Lab 8

Functional Programming (ITI0212)

2020.04.08

This week we learned about how we can use indexed types to make data structures that know their own shape. We saw how the shape of a `List` is its length, and thus how a `Vect` is a list that knows its shape.

Then we reviewed the type of (node-labeled) binary trees, and saw how we could create a shapely version of these as well. We saw how using shapely data structures can make programming easier and safer by turning programming errors into statically- and automatically-checked type errors.

Imports and Namespaces

For this week's lab exercises you will need access to the files `shapeless_tree.idr` and `shapely_tree.idr` that were developed during the lecture. You should find them on the course web page next to the current document.

In order to access the contents of those files from your lab file, you will need to place them together in the same directory and include the following lines at the beginning of your lab file:

```
import shapeless_tree
import shapely_tree
```

These *import statements* tell Idris to search its default path (which always includes the current directory) for Idris files with the given names. If the search is successful then the types and functions that are exported by these files become available for use within the current file when qualified by their respective module names.

For example, the file `shapeless_tree.idr` defines a module called `ShapelessTree`, which in turn contains the type `Tree`. You can refer to this type within your own Idris program as `ShapelessTree.Tree`. However, if the thing you want to refer to is unambiguous, either because it is the unique thing in scope with the given name, *or* because it can be disambiguated by its type from other things in scope with the same name, then you may omit the module namespace prefix.

Thus `ShapelessTree.Tree` can be disambiguated from `ShapelyTree.Tree` if you give either at least one argument whose type Idris can infer. This is because `ShapelessTree.Tree` takes only one argument, a `Type` parameter, whereas `ShapelyTree.Tree` takes two arguments, first a `TreeShape` index and next a `Type` parameter. Because a term of type `TreeShape` is not a `Type`, and vise-versa, Idris can figure out which of the two `Tree`s was intended. For example, "`Tree Nat`" must mean `ShapelessTree.Tree Nat` while "`Tree LeafShape`" must mean `ShapelyTree.Tree LeafShape`.

Warming Up

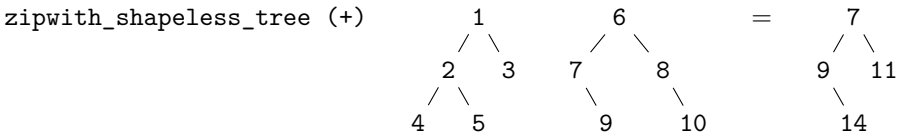
To begin, we'll write a simple recursive function on trees of both the shapeless and shapely variety.

Task 1
Write the "zip-with" function for shapeless trees:

```
zipwith_shapeless_tree  :  (a -> b -> c) ->
  Tree a -> Tree b -> Tree c
```

This function should return the (shapeless) tree resulting from applying the given binary function to the pair of elements in the corresponding positions of its two argument trees. As with `zip_tree`, it should silently truncate the result tree to the intersection of the shapes of the two arguments.

For example:



Task 2
Now write the same "zip-with" function for shapely trees:

```
zipwith_shapely_tree  :  (a -> b -> c) ->
  Tree shape a -> Tree shape b -> Tree shape c
```

Note that, like `zipWith` for `Vect`s, the type enforces the invariant that the two input trees and the output tree all have the same shape.

When developing this function interactively, pay attention to the clauses generated by the case-split and try using term search to solve the sub-goals.

Task 3
Recall the shapely tree zipping function presented in lecture:

```
zip_tree  :  Tree shape a -> Tree shape b -> Tree shape (Pair a b)
```

Write its inverse function,

```
unzip_tree  :  Tree shape (Pair a b) -> Pair (Tree shape a) (Tree shape b)
```

Shaping Up

Recall from lecture that it is easy to make a list forget its shape; i.e., to turn a `Vect` into a `List` containing the same elements in the same order. This is true for trees as well.

Task 4
Write the function `forget_shape` that turns a `ShapelyTree.Tree` into the `ShapelessTree.Tree` containing the same elements in the same positions:

```
forget_shape  :  ShapelyTree.Tree shape type -> ShapelessTree.Tree type
```

As in the case of lists, recovering the shape of a shapeless tree requires a little more work because we need to compute the shape in order to specify the type.

Task 5
Write the function `learn_shape` that turns a `ShapelessTree.Tree` into the `ShapelyTree.Tree` containing the same elements in the same positions:

```
learn_shape  :
  (tree : ShapelessTree.Tree type) -> ShapelyTree.Tree ?shape type
```

Before writing this function, you should first figure out how to solve the goal `?shape` in the signature. The best way to do this is to write an auxiliary function that computes the shape of a shapeless tree.

Wrapping Up

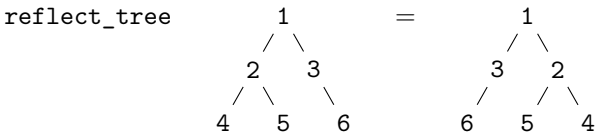
Now that we have some experience with trees and their shapes, let's try a couple more shapely exercises.

Task 6
Write a function that reflects a shapely tree (this is the shapely version of a function that you wrote already in lab 6).

```
reflect_tree  :  Tree shape type -> Tree ?reflected_shape type
```

You will again need to compute the shape in order to specify the type. However, once you have written the shape-computing function, term search should be able to solve the rest of the exercise for you.

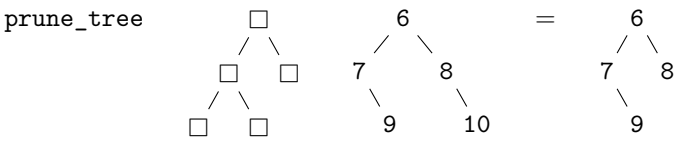
Your function should behave like this:



Task 7
Write a function that takes a tree shape and a shapely tree and "prunes" the tree to the desired shape:

```
prune_tree  :  (template_shape : TreeShape) -> Tree tree_shape type ->
  Tree ?pruned_shape type
```

Your function should behave like this:



Of course, you'll need to be able to compute the shape of a pruned tree in order to express the type of the result.