Lab 8

Functional Programming

2024-04-05

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

We met two new interfaces for 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 for type constructors. A *functor* is a type constructor `t : Type -> Type` 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 `join : Monad t => t (t a) -> t a` and `(>>=) : Monad t => t a -> (a -> t b) -> t b` that behave reasonably. Because `do`-notation is syntactic sugar for `(>>=)`, we can use it not only 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) 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
An *endomorphism* is a function from a type to itself. Write a semigroup implementation for the type of endomorphisms on an arbitrary type:

```
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 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 []
```

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

Task 6
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 `<*>`.

Write the arity 0 mapping function for applicative functor types:

```
map0  :  Applicative t => a -> t a
```

Your definition of `map0 x` should be an expression involving only `x`, `pure`, and `<*>`.

Challenge: Write the type and definition for `map3`, and try to identify the general pattern for `map`*n*.

Task 7
Try to guess the value of each of the following expressions; then ask Idris to evaluate them to see if your prediction was correct:

- `the (List _) $ map0 3`
- ``the (List _) $ map1 ( `mod` 2 == 0) [1, 2, 3]``
- `the (List _) $ map2 MkPair  [1,2,3] ['a','b','c']`

Describe in words what `map2` does for the applicative functor `List`.

Task 8
Write a higher-order function that uses a given function to transform the element at the specified index of a list:

```
transform  :  (f : a -> a) -> (index : Nat) -> List a -> List a
```

If the index is out-of-bounds for the list then your function should behave like the identity function. For example:

```
Lab8> transform S 0 [1, 2]
[2, 2]
Lab8> transform S 1 [1, 2]
[1, 3]
Lab8> transform S 3 [1, 2]
[1, 2]
```

Now import `Data.String` and use your `transform` function, together with the following standard library functions (`:doc` them!),

- `words : String -> List String`,
- `unwords : List String -> String`,
- `unpack : String -> List Char`,
- `pack : List Char -> String`,
- `toUpper : Char -> Char`.

to write a function that capitalizes the first letter of each word in a string:

```
titlecase  :  String -> String
```

For example:

```
Lab8> titlecase "it was the best of times it was the worst of times."
"It Was The Best Of Times It Was The Worst Of Times."
```

Note: You can (and should!) write this function as a point-free one-liner, using the fact that `List` is a `Functor`. Here is a hint to get you started:

```
titlecase  =  unwords . ?goal . words
```