Lab 8

Functional Programming (ITI0212)

2022-03-18

This week we saw interfaces, which are a collection of methods, representing the notion of a family of types with common operations (e.g. “numeric types”, whose values we expect to be able to add and multiply). Interfaces may be parameterized, which allow us to provide different implementation for different types.

Interfaces may be used to constrain generic functions, other interfaces, and implementations of interfaces.

The Idris standard library defines many useful interfaces such as `Num`, `Show`, `Eq`, `Ord`, `Cast`, etc. We can also define our own interfaces and provide implementations of any interface. Implementations may not overlap, but we can use named interfaces to provide multiple implementations.

Comparing Strings

The default `Ord` implementation for `String`s compares them *lexicographically*, that is according to dictionary order – but also with uppercase letters coming before lowercase (thus `"A" < "a"` evaluates to `True`).

Task 1
Write a named `Ord` implementation for `String`s that compares them according to their length:

```
implementation [len] Ord String where
```

Note: if you choose to implement the `<` method, you will need to use the prefix syntax to call the named implementation, e.g. `(<) @{len} "a" "ab"`.

Comparing Lists

The default `Eq` implementation for `List`s 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 task you will need to import `Data.List`.

Task 2
Write a named `Eq` implementation 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`

Preorders

The `Ord` interface from the standard library allows us to implement *total* orders on the values of a type: an implementation of `Ord` for a given type allows us to compare any two values of that type.

A preorder is a more general order relation, which is simply a binary predicate $\leqslant$, having the properties of reflexivity ($\forall x, x \leqslant x$) and transitivity ($\forall xyz, x \leqslant y \wedge y \leqslant z \implies x \leqslant z$).

Later in the course we will see how to specify these properties in Idris, but for this lab a preorder is just a binary predicate whose implementations we should manually ensure to be reflexive and transitive.

For example, “divides” defines a preorder on the natural numbers: we write $n \leqslant m$ for “n divides m”.

Task 3
Write an interface `PreOrd` for preorders and implement the “divides” preorder on `Integer`. Convince yourself that your implementation is reflexive and transitive.

Hint: you may find the `mod` function useful, where `mod n m` is the remainder when dividing `n` by `m`.

Task 4
Recall the type of arithmetic expressions from the lecture:

```
data AExpr : Num n => Type -> Type where
  V : n -> AExpr n
  Plus : AExpr n -> AExpr n -> AExpr n
  Times : AExpr n -> AExpr n -> AExpr n
```

Write an implementation of `Show` for `AExpr n` that displays arithmetic expressions in infix notation and fully parenthesized, e.g. `show (Plus (V 2)(Times (V 3)(V 0)))` should evaluate to `(2+(3*0))`.

Hint: your implementation will need more than one constraint.