185.A05 Advanced Functional Programming SS 21

Monday, 19 April 2021

Assignment 4

on Chapter 9 thru 13, and 5

Topics: Type Classes, Higher-order Type Classes, and Testing

Submission deadline: Monday, 10 May 2021, 12am (extended by two weeks!)

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 0. If you have any questions regarding them, ask your questions in the TISS forum. Following these instructions is essential to ensure a smooth processing of your submitted file with the test system.

2. Store all functions to be written for this assignment in a top-level file named

   `Assignment4.hs`

   of your group directory. Take advantage of the template file `Assignment4.hs` and extend it as required. Comment your program meaningfully, and use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 4.

3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment4.hs`. Import declarations for self-defined modules will fail: Only `Assignment4.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming assignments:

Type classes and higher-order type classes. Sets are collections of pairwise disjoint items without any particular order. In this assignment we consider pseudo-heterogeneous sets defined in terms of values of a new list type:

```
data Elem     = I Int | S String | C Char | B Bool deriving Eq
data Set a    = Empty | a :> (Set a)
type PreSet a = Set a
```

```
make_set :: Eq a => (PreSet a) -> (Set a)
make_set = remove_duplicates

remove_duplicates :: Eq a => (PreSet a) -> (Set a)
remove_duplicates ...
```

Complete the implementation of `remov_duplicates` and make the types and type constructors `Set Elem` and `Set` instances of the following type and higher-order type classes:

1. `remove_duplicates`: Complete the implementation as its name suggests.
2. `Eq`: Two sets s_1 and s_2 are equal iff every element of s_1 is also an element of s_2 and vice versa.
3. `Show`: The empty set `Empty` shall be displayed as `"{}"`. Non empty sets like `(I 42 :> (B True :> (S "Fun" :> (C 'a' :> Empty))))` shall be displayed as `"{42,True,"Fun",'a'}"`. This means, constructors of `Elem` values shall be dropped, the set constructor `(:>)` be replaced by colon, and curly brackets added at the beginning and end.
4. `Monoid`: The meaning of `mappend` shall be given by set union, the meaning of `mempty` and `mconcat` shall be analoguous to their list monoid counterparts adjusted for sets.
5. `Functor`: The meaning of `fmap` shall be analoguous to its list functor counterpart adjusted for sets (i.e., for sets and injective map arguments, `fmap` yields sets, and pre-sets (typically) otherwise).
6. `Applicative`: The meaning of `pure` and `(<*>)` shall be analoguous to their list applicative counterparts adjusted for sets.
7. `Monad`: The meaning of `(>>=)`, `(>>)`, `(return)`, and `failure` shall be analoguous to their list monad counterparts adjusted for sets (i.e. monadic operations shall yield sets, not pre-sets, where applicable).
8. **Without submission:** Prove or refute the validity of the monoid, functor, applicative, and monad laws of your instances.

All member functions of type and higher-order type classes will only be tested for proper sets, not pre-sets.

Consider now a different set type:

```
data Set' a d e f    = Empty'
                     | a 'A' (Set' a d e f)
                     | d 'D' (Set' a d e f)
                     | e 'E' (Set' a d e f)
                     | f 'F' (Set' a d e f)
type PreSet' a d e f = Set' a d e f
```

```
make_set' :: (Eq a,Eq d,Eq e,Eq f) => (PreSet' a d e f) -> (Set' a d e f)
make_set' = remove_duplicates'

remove_duplicates' :: (Eq a,Eq d,Eq e,Eq f) => (PreSet' a d e f) -> (Set' a d e f)
remove_duplicates' ...
```

Complete the implementation of `remov_duplicates'` and make the types and type constructors `Set a d e f` and `Set Int String` instances of the following type and higher-order type classes (using the contexts `(Eq a, Eq d, Eq e, Eq f) =>`, `(Show a, Show d, Show e, Show f) =>`, where applicable):

9. `remove_duplicates'`: Analogously to exercise 1.
10. `Eq`: Analogously to exercise 2.
11. `Show`: Analogously to exercise 3.
12. **Without submission:**
    - 12.1 What other set operations could have been used for implementing `mappend` of type class `Monoid`?
    - 12.2 Can `(Set' Int String)` be made an arrow? Straightforward implementations of `(>>>)` and `first` seem obvious; an implementation of `pure` seems less obvious. Are there some? Are the final instances meaningful instances of `Arrow`? Compare them with the arrow examples of Chapter 13. What is similar? What is different?
    - 12.3 What is about the other types and (partially evaluated) type constructors? Can
      - 12.3.1 `Set` be made an arrow?
      - 12.3.2 `Set'`, `(Set' a)`, `(Set' a d)`, `(Set' a d e)`, `(Set' a d e f)` be made members of `Eq`, `Show`, `Monoid`, `Functor`, `Applicative`, and `Monad`?

    Explain your reasoning.

All member functions of type classes will only be tested for proper sets, not pre-sets.

Testing. Having proved or refuted the validity of the laws of the (higher-order) type classes for sets in exercise 8, we next want to double-check if some of these results can also be validated by automatic testing as supported by the QuickCheck library.

To this end, implement a generator for pre-sets of type `PreSet` (not `PreSet'`). Make sure that the size of generated pre-sets is about 5 on average. Use `make_set` to transform generated pre-sets into sets, and use these sets as test inputs for the following properties challenging the validity of (some of) the type class laws.

13. Pre-set and set properties, generator properties
    - 13.1 `prop_1 :: (PreSet Elem) -> Property` tests that the pre-set generator alone yields only occasionally (proper) sets.

13.2 `prop_2 :: (PreSet Elem) -> Property` tests that the pre-set generator in combination with `make_set` yields (proper) sets.

13.3 `prop_3 :: (PreSet Elem)-> Property` tests that removing duplicates in a list twice instead of once leaves a set invariant, i.e. *remove_duplicates . remove_duplicates* equals *remove_duplicates*.

14. Reporting features of QuickCheck

Use QuickCheck's reporting features to collect more detailed information about the test data generated and used. To this end extend the implementation of property `prop_2` using the QuickCheck combinators `trivial`, `classify`, and `collect`, respectively (note, these combinators might be renamed in recent QuickCheck versions). Using

14.1 `trivial`, `prop_2a` shall report the percentage of *trivial* test inputs. As *trivial* we consider test inputs with up to 1 element. A possible report could thus be:

```
OK, passed 100 tests (45% trivial).
```

14.2 `classify`, `prop_2b` shall report the percentages of test inputs with up to 2 elements, 3 to 5 elements, and 6 or more elements. A possible report could thus be:

```
OK, passed 100 tests.
46% of test inputs with up to two elements.
38% of test inputs with three to five elements.
16% of test inputs with six or more elements.
```

14.3 `collect`, `prop_2c` shall report the percentages of all test inputs, i.e., the histogram of test inputs. An excerpt of a possible report, where the numbers following the per centage symbol denote the number of elements of the test input:

```
OK, passed 100 tests.
24% 0.
18% 1.
12% 2.
10% 3.
...
1% 12.
...
```

15. Monoid laws and properties

15.1 `prop_monol1 :: (Set Elem) -> Property` tests the validity of monoid law *MonoL1*.

15.2 `prop_monol2 :: (Set Elem) -> Property` tests the validity of monoid law *MonoL2*.

15.3 `prop_monol3 :: (Set Elem) -> (Set Elem) -> (Set Elem) -> Property` tests the validity of monoid law *MonoL3*.

15.4 `prop_mappend :: (Set Elem) -> (Set Elem) -> Property` tests if the sum of the number of elements of the two arguments of `mappend` equals the number of elements of the set yielded by applying `mappend` to them.

16. **Without submission:** Is QuickCheck powerful enough to test the validity of functor, applicative, monad, and arrow laws, too? E.g., what would be the type of properties testing the validity of functor law

16.1 *FL1*: `prop_fl1 :: ... -> Property`?

16.2 *FL2*: `prop_fl2 :: ... -> Property`?

Explain your reasoning.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller