Universität Bremen

Praktische Informatik 3 WS 14/15

Christoph Lüth
Sandor Herms
Daniel Müller
Jan Radtke
Henrik Reichmann
Sören Schulze
Felix Thielke

8. Übungsblatt

Ausgabe: 09.12.14
Abgabe: 19.12.14

8.1 *Meine kleine Uhr* *5 Punkte*

In dieser Aufgabe implementieren wir eine ganz einfache Uhr, die tickt und an der wir die Zeit ablesen können. Unsere Uhr ist natürlich keine gewöhnliche Uhr, sie ist ein abstrakter Datentyp mit folgenden Operationen:

```
data Clock
clock :: Clock
tick  :: Clock→ Clock
secs  :: Clock→ Int
mins  :: Clock→ Int
hrs   :: Clock→ Int
```

Hierbei erzeugt Clock eine neue Uhr mit der Zeit 0, und tick erhöht die Zeit im eine Hunderstel Sekunde. Die Operationen sollen folgende Eigenschaften erfüllen:

1. Nach 100 Ticks soll sich die Anzahl der Sekunden um eine erhöht haben, oder wieder auf null gesprungen sein;
2. nach entsprechend $60 * 100$ Ticks die Anzahl der Minuten, und nach $60 * 60 * 100$ Ticks die Anzahl der Stunden;
3. die Sekunden und Minuten sollen immer zwischen 0 und 59, und die Stunden zwischen 0 und 23 sein.

Implementieren Sie den abstrakten Datentypen Clock, formulieren Sie die obigen Eigenschaften als Axiome und testen diese mit *QuickCheck*.

Hinweis: Um eine Uhr mit einer zufälligen Zeit zu erzeugen, nutzen Sie folgendes Stück Code (wobei Clock der Konstruktur des Datentypen ist):

```
instance Arbitrary Clock where
  arbitrary = do
    t ← choose (0, 24*60*60*100)
    return $ Clock t
```

8.2 *Eine Menge Zahlen* *15 Punkte*

In dieser Aufgabe soll der abstrakte Datentyp von *endlichen Mengen von natürlichen Zahlen* spezifiziert und implementiert werden.

Auf diesem abstrakten Typen soll es folgende Operationen geben:

- die leere Menge (empty);
- das Erzeugen einer Menge mit genau einer Zahl (singleton), wobei die Zahl zwischen einschließlich 0 und einer Konstanten maxSetInt liegen soll;
- das Löschen einer Zahl aus einer Menge (delete);
- der Test, ob eine Zahl in einer Menge enthalten ist (isElem);
- die Vereinigung von zwei Mengen (union);

- die Schnittmenge von zwei Mengen (intersect);
- das Komplement einer Menge (complement);
- die Teilmengenbeziehung (subset) zwischen zwei Mengen;
- die Gleichheit zweier Mengen (==) zwischen zwei Mengen.

Diese Operationen sollen unter anderem folgende Eigenschaften erfüllen:

(1) Die leere Menge enthält keine Elemente.

(2) Die einelementige Menge aus einer Zahl enthält diese Zahl.

(3) Wenn eine Zahl in einer einer einelementigen Menge aus einer Zahl enthalten ist, dann sind die beiden Zahlen gleich.

(4) Eine Zahl ist nicht in der Menge enthalten, aus der sie gelöscht wurde.

(5) Ein Zahl ist in der Menge, aus der eine andere Zahl gelöscht wurde, genau dann enthalten, wenn sie in der Menge ohne die gelösche Zahl enthalten ist.

(6) Eine Menge ist genau dann eine Teilmenge von einer anderen Menge, wenn alle Zahlen, die in der ersten Menge enthalten sind, auch in der zweiten Menge enthalten sind.

(7) Wenn eine Zahl in einer Menge enthalten ist, dann nicht in ihrem Komplement, und umgekehrt.

(8) Eine Zahl ist in der Vereinigung zweier Mengen enthalten, wenn sie in mindestens einer der beiden zu vereinigenden Mengen enthalten ist.

(9) Eine Zahl ist in der Schnittmenge zweier Mengen enthalten, wenn sie in beiden Mengen enthalten ist.

Geben Sie zuerst eine Signatur für die Operationen an, und formalisieren Sie die oben aufgeführten Eigenschaften als Axiome. Geben Sie mindest noch fünf weitere sinnvolle und nicht-triviale Axiome an. Formulieren Sie alle Axiome als Eigenschaften für quickCheck.

(8 *Punkte*)

Implementieren Sie jetzt den ADT, und testen Sie Ihre Implementation mit den Eigenschaften in quickCheck. Hierbei sollen die Mengen von Zahlen durch Bitvektoren (in unserem Falle vom Typ Integer) kodiert werden, deren i-tes Bit die Zahl i repräsentiert. Die Mengenoperationen können dann effizient durch binäre Operationen, wie sie von dem Haskell-Modul Data.Bits bereitgestellt werden, implementiert werden (Vereinigung durch binäre Disjunktion (.|.), Schnitt durch binäre Konjunktion (.&.), etc.). Ihre Implementation sollte den Typ Integer verkapseln, und alle Operationen direkt auf Integer ausführen; die Mengen sollen *nicht* als Liste oder Baum repräsentiert werden.

(7 *Punkte*)

Hinweis: Zum Testen können Sie folgendes Stück Code benutzen, welches zufällige Mengen von bis zu 100 Elementen erzeugt:

```
instance Arbitrary BitSet where
  arbitrary = do
    size ← choose (0, 100 :: Int)
    xs ← vectorOf size (choose (0, maxSetInt))
    return $ foldl (λ s i → union s (singleton i)) empty xs
```

? *Verständnisfragen*

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?
2. Warum „finden Tests Fehler“, aber „zeigen Beweise Korrektheit“, wie in der Vorlesung behauptet? Stimmt das immer?
3. Müssen Axiome immer ausführbar sein? Welche Axiome wären nicht ausführbar?