Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
25.10.2020
Deadline: 23.11.2020, 23:59

Functional Programming and Verification

Sheet 2

Sustainability Exercises

The "Nachhaltige Hochschultage Bayern" are taking place this week (16.–19.11). Consider attending some of the workshops and lectures. More information can be found here.

Tutorial Exercises

Exercise T2.1 Axiom of Comprehension

Using list comprehension, implement the following functions:

a) Write a function `allSums :: [Integer] -> [Integer] -> [Integer]` such that, for every x, y in xs, ys, `allSums xs ys` contains $x + y$.

b) Write a function `evens :: [Integer] -> [Integer]` that removes all odd numbers of a list.

c) Write a function `nLists :: [Integer] -> [[Integer]]` such that `nLists xs` returns a list that contains every list $[1, \ldots, x]$ for each x in xs.

d) Using only one list comprehension, write a function

```
allEvenSumLists :: [Integer] -> [Integer] -> [[Integer]]
```

such that `allEvenSumLists xs ys` computes `nLists (evens (allSums xs ys))`. Write a QuickCheck test that verifies this equivalence.

Exercise T2.2 Cantor's Paradise

For the following exercises, you can use the function `elem :: Integer -> [Integer] -> Bool` that returns whether an element is contained in a list.

The great Georg Cantor did not only teach us about encodings of pairs, but also created the realm of set theory. In this exercise, we shall praise this idea by encoding basic set-theoretic notions using our beloved Haskell lists: we say that a list `l :: [Integer]` is a set if and only if `l` contains no duplicates.

a) Search for the function `nub` in `Data.List` on Hoogle (use your VSCodium-Hoogle-Plugin). Use it to define a function `toSet :: [Integer] -> [Integer]` such that `toSet l` removes all duplicates of `l`.

b) Define a function `isSet :: [Integer] -> Bool` such that `isSet l` holds if and only if `l` is a set. Check that `isSet (toSet s)` holds using QuickCheck.

c) Define a function `union :: [Integer] -> [Integer] -> [Integer]` such that `union s t` returns the union $s \cup t$. Write some QuickCheck tests to verify your implementation:

- Check that `union s t` indeed returns a set.
- Mathematically, we have $a \in S \cup T$ if and only if $a \in S$ or $a \in T$. Check that your implementation satisfies this property.

d) (Optional) Define a function `intersection :: [Integer] -> [Integer] -> [Integer]` such that `intersection s t` returns the intersection $s \cap t$. Again, write some QuickCheck tests to verify your implementation.

e) (Optional) Define a function `diff :: [Integer] -> [Integer] -> [Integer]` such that `diff s t` returns the difference $s \setminus t$.

Note: You can use `quickCheckWith (stdArgs { maxSize=4, maxDiscardRatio=40 })` to check your properties. This will cause QuichCheck to only generate small parameters (up to size 4) and give up if the number of discarded tests exceeds 40. You can use `verboseCheckWith` instead of `quickCheckWith` to see all generated parameters.

Exercise T2.3 $\frac{Fract}{ions}$

We can represent a fraction $\frac{a}{b}$ as a tuple `(a,b)`. Two fractions should be equal whenever they represent the same value, e.g. `(1,2)` and `(3,6)` represent the same value.

a) Write a function

```
eqFrac :: (Integer, Integer) -> (Integer, Integer) -> Bool
```

that decides whether two fractions are equal.

b) Write some QuickCheck tests that verify interesting properties of `eqFrac`, e.g. reflexivity, symmetry, the cancellation law $m/n = (m \cdot k)/(n \cdot k)$, etc.

Exercise T2.4 (Optional) Potentially Dangerous$^{2^{2^{2^2}}}$

The function

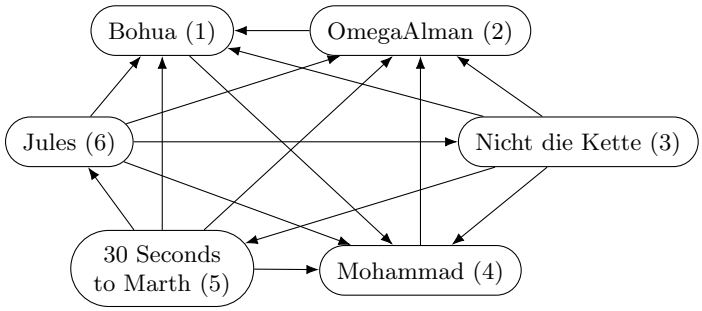
```
pow2 :: Integer -> Integer
pow2 0 = 1
pow2 n | n > 0 = 2 * pow2 (n - 1)
```

implements $n \mapsto 2^n$ für $n \geq 0$. For a given n, the computation takes n steps. For example:

$$2^{100} = 2 \cdot 2^{99} = 2 \cdot 2 \cdot 2^{98} = 2 \cdot 2 \cdot 2 \cdot 2^{97} = \cdots = \overbrace{2 \cdot 2 \cdot \ldots \cdot 2}^{100 \text{ times}} \cdot 1$$

Create a more efficient version that takes at most $\lceil 2 \log_2 n \rceil$ steps. The identities $2^{2n} = (2^n)^2$ and $2^{2n+1} = 2 \cdot 2^{2n}$ might be useful. For example:

$$\begin{aligned} 2^{100} &= (2^{50})^2 = ((2^{25})^2)^2 = ((2 \cdot 2^{24})^2)^2 = ((2 \cdot (2^{12})^2)^2)^2 = ((2 \cdot ((2^6)^2)^2)^2)^2 \\ &= ((2 \cdot (((2^3)^2)^2)^2)^2)^2 = ((2 \cdot (((2 \cdot 2^2)^2)^2)^2)^2)^2 \end{aligned}$$



Player	Dominion	Dominators
1	$\{4\}$	$\{2,3,5,6\}$
2	$\{1\}$	$\{3,4,5,6\}$
3	$\{1,2,4,5\}$	$\{6\}\}$
4	$\{2\}$	$\{1,3,5,6\}$
5	$\{1,2,4,6\}$	$\{3\}\}$
6	$\{1,2,3,4\}$	$\{5\}$

Figure 1: An example tournament graph and the corresponding lists of dominions and dominators for each player.

Homework

You need to collect 7 out of 10 points (P) to collect a coin.

Hint: some functions from the `List` library may be useful for these exercises.

Exercise H2.1 Guessing games [a: 2P, b: 2P]

This exercise is all about games; however, not your everyday boring games like Catch but rather exciting mathematical guessing games. As it would disappointing to just guess the winner of a game, we instead want to calculate the winner of the following games using the exciting Haskell programming language. In both exercises, the input is a list of pairs where the first component of the pair is the name of the player and the second component is a number representing the player's bid. You may assume that there is at least one player, that the names are unique and that the bids lie between 0 and 100.

a) The first game is "Guess $\frac{2}{3}$ of the average". Let a be the avarage of the player's bids. Write a function `twoThirdsAverageWinners :: [(String, Int)] -> [String]` that outputs those players whose guess is the least far off from $\lfloor \frac{2*a}{3} \rfloor$. The order of the players does not matter.

b) The second game is "Unique bid auction" which is an auction with a spin. Write a function `lowestUniqueBidder :: [(String, Int)] -> String` that outputs the bidder with the lowest unique bid. If no such player exists, output `"Nobody"`.

Exercise H2.2 Choose only the best [a,b,c,d,e,f: 1P each]

A long-standing tradition of (part of) the *Chair for Logic and Verification* is holding tournaments in *Super Smash Bros.*™ *Melee for the Nintendo*® *GameCube*.

The tournament mode is a *round robin*: each of the players (labelled 1 to n) plays against each other player exactly once (no ties). The result is a *tournament graph* (see Figure 1), in which an edge $i \to j$ indicates that player i defeated player j.

The MC Sr now has the task to determine who the best players are. Half-remembering some social choice theory he learnt long ago, he wants to use some of these fancy concepts: given the tournament results, you will implement various ways of returning a set of *winners*.

The input that you receive is a list of lists `tournament :: [[Int]]`, where the $i-1$-th element contains the list of all the players that player i defeated. This is called the *dominion* of player i, formally $D(i) := \{j \mid i \to j\}$ You can (and should) use the following function to compute it:

```
dominion :: [[Int]] -> Int -> [Int]
dominion tournament i = rel !! (i - 1)
```

To find out whether i defeated j, you can write `j `elem` dominion rel i`.

You may also use the following function that returns the list of all players (i.e. $[1, \ldots, n]$):

```
players :: [[Int]] -> [Int]
players tournament = [1..length tournament]
```

The MC would now like you to implement three different ways to determine the best overall players: Copeland's rule (CO), the *uncovered set* (UC), and the *top cycle* (TC). Your plan of attack is to implement the following functions:

a) The *dominators* of i are the set of all players who defeated i, i.e. $\overline{D}(i) := \{j \mid j \to i\}$.

```
dominators :: [[Int]] -> Int -> [Int]
dominators tournament i = ...
```

b) We say that i *covers* j (written as $i \; C \; j$) if i defeated everyone that j defeated.

```
covers :: [[Int]] -> Int -> Int-> Bool
covers tournament i j = ...
```

c) A set of players X is called *dominant* if it is non-empty and every player in it defeated every player not in it.

```
dominant :: [[Int]] -> [Int] -> Bool
dominant tournament is = ...
```

d) CO is defined as the set of those players who defeated the most other players, i.e. who have the maximum dominion size, i.e. $\text{CO} := \arg\max_i |D(i)|$.

```
copeland :: [[Int]] -> [Int]
copeland tournament = ...
```

Hint: the function `maximum xs` tells you what the largest element in `xs` is.

e) UC is the set of all players that are covered by no one (except themselves, of course), i.e. $\text{UC} := \{x \mid \nexists y.\ y \neq x \wedge y \; C \; x\}$.

```
uncoveredSet :: [[Int]] -> [Int]
uncoveredSet tournament = ...
```

f) TC is the smallest dominant set (this is unique).

```
topCycle :: [[Int]] -> [Int]
topCycle tournament = ...
```

Hint: the function `subsequences` from `Data.List` might help you to compute the set of all "subsets" of $\{1, \ldots, n\}$. You can use the helper function `shortest` from the template to get the shortest list from a list of lists.

Note that whenever one of your functions receives a set of players as an input from the MC Sr's tests, you may assume that it is sorted and contains no duplicates. When you return a set, it need not be sorted and it may contain duplicates.

Example: Let's look at our example graph in Figure 1. The dominions and dominators of each player are listed in the table. Players 3,5,6 cover players 1,2,4, and these are the only coverings. Consequently, UC $= \{3, 5, 6\}$. Players 3,5,6 all have 4 wins, so CO $= \{3, 5, 6\}$. The only dominant sets are $\{3, 5, 6\}$ and $\{1, 2, 3, 4, 5, 6\}$, so TC $= \{3, 5, 6\}$.

Only for the *Wettbewerb*: the only functions that count for the competition are the three last ones and the winning criterion is again least number of tokens. You may use all other functions from Exercise H2.2 for free when implementing them. However: to make things a bit more interesting, the MC Sr requires you to implement these functions in such a way that they cope with large inputs (e.g. 1000 players) and still terminate in a reasonable time (i.e. a few seconds). You can check whether that is the case with some of the provided tournament data on Zulip.

> Climate change isn't an "issue" to add to the list of things to worry about, next to health care and taxes. It is a civilizational wake-up call. A powerful message – spoken in the language of fires, floods, droughts, and extinctions – telling us that we need an entirely new economic model and a new way of sharing this planet. Telling us that we need to evolve.
>
> — Naomi Klein in "This Changes Everything: Capitalism vs. the Climate"