Formalising Finger Trees in Agda

Submitted April 2016, in partial fulfilment of
the conditions of the award of the degree **BSc (Hons) Computer Science.**

Zimu QIN
4230482
School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature ________________________

Date _____/_____/_____

Abstract

Finger trees are a persistent functional data structure based on an optimization of 2-3 trees[3] supporting access to the ends in amortized constant time, and concatenation and splitting in time logarithmic in the size of the smaller piece [5]. This project aims to formalise finger trees in Agda, which is a dependently typed functional programming language and proof assistant. This report consists of an introduction of the data structure, some background knowledge, design ideas, preliminaries, implementation in Agda, and some further discussion.

Acknowledgements

I would like to thank my supervisor, Dr. Thorsten Altenkirch sincerely for helping me and giving me advices whenever I need. Without his help, this dissertation would never be possibly done.

Contents

1 Introduction

The project aims to formalise finger trees in Agda[1], a dependently typed functional programming language and proof assistant. While in functional language, lists are used mostly as it is simple to use and well-supported, sometimes the using of lists can be inefficient and inappropriate. Thus, sometimes other data structures need to be implemented[7]. However, such data structures are often complex and difficult to implement, so it is better to introduce a general-purpose data structure, which can serve other complex data structures in a relatively easy way.

Finger trees are a persistent functional data structure based on an optimisation of 2-3 trees[3] supporting access to both ends in amortised constant time, and concatenation and splitting in time logarithmic in the size of the smaller piece[5]. Finger trees and operations on them are relatively simple and easy to implement and they can serve efficient implementations of many abstract data structures, like sequences and priority queues. Using the functional language Agda, it is possible to not only implement finger trees and operations on them, but also prove the invariants of finger trees.

2 Background

2.1 Related Work

The data structure was first introduced in the paper **Finger Trees: A Simple General-purpose Data Structure**.

First, we will introduce a tree called 2-3 tree. A 2-3 tree is either a leaf, or a node of two 2-3 trees of same height (called 2-node), or a node of three 2-3 trees of same height (called 3-node). An example is given as below:

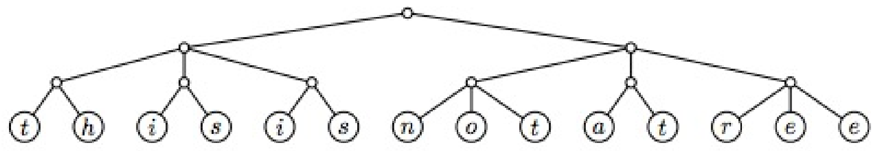

Figure 1: An Example of 2-3 Trees [5]

A finger tree is a 2-3 tree with quick access points at both ends (also called fingers [4]). To get the structure, imagine we recursively take the parent of the leftmost leaf and the parent of the rightmost leaf to combine a new node, which leads to a tree which is essentially the idea of finger trees. The correspondence finger tree of the above 2-3 tree is like below:

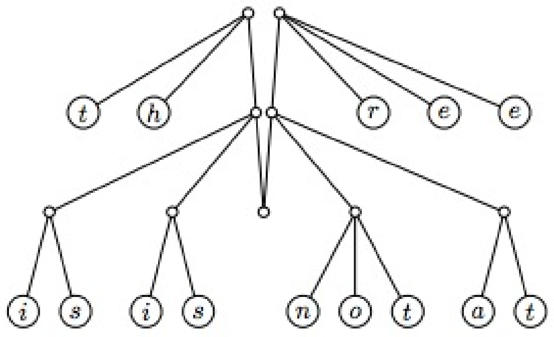

Figure 2: An Example of Finger Trees [5]

According to the paper, the basic structure can be defined as below in Haskell, another functional programming language [5]:

```
data FingerTree a = Empty
   | Single a
   | Deep (Digit a) (FingerTree (Node a)) (Digit a)
data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node a = Node2 a a | Node3 a a a
```

Now the standard library *Data.Sequence* in Haskell has used this data structure as an application of finger trees.

As the Haskell implementation does not guarantee the coherent parameterisation and specialisation of finger trees, let alone the correctness of their implementation, Sozeau later introduced the certified implementation of finger trees in COQ, a proof assistant [9].

Finger trees are a very powerful application of monoids. With the different choices for the monoid related to the data structure, other data structures could be implemented. Thus, it can be used as a framework to implement other complex data structures.

In this dissertation, we will follow the Haskell implementation, and with the help of dependent data types, we can implement a certified version of finger trees in Agda.

2.2 Agda

Some library data types and functions are used in this project, thus in this subsection, I will introduce some of them which are important afterwards. For simplicity, we will remove the universe polymorphism from the definitions to make them look clearer, as it is not related to this project.

2.2.1 Natural Numbers

Natural numbers are the non-negative integers. In Agda's standard library *Data.Nat*, it is defined recursively. Every natural number is either zero or the successor of another natural number. The definition is given as below:

```
data ℕ : Set where
  zero : ℕ
  suc : (n : ℕ) → ℕ
```

For example, the natural number 2 is the successor of 1 which is the successor of 0, so it can be expressed as $suc\,(suc\,zero)$.

The plus operator + can be easily implemented using recursion:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

2.2.2 Lists

Lists are the most common data structure used in functional programming for its simplicity. The definition in Agda's standard library *Data.List* is defined recursively. Every list is either empty or a head element attached with its tail. The definition is given as below:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : (x : A) (xs : List A) → List A
```

For example, the list with element 1 and 2 in this order can now be expressed as 1 :: (2 :: []).

Then, the concatenation operator ++ is trivial to be implemented using recursion. The definition in the library is given as below:

```
_++_ : {A : Set} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

2.2.3 Maybe

Maybe is a simple type which can be used when the value may not exist. The constructor *just* is used to wrap a value, while *nothing* is used when there is no values.

```
data Maybe (A : Set) : Set where
  just : (x : A) → Maybe A
  nothing : Maybe A
```

2.2.4 Booleans

Booleans [8] are the data type only having two values, true and false which represents truth values of logic and Boolean algebra.

```
data Bool : Set where
  true : Bool
  false : Bool
```

2.2.5 Product

Product is simply a mix with two different values, similar to the type *Pair* in Haskell:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

2.2.6 Propositional Equality

In Agda, the propositional equality [2] between two values of the same type is defined using a simple data type defined in *Relation.Binary.PropositionalEquality*:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Notice that there is only one constructor $refl$ for the data type when the input values equal, which means if two values x and y of the same type are unequalled, $x \equiv y$ is still legal, as it matches the type, but it would lead to the empty set, as no constructors can be used on them. The constructor $refl$ also show that the propositional equality is reflexive.

Propositional equality is symmetric, and transitive which can be easily proven by simple pattern matching:

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Substitutivity is an important concept of equality, which shows that in any type, we can replace a term in it with another equalled value, while the meaning of the type is still the same. That is for all x and y, if x equals y, and $P\,x$ holds, then $P\,y$ also holds. It can also be proven using pattern matching:

```
subst : {A : Set} {x y : A} (P : A → Set) → x ≡ y → P x → P y
subst P refl p = p
```

Congruence is also important which shows that all functions respect the propositional equality, i.e. if f is applied on two propositionally equalled values, then the results are also propositionally equalled. It is useful when we need to apply equality in a sub-term of the equalled values. It can be proven using pattern matching:

```
cong : {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

2.2.7 Heterogeneous Equality

While propositional equality compares two values of the same type, heterogeneous equality compares two values of the different types. It is useful when Agda is not smart enough to see the same types. For example, if we substitute the value in a dependent data type with an equalled value, the output type is not the same as the input type, although ideally they should be the same. The definition of heterogeneous equality is similar to propositional equality, as defined in *Relation.Binary.HeterogeneousEquality*:

```
data _≅_ {A : Set} (x : A) : {B : Set} → B → Set where
  refl : x ≅ x
```

Then, we can prove that after the substitution, the value is heterogeneously equalled to the value before substitution:

```
≡-subst-removable : {A : Set} (P : A → Set) {x y : A}
  (eq : x ≡ y) (z : P x) →
  subst P eq z ≅ z
≡-subst-removable P refl z = refl
```

3 Design

In this section, we will talk about the design ideas behind independent finger trees. Unlike Haskell, we could make the use of independent data type to get rid of the uses of nested data type and smart constructors to make the definition clearer.

3.1 Simple Sequences

In this section, we would talk about the design of the data structure. Recall that the basic structure of Haskell implementation is:

```
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)

data Digit a = One a | Two a a | Three a a a | Four a a a a

data Node a = Node2 a a | Node3 a a a
```

We can see that the definition of the finger trees uses nested data type to make sure the level of the current node of finger trees matches the height of fast-accessed 2-3 trees at both ends. For example, the tree of level zero would have the type $FingerTree\,a$ while the tree of level one would have the type $FingerTree\,(Node\,a)$. While it can also be done by using *Agda*, we could use dependent data type to produce a equivalent but clearer definition.

We can first redefine the 2-3 node by using inference rules:

$$\frac{x : A}{Leaf\ x\ : Node\ A\ zero}$$

$$\frac{x\ y\ : Node\ A\ n}{Node2\ x\ y\ : Node\ A\ (suc\ n)}$$

$$\frac{x\ y\ z\ : Node\ A\ n}{Node3\ x\ y\ z\ : Node\ A\ (suc\ n)}$$

As we can see, instead of nested the data type, we could add a natural number which the type of the node depend on. We also add a new constructor *Leaf* to wrap the element to get a node with depth zero.

Then, we could use it to redefine the finger tree type:

$$\frac{}{Empty\ : FingerTree\ A\ n}$$

$$\frac{x\ : Node\ A\ n}{Single\ x\ : FingerTree\ A\ n}$$

$$\frac{pr\ sf\ : Digit\ (Node\ A\ n) \quad m\ : FingerTree\ A\ (suc\ n)}{Deep\ pr\ m\ sf\ : FingerTree\ A\ n}$$

As we can see, the natural number cached in the data type *FingerTree* represents the level of the tree. According to the inference rule, we use this natural number to make sure that the level of the current node of finger trees equals the height of fast-accessed 2-3 trees at both ends.

3.2 Annotated Sequences

As for annotated sequences, we could get rid of the use of smart constructors, using cached monoid values. As for digits, we redefine it to only accept nodes as stored values.

First, we redefine the 2-3 nodes by using inference rules:

$$\frac{x : A}{\mathit{Leaf}\ x\ : \mathit{Node}\ A\ (\mathit{measure}\ x)\ \mathit{zero}}$$

$$\frac{x\ : \mathit{Node}\ A\ v_1\ n \quad y\ : \mathit{Node}\ A\ v_2\ n}{\mathit{Node2}\ x\ y\ : \mathit{Node}\ A\ (v_1 \oplus v_2)\ (\mathit{suc}\ n)}$$

$$\frac{x\ : \mathit{Node}\ A\ v_1\ n \quad y\ : \mathit{Node}\ A\ v_2\ n \quad z\ : \mathit{Node}\ A\ v_3\ n}{\mathit{Node3}\ x\ y\ z\ : \mathit{Node}\ A\ (v_1 \oplus v_2 \oplus v_3)\ (\mathit{suc}\ n)}$$

Then, we can redefine digits, which are now lists of 2-3 nodes with the same height, but maybe different monoid values:

$$\frac{a\ : \mathit{Node}\ A\ v_1\ n}{\mathit{One}\ a\ : \mathit{Digit}\ A\ v_1\ n}$$

$$\frac{a\ : \mathit{Node}\ A\ v_1\ n \quad b\ : \mathit{Node}\ A\ v_2\ n}{\mathit{Two}\ a\ b\ : \mathit{Digit}\ A\ (v_1 \oplus v_2)\ n}$$

$$\frac{a\ : \mathit{Node}\ A\ v_1\ n \quad b\ : \mathit{Node}\ A\ v_2\ n \quad c\ : \mathit{Node}\ A\ v_3\ n}{\mathit{Three}\ a\ b\ c\ : \mathit{Digit}\ A\ (v_1 \oplus v_2 \oplus v_3)\ n}$$

$$\frac{a\ : \mathit{Node}\ A\ v_1\ n \quad b\ : \mathit{Node}\ A\ v_2\ n \quad c\ : \mathit{Node}\ A\ v_3\ n \quad d\ : \mathit{Node}\ A\ v_4\ n}{\mathit{Four}\ a\ b\ c\ d\ : \mathit{Digit}\ A\ (v_1 \oplus v_2 \oplus v_3 \oplus v_4)\ n}$$

After that, we could finally redefine finger trees. Note that the type of prefix and suffix is no longer *Digit* (*Node A n*) because of the change above.

$$\frac{}{\mathit{Empty}\ : \mathit{FingerTree}\ A\ \emptyset\ n}$$

$$\frac{x\ : \mathit{Node}\ A\ v\ n}{\mathit{Single}\ x\ : \mathit{FingerTree}\ A\ v\ n}$$

$$\frac{pr\ : \mathit{Digit}\ A\ v_1\ n \quad m\ : \mathit{FingerTree}\ A\ v_2\ (\mathit{suc}\ n) \quad sf\ : \mathit{Digit}\ A\ v_3\ n}{\mathit{Deep}\ pr\ m\ sf\ : \mathit{FingerTree}\ A\ (v_1 \oplus v_2 \oplus v_3)\ n}$$

4 Preliminaries

Before starting to implement the finger tree, there are some preliminaries that need to be done first.

4.1 Monoids

Monoids[6] are the key to build other data structures on top of finger trees as different choices of monoids would result in different measured values cached in finger trees which make it possible to allow different behaviour when operating on finger trees.

To define the new record type for monoid, first a new record type for checking monoid laws needs to be defined. As a monoid is a type with an identity element and an associative operation, the record type takes an identity element and the associative operation and use function *id*1 and *id*2 to check the property of identity element and use *assoc* to check the associativity of the operation:

```
record IsMonoid {A : Set} (∅ : A) (_⊕_ : A → A → A) : Set where
  field
    id1 : (x : A) → x ≡ x ⊕ ∅
    id2 : (x : A) → x ≡ ∅ ⊕ x
    assoc : (x y z : A) → (x ⊕ y) ⊕ z ≡ x ⊕ (y ⊕ z)
```

Then, the new record type for monoids can be defined as below:

```
record Monoid (A : Set) : Set where
  field
    ∅ : A
    _⊕_ : A → A → A
    isMonoid : IsMonoid ∅ _⊕_
```

Then, for example, we can easily define the monoid formed by natural numbers under addition.

```
monoidExample : Monoid ℕ
monoidExample = record {∅ = zero; _⊕_ = _+_;
  isMonoid = record {id1 = id1′; id2 = λ _ → refl;
    assoc = assoc′}}
    where
      id1′ : (x : ℕ) → x ≡ x + zero
      id1′ zero = refl
      id1′ (suc x) = cong suc (id1′ x)
      assoc′ : (x y z : ℕ) → (x + y) + z ≡ x + (y + z)
      assoc′ zero y z = refl
      assoc′ (suc x) y z = cong suc (assoc′ x y z)
```

4.2 Reductions

Reduce [5] is a type that defines a group of data structures which can be reduced to a summary value. To define this record type, function *reducer* is used to right-associative reduce of the structure while *reducel* is used to left-associative reduce:

```
record Reduce (F : Set → Set) : Set₁ where
  field
    reducer : {A : Set} {B : Set} → (A → B → B) → F A → B → B
    reducel : {A : Set} {B : Set} → (B → A → B) → B → F A → B
```

For the reducible type list, the reduce operations has already been defined by *foldr* and *foldl* in the standard library **Data.List**, but the order of arguments of *foldr* needs to be swapped to match the type of function *reducer*:

```
reduceList : Reduce List
reduceList = record {reducer = λ f xs z → foldr f z xs;
  reducel = foldl}
```

Now this type could be used for converting between data structures. A function *toList* is defined to convert any reducible type to list:

```
toList : {F : Set → Set} {{r : Reduce F}} {A : Set} → F A → List A
toList s = reducer _::_ s []
```

4.3 Measurements

Measured is a simple type that defines the many-to-one mapping from one set of values to another set of values. The function *measure* provides the way of mapping.

```
record Measured (A : Set) (V : Set) : Set where
  field
    measure : A → V
```

Then, for example, we can define a mapping from lists to natural numbers which is to measure the length of the list:

```
measuredExample : {A : Set} → Measured (List A) ℕ
measuredExample = record {measure = length}
```

5 Implementation

5.1 Simple Sequences

5.1.1 2-3 Trees

A 2-3 tree is a tree whose nodes have 2 or 3 children, and leaves have the same level. To ensure all leaves have the same level, a natural number is cached to record the height of the current node of 2-3 trees. Only trees that had the same height can be the sub-trees of other trees. A leaf is simply a 2-3 tree with height zero.

```
data Node (A : Set) : ℕ → Set where
  Node2 : {n : ℕ} → Node A n → Node A n → Node A (suc n)
  Node3 : {n : ℕ} → Node A n → Node A n → Node A n → Node A (suc n)
  Leaf : A → Node A zero
```

5.1.2 Finger Trees

A finger tree is a tree that has quick access points (also called fingers) which form 2-3 trees at both ends and the height of the 2-3 tree has the same level as the current node in finger trees to provide a fast access to both ends.

The new data type *Digit* is defined to provide finger trees quick access points at both ends:

```
data Digit (A : Set) : Set where
  One   : A → Digit A
  Two   : A → A → Digit A
  Three : A → A → A → Digit A
  Four  : A → A → A → A → Digit A
```

A natural number is cached in finger trees to record the level of the current node of finger trees and make sure it matches the height of fast-accessed 2-3 trees at both ends:

```
data FingerTree (A : Set) (n : ℕ) : Set where
  Empty  : FingerTree A n
  Single : Node A n → FingerTree A n
  Deep   : Digit (Node A n) → FingerTree A (suc n)
           → Digit (Node A n) → FingerTree A n
```

Then, the new data types could be recorded in *Reduce* as they are all reducible. First, reduction on digits is defined as it is trivial (For simplicity, only the definition of right-associative reduction is shown):

```
reducerDigit : {A : Set} {B : Set} → (A → B → B)
  → Digit A → B → B
reducerDigit _≺_ (One  a)       z  =  a ≺ z
reducerDigit _≺_ (Two  a b)     z  =  a ≺ (b ≺ z)
reducerDigit _≺_ (Three a b c) z  =  a ≺ (b ≺ (c ≺ z))
reducerDigit _≺_ (Four a b c d) z  =  a ≺ (b ≺ (c ≺ (d ≺ z)))
```

Reduction on nodes would be slightly different from the existing Haskell implementation. While Haskell used nested data types to make sure the level of finger trees and the height of 2-3 trees matches, in Agda, a cached value is used to record height. Thus, reductions on nodes have to be defined recursively:

```
reducerNode : {A : Set} {B : Set} {n : ℕ} → (A → B → B)
  → Node A n → B → B
reducerNode _≺_ (Node2 a b) z  =  reducerNode _≺_ a
                                  (reducerNode _≺_ b z)
reducerNode _≺_ (Node3 a b c) z  =  reducerNode _≺_ a
                                  (reducerNode _≺_ b
                                    (reducerNode _≺_ c z))
reducerNode _≺_ (Leaf x)      z  =  x ≺ z
```

Reduction on finger trees is a little tricky and different from the existing Haskell implementation. As a quick access point is some 2-3 trees wrapped with the data type *Digit*, the prefix and suffix of the finger tree needs to be reduced on nodes and then on digits. For simplicity and clarity, function composition is used to provide an elegant definition:

```
reducerFingerTree : {A : Set} {B : Set} {n : ℕ} → (A → B → B)
  → FingerTree A n → B → B
reducerFingerTree _≺_ Empty     z  =  z
reducerFingerTree _≺_ (Single x) z  =  reducerNode _≺_ x z
reducerFingerTree _≺_ (Deep pr m sf) z  =
  (reducerDigit ∘ reducerNode) _≺_ pr
    (reducerFingerTree _≺_ m
      ((reducerDigit ∘ reducerNode) _≺_ sf z))
```

Then, we could prove that reductions on finger trees would produce the same output as the equivalent lists. We begin this with the simple proof that reductions on digits would produce the same output as the equivalent lists. It is easy to prove, as digits are defined non-recursively, and only have four cases. Thus, we only need to pattern match on the all four cases:

```
lemmaDigit : {A B : Set}
  (d : Digit A) (op : A → B → B) (z : B) →
  reducer {{reduceDigit}} op d z ≡
  reducer {{reduceList}} op (toList {{reduceDigit}} d) z
lemmaDigit (One a) op z = refl
lemmaDigit (Two a b) op z = refl
lemmaDigit (Three a b c) op z = refl
lemmaDigit (Four a b c d) op z = refl
```

When it comes to the proof of nodes and finger trees, it is not so easy to prove, as they are defined recursively. Thus, we need to use several recursive proofs and helper lemmas. First, we need two simple lemmas of lists:

```
listAssoc : {A : Set} (xs ys zs : List A) →
    (xs ++ ys) ++ zs ≡ xs ++ ys ++ zs
listAssoc [] ys zs = refl
listAssoc (x :: xs) ys zs = cong (_::_ x) (listAssoc xs ys zs)
lemmaList : {A B : Set} (xs : List A) (ys : List A)
  (op : A → B → B) (z : B) →
    reducer {{reduceList}} op xs
    (reducer {{reduceList}} op ys z) ≡
    reducer {{reduceList}} op (xs ++ ys) z
lemmaList [] ys op z = refl
lemmaList (x :: xs) ys op z = cong (op x) (lemmaList xs ys op z)
```

Then, we need another proof of nodes. For simplicity, we only include the proof for 2-node. The proof for 3-node is similar (here we use the module
$\equiv -Reasoning$ in standard library to provide a clearer step-by-step equality proof started by the keyword *begin*, and ended by the Q.E.D symbol ■):

```
lemmaNode1 : {A : Set} {n : ℕ} (a : Node A n) (xs : List A) →
  reducerNode _::_ a xs ≡ reducerNode _::_ a [] ++ xs
lemmaNode1 (Node2 a b) xs =
  begin
    reducerNode _::_ a (reducerNode _::_ b xs) ≡⟨
    cong (reducerNode _::_ a) (lemmaNode1 b xs) ⟩
    reducerNode _::_ a (reducerNode _::_ b [] ++ xs) ≡⟨
    lemmaNode1 a (reducerNode _::_ b [] ++ xs) ⟩
    reducerNode _::_ a [] ++ reducerNode _::_ b [] ++ xs ≡⟨
    sym (listAssoc (reducerNode _::_ a []) (reducerNode _::_ b []) xs) ⟩
    (reducerNode _::_ a [] ++ reducerNode _::_ b []) ++ xs ≡⟨
    cong (flip _++_ xs) (sym (lemmaNode1 a (reducerNode _::_ b []))) ⟩
    reducerNode _::_ a (reducerNode _::_ b []) ++ xs ■
```

With the aid of above lemmas, we could finally prove that reductions on nodes would produce the same output as the equivalent lists:

```
lemmaNode2 : {n : ℕ} {A B : Set}
  (node : Node A n) (op : A → B → B) (z : B) →
  reducer {{reduceNode}} op node z ≡
  reducer {{reduceList}} op (toList {{reduceNode}} node) z
lemmaNode2 (Node2 a b) op z =
  begin
    reducerNode op a (reducerNode op b z) ≡⟨
    lemmaNode2 a op (reducerNode op b z) ⟩
    reducer {{reduceList}} op (toList {{reduceNode}} a)
    (reducerNode op b z)
    ≡⟨
    cong (reducer {{reduceList}} op (toList {{reduceNode}} a))
    (lemmaNode2 b op z)
    ⟩
    reducer {{reduceList}} op (toList {{reduceNode}} a)
    (reducer {{reduceList}} op (toList {{reduceNode}} b) z)
    ≡⟨
    lemmaList (toList {{reduceNode}} a) (toList {{reduceNode}} b) op z
    ⟩
    reducer {{reduceList}} op
    (reducerNode _::_ a [] ++ reducerNode _::_ b []) z
    ≡⟨
    cong (λ x → reducer {{reduceList}} op x z)
    (sym (lemmaNode1 a (reducerNode _::_ b [])))
    ⟩ reducer {{reduceList}} op (toList {{reduceNode}} (Node2 a b)) z ∎
```

Similarly, we could also prove reductions on finger trees would produce the same output as the equivalent lists. For more details, please see the code.

5.1.3 Deque Operations

Adding a 2-3 tree to the left (or right) of the finger trees whose level equals the height of the 2-3 tree is trivial except when the prefix has already been full, in which case, three elements are wrapped into a new 2-3 tree and pushed into the next level of the finger tree and the other two are left in the prefix.

```
infixr 5 _◁_
_◁_ : {A : Set} {n : ℕ} → Node A n → FingerTree A n → FingerTree A n
a ◁ Empty    = Single a
a ◁ Single b = Deep (One a) Empty (One b)
a ◁ Deep (One b)         m sf = Deep (Two a b) m sf
a ◁ Deep (Two b c)       m sf = Deep (Three a b c) m sf
a ◁ Deep (Three b c d)   m sf = Deep (Four a b c d) m sf
a ◁ Deep (Four b c d e) m sf = Deep (Two a b) (Node3 c d e ◁ m) sf
```

Then, the lifting for ◁ and ▷ operators could be defined. The definitions are slightly different from the Haskell implementation as we had to make all the leaves 2-3 trees by using the constructor *Leaf*:

```
_◁′_ : {F : Set → Set} {{r : Reduce F}} {A : Set} → F A
  → FingerTree A zero → FingerTree A zero
_◁′_ = reducer (_◁_ ∘ Leaf)


_▷′_ : {F : Set → Set} {{r : Reduce F}} {A : Set} → FingerTree A zero
  → F A → FingerTree A zero
_▷′_ = reducel ((λ f x → f (Leaf x)) ∘ _▷_)
```

Then the function *toTree* could be defined to convert any reducible data structure to finger tree.

```
toTree : {F : Set → Set} {{r : Reduce F}} {A : Set} → F A → FingerTree A zero
toTree s = s ◁′ Empty
```

After this, we could prove that when we convert a list to finger tree, and back to list, it is still the same list. The main idea is that we need to prove it recursively. The base case is empty list, which is obviously correct. Then we prove for all non-empty lists $x :: xs$, converting the whole list to tree is equal to converting the tail to tree and add the head to the left of the tree. In order to prove this, we need three simple lemmas. The first one is the proof that for a sequence $x :: xs$, reducing on it as a whole equals reducing on the tail xs and then the head x.

```
lemmax1 : {A B : Set} {n : ℕ}
  (ft : FingerTree A n) (node : Node A n) (op : A → B → B)
  (z : B) →
  reducerFingerTree op (node ◁ ft) z ≡
  reducerNode op node (reducerFingerTree op ft z)
lemmax1 Empty a op z = refl
lemmax1 (Single x) a op z = refl
lemmax1 (Deep (One b) m sf) a op z = refl
lemmax1 (Deep (Two b c) m sf) a op z = refl
lemmax1 (Deep (Three b c d) m sf) a op z = refl
lemmax1 (Deep (Four b c d e) m sf) a op z =
  cong (λ x → reducerNode op a (reducerNode op b x))
    (lemmax1 m (Node3 c d e) op ((reducerDigit ∘ reducerNode) op sf z))
```

The second lemma is that reducing on a list equals reducing on the equivalent finger trees:

```
lemmax2 : {A B : Set}
  (xs : List A) (op : A → B → B) (z : B) →
  reducer {{reduceFingerTree}} op (toTree {{reduceList}} xs) z ≡
  reducer {{reduceList}} op xs z
lemmax2 [] op z = refl
lemmax2 (x :: xs) op z = trans (lemmax1 (toTree {{reduceList}} xs) (Leaf x) op z)
  (cong (op x) (lemmax2 xs op z))
```

Finally, we could prove the lemma that when we convert a list to a finger tree, and back to list, it is still the same list with the help of second lemma and another helper lemma that for all lists xs, $toList\, xs \equiv xs$:

```
lemmax : {A : Set}
  (xs : List A) →
  toList {{reduceFingerTree}} (toTree {{reduceList}} xs) ≡ xs
lemmax xs = trans (lemmax2 xs _::_ []) (lemma xs)
  where lemma : {A : Set} (xs : List A) → reducer {{reduceList}} _::_ xs [] ≡ xs
    lemma [] = refl
    lemma (x :: xs) = cong (_::_ x) (lemma xs)
```

It is impossible to prove that when we convert a tree to list, and back to tree, it is still the same tree though, because there are multiple ways to represent the same sequence. For example, the sequence 1 :: 2 :: 3 :: [] could be represented as either
$Deep\ (One\ 1)\ Empty\ (Two\ 2\ 3)$ or $Deep\ (Two\ 1\ 2)\ Empty\ (One\ 3)$.

5.1.4 Viewing

Then we can define types to view from the left and right end of the sequence which is either an empty sequence or a head element with its tail attached to it. For simplicity, we only include the view from the left end of the sequence. The view from the right end is similar.

```
data ViewL (S : Set → Set) (A : Set) : Set where
  NilL : ViewL S A
  ConsL : A → S A → ViewL S A
```

To define the function *viewL* to view from the left end of the sequence, We need a helper function *deepL* which takes a "deep" finger tree whose prefix might be null, to produce a legal finger tree. It is trivial to define it in Agda using mutual recursion with keyword *mutual*:

```
mutual
  viewL : {A : Set} {n : ℕ} → FingerTree A n
    → ViewL (λ _ → FingerTree A n) (Node A n)
  viewL Empty          = NilL
  viewL (Single x)     = ConsL x Empty
  viewL (Deep pr m sf) = ConsL (headDigit pr) (deepL (tailDigit pr) m sf)

  deepL : {A : Set} {n : ℕ} → NDigit (Node A n) → FingerTree A (suc n) →
    Digit (Node A n) → FingerTree A n
  deepL Zero m sf with viewL m
  ... | NilL        = digitToTree sf
  ... | ConsL a m′  = Deep (nodeToDigit a) m′ sf
  deepL (One a)        m sf = Deep (One a)        m sf
  deepL (Two a b)      m sf = Deep (Two a b)      m sf
  deepL (Three a b c) m sf = Deep (Three a b c) m sf
  deepL (Four a b c d) m sf = Deep (Four a b c d) m sf
```

Then we can define the function *isEmpty* to decide whether the sequence is empty or not using pattern matching on the output of *viewL* we defined above.

```
isEmpty : {A : Set} {n : ℕ} → FingerTree A n → Bool
isEmpty t with viewL t
... | NilL = true
... | ConsL _ _ = false
```

It is trivial to define the head and tail functions. Notice an implicit argument *lemma* is required as an input of *headL* to make sure the tree cannot be empty.

```
headL : {A : Set} {n : ℕ} (t : FingerTree A n) {lemma : isEmpty t ≡ false} →
  Node A n
headL t {_} with viewL t
headL t { } | NilL
headL t {_} | ConsL a _ = a

tailL : {A : Set} {n : ℕ} (t : FingerTree A n) → FingerTree A n
tailL t with viewL t
... | NilL = Empty
... | ConsL _ x′ = x′
```

5.1.5 Concatenation

Concatenation of two sequences can be simple by using deque operations defined above and recursively insert all the elements in one sequence to another. However, this is not efficient enough. When we try to concatenate two "deep" trees, clearly that the prefix of the first tree, and the suffix of the second tree remain the same, and the middle part of the new tree is combined by other parts. Here, we use accumulator style as defined in Haskell implementation to define the function of concatenating two trees and a list of nodes to be concatenated in the middle. We define two data structures, non-empty lists $List^{+}$ and lists with at least two elements $List^{++}$, to make sure the lists to pass to the function *nodes* below should have at least two elements.

```
app3 : {A : Set} {n : ℕ} → FingerTree A n → List (Node A n) → FingerTree A n →
  FingerTree A n
app3 Empty ts xs = _◁′_ {{reduceList}} ts xs
app3 xs ts Empty = _▷′_ {{reduceList}} xs ts
app3 (Single x) ts xs = x ◁ _◁′_ {{reduceList}} ts xs
app3 xs ts (Single x) = _▷′_ {{reduceList}} xs ts ▷ x
app3 (Deep pr1 m1 sf1) ts (Deep pr2 m2 sf2) = Deep pr1
  (app3 m1
    (nodes
      ((toList⁺ sf1 ⁺++ ts)
        ⁺++⁺ toList⁺ pr2))
    m2)
  sf2
```

A helper function *nodes* is needed to convert a list of nodes to a list of nodes whose height increase by one. Notice in Agda implementation, we force the input list to have at least two elements by using a new data type $List^{++}$.

```
nodes : {A : Set} {n : ℕ} → List⁺⁺ (Node A n) → List (Node A (suc n))
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)
```

Then, the concatenation operator can be defined as below:

```
_⋈_ : {A : Set} {n : ℕ} → FingerTree A n → FingerTree A n → FingerTree A n
xs ⋈ ys = app3 xs [] ys
```

5.2 Annotated Sequences

Recall that we have defined a record type of measurement as below:

```
record Measured (A : Set) (V : Set) : Set where
  field
    measure : A → V
```

In this subsection, it is used to provide a way to measure any leaves stored in the finger tree to its annotated monoid value. This could be very useful to the data type. For instance, if we need to measure the size of a sequence, we only need the monoid of natural numbers with addition, and measure each leaf to one which means it is a node of size one. Notice that this particular monoid is commutative, but in general, this is not needed for all cases. The implementation of annotated sequence is similar to the simple sequence in most parts, therefore, in this subsection, we would focus on the different parts.

5.2.1 Caching Measurements

Unlike in Haskell implementation, we can benefit from the dependent data type by caching the monoid value directly in the data type itself, which means we do not need extra smart constructors as in Haskell, because we can embed all the monoid computations in the constructors of the data type. For simplicity, we change the data type of *Digit* to make it compute the monoid value while construction instead of compute it when it is measured as in Haskell implementation.

```
data Node {V : Set} {{m : Monoid V}} (A : Set) : V → ℕ → Set where
  Node2 : {v1 v2 : V} {n : ℕ} → Node A v1 n → Node A v2 n →
    Node A (v1 ⊕ v2) (suc n)
  Node3 : {v1 v2 v3 : V} {n : ℕ} → Node A v1 n → Node A v2 n → Node A v3 n →
    Node A ((v1 ⊕ v2) ⊕ v3) (suc n)
  Leaf : {v : V} → A → Node A v zero
```

```
data Digit {V : Set} {{m : Monoid V}} (A : Set) : V → ℕ → Set where
  One   : {v : V}          {n : ℕ} → Node A v n → Digit A v n
  Two   : {v1 v2 : V}      {n : ℕ} → Node A v1 n → Node A v2 n →
          Digit A (v1 ⊕ v2) n
  Three : {v1 v2 v3 : V}   {n : ℕ} → Node A v1 n → Node A v2 n → Node A v3 n →
          Digit A ((v1 ⊕ v2) ⊕ v3) n
  Four  : {v1 v2 v3 v4 : V} {n : ℕ} → Node A v1 n → Node A v2 n → Node A v3 n →
          Node A v4 n → Digit A (((v1 ⊕ v2) ⊕ v3) ⊕ v4) n
```

```
data FingerTree {V : Set} {{m : Monoid V}} (A : Set) : V → ℕ → Set where
  Empty : {n : ℕ} → FingerTree A ∅ n
  Single : {v : V} {n : ℕ} → Node A v n → FingerTree A v n
  Deep   : {v1 v2 v3 : V} {n : ℕ} → Digit A v1 n → FingerTree A v2 (suc n) →
            Digit A v3 n → FingerTree A ((v1 ⊕ v2) ⊕ v3) n
```

It is easy to get the monoid value out of the data type as it is cached in its type signature.

```
getNodeV : {A : Set} {V : Set} {{m : Monoid V}} {v : V} {n : ℕ} → Node A v n
  → V
getNodeV {v = v} _ = v


getDigitV : {A : Set} {V : Set} {{m : Monoid V}} {v : V} {n : ℕ} → Digit A v n
  → V
getDigitV {v = v} _ = v


getV : {A : Set} {V : Set} {{m : Monoid V}} {v : V} {n : ℕ} → FingerTree A v n
  → V
getV {v = v} _ = v
```

5.2.2 Deque Operations

Unlike in Haskell implementation, when adding the measurement to the finger trees, because the measured value is recorded in the type, some type mismatching problem would be caused. For example, let us have a look at the first case of the definition of ◁ when we add the measurement,

```
_◁_ : {V : Set} {{m : Monoid V}} {A : Set} {n : ℕ} {v1 v2 : V} → Node A v1 n →
  FingerTree A v2 n → FingerTree A (v1 ⊕ v2) n
a ◁ Empty = Single a
```

It is no longer accepted by Agda, as according the definition, *Single a* is now of type $FingerTree\ A\ v\ n$ where v is the value cached in a. However, the expected type is $FingerTree\,A\,(v \oplus \emptyset)\,n$. Although we can see that they are actually the same type because v is equal to $v \oplus \emptyset$ according to $id1$ defined in $IsMonoid$, we need to provide a proof of v being equal to $v \oplus \emptyset$, and then substitute v with $v \oplus \emptyset$ to eliminate the error of type mismatching.

The substitution can be done by using the function *subst* defined in the standard library. We can use the definition to define functions to substitute the values cached in the data types we defined with other equalled values.

```
substNode : {A : Set} {n : ℕ} {V : Set} {{m : Monoid V}} {v1 v2 : V} → v1 ≡ v2 →
   Node A v1 n → Node A v2 n
substNode {A} {n} = subst (λ v → Node A v n)

substDigit : {A : Set} {n : ℕ} {V : Set} {{m : Monoid V}} {v1 v2 : V} → v1 ≡ v2 →
   Digit A v1 n → Digit A v2 n
substDigit {A} {n} = subst (λ v → Digit A v n)

substFingerTree : {A : Set} {n : ℕ} {V : Set} {{m : Monoid V}} {v1 v2 : V} →
   v1 ≡ v2 → FingerTree A v1 n → FingerTree A v2 n
substFingerTree {A} {n} = subst (λ v → FingerTree A v n)
```

Then, we can change the definition of ◁ to make it accepted by Agda:

```
_◁_ : {V : Set} {{m : Monoid V}} {A : Set} {n : ℕ} {v1 v2 : V} → Node A v1 n →
   FingerTree A v2 n → FingerTree A (v1 ⊕ v2) n
a ◁ Empty                    = substFingerTree {!!} (Single a)
a ◁ Single b                 = substFingerTree {!!} (Deep (One a) Empty (One b))
a ◁ Deep (One b) m sf        = substFingerTree {!!} (Deep (Two a b) m sf)
a ◁ Deep (Two b c) m sf      = substFingerTree {!!} (Deep (Three a b c) m sf)
a ◁ Deep (Three b c d) m sf  = substFingerTree {!!} (Deep (Four a b c d) m sf)
a ◁ Deep (Four b c d e) m sf  = substFingerTree {!!} (Deep (Two a b)
   (Node3 c d e ◁ m)
   sf)
```

Note that we need to provide proofs in the unsolved goals. (denoted as {!!})

Other functions can also be modified in this way in order to deal with the cached value in the sequence.

In order to prove the lemma that when we convert a list to tree and back to list, we will need some extra helper lemmas. With the use of heterogeneous equality, we first need to prove that the tree after substituting the value cached in with an equalled value, will be the tree heterogeneously equalled to the tree before substitution.

```
lemmax1 : {V : Set} {{m : Monoid V}} {A : Set} {v1 v2 : V} {n : ℕ}
   (ft : FingerTree A v1 n) (eq : v1 ≡ v2) →
   substFingerTree eq ft ≅ ft
lemmax1 ft refl = H.refl
```

Then, we also need to produce a lemma of congruence law which shows all functions respect the heterogeneous equality, i.e. if f is applied on two heterogeneously equalled values, then the results are also heterogeneously equalled.

```
cong′ : {I : Set} {i j : I}
   → (A : I → Set) {B : {k : I} → A k → Set} {x : A i} {y : A j}
   → i ≡ j
   → (f : {k : I} → (x : A k) → B x)
   → x ≅ y
   → f x ≅ f y
cong′ _ refl _ H.refl = H.refl
```

Then we could prove the lemma in a similar way that we prove it in section 5.1.3.

5.2.3 Viewing

Viewing a finger tree with an annotated value is much different in order to be implemented in Agda. Recall that previously, we use mutual recursion of *viewL* and *deepL* and part of the definition of *deepL* is as below:

```
deepL Zero m sf with viewL m
... | NilL = digitToTree sf
```

Now, as we need a proof of the equality of values, we need to know that the value annotated in m is $\emptyset$ which is provable when we know that $viewL\ m$ is equal to $NilL$. Although it is obvious to us because we have pattern matched on it, Agda cannot directly save the equality for further use. We will need an extra function to keep record of the equality of pattern matching. In Agda, this can be done by using the function *inspect*. And When *inspect* is used, termination check of Agda would fail.

One way of solving it is to get rid of the mutual recursion used before. Therefore, instead of defining the *viewL* first, and use it to define *headL* and *tailL*. We first give the definition of *headL* which is trivial. (For simplicity, we use *Maybe* type as the output)

```
headL : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (ft : FingerTree A v n) → Maybe (Node A (headV ft) n)
headL Empty      = nothing
headL (Single x) = just x
headL (Deep pr _ _) = just (headDigit pr)
```

Then we can use the definition above to define *tailL*. Then, we need to consider three cases:

1. When the sequence has less than two elements, its tail is empty
2. When the prefix of the sequence has more than one nodes, we can simply remove the first node of it.

3. When the prefix of the sequence has one node, then we need to find the head and the tail of the middle part of the tree. After that, we convert the head node of it to the digit to be put in the prefix, and put the tail of the previous middle part to be the new middle part, while suffix of it remains the same.

Notice that in order to convice Agda that for all non-empty trees, *headL* of them would return the element, we need to give all the cases of the possible trees, because we pattern matched on all the cases in the function *headL*. Therefore, we pattern match on all the possible cases on *m* (The middle part of the tree), and give elegant definition when it is constructed by constructor *Empty* or *Single*, without the help of *headL*. When it is constructed by the constructor *Deep*, we use *inspect* to eliminate the impossible case that the result of *headL* is *nothing*.

```
tailL : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (ft : FingerTree A v n) → FingerTree A (tailV ft) n
tailL Empty = Empty
tailL (Single x) = Empty
tailL (Deep (One a) Empty sf) = substFingerTree {!!} (digitToTree sf)
tailL (Deep (One a) (Single x) sf) = substFingerTree {!!}
  (Deep (nodeToDigit x) Empty sf)
tailL (Deep (One a) (Deep pr m sf) sf2) with headL (Deep pr m sf) |
  inspect headL (Deep pr m sf)
... | nothing | [ () ]
... | just x | _ = substFingerTree {!!}
                  (Deep (nodeToDigit x) (tailL (Deep pr m sf)) sf2)
tailL (Deep (Two a b)     m sf) = Deep (One b) m sf
tailL (Deep (Three a b c) m sf) = Deep (Two b c) m sf
tailL (Deep (Four a b c d) m sf) = Deep (Three b c d) m sf
```

Now, we can finally use the definitions above to define the function *viewL* for the finger trees with annotated value.

```
viewL : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (ft : FingerTree A v n) →
  ViewL (λ _ → FingerTree A (tailV ft) n) (Node (headV ft) n)
viewL t with headL t
... | nothing = NilL
... | just x = ConsL x (tailL t)
```

5.2.4 Concatenation

In order to implement the concatenation of annotated sequences, the list accumulator used before cannot be used anymore, as the list can only contain elements of the same type but the type of nodes are different once their annotated value is not the same. Thus, we need to define a new data type which can provide a sequence of different nodes. It is easy to define the type as it is similar to the definition of list:

```
data NodeList {V : Set} {{m : Monoid V}} (A : Set) : V → ℕ → Set where
  [] : {n : ℕ} → NodeList A ∅ n
  _::_ : {v1 v2 : V} {n : ℕ} (x : Node A v1 n) (xs : NodeList A v2 n) →
    NodeList A (v1 ⊕ v2) n
```

5.2.5 Splitting

We can now move on to the splitting part. To split a tree, we need a predicate on the measurements and an accumulator. The accumulator is used to append all the monoid values stored before the current node. Splitting on trees would produce three parts:

1. A leaf node in the tree that before it, the predicate applied to the accumulator evaluates to $false$ and after it, the predicate applied to the accumulator evaluates to $true$.
2. The tree before the node.
3. The tree after the node.

To wrap all three parts into a single data type, we therefore define *Split*:

```
data Split (F1 : Set → Set) (A : Set) (F2 : Set → Set) : Set where
  split : F1 A → A → F2 A → Split F1 A F2
```

The figure below illustrates how splitting on trees work:

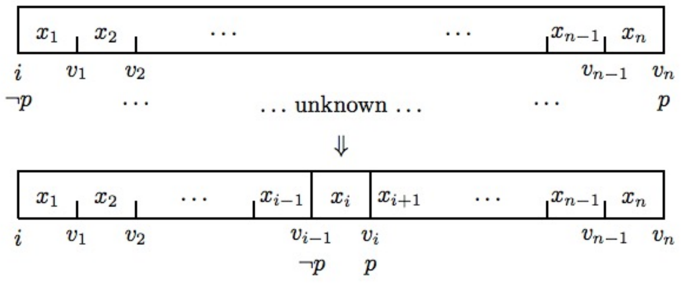

Figure 3: Functioning Illustration of splitting on trees [5]

Splitting on digits is trivial but more complicated than the Haskell implementation, as we need to pass the termination check by giving all the cases manually.

```
splitDigit : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (as : Digit A v n) →
  Split (λ _ → NDigit A (splitV1 p i as) n)
    (Node A (splitV2 p i as) n)
    (λ _ → NDigit A (splitV3 p i as) n)
splitDigit p i (One a) = split Zero a Zero
splitDigit p i (Two {v1} a b) with p (i ⊕ v1)
... | true = split Zero a (One b)
... | false = split (One a) b Zero
splitDigit p i (Three {v1} {v2} a b c) with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2)
... | true | _ = split Zero a (Two b c)
... | false | true = split (One a) b (One c)
... | false | false = split (Two a b) c Zero
splitDigit p i (Four {v1} {v2} {v3} a b c d) with p (i ⊕ v1) |
  p ((i ⊕ v1) ⊕ v2) |
  p (((i ⊕ v1) ⊕ v2) ⊕ v3)
... | true | _ | _ = split Zero a (Three b c d)
... | false | true | _ = split (One a) b (Two c d)
... | false | false | true = split (Two a b) c (One d)
... | false | false | false = split (Three a b c) d Zero
```

Splitting on trees is more tricky to implement, but the idea is very easy. For a tree of only single element x, the result is clearly *split Empty x Empty*. For a "deep" tree, according to the value $v1$ cached in the prefix, the value $v2$ cached in the middle subtree, and the value $v3$ cached in the suffix, it is easy to find out where the leaf we need to find is (assume predicate is p and the accumulator is i):

1. If $p(i \oplus v1)$ evaluates to *true*, then the leaf is in the prefix, we just need to call *splitDigit* on the prefix to get the leaf.

2. Else, if $p(i \oplus v1 \oplus v2)$ evaluates to *true*, then the leaf is in the middle subtree, we can call *splitTree* recursively to get a node of height one, and then split on it to get the result.

3. Otherwise, the leaf is in the suffix, we just need to call *splitDigit* on the suffix to get the leaf.

```
splitTree : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ} (p : V → Bool)
  (i : V) (ft : FingerTree A v n) {le : isEmpty ft ≡ false} →
  Split (λ _ → FingerTree A (splitTreeV1 p i ft) n)
    (Node A (splitTreeV2 p i ft) n)
    (λ _ → FingerTree A (splitTreeV3 p i ft) n)
splitTree p i t = {!!}
```

For a more complete version, please see appendix B.

Then, we could define a general version of split on trees without the accumulator:

```
split′ : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (ft : FingerTree A v n) →
  FingerTree A (split′V1 p ft) n × FingerTree A (split′V2 p ft) n
split′ p Empty = Empty , Empty
split′ p (Single x) with p v
... | true = Empty , substFingerTree {!!} (Single x)
... | false = Single x , Empty
split′ p (Deep pr m sf) with splitTree p ∅ (Deep pr m sf) {refl}
... | split l x r with p (getV (Deep pr m sf))
... | true = l , x ◁ r
... | false = Deep pr m sf , Empty
```

We can now use the definition above to define *takeUntil* and *dropUntil*:

```
takeUntil : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (ft : FingerTree A v n) → FingerTree A (split′V1 p ft) n
takeUntil p t = proj₁ (split′ p t)


dropUntil : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (ft : FingerTree A v n) → FingerTree A (split′V2 p ft) n
dropUntil p t = proj₂ (split′ p t)
```

5.3 Application

Now we can use the implementation of finger trees to implement other complex data structures in a relatively easy way. The following are two simple example applications that can be built on the top of finger trees:

5.3.1 Random-Access Sequence

To implement random-access sequence, we need to firstly define a monoid of size, which is simply the monoid formed by natural numbers under addition. The definition has been given in section 4.1 as an example. For clarity, we rename it to *monoidSize*.

To provide the means of measurement, we simply measure each leaf to 1, no matter what is stored in it:

```
measuredSize : {A : Set} → Measured A ℕ
measuredSize = record {measure = λ _ → 1}
```

Then we can give a method to convert any lists to random-access sequences:

```
listToSeq : {A : Set} (xs : List A) →
   FingerTree {{monoidSize}} A (length xs) zero
listToSeq = listToTree {{monoidSize}} {{measuredSize}}
```

Then, it is trivial to define a function to access a element stored in it according to its index:

```
_!_ : {v : ℕ} {A : Set} (ft : FingerTree {{monoidSize}} A (suc v) zero)
   (i : ℕ) → A
_!_ t i with splitTree {{monoidSize}} (_<_ i) 0 t {{!!}}
... | split _ (Leaf x) _ = x
```

Note that with the help of the dependent data type, we could now force the input to be an non-empty finger tree in the type, without the use of extra lemmas or *Maybe* type.

5.3.2 Priority Queue

In order to implement priority queue, we need a new data type to represent the priority first:

```
data Prio (A : Set) : Set where
  mInfty : Prio A
  prio : (a : A) → Prio A
```

Then we can implement the operator on priority. For simplicity, we let the priority value to be natural number.

```
_⊕′_ : Prio ℕ → Prio ℕ → Prio ℕ
mInfty ⊕′ p = p
p ⊕′ mInfty = p
prio m ⊕′ prio n with m « n
... | true = prio n
... | false = prio m
```

Then, we can define the monoid of *Prio* ℕ:

```
monoidKey : {A : Set} → Monoid (Key A)
monoidKey {A} = record {∅ = noKey; _⊕_ = _⊕′_;
  isMonoid = {!!}}
```

Measuring the priority of elements is simply putting it to be wrapped by *Prio*. For simplicity, we let the elements to be natural numbers:

```
measuredPrio : Measured ℕ (Prio ℕ)
measuredPrio = record {measure = λ x → prio x}
```

Then, we can define the function to remove the max value from the tree, and return the value and the remaining tree:

```
extractMax : {v : Prio ℕ} (ft : FingerTree {{monoidPrio}} ℕ v zero) →
  Maybe (ℕ × FingerTree {{monoidPrio}} ℕ {!!})
extractMax Empty = nothing
extractMax (Single (Leaf x)) =
  just (x , substFingerTree {{monoidPrio}} {!!} Empty)
extractMax (Deep pr m sf)
  with splitTree {{monoidPrio}} (_<=_ (getV (Deep pr m sf)))
    mInfty (Deep pr m sf) {refl}
... | split l (Leaf x) r = just (x , l ⋈ r)
```

6 Discussion

6.1 Inflexibility

Recall that the definition of the record type *Reduce* is like below:

```
record Reduce (F : Set → Set) : Set₁ where
  field
    reducer : {A : Set} {B : Set} → (A → B → B) → F A → B → B
    reducel : {A : Set} {B : Set} → (B → A → B) → B → F A → B
```

It is providing means to recursively applying the input the function to the reducible type with the help of an accumulator from the right or left to get the summary value. The type of the input function is $A \to B \to B$, which means before and after applying the function, the type should be the same as the final output value. However, this means that the function of finger trees cannot be used anymore because the type of finger trees keep changing after different operations because of the change in annotated value. While *Reduce* can still be used to convert any reducible types, especially finger trees, to lists, we cannot provide a way of converting them to finger trees neatly.

There are several workarounds. One of them is to first convert any reducible types to lists, and then, we convert list to trees which is very easy to implement using recursive functions:

```
listToTree : {V : Set} {{m : Monoid V}} {A : Set} {{mea : Measured A V}} →
  (xs : List A) → FingerTree A (foldr (_⊕_ ∘ measure) ∅ xs) zero
listToTree [] = Empty
listToTree (x :: xs) = Leaf x ◁ listToTree xs
```

Then we can finally define a method of converting any reducible types to finger trees:

```
toTree : {F : Set → Set} {{r : Reduce F}} {A : Set} {V : Set} {{m : Monoid V}}
  {{mea : Measured A V}} (xs : F A) → FingerTree A _ zero
toTree = listToTree ∘ toList
```

Clearly this method is not as efficient as the one we defined before for the trees without measurements, because it is doing two conversions. This would not affect the time complexity.

Another alternative way is to define a single function for each reducible type to convert it to a finger tree wherever needed. It is easy to do so similarly to the example above of *listToTree*. This is perfectly fine to define functions like *digitToTree* to implement other parts of the project, which is what I have chosen to do, but it also eliminates some elegant ideas behind the use of *Reduce*. One of them is that with the use of *Reduce*, if we have a reducible type to convert to finger trees, we only need to record it as an instance of *Reduce* type, and use the predefined function.

6.2 Code Pollutions

There are some codes in the Agda implementation that may slow down the running time of the operations. Some of them are the proofs of equality of monoid values. As we talk about before, they are essential to the Agda implementation to get the correct type, but as they need to be plugged in the operation, each time we run the operation, we also need to run the proof to see if the type matches. This might cause inefficiency although it would not affect the time complexity.

Some of them are the helper functions to calculate the monoid value for the operations. For example, let us have a look at the type signature of *splitDigit*.

```
splitDigit : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (as : Digit A v n) →
    Split (λ _ → NDigit A (splitV1 p i as) n)
      (Node A (splitV2 p i as) n)
      (λ _ → NDigit A (splitV3 p i as) n)
```

As we can see, to get the result of *splitDigit*, we need to call *splitV*1, *splitV*2 and *splitV*3 to match the type of it, that is, in order to get the correct result, we actually split on digits for four times, which may cause inefficiency. Again, it would not affect the time complexity.

6.3 Future Work

There are other strong applications of finger trees, like ordered sequences, which could be implemented using the current implementation. However, with the current implementation of the finger trees, it is a bit tricky to implement them.

For example, while merging two ordered sequences, we need to recursively view and split on trees. This could not be done because it would not pass the termination check because the structure after viewing and splitting is not reducing, thus it is not sure that it will finally eliminate to the base cases. There are ways to work around. One way that I come up with is to add a new argument which represents the size of the tree in the operation, and with the proof that the size reduces after viewing or splitting, we should be able to convince Agda that this operation would finally terminates.

7 Conclusion

This dissertation follows the initial implementation introduced by Hinze and Paterson [5] to implement finger trees in Agda. In the end, the data structures and all basic operations on them are successfully implemented, with invariants proven. Some applications introduced by Hinze and Paterson are also built on the top of current implementation. As this is, as far as I am concerned, the first time for finger trees to be fully implemented in Agda, some tricky problems raised during the dissertation which may took me weeks to solve a single one, finally they are all solved. The final result of this dissertation is a certified implementation of independent finger trees.

The complete work can be found in the following url:
`https://github.com/ZimuQin/G53IDS`

A Monoidic Lemma Examples

The following lemmas could be used by the definition of operations on finger trees, and can be proved by using monoid laws. We use the module
$\equiv -Reasoning$ in standard library to provide a clearer step-by-step equality proof started by the keyword *begin*, and ended by the Q.E.D symbol ■.

```
lemma1 : {V : Set} {{m : Monoid V}} {v1 v2 v3 v4 : V} →
  _⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4 ≡
  _⊕_ v1 (_⊕_ (_⊕_ v2 v3) v4)
lemma1 {_} {v1} {v2} {v3} {v4} =
  begin
    _⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4 ≡⟨
    cong (flip _⊕_ v4) (assoc v1 v2 v3)
    ⟩
    _⊕_ (_⊕_ v1 (_⊕_ v2 v3)) v4 ≡⟨
    assoc v1 (_⊕_ v2 v3) v4
    ⟩
    _⊕_ v1 (_⊕_ (_⊕_ v2 v3) v4) ■


lemma2 : {V : Set} {{m : Monoid V}} {v1 v2 v3 v4 v5 : V} →
  _⊕_ (_⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4) v5 ≡
  _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5)
lemma2 {_} {v1} {v2} {v3} {v4} {v5} =
  begin
    _⊕_ (_⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4) v5 ≡⟨
    cong (flip _⊕_ v5) lemma1
    ⟩
    _⊕_ (_⊕_ v1 (_⊕_ (_⊕_ v2 v3) v4)) v5 ≡⟨
    assoc v1 (_⊕_ (_⊕_ v2 v3) v4) v5
    ⟩
    _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5) ■


lemma3 : {V : Set} {{m : Monoid V}} {v1 v2 v3 v4 v5 v6 : V} →
  _⊕_ (_⊕_ (_⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4) v5) v6 ≡
  _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5) v6)
lemma3 {_} {v1} {v2} {v3} {v4} {v5} {v6} =
  begin
    _⊕_ (_⊕_ (_⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4) v5) v6 ≡⟨
    cong (flip _⊕_ v6) lemma2 ⟩
    _⊕_ (_⊕_ v1 (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5)) v6 ≡⟨
    assoc v1 (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5) v6 ⟩
    _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5) v6) ■
```

```
lemma4 : {V : Set} {{m : Monoid V}} {v1 v2 v3 v4 v5 v6 v7 : V} →
  _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) v6)) v7 ≡
  _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5) v6) v7)
lemma4 {_} {v1} {v2} {v3} {v4} {v5} {v6} {v7} =
  begin
    _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) v6)) v7 ≡⟨
    cong (λ vx → _⊕_ (_⊕_ (_⊕_ v1 v2) vx) v7) (assoc (_⊕_ v3 v4) v5 v6)
    ⟩
    _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ v3 v4) (_⊕_ v5 v6))) v7 ≡⟨
    cong (λ vx → _⊕_ (_⊕_ (_⊕_ v1 v2) vx) v7) (assoc v3 v4 (_⊕_ v5 v6))
    ⟩
    _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ v3 (_⊕_ v4 (_⊕_ v5 v6)))) v7 ≡⟨
    cong (flip _⊕_ v7) (assoc v1 v2 (_⊕_ v3 (_⊕_ v4 (_⊕_ v5 v6)))) ⟩
    _⊕_ (_⊕_ v1 (_⊕_ v2 (_⊕_ v3 (_⊕_ v4 (_⊕_ v5 v6))))) v7 ≡⟨
    assoc v1 (_⊕_ v2 (_⊕_ v3 (_⊕_ v4 (_⊕_ v5 v6)))) v7 ⟩
    _⊕_ v1 (_⊕_ (_⊕_ v2 (_⊕_ v3 (_⊕_ v4 (_⊕_ v5 v6)))) v7) ≡⟨
    cong (λ vx → _⊕_ v1 (_⊕_ vx v7))
    (sym (assoc v2 v3 (_⊕_ v4 (_⊕_ v5 v6))))
    ⟩
    _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ v2 v3) (_⊕_ v4 (_⊕_ v5 v6))) v7) ≡⟨
    cong (λ vx → _⊕_ v1 (_⊕_ vx v7))
    (sym (assoc (_⊕_ v2 v3) v4 (_⊕_ v5 v6)))
    ⟩
    _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) (_⊕_ v5 v6)) v7) ≡⟨
    cong (λ vx → _⊕_ v1 (_⊕_ vx v7))
    (sym (assoc (_⊕_ (_⊕_ v2 v3) v4) v5 v6))
    ⟩ _⊕_ v1 (_⊕_ (_⊕_ (_⊕_ (_⊕_ (_⊕_ v2 v3) v4) v5) v6) v7) ■
```

```
lemma5 : {V : Set} {{m : Monoid V}} {v1 v2 v3 v4 v5 v6 v7 : V} →
  _⊕_ (_⊕_ v1 (_⊕_ v2 (_⊕_ (_⊕_ v3 v4) v5))) (_⊕_ v6 v7) ≡
  _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) v6)) v7
lemma5 {_} {v1} {v2} {v3} {v4} {v5} {v6} {v7} =
  begin
    _⊕_ (_⊕_ v1 (_⊕_ v2 (_⊕_ (_⊕_ v3 v4) v5))) (_⊕_ v6 v7) ≡⟨
    cong (flip _⊕_ (_⊕_ v6 v7))
    (sym (assoc v1 v2 (_⊕_ (_⊕_ v3 v4) v5)))
    ⟩
    _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ v3 v4) v5)) (_⊕_ v6 v7) ≡⟨
    assoc (_⊕_ v1 v2) (_⊕_ (_⊕_ v3 v4) v5) (_⊕_ v6 v7) ⟩
    _⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) (_⊕_ v6 v7)) ≡⟨
    cong (_⊕_ (_⊕_ v1 v2)) (sym (assoc (_⊕_ (_⊕_ v3 v4) v5) v6 v7)) ⟩
    _⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) v6) v7) ≡⟨
    sym (assoc (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) v6) v7) ⟩
    _⊕_ (_⊕_ (_⊕_ v1 v2) (_⊕_ (_⊕_ (_⊕_ v3 v4) v5) v6)) v7 ■
```

B Complete Implementation of Splitting

In section 5.2.5, we have talked about splitting on annotated finger trees. However, we have left some goals which are a little tricky to implement. Here we give the complete implementation in Agda as below (For more details, please see the source files):

B.1 Splitting on Digits

To split on digits, firstly we need three helper *splitV*1, *splitV*2 and *splitV*3 functions to figure out the cached monoidic values for the three parts after splitting:

```
splitV1 : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (as : Digit A v n) → V
splitV1 p i (One a) = ∅
splitV1 p i (Two {v1} a b) with p (i ⊕ v1)
... | true = ∅
... | false = v1
splitV1 p i (Three {v1} {v2} a b c) with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2)
... | true | _ = ∅
... | false | true = v1
... | false | false = v1 ⊕ v2
splitV1 p i (Four {v1} {v2} {v3} a b c d)
  with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2) | p (((i ⊕ v1) ⊕ v2) ⊕ v3)
... | true | _ | _ = ∅
... | false | true | _ = v1
... | false | false | true = v1 ⊕ v2
... | false | false | false = (v1 ⊕ v2) ⊕ v3


splitV2 : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (as : Digit A v n) → V
splitV2 p i (One {v} a) = v
splitV2 p i (Two {v1} {v2} a b) with p (i ⊕ v1)
... | true = v1
... | false = v2
splitV2 p i (Three {v1} {v2} {v3} a b c) with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2)
... | true | _ = v1
... | false | true = v2
... | false | false = v3
splitV2 p i (Four {v1} {v2} {v3} {v4} a b c d)
  with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2) | p (((i ⊕ v1) ⊕ v2) ⊕ v3)
... | true | _ | _ = v1
... | false | true | _ = v2
... | false | false | true = v3
... | false | false | false = v4
```

```
splitV3 : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (as : Digit A v n) → V
splitV3 p i (One a) = ∅
splitV3 p i (Two {v1} {v2} a b) with p (i ⊕ v1)
... | true = v2
... | false = ∅
splitV3 p i (Three {v1} {v2} {v3} a b c) with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2)
... | true | _ = v2 ⊕ v3
... | _ | true = v3
... | _ | _ = ∅
splitV3 p i (Four {v1} {v2} {v3} {v4} a b c d)
  with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2) | p (((i ⊕ v1) ⊕ v2) ⊕ v3)
... | true | _ | _ = (v2 ⊕ v3) ⊕ v4
... | _ | true | _ = v3 ⊕ v4
... | _ | _ | true = v4
... | _ | _ | _ = ∅


splitDigit : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (as : Digit A v n) →
    Split (λ _ → NDigit A (splitV1 p i as) n)
      (Node A (splitV2 p i as) n)
      (λ _ → NDigit A (splitV3 p i as) n)
splitDigit p i (One a) = split Zero a Zero
splitDigit p i (Two {v1} a b) with p (i ⊕ v1)
... | true = split Zero a (One b)
... | false = split (One a) b Zero
splitDigit p i (Three {v1} {v2} a b c) with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2)
... | true | _ = split Zero a (Two b c)
... | false | true = split (One a) b (One c)
... | false | false = split (Two a b) c Zero
splitDigit p i (Four {v1} {v2} {v3} a b c d)
  with p (i ⊕ v1) | p ((i ⊕ v1) ⊕ v2) | p (((i ⊕ v1) ⊕ v2) ⊕ v3)
... | true | _ | _ = split Zero a (Three b c d)
... | false | true | _ = split (One a) b (Two c d)
... | false | false | true = split (Two a b) c (One d)
... | false | false | false = split (Three a b c) d Zero
```

B.2 Splitting on Trees

Firstly we need to define a data structure of null-able digits to be used by the function *deepL* and *deepR*.

```
data NDigit {V : Set} {{m : Monoid V}} (A : Set) : V → ℕ → Set where
  Zero : {n : ℕ} → NDigit A ∅ n
  One : {v : V} {n : ℕ} → Node A v n →
        NDigit A v n
  Two : {v1 v2 : V} {n : ℕ} → Node A v1 n → Node A v2 n →
        NDigit A (v1 ⊕ v2) n
  Three : {v1 v2 v3 : V} {n : ℕ} → Node A v1 n → Node A v2 n → Node A v3 n →
        NDigit A ((v1 ⊕ v2) ⊕ v3) n
  Four : {v1 v2 v3 v4 : V} {n : ℕ} → Node A v1 n → Node A v2 n → Node A v3 n →
        Node A v4 n → NDigit A (((v1 ⊕ v2) ⊕ v3) ⊕ v4) n
```

Then, we need to implement a new version of *deepL* (and *deepR*, which is similar) which does not depend on mutual recursions in order to pass termination check:

```
deepL : {V : Set} {{m : Monoid V}} {A : Set} {v1 v2 v3 : V} {n : ℕ} →
  NDigit A v1 n → FingerTree A v2 (suc n) → Digit A v3 n →
  FingerTree A ((v1 ⊕ v2) ⊕ v3) n
deepL Zero Empty sf = substFingerTree lemma (digitToTree sf)
  where lemma : {V : Set} {{m : Monoid V}} {v : V} →
          v ≡ _⊕_ (_⊕_ ∅ ∅) v
    lemma {_} {v} = begin
      v ≡⟨ id2 v ⟩
      _⊕_ ∅ v ≡⟨ cong (flip _⊕_ v) (id1 ∅) ⟩ _⊕_ (_⊕_ ∅ ∅) v ■
deepL Zero (Single x) sf = substFingerTree lemma
  (Deep (nodeToDigit x) Empty sf)
  where lemma : {V : Set} {{m : Monoid V}} {v1 v2 : V} →
      _⊕_ (_⊕_ v1 ∅) v2 ≡ _⊕_ (_⊕_ ∅ v1) v2
    lemma {_} {v1} {v2} = begin
      _⊕_ (_⊕_ v1 ∅) v2 ≡⟨ cong (flip _⊕_ v2)
      (sym (id1 v1)) ⟩
      _⊕_ v1 v2 ≡⟨ cong (flip _⊕_ v2) (id2 v1) ⟩
      _⊕_ (_⊕_ ∅ v1) v2 ■
deepL Zero (Deep pr m sf) sf2
  with headL (Deep pr m sf) | inspect headL (Deep pr m sf)
... | nothing | [ () ]
... | just x | _ = substFingerTree (lemma {pr = pr})
             (Deep (nodeToDigit x) (tailL (Deep pr m sf)) sf2)
  where lemma : {A : Set} {V : Set} {{m : Monoid V}} {v1 v2 v3 v4 : V}
    {pr : Digit A v1 _} →
    _⊕_ (_⊕_ (headDigitV pr) (_⊕_ (_⊕_ (tailDigitV pr) v2)
      v3)) v4 ≡ _⊕_ (_⊕_ ∅ (_⊕_ (_⊕_ v1 v2) v3)) v4
    lemma {v1 = v1} {v2} {v3} {v4} {pr} =
            begin
              _⊕_ (_⊕_ (headDigitV pr) (_⊕_ (_⊕_ (tailDigitV pr) v2) v3)) v4 ≡⟨
              cong (flip _⊕_ v4)
              (sym (assoc (headDigitV pr) (_⊕_ (tailDigitV pr) v2) v3))
              ⟩
              _⊕_ (_⊕_ (_⊕_ (headDigitV pr) (_⊕_ (tailDigitV pr) v2)) v3) v4 ≡⟨
              cong (λ v → _⊕_ (_⊕_ v v3) v4)
              (sym (assoc (headDigitV pr) (tailDigitV pr) v2))
              ⟩
              _⊕_ (_⊕_ (_⊕_ (_⊕_ (headDigitV pr) (tailDigitV pr)) v2) v3) v4 ≡⟨
              cong (λ v → _⊕_ (_⊕_ (_⊕_ v v2) v3) v4) (lemma6 pr) ⟩
              _⊕_ (_⊕_ (_⊕_ v1 v2) v3) v4 ≡⟨
              cong (flip _⊕_ v4) (id2 (_⊕_ (_⊕_ v1 v2) v3)) ⟩
              _⊕_ (_⊕_ ∅ (_⊕_ (_⊕_ v1 v2) v3)) v4 ■
deepL (One a) m sf = Deep (One a) m sf
deepL (Two a b) m sf = Deep (Two a b) m sf
deepL (Three a b c) m sf = Deep (Three a b c) m sf
deepL (Four a b c d) m sf = Deep (Four a b c d) m sf
```

Then, we need to implement the most tricky part. We use a mutual recursion between four functions, in which, three helper functions are used to figure out the cached monoidic values for the three parts after splitting. Note that in order to make sure we are only using *splitTree* on non-empty trees, for the "deep" trees, we need to pattern match on all the possible cases of the middle subtree of the tree.

```
mutual
  splitTreeV1 : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
    (p : V → Bool) (i : V) (ft : FingerTree A v n) → V
  splitTreeV1 p i Empty = ∅
  splitTreeV1 p i (Single x) = ∅
  splitTreeV1 p i (Deep {v1} pr Empty sf) with p (i ⊕ v1)
  ... | true = splitV1 p i pr
  ... | false = _⊕_ v1 (splitV1 p (_⊕_ i v1) sf)
  splitTreeV1 p i (Deep {v1} {v2} {v3} pr (Single x) sf) with p (i ⊕ v1)
  ... | true = splitV1 p i pr
  ... | false with p ((i ⊕ v1) ⊕ v2)
  ... | true = _⊕_ v1 (splitV1 p (_⊕_ i v1) (nodeToDigit x))
  ... | false = _⊕_ (_⊕_ v1 v2) (splitV1 p (_⊕_ (_⊕_ i v1) v2) sf)
  splitTreeV1 p i (Deep {v1} pr (Deep pr2 m sf2) sf) with p (i ⊕ v1)
  ... | true = splitV1 p i pr
  ... | false with p ((i ⊕ v1) ⊕ getV (Deep pr2 m sf2))
  ... | true with splitTree p (i ⊕ v1) (Deep pr2 m sf2) {refl}
  ... | split l xs r =
    _⊕_ (_⊕_ v1 (getV l)) (splitV1 p (_⊕_ (_⊕_ i v1) (getV l)) (nodeToDigit xs))
  splitTreeV1 p i (Deep {v1} pr (Deep pr2 m sf2) sf) | false | false =
    _⊕_ (_⊕_ v1 (getV (Deep pr2 m sf2))) (splitV1 p (_⊕_ (_⊕_ i v1)
      (getV (Deep pr2 m sf2)))sf)


splitTreeV2 : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (ft : FingerTree A v n) → V
splitTreeV2 p i Empty = ∅
splitTreeV2 p i (Single {v} x) = v
splitTreeV2 p i (Deep {v1} pr Empty sf) with p (i ⊕ v1)
... | true = splitV2 p i pr
... | false = splitV2 p (_⊕_ i v1) sf
splitTreeV2 p i (Deep {v1} {v2} pr (Single x) sf) with p (i ⊕ v1)
... | true = splitV2 p i pr
... | false with p ((i ⊕ v1) ⊕ v2)
... | true = splitV2 p (_⊕_ i v1) (nodeToDigit x)
... | false = splitV2 p (_⊕_ (_⊕_ i v1) v2) sf
splitTreeV2 p i (Deep {v1} pr (Deep pr2 m sf2) sf) with p (i ⊕ v1)
... | true = splitV2 p i pr
... | false with p ((i ⊕ v1) ⊕ getV (Deep pr2 m sf2))
... | true with splitTree p (i ⊕ v1) (Deep pr2 m sf2) {refl}
... | split l xs r = splitV2 p (_⊕_ (_⊕_ i v1) (getV l)) (nodeToDigit xs)
splitTreeV2 p i (Deep {v1} pr (Deep pr2 m sf2) sf) | false | false =
  splitV2 p (_⊕_ (_⊕_ i v1) (getV (Deep pr2 m sf2))) sf
```

```
splitTreeV3 : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
  (p : V → Bool) (i : V) (ft : FingerTree A v n) → V
splitTreeV3 p i Empty = ∅
splitTreeV3 p i (Single x) = ∅
splitTreeV3 p i (Deep {v1} {_} {v3} pr Empty sf) with p (i ⊕ v1)
... | true = _⊕_ (splitV3 p i pr) v3
... | false = splitV3 p (_⊕_ i v1) sf
splitTreeV3 p i (Deep {v1} {v2} {v3} pr (Single x) sf) with p (i ⊕ v1)
... | true = _⊕_ (_⊕_ (splitV3 p i pr) v2) v3
... | false with p ((i ⊕ v1) ⊕ v2)
... | true = _⊕_ (splitV3 p (_⊕_ i v1) (nodeToDigit x)) v3
... | false = splitV3 p (_⊕_ (_⊕_ i v1) v2) sf
splitTreeV3 p i (Deep {v1} {_} {v3} pr (Deep pr2 m sf2) sf) with p (i ⊕ v1)
... | true = _⊕_ (_⊕_ (splitV3 p i pr) (getV (Deep pr2 m sf2))) v3
... | false with p ((i ⊕ v1) ⊕ (getV (Deep pr2 m sf2)))
... | true with splitTree p (i ⊕ v1) (Deep pr2 m sf2) {refl}
... | split l xs r = _⊕_
     (_⊕_ (splitV3 p (_⊕_ (_⊕_ i v1) (getV l))
     (nodeToDigit xs))
       (getV r))
     v3
splitTreeV3 p i (Deep {v1} pr (Deep pr2 m sf2) sf) | false | false =
  splitV3 p (_⊕_ (_⊕_ i v1) (getV (Deep pr2 m sf2))) sf
```

```
splitTree : {V : Set} {v : V} {{m : Monoid V}} {A : Set} {n : ℕ}
(p : V → Bool) (i : V) (ft : FingerTree A v n) {le : isEmpty ft ≡ false} →
Split (λ _ → FingerTree A (splitTreeV1 p i ft) n)
   (Node A (splitTreeV2 p i ft) n)
   (λ _ → FingerTree A (splitTreeV3 p i ft) n)
splitTree p i Empty { }
splitTree p i (Single x) = split Empty x Empty
splitTree p i (Deep {v1} pr Empty sf) with p (i ⊕ v1)
... | true with splitDigit p i pr
... | split l x r = split (nDigitToTree l) x
   (substFingerTree (cong (λ x₁ → _⊕_ x₁ _) (sym (id1 _)))
   (deepL r Empty sf))
splitTree p i (Deep {v1} pr Empty sf) | false
   with splitDigit p (_⊕_ i v1) sf
... | split l x r = split (substFingerTree
      (cong (λ x₁ → _⊕_ x₁ (splitV1 p (_⊕_ i v1) sf))
      (sym (id1 v1))) (deepR pr Empty l)) x (nDigitToTree r)
splitTree p i (Deep {v1} {v2} pr (Single a) sf) with p (i ⊕ v1)
... | true with splitDigit p i pr
... | split l x r = split (nDigitToTree l) x (deepL r (Single a) sf)
splitTree p i (Deep {v1} {v2} pr (Single a) sf) | false with p ((i ⊕ v1) ⊕ v2)
... | true with splitDigit p (_⊕_ i v1) (nodeToDigit a)
... | split l x r = split
      (substFingerTree (cong (λ x₁ → _⊕_ x₁
         (splitV1 p (_⊕_ i v1) (nodeToDigit a)))
         (sym (id1 v1))) (deepR pr Empty l)) x
         (substFingerTree (cong (λ x₁ → _⊕_ x₁ _)
         (sym (id1 _))) (deepL r Empty sf))
splitTree p i (Deep {v1} {v2} pr (Single a) sf) | false | false
   with splitDigit p ((i ⊕ v1) ⊕ v2) sf
... | split l x r = split (deepR pr (Single a) l) x (nDigitToTree r)
splitTree p i (Deep {v1} pr (Deep pr2 m sf2) sf) with p (i ⊕ v1)
... | true with splitDigit p i pr
... | split l x r = split (nDigitToTree l) x (deepL r (Deep pr2 m sf2) sf)
splitTree p i (Deep {v1} pr (Deep pr2 m sf2) sf) | false
   with p ((i ⊕ v1) ⊕ (getV (Deep pr2 m sf2)))
... | true with splitTree p (i ⊕ v1) (Deep pr2 m sf2) {refl}
... | split ml xs mr
   with splitDigit p (_⊕_ (_⊕_ i v1) (getV ml)) (nodeToDigit xs)
... | split l x r = split (deepR pr ml l) x (deepL r mr sf)
splitTree p i (Deep {v1} pr (Deep pr2 m sf2) sf) | false | false
   with splitDigit p (_⊕_ (_⊕_ i v1) (getV (Deep pr2 m sf2))) sf
... | split l x r = split (deepR pr (Deep pr2 m sf2) l) x (nDigitToTree r)
```

References

[1] The Agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2015. [Online; accessed 10-Dec-2015].

[2] Abel, A. Agda : Equality. `http://www2.tcs.ifi.lmu.de/~abel/Equality.pdf`, 2012. [Online; accessed 20-Mar-2016].

[3] Cormen, T. H. *Introduction to algorithms.* MIT press, 2009.

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

[5] Hinze, R., and Paterson, R. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming 16*, 02 (2006), 197–217.

[6] Lallement, G. *Semigroups and combinatorial applications.* John Wiley & Sons, Inc., 1979.

[7] Okasaki, C. *Purely functional data structures.* Cambridge University Press, 1999.

[8] Sikorski, R., Sikorski, R., Sikorski, R., Sikorski, R., and Mathematician, P. *Boolean algebras*, vol. 2. Springer, 1969.

[9] Sozeau, M. Program-ing finger trees in Coq. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 13–24.