Exercise 2.1  Folding over Trees

Define a datatype for binary trees that store data only at leafs.

datatype 'a tree =

Define a function that returns the list of elements resulting from an in-order traversal of
the tree.

fun inorder :: "'a tree ⇒ 'a list"

Here a look at Isabelle/HOL’s standard function fold.

fun fold ::

In order to fold over the elements of a tree, we could use fold f (inorder t) s. However,
from an efficiency point of view, this has a problem. Which?

Define a more efficient function fold2tree, and show that it is correct

fun fold2tree :: "(t ⇒ 'a ⇒ 'a list) ⇒ 'a tree ⇒ 'a list"

lemma "fold2tree (inorder t) = fold f (inorder t) s"

Exercise 2.2  Shuffle Product

To shuffle two lists, we repeat the following step until both lists are empty: Take the
first element from one of the lists, and append it to the result.

That is, a shuffle of two lists contains exactly the elements of both lists in the right
order.

Define a function shuffles that returns a list of all shuffles of two given lists.
theory tree

imports Main

begin

  datatype Tree = Leaf | Node Tree Tree

  fun inorder :: Tree => List where
    inorder Leaf = None
    inorder (Node a b) = (cases a of
      Leaf => inorder b
      Node a b => some a :: inorder b)

end
theory foo2
imports Main
begin
  datatype atree = Leaf a | Node a atree a atree
  datatype atree = Leaf a | Node a atree a atree
  fun inorder :: a tree => a list where
  "inorder_ = undefined"
  fun inorder :: a tree => a list where
  "inorder_ = undefined"
end
Functional Data Structures

Exercise Sheet 2

Exercise 2.1 Folding over Trees

Define a datatype for binary trees that store data only at leaves.

datatype 'a tree = Leaf 'a | Node 'a 'a tree 'a tree

Define a function that returns the list of elements resulting from an in-order traversal of the tree.

fun inorder : 'a tree -> 'a list

Have a look at Isabelle/HOL’s standard function fold.

fold elem tree

In order to fold over the elements of a tree, we could use fold f (inorder t) x. However, from an efficiency point of view, this has a problem. Which?

Define a more efficient function fold'elem, and show that it is correct.

fun fold'elem : {elem -> 'a -> 'a tree -> 'a list} -> 'a list

(value (inorder (Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Node (Leaf 4) (Leaf 5))))))
Functional Data Structures

Exercise Sheet 2

Exercise 2.1 Folding over Trees

Define a datatype for binary trees that store data only at leaves.

cdata type 'a tree =

Define a function that returns the list of elements resulting from an in-order traversal of the tree.

fun inorder : 'a tree -> 'a list

Have a look at Isabelle/HOL’s standard function fold.

fun fold steps

In order to fold over the elements of a tree, we could use fold f (inorder t) s. However, from an efficiency point of view, this has a problem. Which?

Define a more efficient function foldTree, and show that it is correct.

fun foldTree : ('a * 'a -> 'a) -> 'a -> 'a tree -> 'a list

```ocaml
let rec fold r t s =
  match t with
  | None -> r
  | Some (a, t') ->
    let rec fold' f t' s' =
      match t' with
      | None -> r
      | Some (a', t'') ->
        begin
          let f' = fold f a a' in
          f' (fold' f t'' s')
        end
    in
    fold' f t s

by auto

definition "fold_tree (t: 'a tree) f = fold f (fold f t)"

fun fold_tree : 'x * 'a tree * ('a -> 'b) -> 'b tree "fold_tree spec f t s" where
  "fold_tree f = undefined"

lemma "fold_tree f t s = fold_tree_spec f t s"
```

```ocaml
let rec fold r t s =
  match t with
  | None -> r
  | Some (a, t') ->
    let rec fold' f t' s' =
      match t' with
      | None -> r
      | Some (a', t'') ->
        begin
          let f' = fold f a a' in
          f' (fold' f t'' s')
        end
    in
    fold' f t s

by auto

definition "fold_tree (t: 'a tree) f = fold f (fold f t)"

fun fold_tree : 'x * 'a tree * ('a -> 'b) -> 'b tree "fold_tree spec f t s" where
  "fold_tree f = undefined"

lemma "fold_tree f t s = fold_tree_spec f t s"
```
Define a more efficient function `fold_list`, and show that it is correct:

```ocaml
fun fold_list : "[a] ⇒ 'a ⇒ 'a ⇒ 'a" where
    fold_list [Leaf n] = f n
    fold_list (Node r t) = fold_list r (fold_list t f)
```

Lemma: `fold_list (Leaf n) = fold_list Spec T t n` unfolding `fold_list Spec` def

apply (induction 1 arbitrary x)
apply auto
```

Proof: [prove]
```
Goal: No subgoals!

Exercise 2.2 Shuffle Product

To shuffle two lists, we repeat the following step until both lists are empty: Take the first element from one of the lists, and append it to the result.

This is a shuffle of two lists contains exactly the elements of both lists in the right order.

Define a function `shuffles` that returns a list of all shuffles of two given lists:

```ocaml
fun shuffles : "['a list ⇒ 'a list ⇒ 'a list list"]
```

Show that the length of any shuffle of two lists is the sum of the lengths of the original lists.
fun fold_left f t x = fold_left t (fold_left f t x)
  where
  fold_left t Nil x = x
  fold_left (Cons a) t x = fold_left t (Cons (f a) x)

lemma fold_left_nil_f = fold_left_spec f t Nil
  unfolding fold_left_spec_def
  apply (induction t arbitrary a)
  apply auto
  done

theorem fold_left t x s s' = fold_left_spec t Nil x
done
Fun shuffles :: "a list -> "a list list\n  where
  shuffles xs [] = undefined
  | shuffles [] ys = undefined
  | shuffles (x:xs) (y:ys) = undefined
Exercise 2.2 Shuffle Product

To shuffle two lists, we repeat the following step until both lists are empty: Take the first element from one of the lists, and append it to the result.

That is, a shuffle of two lists contains exactly the elements of both lists in the right order.

Define a function shuffle that returns a list of all shuffles of two given lists

```haskell
fun shuffle :: "a list ⇒ 'a list ⇒ 'a list list"
```

```haskell
fun shuffle :: "a list ⇒ 'a list ⇒ 'a list list" where
  "shuffle " [] = []
  | "shuffle " [x] = [x]
  | "shuffle " (y::ys) = map (λx. y::(shuffle xs ys)) @ map (λx. [x] @ (shuffle xs ys)) (shuffle (y::ys) (shuffle xs [y::ys]))
```

Lemma 1 c set (shuffle xs ys) → length t = length xs + length ys

```haskell
prod
```

Fun termination order: [λp. length (send p)] (λe. (λt. length t @ [])) (λe. (λt. length t @ []))
Theorem \( \forall \text{a list } y \text{ set (shuffles } x \text{ y) } = \text{set } \text{length } y \text{ = set } \text{length } x \text{ = set } \text{length } y \).\n
Proof:

1. \text{def \{shuffles\}}: \forall \text{a list } y \text{ set (shuffles } x \text{ y) } = \text{set } \text{length } y \text{ = set } \text{length } x \text{ = set } \text{length } y \).

2. \text{apply induction } y \text{ arbitrary y)}

3. \text{auto}

4. \text{the shuffles.simps}

End.
Theorem \( \exists x. \exists y. (x < y) \rightarrow \text{length } x + \text{length } y \)

apply auto

fun sum \( \): ('a list \( \rightarrow \) 'a \) \( \rightarrow \) 'a \) \( \rightarrow \) 'a \)

| 'a list \( \rightarrow \) 'a |

definition \( \text{sum} \): ('a list \( \rightarrow \) 'a \) \( \rightarrow \) 'a \)

end

consts

\( \text{sum} \): ('a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \times \text{list } 'a \)
fun leix : "a list -> a" where
  "leix [] = 0" | "leix (c::xs) = x + leix xs"

definition "leix' : "a list -> a" = fold (op+) 1 0"
definition "leix" = fold (op+) 1 0" applying induction
apply auto

proof [prove]
goal (1 subgoal): 1. leix [] = leix' []
Exercise 2.3 Fold function

The fold function is a very generic function, that can be used to express multiple other interesting functions over lists. Write a function to compute the sum of the elements of a list. Specify two versions, one direct recursive specification, and one using fold. Show that both are equal.

fun listSum :Nat list → Nat

definition listSum = list.fold op + 0

lemma listSum l = listSum l

Homework 2.1 Distinct lists

Submission until Friday, May 12, 11:59pm. Submit your solution via https://canvas.binghamton.edu. Submit a theory file that runs in Isabelle-2018-1 without errors.

Define a function distinct, that checks whether an element is contained in a list. Define the function directly, not using set.

fun distinct : "a ⇒ Nat list ⇒ bool"

Define a predicate (distinct) to characterize distinct lists, i.e., lists whose elements are pairwise disjoint. Hint: Use the function contains.

fun distinct : "a list ⇒ bool"

Show that a reversed list is distinct if and only if the original list is distinct. Hint: You may require multiple auxiliary lemmas.

lemma distinct (rev x) ⇒ distinct x

Homework 2.2 More on fold

Submission until Friday, May 12, 11:59pm.

Isabelle’s fold function implements a left-fold. Additionally, Isabelle also provides a right-fold foldr.

Use both functions to specify the length of a list.

fun fold : "a list ⇒ a"
Homework 2.3 List Slices

Submission until Friday, May 15, 11:59pm. Specify a function \( \text{slice} \) that, for a list \( x_1, x_2, \ldots, x_n \), returns the slice starting at \( s \) with length \( l \), i.e., \( \{ x_s, \ldots, x_{s+l-1} \} \).

*fun slice :: "a list \to nat \to nat \to a list"*

If \( s \) or \( len \) is out of range, return a shorter (or the empty) list.

*fun slice :: "a list \to nat \to nat \to a list"*

where

Hint: Use pattern matching instead of if-expressions. For example, instead of writing

\[ f \ x = (if \ x > 0 \ then \ \ldots \ else \ \ldots) \]  

you should define two equations \( f 0 = \ldots \) and \( f (Suc \ n) = \ldots \).

Some test cases, which should all hold, i.e., yield \( \text{True} \):

value \( \text{slice } [0,1,2,3,4,5,6] \text{ at } 3 \text{ with } 2 = [3, 4] \) — In range

value \( \text{slice } [0,1,2,3,4,5,6] \text{ at } 7 \text{ with } 2 = [3, 4, 5, 6] \) — Length out of range

value \( \text{slice } [0,1,2,3,4,5,6] \text{ at } 10 \text{ with } 2 = [] \) — Start index out of range

Show that concatenation of two adjacent slices can be expressed as a single slice:

*lemma "\text{slice} x z = \text{slice} x z \text{ at } (z + 1)\)"

Show that a slice of a distinct list is distinct.

*lemma "\text{Distinct} z x = \text{Distinct} (\text{slice} x z)""

Homework 2.1 Distinct lists

in.tum.de. Submit a theory file that runs in Isabelle-2016-1 without errors.

Define a function contains, that checks whether an element is contained in a list. Define the function directly, not using \textit{set}.

*fun contains :: "a \to nat \to bool"

Define a predicate \textit{distinct} to characterize distinct lists, i.e., lists whose elements are pairwise disjoint. Hint: Use the function \textit{contains}.

*fun distinct :: "a list \to bool"

Show that a reversed list is distinct if and only if the original list is distinct. Hint: You may require multiple auxiliary lemmas.

*lemma "\text{distinct} (rev \ x) = \text{distinct} \ x"

Homework 2.2 Mon on fold