# Lab 12

Functional Programming (ITI0212)

2022-04-15

## String manipulation with Semigroup and Monoid

Implementing the `Semigroup` interface for a type `a` allows us to “merge” two values of type `a` with the function `(<+>) : a -> a -> a`. If `a` also implements `Monoid`, we have access to a value `neutral : a` that doesn’t influence what it is merged with.

In the tasks below, you will define functions that generalize functions for string-manipulation to arbitrary typess with implementations of `Semigroup` or `Monoid`.

### Task 1

Define a function `repeat` that takes a natural number `n` and a value `x : a`, and repeats `x` `n` times using `(<+>)`. For any argument `x : a`, `repeat 1 x` should evaluate to `x`. Decide yourself whether it is enough to constrain `repeat` to types with a `Semigroup` implementation, or whether `Monoid` is required. *(Hint: What should `repeat 0 x` evaluate to?)*

*Examples:* If the type `a` is the natural numbers, `repeat` should add a natural number `n` times to itself:[1]

```
> repeat 6 (the Nat 7)
42
```

For strings, it should return repetitions of the input:

```
> repeat 3 "na"
"nanana"
```

### Task 2

Write a function `intersperse`, generic over some `Monoid a`, that returns the alternating application of `(<+>)` to a separator `sep : a` and the elements of a list `xs : List a`.

When `a` is `String`, the function should concatenate a list of strings, with a separator interspersed:

```
> intersperse ", " ["A", "comma", "separated", "string"]
"A, comma, separated, string"
```

Note that there is no trailing `sep` in the above output!

*Hint: This function is very similar to the function `concatenate` defined during lecture 12.*

### Task 3

Combine the above functions to reproduce the following string:

[1] Assuming you define `Monoid Nat` as done in the lecture.

```
poetry : String
poetry = """
  Na Na Na Na Na Na Na Na Batman!
  Na Na Na Na Na Na Na Na Batman!
  Batman! Batman! Batman!
  """
```

The above string literal is a multiline string that includes newline characters (`"\n"`).

## A broken `Functor`

Consider the following datatype, which contains a counter and some value:

```
record Counter (a : Type) where
  constructor MkCounter
  counter : Nat
  value : a
```

**Task 4**
This implementation of `Functor Counter` keeps track of how often `map` was called:

```
implementation Functor Counter where
  map func (MkCounter counter value) =
    MkCounter (counter + 1) (func value)
```

Why is it not a valid implementation? Which functor laws does it violate?

## Parallel computation with `Vect n`

**Task 5**
Give an implementation of `Functor` for `Vect n` where `map` applies a function to all elements of the vector.

A value of type `Vect n (a -> b)` can be seen as `n` computations happening in parallel. Implement `Applicative` so that `(<*>)` "executes" these computations in parallel.

*Hint: `n : Nat` must appear as a non-erased parameter, like so:*

```
implementation {n : Nat} -> Applicative (Vect n) where
```

## The `List` monad

**Task 6**
In the lecture, we hinted at an implementation of `Monad` for `List` given by the standard library. Ignore this fact and write an implementation like this yourself:

```
implementation [Mine] Monad List where
```

You may make use of the implementations of `Functor` and `Applicative` for `List` as provided by the standard library.

Functions `a -> List b` can be seen as *non-deterministic* functions, returning many possible values of type `b`. Your definition of `(>>=)` (or `join`) should reflect that. For example, the definition[2]

[2] Ignore the line `let %hint = ...`, this is an unfortunate hack necessary to convince Idris to use the implementation named `Mine` instead of the default implementation.

```
list_example : List String
list_example =
  let %hint m : ?; m = Mine in
  do
    x <- ["foo" , "bar"]
    y <- ["1", "2"]
    pure $ x ++ y
```

should evaluate to

```
Lab12> list_example
["foo1", "foo2", "bar1", "bar2"]
```