Aufgabe 2: Spießgesellen

Teilnahme-ID: 58227

Bearbeiter/-in dieser Aufgabe:
Dan Vierling

4. November 2021

Inhaltsverzeichnis

1 Lösungsidee

Um einen guten Überblick zu erhalten, werden Donalds Beobachtungen so in eine Tabelle eingetragen:

Früchte	Apfel	Banane	Brombeere	Erdbeere	Pflaume	Weintraube
Micky	x	x	x	-	-	-
Minnie	-	x	-	-	x	x
Gustav	x	-	x	x	-	-

Schüsseln	1	2	3	4	5	6
Micky	x	-	-	x	x	-
Minnie	-	-	x	-	x	x
Gustav	x	x	-	x	-	-

In den Feldern der Tabelle wird markiert, ob eine Person an einer Schüssel war oder nicht bzw. eine Frucht hatte oder nicht. Die Aufgabe ist es, die Schüsseln zu beschränken, in der eine Frucht jeweils sein kann. Eine Frucht kann genau dann in einer Schüssel sein, wenn alle die Leute, die die Frucht hatten, an der Schüssel waren und sonst keiner. Für die Tabelle bedeutet das: Dann, wenn die Spalte unter Frucht und Schüssel gleich ist. Dementsprechend kann man gruppieren:

x - x	- x -	x x -	- - x
Apfel, Brombeere	Pflaume, Weintraube	Banane	Erdbeere
1, 4	3, 6	5	2

Diese Gruppen geben die genauste Information über zusammengehörende Schüsseln und Früchte. Donald will einen Spieß mit Apfel, Brombeere und Weintraube, dafür muss er zu den Schüsseln gehen, die in der Spalte unter der jeweiligen Frucht stehen. Für keine dieser Früchte jedoch kann hier eindeutig eine einzelne Schüssel bestimmt werden. Bei Apfel und Brombeere ist das jedoch kein Problem, denn in ihrer Gruppe befindet sich keine weitere Frucht, die Donald nicht haben will. Wenn er also zu Schüssel 1 und 4 geht, hat er sicher Apfel und Brombeere - nur die irrelevante Reihenfolge ist unbekannt.

Es gibt aber noch die Beobachtung von Daisy. Mit ihrer Beobachtung kann man Pflaume und Weintraube und die Schüsseln 3 und 6 unterscheiden.

Früchte	...	...	Pflaume	Weintraube
Micky			-	-
Minnie			x	x
Gustav			-	-
Daisy			x	-

Schüsseln	...	3	...	6
Micky		-		-
Minnie		x		x
Gustav		-		-
Daisy		-		x

Daraus ergibt sich die genauere Zuordnung, mit der Donald zufrieden sein kann:

Apfel, Brombeere	Banane	Erdbeere	Pflaume	Weintraube
1, 4	5	2	6	3

2 Umsetzung

Dem Programm wird die einzulesende Datei als Kommandozeilenparamter übergeben. Heraus kommen mehrere Infozeilen wie diese:

```
Apfel und Brombeere > 1 und 4
Weintraube | Pflaume > 3 und 6
```

Das ist das Beispiel von oben und man kann ablesen, welche gewünschten Früchte sich in welchen Schüsseln befinden und auf welche weiteren Früchte Donald stoßen könnte.
Das Programm arbeitet nach der oben erklärten Weise. Die genauen Implementationsdetails sind im nächsten Abschnitt aufgezeigt.

2.1 Types.hs

```haskell
module Types where

import Data.Text

type Queue = Int -- Im Programm wurde Schlange statt Schüssel verwendet
type Fruit = Text
type Spit = ([Queue], [Fruit]) -- Steht sowohl für die Spieße, die Donald beobachtet
                               -- hat als auch für die extrahierten Gruppen

type Input = (Int, [Fruit], [Spit]) -- Der Input aus der Datei
type Info = ([Fruit], [Fruit], [Queue]) -- Eine Infozeile

queues :: Spit -> [Queue]
queues = fst

fruits :: Spit -> [Fruit]
fruits = snd
```

2.2 Groups.hs

– Über die Tabelle zu den extrahierten Gruppen

```haskell
module Groups where

import Types
import Data.List (groupBy,nub,sortBy,transpose)
```

```
import Data.Array.Unboxed
import Data.Function (on)

type Column a = (a, UArray Int Bool)
type Table a = [Column a]
```

Im Programm wird zuerst die Vereinigungsmenge aller beobachteten Früchte und aller Schüsseln mit `observed` gebildet.

```
observed :: Eq a => [[a]] -> [a]
observed = nub . concat
```

Diese beiden Mengen werden als Kopfzeile genommen und zwei Tabellen wie am Anfang erstellt, in die die einzelnen Beobachtungen eingetragen werden. `tick` erstellt die angekreuzten Zeilen.

```
makeTable :: Eq a => [a] -> [[a]] -> Table a
makeTable heads rows = zip heads $ fmap arrTransform $ transpose $ fmap tick rows
  where
    arrTransform = listArray (1,length heads) -- performantere Bitarrays
    tick row = fmap (`elem` row) heads
```

Die Spalten werden gruppiert, indem sie entsprechend ihrer „DNA“ zuerst sortiert werden und dann gleiche, aufeinander folgende Spalten zusammengefasst werden.

```
groupTable :: Table a -> [Table a]
groupTable = groupBy ((==) `on` snd) . sortBy (compare `on` snd)
```

`merge` nimmt eine Gruppe von Schüsseln und eine von Früchten und fügt sie so zusammen, dass eine Spalte wie in der zweiten Tabelle entsteht.

```
merge :: Table Queue -> Table Fruit -> Spit
merge qs fs = (fmap fst qs, fmap fst fs)
```

2.3 Extract.hs

– Von den Gruppen zu den Informationen, wo die Wunschfrüchte sind

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE OverloadedStrings #-}

module Extract where

import Types
import Data.Bifunctor (bimap, second)
import Data.List (sort, mapAccumR)
```

`overlap` nimmt zwei Mengen A, B und gibt $(A \setminus B, A \cap B, B \setminus A)$ zurück. Die Implementation unten funktioniert nur, wenn die Listen sortiert sind.

```
class Overlap a where
  overlap :: a -> a -> (a,a,a)

instance (Ord a) => Overlap [a] where
  overlap ls [] = (ls,[],[])
  overlap [] rs = ([],[],rs)
  overlap xxs@(x:xs) yys@(y:ys) =
    case compare x y of
      LT -> let (ls,os,rs) = overlap xs yys in (x:ls,os,rs)
      EQ -> let (ls,os,rs) = overlap xs  ys in (ls,x:os,rs)
      GT -> let (ls,os,rs) = overlap xxs ys in (ls,os,y:rs)
```

```
instance Overlap Spit where
  overlap l r = ((lq,lf), (q,f), (rq,rf))
    where
      (lq,q,rq) = overlap (queues l) (queues r)
      (lf,f,rf) = overlap (fruits l) (fruits r)
```

Deshalb wird hiermit die Eingabe, also nur die Wünsche und die beobachteten Spieße, sortiert.

```
sortInput :: Input -> Input
sortInput = bimap sort (fmap sortSpit)
  where
    sortSpit = bimap sort sort
```

`extract` nimmt die Liste der Wünsche und die Gruppen, die vorher extrahiert wurden. Mit `helper` wird über die Gruppen gemapt. Die Früchte, die sowohl in einer Gruppe als auch im Wunsch enthalten sind, heißen „erfüllt“ (`full`), der Rest der Gruppe „ungewollt“ und der Rest des Wunsches „unerfüllt“ und wird dem nächsten Aufruf von `helper` übergeben. Wenn in der Gruppe kein Wunsch erfüllt wurde, ist die Info über diese Gruppe unnötig. Am Ende steht also eine Liste von Infos (welche gewünschten Früchte, mit welchen Ungewollten, in welchen Schüsseln) und eine Liste von übrig gebliebenen Wünschen, die in keiner beobachteten Gruppe vorkamen.

```
extract :: [Fruit] -> [Spit] -> ([Fruit],[Info])
extract = second catMaybes ... mapAccumR helper
  where
    (...) = (.).(.)
    helper wish spit = let (unwanted,full,unfull) = overlap (fruits spit) wish
                           info
                            | null full = Nothing
                            | otherwise = Just (full,unwanted,queues spit)
                       in (unfull,info)
```

Diese Wünsche müssen sich in den Schüsseln befinden, die nicht beobachtet wurden. Diese Menge findet `getUnknown` mit der Anzahl aller Schüsseln und der Menge der Beobachteten heraus.

```
getUnknown :: Int -> [Int] -> [Int]
getUnknown num obs = filter (not . (`elem` obs)) [1..num]
```

Gibt es also Wünsche, die sich in keiner beobachteten Gruppe befinden, befinden sie sich mit möglicherweise weiteren Früchten in den unbeobachteten Schüsseln. `addUnknown` hängt diese Info an die restliche an, um die Ausgabefunktion zu erleichtern.

```
addUnknown :: [Queue] -> ([Fruit], [Info]) -> [Info]
addUnknown qs (fs,xs)
  | null fs   = xs
  | diff == 0 = (fs,[],qs) : xs
  | otherwise = (fs,["unbekannt"],qs) : xs
  where
    diff = length qs - length fs
```

2.4 Lib.hs

```
module Lib where

import Types
import Groups
import Extract
import Control.Applicative (liftA2)
```

`guard` wirft eine Fehlermeldung, wenn ein Testergebnis falsch ist

```haskell
guard :: String -> Bool -> Either String ()
guard _ True = return ()
guard ms False = Left ms
```

Wenn beim Zusammenführen einer Gruppe von Schüsseln und Früchten etwas nicht stimmt, können die Beobachtungen an einer Stelle nicht zusammengepasst haben. Das wird hier versichert.

```haskell
merges :: Table Queue -> Table Fruit -> Either String Spit
merges qs fs = check >> return (merge qs fs)
  where
    checkLength xs ys = length xs == length ys
    checkContent xs ys = snd (head xs) == snd (head ys)
    checkBoth = liftA2 (liftA2 (&&)) checkLength checkContent
    check = guard "Beobachtungen passen nicht zusammen" $ checkBoth qs fs
```

Wenn die beiden Vereinigungsmengen von 1. nicht gleich groß sind, liegt ebenfalls ein Fehler vor. Hier noch mal ein Überblick:

1. Die Vereinigung der Beobachteten Früchte und Schüsseln wird berechnet
2. Die Beobachtungen kommen in eine Tabelle und es werden je Gruppen extrahiert
3. Die Gruppen werden paarweise zu zusammengehörenden Früchten und Schüsseln verbunden
4. Daraus werden Informationen über die Wünsche gewonnen

```haskell
compute :: Input -> Either String [Info]
compute (num, wish, observation) = do
  let (qObs, fObs) = unzip observation
      qUnion = observed qObs -- 1.
      fUnion = observed fObs
  guard "Die Anzahlen aller beobachteten Schuesseln und Fruechte sind nicht gleich" $
  ↪  length qUnion == length fUnion

  let qGroups = groupTable $ makeTable qUnion qObs -- 2.
      fGroups = groupTable $ makeTable fUnion fObs
      maybeGroups = zipWith merges qGroups fGroups -- 3.
  groups <- sequence maybeGroups
  return $ addUnknown (getUnknown num qUnion) $ extract wish groups -- 4.

makeDonaldHappy :: Input -> Either String [Info]
makeDonaldHappy = compute . sortInput
```

3 Beispiele

Wenn Donald eine Datei mit den Beobachtungen von Micky, Minnie, Gustav und Daisy eingibt, erhält er die Ausgabe:

```
Weintraube > 3
Apfel und Brombeere > 1 und 4
```

Was genau dem Entspricht, was man vom Anfang erwartet. Hier drei Beispiele von der Bwinf-Website:
spiesse5.txt

```
Clementine > 20
Apfel, Grapefruit und Mango > 1, 4 und 19
Tamarinde > 12
Pflaume > 10
Himbeere > 5
Nektarine > 14
Dattel > 6
Banane und Quitte > 3 und 9
Orange und Sauerkirsche > 2 und 16
```

spiesse6.txt

```
Vogelbeere > 6
Erdbeere > 10
Clementine > 7
Quitte > 4
Orange > 20
Himbeere > 18
Rosine und Ugli > 11 und 15
```

spiesse7.txt

```
Dattel, Mango und Vogelbeere > 6, 16 und 17
Tamarinde und Zitrone > 5 und 23
Ugli | Banane > 18 und 25
Clementine > 24
Sauerkirsche und Yuzu > 8 und 14
Apfel, Grapefruit und Xenia | Litschi > 3, 10, 20 und 26
```

Es können gute Aussagen über Donalds Wünsche getroffen werden.
In der folgenden Abwandlung des ersten Beispiels wird die Rolle der beiden „`Unknown`"-Funktionen gezeigt. Donald wünscht sich zusätzlich Maracuja, hat jedoch niemanden damit beobachtet. Außerdem gibt es 2 weitere Früchte, also 8.

```
8
Apfel Brombeere Weintraube Maracuja
4
1 4 5
Apfel Banane Brombeere
3 5 6
Banane Pflaume Weintraube
1 2 4
Apfel Brombeere Erdbeere
2 6
Erdbeere Pflaume
```

Heraus kommt:

```
Maracuja | unbekannt > 7 und 8
Weintraube > 3
Apfel und Brombeere > 1 und 4
```

Ändert man die letzte Zeile und setzt eine 7 ein, wurden die Schüsseln 1 - 7 beobachtet, aber nur 6 Früchte.

```
...
7 6
Erdbeere Pflaume
```

In der Tat, ausgegeben wird:

```
Die Anzahlen aller beobachteten Schuesseln und Fruechte sind nicht gleich
```

Nimmt man eine Zahl von 1, 3-6, sind die Anzahlen zwar gleich, aber:

```
Beobachtungen passen nicht zusammen
```

Als letztes habe ich eine Datei (benchmark.txt, in der Einsendung) mit 256 „Früchten", 670 Beobachtungen mit durchschnittlich 15 Früchten am Spieß und ca. 100 Wünschen generiert. Die Tests waren trotzdem sofort fertig, das größere Problem ist also, dass Donald bei derartigen Events genügend aussagekräftige Beobachtungen macht.