

Lab 3

Functional Programming (ITI0212)

2023-02-17

This week we learned about *parameterized type families*, also known as *type constructors*. These have types `Type -> ... -> Type`. For example, the `List` and `Maybe` type constructors each take one type parameter, and thus have type `Type -> Type`; while the `Pair` and `Either` type constructors each take two type parameters, and thus have type `Type -> Type -> Type`.

We also learned how to write *generic functions*, which act *uniformly* on the types of such families. We saw how to use *implicit arguments* to avoid having to pass type arguments explicitly, and how to use *implicit binding* to shorten the way that we write type specifications for these functions. Recall that the *quantity* `0` is used to indicate that we treat a type generically, and that implicit binding elaboration inserts this for us automatically.

By default, Idris suppresses implicit arguments when displaying types. However, we can use the REPL command `:ti` in order to have Idris show us the full type of an expression, including all implicit arguments.

Task 1

Write any function of the following type:

```
swap_pair : Pair a b -> Pair b a
```

Hint: Recall that this type elaborates to:

```
{0 a : Type} -> {0 b : Type} -> Pair a b -> Pair b a
```

thus any function you write must be generic in both `a` and `b`.

Task 2

Write any function of the following type:

```
swap_either : Either a b -> Either b a
```

Question: Did you have any choice in the functions you wrote in tasks 1 and 2?

Task 3

Write a generic function

```
reverse_list : List a -> List a
```

that reverses the order of the elements of a list; for example:

```
Lab3> reverse_list []
[]
Lab3> reverse_list [1]
[1]
Lab3> reverse_list [1, 2]
[2, 1]
Lab3> reverse_list [1, 2, 3]
[3, 2, 1]
```

Hint: Use recursion on the argument list. The *list concatenation* function that we wrote this week in lecture will be helpful. It is also in the standard library as

```
Prelude.List.(++) : List a -> List a -> List a
```

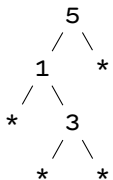
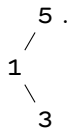
Task 4

The following type constructor defines *node-labeled binary trees* or just “trees” for short.

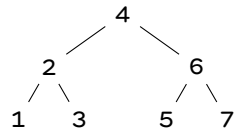
```
data Tree : (a : Type) -> Type where
  -- a tree is either empty:
  Leaf : Tree a
  -- or it is a left subtree, a current element, and a right subtree:
  Node : (l : Tree a) -> (x : a) -> (r : Tree a) -> Tree a
```

It is customary to draw trees as downward-growing diagrams. For example, the `Tree Nat`

```
t1 = Node (Node Leaf 1 (Node Leaf 3 Leaf)) 5 Leaf
```

can be drawn as:  , where the *s represent leaves. We may also omit the leaves: .

Enter the definition of the `Tree` type constructor into your lab file and write the term for `t2 : Tree Nat` corresponding to the following tree.



Hint: If the term you are trying to write gets too unwieldy you can always use a local definition (`let` binding) to divide it into more manageable pieces.

Task 5

Write a generic function

```
size : Tree a -> Nat
```

that counts the number of nodes in a tree; for example:

```
Lab3> size t1
3
Lab3> size t2
7
```

Task 6

A *type isomorphism* is a pair of back-and-forth functions between two types, $f : a \rightarrow b$ and $g : b \rightarrow a$, such that if we apply either one to the result of applying the other then we get back the original argument; that is, for any $x : a$ and $y : b$ we have that $g (f x)$ evaluates to x and $f (g y)$ evaluates to y .

Write a type isomorphism between the types `Nat` and `List Unit`; that is, write functions

```
n_to_lu : Nat -> List Unit
```

```
lu_to_n : List Unit -> Nat
```

so that, for example:

```
Lab3> lu_to_n (n_to_lu 0)
0
Lab3> lu_to_n (n_to_lu 1)
1
Lab3> lu_to_n (n_to_lu 42)
42
Lab3> n_to_lu (lu_to_n [])
[]
Lab3> n_to_lu (lu_to_n [()])
[()]
Lab3> n_to_lu (lu_to_n [(), (), ()])
[(), (), ()]
```