Lab 8

Functional Programming (ITI0212)

2023-03-24

This week we are learning about algebraic interfaces. These are interfaces whose implementations are expected to satisfy certain equations. For example, we expect the method `(==) : Eq a => a -> a -> Bool` to be an *equivalence relation* (i.e., reflexive, symmetric, and transitive).

We met two new interfaces on types. A *semigroup* is a type `a` with an associative combining operation `(<+>) : Semigroup a => a -> a -> a`. If this combining operation has a *neutral element* then the semigroup is a *monoid*. Monoids are useful because they let us combine any finite sequence of things into a single thing.

We also met three new interfaces on type constructors. A *functor* is a type constructor `t` that allows us to map a function over it using the method `map : Functor t => (a -> b) -> t a -> t b`. The functor laws say that mapping must respect the composition structure of functions. A functor is *applicative* if it has methods `pure : Applicative t => a -> t a` and `(<*>) : Applicative t => t (a -> b) -> t a -> t b` that satisfy sensible laws. A *monad* is an applicative functor with the interdefinable methods `(>>=) : Monad t => t a -> (a -> t b) -> t b` and `join : Monad t => t (t a) -> t a` that behave reasonably. Because `do`-notation is syntactic sugar for `(>>=)`, we can use it not just for `IO`, but for any monad.

Task 1
Write down some properties that you expect implementations of the `Ord` interface to satisfy.

Task 2
Confirm for yourself that the exclusive-or operation (see lab 2, task 1) is associative. Then write a semigroup implementation for the booleans, where the combining operation is exclusive-or.

```
implementation  Semigroup Bool  where
```

Extend this to a monoid implementation.

```
implementation  Monoid Bool  where
```

Task 3
Write a semigroup implementation for the type of *endomorphisms* on an arbitrary type, where the combining operation is function composition.

```
implementation  Semigroup (a -> a)  where
```

Extend this to a monoid implementation.

```
implementation  Monoid (a -> a)  where
```

so that, for example:

```
Lab8> ( * 2) <+> ( + 1) $ 3
7
Lab8> ( + 1) <+> neutral <+> ( * 2) $ 3
8
```

Task 4
Write a function that combines a monoid element with itself a given number of times:

```
multiply  :  Monoid a => Nat -> a -> a
```

For example:

```
Lab8> multiply 3 "hello"
"hellohellohello"
Lab8> multiply 3 [1, 2]
[1, 2, 1, 2, 1, 2]
Lab8> multiply 3 True
True
Lab8> multiply 4 True
False
Lab8> multiply 3 ( * 2) 5
40
```

Task 5
Use pattern-matching and structural recursion to write the following function that returns `Just` a list of things just in case all of the argument list elements are `Just` things.

```
consolidate  :  List (Maybe a) -> Maybe (List a)
```

For example:

```
Lab8> consolidate [Just 1, Just 2, Just 3]
Just [1, 2, 3]
Lab8> consolidate [Just 1, Nothing, Just 3]
Nothing
Lab8> consolidate []
Just []
```

Task 6
Now analyze the definition that you wrote in task 5 and try to rewrite it as `consolidate'` using the fact that `Maybe` is a `Functor`. This should allow you to avoid any case analysis in the recursive clause (the base-case clauses will remain unchanged). If you need a hint, refer to `Lecture8.update'`.

Task 7
Recall that in lecture 8 we wrote the arity 2 mapping function for applicative functor types:

```
map2  :  Applicative t => (a -> b -> c) -> t a -> t b -> t c
```

Write the arity 1 mapping function for applicative functor types:

```
map1  :  Applicative t => (a -> b) -> t a -> t b
```

Your definition of `map1 f x` should be an expression involving only `f`, `x`, `pure`, and `<*>`.

Optional challenge: Write `map0` and `map3`, and try to identify the general pattern to `map`n.

Task 8
Recall that `List` is a `Monad` and therefore implements the `join` method. Define this function yourself as:

```
join_list  :  List (List a) -> List a
```

so that `join_list xss` behaves like `join xss` for any `xss : List (List a)`.