Eine Sache lernt man, indem man sie macht.
Cesare Pavese (1908-1950)
italien. Schriftsteller

Für das Können gibt es nur einen Beweis, das Tun.
Marie von Ebner-Eschenbach (1830-1916)
österr. Schriftstellerin

5. Angabe zu Funktionale Programmierung von Fr, 12.11.2021.

Erstabgabe: Fr, 19.11.2021, 12:00 Uhr

Zweitabgabe: Siehe „Hinweise zu Org. u. Ablauf der Übung" (TUWEL-Kurs)

Themen: *Polymorphie, vor- und selbstdefinierte Typklassen, Überladung, rekursive Funktionen, Funktionen höherer Ordnung, hierarchische Funktionssysteme, Typdeklarationen (Typsynonyme, neue Typen, algebraische Typen), Feldsyntax, Muster*

Stoffumfang: *Kapitel 1 bis Kapitel 11, besonders Kapitel 4, 5, 10 und 11.*

- **Teil A, programmiertechnische Aufgaben:** Besprechung am ersten Übungsgruppentermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- Teil B, Papier- und Bleistiftaufgaben: Entfallen auf Angaben 5 bis 7.

Wichtig

1. Befolgen Sie die Anweisungen aus den 'Lies-mich'-Dateien (s. TUWEL-Kurs) zu den Angaben sorgfältig, um ein reibungsloses Zusammenspiel mit dem Testsystem sicherzustellen. Bei Fragen dazu, stellen Sie diese bitte im TUWEL-Forum zur LVA.
2. Erweitern Sie für die für diese Angabe zu schreibenden Rechenvorschriften die zur Verfügung gestellte Rahmendatei

   `Angabe5.hs`

   und legen Sie sie für die Abgabe auf oberstem Niveau in Ihrem *home*-Verzeichnis ab. Achten Sie darauf, dass "Gruppe" Leserechte für diese Datei hat. Wenn nicht, setzen Sie diese Leserechte mittels `chmod g+r Angabe5.hs`.

   Löschen Sie keinesfalls eine Deklaration aus der Rahmendatei! Auch dann nicht, wenn Sie einige dieser Deklarationen nicht oder nicht vollständig implementieren wollen. Löschen Sie auch nicht die für das Testsystem erforderliche Modul-Anweisung `module Angabe5 where` am Anfang der Rahmendatei.
3. Der Name der Abgabedatei ist für Erst- und Zweitabgabe ident!
4. Benutzen Sie keine selbstdefinierten Module! Wenn Sie (für spätere Angaben) einzelne Rechenvorschriften früherer Lösungen wiederverwenden möchten, kopieren Sie diese bitte in die neue Abgabedatei ein. Eine `import`-Anweisung für selbstdefinierte Module schlägt für die Auswertung durch das Abgabesystem fehl, weil Ihre Modul-Datei, aus der importiert werden soll, vom Testsystem nicht mit abgesammelt wird.
5. Ihre Programmierlösungen werden stets auf der Maschine `g0` mit der dort installierten Version von GHCi überprüft. Stellen Sie deshalb sicher, dass sich Ihre Programme (auch) auf der `g0` unter GHCi so verhalten, wie von Ihnen gewünscht.
6. Überzeugen Sie sich bei jeder Abgabe davon! Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einer anderen Maschine, einer anderen GHCi-Version oder/und einem anderen Werkzeug wie etwa Hugs arbeiten!

A Programmiertechnische Aufgaben (beurteilt, max. 100 Punkte)

Erweitern Sie zur Lösung der programmiertechnischen Aufgaben die Rahmendatei `Angabe5.hs`. Kommentieren Sie die Rechenvorschriften in Ihrem Programm zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Wertvereinbarungen für konstante Werte (z.B. `pi = 3.14 :: Float`). Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typdeklarationen, d.h. geben Sie stets deren syntaktische Signatur (kurz: Signatur), explizit an.

Mengen sind Zusammenfassungen einer beliebigen Anzahl von Dingen. Ein Ding darf dabei höchstens einmal in einer Zusammenfassung enthalten sein. Die zusammengefassten Dinge heißen *Elemente*, ihre Zusammenfassung *Menge*. Dabei gelten keinerlei Anforderungen an die Art der zusammengefassten Elemente. Dürfen Elemente mehrfach in einer Zusammenfassung enthalten sein, spricht man von einer *Multimenge*.

Zur Erinnerung sind hier die einschlägigen Operationen und Relationen auf Mengen wiederholt:

- *Mengenoperationen*
  - Vereinigung: $M \cup M' =_{df} \{m \mid m \in M \vee m \in M'\}$.
  - Durchschnitt: $M \cap M' =_{df} \{m \mid m \in M \wedge m \in M'\}$.
  - Differenz: $M \backslash M' =_{df} \{m \in M \mid m \notin M'\}$.
- *Mengenrelationen*
  - Gleichheit: $M = M' \Longleftrightarrow_{df} m \in M \Leftrightarrow m \in M'$.
  - Ungleichheit: $M \neq M' \Longleftrightarrow_{df} \neg(M = N)$.
  - Teilmenge: $M \subseteq M' \Longleftrightarrow_{df} m \in M \Rightarrow m \in M'$.
  - Obermenge: $M \supseteq M' \Longleftrightarrow_{df} M' \subseteq M$.
  - Element: $m \in M \Longleftrightarrow_{df} \{m\} \subseteq M$.
- *Mengenkonstante*
  - Leere Menge: $\emptyset =_{df} \{\}$.

Für Multimengen sind diese Operationen und Relationen entsprechend festgelegt. Die Vereinigung zweier Multimengen M und M' enthält ein Element so oft, wie es der Summe seiner Vorkommen in M und M' entspricht, ihr Durchschnitt so oft, wie es dem Minimum seiner Vorkommen in M und M' entspricht, ihre Differenz so oft, wie es dem Maximum aus 0 und der Differenz seiner Vorkommen in M und M' entspricht. M und M' sind gleich, wenn M' jedes Element von M genauso oft enthält wie M und umgekehrt; ungleich sonst. M ist Teilmenge von M', wenn M' jedes Element von M mindestens genauso oft enthält; M' ist in diesem Fall Obermenge von M. m ist Element von M, wenn m mindestens einmal in M vorkommt.

In Haskell können wir beliebige Anzahlen von Dingen in Listen zusammenzufassen. Enthalten Listen Elemente höchstens einmal, können wir sie als Modellierung von Mengen auffassen, anderenfalls als Modellierung von Multimengen.

In Listen können jedoch anders als in Mengen nur typgleiche Elemente zusammengefasst werden. Listen erlauben deshalb nur die Modellierung *homogener* Mengen und Multimengen (zusammengefasste Elemente müssen typgleich sein), während Mengen und Multimengen *heterogen* sein können (zusammengefasste Elemente können typverschieden sein).

Mithilfe polymorpher algebraischer Datentypen als Elementtypen von Listen können wir jedoch *pseudoheterogene*, also scheinbar heterogene Mengen und Multimengen als Listen modellieren.

In der Folge werden wir mithilfe von Listen *homogene* und *pseudoheterogene* Mengen und Multimengen modellieren und die entsprechenden Mengenoperationen und -relationen darauf implementieren. Um die Namen der Operationen und Relationen für verschiedene Typen überladen und wiederverwenden zu können, führen wir eine neue Typklasse `Menge_von` ein ('Menge von `a`-Werten'), die die entsprechenden Namen zusammen mit einigen Protoimplementierungen bereit stellt. Die Funktion `anzahl`, die für Instanzen dieser Typklasse zu implementieren ist, ist dafür vorgesehen, die Zahl der Vorkommen eines Elements in einer Menge oder Multimenge zu bestimmen; die (0-stellige) Funktion `leer` dafür, die Darstellung der leeren Menge zu liefern.

```
type Nat0 = Int

class Eq a => Menge_von a where
 leer          :: [] a
 vereinige     :: [] a -> [] a -> [] a
 schneide      :: [] a -> [] a -> [] a
 ziehe_ab      :: [] a -> [] a -> [] a
 ist_teilmenge :: [] a -> [] a -> Bool
 ist_obermenge :: [] a -> [] a -> Bool
 ist_element   :: a -> [] a -> Bool
 ist_leer      :: [] a -> Bool
 sind_gleich   :: [] a -> [] a -> Bool
 anzahl        :: a -> [] a -> Nat0

 -- Protoimplementierungen
 leer = []
 vereinige xs ys     = xs ++ ys
 ist_teilmenge xs ys = ist_obermenge ys xs
 ist_obermenge xs ys = ist_teilmenge ys xs
 ist_element x xs    = anzahl x xs >= 1
 ist_leer xs         = xs == leer
 sind_gleich xs ys   = ist_teilmenge xs ys && ist_teilmenge ys xs
```

Bemerkung zur Syntax: Das eckige Klammerpaar `[]` ist ein sog. Typkonstruktor, der Typkonstruktor für Listentypen, kurz, der *Listenkonstruktor*. Der Listenkonstruktor bildet konkrete Typen (wie z.B. `Int`, `Bool`, `(Char,Int -> Int)`,,...) oder Typvariablen (wie z.B. `a`, `b`, `c`, `hugo`, `fridolin`,...) auf den Listentyp vom entsprechenden Elementtyp ab. Unsere bisher verwendeten Schreibweisen wie `[Int]`, `[(Char,Int -> Int)]`, `[a]`, `[hugo]`,... sind syntaktischer Zucker für die zugrundeliegenden Standardschreibweisen `[] Int`, `[] (Char, Int -> Int)`, `[] a`, `[] hugo`, etc., die die Abbildungseigenschaft des Typkonstruktors `[]` zeigen. Die unverzuckerte Typsignatur `vereinige :: [] a -> [] a -> [] a` entspricht deshalb der üblicherweise verwendeten verzuckerten Typsignatur `vereinige :: [a] -> [a] -> [a]`; beide Schreibweisen sind bedeutungsgleich (entsprechend gilt dies auch für die Signaturen der anderen Funktionen in `Menge_von`).

Bevor wir Mengen implementieren, verschaffen wir uns zunächst zwei neue Typen `Zahlraum_0_10` und `Funktion` mit gewissen (Typklassen-) Eigenschaften:

A.1 Machen Sie den algebraischen Typ `Zahlraum_0_10` zur Instanz der Typklasse `Num`, so dass die Operationen `(+)`, `(-)`, `(*)` der Addition, Subtraktion und Multipliklation auf ganzen Zahlen eingeschränkt auf den Zahlbereich von 0 bis 10 entsprechen. Dabei steht der Wert `N` für 0 und der Wert `F` für den fehleranzeigenden Wert, wenn Operationsergebnisse außerhalb des Zahlraums von 0 bis 10 liegen. Ist einer der Operationsoperanden der Fehlerwert `F`, so ist auch das Operationsergebnis der Wert `F`. Die übrigen Funktionen der Typklasse `Num` sollen in offensichtlicher Weise für `Zahlraum_0_10`-Werte implementiert werden:

```
data Zahlraum_0_10 = N | I | II | III | IV | V | VI
                   | VII | VIII | IX | X | F deriving (Eq,Ord,Show)

instance Num Zahlraum_0_10 where ...
```

A.2 Machen Sie den Neuen Typ `Funktion` zu Instanzen der Typklassen `Eq` und `Show`. Zwei Funktionen sind *gleich*, wenn ihr Bild für alle Argumente übereinstimmt, *ungleich* sonst. Die Funktion `show` der Typklasse `Show` soll `Funktion`-Werte mengenähnlich in Form aller Argument/Bild-Paare aufsteigend nach Argumentwerten darstellen, eingeschlossen in geschweifte Klammern, z.B.: `show f ->> "{(N,II),(I,III),(II,IV),(III,V),(IV,VI),(V,VII),(VI,VIII),(VII,IX),(VIII,X),(IX,F),(X,F),(F,F)}"` für `f :: Funktion` mit `f z = \z -> z + II`:

```
newtype Funktion = Fkt { f :: Zahlraum_0_10 -> Zahlraum_0_10 }

instance Eq Funktion where ...

instance Show Funktion where ...
```

Damit sind wir bereit, einige Beispiele homogener Mengen zu implementieren:

A.3 Machen Sie die Typen:

(a) `Int`

(b) `Zahlraum_0_10`

(c) `Funktion`

im Mengensinn zu Instanzen der Typklasse `Menge_von`. Sind Argumente von Mengenoperationen, -relationen keine Mengen (also nicht duplikatfrei), wird stets die Funktion `error "Fehler"` aufgerufen:

```
instance Menge_von Int where ...

instance Menge_von Zahlraum_0_10 where ...

instance Menge_von Funktion where ...
```

Auch polymorphe Typen sind als Elementtyp von Mengen möglich. Mit den folgenden Instanzbildungen werden Mengen über beliebigen Paartypen und über Binärbäumen mit je einer typgleichen Blatt- und Knotenbenennung möglich:

A.4 Machen Sie die Typen `(Paar a b)` und `(Baum a)` mit:

```
newtype Paar a b = P (a,b) deriving (Eq,Show)
data Baum a = Blatt a | Knoten (Baum a) a (Baum a) deriving (Eq,Show)
```

im Mengensinn zu Instanzen der Typklasse `Menge_von`. Sind Argumente von Mengenoperationen, -relationen keine Mengen (also nicht duplikatfrei), wird stets die Funktion `error "Fehler"` aufgerufen:

```
instance (Eq a,Eq b) => Menge_von (Paar a b) where ...

instance Eq a => Menge_von (Baum a) where ...
```

Über denselben Typen möchten wir auch Multimengen bilden können. Da kein Typ ein zweites Mal zu einer Instanz einer Typklasse gemacht werden kann, führen wir zunächst den polymorphen Neuen Typ `(ElemTyp a)` ein:

```
newtype ElemTyp a = ET a
```

A.5 Machen Sie den Typ `(ElemTyp a)` zu Instanzen der Typklassen `Eq` und `Show`. Zwei `(ElemTyp a)`-Werte sind gleich, wenn ihre `a`-Werte gleich sind, ungleich sonst. Die (von `show` gelieferte) Zeichenreihendarstellung von `(ElemTyp a)`-Werten stimmt mit der ihrer `a`-Werte überein:

```
instance Eq a => Eq (ElemTyp a) where ...

instance Show a => Show (ElemTyp a) where ...
```

Damit sind wir bereit, einige Beispiele homogener Multimengen zu implementieren, wobei wir vom polymorphen Elementtyp `(ElemTyp a)` ausgehen:

A.6 Machen Sie den polymorphen Typ `(ElemTyp a)` im Multimengensinn zu einer Instanz der Typklasse `Menge_von`:

```
instance Eq a => Menge_von (ElemTyp a) where ...
```

Als nächstes wollen wir pseudoheterogene (d.h. heterogen erscheinende) Mengen modellieren. Dazu führen wir als Elementtyp den Typ `(PH_ElemTyp a b c d e)` ein:

```
 -- Pseudoheterogener Elementtyp
 data PH_ElemTyp a b c d e = A a | B b | C c | D d | E e deriving (Eq,Show)
```

A.7 Machen Sie den Typ `(PH_ElemTyp a b c d e)` im Mengensinn zur Instanz der Typklasse `Menge_von`. Sind Argumente von Mengenoperationen, -relationen keine Mengen (also nicht duplikatfrei), wird stets die Funktion `error "Fehler"` aufgerufen:

```
instance (Eq a,Eq b,Eq c,Eq d,Eq e)
   => Menge_von (PH_ElemTyp a b c d e) where ...
```

Auch hier gilt, dass wir einen Typ nicht zweimal zu einer Instanz einer Typklasse machen können. Um auch pseudoheterogene Multimengen in vergleichbarer Weise modellieren zu können, führen wir den Typ `(PH_ElemTyp' p q r)` ein:

```
 -- Pseudoheterogener Elementtyp
data PH_ElemTyp' q r s = Q q | R r | S s deriving (Eq,Show)
```

A.8 Machen Sie den Typ `(PH_ElemTyp' p q r)` im Multimengensinn zur Instanz der Typklasse `Menge_von`:

```
instance (Eq p,Eq q,Eq r) => Menge_von (PH_ElemTyp' p q r) where ...
```

A.9 **Ohne Abgabe, ohne Beurteilung:**

(a) Warum modelliert unser Ansatz pseudoheterogene, nicht heterogene Mengen und Multimengen? (pseudo (griech.) bedeutet falsch).

(b) Warum ist der Typkontext `Eq a =>` in der Deklaration der Typklasse `Menge_von` erforderlich? Was stellt der Typkontext sicher?

(c) Können die Typen `Funktion'` und `Funktion''` mit:

```
type Funktion'     = Int -> Int
newtype Funktion'' = F'' { g ::  Int -> Int }
```

zu Instanzen von `Menge_von` gemacht werden? Begründen Sie Ihre Antwort.

(d) Welche Art(en) von Polymorphie kommen auf diesem Aufgabenblatt vor? Wo? Bei Datentypen? Bei Funktionen?

(e) Wo kommen Funktionen höherer Ordnung auf diesem Aufgabenblatt vor?

(f) Bilden einige Funktionen Ihrer Implementierung hierarchische Systeme? Welche? Wie sehen die Aufrufgraphen dieser Systeme aus?

(g) Welche Funktionen haben Sie rekursiv programmiert? Um welchen Rekursionstyp handelt es sich jeweils?

(h) Haben Sie einige Funktionen mithilfe von Listenkomprehensionen programmiert? Wenn ja, welche? Wie wäre es ohne Listenkomprehensionen gegangen?

(i) Warum kann für Multimengen die Fehlerbehandlung entfallen?

(j) Hätten weitere oder/und andere Protoimplementierungen in `Menge_von` helfen können, Implementierungsarbeit zu sparen? Wie?

A.10 **Ohne Beurteilung:** Beschreiben Sie für jede Rechenvorschrift in einem Kommentar knapp, aber gut nachvollziehbar, wie die Rechenvorschrift vorgeht.

A.11 **Ohne Abgabe, ohne Beurteilung:** Testen Sie alle Funktionen umfassend mit aussagekräftigen eigenen Testdaten.

Ausblick: Mithilfe von Typkonstruktorklassen (statt Typklassen wie `Menge_von` könnten weitere Einsparungen bei den Implementierungen der verschiedenen Mengen- und Multimengentypen erreicht werden. Typkonstruktorklassen sind Thema der Vorlesung zur fortgeschrittenen funktionalen Programmierung.

B Papier- und Bleistiftaufgaben

Entfallen auf Angaben 5 bis 7.

Iucundi acti labores.
Getane Arbeiten sind angenehm.
Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller