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
29.11.2019
Deadline: 09.12.2019, 23:59

Functional Programming and Verification

Sheet 7

IMPORTANT: We use the cyp ("Check your proof") format for our proofs. Use the templates from the submission website to check your proofs by structural induction. The submission system can check these proofs against the provided templates. Proofs by computation induction cannot automatically be checked by the system but still have to be written in the cyp format.

Each proof step must use *one* of these defining equations, the inductive hypothesis (IH), or an axiom. You have to state the justification for each step.

Tutorial Exercises

Exercise T7.1 Ultrafilter

Prove the proposition

```
filter p . filter p = filter p
```

where `filter :: (a -> bool) -> [a] -> [a]` and `(.) :: (b -> c) -> (a -> b) -> a -> c` are defined as

```
filter f [] = []
filter f (x : xs) = if f x then x : filter f xs else filter f xs

(f . g) x = f (g x)
```

You may also use the following axioms about if-expressions:

```
axiom if_True : (if True  then x else y) .=. x
axiom if_False: (if False then x else y) .=. y
```

Exercise T7.2

1. Write a function `iterWhile :: (a -> a -> Bool) -> (a -> a) -> a -> a` such that `iterWhile test f x` iterates `f` until `test x (f x)` is false, and then returns `x`.
2. Use `iterWhile` to implement a function `fixpoint :: Eq a => (a -> a) -> a -> a` that iterates a function `f` until it finds a value `x` such that `f x = x` and then returns this value.

   Bonus: Use `fixpoint` to implement a square root function.

3. Use `iterWhile` to implement a function `findSup :: Ord a => (a -> a) -> a -> a -> a` such that `findSup f m x` finds the largest value $f^n x$ that is at most `m` assuming that f is strictly monotonically increasing.

Exercise T7.3 Mapidi Map

You have already seen the function `map`, which applies a function to each element of a list, in the lecture. In this exercise you will implement a stronger version of this function, which also has access to the already processed part of the list.

Concretely, the function

```
mapState :: (s -> a -> (b,s)) -> s -> [a] -> ([b], s)
```

takes a function f, an initial state s and a list xs. Here, f is applied to a state and an element of the list and returns the processed element as well as a new state. Your `mapState` is supposed to apply f to each element in the list, updating the state at each step.

1. Implement `mapState`

   Examples:

   ```
   inc s x = ((s, x), s + 1) -- Auxiliary function

   mapState inc 0 [] = ([], 0)
   mapState inc 0 ['a'] = ([(0, 'a')], 1)
   mapState inc 0 "abc" = ([(0, 'a'), (1, 'b'), (2, 'c')], 3)
   ```

2. Implement the regular `map` function using `mapState`.
3. Write a function `f :: String -> Char -> (Char, String)` such that `mapState f [] xs` capitalizes the first occurrence of any letter in the string `xs`.

   Examples:

   ```
   fst (mapState f [] "abrakadabra") == "ABRaKaDabra"
   fst (mapState f [] "ottos mops kotzt") == "OTtoS MoPs KotZt"
   ```

   Use `toUpper` from `Data.Char` to get the uppercase version of a character.

Exercise T7.4 Everything Is A Fold

Using `foldr`, implement the following functions:

1. `compose :: [(a -> a)] -> a -> a` such that compose $[f_1, \dots, f_n] = f_1(\dots(f_n(\cdot)\dots)$.
2. `fib :: Integer -> Integer` computing the Fibonacci numbers.
3. `length' :: [a] -> Integer` computing the length of a list.
4. `reverse' :: [a] -> [a]` reversing a list.
5. `map' :: (a -> b) -> [a] -> [b]` mapping a function over a list.
6. `inits' :: [a] -> [[a]]` computing the prefixes of list.

Homework

IMPORTANT: The submission system can process multiple file at once. You have to upload all files (h71.cprf and Exercise_7.hs) at once. Moreover, you have to use the provided filenames when uploading, that is do not change the filenames of the template files.

You need to collect 4 out of 5 points (P) to pass this sheet. The needed background theories/definitions (*.cthy files) for the following exercises can be found on moodle.

Exercise H7.1 Left, right, left, right, ... [2P]

This exercise is all about the two different fold functions `foldl` and `foldr`, which are defined as follows:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs

foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

The function signatures are similar; however, there is a key difference in their functionality: As the names suggest, `foldl` performs a left-associative and `foldr` a right-associative fold, respectively. More concretely, we have that

```
foldl f z [x1, x2, ..., xn] = (...((z `f` x1) `f` x2) `f` ...) `f` xn

foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ... (xn `f` z)...)
```

Let `f` be a binary operator that is commutative with respect to `a`, i.e. `f x a = f a x` for all `x`, and associative. Prove the statement `Lemma: foldl f a .=. foldr f a`.

Exercise H7.2 Grokking The Glob [2P]

Implement a function `matchesPath :: String -> String -> Bool` that takes a path, which may contain glob[1] patterns, and determines whether it matches the second path, which is an absolute path without any glob patterns. The components of a path are separated by slashes (`/`) and never contain the characters of the glob patterns or a slash. You don't need to consider relative paths; that is, the components of a path are never "`.`" or "`..`". Files are referred to without a slash at the end while directories always end with a slash, e.g. `/home/test` is a file `test` in the directory `/home` whereas `/home/test/` is a directory in `/home`.

[1] Glob patterns on Wikipedia

We restrict ourselves to a subset of the glob patterns: `*`, `?`, `**`, and the pattern `{a,b,c}`, which matches one of the paths `a`, `b`, or `c`. The pattern `*` matches any number of characters and `?` matches one character. The pattern `{...}` may also contain relative paths, e.g. `/home/{a,b/out/*.zip}` matches `/home/a` and `/home/b/out/d.zip`. Finally, if a component only consists of n instances of the pattern `*`, where $n \geq 2$, then this matches at least one path component up to an arbitrary number of components. For example, `/home/**/test` matches `/home/a/test` and `/home/b/c/test` but it does not match `/home/test`.

Note: There will be public and hidden tests for this exercise. Creating a lookup table for the public tests will result in zero points once your tutor is grading the exercise.

Exercise H7.3 Transitive Closure [1P]

From — possibly traumatic — experiences in previous semesters you are familiar with *relations*, which we can represent as lists of tuples `[(a,b)]` in Haskell. In this exercise we will work with relations where each pair is annotated with an integer, which could e.g. represent a distance between two elements. Concretely, we will represent relations as `[(Integer,(a,b))]`.

- Implement a function `comp :: [(Integer,(a,b))] -> [(Integer,(b,c))] -> [(Integer,(a,c))]` that composes two relations while taking the sum of the labels.
- Implement a function

  ```
  symcl :: Eq a => [(Integer,(a,a))] -> [(Integer,(a,a))]
  ```

  that computes the symmetric closure of a relation. That is, for any pair `(a,b)` that is included in the input relation, the output relation should also contain `(b,a)`. If there are multiple identical pairs with different weights, only the pair with the lowest weight should be included in the output.
- **(Wettbewerb)** Implement a function

  ```
  trancl :: Eq a => [(Integer,(a,a))] -> [(Integer,(a,a))]
  ```

  that computes the transitive closure of a relation. For any a this relation should contain all pairs `(a,b)` where a is related to b through transitivity. The weight of these pairs should be the smallest sum of weights through which b can be reached from a.

  For example:

  ```
  trancl [(10,(1,3)),(3,(1,2)),(5,(2,3)),(2,(2,4))]
  ```

  should return

  ```
  [(2,(2,4)),(3,(1,2)),(5,(2,3)),(5,(1,4)),(8,(1,3))]
  ```

  Notice that the weight of the pair `(1,3)` is 8, because it can be reached as `(1,2) -> (2,3)`, rather than directly from 1.

  For the *Wettbewerb* we are asking you to optimize your implementation of `trancl` for number of tokens. Make sure to include anything that is used for `trancl`, e.g. the `comp` function should you choose to use it, inside the `{-WETT-}` tag.

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.

It's called curried because it's a bit spicy.

— Thorsten Altenkirch