und entweder `foldl` oder `foldr` einen Lambda-Ausdruck in Haskell-Notation, den Sie `iprod''` nennen und der sich sich wie `iprod` und `iprod'` verhält.

```
iprod'' :: Num a => [a] -> [a] -> a

iprod'' = \xs ys -> foldr (\(a, b) r -> r + a * b)
                          0 (zip xs ys)
```

Aufgabe 3 Auswertung (1+1+1 Punkte)

a) Gegeben seien die Haskell-Definitionen

```
index = \n -> -4
f = \n -> if n<0 then n*n else sqrt n
```

und folgende Haskell-Ausdrücke:

```
index (f (f (index 0)))
index (f (f ((\n -> -4) 0)))
```

- Ist dieser Auswertungsschritt korrekt für die applikative Auswertungsreihenfolge?
  ☒ Ja ☐ Nein
- Ist dieser Auswertungsschritt korrekt für die normale Auswertungsreihenfolge?
  ☐ Ja ☒ Nein

b) Gegeben seien die folgenden Haskell-Definitionen:

```
index = \n -> -4
f = \n -> if n<0 then n*n else sqrt n
```

Der Ausdruck `index (f (f (index 0)))` soll in normaler Auswertungsreihenfolge ausgewertet werden. Kreuzen Sie bitte in der folgenden Liste die Antwort an, deren Ausdruck entsprechend dieser Reihenfolge unmittelbar auf den gegebenen Ausdruck folgt:

1. ☐ `index (f (f ((\n -> (-4)) 0)))`
2. ☐ `(-4)`
3. ☒ `(\n -> (-4)) (f (f (index 0)))`
4. ☐ `(\n -> (-4)) (f (f (-4)))`
5. ☐ Keine der obigen Möglichkeiten, korrekt ist: ____

c) Gegeben sei folgende Haskell-Definition:

```
wurzel = \x -> sqrt x
```

Ergänzen Sie den folgenden Auswertungsschritt entsprechend der verzögerten Auswertungsreihenfolge.

```
(\n -> if n<0 then n*n else wurzel n) a
```

`if` a<0 `then` ^a*^a `else` wurzel ^a

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 "__2__" L) "__*__" (K L "__3__" L))
      "__/=__" (K __L__ "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 = ["__2__", "__*__", "3", "__/=__", "__5__"]
```

Auswertung: __True__

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 ___L___ = b

foldrBB f b (___K___ l a r) =

  ___foldr BB___ f (f ___a___ (foldrBB ___f___ ___b___ ___r___)) ___l___
```

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 ___Foldable___ ___BB___ where

  ___foldr___ = ___foldrBB___
```

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 = ___foldr___ (+) 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 ___a___ = ListElem ___a___ (___LinkedList___ a)
                        | ListSentinel;
```

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 (___x___:xs) = ___ListElem___ x (___ListToll___ ___xs___)
```

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 ___ListSentinel___ = ""
showLL (ListElem ___a___ ___ListSentinel___) = show a
showLL (ListElem ___a___ ___rest___) = concat [___show___ a, ">>>", ___showLL___ rest]
```

Aufgabe 7 Typklassen (1+1+1 Punkte)

Gegeben seien die folgenden Typsignaturen.

```
append2 :: (Applicative f, Monoid (f a)) => f a -> f a -> f a
append2 x m = ...

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

a) Vervollständigen Sie die Haskell-Funktion `append2`, sodass der monoidale Wert `x` zweimal von links mit dem monoidalen Wert `m` verknüpft wird. Wenn beispielsweise der applikative Funktor `f` die Liste ist, so soll `append2 [1] [2]` den Wert `[1, 1, 2]` ergeben.

Nehmen Sie keine weiteren Funktionen an außer jenen, die durch die angegebenen Typklassen definiert sind. Verwenden Sie keine zusätzlichen Hilfsfunktionen und lediglich explizite Parameter.

```
append2 :: (Applicative f, Monoid (f a)) => f a -> f a -> f a

append2 x m = x `mappend` x `mappend` m
```

b) Unter der Verwendung der Funktion `append2` vervollständigen Sie die Haskell-Funktion `twice`, sodass alle Werte im faltbaren, applikativen Funktor `t` verdoppelt werden. Beispielsweise ergibt `twice [1, 2, 3]` den Wert `[1, 1, 2, 2, 3, 3]`. Formulieren Sie Hilfsfunktionen als Lambda-Ausdrücke mit ausschließlich expliziten Parametern.

```
twice :: (Foldable t, Applicative t, Monoid (t a)) => t a -> t a

twice ls = foldr (\l r -> append2 (pure l) r) mempty ls
```

c) Unter Verwendung der Funktion `append2` und `(.)` formulieren Sie die Funktion `twice` als Haskell-Funktion höherer Ordnung ohne explizite Parameter. Verwenden Sie auch keine expliziten Parameter in Lambda-Ausdrücken.

```
twice :: (Foldable t, Applicative t, Monoid (t a)) => t a -> t a

twice = foldr (append2 . pure) mempty
```

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) = do
  a' <- evalDo a
  b' <- evalDo 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`. Verwenden Sie zur Ausformulierung von `evalBd` nicht die Funktion `evalDo`.

```
evalBd :: Expr -> Maybe Double
evalBd (Num a)   = if a<0 then Nothing else return a
evalBd (Add a b) = evalBd a >>= \a' ____ -> evalBd b
                      >>= \b' -> return (a' + b')
```

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) = evalBd a
                   >>= \a' -> evalBd b
                   >>= \b' -> if a'-b'<0
                              then Nothing
                              else return (a' - b')
```