Lab 9

Functional Programming (ITI0212)

2020.04.15

This week we learned about Idris's system for managing name overloading. An *interface* is a collection of named type signatures, called "methods", involving a collection of typed bound variables. The interface represents a constraint on these variables. An *implementation* of an interface, called an "instance", provides definitions of a basis for the methods, and represents a solution to the constraint represented by the interface.

Interfaces can themselves depend on other interfaces, and implementations can be either named instances or unnamed *default instances*. There can be at most one default instance of a given interface for each assignment of its bound variables.

Multisets

A **multiset** (or "bag") is a data structure that is like a list, except that the order of the elements is irrelevant. There are several possible ways to represent multisets in Idris, but for our present purposes the easiest one will be to simply encode them as `List`s. Two multisets, when encoded as lists, are *equal* just in case each is a *permutation* of the other. A multiset `xs` is contained in a multiset `ys` just in case each element that occurs with multiplicity m in `xs` occurs with multiplicity n in `ys` with $m \leq n$.

A **preorder** is reflexive and transitive binary relation on a collection of objects. Recall that a binary relation $- \sqsubseteq -$ on a collection of objects A is:

reflexive if for each $x \in A$, we have that $x \sqsubseteq x$, and

transitive if for each $x, y, z \in A$, we have that if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$.

The following interface is meant to specify a preorder structure on a type:

```
interface  Preorder (a : Type)  where
  leq : a -> a -> Bool
```

The method `leq` is intended to represent the relation $- \sqsubseteq -$; i.e., we should have $x \sqsubseteq y$ just in case `leq x y` is `True`.

Task 1
First type the `Preorder` interface into your lab file. Next define a `Preorder` instance for `List`s whose element type is an instance of `Eq` in such a way that `leq xs ys` is `True` just in case `xs` $\sqsubseteq$ `ys` when `xs` and `ys` are interpreted as multisets and $- \sqsubseteq -$ is interpreted as multiset containment.

In other words, complete the definition of the `leq` method in:

```
implementation  Eq a => Preorder (List a)  where
  leq xs ys  =  ?MultisetPreorder
```

This function should behave as follows:

```
leq [] [5]  =  True
leq [2 , 1] [1 , 2]  =  True
leq [1 , 1 , 2] [1 , 2 , 2]  =  False
```

Hint: the standard library functions `Prelude.List.elem` and `Prelude.List.delete` may be useful.

Task 2
Write a named implementation of the `Eq` interface for `Lists` that determines multiset equality; i.e., complete the definition of the `(==)` method in:

```
implementation [Multiset]  Eq a => Eq (List a)  where
  xs == ys  =  ?MultisetEquality
```

so that `[1,2] == [2,1] = False` but `(==) @{Multiset} [1,2] [2,1] = True` [1].

Hint: using task 1, this should be a one-liner.

Applicative Functors

Recall that an *applicative functor* structure lets us lift functions of arbitrary arity into a parameterized type constructor.

Task 3
Complete the definition of the function `consolidate`:

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

such that

```
consolidate [Just 1 , Just 2 , Just 3]  =  Just [1 , 2 , 3]
consolidate [Just 1 , Nothing , Just 3]  =  Nothing
```

After completing the type-directed recursive definition, rewrite your definition using the `Maybe` instances of the `Applicative` methods (`pure` and `(<*>)`) and without performing any case analysis on a term of `Maybe` type. The terms `Nothing` and `Just` should not occur anywhere in your definition.

Task 4
Complete the definition of the function `applicify`, which takes any binary operator and extends it to any applicative type constructor:

```
applicify  :  {t : Type -> Type} -> Applicative t =>
  (op : a -> a -> a) -> t a -> t a -> t a
```

Using your definition, you can easily define operators such as:

```
infixl 7 +?
(+?)  :  Num a => Maybe a -> Maybe a -> Maybe a
(+?)  =  applicify (+)
```

[1] I concede that this is horrible concrete syntax.

```
infixl 7 +*
(+*)  :  Num a => Vect n a -> Vect n a -> Vect n a
(+*)  =  applicify (+)
```

that behave as follows:

```
Just 3 +? Just 4 +? Just 5  =  Just 12
Just 3 +? Nothing +? Just 5  =  Nothing
[1,2,3] +* [4,5,6] +* [7,8,9]  =  [12,15,18]
```

What Does it Do, and Why?

Task 5
Write down the type signatures for the `List` and `Vect n` instances of the methods in the `Functor`, `Applicative`, and `Monad` interfaces (`map`, `pure`, `(<*>)`, and `join`).

Task 6
Use the Idris REPL to explore the behavior of each of these method instances (the function `the` will be helpful here). Write a brief English description of the behavior of each of the 8 functions just described. In the cases where the behavior of the `List` and `Vect n` instances of a method differ, explain why these differences are required by the respective types.