Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
L. Noschinski, L. Hupel, Dr. J. Blanchette, M. Eberl

WS 2014/15
21.11.2014
Abgabe: 28.11.2014

Einführung in die Informatik 2
7. Übung

Aufgabe G7.1 Brüche sind auch Zahlen

Im letzten Blatt haben wir Haskells vordefinierten Typ für Brüche `Rational` verwendet. Nun werden wir unsere eigene Variante implementieren.

Definieren Sie dazu einen Typ `Fraction` mit einem Konstruktor `Over :: Integer -> Integer -> Fraction`.

1. Damit sich die Brüche nicht benachteiligt fühlen, sollen die üblichen mathematischen Operatoren auch mit Brüchen funktionieren. Definieren Sie daher `Fraction` als eine Instanz der Typklasse `Num`.

```
class Num a where
  (+), (-), (*)       :: a -> a -> a
  negate              :: a -> a
  fromInteger         :: Integer -> a
  -- The functions 'abs' and 'signum' should
  -- satisfy the law:
  --
  -- > abs x * signum x == x
  abs                 :: a -> a
  signum              :: a -> a
```

2. Bei der Definition eines Datentypes kann Haskell diesen automatisch zu einer Instanz von `Eq` machen (mittels `deriving Eq`). Ist diese automatisch abgeleitete Instanz hier sinnvoll? Wenn nein, wie sollte sie stattdessen aussehen?
3. Schreiben Sie QuickCheck-Tests, die das in der Definition angegebene Gesetz sowie weitere wünschenswerte Eigenschaften prüfen, z.B. dass sich `(-)` konsistent zu `negate` und `(+)` verhält.

Weiterführende Hinweise:

- Die Typklasse `Num` enthält keinen Divisionsoperator, dieser ist in der Typklasse `Fractional` implementiert. In einem echten Programm würde man `Fraction` daher zu darüber hinaus zu einer Instanz von `Fractional` machen.
- Die Funktion `fromInteger` wird von Haskell benutzt, um ein Integer-Literal in den gesuchten Typ umzuwandeln. Der Ausdruck `3 :: Fraction` ist also der Bruch `(fromInteger 3) :: Fraction`. Die Funktion `fromRational` aus der Typklasse `Fractional` macht das gleiche für Dezimalliterale (z.B. `3.14`).

Aufgabe G7.2 Vereinfachen Sie die Definition!

Geben Sie alternative Definitionen mit möglichst kleiner Anzahl von Parametern (links vom Gleichheitszeichen) für die folgenden Funktionen an. Schreiben Sie dabei alle bestehenden λ-Ausdrücke um und führen Sie keine neuen λ-Ausdrücke ein (λ = lambda = \).

```
f1 xs = map (\x -> x + 1) xs
f2 xs = map (\x -> 2 * x) (map (\x -> x + 1) xs)
f3 xs = filter (\x -> x > 1) (map (\x -> x + 1) xs)
f4 f g x = f (g x)
f5 f g x y = f (g x y)
f6 f g x y z = f (g x y z)
f7 f g h x = g (h (f x))
```

Aufgabe G7.3 Figurativ

Wir wollen einfache geometrische Figuren in Haskell als Datentyp modellieren.

1. Definieren Sie einen Datentyp `Shape`. Dieser soll zwei Konstruktoren haben: `Circle` mit einem `Integer` als Radius und `Rectangle` mit zwei `Integer`n für Länge und Breite.

   Hinweis: Zum Ausprobieren auf der Konsole bietet es sich an, für den Datentyp eine `Show`- und eine `Eq`-Instanz zu definieren. Automatisch geht das mittels

   ```
   data Shape = ... deriving (Eq, Show)
   ```

2. Schreiben Sie eine Funktion `isValid :: Shape -> Bool`, die genau dann `True` zurückgibt, wenn die Form keine negative Länge, Breite oder Radius hat.
3. Implementieren Sie eine Funktion, die eine Figur bezüglich ihres Umfangs um einen Faktor r skaliert.

   Beispiel:

   ```
   scale 2 (Rectangle 1 3) = Rectangle 2 6
   scale 5 (Circle 4) = Circle 20
   ```

4. Fügen Sie dem Datentyp einen weiteren Konstruktor für Dreiecke hinzu. Durch wie viele `Integer`s ist ein Dreieck vollständig bestimmt (ohne Berücksichtigung von Position oder Drehung im Koordinatensystem)?

   Passen Sie `isValid` und und `scale` so an, dass sie auch mit Dreiecken zurecht kommen. Welche zusätzliche Eigenschaft von Dreiecken sollte in `isValid` geprüft werden?

Aufgabe H7.1 Darf ich bitten?

Eine wichtige Anwendung von Typklassen ist die Spezifikation abstrakter Datenstrukturen. So kann z.B. ein Algorithmus auf beliebigen Mengen arbeiten, unabhängig von der internen Repräsentation. Für diese Aufgabe sollen Sie *bit sets* implementieren, d.h. eine Datenstruktur, die nichtnegative Integer auf `True` oder `False` abbildet.

Hierfür sei die folgende Typklasse gegeben:

```
class BitSet a where
  empty    :: a                  -- leeres bit set
  getBit   :: Int -> a -> Bool   -- i-tes Bit gesetzt?
  setBit   :: Int -> a -> a      -- setze i-tes Bit
  unsetBit :: Int -> a -> a      -- lösche i-tes Bit
```

Formal ausgedrückt sollen für alle *bit sets* `s` und alle nichtnegativen Integer `i` und `j` die folgenden Gesetze gelten:

```
getBit i empty           == False
getBit i (setBit j s)    == if i == j then True  else getBit i s
getBit i (unsetBit j s) == if i == j then False else getBit i s
```

Aufgaben:

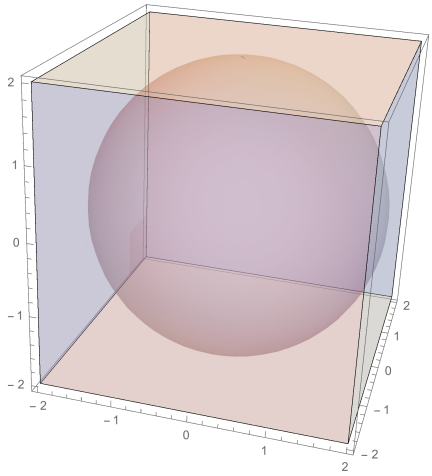
1. Schreiben Sie `BitSet`-Instanzen für die Datentypen `BoolListBitSet` und `IntegerBitSet` (siehe Vorlage). Die Implementation ist beliebig, muss aber die obigen Gesetze erfüllen.

   Hinweis: Funktionen aus `Data.Bits` können hilfreich sein, z.B. `.&.`, `.|.` und `complement`.

2. *(optional)* Implementieren Sie die Funktion `fromList` mithilfe der Funktionen `empty`, `setBit`, `unsetBit`. Diese Funktion soll eine Liste von `Int` in ein *bit set* umwandeln, in dem genau die angegebenen Bits gesetzt sind.

Hinweis: Um Ihre Funktionen zu testen, müssen Sie Haskell sagen, welcher konkrete Typ verwendet werden soll. Wenn Sie einen Fehler wegen *type variable ambiguity* bekommen, müssen Sie einen passenden Teilausdruck entsprechend annotieren, z.B. `empty :: BoolListBitSet` oder `getBit 2 (fromList [1,2,3] :: IntegerBitSet)` .

Aufgabe H7.2 Figurativ II



In dieser Aufgabe definieren Sie einen Datentypen für dreidimensionale geometrische Körper. Wir betrachten Quader und Kugeln in einem ganzzahligen Koordinatensystem. Punkte werden dargestellt als `type Point3 = (Integer, Integer, Integer)`.

1. Definieren Sie einen Datentyp `Shape3` mit je einem Konstruktor für `Sphere` und `Cuboid`. Die genaue Repräsentation ist Ihnen überlassen, allerdings müssen die Konstruktoren genau so heißen, damit Ihre Abgabe getestet werden kann.

2. Damit Benutzer von Ihrer konkreten Repräsentation unabhängig sind, stellen Sie die folgenden beiden Funktionen bereit:

```
makeCuboid :: Point3 -> Point3 -> Shape3
makeSphere :: Point3 -> Integer -> Shape3
```

   Das Ergebnis von `makeCuboid` p_1 p_2 soll der Quader mit den zwei gegenüberliegenden Punkten p_1 und p_2 sein (orthogonal zum Koordinatensystem). `makeSphere` p r hingegen soll der Kugel mit dem Mittelpunkt p und dem Radius r entsprechen.

3. Schreiben Sie eine Funktion, die die sogenannte *bounding box* eines Körpers ermittelt. Dies ist definiert als der kleinste Quader mit ganzzahligen Koordinaten, der den Körper umschließt. Der Typ der gesuchten Funktion ist `Shape3 -> (Point3, Point3)`, wobei der Rückgabewert ein Paar von beliebigen gegenüberliegenden Eckpunkten sein soll.

   Beispiel: In der Abbildung sehen Sie die Kugel mit $r = 2$ und dem Mittelpunkt $(0, 0, 0)$ sowie zugehöriger *bounding box*. Gültige Rückgabewerte sind z.B. $\big((-2, -2, -2), (2, 2, 2)\big)$ und $\big((2, -2, -2), (-2, 2, 2)\big)$.

4. *(optional)* Definieren Sie eine Funktion `overlapping :: Shape3 -> Shape3 -> Bool`, die testet, ob sich die *bounding boxes* zweier Körper überlappen oder berühren.

   Hinweis: Für diese Aufgabe ist es sinnvoll, wenn `boundingBox` nicht ein beliebiges Paar von Eckpunkten zurückliefert, sondern das Paar von minimalem und maximalem Eckpunkt.

Aufgabe H7.3 Liniert

Wir stellen eine (horizontale) Strecke mit dem Typ `Line` dar.

```
data Line a = Line (a, a) a deriving Show
```

Dabei beschreibt `Line (x,y) l` die Strecke zwischen (x, y) und $(x + l, y)$. Sie dürfen annehmen, dass `l` immer positiv ist. Schreiben Sie eine Funktion, die überlappende Strecken zusammenfasst:

```
longestLines :: (Num a, Ord a) => [Line a] -> [Line a]
```

Das heißt: Zeichnet man die Strecken auf ein Blatt Papier, so sollen `xs` und `longestLines xs` das gleiche Bild ergeben. `longestLines xs` soll aber aus so wenigen Strecken wie möglich bestehen. Die Reihenfolge der Strecken ist egal.

Beispiel: Abb. 1 entspricht der Liste

```
xs = [Line (0,2) 2, Line (1,4) 3, Line (4,3) 5,
      Line (8,4) 2, Line (3,4) 3, Line (2,2) 1] :: [Line Integer]
```

und Abb. 2 zeigt die zusammengefassten Strecken.

Also ist `longestLine xs` gleich (einer Umsortierung von):

```
[Line (0,2) 3, Line (4,3) 5, Line (1,4) 5, Line (8,4) 2]
```

Hinweis: Diese Aufgabe lässt sich gut in kleinere Funktionen zerlegen:

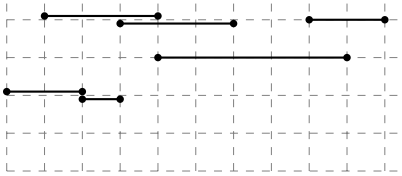


Abbildung 1: Zeichnung von `xs`

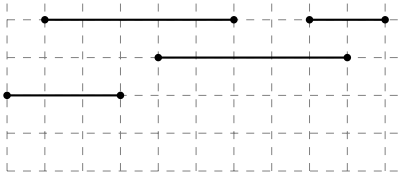


Abbildung 2: Zeichnung von `longestLines xs`

- Strecken auf einer Höhe (beschrieben durch x-Position und Länge) zusammenfassen:
  `merge :: (Num a, Ord a) => [(a,a)] -> [(a,a)]`,
- Gruppieren nach y-Position:
  `groupY :: (Num a, Ord a) => [Line a] -> [(a, [(a,a)])]`
- Die Gruppierung wieder auflösen:
  `ungroupY :: [(a, [(a,a)])] -> [Line a]`.

Diese drei Funktionen lassen sich dann zu der gewünschten Funktion kombinieren.

Aufgabe H7.4 Godzilla gegen Manhattan (Wettbewerbsaufgabe)

Godzilla ist wieder aufgewacht und greift diesmal New York City an. Die Bürger, die nicht rechzeitig die Stadt verlassen konnten, verstecken sich alle in verschieden, über die ganze Stadt verteilten Schutzräumen. Falls ein dieser Schutzräume angegriffen wird, müssen dessen Bewohner zum nächsten Schutzraum flüchten.

In dieser Aufgabe geht es darum, aus einer Liste von Schutzraum-Koordinaten einen Fluchtplan zu berechnen: für jeden Schutzraum ist der jeweils nächste Schutzraum zu bestimmen. Beachten Sie dabei, dass die Relation „x ist der nächste Schutzraum zu y“ im Allgemeinen nicht symmetrisch ist. Gegeben drei Schutzräume mit Koordinaten $(0,0)$, $(2,0)$ und $(3,0)$, ist $(2,0)$ der nächste Schutzraum von $(0,0)$, aber $(0,0)$ ist nicht der nächste von $(2,0)$.

Bei der Berechnung von Distanzen muss beachtet werden, dass die Bürger nicht einfach *wegfliegen* können. Sie können sich nur in Nord-Süd- und Ost-West-Richtung bewegen, da die New Yorker Straßen (mit einigen Ausnahmen, vor allem der Broadway) ein Gitter bilden. Der Abstand zwischen Schutzräumen wird deshalb mit der sogenannten *Manhattan-Metrik* (oder *Manhattan-Distanz*) gemessen:

$$\mathit{distance}((x_1, y_1), (x_2, y_2)) = |x_2 - x_1| + |y_2 - y_1|$$

Gegeben sei das Typsynonym

```
type Point = (Integer, Integer)
```

Ihre Aufgabe besteht darin, die Funktion

```
nearestNeighbors :: [Point] -> [(Point, Point)]
```

zu implementieren, die eine Liste von Schutzräumen als Argument nimmt und eine Liste von Paaren der Form (*Schutzraum*, *nächster Schutzraum*) zurückliefert. Zum Beispiel:

```
nearestNeighbors [(0,0), (2,0), (3,0)] ==
  [((0,0), (2,0)), ((2,0), (3,0)), ((3,0), (2,0))]
nearestNeighbors [(0,0), (2,-10), (3,0)] ==
  [((0,0), (3,0)), ((2,-10), (3,0)), ((3,0), (0,0))]
nearestNeighbors [(0,0), (0,0)] ==
  [((0,0), (0,0)), ((0,0), (0,0))]
  -- oder [((0,0), (0,0))]
  -- oder [((0,0), (0,0)), ((0,0), (0,0)), ((0,0), (0,0)),] ...
nearestNeighbors [(0,0), (1,0), (2,0)] ==
  [((0,0), (1,0)), ((1,0), (0,0)), ((2,0), (1,0))]
  -- oder [((0,0), (1,0)), ((1,0), (0,0)), ((2,0), (1,0)), ((1,0), (2,0))]
nearestNeighbors [] == []
nearestNeighbors [(0,0)]   -- unspezifiziert (darf Exception werfen)
```

Wie an den Beispielen zu erkennen ist, können mehrere Schutzräume an der gleichen Position stehen. Ist dies der Fall, also enthält die Eingabeliste zwei mal `(x,y)`, dann muss in der Ausgabeliste `(x,y)` auf `(x,y)` abgebildet werden, da dies der nächste Schutzraum ist.

Die Reihenfolge der Elemente in der Ausgabeliste darf beliebig sein. Die Ausgabeliste darf für jeden Schutzraum `(x,y)` aus der Eingabe beliebig viele (aber ≥ 1) Paare `((x,y), (x',y'))` in der Ausgabeliste enthalten, jedoch muss jedes dieser `(x',y')` minimalen Abstand zu `(x,y)` haben.

Für den Wettbewerb zählt die Effizienz, vor allem die Skalierbarkeit. Ihre Implementierung sollte mit langen Listen (z.B. 10 000 Schutzräumen) möglichst effizient vorgehen. Lösungen, die sich effizienztechnisch ähnlich verhalten, werden bezüglich ihrer Tokenanzahl verglichen.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die {`-WETT-`}...{`-TTEW-`} Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.