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
10.01.2020
Deadline: 21.01.2020, 23:59

Functional Programming and Verification

Sheet 12

Tutorial Exercises

Exercise T12.1 Abstract Data Types: Maps

Note: Please use the templates `AssocList.hs` and `AssocListTests.hs` provided on moodle for this exercise.

You have already seen association lists `Eq k => [(k,v)]` as a way to represent maps with keys k und values v. In order to prevent user from creating invalid association lists (e.g. containing multiple values for some key), we want to hide the implementation in a module.

1. Define a module `AssocList` that only exports a type `Map k v` and the following functions:

```
newtype Map k v = ...
empty :: Map k v
insert :: Eq k => k -> v -> Map k v -> Map k v
lookup :: Eq k => k -> Map k v -> Maybe v
delete :: Eq k => k -> Map k v -> Map k v
keys :: Map k v -> [k]
```

   Calling `insert` with an existing key should replace the associated value. Internally, the maps should be represented using association lists.

   Note: Prelude also exports a function `lookup`. To prevent naming conflicts, you can hide this import using `import Prelude hiding (lookup)`.

2. Define a function `invar :: Eq k => Map k v -> Bool` in `AssocList` that checks whether the map does not contain duplicate keys. Then define QuickCheck properties in a separate module that check whether `invar` is invariant under all functions returning a map as discussed in the lecture (slide 340).

   Note: To check your properties, say `prop_invarInsert`, you need to explicitly tell QuickCheck the types of the values to generate. For example:

```
quickCheck (prop_invarInsert :: Int -> String ->
                              AL.Map Int String -> Property)
```

3. We next check if our implementation (imported as `AL.Map`) behaves correctly when compared to the `Map` datatype provided by `Data.Map` from the package `containers` (imported as `DM.Map`). Define a function `hom :: Ord k => AL.Map k v -> DM.Map k v` that transforms our maps to the one provided by the containers library. Then check whether AL.Map simulates DM.Map by defining QuickCheck properties for every function in `AssocList` as discussed in the lecture (slide 340).

Exercise T12.2 Substitute Teacher

We consider a simple programming language, namely λ-calculus. It is the basis for most functional programming languages including Haskell. A program in λ-calculus is built up from terms, which are either

- just a variable, e.g. x,
- an anonymous function definition $(\lambda x.\ T)$, which binds a variable x in term T, or
- a function application $T\ U$ where both T and U are terms themselves.

Note that bound variable names are interchangeable whereas this is not the case for free variables. For example, the terms $(\lambda x.\ x\ y)$ and $(\lambda z.\ z\ y)$ are equal while the terms $(\lambda x.\ x\ y)$ and $(\lambda x.\ x\ z)$ are not equal. Start by defining a datatype `Term` in Haskell that models the λ-calculus using Strings for variable names. On this datatype, implement the following functions:

1. Instantiate `Show` for `Term` by representing variables as just their name, λ-abstractions as `(\x -> T)` like in Haskell and use spaces for function application. *Bonus*: Omit unnecessary parenthesis.
2. Define `freeVars :: Term -> [String]` which collects all free variables in a term, i.e. variables that are not bound by an enclosing λ-abstraction.
3. Implement a function `substVar :: String -> Term -> Term -> Term` which, when called with `substVar x r t`, replaces all free occurrences of `x` in `t` by the term `r`. *Important:* the function `substVar` makes a key assumption about the terms `t` and `r`. To find out what the assumption is, think about what happens when substituting x with the variable y in the equivalent terms $(\lambda y.\ x\ y)$ and $(\lambda z.\ x\ z)$.

Homework

You need to collect 3 out of 4 points (P) to pass this sheet.

Exercise H12.1 Abstract Data Types: Vector [1P]

In this exercise you will model *Vectors*, which are essentially resizable arrays. By convention, we will index the cells of our vectors beginning with 0.

Define a module `Vector` that only exports a type `Vector a` and the following functions:

```
newtype Vector a = ...
newVector :: Int -> Vector a
size :: Vector a -> Int
capacity :: Vector a -> Int
resize :: Vector a -> Int -> Vector a
set :: Vector a -> a -> Int -> Maybe (Vector a)
get :: Vector a -> Int -> Maybe a
```

`newVector n` should return an empty vector of size n. `size` returns the size of the vector, whereas `capacity` returns the number of empty cells in the vector.

`resize` changes the size of the vector by either adding new empty cells or by truncating the cells with the highest indices.

`set v x i` should set the cell at index `i` to `x`. If `i` is not a valid index, the function should return `Nothing`.

`get v i` returns the element in cell `i`. If that cell is empty or if `i` is not a valid index, it should return `Nothing`.

Exercise H12.2 e 3FW< PV [1P]

One gloomy night, the MC Jr realised that valuable information on his laptop (cat pictures) is not encrypted. He hence decides to take action and remembers that some professor once taught him how to design proper encryption schemes. However, all the MC Jr remembers is that, yet again, it has something to do prime numbers. Even though he is well aware of the staggering value of his cat pictures and that all hand-crafted encryption schemes are doomed to failure, he still decides to roll his own.

1. (**Competition**) Write a function `encrypt :: String -> String` that encrypts a string in the following way: Let $c_1 \cdots c_n = s$ be the characters of s and $p_1, \ldots, p_j$ be all prime numbers smaller or equal than n. Then `encrypt s` will remove the characters at positions $p_1, \ldots, p_j$ from s and prepend them to the resulting string. For example:

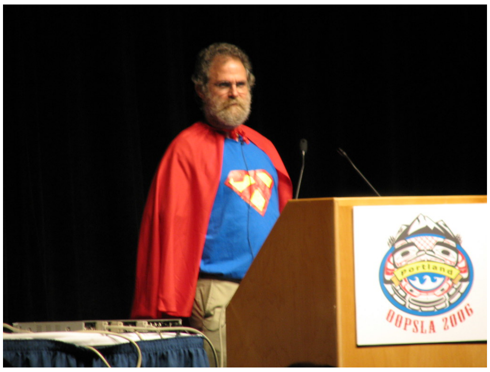
```
encrypt "Hallo" = "aloHl"
encrypt "unbreakable" = "nbekeuraabl"
encrypt "never roll your own encryption scheme" =
  "evrr oonco ene ollyur w enryptinschem"
```

   Then write a function `decrypt :: String -> String` that decrypts a string.

   For the competition, the MC Jr asks you to code-golf both the encryption and decryption function until they look more undecipherable than any word encrypted by the encryption scheme; the fewer tokens, the better. You can use the provided `primes` and `isPrime` functions from the template without including them in the `{-WETT-}...{-TTEW-}` tags. oo uGdlck!

> **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.

2. Write a function `main :: IO ()` that reads a filename `f` and an action string from stdin (on separate lines). If the action string is “encrypt”, the function should encrypt the



Philipp Wadler (designer of Haskell) in his legendary lambda calculus superman costume

contents of the file and write the encrypted version to the file `f.encrypt` (where `f` is the original filename). Otherwise it should decrypt the file, writing to `f.decrypt`.

Exercise H12.3 Capture Me If You Can! [1: 1P, 2: 1P]

Implement the following functionality for the datatype `Term` of λ-terms:

1. In the tutorial exercise we saw that two terms are equivalent if we can rename the bound variables of the former such that we obtain the latter term. For example, we can rename x to y in $(\lambda x.\ x\ z)$ to obtain equivalent term $(\lambda y.\ y\ z)$. This is called α-equivalence. Instantiate `Eq` for `Term` such that `(==)` accounts for α-equivalence.
2. Implement a function `betaRed :: Term -> Term` that performs β-reduction on a lambda term of the form $(\lambda x.\ T)\ U$, i.e. `betaRed` produces the term T with all free occurences of x in T replaced by U (free occurences of x in T are bound by the enclosing λ). As an example, $(\lambda x.\ x\ z)\ (x\ y)$ reduces to $(x\ y)\ z$. *Note:* you can use `substVar` from the tutorial but you may need to rename bound variables beforehand, e.g. $(\lambda x.\lambda y.\ x\ y)\ y$ should result in a term that is α-equivalent to $(\lambda a.\ y\ a)$.

> Nenn es dann, wie du willst,
> Nenn's Glück! Herz! Liebe! Gott
> Ich habe keinen Namen
> Dafür! Gefühl ist alles;
> Name ist Schall und Rauch,
> Umnebelnd Himmelsglut.
>
> — Goethe's Faust