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

WS 2019/20
25.10.2019
Deadline: 06.11.2019, 00:00

# Functional Programming and Verification

**Sheet 2**

## Tutorial Exercises

### Exercise T2.1 Axiom of Comprehension

Using list comprehension, implement the following functions:

1. Write a function `allSums :: [Integer] -> [Integer] -> [Integer]` such that, for every $x, y$ in $xs, ys$, `allSums xs ys` contains $x + y$.
2. Write a function `evens :: [Integer] -> [Integer]` that removes all odd numbers of a list.
3. Write a function `nLists :: [Integer] -> [[Integer]]` such that `nLists xs` returns a list containing every list $[1, \ldots, x]$ for $x$ in $xs$.
4. 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 property.

### Exercise T2.2 Cantor's Paradise

The great Georg Cantor did not only teach us about encodings of pairs, but also created the realm of set theory. In this exercise (and its marvellous follow-up homework), we shall praise this idea by encoding basic set-theoretic notions using our beloved Haskell lists.

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

1. We say that a list `l :: Integer` is a set if and only if `l` contains no duplicates. Define a function `toSet :: [Integer] -> [Integer]` such that `toSet l` removes all duplicates of `l`.
2. 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.
3. 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:
   a) Check that `union s t` indeed returns a set.
   b) 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.

4. 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.
5. Define a function `diff :: [Integer] -> [Integer] -> [Integer]` such that `diff s t` returns the difference $s \setminus t$.

*Note:* 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.

1. Write a function

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

   that decides wheter two fractions are equal.
2. Write some QuickCheck tests that verify interesting properties of `eqFrac`, e.g. reflexitivity, symmetry, the cancellation law $m/n = (m \cdot k)/(n \cdot k)$, etc.

### Exercise T2.4 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:

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

## Homework

You need to collect 4 out of 6 points (P) to pass this sheet. Blocks marked with **Trivia** contain information that is not needed to solve the exercises but almost surely is appealing to the interested reader.

Hint: Some functions from the `List` library may be useful for these exercises. In particular, consider `null`, `length`, `minimum`, and `maximum`.

### Exercise H2.1 Let's Play a Game [1+2: 1P, 3: 1P]

You are the *Master of Competition* for an exciting game of *Guess the Average*. In this game, each competitor guesses a natural number between 0 and 100 that they think will be closest to the average guess.

You receive the guesses in the form of an association list of type `[(String,Int)]`. Each entry in the list is a pair of the competitor's name and their guess. We use `Int` instead of `Integer` so that you can use functions from the List library, some of which return `Int` for historical reasons.

Your task is to determine the winner(s) of the game.

1. Begin by writing a function

   ```
   removeInvalidGuesses :: [(String,Int)] -> [(String,Int)]
   ```

   that removes any invalid guesses, i.e. guesses which do not lie between 0 and 100 or were submitted without a name.

2. Write a function

   ```
   average :: [(String,Int)] -> Int
   ```

   that computes the average guess. If there are no guesses, simply return 0. (We use integer division here, since we have not introduced floating point numbers yet)

3. Write a function

   ```
   winners :: [(String,Int)] -> [String]
   ```

   that computes the competitors whose guesses are closest to the average guess. Be sure to first restrict the input to valid guesses.

### Exercise H2.2 (**Competition**) Bernoulli numbers [1P]

This week's competition exercise revolves around one of the MC's favourite sequence of numbers: the *Bernoulli numbers* $B_n$. The following table lists the first nine of them:

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $B_n$ | 1 | $-\frac{1}{2}$ | $\frac{1}{6}$ | 0 | $-\frac{1}{30}$ | 0 | $\frac{1}{42}$ | 0 | $-\frac{1}{30}$ | 0 | $\frac{5}{66}$ | ... |

Jacob Bernoulli

**Trivia**

These numbers appear in many places in mathematics for no apparent reason: For instance, Faulhaber's formula states that the sum of the $p$-th powers of the first $n$ integers is a polynomial in $n$, namely

$$\sum_{k=1}^{n} k^p = \frac{1}{p+1} \sum_{j=0}^{p} (-1)^j \binom{p+1}{j} B_j n^{p+1-j} \ .$$

You probably know the case $p = 1$, which is Gauß's sum formula $\sum_{k=1}^{n} k = \frac{1}{2} n(n+1)$.

They also make an appearance in the famous Riemann $\zeta$ function, which has deep connections to the multiplicative structures of the integers and the distribution of prime numbers: At negative integers, it takes the value

$$\zeta(-n) = -\frac{B_{n+1}}{n+1} \ .$$

Your task is to implement the function

```
bernoulli :: Integer -> Rational
```

that, given a non-negative integer $n$, returns $B_n$. Note that since Bernoulli numbers are *rational* numbers, we use Haskell's `Rational` type.

The most basic way to compute Bernoulli numbers is by using the following recurrence:

$$B_0 = 1 \qquad\qquad B_n = \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k}{k-n-1} \text{ for } n > 0$$

For the competition, the MC will again rank the solutions by their number of tokens (and possibly other criteria like performance or beauty to break any ties). For reference, the MC Sr's shortest solution following the above approach has 42 tokens (not counting the type signature). However, the above recurrence is just a suggestion – there are many other ways to compute Bernoulli numbers and some of them might lead to a shorter solution. Rumour has it that there is a solution with only 26 tokens. Can you beat that?

*Hints:*

- The function `sum :: [Rational] -> Rational` from the library computes the sum of a list of values, e. g. `sum [1,2,3] = 6`.
- For integers $m$ and $n$, you can write the fraction $\frac{m}{n}$ as `m % n` in Haskell. This is also the format in which fractions will be printed.
- The function `fromIntegral` from the library converts an `Int` or `Integer` into a `Rational`, e. g. `fromIntegral 3 == 3 % 1`.

- To compute the binomial coefficient $\binom{m}{k}$, you may use the `choose` function from the template, e. g. `(49 `choose` 6) == 13983816`. However, note that if you use it and want to take part in the competition, you have to put it within the `{-WETT-}` tags!
- It's okay if your implementation is fairly slow for inputs above 10 or 15. However, it has to pass the tests on the server.
- **But:** No cheating! Your implementation has to work (theoretically) for *all* inputs, no matter how big, even if it practically takes forever. If you use a look-up table for all inputs $\leq 100$, hoping that inputs bigger than 100 will not get tested, the MC *will* find out and award you 0 points.
- Also: Do not plagiarise. The MC encourages you to find inspiration on the internet and textbooks, but do not copy-paste somebody else's code. You *must* write the code yourself.
- If you're not sure if what you are doing is allowed, you can ask the MC via `eberlm@in.tum.de`.

**Important:** If you submit a competition exercise, you agree that we are allowed to publish your name as part of the competition on our website. If you just want to submit a competition exercise as part of your homework without taking part in the competition, you can just remove the {`-WETT-`}...{`-TTEW-`} comments of your submission.

**Exercise H2.3** I G∅t the P∅wer [1: 1P, 2+3: 1P, 4+5: 1P]

Since we all very much do love set theory (and we want to get to know more mathematicians), we will now implement even more set-theoretic functions while also climbing the level of abstraction, that is we will build sets of sets (of sets...). For this, we provide you with two generalised helper functions from the tutorial

```
toSet :: Eq a => [a] -> [a]
union :: Eq a => [a] -> [a] -> [a]
```

that create a set and unify two sets, respectively, even if the set itself contains other sets. Do not worry about the "`Eq a =>`" annotation; this feature will be explained in an upcoming lecture.[1] For example, we have

```
toSet [[2],[1],[2]]   = [[2],[1]]
union [[2],[1]] [[]] = [[2],[1],[]]
union [[[1,2]]] [[[2]]] = [[[1,2]],[[2]]]
```

As a restriction, we cannot, however, unifiy sets that have different levels of nested sets; for example, an invokation of `union [[2]] [1]` will fail.

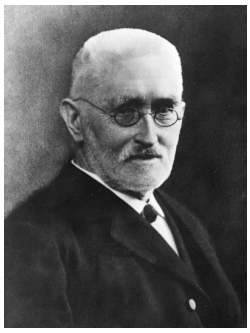
1. Define functions

   ```
   power :: [Integer] -> [[Integer]]
   subsetEq :: [Integer] -> [Integer] -> Bool
   ```

[1] "`Eq a`" basically restricts the function to types that work with equality.

Helen Keller (addendum to sheet 1)

Emanuel Sperner



Richard Dedekind

André Weil

such that

a) `power s` returns the power set $2^s$ of `s`, e.g. `power [1,2] = [[],[1],[2],[1,2]]`, and

b) `subsetEq s t` holds if and only if $s \subseteq t$.

2. Given two sets $s, t$ we say that $s, t$ are *comparable* if $s \subseteq t$ or $t \subseteq s$. If $s, t$ are not comparable, we say that $s, t$ are *incomparable*.

   Define a function

   ```
   comparable :: [Integer] -> [Integer] -> Bool
   ```

   such that `comparable s t` holds if and only if $s, t$ are comparable. For example, in our encoding, the sets `[1],[1,2]` as well as `[],[1]` are comparable whereas `[1,2],[1,3]` are incomparable.

3. Given a set of sets $A$, we say that $A$ is an *antichain* if and only if $s, t$ are incomparable for any distinct $s, t \in A$. Define a function

   ```
   isAntichain :: [[Integer]] -> Bool
   ```

   such that `isAntichain a` holds if and only if $a$ is an antichain. For example, in our encoding, the sets

   ```
   []        [[]]        [[1],[2]]        [[1],[2,3],[2,4,5]]
   ```

   are antichains whereas

   ```
   [[],[1]]        [[1],[1,2]]        [[1],[2,3],[1,4,5]]
   ```

   are not antichains.

4. Define a function

   ```
   antichains :: Integer -> [[[Integer]]]
   ```

such that `antichains n` returns a list of all antichains of the powerset $2^{\{1,\dots,n\}}$.[2] For example,

```
antichains 1 = [[], [[]], [[1]]]
antichains 2 = [[], [[]], [[1]], [[2]], [[1], [2]], [[1,2]]]
```

It is okay if your implementation is fairly slow for inputs above 3 – cf. note at the end of the sheet.

*Hint:* use the ~~force~~ power.

5. Define a function

```
maxAntichainSize :: [[[Integer]]] -> Int
```

such that `maxAntichainSize s` returns the size of the largest antichain in `s`. The function also needs to work for inputs that contain elements that are not antichains. If there are no antichains, simply return 0. Use the list function `length` to compute the size of a set.

Sperner's theorem now tells us the size of the maximum antichain of the set $2^{\{1,\dots,n\}}$, namely `maxAntichainSize (antichains n)` $= \binom{n}{\lfloor n/2 \rfloor}$ for any $n \in \mathbb{N}_+$. Write a QuickCheck test

```
prop_spernersTheorem :: Integer -> Property
```

that verifies this claim.

**Trivia**

You might now also wonder what the size of `antichains n` is; in other words: what is the number of antichains of the powerset $2^{\{1,\dots,n\}}$? The MC Jr. challenges you to find an efficient method to compute these numbers – famuously known as Dedekind numbers (sequence A000372 in the OEIS) – for arbitrary $n$. But beware: no closed formula and no value for $n > 8$ are known to this day.

There is also an interesting connection between antichains and chain decompositions: Given a set of sets $C$, we say that $C$ is a *chain* if and only if $s, t$ are comparable for any distinct $s, t \in C$. A *chain decomposition* $\mathcal{D}$ of some powerset $2^S$ is a set of disjoint chains of $2^S$ such that $\bigcup_{C \in \mathcal{D}} C = 2^S$. Dilworth's theorem now states that the size of the smallest chain decomposition of $2^{\{1,\dots,n\}}$ equals the size of the maximum antichain over $2^{\{1,\dots,n\}}$. For example, for $n = 2$, we have a smallest chain decomposition $\mathcal{D} = \{\{\emptyset, \{1\}, \{1,2\}\}, \{2\}\}$ and a maximum antichain $A = \{\{1\}, \{2\}\}$.

> *God exists since mathematics is consistent, and the Devil exists since we cannot prove it.*
>
> — André Weil

You might want to check out Gödel's Incompleteness Theorems.

[2] *Note:* for $n < 1$, we have $\{1, \dots, n\} = \emptyset$.