**Razvan Kusztos**

# Verified functional datastructures in Agda

Diploma in Computer Science

Girton College

April 23, 2017

# Proforma

| | |
|---|---|
| Name: | **Razvan Kusztos** |
| College: | **Girton College** |
| Project Title: | **Verified functional datasturcures and algorith** |
| Examination: | **Part II Project** |
| Word Count: | **0[1] (well less than the 12000 limit)** |
| Project Originator: | Dr Timothy Griffin |
| Supervisor: | Dr Timothy Griffin |

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

## Declaration

I, [Name] of [College], being a candidate for Part II of the Computer Science Tripos [or the Diploma in Computer Science], hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

# Contents

# List of Figures

## Acknowledgements

This document owes much to an earlier version written by Simon Moore [**?**]. His help, encouragement and advice was greatly appreciated.

# Chapter 1

# Introduction

## 1.1 Verified Programming

Verification of program correctness is paramount for any successful application. There are many ways and paradigms used in industry that enforce this ideas. Most commonly, hard-coded tests are run against the program. In this project I will focus on formal verification, that is, checking that a program is correct under some formal, mathematical modeling, rather than through its behaviour.

Formal verification has had successful application in areas such as cryptography (Criptol), hardware specification (The verification system in Verilog) or compiler construction (CompCert). A more general approach implies verifying arbitrary software programs. Automated theorem provers such as Coq, Isabelle or Agda achieve this goal. Although their principles have been around for (..), their industrial application remains still niche (CompCert is an example project using Coq). Isabelle's open archive of proofs contains mainly proofs concerned with mathematical objects, with very few example of algorithms or data structures [1].

Languages like Agda have been designed first and foremost as general programming languages, with formal verification capacities on the side. Their development is in a relatively early stage, and are mainly employed for research.

## 1.2 Functional Data Structures

Functional data structures have been long thought to be inefficient and belonging solely to academia. Solving past constraints are however turning the tides, with multicores and lots of memory easily accessible. The work of Okasaki [**?**], namely his book,"Purely Functional Data Structures", has gone a great way in solving the imbalance between functional data structures and the vast collection of efficient imperative structures. His discussion of implicit recursive slowdown

[1] https://www.isa-afp.org

and numerical representations of types has inspired the advent of many data structure. An example at which I will come back later is the Finger Tree.

The main constraint that functional data structure have imposed on them is that they are persistent. That is, any destructive operation, such as updating an element in a list is expected to preserve in memory both the previous version and the new version. This constraint is not de facto in many imperative implementation (consider updating an element in a C array). Solutions for achieving this goal in an imperative environment are tedious and incur and extra cost. By working functionally, we need to work on top of this constraint – point where the implicit recursive slowdown comes at the rescue.

Having tackled the problem of efficiency that has been long believed to be an argument against functional programming, the obvious advantage of the lack of side effects can be fully appreciated. The lack of side effects makes reasoning about functions a more tractable problem, and therefore aids coming up with verified programs.

## 1.3 Dependent Types

In traditional functional programming languages such as SML, Ocaml or Haskell, there is a clear barrier between types and values. In a dependently typed programming language, such as Agda, Coq or Idris, this distinctions fades away. Types and values are placed under the same grammar, introducing general terms. Whereas before types could depend on other types (i.e. parametric polymorphism), types can now also depend on values.

Careful use of this expressive power can aid the user by reducing much of the run-time checks needed to ensure proper execution of the program.

Consider, for example, the case of performing the sum of two n-dimensional vectors. In a non-dependent setting, all the vectors would have the same time, regardless of their size. Implementing a correct summation method would entail checking at run-time whether the vectors have the same length. With a dependent typed language, we can enforce this in the type.

Traditionally, proofs about the programs are presented in a separate environment, most commonly pen and paper. Since we allow types to depend on values, we can reason about both code and proofs in the same environment (under the Curry Howard isomorphism – see Section whatever)

## 1.4 Summary

In the next section I will introduce Agda, explain how logic reasoning fits tightly with the theoretical basis of Agda and the FingerTree data structure.

In the implementation section I will present an implementation of FingerTree, made possible by dependent typing, which ensures correctness, as well as proofs related to this data structure. I will show how specializations of the FingerTree to other data-structures, such as Random Access Sequences or ... maintain the correctness properties. This can be seen as a software engineering approach using dependent types. Finally, I will outline some difficulties which arise because of Agda's totality and show some ways of overcoming them.

In the evaluation section, I will ....

# Chapter 2

# Preparation

## 2.1 Agda

Agda is a dependently typed programming language based on the predicative Martin Lof type theory. It was introduced in Ulf Norell's phd thesis, as a bridge between practical programming and the world of well-established automated theorem provers (like Coq).

I have chosen to implement this project in Agda for a number of reasons:

- dependent types.
- simplicity, both in available features and the syntax.
- a suitable learning curve.
- lack of predefined tactics, which makes it easier to observe patterns in programming, errors or issues.[1]
- specifically for implementing Finger Trees, for reasons that I will come back to in section whatever.

## 2.2 Curry Howard Isomorphism

The main mechanism employed in computer assisted proofs with dependent types is the observation (due to Curry) that there exists a one-to-one correspondence between propositions in formal logic and types. The original example, given by Howard is the bijection between the intuitionistic natural deduction and the simply typed lambda calculus. Using this principle, the predicative quantifier ∀ corresponds to a dependent product [**?**] , enabled by dependent types.

Althugh a comparison between CiC and ITT would be interesting, I could not find any literature on this topic. The development of Coq was influenced

[1]one is free to define their tactics, using reflection – an example is in using the monoid solver

| Type system | Logic |
|---|---|
| Simply Typed LC | Gentzen Natural Deduction (Gentzen) |
| PLC | Second Order Propositional Logic |
| CoC | Higher Order Predicate Logic [2] |
| ITT | Higher Order Predicate Logic - basis of Agda |
| CiC | Higher Order Predicate Logic - basis of Coq |

Table 2.1: Curry Howard Relation between various systems

by Martin Lof's theory through the presence of inductive types[**?**]. A notable difference is that Coq's sort system differentiates between Prop (the type of propositions) and Type(i), whereas Agda only has a family Set(i).

In Agda, the Curry Howard relations introduce the following recipes for generating proofs.

| Logic Formula | ITT Notation | Agda Notation |
|---|---|---|
| $\bot$ | $\emptyset$ | $\bot$ |
| $A \vee B$ | $A + B$ | $A \uplus B$ |
| $A \wedge B$ | $A \times B$ | $A \times B$ |
| $A \supset B$ | $A \rightarrow B$ | $A \rightarrow B$ |
| $\exists x : A.B$ | $\Sigma x : A.B$ | $\Sigma\ A\ B$ |
| $\forall x : A.B$ | $\Pi x : A.B$ | $(x : A) \rightarrow B$ |

Table 2.2: Curry Howard Relation in Agda

The proof of a proposition in this logic is equivalent to building a term that has the corresponding type.

## 2.2.1 Some example proofs in Agda.

A very important family of types in Agda is the propositional equality.

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

Having a single constructor, it corresponds to the proposition that two elements of the same type can only be equal if they are in fact the same element.

**Associativity of Natural Numbers.** Consider proving some properties of the natural numbers, such as associativity:

```
+assoc : ∀ (x y z : ℕ) → (x + (y + z)) ≡ ((x + y) + z)
+assoc zero y z = refl
+assoc (suc x) y z rewrite +assoc x y z = refl
```

The type indicates what is being proved, whereas the definition consists of an example proof.

**List reverse properties** Consider the following implementation of reverse, using a helper function:

```
rev′ : {A : Set} → (List A) → (List A) → List A
rev′ [] ys = ys
rev′ (x ∷ xs) ys = rev′ xs (x ∷ ys)

rev : {A : Set} → List A → List A
rev xs = rev′ xs []
```

The main goal of this exercise is to prove that $\forall xs : ListA, rev(rev(xs)) \equiv xs$. We first need to prove some helper statements about reverse, of which I have included the type declarations[3]

```
rev′−rev : ∀ {A}
  → (xs : List A)
  → (ys : List A)
  → rev′ xs ys ≡ (rev xs) ++ ys


rev−app−lemma : {A : Set}
  → (xs : List A)
  → (ys : List A)
  → rev (xs ++ ys) ≡ (rev ys) ++ (rev xs)
```

Finally, the main proof is presented using the Equational Reasoning module, which aids writing more readable proofs. This is the format in which most proofs are written throughout the dissertation.

```
rev−lemma : {A : Set}
    → (xs : List A)
    → rev (rev xs) ≡ xs
rev−lemma [] = refl
rev−lemma (x ∷ xs) =
  begin
    rev (rev′ xs (x ∷ []))
  ≡⟨ cong rev (rev′−rev xs (x ∷ [])) ⟩
    rev ((rev xs) ++ x ∷ [])
  ≡⟨ rev−app−lemma (rev xs) (x ∷ []) ⟩
    x ∷ rev (rev xs)
  ≡⟨ cong (λ a → x ∷ a) (rev−lemma xs) ⟩
    x ∷ xs
  ∎
```

[3]full implementation is in the appendix

It is worth emphasizing the dual use of the typing system, both for proving correctness and providing abstraction. The type declaration is many times sufficient for understanding the purpose of the implementation.

### 2.2.2 Induction

As in the previous example, we can see that induction is a key means of proof. In this example, we perform a structural induction on the possible constructor of *a* as a natural number. In the second case, we also perform a natural mathematical induction step. Assuming that *+assoc a b c* holds for some a, b and c, we want to show that *+assoc (a + 1) b c* holds.

## 2.3 Totality

Agda and Coq are both examples of total function programming languages. This constrains all the defined functions to be total.

**In a mathematical sense**, this means that they must be defined for all inputs. Consider the declaration of the head of a list.

```
head : ∀ {A} → List A → A
head (x :: xs) = x
```

Although this function is acceptable in Haskell, it will not type check in Agda. To mitigate this inconvenience, it is straightforward to use the Maybe monad.

**In a computational sense** however, functions must also be strongly terminating on all the inputs. This has to do with the logical consistency. We trade off the Turing completeness for ensuring that all constructed terms correspond to valid proofs.

However, due to a well known result, termination checking is an undecidable problem. For this reason, Agda (and Coq) have to use heuristics to determine whether recursive calls will eventually terminate.

The way Agda deals with this problem is by ensuring that with every call to the function in a recursion stack, its argument becomes structurally smaller [**?**] [4]. The ordering relation is recursively defined by

$$\forall i : \mathbb{N}.\forall w : Set_i.w < C(..., w, ...)$$

where C is an inductive data type constructor.

[4]http://www2.tcs.ifi.lmu.de/ abel/foetuswf.pdf

### 2.3.1 Sized types

One can very simply imagine operations that hide this structural less-than relation. For this reason, the concept of 'Sized types' has been introduced. Under this paradigm, the data structure should be indexed by a type for which the structural relation is obvious at all times. Such examples are Nat or Size present in the standard library.

The difficulty of using either of these will become apparent in the context of Finger Trees. However, it is worth noting here that the incompleteness [**?**] of the termination checker is making programming unnecessarily hard in some cases.

## 2.4 Correct Data Structures in Agda

Much of the effort in programming goes to preserving invariants, i.e. facts the programmer needs to ensure about data structures in order for them to behave as expected. That is, we need to make sure that the following statements hold:

- the constructors can only produce correct[5] instances
- any function that takes as input a correct instance can only output a correct instance

If the type of the data structure ensures the invariants we want, both these propositions become true via the Curry Howard isomorphism.

**Sorting Lists.** Consider, for example, implementing a function that sorts lists. That is, the input is a normal list and the output should be a sorted list containing all the elements in the argument list. We can provide a type encoding of what it means to be a sorted list containing some set of elements.

```
data SortedList : {n : ℕ} → Vec A n → Set where
  [] : SortedList []
  [_] : (x : A) → SortedList (x ∷ [])
  _∷_ : ∀ {n : ℕ} {ys : Vec A n} {zs}
    → (x : A)
    → (xs : SortedList ys)
    → (all (λ a → x ≤ a) ys ≡ true)
    → (x ins ys ≡ zs)
    → (SortedList zs)
```

Here, the *all* function tests whether a predicate holds in the entirety of a list, and the _*ins*_≡_ operator should be read as: If x ins xs ≡ ys, then I can insert x somewehre in xs to obtain ys.

[5]i.e. correct with respect to the invariants

In order to define a correct sorting function, we assign the following type signature [6]

```
sort : ∀ {n : ℕ} → (xs : Vec A n) → (SortedList xs)
```

This definition can be read in two ways: From a **logical** point of view, it is a proof that all lists can be sorted. However, from a **computational** point of view, it represents the type signature of all sorting functions that can be coded in Agda. [7]

This example should prove the expressiveness that dependent typing makes available, as well as its capacity for abstraction. In implementing the Finger Trees, I will aim for a similar verification method.

### 2.4.1 Nested Types

In order to move on to the Finger Trees, I first have to introduce an alternative way through which a certain family of invariants could be kept true, without using dependent types.

Nested types[**?**], also known as irregular types or polymorphic recursions, can aid in enforcing structure in data types, such as full binary trees, cyclic structures [**?**] or square matrices [**?**]. The idea is that when declaring an inductive data structure, occurrences of the type on the right hand side are allowed to appear with different type parameters.

Agda is particularly expressive since it allows declaring functions on such data types, as opposed to SML and older versions of Haskell.

*List* is an example of a **regular** data type. The recursive call to List is restricted to the type parameter A

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

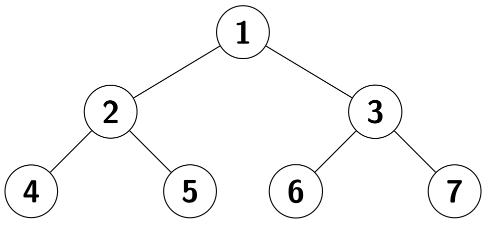
A slight modification, which recursively calls Lists with the type parameter A x A is used to represent a full binary tree (this has been introduced as Nest [**?**]).

```
data Nest (A : Set) : Set where
  Nil : Nest A
  Cons : A → Nest (A × A) → Nest A


example : Nest ℕ
```

[6] I have encoded the lists as length-indexed vectors in order to ensure that the termination checker accepts my definitions. It would not have worked by simply using Lists.

[7] I have implemented, as an example, the selection sort which is present in the appendix

```
example = Cons 1 (Cons (2 , 3) (Cons ((4 , 5) , (6 , 7)) Nil))
```



The same principles apply in the case of Finger Trees, which is based on a full 2-3 tree, with labels in the leafs only.

## 2.5 Notes

- add the stuff about starting point, requirements and whatever everyone talks about.

# Chapter 3

# Implementation

## 3.1 FingerTrees - Introduction

Finger Trees are a data structure introduced by Ralph Hinze and Robert Patinsson, based on Okasaki's principle of implicit recursive slowdown.
Initially meant as a double ended queue with constant amortized time append, their structure, together with the cached measurements, allow specialization to Random Access Sequences, or Priority Queues by simple instantiation.

The underlying structure is that of a full 2-3 tree, with labels solely at the leafs. For efficient insertion and deletions, the tree is surrounded by buffers at each level, which amortize the cost of appending at either end. Furthermore, the data structure is accompanied by a measurement function and a binary operator, such that the reduced measures of all nodes in a subtree is cached in all the joints. These are necessary for searching or splitting.

### 3.1.1 Invariants

The efficiency is achieved by keeping two invariants on the data structure:

- The tree is full and all the leaves occur on the last level.
- The measurements are correct[1]

[1] •

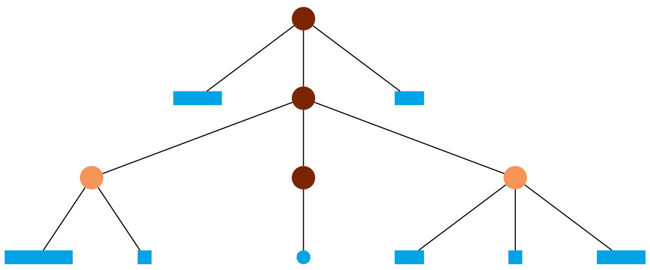


Figure 3.1: Example finger tree

Working in Agda, a dependently typed language, which moreover allows the use of nested types, we can keep these invariants soley in the type of the Finger Tree. More specifically,

- The nested typing will ensure fullness of the tree.
- Choosing measurements as the type index ensures their correctness.

### 3.1.2 Previous Work

Finger Trees have been previously implemented and proved correct. I will outline some previous results, as well as their limitations, providing more incentives for this dissertation. I have included all related implementations I could find and I do not guarantee they are the only ones.

- Basic Implementation in Agda.
  This version can be found on GitHub[2]. Its mentioned intention is to closely follow the original paper. It also uses introduces the idea of Sizing, although only in the type declaration (and constructors). Since the constraints are not present in functions that modify the data type, they do not really aid correctness proofs. It has no proofs associated with it, and it didn't type check on my machine.

- Implementation in Coq.
  This implementation is provided by Matthiew Souzeau[**?**] as a proof of concept for Russell, a Coq extension. I have drawn great inspiration from that paper, and I was particularly drawn by its small caveat, onto which I will return at the end of this chapter. Although a full and working implementation, I argue that this dissertation is valuable in its own, given my aforementioned reasons for choosing Agda as the programming language, and providing a solution to some caveats.

- Implementation in Isabelle.
  Another working implementation has been done in Isabelle. However, this implementation diverges from the original specification of the data structure, removing the nesting. The two invariants that I have mentioned are maintained explicitly, due to the lack of dependent types.

  The implementation of this data structure in both Coq and Isabelle, two established theorem provers might argue both for the complexity involved, and for its interesting particularities.

[2].

## 3.2 Finger Trees - Implementation

### 3.2.1 Data type declaration

The Finger Tree is originally polymorphic in two types:

- **A** : this is the type of the elements that are contained in the Finger Tree
- **V** : this is the type of measures.

In order to to mimic Haskell's typeclasses, I have carried around, for each A and V, two constructs:

- **Monoid**[3] **V**: which contains a neutral element(**ε**), a binary operator(**•**), and the monoid axioms, and a comparison operator.
- **Measured A V** : which consists of a norm function : $\|\_\| : A \to V$

**Node** corresponds to nodes in the underlying 2-3 tree implementation, having two constructors that contain two and respectively three items. Moreover, **Node**s can only be constructed if provided with a measurement tag and a correctness proof.

```
data Node {a} (A : Set a)(V : Set a )
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄ : Set a where
  Node2 : (v : V)
    → (x : A) → (y : A)
    →  (v ≡ ‖ x ‖ • ‖ y ‖)
    → Node A V
  Node3 : (v : V)
    → (x : A) → (y : A) → (z : A)
    → (v ≡ ‖ x ‖ • ‖ y ‖ • ‖ z ‖)
    → Node A V
```

**Digit**s were in presented in the original paper as lists, but this definition limits them to have one to four elements.

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

Finally, the **FingerTree** is a family of types, indexed by a measurement $\mu$. The measurement's correctness is enforced in all the constructors. Note the nested

[3]see AlgebraStructures.agda

type and the universal quantification over possible sizes for the recursive call. Apart from the measurement addition, the rest corresponds to the original paper.

```
data FingerTree {a} (A : Set a)(V : Set a)
      ⦃ mo : Monoid V ⦄
      ⦃ m : Measured A V ⦄ :
      {μ : V} → Set a where
   Empty  :  FingerTree A V {ε}
   Single :  (e : A) → FingerTree A V {‖ e ‖}
   Deep   :  {s : V}
      → (pr : Digit A)
      → FingerTree (Node A V) V {s}
      → (sf : Digit A)
      → FingerTree A V {measure−digit pr • s • measure−digit sf}
```

**Smart Constructors** We also build smart constructors that fill in the measurement, provided with the appropriate number of elements

```
node2 : ∀ {a} {A : Set a}{V : Set a }
   ⦃ mo : Monoid V ⦄
   ⦃ m : Measured A V ⦄
   → A → A → Node A V
node2 x y = Node2 (‖ x ‖ • ‖ y ‖) x y refl

node3 : ∀ {a} {A : Set a}{V : Set a }
   ⦃ mo : Monoid V ⦄
   ⦃ m : Measured A V ⦄
   → A → A → A → Node A V
node3 x y z = Node3 (‖ x ‖ • ‖ y ‖ • ‖ z ‖) x y z refl
```

### 3.2.2 Graphical example

Considering Figure, 3.1, I have colour-coded the nodes as follows:

| Symbol | Constructor |
|---|---|
| ● | Deep |
| ● | Node |
| ▬ | Digit (of various lengths) |
| • | A |

### 3.2.3 Indexing on the measurement

The reason for indexing on the measurement is twofold. Firstly, we index on the measurement in order to verify the correctness of the measurement in operations such as appending an element or splitting. Secondly, the index was chosen in order to allow implementing a 'size' that depends on all elements in the finger tree.

Consider a sizing that would take into account the shape of the tree only (as it is the case of Size described previously).

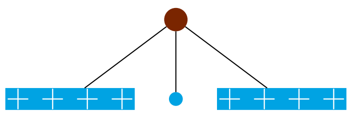


Figure 3.2: Bigger Finger Tree

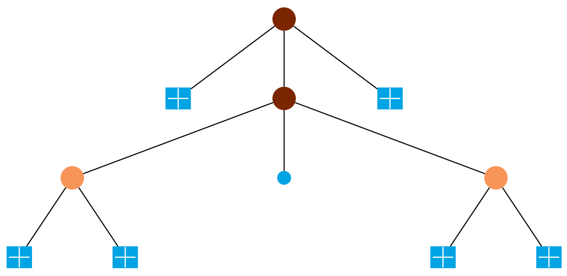


Figure 3.3: Smaller finger Tree

In Figures 3.2 and 3.4, it is an ambiguous question which of the two trees should be considered to have a bigger size. *Size* implements a partial order between data types, with no definite reference points, whereas here we are concerned with an absolute order.
As suggested by Matthew Souzeau [**?**], a sizing that reflects the number of elements is ideal. We can use the already existing measurement index to achive this goal.

### 3.2.4 *Cons* and *Snoc*

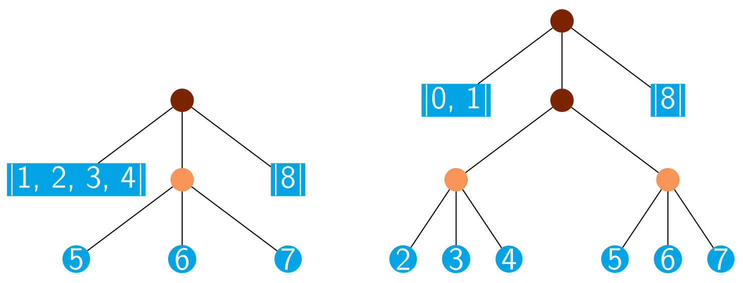
*Cons* is the operator that appends an element to the left of the finger tree.

The implementation is straight forward if there is room in the left-most digit. Otherwise, we have to recursively insert and reform parts of the finger tree.

Ultimately, for the correctness part, we are concerned whether the output tree is a correct finger tree (enforced by the type) with a correct measurement.

That is, by inserting an element x,

$$\|x \triangleleft ft\| = \|x\| \cdot \|ft\|$$



(a) Before Cons (b) After Cons

Figure 3.4: Recursive cons operation

```
_◁_ : ∀ {a} {A : Set a} {V : Set a}
    ⦃ mo : Monoid V ⦄
    ⦃ m : Measured A V ⦄
    {s : V}
    → (x : A)
    → FingerTree A V ⦃ mo ⦄ ⦃ m ⦄ {s}
    → FingerTree A V ⦃ mo ⦄ ⦃ m ⦄ {‖ x ‖ • s}
```

Each case in the definition is accompanied by a proof that the measurement of the output finger tree is correct with respect to the topmost definition. These proofs are all derived from monoid properties imposed on the operation.

```
_◁_ {l} {A} {V} ⦃ mo ⦄ a Empty
    rewrite (Monoid.ε−right mo) ‖ a ‖
    = Single {l}{A}{V} a
_◁_ {l} {A} {V} ⦃ mo ⦄ ⦃ m ⦄ {.(‖ e ‖)} a (Single e)
    rewrite assoc−lemma1 ⦃ mo ⦄ ⦃ m ⦄ a e
    = Deep (One a) Empty (One e)
a ◁ Deep (One b) ft sf
    rewrite •−assoc (‖ a ‖) (‖ b ‖) (measure−tree ft • measure−digit sf)
    = Deep (Two a b) ft sf
a ◁ Deep (Two b c) ft sf
    rewrite •−assoc (‖ a ‖) (‖ b ‖ • ‖ c ‖) (measure−tree ft • measure−digit sf)
    = Deep (Three a b c) ft sf
a ◁ Deep (Three b c d) ft sf
    rewrite •−assoc (‖ a ‖) (‖ b ‖ • ‖ c ‖ • ‖ d ‖) (measure−tree ft • measure−digit sf)
    = Deep (Four a b c d) ft sf
a ◁ Deep (Four b c d e) ft sf
    rewrite assoc−lemma2 a b c d e (measure−tree ft) (measure−digit sf)
    = Deep (Two a b) ((node3 c d e) ◁ ft) sf
```

The Finger Tree operations are symmetric on the middle, so the construction of the snoc operator is exactly dual. It's implementation is provided in the source code.

### 3.2.5 *toList*

We will need to prove properties of the finger trees with respect to the elements they contain and their relative position. Therefore, it is handy to be able to transform them to lists, as they encode these properties simply.

This is the conversion between a finger tree and a list[4][5]

```
toList−ft : ∀ {a}{A : Set a}{V : Set a }
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄ {s : V}
  → FingerTree A V {s}
  → List A
toList−ft Empty = []
toList−ft (Single x) = x ∷ []
```

### 3.2.6 Proving correctness of the *cons* operator

Assuming that the implementation of list is correct, we can define the correctness of the cons operator as follows[6]

```
cons−correct : ∀ {a}{A : Set a}{V : Set a }
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄
  {v : V} →
  (x : A) →
  (ft : FingerTree A V {v}) →
  toList−ft (x ◁ ft) ≡ (x ∷ []) ++ (toList−ft ft)
```

### 3.2.7 View from the Left/Right

As suggested in the original paper, the structure of the finger tree is complicated and users can benefit from a higher level representation. Furthermore, we have no mechanism yet of 'deconstructing' a sequence.

In this case, we will 'view' each finger tree as the product between an element and the remaining finger tree.

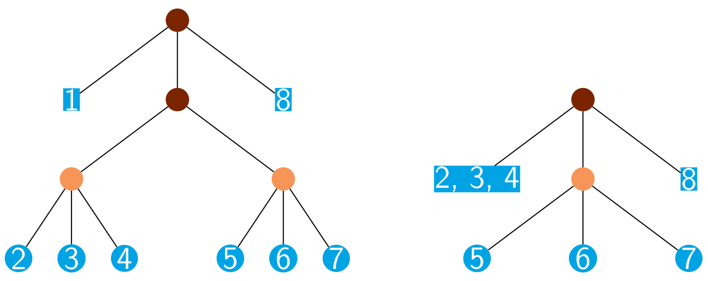
[4]toList-dig is a straightforward conversion.
[5]flatten-list transforms a list of Nodes into a list of As.
[6]An example term of this type is in the appendix

```
data ViewL {a}(A : Set a)(V : Set a)
          ⦃ mo : Monoid V ⦄
          ⦃ m : Measured A V ⦄ :
          {s : V} → Set a where
  NilL :  ViewL A V {ε}
  ConsL : ∀ {z}
          (x : A)
          → (xs : FingerTree A V {z})
          → ViewL A V {‖ x ‖ • z}
```

This data type also enforces the correctness of the measurement, being indexed in the same way as the finger tree.

We need to implement a procedure that transforms between the two.

As it is the case of the Cons operator, most cases are superfluous. The complicated case arises when the leftmost digit contains a single entry.



(a) Before ViewL (b) After ViewL

Figure 3.5: ViewL operation (only included the tails)

As you can see, the composition of cons and viewL is not a no-op, but they both preserve the order of the elements.

```
mutual

  viewL : ∀ {a} {A : Set a}{V : Set a }
        ⦃ mo : Monoid V ⦄
        ⦃ m : Measured A V ⦄
        {i : V} → FingerTree A V {i}
        → ViewL A V {i}
  viewL Empty = NilL
  viewL ⦃ mo ⦄ ⦃ m ⦄ (Single x)
    rewrite sym (Monoid.ε−right mo ‖ x ‖)
    = ConsL x Empty
  viewL ⦃ mo ⦄ ⦃ m ⦄ (Deep pr ft sf)
    rewrite measure−digit−lemma1 ⦃ mo ⦄ ⦃ m ⦄ pr ft sf
    = ConsL (head−dig pr) (deepL (tails−dig pr) ft sf)
```

```
deepL : ∀ {a}{A : Set a}{V : Set a }
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄
  {s : V}
  → (pr : Maybe (Digit A))
  → (ft : FingerTree (Node A V) V {s})
  → (sf : Digit A)
  → FingerTree A V {measure−maybe−digit pr • s • measure−digit sf}
-- deepL pr ft sf = {! !}
deepL (just x) ft sf = Deep x ft sf
deepL nothing ft sf with viewL ft
deepL ⦃ mo ⦄ ⦃ m ⦄ nothing ft sf | NilL
  rewrite (Monoid.ε−left mo) (ε • measure−digit sf)
    | (Monoid.ε−left mo) (measure−digit sf)
  = toTree−dig sf
deepL nothing ft sf | ConsL (Node2 x x₁ x₂ r) x₃
  rewrite r
    | assoc−lemma3 x₁ x₂ (measure−tree x₃) sf
  = Deep (Two x₁ x₂) x₃ sf -- Deep (Two x₁ x₂) x₃ sf
deepL nothing ft sf | ConsL (Node3 x x₁ x₂ x₃ r) x₄
  rewrite r
    | assoc−lemma4 x₁ x₂ x₃ (measure−tree x₄) sf
  = Deep (Three x₁ x₂ x₃) x₄ sf -- Deep (Three x₁ x₂ x₃) x₄ sf
```

### 3.2.8 Proving Correctness of *viewL*

We can proceed in an analogous way to the correctness of cons, by constructing an appropriate to-list conversion for views, and then proving that the list representations coincide.

However, we stumble upon simple property that is unnecessarily hard to prove.

That is, we would like to prove that $viewL(ft) \equiv NilL \iff ft \equiv Empty$. This fact is obvious given the associated definitions. Unfortunately, *Propositional Equality* cannot allow a term of this form, since for an arbitrary $\sigma \in V$, *FingerTree A V* $\{\sigma\}$ does not have the same type as Empty. (i.e. *FingerTree A V* $\{\epsilon\}$)

Changing the statement of the problem slightly to ∀ *ft : FingerTree A V* $\{\epsilon\}$ $viewL(ft) \equiv NilL \iff ft \equiv Empty$ allows the definition. However, the typechecker will get stuck in trying to pattern match on ft. The reason is that it cannot find terms in $V$ to satisfy the constrained indexing.

Indeed, if we try to prove this statement on a simpler version of *FingerTree* that is indexed by Size [7], it is a straightforward exercise:

```
view−lemma3 : ∀ {a}{A : Set a} {V : Set a }
   ⦃ mo : Monoid V ⦄
   ⦃ m : Measured A V ⦄
   → (ft : FingerTree A V)
   → (viewL ft ≡ NilL)
   → (ft ≡ Empty)
view−lemma3 Empty p = refl
view−lemma3 (Single x) ()
view−lemma3 (Deep x x₁ ft x₂) ()
```

A solution to this issue was suggested by McKinna[?], using an alternate implemenation of equality – *Heterogeneous Equality*, which works across types.

Using heterogeneous equality, we can now write the type of the original statement in Agda, as well as pattern-match on the finger tree. However, some problems related to the *with* construct resurface.

### 3.2.9 *with* and *rewrite*

The implementation abounds in use of *with* and *rewrite* statements. *with* allows, inspired by the work of McBride and McKinna [?], to pattern match on an intermediate computation. *rewrite* replaces an expression on the left hand side, by making use of a supplied equality relation.

Their use could not be avoided in the implementation of operators that act on indexed data types. As part of the type checking, Agda has to unify the expected type of the result and the actual type of the result.

**Example.** Consider the implementation of append on Vectors, but with a slight difference in terms of the index of the result. We return a vector of size $m + n$ instead of $n + m$. The type-checker cannot prove that $+$ commutes, since it is not superfluous. This does not type-check.

```
append : ∀ {a n m} → {A : Set a}
   → Vec A n
   → Vec A m
   → Vec A (m + n)
append [] ys = ys
append (x ∷ xs) ys = x ∷ append xs ys
```

The same situation arises in the proofs about correctness of the measure semantics. We have to provide the proof. Due to the nature of the *Builtin.Equality*

[7]so that Empty can assume any size as long as it is smaller than its FingerTree derivatives (such as Deep sf Empty pr)

relation, I could not find a way to expand the functions appropriately, as suggested in the documentation [8]

**Discussion about *with*.** There is a number of issues related to the *with* statement that I have stumbled upon.

- Computing terms that are hidden behind a *with* statement requires recomputation of the expression present in the *with* clause. In the *FingerTree* case, this occurs because of the *rewrite* statements present throughout the implementation of *cons*, *viewL*, *deepL* etc. This causes a mild inconvenience by having to reiterate the same *rewrite*s whenever one writes proofs about those definitions.

- Whenever an argument of a function is hidden by a *with abstraction*, the definition of that function cannot use further with statement containing the abstracted expression of the argument. This is discussed in the documentation: Ill-typed with-abstraction [9]
  This is the reason proving correctness of ViewL seemed complicated.

- The type of terms hidden by *with abstraction* is not available, as the feature of type checking in this conditions has not been implemented.

- There are cases in which the termination-checker is confused in the presence of *with*

  Consider this example, where we try to append an element at the end of a list.

  ```
  snoc : ∀ {A} → A → List A → List A
  snoc x xs with xs
  snoc x xs | [] = x ∷ []
  snoc x xs | y ∷ ys = y ∷ (snoc x ys)
  ```

Taking this issues into account, I have tried to come up with solutions to some of them. I will present them after finishing the main implementation of the Finger Tree.

I should note here that the proofs that I am making are still providing a powerful verification, since

- The FingerTree maintains all invariants

- The measurement semantics are preserved an used sanely.

[8]http://agda.readthedocs.io/en/latest/language/with-abstraction.html
[9]https://agda.readthedocs.io/en/v2.5.2/language/with-abstraction.html

### 3.2.10 Folding

We can further implement the fold operation and show its correctness. I will only present, as an example the fold-left implementation.

On lists, foldl f s [x, y, z] = f (f (f s x) y) z. We can extend this to FingerTrees. Defining folds on **Node** and **Digit** is trivial, since they are just length constrained lists.

As a helper function, I define an operation to flatten a list of nodes:

```
flatten−list : ∀ {a}{A : Set a}{V : Set a }
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄
  → List (Node A V)
  → List A
flatten−list [] = []
flatten−list (x :: xs) = (toList−node x) ++ (flatten−list xs)
```

We can then implement the foldl on finger trees as:

```
foldl : ∀ {a} {A : Set a} {V : Set a}
    {W : Set a}
    ⦃ mo : Monoid V ⦄
    ⦃ m : Measured A V ⦄
    {s : V}
    → (W → A → W)
    → W
    → FingerTree A V {s}
    → W
foldl f i Empty = i
foldl f i (Single e) = f i e
foldl {W = W} f i (Deep pr ft sf) =
  foldl−dig f (foldl (foldl−node f) (foldl−dig f i pr) ft) sf
```

### 3.2.11 Proving correctness of Fold Left

Next, we will show that the previous implementation is sane, by seeing whether folding over a finger tree is equivalent to folding over its list representation.

```
foldl−correct : ∀ {a} {A : Set a}{V : Set a}
  {W : Set a}
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄
  → {s : V}
  → (f : W → A → W)
  → (σ : W)
```

```
    → (ft : FingerTree A V {s})
    → (foldl f σ ft ≡ Data.List.foldl f σ (toList−ft ft))
```

Furthermore, fold-left has some interesting properties when it comes to its relation to the measurement. We can prove that if we fold over the finger tree using the **Monoid** and the **Measure**[10] over which it is instantiated, we obtain the same result as the measure. This result is important as a sanity check of the measure semantics we are trying to preserve throughout the implementation:

```
foldl−lemma0 : ∀ {a} {A : Set a} {V : Set a}
    ⦃ mo : Monoid V ⦄
    ⦃ m : Measured A V ⦄
    → {s : V}
    → (v : V)
    → (ft : FingerTree A V {s})
    → (foldl foldfun v ft ≡ v • s)
```

## 3.2.12 Splitting

Splitting is an extension of the ViewL paradigm, by allowing extraction of elements arbitrarily deep in the FingerTree. It consists of a left side, a middle element, and a right side.[11]

The same issues occur, as it was the case of viewL. I have tried in this implementation to keep the usage of *with* at a minimum. This turned out to be a very difficult task, so correctness of this method can only be provided in terms of its measure, which is being taken care of by the indexing.

The split is done over a boolean predicate, $p \in V \to Bool$ and a starting value, $i \in V$. The method iterates through the FingerTree, element by element, at each step increasing $i$ by the measure of the current element, $\|x\|$. The split is done when the value of the predicate $p(i) = True$, and the element at which this change occurs becomes the middle.

As with ViewL, we create an additional data-type that will represent the result. It is indexed by the measurement, ensuring correctness.

```
data Split−d {a} (A : Set a) (V : Set a)
      ⦃ mo : Monoid V ⦄
      ⦃ m : Measured A V ⦄ :
      {μ : V} → Set a where
   split−d : ∀ {μ₁ : V} {μ₂ : V}
      → (FingerTree A V {μ₁})  -- left side
      → (x : A)                -- middle
      → (FingerTree A V {μ₂})  -- right side
```

[10]foldfun v x = v • ‖ x ‖

[11]the sides are correct FingerTrees

```
    → Split−d A V {μ₁ • ‖ x ‖ • μ₂}
```

Since this implementation is long a full of necessary proofs about the types, I will only provide the most important snippets.

**Discussion** This implementation was also made difficult because of the partiality of the function in the original paper. Wrapping things in the Maybe monad can confuse the type-checker at times.

**The main procedure** pattern matches on the constructors for the Finger Tree provided as argument. Notice that the typing already maintains the invariant.

```
split−Tree : ∀ {a} {A : Set a} {V : Set a}
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄
  {μ : V} -- type class information
  → (p : V → Bool) → (i : V) -- predicate and inital value
  → (ft : FingerTree A V {μ}) -- argument
  → Maybe (Split−d A V {μ})
split−Tree p i Empty
  = nothing -- cannot split an empty tree
split−Tree p i (Single e)
  = just (split−Tree−single p i e) -- superfluous case
split−Tree p i (Deep pr ft sf)
  = just (split−Tree−if p i pr ft sf vpr refl vft refl) -- recursive case
  where
    vpr = p (i • (measure−digit pr))
    vft = p ((i • measure−digit pr) • measure−tree ft)
```

The **split-Tree-if** function splits the computation in three cases, depending where the predicate changes to *True*. This could happen after the during the prefix **pr**, during the nested finger tree **ft** or during the suffix **sf**

```
split−Tree−if : ∀ {a} {A : Set a} {V : Set a}
  ⦃ mo : Monoid V ⦄
  ⦃ m : Measured A V ⦄
  {μ : V}
  → (p : V → Bool) → (i : V)  -- predicate and initial value
  → (pr : Digit A)  -- prefix
  → (ft : FingerTree (Node A V) V {μ}) -- nested tree
  → (sf : Digit A)  -- suffix
  → (vpr : Bool)  -- value of predicate after prefix
  → (vpr ≡ p (i • measure−digit pr)) -- correctness check
```

```
    → (vft : Bool)   -- value of predicate after tree
    → (vft ≡ p ((i • measure−digit pr) • (measure−tree ft))) -- check
    → Split−d A V {(measure−digit pr) • μ • (measure−digit sf)}
split−Tree−if p i pr ft sf false pr1 false pr2
    = split−Tree2 p ((i • measure−digit pr) • (measure−tree ft)) pr ft sf
    -- case2 : predicate becomes true in suffix or not at all
split−Tree−if p i pr ft sf false pr1 true pr2
    = split−Tree3 p i pr ft sf (sym pr1) (sym pr2)
    -- case3 : predicate becomes true in tree
split−Tree−if p i pr ft sf true pr1 vft pr2
    = split−Tree1 p i pr ft sf
    -- case1 : predicate becomes true in prefix
```

The full implementation requires more advanced agda syntax, as well as the implementation of the ViewR and deepR. It is present in the Appendix.

## 3.3 Other recursive definitions

The difficulties of writing a recursive function that outputs a value of a type dependent on one of the arguments become apparent with the example of reversing.

The implementation of reverse is straight-forward in terms of folding. We could, ideally, reverse a FingerTree simply by

$$reverse\ ft = foldl\ \_\triangleright\_\ Empty\ ft$$

However, declaring this in this form is impossible because of two resons:

- **_▷_** is a dependent function, incompatible with our definition of **foldl** [12]

  It is possible to implement an analogous dependent version of foldl. Doing so will however yield a specific solution to the reverse function of the FingerTree. Transforming the FingerTree first to a list and then folding it is another strategy for achieving a similar result. Unfortunately, this will incur an unnecessary computational cost.

- The index of the output FingerTree depends on the argument as well, yielding in this case

$$\text{measure-ft (reverse ft)} = \text{foldl foldfun } \epsilon \text{ (List.reverse (toList-ft ft))}$$

In this example, the cost of dependent types becomes very clear, and can be avoided by providing a non-dependent interface.

[12]https://nicolaspouillard.fr/publis/explore-iso.pdf

### 3.3.1 Reverse

The non-dependent interface is constructed through a dependent pair. The function **pack** makes the transformation, and the **cons-pair** is simply the extension of **cons** to the pair.

```
reverse−ft : ∀ {a} {A : Set a} {V : Set a}
      ⦃ mo : Monoid V ⦄
      ⦃ m : Measured A V ⦄
      → (Σ V (λ v → FingerTree A V {v}))
      → (Σ V (λ v → FingerTree A V {v}))
reverse−ft {a} {A} {V} pair =
   foldl−pair cons−pair (pack−ft {A = A} {V = V} Empty) pair
```

Note that this implementation no longer constrains the result to the argument. The only verified property of this is that the result will itself be a correctly constructed FingerTree.

## 3.4 Random Access Sequences

By proper instantiation of the measurement function and monoid, we can specialize the Finger Tree to various data structures. The first suggested on is the Random Access Sequence.

The measurement function will assign the value of 1 to any element, and the monoid is simply that of the natural numbers with addition. For convenience, I will wrap both the size and the entries is special types, to aid with the use of instance arguments.

### 3.4.1 SizeW and Entry

The *SizeW* is simply a wrapper around *Nat*[13]

```
data SizeW {a} : Set a  where
   size : ∀ (n : ℕ) → SizeW {a}
ε : ∀ {a : Level} → SizeW {a}
ε = size 0

_•_ :  ∀ {a} → SizeW {a} → SizeW {a} → SizeW {a}
size n • size m = size (n + m)
```

The properties of the *Nat* carries directly to*SizeW*, so that we can populate a *Monoid SizeW*

[13]for compatibility with the rest of the implementation, I had to assign an arbitrary universe level *a*

```
instance size−monoid : ∀ {a} → Monoid (SizeW {a})
size−monoid = monoid ε _•_ ε• •ε •−assoc _<ᵗ_
```

*Entry A* is a wrapper around elements of type A, given by the constructor *entry*.

```
measure : ∀ {a}{A : Set a} → (x : Entry A) → SizeW {a}
measure x = SizeW.size 1

instance entry−measure : ∀{a}{A : Set a} → Measured (Entry A) SizeW
entry−measure = measured measure
```

### 3.4.2 Seq Instance

The Sequence instance is simply an alias for a Finger Tree, instantiated with an arbitrarily typed Entry and the SizeW monoid.

```
Seq : ∀ {a}(A : Set a) SizeW → Set a
Seq {a} A s = FingerTree (Entry A) (SizeW {a}) {s}
```

#### Retrieving the nth element

The naive implementation would simply call the *viewL* n times, yielding an amortized linear cost.

However, we can find an $\mathcal{O}(log(n))$ implementation, by using *split*. The predicate checks whether the current value is smaller than n, and we will start iterating from 0.

```
_!_ : ∀ {a}{A : Set a}{s : SizeW} → Seq A s → ℕ → Maybe A
seq ! n with split−Tree (λ x → (size n) <ˢ x) (size 0) seq
seq ! n | just (split−d _ x _) = just (getEntry x)
seq ! n | nothing = nothing
```

The call to split allows us to view the nth element, as well as the sequences that remain to the left and to the right.

#### Setting an element

This also suggests a possible implementation for setting an element, which is rather inefficient, but provided for completeness, using Finger Tree concatenation.

```
set : ∀ {a}{A : Set a}{s : SizeW} → Seq A s → ℕ → A → Seq A s
set seq n y with split−Tree (λ x → size n SizeW.<ˢ x) SizeW.ε seq
set seq n y | just (split−d left x right)
  rewrite •−assoc (measure−tree left)
```

```
          (size 1)
          (measure−tree right)
        = concat ((entry y) ▷ left) right
    set seq n y | nothing = seq
```

## 3.5 Writing recursive definitions

As mentioned in section whatever, the use of *with* can sometimes confuse the termination checker. This, combined with the nature of the *viewL*, makes it hard to make us of the abstraction brought by deconstructing FingerTrees as we would deconstruct normal Linked Lists.

In fact, Agda's termination checker cannot prove that, for any finger tree *ft*, viewL (x *cons* ft) is structurally bigger than ft. The reason is that the *cons* operator has to be performed before the call to viewL.

It has been suggested in Matthiew Souzeau's paper [**?**] that to overcome this problem, one must find a suitable indexing that reflects the number of elements in the Finger Tree. This is exactly what the Random Access Sequence achieves.

### 3.5.1 Well founded induction

In order to write a recursive definition against these constraints, we need to convert an arbitrary well founded relation to a structural less-than relation.

In this case, concerning *SizeW*, the well founded relation comes free from the natural ordering of Nat.

According to the definition used in this context [14], we say that a binary relation is well-founded if all elements in the carrier set are Accessible. For an element to be Accessible, we require inductively that all the smaller elements are themselves Accessible.

By implementing the accessibility relation in Agda, we transform any given order into the structural order which Agda recognizes.

### 3.5.2 Packing the Sequence

The ordering of the sizes naturally extends to an ordering of the Sequences, so that removing an element from a Sequence necessarily yields a smaller Sequence.

In this case, we can use Larry Paulson's results [15] to construct a Well-Founded relation on Sequences it is sufficient to have a Well-Founded relation on the Sizes.

[14]http://adam.chlipala.net/cpdt/html/GeneralRec.html
[15]also implemented in the standard library

```
-- comparing Seq-pairs is just comparing the size component
_⋖_ : ∀ {a} {A : Set a} → Seq−pair A → Seq−pair A → Set a
_⋖_ = _<<_ on to−size

open Inverse−image
   {A = Seq−pair A}
   {_<_ = _<<_}
   to−size
   renaming (well-founded to «-<-wf)


-- the comparison of the Seq-pair is well-founded
   <−WF = <<−<−wf <<−WF
   open WF.All (<−WF) renaming (wfRec to <rec)
```

Having obtained a proof of the well-foundeness, we can proceed by creating a recursor object and writing a recursive definition.

### 3.5.3 Reversing - Until a better example is found

We have already implemented the reverse for the FingerTree through the fold.

```
rev : Seq−pair A → Seq−pair A
rev π = <rec a _ go π
   module Rev where
   go : ∀ s → (∀ p → p ⋖ s → Seq−pair A) → Seq−pair A
   go ⟨ fst , snd ⟩ rec with viewL snd
   go ⟨ .(size 0) , snd ⟩ rec | NilL = pack Empty
   go ⟨ _ , snd ⟩ rec | ConsL x xs =
      rec (pack (xs)) (one−step−lemma (measure−tree xs)) ▷′ x
```

It is arguable whether this is actually a solution to the abstraction problem. We have indeed been able to write a definition of the reverse function using an abstract view. However, the solution itself is probably less readable than a solution that reverses directly on the Finger Tree.

Furthermore, in this form, we no longer work with dependently typed functions, since the indexes have been covered up by the dependent pair. No correctness is enforced by the typing system. Trying to prove correctness of this function brings back the problems caused by *with*. In particular, we have an induction proving mechanism, but we are stopped by the inability to prove an inductive step on viewL.

**For simple properties,** it is sufficient to output a new dependent type that enforces those property. As a concrete examples, we want to show that the size of the reversed sequence is equal to that of the original sequence.

Let **Same-Size-Seq** be a family indexed by the **Seq-pair**, with a constructor enforcing this property. [16]

```
data Same−Size−Seq : (s : SizeW {a}) → Set a where
  ssseq : ∀ {s} {z} → (Seq A s) → (Seq A z) → (s ≡ z) → Same−Size−Seq s
```

Reimplementing the **rev** method with this output type is straight-forward and achieves the correctness goal. It is presented in the Appendix.

**For arbitrary properties,** a powerful approach would be to make the induction process obvious

```
property : Seq−pair A → Set a
```

The problem that remains is proving the inductive step. This seemed impossible to me, because of the limitations of the *view* operation.

```
inductive−step : ∀ {s : SizeW}
  → (seq : Seq A ((size 1) • s))
  → (x : Entry A)
  → (xs : Seq A s)
  → (viewL seq ≡ ConsL x xs)
  → (property (pack xs))
  → (property (pack seq))
```

## 3.5.4 Notes

- add concatenation in the finger tree implementation, with all the proofs
- start thinking about evaluation

[16]At the time of implementation, I had not realized this trick yet. It would be an interesting extension to change the mechanism employed by the *rewrite* to use something like this, and allow using Equality-Reasoning for type-level equality

# Chapter 4

# Evaluation

In the previous sections, I have experimented with various ways to implement and prove correct a data structure using Agda.

The running example was the implementation of the FingerTree, chosen because of the combination of many concepts in a single data structure.

- The data structure is used extensively in functional programming, being part of the Haskell standard library. [1]
- It is based on the existence of a monad and a measurement function, whose assumptions provide interesting starting points for proofs.
- It is a nested type, therefore not widely supported in functional languages. [2]
- It allowed a non-trivial index in the dependent implementation, namely the measure.
- It is a good example to outline the limitations of languages like agda with respect to the termination checker.

This project was originally intended as an supportive argument towards the use of dependent types, and understanding what causes the unpopularity in industry.

I will therefore compare two instances of the Random Access Sequence, one using the implementation of the FingerTree as presented in the Implementation section, and the other one using a version with a trivial index (Size), with no correctness enforced through dependent types.

[1] https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Sequence.html

[2] consider, for example, the implementation in Isabelle, which is forced to redefine the data structure accordingly

## 4.1 Run-time

The first comparison that could be made is too see whether the dependent typing incurs an additional cost on the programming. This is not per se an evaluation of the code, but more of the extraction process and the compilation.

The current version of Agda (2.5.1.1) makes available three compilers, all tagged as 'experimental': *ghc*, *js* and *uhc*. Unfortunately, the two last ones did not work on my machine, causing it to crash or filling up all the available RAM (16GB).

I have only used my machine for these experiments, with specifications detailed in table. To reduce the error in the results, I have limited the related processes' usage to a single core.

Furthermore, I have repeated each experiment between 10 and 1000 times (limiting each measurement to approx 10 minutes) and only reported the smallest value, which could be read as a lower bound of the runtime.

| Component | Setting |
|---|---|
| Operating System | Ubuntu 16.04.1 LTS 64-bit |
| Kernel | Linux v4.4.0-72-generic |
| CPU | Intel® Core™ i7-4702HQ, 2.20 GHz |
| RAM | 16 GB |
| Agda version | 5.2.1.1 |

Table 4.1: Machine specifications

### 4.1.1 *cons*

For the first experiment, I am evaluating the run-time of consing $2^n$ elements to a sequence. $n$ is limited by the available memory of the system.

For this purpose, I have implemented a simple recursive consing function, and forced[3] its evaluation by a *tab* operation[4]. The output is done via the Haskell-like Monad way.

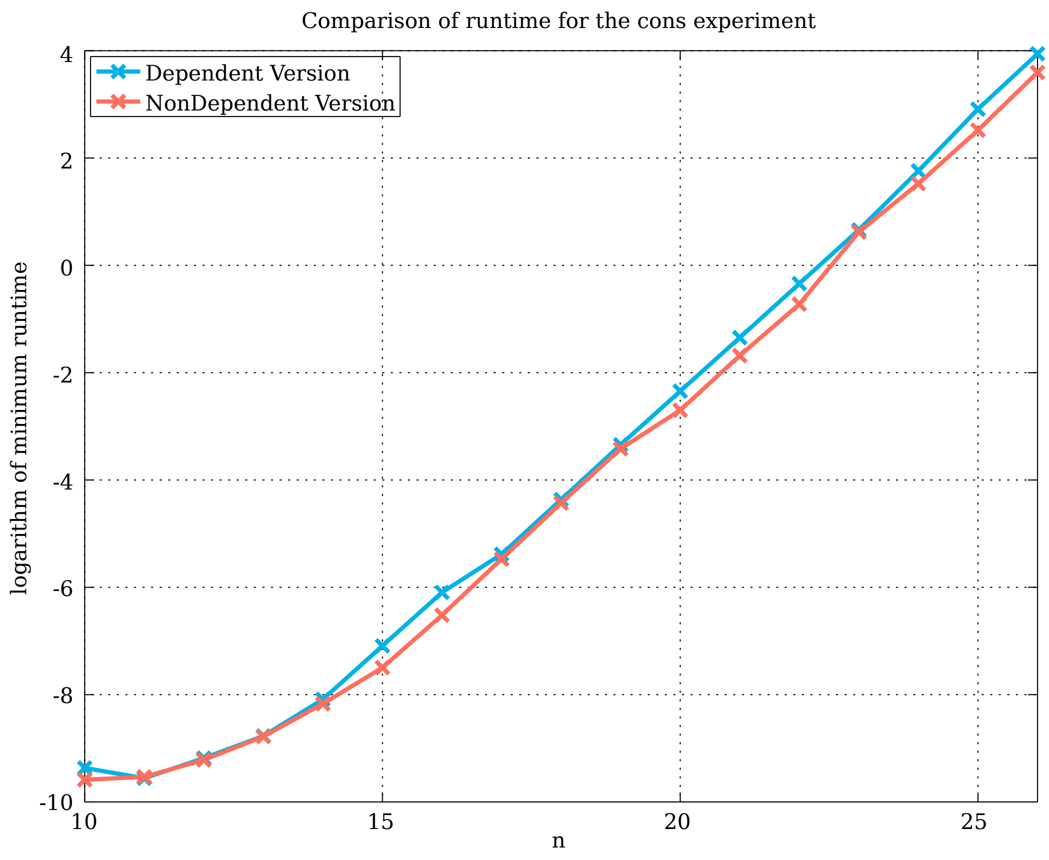
```
big−seq : (n : ℕ) → Seq ℕ (size n)
big−seq zero = Empty
big−seq (suc n) = (entry n) ◁ (big−seq n)


main = (putStrLn (toCostring "Hello") >>=
   (λ x → return (big−seq 1024) >>=
   (λ x → putStrLn (toCostring (show−maybe(x ! 1)))) >>=
   (λ x → return 1)))
```

[3]Agda has lazy evaluation

[4]Getting the first element is guaranteed to have a negligeable run-time, since it resides in the leftmost digit, requiring only three function calls

Bellow is a plot of the runtimes for the dependent and the non-dependent versions, on a log-log scale.

Figure 4.1: Consing experiment

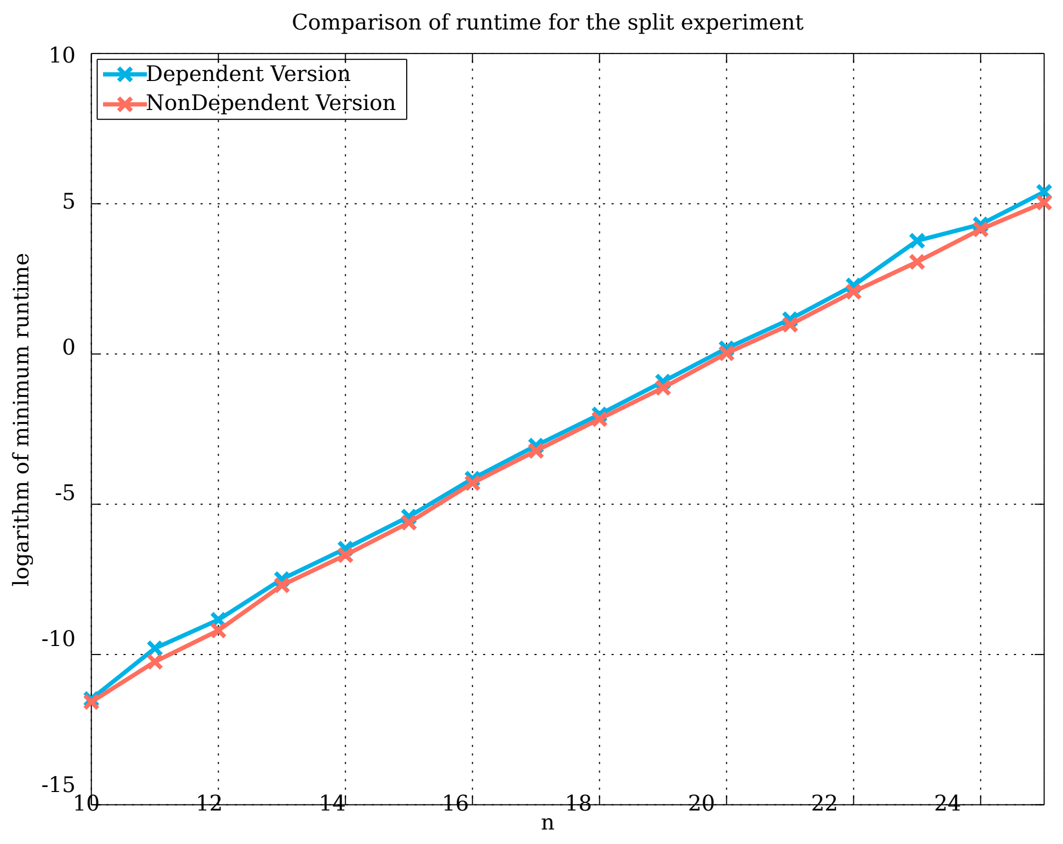


### 4.1.2 split

This experiment is an extension of the previous one. The difference is that we require tab operation that depends on the length of the sequence. I have chosen for this purpose to identify the middle element.

Also, since we require creating a big-sequence to start with, I have subtracted the results from the previous experiment, so the values are the minimum time to extract the middle element from an increasingly long sequences.

Figure 4.2: Splitting Experiment



I is clear from both these experiments that the dependent versions incurs a higher computational cost. Moreover, the ends show that the divergence between the two depends on the input size. However, this is not a limitation of dependent types, but of the compiler and extracting tools. All the type annotations could be removed without damaging correctness.

### 4.1.3 reversing and problems with the compiler

I have tried to repeat the same type of experiment, comparing the run-time of the **reverse-ft** method, implemented using **foldl**, and the *rev* method, which uses the **ViewL**.

Unfortunately, the compilation of the latter method was not successful. Whereas the *normalization* tool always returns the correct result, running a compiled version has caused a *Segmentation Fault* error. This is quite damaging to the application, since a compiler that doesn't preserve the semantics of the code doesn't necessarily preserve correctness.

## 4.2 Heuristics for Effort

There is no doubt that creating a dependent and verified version of a particular algorithm or data structure incurs an additional cost. In this section, I will try to explore the effort ratio between the implementing a simplistic solution versus a formally verified one.

It is worth emphasizing that, although computing or estimating efforts is in general an ill-defined task, some heuristics could help compare the different versions of the same data structure and see if the figures carry on across data structures.

I have suggested using two heuristics, one computing the explosion in SLOC[5], and the other one quantifying the axiom and lemma content in the definitions.

### 4.2.1 Lines of code

I will present the ratio between the lengths of different versions, commenting on what they achieve. Although arguably, counting lines of code is not always representative, in this case it seems suitable since:

- All the presented code has been written by a single programmer[6]
- I have been consistent with jumping to new lines.

I will make a clear distinction between *internal* verification and *external* verification [**?**]. By the former I mean properties that are made clear through the type of the definitions themselves (e.g. the use of measure as an index), while the latter refers to proofs which are carried outside of the definition (e.g. foldl-correct).

For this metric, I have decided to discard all the *external*ly verified properties, since there is no bound on their number. On the other hand, *internal* verification directly affects the implementation's difficulty.

Furthermore, I will separate all lines into three categories: type annotations, proofs (related to type checking) and the actual implementation.

**Selection Sort** The selection sort procedure, presented in section whatever is an example of a fully formally verified procedure, as all that can be expected of a sorting function is encoded in the type system. This differs from the FingerTree implementation, which only enforces correctness of the measurement.

[5]Source Lines of Code

[6]it is arguable whether the metrics will therefore show something about using dependent types or just about myself as a programmer

Table 4.2: SLOC for Sorting

| Content | Verified | Not Verified | Ratio |
|---|---|---|---|
| Implementation | 18 | 14 | 1 |
| Type annotations | 24 | 3 | 8 |
| Proofs | 124 | 0 | n/a |
| Other | 53 | 50 | 1 |
| Total | 219 | 67 | 3 |

The unverified example in this case does not run in Agda. It would require proofs of termination. I have only presented this as a reference point for the Finger Tree example.

Table 4.3: SLOC for Finger Tree

| Content | Measured Version | Size Version | Ratio |
|---|---|---|---|
| Implementation | 185 | 162 | 1 |
| Type annotations | 144 | 68 | 2 |
| Data declarations | 51 | 32 | 1 |
| Proofs | 350 | 30 | 12 |
| Other | 30 | 22 | 1 |
| Total | 760 | 314 | 2.5 |

**Finger Tree** It is interesting to see the same ratios remaining consistent. As a sanity check, the code related to implementation has stayed constant. The proofs carry around half of the code for a verified version. This depends of course on the number of invariants we are trying to maintain. A further increase in the lines that express type signature lead to an overall 2.5 factor of explosion in the code.

As a side note, the effort as illustrated here is not directly related to difficulty, since the more expressive types in verified versions make the goals of implementation clearer.

### 4.2.2 Lemma Usage

The previous examples did not include any metric about the externally verified properties. Performing this evaluation experiment showed that all the lemmas are constructed from calls to our assumptions (base lemmas).

- about the equivalence relation: *refl*, *sym*, *cong* and *trans*
- about the monoid relation: *associativity*, $\epsilon$ is *neutral element*
- about List: *associativity* of ++ and *[]* is *neutral element*

I have performed an analysis of what lemmas are being used by each declaration, followed by a flattening of the result to the base lemmas. The final numbers can be seen as a metric of effort, but also such an analysis could potentially be helpful to refactoring. A summary of results if given below:

Table 4.4: Lemma Usage for Internal Verification

| Lemma | cons | snoc | viewL | viewR | split |
|---|---|---|---|---|---|
| refl | 1 | 0 | 3 | 3 | 48 |
| sym | 2 | 16 | 5 | 8 | 23 |
| cong | 3 | 11 | 1 | 5 | 15 |
| trans | 3 | 23 | 6 | 10 | 46 |
| ε-left | 1 | 2 | 4 | 2 | 22 |
| ε-right | 1 | 0 | 2 | 4 | 13 |
| •-assoc | 6 | 18 | 3 | 8 | 21 |

## 4.2.3 Discussion