Softwareparadigmen SS 2017, Übungsblatt 2

Abgabe: 10. Mai 2016, bis 16:00 Uhr vor dem Sekretariat IST, Infeldgasse 16b, 2. OG

Allgemeine Hinweise:

- EXP-Ausdrücke werden zur besseren Lesbarkeit in `Festbreitenschrift` statt unterstrichen geschrieben.
- Die Beweise müssen in der Form wie sie in der Vorlesung präsentiert wurden durchgeführt werden (vollständige bzw. strukturelle Induktion).
- Beispiele mit konkreten Werten stellen alleine keinen allgemeinen Beweis dar und werden zu 0 Punkten auf das jeweilige Beispiel führen.
- Wenn Sie Annahmen treffen, müssen diese kurz beschrieben werden
- Achten Sie darauf, welche Funktionen und Prädikate verwendet werden dürfen. Sollten Sie Hilfsfunktionen schreiben bzw. Lemmata verwenden, muss auch deren Korrektheit gezeigt werden!
- Bei den Scala-Beispielen sind die Funktionen, über die Induktion betrieben wird, "von Hand", ohne die Zuhilfename von Funktionen aus der Standard-Library, zu implementieren. Sonst würde der Induktionsbeweis ja nicht funktionieren.

Datentypen:

- $\mathbb{N} = \text{Ziffern} \cup \{\texttt{plus}, \texttt{minus}, \texttt{mult}, \texttt{lt?}, \texttt{gt?}, \texttt{eq?}\}$
- $\mathbb{L} = \{\texttt{build}, \texttt{nil}, \texttt{first}, \texttt{rest}, \texttt{empty?}, \texttt{atom?}\}$
- $\mathbb{B} = \{\texttt{T}, \texttt{F}, \texttt{and}, \texttt{or}, \texttt{not}, \texttt{isTrue?}, \texttt{isFalse?}\}$

In der Zielmenge der Interpretationsfunktion über den Datentypen dürfen normale mathematische Notation und Listenschreibweise benutzt werden, soweit korrekt und eindeutig.

Beispiel 1 (1.5 P.)

Steigende Faktorielle[1] sind definiert als:

$$m^{\overline{n}} = \underbrace{m(m+1)(m+2)\cdots(m+n-1)}_{n \text{ Faktoren}},$$

wobei $m^{\overline{0}} = 1$. Schreiben sie eine EXP-Funktion `risingFactorial` über $\mathbb{N}$ mit Parametern `m` und `n`, die $m^{\overline{n}}$ für $m = \omega(\texttt{m})$ und $n = \omega(\texttt{n})$ berechnet, und beweisen Sie ihre Korrektheit unter Verwendung der Interpretationsfunktion.

[1] https://en.wikipedia.org/wiki/Falling_and_rising_factorials

Beispiel 2 (2.0 P.)

Schreiben Sie eine EXP-Funktion `parity` über $\mathbb{L} + \mathbb{B}$. Diese soll, gegeben eine nicht-verschachtelte Liste von Booleans `xs`, die Parität[2] der Liste berechnen, also `T`, wenn die eine ungerade Anzahl der Elemente `T` ist, sonst `F` (0, die Länge der leeren Liste, zählt als gerade). Zeigen Sie die Korrektheit der Implementierung unter Verwendung der Interpretationsfunktion. (Eine selbst definierte `xor`-Funktion darf verwendet werden, dies muss dann aber auch korrekt argumentiert werden.)

Beispiel 3 (2.5 P.)

Schreiben sie eine EXP-Funktion `sumProd` über $\mathbb{L} + \mathbb{N}$, welche das Skalarprodukt zwischen zwei Listen `xs` und `ys`, plus einer Konstanten `z` berechnet (also die Summe der Produkte der korrenspondierenden Elemente, plus den dritten Parameter für leere Listen). Beispiele:

$$\begin{aligned}
&I(\delta, \omega, \texttt{sumProd(nil, nil, z)}) = \omega(\texttt{z}),\\
&I(\delta, \omega, \texttt{sumProd(build(a, build(b, nil)),}\\
&\qquad\qquad \texttt{build(c, build(d, nil)), z)}) = \omega(\texttt{z}) + \omega(\texttt{a}) \cdot \omega(\texttt{c}) + \omega(\texttt{b}) \cdot \omega(\texttt{d}).
\end{aligned}$$

Sie können davon ausgehen, dass `xs` und `ys` die gleiche Länge haben; für ungleich lange Listen ist das Verhalten undefiniert. Zeigen Sie die Korrektheit der Implementierung unter Verwendung der Interpretationsfunktion (Hinweis: die Induktion sollte über `xs` und `ys` gleichzeitig geführt werden).

Beispiel 4 (1.5 P.)

Schreiben sie eine Scala-Funktion `drop: (Int, List[Int]) => List[Int]`, sodass `drop(n, xs)` die ersten `n` Elemente von `xs` entfernt und den Rest (bzw. eine leere Liste, falls `n` $\geq$ `xs.length`) zurückgibt. Zeigen sie mit struktureller Induktion für die Funktion für alle `xs: List[Int]` und `m`, `n` ≥ 0 die Invariante

```
drop(m, drop(n, xs)) = drop(m + n, xs).
```

Beispiel 5 (2.5 P.)

Gehen Sie von folgender Datenstruktur für Bäume aus:

```
sealed trait Tree
final case class Branch(left: Tree, right: Tree) extends Tree
final case class Leaf(value: Int) extends Tree
```

Schreiben sie zuerst Scala-Funktionen `depth: Tree => Int` und `width: Tree => Int`, die die Tiefe und Breite von Bäumen berechnen. Die Tiefe ist dabei definiert als die maximale Schachtelung aller Teilbäume (`0` für ein einzelnes `Leaf`), und die Breite als Anzahl der `Leaf`s des Baumes (`1` für ein einzelnes `Leaf`). Beispiel: für

```
val t = Branch(Leaf(0), Branch(Leaf(0), Leaf(0)))
```

ist `width(t) == 3` (3 `Leaf`s) und `depth(t) == 2` (maximale Schachtelung: 2 `Branch`es).

Zeigen Sie mittels struktureller Induktion für alle Bäume `t`:

```
width(t) <= pow(2, depth(t))
```

Dabei ist `pow: (Int, Int) => Int` die Exponentialfunktion für Integers[3]. Sie dürfen weiters auf die Funktion `max` aus `scala.math` oder die `max`-Methode der Klasse `Int` zurückgreifen, und diese sowie `pow` als korrekt betrachten.

Tip: $2^a + 2^b \leq 2^{\max(a,b)} + 2^{\max(a,b)}$.

[2] `https://en.wikipedia.org/wiki/Parity_function`

[3] Nachdem diese Funktion in der Standard-Library so nicht vorhanden ist, können Sie sich für Experimente zB. mit folgender Definition behelfen:

```
def pow(m: Int, n: Int) = List.fill(n)(m).product
```

Beispiel 6 (2.5 P.)

Eine Variante von Beispiel 3: schreiben Sie eine Scala-Funktion `sumProd: (List[Int], List[Int], Int) => Int`, die das Skalarprodukt von zwei Listen plus Konstante wie oben beschrieben berechnet. Zeigen Sie mittels struktureller Induktion, dass für alle (gleich langen) `xs, ys: List[Int]` und `z: Int` gilt:

```
sumProd(xs, ys, z) == sum(zipMul(xs, ys), z),
```

wobei

```
def sum(l: List[Int], z: Int): Int = l match {
  case Nil => z
  case x::xs => sum(xs, x + z)
}

zipMul(l1: List[Int], l2: List[Int]): List[Int] = (l1, l2) match {
  (Nil, Nil) => Nil
  (x::xs, y::ys) => (x * y)::zipMul(xs, ys)
}
```

Dabei dürfen `sum` und `zipMul` natürlich nicht in Ihrer Implementierung vorkommen. “Nicht sinnvolle” Fälle sind wieder undefiniert (Sie brauchen keine Exceptions zu werfen oder zu behandeln, sondern können den im Fehlerfall wahrscheinlich auftretenden `MatchError` ignorieren). Wie oben ist es wieder günstig, die Induktion über `xs` und `ys` gleichzeitig zu betreiben (und als Tip: der Beweis ist kürzer, wenn die Funktion tail-rekursiv[4] ist).

Viel Erfolg!

[4] `https://en.wikipedia.org/wiki/Tail_call`