Softwareparadigmen SS 2018, Übungsblatt 2

Abgabe: 23. Mai 2018, bis 16:00 Uhr in Ihrem Gruppen-SVN des Institutes.
Ordnerstruktur: <Repository-URL>/sol2_xx.pdf → xx = Gruppennummer
Es sind lediglich Abgaben im PDF Format gültig. Bitte geben Sie nur **ein** Dokument mit dem Namen "sol2_xx.pdf" ab, das alle Lösungen für das zweite Aufgabenblatt beinhaltet. Die Übung findet in den bereits beim ersten Aufgabenblatt registrierten 3er-Gruppen statt. Es wird empfohlen, das in template.txt verwendete Format einzuhalten.

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!
- Die in dieser Angabe definierten Funktionen dürfen Sie als korrekt betrachten.
- Bei der Definition von Scala-Funktionen dürfen keine Funktionen aus der Standard-Library verwendet werden.
- Bei Behauptungen, bei denen Listen mit `append(as, bs)` verkettet werden, brauchen Sie die Induktion nur über `as` führen, da Sie dann davon ausgehen können, dass die Induktion über `bs` ebenfalls korrekt ist.

Datentypen:

- $\mathbb{N}$ = Natürliche Zahlen inkl. $0 \cup \{\texttt{plus}, \texttt{minus}, \texttt{mult}, \texttt{lt?}, \texttt{gt?}, \texttt{eq?}\}$
- $\mathbb{L} = \{\texttt{build}, \texttt{nil}, \texttt{first}, \texttt{rest}, \texttt{empty?}, \texttt{atom?}\}$
- $\mathbb{B} = \{\texttt{T}, \texttt{F}, \texttt{and}, \texttt{or}, \texttt{not}, \texttt{isTrue?}, \texttt{isFalse?}\}$

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

Beispiel 1 (2.5 P.)

Schreiben Sie die EXP-Funktionen `div(a, b)` und `mod(a, b)` über $\mathbb{N}^0$, die eine Ganzzahldivision mit Rest durchführen. Die Funktionen akzeptieren als Parameter jeweils den Dividenden `a` und den Divisor `b`. `div` berechnet das Ergebnis der Division, `mod` den Rest. Sie können annehmen, dass der Divisor nicht 0 ist.
Beweisen Sie die Korrektheit beider Funktionen mittels Induktion über die Interpretationsfunktion. Sie können die Operationen / und % über den ganzen Zahlen und die damit verbundenen Rechenregeln als korrekt und gegeben nehmen.

Beispiel 2 (2.0 P.)

Die x86-Instruktion $POPCNT$ berechnet den Population Count einer Ganzzahl. Dies ist die Anzahl an Einsen, die in der Binärdarstellung dieser Zahl vorkommt. So ist beispielsweise $POPCNT(13) = POPCNT(0b1101) = 3$. Schreiben Sie die EXP-Funktion `popcnt(i)` über $\mathbb{N}^0$, die den Population Count des Parameterwertes `i` berechnet. Sie dürfen die vorhin geschriebenen Funktionen `div` und `mod` verwenden und als korrekt betrachten. Zeigen Sie die Korrektheit Ihrer Implementierung, indem Sie `popcnt(i)` $= POPCNT(i)$ induktiv beweisen. Überlegen Sie sich zuerst, wie Sie den Induktionsschritt ansetzen.
Folgende Invarianten dürfen Sie als gegeben nehmen:

$$POPCNT(0) = 0$$
$$POPCNT(2 \cdot i) = POPCNT(i)$$
$$POPCNT(2 \cdot i + 1) = POPCNT(i) + 1$$

Beispiel 3 (2.0 P.)

Gegeben sei folgendes EXP-Programm über dem Datentyp der Listen, welches die maximale Verschachtelungstiefe einer Liste berechnet:

```
δmax =if eq?(x1, nil) then
      x2
      else if eq?(x2, nil) then
      x1
      else
      build(max(rest(x1), rest(x2)), nil)
δdepth =if atom?(first(x1)) then
      depth(rest(x1))
      else if eq?(x1, nil) then
      nil
      else
      max(build(depth(first(x1)), nil), depth(rest(x1)))
```

Das Programm gibt eine verschachtelte Liste zurück, deren Verschachtelungstiefe einer Zahl entspricht. So wäre die Liste `[[[]]]` mit der Zahl 3 gleichzusetzen.
Die Korrektheit von `depth(x1)` wurde bereits im Übungsskriptum in Beispiel 2.17 auf Seite 45 bewiesen. Allerdings liefert die Funktion `max(x1, x2)` nicht immer das gewünschte Ergebnis. Dadurch funktioniert das gesamte Programm nicht wie gewünscht. Ihre Aufgaben:

1. Beschreiben Sie, unter welchen Umständen die Funktion `max(x1, x2)` im Kontext von `depth(x1)` einen falschen Output liefert. Diskutieren Sie dabei Schritt für Schritt die einzelnen Statements und die an die beiden Funktionen übergebenen Argumente und beschreiben Sie Ihre Beobachtungen. Tipp: Verwenden Sie die Definitionen des Datentyps der Listen.

2. Zeigen Sie unter Verwendung der Interpretationsfunktion in Kombination mit einem konkreten Beispiel, dass das Programm nicht den gewünschten Output liefert.

3. Korrigieren Sie den Fehler, sodass das Programm die maximale Verschachtelungs-Tiefe einer Liste berechnet.

4. Zeigen Sie unter Verwendung der Interpretationsfunktion in Kombination mit einem konkreten Beispiel, dass Ihr korrigiertes Programm jetzt den gewünschten Output liefert.

Beispiel 4 (1.5 P.)

Gegeben ist die Scala-Funktion

```
def append(as: List[Int], bs: List[Int]): List[Int] = as match {
  case Nil => bs
  case a::as => a::append(as, bs)
}
```

die zwei Listen `as` und `bs` konkateniert. Zum Beispiel gibt `append(List(1, 2), List(3, 4, 5))` die Liste `List(1, 2, 3, 4, 5)` zurück.
Schreiben Sie eine Scala-Funktion `count: (List[Int]) => Int`, sodass `count(needle, haystack)` die Anzahl der Elemente gleich `needle` in der Liste `haystack` zurückgibt. So soll `count(1, List(1, 3, 2, 1, 1, 3))` beispielsweise `3` zurückgeben. Zeigen sie mittels struktureller Induktion, dass

```
count(append(as, bs), e) == count(as, e) + count(bs, e)
```

gilt.

Beispiel 5 (2.0 P.)

Gegeben sind folgende Scala-Funktionen:

```
def sumList(ls: List[Int]): Int = ls match {
  case l::ls => l + sumList(ls)
  case Nil => 0
}

def reverse(ls: List[Int]): List[Int] = ls match {
  case l::ls => append(reverse(ls), List(l))
  case Nil => List()
}
```

Die Funktion `sumList(ls)` berechnet die Summe ihrer Elemente und `reverse(ls)` liefert eine Liste mit den Elementen der gegebenen Liste in umgekehrter Reihenfolge. So ist `sumList(List(1, 2, 3)) == 6` und `reverse(List(1, 2, 3)) == List(3, 2, 1)`. Zeigen Sie induktiv, dass

```
sumList(as) == sumList(reverse(as))
```

gilt.

Beispiel 6 (2.5 P.)

Gegeben sind folgende Scala-Funktionen:

```
def odd(ls: List[Int]): Boolean = ls match {
  case Nil => false
  case _::ls => even(ls)
}

def even(ls: List[Int]): Boolean = ls match {
  case Nil => true
  case _::ls => odd(ls)
}
```

`even(ls)` und `odd(ls)` geben zurück, ob die Anzahl der Elemente in einer Liste gerade bzw. ungerade ist. Beispielsweise sind `even(List(1, 2))` und `odd(List(3, 4, 5))` wahr, aber `even(List(3, 4, 5))` und `odd(List(1, 2))` falsch.
Zeigen Sie durch strukturelle Induktion die Gültigkeit der Behauptung

```
even(append(as, bs)) == (even(as) == even(bs))
```

Viel Erfolg!