Programmierung

05. Übungsblatt

Zeitraum: 09. – 13. Mai 2016

Übung 1 (AGS 12.3.13)
Folgende Definitionen seien gegeben:

```
1  data Tree = Leaf Int | Node Tree Tree
2
3  add :: Tree -> Int
4  add (Leaf a)        = a
5  add (Node t1 t2)    = (add t1) + (add t2)
6
7  sub :: Tree -> Int
8  sub (Leaf a)        = a
9  sub (Node t1 t2)    = (sub t1) - (sub t2)
10
11 neg :: Int -> Tree -> Tree
12 neg i (Leaf a)     = Leaf (a*i)
13 neg i (Node t1 t2) = Node (neg i t1) (neg (-i) t2)
```

Zeigen Sie für die oben aufgeführten Definitionen mit Hilfe der Induktion über Bäume, dass die folgende Gleichung für jeden Baum `t :: Tree` und jede ganze Zahl `i :: Int` erfüllt ist:

```
add (neg i t) = i * (sub t).
```

Geben Sie bei Umformungen die jeweils benutzten Gesetzmäßigkeiten / Definitionen an.

Übung 2 (AGS 12.4.1 ⋆)

(a) Bestimmen Sie für jeden der folgenden λ-Terme t die Mengen $FV(t)$ und $GV(t)$:
- $(\lambda x.x\ y)\,(\lambda y.y)$
- $(\lambda x.(\lambda y.z\ (\lambda z.z\ (\lambda x.y))))$
- $(\lambda x.(\lambda y.x\ z\ (y\ z)))\,(\lambda x.y\ (\lambda y.y))$

(b) Reduzieren Sie die folgenden λ-Terme zu Normalformen. Schreiben Sie – bevor Sie einen Ableitungsschritt ausführen – für die relevanten (Teil-)Ausdrücke die Mengen der freien bzw. der gebundenen Vorkommen von Variablen auf.
- $(\lambda x.(\lambda y.x\ z\ (y\ z)))\,(\lambda x.y\ (\lambda y.y))$
- $(\lambda h.(\lambda x.h\ (x\ x))\,(\lambda x.h\ (x\ x)))\,((\lambda x.x)\,(+\ 1\ 5))$
- $(\lambda f.(\lambda a.(\lambda b.f\ a\ b)))\,(\lambda x.(\lambda y.x))$

Übung 3 (AGS 12.4.32)

(a) Berechnen Sie die Normalform des untenstehenden λ-Terms, indem Sie ihn *schrittweise* reduzieren. Geben Sie dabei vor jedem Schritt für die relevanten Teilausdrücke die Mengen der gebunden bzw. frei vorkommenden Variablen an.

$$(\lambda f\,x.f\,f\,x)(\lambda y.x)z$$

(b) Gegeben sei der λ-Term

$$\langle F\rangle = \Big(\lambda f\,x\,y\,z.\langle\text{ite}\rangle\ \big(\langle\text{iszero}\rangle\ (\langle\text{sub}\rangle\ x\ y)\big) \\ \big(\langle\text{add}\rangle\ y\ z\big) \\ \Big(\langle\text{succ}\rangle\ \big(f\ (\langle\text{pred}\rangle\ x)\,(\langle\text{succ}\rangle\ y)\,(\langle\text{mult}\rangle\ \langle 2\rangle\ z)\big)\Big)\Big).$$

Berechnen Sie schrittweise die Normalform des Terms $\langle Y\rangle\langle F\rangle\langle 6\rangle\langle 5\rangle\langle 3\rangle$. Schreiben Sie für jeden Aufruf von $\langle F\rangle$ jeweils zwei Zeilen: eine in der Sie die Werte der Parameter des Aufrufs protokollieren, und eine in der Sie ihre Auswertung skizzieren. Führen Sie im Rechenprozess zweckmäßige Abkürzungen der λ-Terme ein.

(c) Gegeben sei die folgende Haskell-Funktion:

```
g :: Int -> Int -> Int
g 0 y = 2 * (y + 1)
g x 0 = 2 * (x + 1)
g x y = 4 + g (x - 1) (y - 1)
```

Geben Sie einen λ-Term $\langle G\rangle$ an, so dass $\mathtt{g} = \langle Y\rangle\langle G\rangle$ gilt. Sie dürfen dabei die in der Vorlesung vorgestellten Terme nutzen.

Zusatzaufgabe 1 (AGS 12.4.33 $\star$)

(a) Berechnen Sie die Normalform des untenstehenden λ-Terms, indem Sie ihn *schrittweise* reduzieren. Geben Sie dabei vor jedem Schritt für die relevanten Teilausdrücke die Mengen der gebunden bzw. frei vorkommenden Variablen an.

$$\big(\lambda f\,x.x(f\,(f\,x))\big)(\lambda y.x\,y)$$

(b) Gegeben sei der λ-Term

$$\langle F\rangle = \Big(\lambda f\,x\,y.\langle\text{ite}\rangle\ \big(\langle\text{iszero}\rangle\ y\big)\ \big(\langle\text{mult}\rangle\ x\ \langle 2\rangle\big) \\ \Big(\langle\text{ite}\rangle\ \big(\langle\text{iszero}\rangle\ (\langle\text{mod}\rangle\ y\ \langle 2\rangle)\big)\ \big(f\ x\ (\langle\text{pred}\rangle\ y)\big) \\ \big(f\ (\langle\text{mult}\rangle\ x\ \langle 2\rangle)\ (\langle\text{pred}\rangle\ y)\big)\Big)\Big).$$

Geben Sie eine Haskell-Funktion `f` an, so dass $\mathtt{f} = \langle Y\rangle\langle F\rangle$ gilt!

(c) Betrachten Sie den λ-Term $\langle F\rangle$ aus Teilaufgabe **(b)**. Berechnen Sie die Normalform des Terms $\langle Y\rangle\langle F\rangle\langle 2\rangle\langle 1\rangle$. Schreiben Sie für jeden Aufruf von $\langle F\rangle$ jeweils **zwei Zeilen**: eine in der Sie die Werte der Parameter des Aufrufs protokollieren, und eine in der Sie ihre Auswertung skizzieren.

Zusatzaufgabe 2 (AGS 12.3.19)

Folgende Definitionen seien gegeben:

```
1 data Tree a = Node a (Tree a) (Tree a) | Leaf a
2
3 mirror :: Tree a -> Tree a
4 mirror (Node x t1 t2) = Node x (mirror t2) (mirror t1)
5 mirror (Leaf x) = Leaf x
6
```

```
7  yield :: Tree a -> [a]
8  yield (Node _ t1 t2) = yield t1 ++ yield t2
9  yield (Leaf x) = [x]
```

Die folgende Aussage soll mittels struktureller Induktion über Bäumen bewiesen werden:
(A): Für jeden Typ `a` und jeden Baum `t :: Tree a` gilt

```
reverse (yield t) = yield (mirror t).
```

Nutzen Sie folgende Eigenschaften: Für alle Typen `a`, Werte `x :: a` und Listen `xs, ys :: [a]` gilt
(E1): `reverse [x] = [x]`, und
(E2): `reverse (xs ++ ys) = reverse ys ++ reverse xs`.
Bearbeiten Sie die folgenden Teilaufgaben; geben Sie bei jeder Umformung die benutzte *Definition*, *Eigenschaft* bzw. *Induktionsvoraussetzung* an; quantifizieren Sie alle Variablen.

(a) Zeigen Sie den Induktionsanfang.
(b) Geben Sie die Induktionsvoraussetzung *vollständig* an.
(c) Zeigen Sie den Induktionsschritt.

Zusatzaufgabe 3 (AGS 12.3.21)
Folgende Definitionen seien gegeben:

```
1  dup :: [Int] -> [Int]
2  dup []     = []
3  dup (x:xs) = x : (dup xs ++ dup xs)
4
5  pow :: [Int] -> Int
6  pow []     = 1
7  pow (x:xs) = 2 * pow xs
8
9  length :: [Int] -> Int
10 length []     = 0
11 length (x:xs) = 1 + length xs
```

Die folgende Aussage soll mittels struktureller Induktion über Listen bewiesen werden:
(A): Für jede Liste `xs :: [Int]` gilt:

```
length (dup xs) = pow xs - 1.
```

Bearbeiten Sie die folgenden Teilaufgaben; geben Sie bei jeder Umformung die benutzte *Definition* oder *Beziehung*, bzw. die *Induktionsvoraussetzung* an; quantifizieren Sie alle Variablen.

(a) Zeigen Sie den Induktionsanfang.
(b) Geben Sie die Induktionsvoraussetzung *vollständig* an.
(c) Zeigen Sie den Induktionsschritt. Sie dürfen folgende bereits bewiesene Beziehung nutzen.
(B): Für alle Listen `ys, zs :: [Int]` gilt:

```
length (ys ++ zs) = length ys + length zs.
```