Functional Programming and Verification

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler — WS 2018/19

Exercise Sheet 10 — Deadline: 13.01.2019

General Information

Detailed information about the lecture, tutorials and homework assignments can be found on the lecture website[1]. Solutions have to be submitted to Moodle[2]. Upload your solutions as a single file named 'hw10.ml' to moodle. Since submissions are tested automatically, solutions that do not compile or do not terminate within a given time frame cannot be graded and thus result in 0 points. If you do not manage to get all of your implementations compiling and/or terminating, comment them out and use the original definition (if you comment out the definition, the tests won't compile)! Use Piazza[3] to ask questions and discuss with your fellow students.

Automatic Testing

Testing your solutions is more difficult on this sheet as tests do not compile until you get most of your modules correct and some implementation details cannot be tested automatically. Thus, all tests are commented and you should enable them only when you have implemented the respective module. Consider the tests on this sheet as a hint, so passing tests is no guarantee that your implementation is as intended, but they will give you a basic guideline for your implementations. The grading of your solution will happen semi-automatically, so your tutors will have a look at your submission.

Assignment 10.1 (L) Map

We model sets of values using modules with signature

```
module type Set = sig
  type t
  val to_string : t -> string
end
```

Furthermore, we define a signature for maps (mappings from keys to values):

```
module type Map = sig
  type key
  type value
  type t
  val empty : t
  val set : key -> value -> t -> t
  val get : key -> t -> value
  val get_opt : key -> t -> value option
  val to_string : t -> string
end
```

[1]https://www.in.tum.de/i02/lehre/wintersemester-1819/vorlesungen/functional-programming-and-verification/

[2]https://www.moodle.tum.de/course/view.php?id=44932

[3]https://piazza.com/tum.de/fall2018/in0003/home

Where the functions have the following semantics:

- `set` updates the mapping such that key is now mapped to value.
- `get` retrieves the value for the given key and throws a `Not_found` exception if no such key exists in the map.
- `get_opt` retrieves the value for the given key or `None` if the key does not exist.
- `to_string` produces a string representation for the mapping, e.g.:

```
"{ 1 -> \"x\", 5 -> \"y\" }"
```

Perform these tasks:

1. Implement a module `StringSet` of signature `Set` to model sets of strings.
2. Define a signature `OrderedSet` that extends the `Set` signature by a `compare` function with the usual type.
3. Implement a functor `BTreeMap` that realizes the `Map` signature and uses a binary tree to store *key-value*-pairs. The functor takes key and value sets as arguments.
4. Implement a `StringSet` and an ordered `IntSet` module.
5. Use the `BTreeMap` functor to define modules for *int-to-string* and *int-to-int* maps.
6. Insert some values into an *int-to-string* map and print its string representation.

Hint: If a module's signature is too abstract to be used with concrete values, make use of sharing constraints[4].

[4] `https://v1.realworldocaml.org/v1/en/html/functors.html`

Assignment 10.2 (H) Matrix [20 Points]

We define the signature

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mul : t -> t -> t
  val compare : t -> t -> int
  val to_string : t -> string
end
```

of an algebraic ring structure, where `add` and `mul` are the two binary operations on the set of elements of type `t`. Values `zero` and `one` represent the identity elements regarding *addition* and *multiplication*. The function `compare` establishes a total order on the elements of the set. The auxiliary function `to_string` is used to generate a string representation of an element.

Furthermore, a matrix is given by the signature:

```
module type Matrix = sig
  type elem
  type t
  val create : int -> int -> t
  val identity : int -> t
  val from_rows : elem list list -> t
  val set : int -> int -> elem -> t -> t
  val get : int -> int -> t -> elem
  val transpose : t -> t
  val add : t -> t -> t
  val mul : t -> t -> t
  val to_string : t -> string
end
```

Here, `elem` and `t` represent the types of the elements of the matrix and the matrix itself, respectively. The functions have the following semantics:

- `create n m` creates an empty (all zeroes) $n \times m$ matrix.
- `identity n` creates an $n \times n$ identity matrix.
- `from_rows l` creates a matrix from the given list of rows (lists of elements). You may assume that `l` is non-empty and all lists have the same length.
- `set r c v m` sets the element at row `r` and column `c` in matrix `m` to value `v`.
- `get r c m` returns the element at row `r` and column `c` from matrix `m`.
- `transpose m` returns the transpose of matrix `m`.
- `add a b` adds matrices `a` and `b` component-wise.
- `mul a b` computes the matrix multiplication of `a` and `b`.

- `to_string m` produces a string representation of `m`. Columns shall be separated by single whitespaces and rows by a newline character \n.

Perform these tasks:

1. Implement a module `IntRing` that implements the `Ring` signature for the `int` type. [1 point]

2. Implement a module `FloatRing` that implements the `Ring` signature for the `float` type. [1 point]

3. Define a signature `FiniteRing` that extends the `Ring` signature with a value `elems` that represents a list of all elements of the ring's finite set. Make sure that everything in the `Ring` signature is part of `FiniteRing` (without copy-pasting them)! [1 point]

4. Implement a module `BoolRing` that implements the `FiniteRing` signature for the `bool` type. [1 point]

5. Implement a functor `SetRing` that models a ring over the power set of the set in the `FiniteRing` passed as the functor's argument. `SetRing` has to implement the `Ring` signature. We use union and intersection as `add` and `mul` operations. The representation produced by `to_string` is "$\{e_1, \ldots, e_n\}$". For `compare` we use a bit of an unconventional order. First, we order the elements in the set by their own order and then compare the sets lexicographically, e.g. $\{\} < \{1, 2, 3\} < \{2\} < \{2, 3\} < \{3\}$. [3 points]

6. Implement a functor `DenseMatrix` that satisfies the `Matrix` signature. The argument of the functor is a `Ring` that provides everything to implement the matrix operations. The `DenseMatrix` is supposed to store all elements in the matrix in a list of rows, which are lists of elements. [6 points]

7. Implement a functor `SparseMatrix` that satisfies the `Matrix` signature. The argument of the functor is a `Ring` that provides everything to implement the matrix operations. The `SparseMatrix` stores only those elements of the matrix that are non-zero. [7 points]

Hint: You may assume that all inputs are valid, e.g. no out-of-bounds access. If you wish to handle invalid input, just throw an exception.

Hint: The function implementations need not be particularly efficient or tail-recursive, just keep your implementations simple where possible.