Lab 3

Functional Programming (ITI0212)

2022-02-11

Brief recap: *Parameterized types* are types that depend on one or more types. For example we saw the parameterized types `List`, `Pair`, `Maybe`. Their type constructors have function types of the form `Type -> Type`, `Type -> Type -> Type`, etc. Lowercase names are *type-level variables* representing parameters, and are elaborated by Idris as *implicitly-bound arguments*. Parameterized types can be considered as defining an (infinite) family of types, one for each choice of parameter type(s).

Generic functions are functions whose type signatures involve parameterized types. They are *parametrically polymorphic*: their behaviour has no capacity to differ according to the type of their parameter(s). Sometimes this means there is only one way to implement a generic function of a given type signature, which is helpful both to the human programmer, and Idris' proof search functionality.

Task 1
To implement a generic function:

```
swap : Pair a b -> Pair b a
```

Task 2
To implement two generic functions:

```
inl : a -> Either a b

inr : b -> Either a b
```

Consider: is there more than one possible implementation for the functions in tasks 1 and 2?

Task 3
To implement a generic function that reverses a list (e.g. `reverse' [1,2,3] = [3,2,1]`).

Note: since `reverse` is already defined in the standard library, use a different name such as `reverse'` for your function.

Hint: you may wish to make use of the concatenation function `(++)` from Idris' standard library.

Definition
A node-labelled binary tree is a data structure that holds values of some type at its branch nodes.

We can define the type of binary trees as a parameterized type:

```
data Tree : Type -> Type where
  Leaf : Tree a
  Branch : (left : Tree a) -> (val : a) -> (right : Tree
     a) -> Tree a
```

This says that a `Tree a` is either a `Leaf`, or a `Branch` (a left "sub-tree", a term of type `a` and a right "sub-tree").

Note: we have given names (`left`, `val`, `right`) to our parameters for the `Branch` element constructor. This both improves readability of the code and provides clearer default names when case-splitting (try it in the next task!)

For example, consider the following term of type `Tree Integer`:

```
Branch (Branch Leaf 1 (Branch Leaf 3 Leaf)) 5 Leaf
```

We can mentally picture this term as the following node-labelled tree:

```
       5
      / \
     1   *
    / \
   *   3
      / \
     *   *
```

Task 4
To implement a generic function `size : Tree a -> Nat`, returning the number of values stored in a tree. For example, the size of the tree given in the above definition is 3.

Note: you will need to copy the above parameterized type `Tree` into your file, since this is not in Idris' standard library.

Hint: your function needs only two clauses.

Task 5
To implement a generic function `flatten : Tree a -> List a`, returning a list containing the values stored in a tree.

Task 6
To implement two functions:

```
nat_to_list : Nat -> List Unit

list_to_nat : List Unit -> Nat
```

that are mutually inverse, i.e.

```
nat_to_list (list_to_nat x) = x
```

and

```
list_to_nat (nat_to_list y) = y
```

for all `x : List Unit, y : Nat`.

For now, you can check your functions are mutually inverse by trying a few cases in the REPL. Later in the course we will see how to *prove* (using Idris) that they are mutually inverse!

We say that these functions witness a *type isomorphism* between `List Unit` and `Nat`.