Lab 16

Functional Programming (ITI0212)

2023-05-19

This week we are learning about record types in Idris.

A *record type* is a kind of user-defined type, like an inductive type. Record types don't expand the expressive power of the language, in the sense that for any record type we can define an equivalent inductive type. However, record types are convenient for representing some data structures because Idris automatically generates some things for us. A record type has a named *constructor* and any number of named and typed *fields*. Idris automatically computes the type of the constructor from the types of the fields, and generates *projection functions* to extract field values. Projection function application may optionally be written using postfix *dot syntax*.

Idris automatically generates a *namespace* for each record type, allowing multiple record types with the same field names to coexist in a module. Records have a convenient *update syntax* where we may assign a new field value using "`:=`", or compute one using "`$=`". Like inductive types, record types may be both parameterized and dependent. Finally, we saw that by combining record types and expression search we can re-create Idris's *interface* system.

For this lab you will want to import the standard library modules for `Nat`s, `List`s and `String`s.

Task 1
A (non-negative) *rational number* has a `numerator` and a `denominator`, each of which is a natural number (for this task we will ignore the *constraint* that the denominator can't be zero). Write a record type `Rational` to encode the data of such a number, naming its constructor `(//)` so that we can express a rational number with numerator `m` and denominator `n` as `m//n`. Your rational numbers need not be expressed in *lowest terms*.

```
Lab16> :t 4//6
4 // 6 : Rational
Lab16> numerator (4//6)
4
Lab16> denominator (4//6)
6
```

Hint: don't forget to make an `infix` declaration for `(//)`.

Task 2
Write `Eq` and `Num` instances for the `Rational` type.

For example:

```
Lab16> 1//2 == 3//6
True
Lab16> 1//2 + 2//3
7 // 6
Lab16> 1//2 * 2//3
2 // 6
Lab16> the Rational (fromInteger 42)
42 // 1
```

Task 3
Consider the following record type for representing social media posts:

```
record  Post  where
  constructor  MkPost
  text  :  String
  likes  :  Nat
  comments  :  List Post
```

For convenience, we can use the following `Show` implementation to display posts:

```
partial
implementation  Show Post  where
  show p =
    let
      rec = foldr (++) "" (map show p.comments)
    in
      p.text ++ "\n" ++
      "Likes: " ++ show p.likes ++ "\n" ++
      indent rec
    where
      indent  :  String -> String
      indent  =  unlines . map ("  " ++ ) . lines
```

Write the following functions for creating, liking, and reacting to posts:

```
new_post  :  String -> Post

like_post  :  Post -> Post

react_post  :  String -> Post -> Post
```

So that for the following posts:

```
p0  =  new_post "My brother in Pepe, this is so lit!"
p1  =  react_post "Yeet!" p0
p2  =  like_post p1
p3  =  react_post "LoooooL" p2
```

we have:

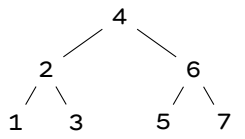
```
Lab16> :exec putStr $ show p3
My brother in Pepe, this is so lit!
Likes: 1
  Yeet!
  Likes: 0
  LoooooL
  Likes: 0
```

Hint: record update syntax will be helpful here.

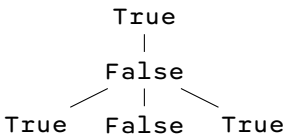
Task 4
A (nonempty) *RoseTree* with element type `a` consists of a `Node` with a `label` of type `a` and a list of `children`, which are themselves `RoseTree`s with element type `a`. Define this data structure as a parameterized record type in Idris so that the following trees have the structure shown below.

```
tree1  :  RoseTree Nat
tree1  =  Node 4 [Node 2 [Node 1 [] , Node 3 []] , Node 6 [Node 5 [] , Node 7 []]]

tree2  :  RoseTree Bool
tree2  =  Node True [Node False [Node True [] , Node False [] , Node True []]]
```



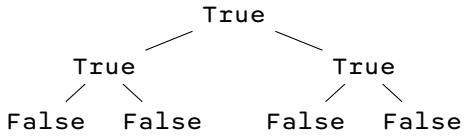
,



Task 5
Write a `Functor` implementation for `RoseTree`s,

```
implementation  Functor RoseTree  where
```

so that mapping the (boolean-valued) predicate for evenness over `tree1` yields the following tree:



Task 6
Write the `index` function for `RoseTree`s,

```
index  :  List Nat -> RoseTree a -> Maybe a
```

so that, for example:

```
Lab16> index [] tree1
Just 4
Lab16> index [1] tree1
Just 6
Lab16> index [0 , 1] tree1
Just 3
Lab16> index [0 , 2] tree1
Nothing
```

Hint: The index function for `List`s in the standard library is `getAt : Nat -> List a -> Maybe a`.

Task 7 (optional challenge)
In fact, it is possible to make a record type for rational numbers that enforces the constraint that the denominator is not zero. We can do this using a *constraint field*. The syntax for such a field is as follows:

```
{auto <field_name> : <field_type>}
```

Here, the braces around the field indicate that the corresponding constructor argument is *implicit*, and the keyword "`auto`" indicates that it is to be found by *search* rather than by unification.

Define a record type of rational numbers with constructor `(///)` that enforces the constraint that the denominator is not zero. Then define `Eq` and `Num` instances for it.

Note: when explicitly inserting an implicit argument in a definition clause Idris makes you write an infix constructor name using prefix notation. I don't know why this is.

If all goes well you should be able to write:

```
Lab16> 1///2 + 2///3
7 /// 6
Lab16> 1///2 * 2///3
2 /// 6
Lab16> 1///2 * 1///0
Error: Can't find an implementation for NonZero 0.
```