10. Aufgabenblatt vom Freitag, den 10. Januar 2020 zur Vorlesung

ALP I: Funktionale Programmierung

Bearbeiter: A. Rudolph und F. Formanek
Tutor: Stephanie Hoffmann
Tutorium 06

Abgabe: bis Montag, den 20. Januar 2020, 10:10 Uhr

1. **Aufgabe** (5 Punkte)

   Sei a ein beliebiger Ausdruck, bzw eine beliebige Variable
   I a ≡ (λx.x) a = a
   v F a ≡ (λxy.xTy) (λxy.y) a = (λy.(λxy.y)Ty) a = (λxy.y) T a = (λy.y) a = a
   F ¬ a ≡ (λxy.y) ¬ a = (λy.y) a = a
   $\Rightarrow also : I \equiv \vee F \equiv F\neg$

2. **Aufgabe** (6 Punkte)

   E = Gleichheitsfunktion aus der Vorlesung, ¬ = Negationsfunktion aus der Vorlesung
   ≠ ≡ λxy.¬(E x y)
   Beweis mit 2 und 4:
   ≠ 2 4 ≡ (λxy.¬(E x y)) 2 4 = ¬(E 2 4) = ¬F = T

   Wir gehen davon aus, dass das Aussehen und auch die Funktionsweise von E und ¬ aus der Vorlesung bekannt sind und nicht weiter ausgeführt werden müssen.

   G = ≥ und ∧ aus der Vorlesung, ≠ siehe oben
   > ≡ λxy.∧ (G x y) (≠ x y)
   Beweis mit 2 und 4:
   > 2 4 ≡ (λxy.∧ (G x y) (≠ x y)) 2 4 = ∧ (G 2 4) (≠ 2 4) = ∧ (F) (F) = F

   Auch hier gehen wir davon aus, dass das Aussehen sowie auch die Funktionsweise von G und ∧ bekannt sind und nicht weiter ausgeführt werden müssen.

   L = (Z(y P x)), ∧ aus der Volesung und ≠ siehe oben
   < ≡ λxy.∧ (L x y) (≠ x y)
   Beweis mit 2 und 4:
   < 2 4 ≡ (λxy.∧ (L x y) (≠ x y)) 2 4 = ∧ (L 2 4) (≠ 2 4) = ∧ (T) (T) = T

   Hier wird natürlich auch davon ausgegangen, dass ∧ bekannt ist, deswegen wird es mal wieder nicht ausgeführt.

3. **Aufgabe** (4 Punkte)
   λxy.xxy
   Beweis mit TF:
   (λxy.xxy)(λxy.x)(λxy.y) = (λxy.x) (λxy.x) (λxy.y) = (λy.(λxy.x))(λxy.y) = λxy.x ≡ T
   Beweis mit FT:
   (λxy.xxy)(λxy.y)(λxy.x) = (λxy.y) (λxy.y) (λxy.x) = (λy.y) (λxy.x) = λxy.x ≡ T
   Beweis mit TT:
   (λxy.xxy)(λxy.x)(λxy.x) = (λxy.x) (λxy.x) (λxy.x) = (λy.(λxy.x)) (λxy.x) = λxy.x ≡ T
   Beweis mit FF:
   (λxy.xxy)(λxy.y)(λxy.y) = (λxy.y) (λxy.y) (λxy.y) = (λy.y) (λxy.y) = λxy.y ≡ F

   ⇒ Das ist dasselbe verhalten, wie das logische Oder auf Wahrheitswerte angewendet

4. **Aufgabe** (4 Punkte)

   f ≡ Y (λrn.Z n 0 (+ 1 (* 2 (r (- n 1)))))
   Da sowohl +, als auch *, -, Z und Y in der Vorlesung besprochen wurden, gehen wir davon aus, dass sie trivial sind. Zur Auffrischung: + = Addition, * = Multiplikation, - = Subtraktion, Z testet ob eine Zahl = 0 ist und Y ist die Funktion, die uns bei der Rekursion hilft.
   Beweis mit f(4) (Was übrigens unglaublich gemein von Frau Esponda ist):
   Aus Platzsparenden gründen, sei (λrn.Z n 0 (+ 1 (* 2 (r (- n 1))))) definiert als f*
   f 4 ≡ Y f* 4 = f* (Y F*) 4
   ⇒ (λrn.Z n 0 (+ 1 (* 2 (r (- n 1))))) (Y f*) 4
   ⇒ (Z 4 0 (+ 1 (* 2 ((Y f*) 3))))
   ⇒ (+ 1 (* 2 (f* (Y f*) 3)))
   ⇒ (+ 1 (* 2 (((λrn.Z n 0 (+ 1 (* 2 (r (- n 1))))) (Y f*) 3)))
   ⇒ (+ 1 (* 2 (Z 3 0 (+ 1 (* 2 ((Y f*) 2))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (f* (Y f*) 2)))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 ((λrn.Z n 0 (+ 1 (* 2 (r (- n 1))))) (Y f*) 2)))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (Z 2 0 (+ 1 (* 2 ((Y f*) 1))))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (f* (Y f*) 1)))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 ((λrn.Z n 0 (+ 1 (* 2 (r (- n 1))))) (Y f*) 1)))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (Z 1 0 (+ 1 (* 2 ((Y f*) 0))))))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (f* (Y f*) 0)))))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 ((λrn.Z n 0 (+ 1 (* 2 (r (- n 1))))) (Y f*) 0)))))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (Z 0 0 (+ 1 (* 2 ((Y f*) 0))))))))))) ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 0))))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 (+ 1 0)))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 (* 2 1))))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 (+ 1 2)))))
   ⇒ (+ 1 (* 2 (+ 1 (* 2 3))))
   ⇒ (+ 1 (* 2 (+ 1 6)))
   ⇒ (+ 1 (* 2 7))
   ⇒ (+ 1 14)
   ⇒ 15 → Wahre Aussage

5. **Aufgabe** (4 Punkte)

   Wenn eine ganze Zahl (a,b) als a-b definiert ist, dann ist die Subtraktion für die ganzen Zahlen (a,b) - (c,d) definiert als: (a+d,b+c). Die zugehörige Lambda-Funktion ist folgende:
   $\{SUB\} \equiv (\lambda ab.(\lambda z.z\ ((a\ T)\ S\ (b\ F))\ ((a\ F)\ S\ (b\ T))))$
   Wobei T das erste Element und F das zweite Element eines Tuples ausgibt und S die Nachfolgerfunktion ist.
   Die Multiplikation für ganze Zahlen (a,b) * (c,d) ist definiert als: (a*c + b*d,a*d + b*c). Die zugehörige Lambda-Funktion ist folgende:
   $\{MUL\} \equiv$
   $(\lambda ab.(\lambda z.z\ (+\ (*\ (a\ T)\ (b\ T))\ (*\ (a\ F)\ (b\ F)))\ (+\ (*\ (a\ T)\ (b\ F))\ (*\ (a\ F)\ (b\ T))))))$
   Wobei T F die Definition von oben beibehalten und + und * die Addition und Multiplikation für Natürliche Zahlen sind.

6. **Aufgabe** (3 Punkte)

   Zuerst die Idee dahinter, damit der Ausdruck klar wird:
   Wir gucken, ob im Tupel (a,b) a > b ist. Ist es das, macht es mehr Sinn unsere Zahl so zu vereinfachen, dass an zweiter Stelle eine 0 steht und an erster a-b. Tritt das nicht ein, ist also a <= b, setzen wir die erste Stelle gleich 0 und die zweite gleich b - a. In Haskell würde das so aussehen:
   ```
   simp (a,b) | (a > b) = (a-b,0)
              | otherwise = (0, b - a)
   ```
   Als Lambda-Funktion haben wir dann:
   $\{SIMP\} \equiv$
   $\lambda a.(\lambda z.z((>\ (a\ T)\ (a\ F))\ (-\ (a\ T)\ (a\ F))\ 0)\ ((<\ (a\ T)\ (a\ F))\ (-\ (a\ F)\ (a\ T))\ 0)))$
   Erneut sind T und F unsere Tupel Funktionen. - ist die Subtraktion für Natürliche Zahlen und sowohl > als auch < wurden auf diesem Blatt auch bereits definiert.

7. **Aufgabe** (4 Punkte)

   Die Idee hinter > für ganze Zahlen: Wenn wir die Zahlen vereinfachen (also irgendwo muss eine 0 stehen) und dann gucken, ob aus den Tupeln (a,b) (c,d) a == c ist und wir danach checken, ob a == 0 ist, ist c automatisch 0. Das heißt wir müssen nur gucken, ob b > d ist und das Ergebnis ausgeben.
   Bleiben wir bei dem Fall, dass a == c ist: Tritt auf, dass a != 0 (und somit auch gleich c != 0) ist, müssen b und d == 0 seinen, da die Zahl ja vereinfacht wurde. Also müssen wir hier nur a > c prüfen und das Ergebnis ausgeben.
   Ist von Anfang an a != c, heißt das auch wieder automatisch, dass b == d == 0 ist. Also müssen wir auch nur hier das Ergebnis von a > c ausgeben. Also lautet die Lambda-Funktion:
   $> \equiv$
   $((\lambda xy.(\lambda ab.(E\ (a\ T)\ (b\ T))\ ((Z\ (a\ T))\ (>\ (a\ F)\ (b\ F))\ (>\ (a\ T)\ (b\ T))))\ (\{SIMP\}\ x)\ (\{SIMP\}\ y))$
   Die Idee hinter $\neq$ ist einfacher: Wenn wir die Zahlen zuerst vereinfachen und dann prüfen, ob a == c und b == d ist, ist das negierte verundete Ergebnis davon unser Output. Unsere Funktion ist also:
   $\neq\ \equiv ((\lambda xy.(\lambda ab.\wedge\ (E\ (a\ T)\ (b\ T))\ (E\ (a\ F)\ (b\ F))))\ (\{SIMP\}\ x)\ (\{SIMP\}\ y))$