Lab 8

Functional Programming (ITI0212)

2021.03.16

Gaussian Integers

Recall the type of Gaussian integers from lecture 8:

```
data GaussianInteger : Type where
  Gauss  :  Integer -> Integer -> GaussianInteger
```

We saw in lecture how to define a `Num` instance for this type so that we could add and multiply them. There is another numeric interface extending `Num` called `Neg`, with a single method called `negate` representing the unary minus ($x \mapsto -x$). Subtraction is implemented using this interface by defining `x - y` as `x + (negate y)`.

Task 1
Write a `Neg` instance for the type of Gaussian integers:

```
implementation  Neg GaussianInteger  where
```

With this you should be able to do things like:

```
> -(Gauss 1 2)
Gauss -1 -2
> (Gauss 1 2) + - (Gauss 3 4)
Gauss -2 -2
```

note: I think that you should be able to use the binary infix subtraction operator `(-)` too, but it doesn't seem to work for me.

Task 2
Write an `Eq` instance for Gaussian integers:

```
implementation  Eq GaussianInteger  where
```

Task 3
Write a named `Ord` instance for Gaussian integers:

```
implementation [lex] Ord GaussianInteger  where
```

which compares them lexicographically:

```
> compare @{lex} (Gauss 1 200) (Gauss 2 1)
LT
> compare @{lex} (Gauss 2 1) (Gauss 2 1)
EQ
> compare @{lex} (Gauss 3 1) (Gauss 2 4)
GT
```

Task 4
Use the `Mag` instance for Gaussian integers defined in lecture to write a named `Ord` instance for Gaussian integers:

```
implementation  [mag] Ord GaussianInteger  where
```

which compares them by magnitude:

```
> compare @{mag} (Gauss 1 200) (Gauss 2 1)
GT
> compare @{mag} (Gauss 2 1) (Gauss 2 1)
EQ
> compare @{mag} (Gauss 3 1) (Gauss 2 4)
LT
```

Comparing Lists

The default `Eq` instance for `Lists` compares them *pointwise*, that is, two lists are considered equal if they have the same elements in the same order:

```
> the (List Nat) [1,2,3] == [3,2,1]
False
> the (List Nat) [1,2,3] == [1,2,3,3]
False
> the (List Nat) [1,2,3] == [1,2,3]
True
```

For the following tasks you will need to import `Data.List`.

Task 5
Write a named `Eq` instance for lists that compares them *setwise*:

```
implementation  [setwise] Eq a => Eq (List a)  where
```

that is, two lists should be considered equal if each element that occurs (at least once) in one of the lists also occurs (at least once) in the other:

```
> (==) @{setwise} [1,2,3] [3,2,1]
True
> (==) @{setwise} [1,2,3] [1,2,3,3]
True
> (==) @{setwise} [1,2,3] [1,2,4]
False
```

hint: the following functions may be useful:

- `elem : Eq a => a -> List a -> Bool`
- `all : (a -> Bool) -> List a -> Bool`

Task 6
Write a named `Eq` instance for lists that compares them *multisetwise*:

```
implementation [multisetwise] Eq a => Eq (List a)  where
```

that is, two lists should be considered equal if each list contains the same number of copies of each element as the other, regardless of order:

```
> (==) @{multisetwise} [1,2,3] [3,2,1]
True
> (==) @{multisetwise} [1,2,3] [1,2,3,3]
False
> (==) @{multisetwise} [1,2,3] [1,2,4]
False
```

hint: the following functions may be useful:

- `elem : Eq a => a -> List a -> Bool`
- `delete : Eq a => a -> List a -> List a`