Lab 13

Functional Programming (ITI0212)

2024-04-30

This week we are learning about decidability and automation in Idris programming.

A *decision procedure* for a predicate is an algorithm that for each index either produces a proof that the predicate holds or else a refutation proving that it does not. In Idris the type constructor for decidability is called `Dec` with constructors `Yes` and `No`. Additionally, there is an interface for types with decidable equality called `DecEq` in the standard library module `Decidable.Equality`.

A *constraint argument* (also called an *auto-implicit argument*) is used to ensure that some validity condition is satisfied. It is written using the double-shafted arrow `=>` and is intended to be found by Idris's term search mechanism. By default this consists of using constructors, recursion, and function literals in order to find a term of a given type, but you may specify additional terms for it to try using the `%hint` directive.

Recall the type constructor for node-labeled binary trees:

```
data  Tree : a -> Type  where
  Leaf  :  Tree a
  Node  :  (l : Tree a) -> (x : a) -> (r : Tree a) ->
     Tree a
```

In this lab you will write a decision procedure for `Tree` equality and use it to make the `Tree` type an instance of the `DecEq` interface.

The first three tasks are simple one-liners. Remember that equality types in Idris are inductively defined with a single constructor called `Refl`, which relates things only with themselves.

Task 1
Convince Idris that the type of equalities between leaves and nodes, and the type of equalities between nodes and leaves, are both empty:

```
implementation  Uninhabited (Leaf = Node l x r)  where

implementation  Uninhabited (Node l x r = Leaf)  where
```

Task 2
Convince Idris that two non-empty trees whose root nodes have different labels cannot be equal:

```
labels_differ  :  Not (x1 = x2) ->
  Not (Node l1 x1 r1 = Node l2 x2 r2)
```

Task 3
Convince Idris that two non-empty trees with differing left or right subtrees cannot be equal:

```
left_trees_differ  :  Not (l1 = l2) ->
  Not (Node l1 x1 r1 = Node l2 x2 r2)

right_trees_differ  :  Not (r1 = r2) ->
  Not (Node l1 x1 r1 = Node l2 x2 r2)
```

Task 4
Using the functions that you wrote in tasks **??**–**??**, write a decision procedure for `Tree` equality, under the constraint that the element type of the trees has decidable equality:

```
decide_tree_eq  :  DecEq a => (t1 , t2 : Tree a) -> Dec (t1 = t2)
```

Hint: Case-split the two argument trees. If they are built from different constructors, use your result from task **??**. If they are both `Leaf`s then they are equal (why?). If they are both `Node`s then compare the labels using your result from task **??** and, if needed, recurse on the subtrees using your results from task **??**.

Task 5
Finally, use the decision procedure that you wrote in task **??** to make the types of trees whose element types are instances of the `DecEq` interface themselves instances of the `DecEq` interface:

```
implementation  DecEq a => DecEq (Tree a)  where
```