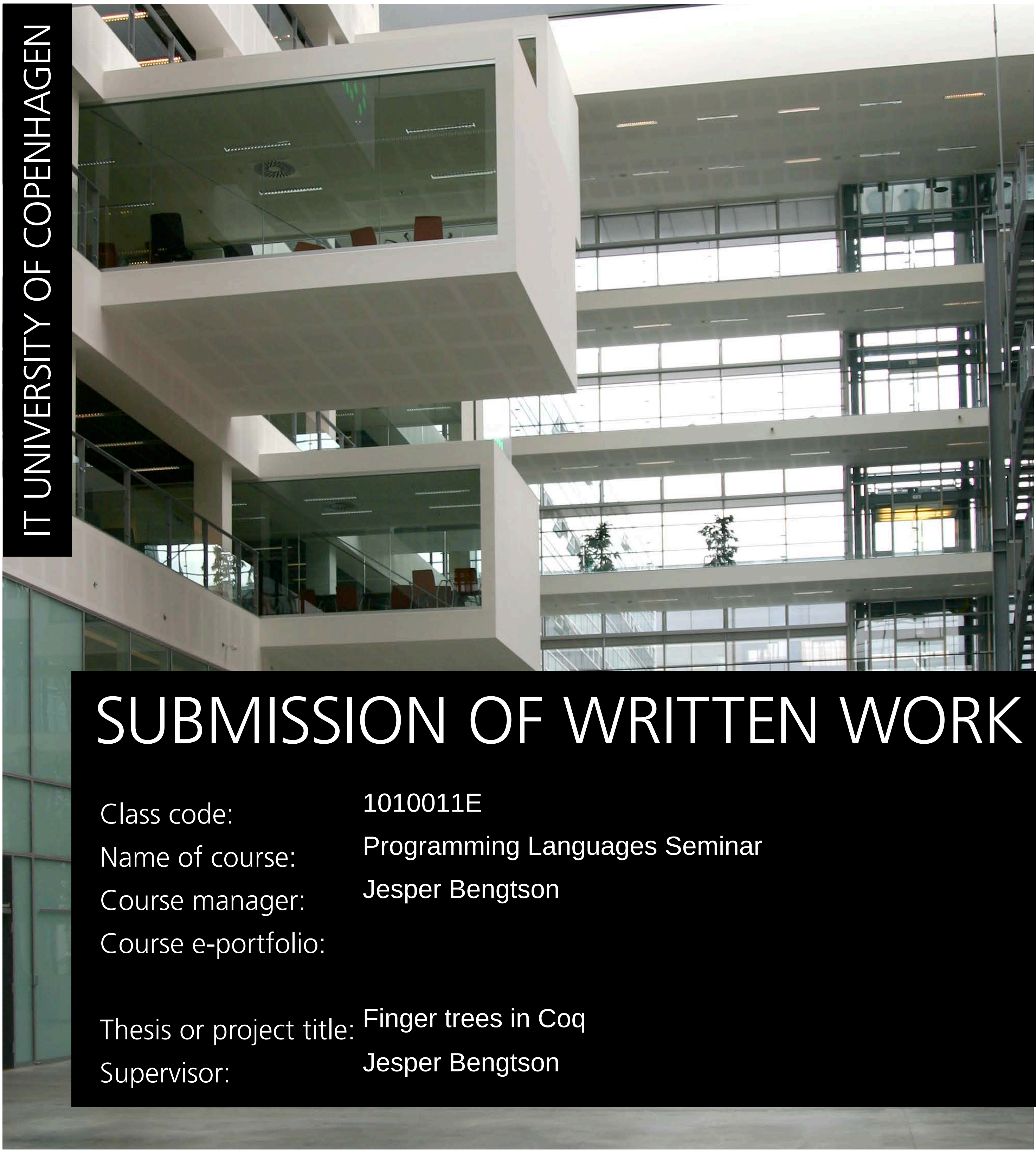

IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:	1010011E
Name of course:	Programming Languages Seminar
Course manager:	Jesper Bengtson
Course e-portfolio:	
Thesis or project title:	Finger trees in Coq
Supervisor:	Jesper Bengtson

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Adam Bjørn Schønemann	03/04-1991	adsc@itu.dk
2.	Oscar Felipe Cerda Toro	18/02-1981	osto@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Contents

Finger trees in Coq

Adam Schønemann
adsc@itu.dk

Oscar Felipe Toro
osto@itu.dk

1 Introduction

This report is a project hand-in for the "Programming Languages Seminar" class at the IT University of Copenhagen. It describes a verified implementation of finger trees in Coq, derived from Hinzes and Paterson's paper on finger trees in Haskell [1].

2 Finger trees

A finger tree is a functional and persistent data-structure that represents a sequence. First published by Leonidas J. Guibas in 1977 [2], and later refined to 2-3 finger trees in Haskell [1], it is used today in many libraries, most notably Haskell's `Data.Sequence` module in the standard library.

Finger trees allow amortized constant time pushing and popping from both ends of the sequence, as well as an amortized logarithmic time append and split operations. Finger trees can also be adapted to work as random-access sequences or priority queues.

2.1 Definition

A finger tree is in many ways like a 2-3 tree, that is, a tree where all nodes have either 2 or 3 children, and all data is contained in the leaves. However, finger trees extend the 2-3 tree with "fingers" - that is, a 2-3 tree with a prefix and a suffix "finger". A finger is typically represented by a "digit" which is a buffer of elements stored left to right. This leads us to the final definition of a finger tree:

A finger tree is either:

- An empty finger tree
- A finger tree containing a single element of type A, or
- A finger tree containing a prefix digit with elements of type A, a deeper finger tree containing elements of of type `node` A, and a suffix digit with elements of type A

where `node` is a 2-3 node. This leads us directly to the following definition in Coq (listing 1):

```
(** A node contains two or three values of A*)
Inductive node (A:Type) : Type :=
| node2: A -> A -> node A
| node3: A -> A -> A -> node A.

(** Digits hold one to four elements of A *)
Inductive digit (A:Type) : Type :=
| one : A -> digit A
| two : A -> A -> digit A
| three : A -> A -> A -> digit A
| four : A -> A -> A -> A -> digit A.

(** A fingertree is either empty, a single thing, or a deeper fingertree
    along with a prefix digit and a suffix digit *)
Inductive fingertree (A:Type) : Type :=
| empty : fingertree A
| single : A -> fingertree A
| deep : digit A -> fingertree (node A) -> digit A -> fingertree A.
```

Listing 1: The inductive definition of finger trees in Coq.

Listing 2 shows a tree representation of a sequence, and figure 1 shows a visualization of this structure. Note that for one sequence, there can be many fingertree representations. The definition of `fingertree` is a so-called *non-regular* or *nested* datatype. This is because in its inductive parameter, its type variable A is instantiated with a new type (namely `node` A). This type encodes that at each deeper level of a tree, its digits contain further nested nodes of A. However, this nested definition forces all recursive definitions on `fingertree`s to use polymorphic recursion – later we shall see that this will make proofs on `fingertree`s considerably harder in Coq.

```
Example ft_ex_01 : fingertree nat :=
  deep (two 1 2)
       (deep (two (node2 3 4) (node2 5 6))
             empty
             (two (node3 7 8 9) (node2 10 11))
       )
       (three 12 13 14).
```

Listing 2: A `fingertree` encoding of the sequence $\{1, 2, \dots, 14\}$

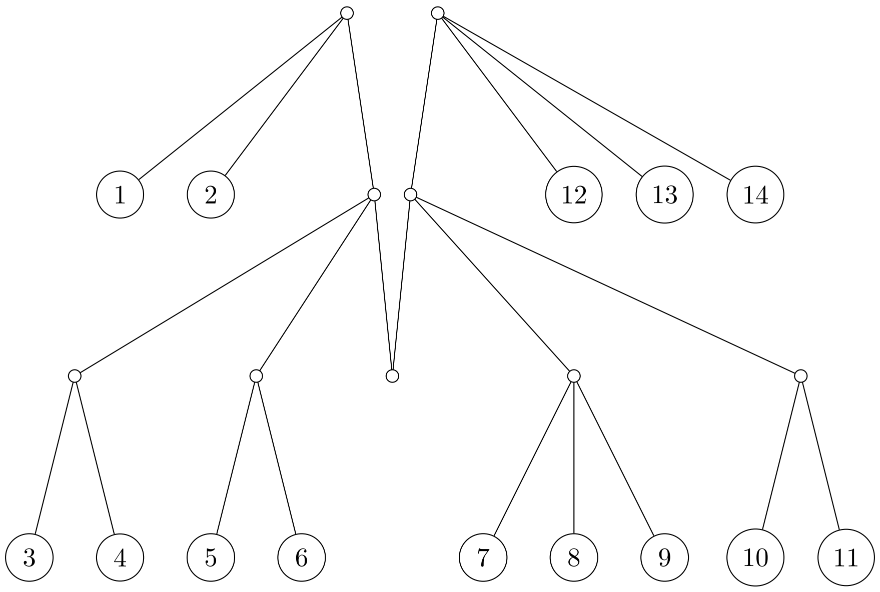

Figure 1: A visualization of the fingertree representation in listing 2.

2.2 Reducible

To capture the fact that a finger tree represents a sequence, we must define how to convert between a tree and a sequence. As functional programmers, we of course love lists, so the sequence we will be focusing on is the list. That is, we want to define two definitions

```
Definition to_list {A : Type} (tr : fingertree A) : list A.
Definition to_tree {A:Type} (s: list A) : fingertree A .
```

However, we can actually be far more general than this, by generalizing a sequence to "something that can be folded over". Another word for fold is "reduce", so we shall use this term in the spirit of [1]. Thus, we define the type-class `reduce` as:

```
Class reduce F := Reduce {
  reducer : forall {A} {B}, (A -> B -> B) -> (F A -> B -> B);
  reducel : forall {A} {B}, (B -> A -> B) -> (B -> F A -> B);
}.
```

A type constructor `F` that implements `reduce` can be reduced from the right or the left by giving it a reducing function and an initial accumulator.

Both `node` and `digit` are type constructors that represent sequences, and can thus be folded over as such:

```
Instance node_reduce : reduce node :=
  {|
    reducer :=  fun A B op nd z =>
                match nd with
                | node2 a b => op a (op b z)
                | node3 a b c => op a (op b (op c z))
                end;

    reducel :=  fun A B op z nd =>
                match nd with
                | node2 b a => op (op z b) a
                | node3 c b a => op (op (op z c) b) a
                end;
  |}.

Instance digit_reduce : reduce digit :=
  {|
    reducer :=  fun A B op dg z =>
                match dg with
                | one a => op a z
                | two a b => op a (op b z)
                | three a b c => op a (op b (op c z))
                | four a b c d => op a (op b (op c (op d z)))
                end;

    reducel :=  fun A B op z db =>
                match dg with
                | one a => op z a
                | two b a => op (op z b) a
                | three c b a => op (op (op z c) b) a
                | four d c b a => op (op (op (op z d) c) b) a
                end;
  |}.
```

We can now also make `fingertree` an instance of `reduce` as such:

```
Instance fingertree_reduce : reduce fingertree :=
  {|
    reducer :=  fix ft_reducer {A} {B} op tr z :=
                match tr with
                | empty => z
                | single x => op x z
                | deep pr m sf =>
                  let op' := reducer op in
                  let op'' := ft_reducer (reducer op) in
                  op' pr (op'' m (op' sf z))
                end;

    reducel :=  fix ft_reducel {A} {B} op z tr :=
                match tr with
                | empty => z
                | single x => op z x
                | deep pr m sf =>
                  let op'  := reducel op in
                  let op'' := ft_reducel (reducel op) in
                  op' (op'' (op' z pr) m) sf
                end;
  |}.
```

Listing 3: Instancing `fingertree` as `reduce`.

Note again that the definitions in Listing 3 make use of polymorphic recursion.

We can now easily define `to_list` for any reducible, including `fingertree`s.

```
Definition to_list {F: Type -> Type} {r: reduce F} {A : Type} (s : F A) : list A :=
  reducer cons s nil.
```

To go the other way around, i.e. `to_tree` we can use the same trick, however we must first define an equivalent operation to `cons` on trees. That is, an operation that can add an A to a fingertree A.

2.3 Adding to trees

Since finger trees are designed to be efficient deques, we can add to both the left and right of the finger tree. We shall go through the implementation of the left add operation only – the right add is similar.

First, consider a tree such as `deep (one 2) empty (one 3)`. We can easily add any number, such as 1, to the left of this tree, simply by exchanging the left prefix `one 2` by `two 1 2`. The same strategy works when the prefix is constructed with either `one`, `two` or `three`. The base cases where the tree is `empty` or `single` are also trivial. However, if the prefix is constructed with `four`, then we cannot simply add a value to the digit. We shall have to shuffle the deeper tree around to make room for the new value in the top-most prefix. Recursion helps us here – we can simply choose to insert the three right-most values of the prefix into the deeper tree, and recursion will make sure they're inserted at the first position where there is "room" for them. The resulting definition in Coq can be seen in listing 4 below:

```
Fixpoint addl {A:Type} (a:A) (tr:fingertree A) : fingertree A :=
  match tr with
  | empty                      => single a
  | single b                   => deep (one a) empty (one b)
  | deep (one b)        m sf => deep (two a b) m sf
  | deep (two b c)      m sf => deep (three a b c) m sf
  | deep (three b c d)  m sf => deep (four a b c d) m sf
  | deep (four b c d e) m sf => deep (two a b) (addl (node3 c d e) m) sf
  end.
```

Listing 4: Adding to the left of a `fingertree`.

Using `addl` we can now define `to_tree` as

```
Definition to_tree {F:Type -> Type} {A:Type} {r:reduce F} (s:F A) :
  fingertree A := reducer addl s empty.
```

2.4 Viewing trees

Another aspect in capturing the sequence represented by a finger tree is to deconstruct the tree as the conjunction of an `A` and a (possibly empty) `fingertree A`. We can call this a “view” of the tree. Finger trees can be viewed from both the left and right side. We shall only elaborate on the left side – the right side view is implemented similarly.

We shall define a new data structure that represents the view as

```
Inductive View_l (S:Type -> Type) (A:Type): Type :=
| nil_l : View_l S A
| cons_l : A -> S A -> View_l S A.
```

Notice that *a)* it closely mirrors how a list is deconstructed, and *b)* it is generalized over all type constructors as in [1] (although we do not really use this generalization).

To construct a `View_l` from a `fingertree` we must define a function `view_l`. We get a similar (albeit reversed) situation to the definition of `addl` above – viewing base cases and trees with prefixes greater than one is trivial, but in the case of `deep (one x) m sf` we have to reshuffle the deeper tree, since we cannot have empty digits.

Consider the value `deep (one x) m sf` of type `fingertree A`. Then, `m` must have type `fingertree (node A)`. If we want to view it from the left, we should get `cons_l x m'` where `m' : fingertree A`. Thus, we must somehow “flatten” `m`. We can view `m` recursively and pattern match on it.

1. if `view_l m = nil_l` we must construct a `fingertree A` out of the suffix `sf : digit A`. Since `digit` implements `reduce`, we can simply call `to_tree sf`.

2. if `view_l m = cons_l a m'`, then `a : node A` and `m' : fingertree (node A)`. We can then construct our new fingertree using the `deep` constructor, by converting `a` to a `digit A` and using `m'` and `sf` as the deeper tree and suffix respectively.

Thus, we get the following Coq definition for `view_l` (listing 5) :

```
Fixpoint view_l {A:Type} (tr:fingertree A) : View_l fingertree A :=
  match tr with
  | empty                     => nil_l
  | single x                  => cons_l x empty
  | deep (four x y z u) m sf => cons_l x (deep (three x y u) m sf)
  | deep (three x y z)  m sf => cons_l x (deep (two x y) m sf)
  | deep (two x y)      m sf => cons_l x (deep (one x) m sf)
  | deep (one x)        m sf =>
    let tail := match view_l m with
                | nil_l => to_tree sf
                | cons_l a m' => deep (to_digit a) m' sf
                end
    in cons_l x tail
  end.
```

Listing 5: Viewing a `fingertree` from the left.

With `view_l` we can implement standard operations on sequences such as `head` and `tail` easily:

```
Definition head_l {A:Type} (default:A) (tr:fingertree A) : A :=
  match view_l tr with
  | nil_l      => default
  | cons_l x _ => x
  end.

Definition tail_l {A:Type} (tr:fingertree A) : fingertree A :=
  match view_l tr with
  | nil_l       => tr
  | cons_l _ tl => tl
  end.
```

2.5 Mapping over trees

For a finger tree tr : fingertree A we can map over it with a function $f : A \to B$ to get a tr' : fingertree B. In general, things that can be mapped over are described by the `functor` typeclass (listing 6):

```
Class functor F := Functor {
  map : forall {A B : Type}, (A -> B) -> F A -> F B;

  map_id : forall {A : Type} fn (f : F A), (forall x, fn x = x) -> map fn f = f;
  map_comp : forall {A B C : Type} (f : B -> C) (g : A -> B) (x : F A),
      map (fun x => f (g x)) x = (fun x => map f (map g x)) x;
}.
```

Listing 6: The functor typeclass.

Dependent types in Coq allow us to encode the functor laws within the typeclass. This ensures that there are only law-abiding instances of `functor`. Furthermore, you can use the functor laws when proving theorems about `functors`.

The `functor` instances for `node` and `digit` are trivial, so we shall not show them here. The `functor` instance for `fingertree` uses polymorphic recursion over the tree, but is otherwise pretty usual (listing 7).

```
Fixpoint ft_map {A B : Type} (fn : A -> B) (tr : fingertree A) : fingertree B :=
  match tr with
  | empty => empty
  | single a => single (fn a)
  | deep pf m sf =>
    deep (map fn pf)
         (ft_map (map fn) m)
         (map fn sf)
  end.

Instance ft_functor : functor fingertree :=
  {|
    map := @ft_map;
    map_id := ft_map_id;
    map_comp := @ft_map_comp;
  |}.
```

Listing 7: The functor instance for `fingertree`s.

The proofs for the functor laws for finger trees are explained in more detail in section 3.2.

2.6 Appending trees

There is a simple linear time algorithm to append two trees by simply using the `reduce` instance together with `addl` as such:

```
Definition append = fun tr1 tr2 => reducer addl tr1 tr2.
```

However, we can do better than this by exploiting the tree-structure of finger trees.

Appending `empty` to another tree `tr` is just `tr`. Appending a `single x` just reduces to `addl x tr` or `addr tr x`, depending on if the `single` is the first or second argument to the append function. The deeper cases are more complicated. To concatenate two trees `deep pf1 m1 sf2` and `deep pf2 m2 sf2` we can definitely use `pf1` as the prefix of our new tree, and `sf2` as the suffix. Re-constructing the "middle" of tree then boils down to combining `m1`, `sf2`, `pf2` and `m2`.

To do this, we shall use an auxiliary function that combines two trees along with a list of "remainder" values:

```
Fixpoint app3 {A:Type} (tr1:fingertree A) (rem : list A) (tr2:fingertree A).
```

To combine `m1`, `sf2`, `pf2` and `m2`, we can use `app3` with `A := node A, tr1 := m1, tr2 := m2` but how do we get `rem : list (node A)`? We must create such a list by combining `sf1` and `pf2`. In [1], digits are represented by lists so its easy to simply append the digits and then create a partial function to convert them to a list of nodes, as seen in listing 8.

```
nodes :: [a] -> [Node a]
nodes [a,b]            = [Node2 a b]
nodes [a,b,c]          = [Node3 a b c]
nodes [a,b,c,d]        = [Node2 a b,Node2 c d]
nodes (a : b : c : xs) = Node3 a b c : nodes xs
```

Listing 8: Haskell function to aggregate a list of `a`s to a list of `Node` `a`s from [1].

However, our representation of digits statically ensures the invariant that digits hold one to four elements, so they're not lists. Secondly, we cannot write such a partial function in Coq without making proofs about such functions very cumbersome to write. The solution that [1] suggests and [3] implements is a 200-line function that handles all combinations of `sf1` and `pf1` and how to combine them into a list of nodes. However, if we formalize the invariant in `nodes` (listing 8), we can get by with a much more elegant definition.

First, we create an inductive that represents lists created from two, three, four or (three and some more) elements of A (listing 9).

```
Inductive app3_list (A : Type) : Type :=
| app3_two   : A -> A -> app3_list A
| app3_three : A -> A -> A -> app3_list A
| app3_four  : A -> A -> A -> A -> app3_list A
| app3_more  : A -> A -> A -> app3_list A -> app3_list A.
```

Listing 9: Inductive that represents lists created by appending digits with remainders.

Note how the inductive mirrors the branches of `nodes` in listing 8. We can then define how to create such lists from two digits `d1, d2 : digit A` and a list of "remainders" `xs : list A` as such (listing 10:

```
Fixpoint dig_app3 {A:Type} (d1 : digit A) (xs : list A) (d2 : digit A): app3_list A :=
  match d1, xs, d2 with
  | one a,        [], one b        => app3_two a b
  | one a,        [], two b c      => app3_three a b c
  | one a,        [], three b c d  => app3_four a b c d
  | one a,        [], four b c d e => app3_more a b c (app3_two d e)

  | two a b,      [], one c        => app3_three a b c
  | two a b,      [], two c d      => app3_four a b c d
  | two a b,      [], three c d e  => app3_more a b c (app3_two d e)
  | two a b,      [], four c d e f => app3_more a b c (app3_three d e f)

  | three a b c,  [], one d        => app3_four a b c d
  | three a b c,  [], two d e      => app3_more a b c (app3_two d e)
  | three a b c,  [], three d e f  => app3_more a b c (app3_three d e f)
  | three a b c,  [], four d e f g => app3_more a b c (app3_four d e f g)

  | four a b c d, [], one e        => app3_more a b c (app3_two d e)
  | four a b c d, [], two e f      => app3_more a b c (app3_three d e f)
  | four a b c d, [], three e f g  => app3_more a b c (app3_four d e f g)
  | four a b c d, [], four e f g h => app3_more a b c (app3_more d e f (app3_two g h))

  | one a,        (x :: xs), _     => dig_app3 (two a x) xs d2
  | two a b,      (x :: xs), _     => dig_app3 (three a b x) xs d2
  | three a b c,  (x :: xs), _     => dig_app3 (four a b c x) xs d2
  | four a b c d, (x :: xs), _     => app3_more a b c (dig_app3 (two d x) xs d2)
  end.
```

Listing 10: Appending two digits and a list of remainders.

While certainly not succinct, it is significantly shorter than 200 lines of code. Finally, we can implement `nodes` in Coq defined recursively over `app3_list` (listing 11):

```
Fixpoint nodes {A : Type} (xs : app3_list A) : list (node A) :=
  match xs with
  | app3_two a b         => [node2 a b]
  | app3_three a b c     => [node3 a b c]
  | app3_four a b c d    => [node2 a b; node2 c d]
  | app3_more a b c xs' => node3 a b c :: nodes xs'
  end.
```

Listing 11: Implementing `nodes` using recursion over `app3_list`.

This leads us to the final definition of `app3` (listing 12):

```
Definition addl' := reducer addl.
Definition addr' := reducel addr.

Fixpoint app3 {A:Type} (tr1:fingertree A) (rem : list A) (tr2:fingertree A)
  : fingertree A :=
  match tr1, tr2 with
  | empty, _                          => addl' rem tr2
  | _, empty                          => addr' tr1 rem
  | single x, _                       => x <| addl' rem tr2
  | _, single x                       => addr' tr1 rem |> x
  | deep pr1 m1 sf1, deep pr2 m2 sf2 =>
      let a3l := dig_app3 sf1 rem pr2 in
      deep pr1 (app3 m1 (nodes a3l) m2) sf2
  end.
```

Listing 12: Definition of `app3`.

We can then use `app3` to define `append` (and subsequently some notation for it).

```
Definition append {A:Type} (tr1 tr2 : fingertree A) : fingertree A :=
  app3 tr1 [] tr2.

Notation "t1 >< t2" := (append t1 t2)
                   (at level 62, left associativity).
```

2.7 Reversing trees

As with concatenating trees, one can easily create a reverse function using the `reduce` instance for `fingertree`. However, we can again do better by exploiting the structure of the fingertree.

First, consider reversing a 2-3 node:

```
Definition reverse_node {A: Type} (n: node A): node A  :=
  match n with
  | (node2  a b)  => node2 b a
  | (node3 a b c) => node3 c b a
  end.
```

This will work fine for the “second” level of a finger tree, but in the deeper levels it will not give us the intended effects - the inner nodes will not be reversed. Thus, we must thread a function for “inner reversals” through our definitions:

```
Definition reverse_node {A: Type}
          (f: A -> A) (n: node A): node A  :=
  match n with
  | (node2  a b)  => node2 (f b) (f a)
  | (node3 a b c) => node3 (f c) (f b) (f a)
  end.

Definition reverse_digit {A: Type}
         (f: A -> A) (d: digit A): digit A  :=
  match d with
  | one a        => one (f a)
  | two a b      => two (f b) (f a)
  | three a b c  => three (f c) (f b) (f a)
  | four a b c d => four (f d) (f c) (f b) (f a)
  end.
```

Listing 13: Reversing nodes and digits.

In listing 13, the function `f` takes care of reversing whatever is contained within the data structures. With that defined, reversing a `fingertree` can be done simply.

```
Fixpoint reverse_tree {A: Type} (f: A -> A) (tr: fingertree A)
  : fingertree A :=
  match tr with
  | empty        => empty
  | single x     => single (f x)
  | deep pr m sf =>
    deep (reverse_digit f sf)
         (reverse_tree (reverse_node f) m)
         (reverse_digit f pr)
  end.
(** Reverse a tree by calling reverse_tree starting with the identity function *)
Definition reverse {A: Type} : fingertree A -> fingertree A :=
  reverse_tree (fun x => x).
```

Listing 14: Reversing a `fingertree`.

Again, we need polymorphic recursion in the inductive case – in the recursive call `A := node A` and `f := reverse_node f`, thus reversing the nested nodes at the deeper levels of the tree.

3 Proofs

This section will go through a few of the proofs of theorems constructed to verify the correctness of the implementation.

3.1 Tree equivalence

First, we must define a new notion of equivalence for finger trees. Since the same sequence can be represented by several finger trees, syntactic equality is too strict. Instead, we choose to define two trees t_1 and t_2 as being equivalent if their right-reduction with any operator from any accumulator are equal. That leads us to the following definition of tree equality (listing 15):

```
Definition tree_eq {A : Type} (t1 t2 : fingertree A) :=
  forall (B : Type) (acc : B) (op : A -> B -> B),
    reducer op t1 acc = reducer op t2 acc.

Notation "t1 >=< t2" := (tree_eq t1 t2)
                          (at level 60, right associativity).
```

Listing 15: Tree equality.

This should capture the fact that the two trees represent the same sequence.

3.2 Functor laws

As established in section 2.5, finger trees are functors. To instantiate `fingertree` as a functor, we must prove the functor laws which are:

$$\begin{array}{rcc} \mathsf{map}_{id} & : & \mathsf{map\ id} = \mathsf{id} \\ \mathsf{map}_{\circ} & : & \forall f\ g,\ \mathsf{map}\ f \circ \mathsf{map}\ g = \mathsf{map}\ (f \circ g) \end{array}$$

Their Coq translations use the functions fully applied instead, since we want to treat function equality extensionally (although we will need to assume functional extensionality in the proofs anyway).

Proof of $\mathbf{map}_{id}$

Recall the definition of map over fingertrees (listing 16):

```
Fixpoint ft_map {A B : Type} (fn : A -> B) (tr : fingertree A) : fingertree B :=
  match tr with
  | empty => empty
  | single a => single (fn a)
  | deep pf m sf =>
    deep (map fn pf)
         (ft_map (map fn) m)
         (map fn sf)
  end.
```

Listing 16: `map` defined on finger trees.

We shall prove `map_id : forall {A : Type} (f : fingertree A), map (fun x => x) f = f.`

The proof proceeds by induction. The base cases are trivial. In the inductive case, we get

```
deep (map (fun x => x) pf)
     (ft_map (map (fun x => x)) m)
     (map (fun x => x) sf) =
deep pf m sf
```

By applying `map_id` from the functor instance of `digit` twice, we get

```
deep pf (ft_map (map (fun x => x)) m) sf = deep pf m sf
```

By eta conversion, we can rewrite the above as

```
deep pf (ft_map (fun n : node A => map (fun x => x) n) m) sf = deep pf m sf
```

Again, applying `map_id` from the functor instance of `node`, we get

```
deep pf (ft_map (fun n : node A => n) m) sf = deep pf m sf
```

Which, by applying the induction hypothesis, proves our goal.

Proof of map$_\circ$

Our goal is to prove

```
 ft_map_comp {A B C: Type} (f: B -> C) (g: A -> B) (tr: fingertree A) :
     ft_map (fun x => f (g x)) tr = ft_map f (ft_map g tr).
```

The proof proceeds by induction over `fingertree`s. The base cases are trivial.

In the inductive case, we get

```
deep (map (fun x : A => f (g x)) pf)
     (ft_map (map (fun x : A => f (g x))) m)
     (map (fun x : A => f (g x)) sf) =
deep (map f (map g pf))
     (ft_map (map f) (ft_map (map g) m))
     (map f (map g sf))
```

By applying `map_comp` from the functor instance of `digit` twice, we get

```
deep (map f (map g pf))
     (ft_map (map (fun x : A => f (g x))) m)
     (map f (map g sf)) =
deep (map f (map g pf))
     (ft_map (map f) (ft_map (map g) m))
     (map f (map g sf))
```

Again, by eta conversion we can replace the inner map over nodes with `fun n : node A => map (fun x : A => f (g x)) n` which we can rewrite using the `map_comp` lemma from the functor instance of `node`. The resulting term is

```
deep (map f (map g pf))
     (ft_map (fun n : node A => map f (map g n)) m)
    (map f (map g sf)) =
deep (map f (map g pf))
     (ft_map (map f) (ft_map (map g) m))
     (map f (map g sf))
```

which we can prove using the induction hypothesis.

3.3 Converting trees to lists

To verify the implementation of `to_list` (and thus also `reducer` on which it relies), we set out to prove the following theorem:

$$\forall (A : \mathsf{Type})\ (xs \in \mathsf{list}\ A),\ \mathsf{to\_list}\ (\mathsf{to\_tree}\ xs) = xs$$

The theorem states that `to_list` is the inverse of `to_tree` for lists.

The proof is by induction on xs. The base case is trivial. In the inductive case, we get

```
ft_reducer cons (a <| fold_right addl empty xs) [] = a :: xs
```

because `to_list = reducer cons tr []` and `to_tree = reducer addl empty xs` and `reducer` for lists is simply `fold_right`.

We can easily convince ourselves that reducing any tree `a <| tr` from the right with an operator `op` will in one unfolding of `reducer` become `op a (reducer op tr acc)`, and we can use this lemma to rewrite the above to

```
a :: ft_reducer cons (fold_right addl empty xs) [] = a :: xs
```

We can now apply our induction hypothesis (again remembering how `to_list` and `to_tree` are defined) to solve our goal.

3.4 Append

To verify `append` we prove

```
Theorem tree_append_hom : forall {A:Type} (tr1 tr2 : fingertree A),
  to_list (tr1 >< tr2) = to_list tr1 ++ to_list tr2.
```

which in fact states that `to_list` is a monoid homomorphism from $(\mathsf{fingertree}, \mathsf{><}, \mathsf{empty})$ to $(\mathsf{list}, \mathsf{++}, [])$.

However, we cannot prove this directly, as the induction hypothesis becomes to weak. To strengthen it, we must generalize the lemma to the form

```
forall {A:Type} (tr1 tr2 : fingertree A) acc xs,
  reducer op (app3 tr1 xs tr2) =
  reducer op tr1 (reducer op (to_tree xs) (reducer op tr2 acc)).
```

Notice that even `cons` was generalized to any operator. This is necessary because of the non-regular form of the inductive `fingertree`. As we shall also see later, this non-regularity complicates our proves significantly, as we are forced to work in very general terms.

The proof of the proposition above is lengthy and uses several lemmas, so we shall only cover its general structure.

The proof proceeds by induction on `tr1`.

1. In the case where `tr1 = empty` we get

```
ft_reducer op (addl' xs tr2) acc =
ft_reducer op (to_tree xs) (ft_reducer op tr2 acc)
```

Since `addl' xs tr2` simply adds all elements in `xs` to the left of `tr2`, we can convince ourselves that folding over this compound tree is the same as folding over `to_tree xs` with an accumulator produced by right-reducing `tr2` with op from `acc`, which is the right hand side.

2. In the case where `tr1 = single a` we shall have to perform case analysis on `tr2` according to the definition of `app3` (listing 12).

2.1. When `tr2 = empty` we get

```
ft_reducer op (addr' (single a) xs) acc =
op a (ft_reducer op (to_tree xs) acc)
```

which is true based on a dual argument to the case where `tr1 = empty`, with `addl'` replaced by `addr'`.

2.2. When `tr2 = single a0` we get

```
ft_reducer op (a <| addl' xs (single a0)) acc =
op a (ft_reducer op (to_tree xs) (op a0 acc))
```

as our goal which becomes

```
op a (ft_reducer op (addl' xs (single a0)) acc) =
op a (ft_reducer op (to_tree xs) (op a0 acc))
```

by a lemma and is then solved by an argument similar to the previous cases.

2.3. When `tr2 = deep pf m sf` we get

```
ft_reducer op (a <| addl' xs (deep pf m sf)) acc =
op a (ft_reducer op (to_tree xs)
     (digit_reducer op pf
        (ft_reducer (nd_reducer op) m (digit_reducer op sf acc))))
```

which is a bit harder to make sense of, but it can be solved by a similar line of reasoning to the other cases.

3. When `tr1 = deep pf1 m1 sf1` we again do case analysis on `tr2` according to the definition of `app3`.

3.1 When `tr2 = empty` we get a dual case to the very first case in the proof.

3.2 When `tr2 = single a` we get a dual situation to case 2.3.

3.3 When `tr3 = deep pf2 m2 sf2` we get the following goal:

```
digit_reducer op m1
  (ft_reducer (nd_reducer op) (app3 sf1 (nodes (dig_app3 d xs pf2)) m2)
     (digit_reducer op sf2 acc)) =
digit_reducer op m1
  (ft_reducer (nd_reducer op) sf1
     (digit_reducer op d
        (ft_reducer op (to_tree xs)
           (digit_reducer op pf2
              (ft_reducer (nd_reducer op) m2 (digit_reducer op sf2 acc))))))
```

While the goal might seem dauntingly large, it is in fact just an instance of the original goal, which means we can use our induction hypothesis!

After applying the induction hypothesis, and using the fact that functions are deterministic, we get the following goal

```
ft_reducer (nd_reducer op) (to_tree (nodes (dig_app3 sf1 xs pf2)))
  (ft_reducer (nd_reducer op) m2 (digit_reducer op sf2 acc)) =
digit_reducer op sf1
  (ft_reducer op (to_tree xs)
     (digit_reducer op pf2 (ft_reducer (nd_reducer op) m2 (digit_reducer op sf2 acc))))
```

which can be proven by induction over xs. The proposition is proved!

Using this monoid homomorphism `tree_append_hom` we can easily also prove associativity of `><`, i.e. that $\forall tr_1\ tr_2\ tr_3,\ tr_1$ `><` $(tr_2$ `><` $tr_3)$ `>=<` $(tr_1$ `><` $tr_2)$ `><` tr_3.

3.5 Reverse

To verify `reverse` we prove two propositions – `reverse_involutive` and `reverse_to_list`.

Proof of `reverse_involutive`

Our goal is to prove

```
Theorem reverse_involutive {A B : Type} (tr : fingertree A) :
  forall fn (H : forall x, fn (fn x) = x),
        (reverse_tree fn (reverse_tree fn tr)) = tr.
```

Listing 17: Reversing a tree twice gives you back the same tree.

The proof of `reverse_involutive` proceeds by induction on `tr`. Notice the assumption `H : forall x, fn (fn x) = x` – we shall use this in the proof.

When `tr = empty` the goal is trivially true.

When `tr = single a` the goal becomes

```
single (fn (fn a)) = single a
```

Using `H` this is easily proven.

When `tr = deep pf m sf` the goal becomes

```
1 deep (reverse_digit fn (reverse_digit fn pf))
2      (reverse_tree (reverse_node fn) (reverse_tree (reverse_node fn) m))
3      (reverse_digit fn (reverse_digit fn sf)) = deep pf m sf
```

We can use our induction hypothesis to rewrite line 2, yielding the goal

```
1 deep (reverse_digit fn (reverse_digit fn pf))
2      m
3      (reverse_digit fn (reverse_digit fn sf)) =
4 deep pf m sf
```

This can be trivially solved by case analysis on `pf` and `sf` and applying `H`.

Proof of `reverse_to_list`

Our goal is to prove

```
Theorem reverse_to_list {A : Type} (tr : fingertree A) :
  rev (to_list tr) = to_list (reverse tr).
```

Listing 18: Reversing a tree and then converting it to a list is the same as converting the tree to a list and then reversing it.

While proving that reverse is involutive was relatively straightforward, this is (unfortunately) not so for this otherwise innocent looking theorem.

Our problem again lies with the non-regular structure of `fingertree`. We must generalize not only the accumulator of `to_list` and the inner involutive function of `reverse`, but also the operator, which we would like to be simply `cons`. However, mentioning `cons` in the lemma gives us a too weak induction hypothesis, because the operator in the recursive call becomes `nd_reduce cons`. But then, generalizing `cons` to any operator makes the proposition false.

One way to work around this is to create a custom induction principle. We have experimented (with success) with two different induction principles:

3.5.0.1 Factoring out nestedness with dependent types

```
(** Lift a type n times into node *)
Fixpoint node_lift (n:nat) (A:Type) : Type :=
  match n with
  | O => A
  | S n' => node (node_lift n' A)
  end.

Axiom fingertree_lift_ind
   : forall P : (forall (n : nat) (A : Type), fingertree (node_lift n A) -> Prop),
     (forall (n:nat) (A : Type), P n A empty) ->
     (forall (n:nat) (A : Type) (a : node_lift n A), P n A (single a)) ->
     (forall (n:nat) (A : Type) (d : digit (node_lift n A))
        (f1 : fingertree (node_lift (S n) A)),
         P (S n) A f1 -> forall d0 : digit (node_lift n A), P n A (deep d f1 d0)) ->
     forall (n:nat) (A : Type) (f2 : fingertree (node_lift n A)), P n A f2.
```

Listing 19: The `fingertree_lift_ind` induction principle for `fingertree`.

`fingertree_lift_ind` (listing 19) does induction over `fingertree`s, but uses dependent types to factor out the nested definition of `fingertree`s by indexing over the nested type using a natural number. This allows us to talk about `cons` directly as such:

```
Lemma reverse_rev_ft {A : Type} (n : nat) (tr : fingertree (node_lift n A)) :
  forall (acc acc' : list A),
    rev acc' ++ ft_reducer (nd_red_lift n cons)
        (reverse_tree (rev_lift n ident) tr) acc =
    rev (ft_reducer (nd_red_lift n cons) tr acc') ++ acc.
```

As can be seen, everything must be expressed as lifted n times into the `node` type constructor. It makes the proposition a bit unwieldy to express and, sure enough, the proof is also less than elegant. Additionally, the fact that the custom induction principle must be axiomatized potentially opens the door to inconsistencies.

3.5.0.2 Induction over finger trees as sequences

A better induction principle recaptures the fact that finger trees represent sequences, and as such every `tr : fingertree` can be expressed as `empty` or `x <| tr'`. This is essentially the same concept that `view_l` captures, however it is not easy to convince Coq that such a view in fact covers `fingertree`. However, we can assert this fact by creating a custom induction principle as such:

```
Axiom ft_left_ind :
  forall (P : (forall (A : Type), fingertree A -> Prop)),
         (forall (A : Type), P A empty) ->
         (forall (A : Type) (tr : fingertree A) (x : A), P A tr -> P A (x <| tr)) ->
         forall (A : Type) (tr : fingertree A), P A tr.
```

Figure 2: Induction principle for induction over finger trees by viewing them as left-constructed sequences (basically lists).

Using the sequential induction principle in Figure (2), we can write the general form as

```
Theorem reverse_reducer {A : Type} (tr : fingertree A) :
  forall acc acc' fn, (forall x, fn (fn x) = x) ->
                  rev acc' ++ reducer cons (reverse_tree fn tr) acc =
                  rev (reducer cons (map fn tr) acc') ++ acc.
```

Notice that while the accumulators are generalized, we can mention `cons` directly because our custom induction principle has eliminated the nested `node` type application by viewing finger trees as sequences. Proving the above is then quite simple.

3.5.0.3 Using Coq's standard induction principle

While the above approaches enables us to side-step the non-regularity of the `fingertree` inductive, the downside is that they both potentially introduce inconsistency into our logic since they're axioms. It turns out, we can in fact prove `reverse_to_list` using the standard induction principle that Coq generates for `fingertree`! The key is to accurately capture the way `cons`, `nd_reducer cons`, `nd_reducer (... nd_reducer cons)` interacts with `rev` in an assumption.

```
Lemma reverse_reducer
        {A B : Type} (tr : fingertree A) :
  forall acc acc' (fn : A -> A) (op : A -> list B -> list B)
    (H : forall x, fn (fn x) = x)
    (H0 : forall a acc acc', rev acc' ++ op (fn a) acc = rev (op a acc') ++ acc),
    rev acc' ++ reducer op (reverse_tree fn tr) acc =
    rev (reducer op tr acc') ++ acc.
```

Figure 3: The generalized form of `reverse_to_list` proven using the standard induction hypothesis.

Here, `H0` expresses that `op` is an operator that behaves like `cons` in the context of `rev` and `fn`. If you look closely, you can in fact see that the conclusion is identical to `H0` except that `a = tr, fn = reverse_tree fn, op = reducer op`. To prove listing 3, we'll also need to prove the same form for nodes:

```
rev acc' ++ nd_reducer op (reverse_node fn n) acc =
rev (nd_reducer op n acc') ++ acc.
```

and that `node_reverse fn` is involutive if `fn` is involutive, but these lemmas are easily proven. In the end, proving the theorem using standard induction was in fact not that hard, but finding the correct constraints to put on `op` was very challenging!

4 Related work

There exists multiple verified implementations of finger trees.

In 2010, Nordhoff, Körner and Lammich published a certified version of finger trees in Isabelle [4]. This work is also based on [1].

Matthieu Sozeau proved in 2007 the correctness of Hinze and Patterson's implementation [1] of Finger Trees [3] using Coq and Russell, a programming language created by himself (and later integrated into Coq as "Program"). Sozeau describes the impediments that the lack of distinction between propositions and types can lead to when proofs and computations are combined, namely that proofs are only useful at compile time and thus must be elided from the executable code. A solution he mentions is given by the Prop/Type distinction in Coq, where Prop is used for the type of propositions and Type as the type of computational types, but the drawback is that as soon a proposition appears in the type of a function, you have two options: *a)* Implement the function using LTac, thereby making the proofs ergonomic but the computational aspects in-elegant (and the generated body will be near-unreadable) or *b)* Implement the function in Gallina, which is nice for functional programming, but is awkward for proofs. Neither option is satisfactory.

Using the `Program` vernacular, Coq uses the Russel type checker that allows writing algorithmic code of strongly specified functions and forgetting about proofs required by Coq that need to be established as correct.

Its principal purpose is to allow the user to use structural induction, complete pattern matching and total functions only in a language that compiles into CIC, which can be type checked by Coq.

The program will type check in Russel only if the specification is structurally well-typed and proofs are not going to be relevant at the type-checking step but later, when recombining obligations with the algorithmic skeleton that was written.

Sozeau presents finger trees as follows, where `v` is a type that carries a Monoid structure :

```
Inductive fingertree A (measure : A -> v) : v -> Type :=
 | Empty : fingertree measure epsilon
 | Single : forall x : A, fingertree measure (measure x)
 | Deep : forall (l : digit A) (ms : v),
   fingertree (A:=node measure) (node_measure measure) ms
   -> forall r : digit A, fingertree measure
     (digit_measure measure l · ms · digit_measure measure r).
```

The main difference with our implementation is that Sozeau implements measures, as we can see in this definition. Measures in Finger Trees are annotations represented by a reduction with some Monoid. The measure is usually cached in nodes and the "deep" constructor of finger trees. Measures are typically used to specialize finger trees to certain domains - e.q. priority queues or random access sequences. For example, elements in the tree can be measured with the function that constantly returns 1. The cached measure is thus the number of elements in the tree, yielding a length function in constant time. Such a measure can also be used to give logarithmic random access into a finger tree. As can be seen, the measure is promoted into the type level in Sozeau's implementation.

In the Coq definition of a finger tree above, the measure of the tree is promoted into the type-level. Therefore, the measure can be used to prove invariants about operations on trees. An obvious example is that appending two trees with measure m_1 and m_2 respectively, we get a new tree with measure $m_1 \cdot m_2$, where $\cdot$ is the monoidal operation.

The measure on finger trees is also used by Sozeau to ensure that defined recursions on the view of a tree are just as valid as those defined by recursion on the tree itself.

If we take `tree_size` as the measure, we could prove that ∀ `(t :fingertree) view_L t` returns a view of size `tree_size t` to prove the correctness of a recursive call on the tail of a view. Unfortunately this is not so simple in this dependently typed implementation either, because heterogeneous equality (the aptly named John Major equality in Coq) [5] is necessary to compare two objects of (potentially) different types.

Sozeau's `digit` is very close to our definition, which at the same time, follows the same Haskell implementation presented by Hinze and Paterson, but in Sozeau's implementation, functions on `digit` also require a proof.

The author shows how properties are defined in arguments, for instance, to define `add_digit_left`. First, a property `full` is defined:

```
Definition full (x : digit) : Prop :=
match x with Four _ _ _ _ => True | _ => False end.
```

Then, `add_digit_left` takes an element of type `a` and a `digit` with the property: "a digit has to be not full" declared in its type:

```
Program Definition add_digit_left (a : A)
  (d : digit | ~full d) : digit :=
    match d with
    | One x => Two a x
    | Two x y => Three a x y
    | Three x y z => Four a x y z
    | Four_ _ _ _ => !
    end.
```

In the pattern match the case four is marked with an exclamation mark (!) to assert that a branch is inaccessible. The property `¬full d` is used to prove that this branch will never be reached.

As mentioned previously, `append` is defined also using dependent types which states the function specification. The definition generates 100 obligations related to measures. Five mutually recursive functions define `app` and they are quite verbose (over 200 lines in total) because a specialisation of concatenation has to be defined for each combination of the digits at the right of the left tree and the left of the right tree.

Summarising, we use a different approach than [3] in several ways. Our implementation is closer in spirit to the original by Hinze and Paterson [1]. However, our implementation is also incomplete, since we have not incorporated measures. Sozeau's use of dependent types gives rise to a different style of verification, sometimes referred to as "correct by construction", where the invariants are specified directly in the types of definitions. These invariants (ideally) guarantee that the definitions cannot be typed without being correct. Our approach to verification is more in the vein of "post-hoc" verification, where the definitions are simply typed and the invariants stated and proven separately. Both approaches have their strengths and weaknesses – Sozeau's approach gives vastly more complicated types which can be hard to work with. On the other hand, the fact that the invariants are encoded directly into the data structures does allow for more succinct and elegant proofs. On the other hand, preserving the sharp distinction between propositions and types also yields significantly simpler definitions that can be a lot easier to debug.

While Sozeau's implementation does in fact implement measures, and is thus a bit more complete, we have also contributed new proofs about reversing finger trees and mapping over finger trees. Furthermore, we've proven tree append correct in terms of a monoid homomorphism to lists, and shown how to use an intermediate data structure to significantly shorten the implementation of `app3`.

5 Conclusion

In conclusion, we have implemented and verified an implementation of finger trees. Our implementation includes a verified reversal function, which interestingly the original Haskell paper [1] does not mention, and accordingly neither [1] nor [4] touch on the correctness of it. However, in the Haskell implementation of `Data.Sequence`, which is used in production today, we found the `reverse` function, which is the one we based the implementation presented in this project on.

We consider the main contribution of our work to be the certification of an implementation of `reverse` for finger trees, plus a new approach to implementing a verifiable `append` function, which is more elegant and significantly shorter than the existing approaches.

References

[1] R. Hinze and R. Paterson, “Finger trees: A simple general-purpose data structure,” *Journal of Functional Programming*, vol. 16, no. 2, pp. 197–217, 2006.

[2] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, “A new representation for linear lists,” in *Proceedings of the ninth annual ACM symposium on Theory of computing*, pp. 49–60, ACM, 1977.

[3] M. Sozeau, “Programming finger trees in coq,” in *ACM SIGPLAN Notices*, vol. 42, pp. 13–24, ACM, 2007.

[4] B. Nordhoff, S. Körner, and P. Lammich, “Finger trees,” *Archive of Formal Proofs*, Oct. 2010. http://isa-afp.org/entries/Finger-Trees.shtml, Formal proof development.

[5] C. McBride, “Elimination with a motive,” in *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES ’00, (London, UK, UK), pp. 197–216, Springer-Verlag, 2002.