Lab 5

Functional Programming

2024-02-23

This week we are learning about programming interfaces. An Idris *interface* is similar to a Haskell typeclass, a Rust trait, or a Java abstract class. An interface serves as a *constraint* on a collection of types or type constructors. The elements of an interface are its *methods*. To *implement* an interface we must provide definitions for a *basis* of its methods.

We saw how to define and implement interfaces and learned about the following interfaces from the standard library:

- `Show`: for types whose elements have a string representation
- `Num`: for numeric types
- `Eq`: for types whose elements can be compared for equality
- `Ord`: for types whose elements can be ordered
- `Cast`: for transforming elements of one type into elements of another

We also saw how to define and invoke *named implementations* of an iterface and how to use the function `the : (0 a : Type) -> a -> a` to disambiguate types in the interactive interpreter.

Task 1
Recall the `Shape` type from week 2. Write a `Show` implementation for `Shape`s so that:

```
Lab5> show $ Circle 5
"Circle with radius 5.0"
Lab5> show $ Rectangle 3 4
"Rectangle with width 3.0 and height 4.0"
Lab5> show $ IsosTriangle 2 2
"Isosceles triangle with base 2.0 and height 2.0"
Lab5> show $ RegularPolygon 5 (7/2)
"Regular 5-gon with side length 3.5"
```

Task 2
Recall the `AExp` type constructor for arithmetic expressions from week 5. Write an *evaluator* for arithmetic expressions over numeric types:

```
eval  :  Num a => AExp a -> a
```

For example:

```
Lab5> eval $ Val 2 `Plus` Val 2
4
Lab5> eval $ Val 2 `Times` (Val 3 `Plus` Val 4)
14
```

Task 3
The implementation of the `Eq` interface for arithmetic expressions that we wrote in class compares expressions structurally. Write an implementation of the `Eq` interface for arithmetic expressions over numeric types that instead compares them by value:

```
Lab5> (Val 40 `Plus` Val 2) == (Val 6 `Times` Val 7)
True
Lab5> Val 0 == Val 1
False
```

Hint: use your `eval` function.

Task 4
Write an implementation of the `Ord` interface for arithmetic expressions over numeric types that orders them according to their values:

```
Lab5> Val 3 <= Val 3
True
Lab5> compare (Val 1) (Val 2)
LT
Lab5> max (Val 3 `Plus` Val 2) (Val 3 `Times` Val 2)
Times (Val 3) (Val 2)
```

Hint: you only need to implement one method.

Task 5
The default implementation of the `Eq` interface for `List` types compares lists for equality *element-wise*:

```
Lab5> (==) [1,2,3] [1,2,3]
True
Lab5> (==) [1,2,3] [3,2,2,1]
False
Lab5> (==) [1,2,3] [1,2,4]
False
```

Write a named implementation of the `Eq` interface for `List` types that compares them *set-wise*:

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

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

```
Lab5> (==) @{setwise} [1,2,3] [1,2,3]
True
Lab5> (==) @{setwise} [1,2,3] [3,2,2,1]
True
Lab5> (==) @{setwise} [1,2,3] [1,2,4]
False
```

Hint: the following standard library functions may be helpful:

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

Task 6
In class this week we saw how to write the *quicksort* algorithm recursively. A key component of the *mergesort* algorithm is the following function:

```
merge_list  :  Ord a => List a -> List a -> List a
```

which combines two lists into one so that the head of the result list is the smaller of the heads of the two argument lists, and the tail of the result list is the result of recursively merging the remainder of the two argument lists. This has the effect that, if the two argument lists are already sorted then the result list will also be sorted. Write the `merge_list` function so that:

```
  Lab5> merge_list [] [1,3,2]
  [1, 3, 2]
  Lab5> merge_list [1,3,2] []
  [1, 3, 2]
  Lab5> merge_list [1,4] [2,3]
  [1, 2, 3, 4]
  Lab5> merge_list [4,1] [3,2]
  [3, 2, 4, 1]
```

Task 7
In Lab 3 you defined a type isomorphism between the types `Nat` and `List Unit`. Define *lossless* `Cast` implementations back and forth between these two types.