Razvan Kusztos

Verified functional data structures in Agda

Part II Project in Computer Science

Girton College

May 12, 2017

Proforma

Name:	**Razvan Kusztos**
College:	**Girton College**
Project Title:	**Verified functional data structures and algorithms in Agda**
Examination:	**Part II Project**
Word Count:	**0**[1] **(well less than the 12000 limit)**
Project Originator:	Dr Timothy Griffin
Supervisor:	Dr Timothy Griffin

Aims

In this project, I aim to use Agda for the implementation and formal verification of the FingerTrees, an important data structure in the functional world which presents a number of peculiarities. This work has been performed in more widely used theorem provers, like Coq and Isabelle. It is interesting to see to what extent such results can be replicated using a simple tool, and whether similar problems emerge. Moreover, I want to understand what is that makes dependent typing far from being an industrial standard, by exploring formal verification.

Summary

In the following, I begin with a concise introduction to Agda and results in the fields of functional data structures and formal verification using dependent types. I am presenting a dependently typed implementation of the FingerTree, showing how difficulties can arise because of the nature of the data structure and limitations to the environment. I conclude by comparing dependently typed and non-dependently typed implementations, exploring the weight formal verification adds to programming and discussing relations to previous work.

Special Difficulties

None.

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

Declaration

I, Razvan Kusztos of Girton College, being a candidate for Part II of the Computer Science Tripos, 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

Chapter 1

Introduction

1.1 Verified Programming

Verification of program correctness is paramount to any successful application. Most commonly, hard-coded tests are used in software testing. In this project I will focus on formal verification, that is, checking whether a program is correct under some formal mathematical modelling, rather than through its behaviour.

Formal verification has had successful application in areas such as cryptography (Cryptol[1]), hardware specification (The verification system in Verilog) or compiler construction (CompCert[11]). A more general approach implies verifying arbitrary software programs. Automated theorem provers such as Coq, Isabelle or Agda achieve this goal. Although the principles they use have been around for decades, their industrial application has remained niche. For instance, Isabelle's open archive of proofs contains mainly proofs concerned with mathematical objects, with very few examples of algorithms or data structures. [2]

1.2 Functional Data Structures

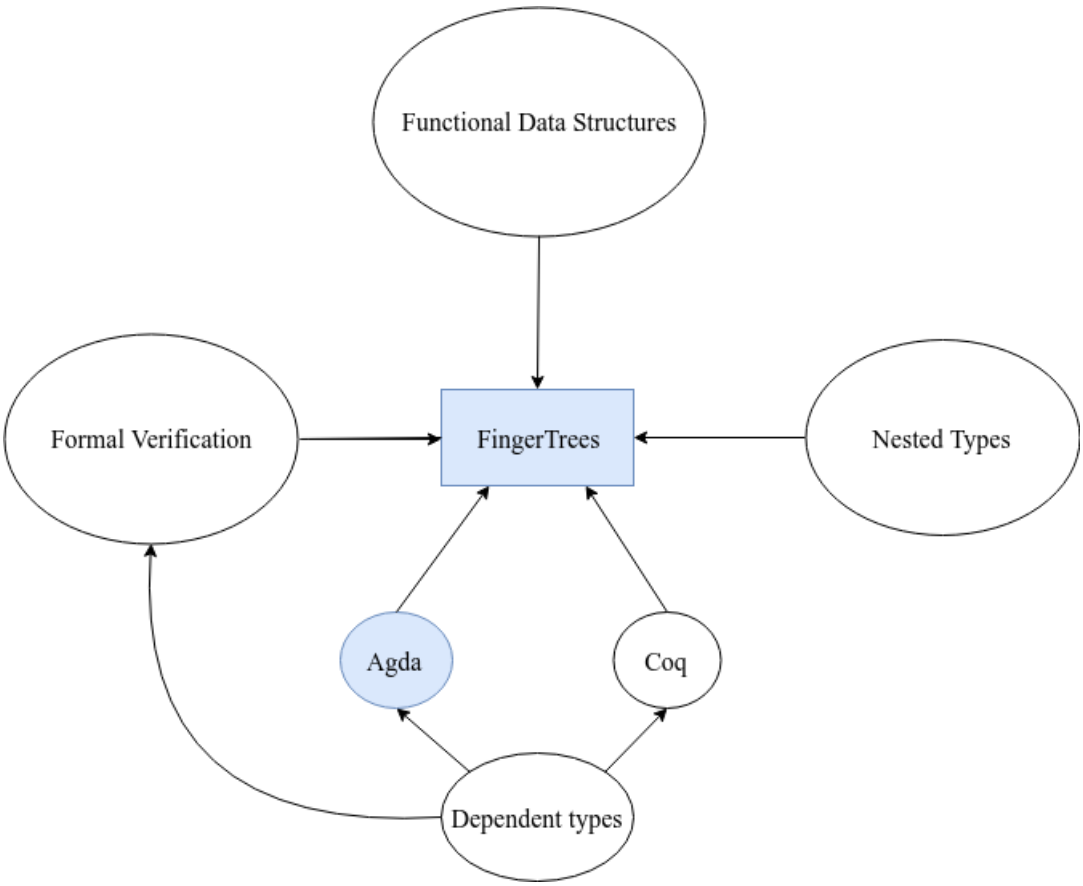
In the context of programming, we can distinguish two main paradigms: a stateful approach, used in most imperative programming languages, and a stateless approach, seen in pure functional programming.

The formal treatment of these two approaches differs. In imperative languages, functions can alter the state of the program and return different results at different times, even when called with the same arguments. In this context, verification is done in a different environment, using systems such as Hoare Logic.

Functional programming languages enforce the idea of statelessness, bringing functions closer to mathematical functions. They always return the same result for a given argument, regardless of when and where they are called. This concept is becoming popular in industry again, in light of the advantages it brings in concurrent programming: lack of shared state and laxer execution order.

[1] http://cryptol.net/documentation.html

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



Functional data structures have long been thought to be inefficient and belonging solely to academia. The work of Okasaki [16], namely his book, *Purely Functional Data Structures*, has gone a great way in mitigating the imbalance between the vast collection of efficient imperative structures and the esoteric functional data structures. His discussion of implicit recursive slowdown and numerical representations of types has inspired the design of many data structures, which are becoming part of standard libraries in functional languages.

A property of functional data structures is that they are persistent. Thus, 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. Persistence is not usually enforced in imperative implementation (consider updating an element in a C array). Solutions for achieving this goal in an imperative environment are tedious and incur an extra run-time cost.

However, amortisation techniques like the ones described by Okasaki introduce efficient ways of achieving this goal in a functional setting. Furthermore, the lack of side effects makes reasoning about functions a tractable problem, and opens up many possibilities for verifying programs.

Languages like Agda[15] have been designed firstly as general programming languages, and secondly as theorem provers. Their development is in an early stage, and are mainly employed in research.

Alongside Coq and Idris, Agda is based on the theory of dependent types. In these environments, types and values are part of the same grammar, forming *terms*. This enforces expressivity and allows reasoning about code and proofs in the same environment (under the Curry Howard isomorphism).

In this dissertation, I aim to show some approaches to verifying data structures and algorithms using the expressive power of Agda. I have chosen, as a running example, the

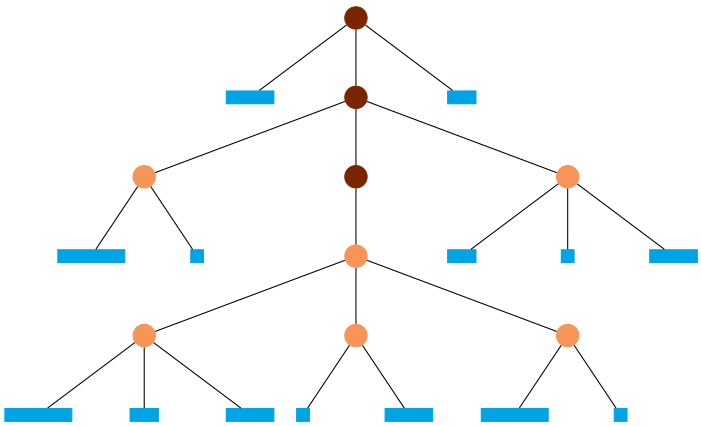


Figure 1.1: Example finger tree

implementation of the FingerTree [9].

FingerTrees are a modification of the 2-3 Trees[3] which allows efficient insertion at the beginning or the end. Its abstract data type is that of a double-ended queue. Furthermore, FingerTrees can be associated with a measurement function and an operator (described in section 2.7.1) that allows specialisation to other abstract data types, such as Random Access Sequences, Priority Queues or Interval Trees.

The work of implementing the FingerTree in a dependently typed environment has been carried out by Matthieu Souzeau [18]. **A goal of this dissertation** is to see to what extent the same result can be achieved in Agda, experimenting with various patterns for proving correctness.

1.3 Summary

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

In the third chapter, I present a dependently typed implementation of FingerTree which ensures correctness, as well as proofs related to this data structure. I show how specialising the FingerTree to a Random Access Sequence maintains the correctness properties. This can be seen as a software engineering approach using dependent types: building new blocks on top of verified blocks. Finally, I outline some difficulties arising in Agda and discuss ways of overcoming them.

In the evaluation chapter, I analyse the run-time difference and compare the effort of implementing a dependently typed data structure versus a non-dependently typed one. I discuss how the current implementation achieves the goals set in the Coq implementation.

[3] That is, trees whose nodes can have either two or three children

Chapter 2

Preparation

2.1 Agda

Agda[15] is a dependently typed programming language based on the predicative Martin Lof type theory. It was introduced in Ulf Norell's PhD thesis [14], 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 FingerTrees, for reasons that I will come back to in section 2.7.

In the following, I am describing programming with dependent types, as well as how formal verification is performed in Agda. This is exemplified by building increasingly advanced data structures.

2.2 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 distinction fades away. Types and values are placed under the same grammar, introducing general terms. This expressive power can aid the user and reduce many of the run-time checks needed to ensure proper execution of the program.

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

2.3 Curry-Howard Isomorphism

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 incorporate both the code and the proofs in the same environment. 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 $\forall$ (for all) corresponds to a dependent product, enabled by dependent types.

Type system	Logic
Simply Typed LC	Gentzen Natural Deduction (Gentzen)
System F	Second Order Propositional Calculus
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

Although a comparison between CiC (Calculus of Inductive Constructions) and ITT (Intuitionistic Type Theory) would be interesting, The literature on this topic is not readily available. The development of Coq was influenced by Martin Lof's theory through the presence of inductive types[5]. 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). A side by side comparison of Agda and Coq is present on the Agda website. [3]

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

Logic Formula	ITT Notation	Agda Notation
$\bot$	$\emptyset$	$\bot$
$A \vee B$	$A + B$	$A \uplus B$
$A \wedge B$	$A \times B$	$A \times B$
$A \supset B$	$A \to B$	$A \to B$
$\exists x : A.B$	$x : A.B$	$\sum A\ B$
$\forall x : A.B$	$x : A.B$	$(x : A) \to B$

Table 2.2: Curry-Howard Relation in Agda

Proving a proposition in this logic is equivalent to constructing a term that has the corresponding type.

2.3.1 Proofs in Agda

An important family of types in Agda is the propositional equality. It corresponds to the proposition that two elements of the same type can only be equal if they are in fact the same

[3] http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.AgdaVsCoq

element. This is the basis of all the proofs in this dissertation. Constructing a term of type $a \equiv b$ represents a statement that a and b are equivalent.

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

Agda Notation. In this example, we declare a data structure, ≡, in mixfix notation. That is, wherever there is an underscore (_), we place an argument. This syntax improves the readability of code (rather than the usual, prefix notation ≡(x, y)).

The declarations before the colon (:) are parameters. The {A : Set a} corresponds to a polymorphic type, found in most functional languages (an example of this is the type of lists in Haskell: [a] or ML: 'a list). However, (x : A) is an example of a dependent type argument, specific to Agda (or Coq, Idris).

The declarations surrounded by curly brackets ({...}) are implicit arguments, which are filled in automatically by the type-checker. The others need to be explicitly specified.
The type signature after the colon (:) represents the type of the data structure; in this case, we are concerned with a dependent type indexed on elements of type A. Construction declarations follow on the subsequent lines, after the keyword where, constructors are declared.

Since this data-type has a single constructor, refl, the only value accepted as argument by the data-type has to be equal to the parameter x. This explains why constructing a term of this type corresponds to a proof of equivalence.[4]

Associativity of Natural Numbers. Consider proving properties of the natural numbers, such as associativity. First, we need to declare the natural numbers in Agda. For this we will use the so called *Peano* natural numbers, an inductive data type:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Consider the definition of associativity. In mathematical notation, this would be written as:

$$\forall x, y, z \in \mathbb{N}. a + (b + c) = (a + b) + c$$

Using Table 2.2, we can translate that into the following type signature:

```
+assoc : (x y z : ℕ) → (x + (y + z)) ≡ ((x + y) + z)
```

[4] I have not mentioned *Set*. This is the type of small types in Agda. Because of the need for *Set* to have a type as well, a family of types is built, so that the type with the higher argument contains the type with the smaller argument. This explains the *a* argument in the definition of ≡.

We declare a function of this type by making a case split on the x variable, considering the possible constructors for the type of natural numbers. In the first line, the result becomes obvious to the type-checker, since: $(\ zero + (y + z)) = y + z = ((zero + y) + z)$ is solved by Agda's normalisation tool. In the second case, the type-checker requires more evidence. I use the rewrite operation, which has the effect of transforming

```
A rewrite B ≡ C
```

by replacing all occurrences of B in A with C (formally A[C/B]).
The full code:

```
+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
```

2.3.2 Induction

In the previous example, we can see that induction is a key means of proof. In the first case, I perform structural induction on all possible constructors of x as a natural number. In the second case, I also perform a natural mathematical induction step. Assuming that +assoc *x y z* holds for some x, y and z, we want to show that +assoc *(x + 1) y z* holds.

2.3.3 Further Example: properties of list reverse

Consider the following implementation of reverse, using a helper function, rev′ as reverse with accumulator:

```
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[5]:

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

[5]The full implementation is in the Appendix.

```
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 facilitates the writing of more readable proofs. This is the format in which most proofs are written throughout the dissertation. I have indented the code such that, in between begin ... ∎, the extra-indented lines correspond to successive transformations of the left-hand side of our formula, using lemmas provided inside the triangular brackets (≡⟨ ... ⟩):

```
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
  ∎
```

It is worth emphasising the dual use of the typing system, both for proving correctness and providing abstraction. The type declaration can be sufficient for understanding the function of the code.

2.4 Totality

Agda and Coq are both examples of total function programming languages. This further totality constraint imposes all the defined functions to be total.

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

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

This function doesn't type check in many languages; however, some languages could allow run-time errors to be thrown if a nonexistent case is reached. In Agda, to mitigate this constraint, it is straightforward to use the Maybe monad.

In a computational sense however, functions must be strongly terminating on all the inputs. This has to do with logical consistency. We trade off the Turing completeness in order to ensure 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 rely on heuristics for determining whether recursive calls do eventually terminate.

Agda's solution to this problem is to ensure that in every recursive call, arguments become structurally smaller [3]. This order relation is recursively defined by

$$\forall i : Nat.\forall w : Set_i.w < C(..., w, ...)$$

where *C* is an inductive data type constructor.

2.4.1 Sized types

Some operations hide this structural less-than relation (examples are provided in Section 3.1.8). For this reason, the concept of *Sized Types* has been added to the language. A comparison between the two approaches has been studied by D. Thibodeau[20]. Implementing sized types requires indexing all inductive types by a *Size* (i.e. making them depend on values of that particular type: for example, Vec is indexed by Nat) while ensuring that the *Size* decreases in recursive calls. [1]

Employing *Sized Types* in the context of *FingerTrees* turns out to be a difficult endeavour. (More generally, this is a problem occurring with some nested types which I am detailing in Appendix 6.3). It is worth emphasising that the incompleteness [5] of the termination checker makes programming challenging in many cases. A significant proportion of the proofs in Agda code are directly related to termination.

2.5 Correct Data Structures in Agda

A substantial effort in certified programming goes to preserving invariants, i.e. facts that the programmer needs to ensure about data structures in order for them to behave as expected. We need to confirm that during the execution of the program, the following statements hold:

- Constructors can only produce correct objects (i.e. that respects some arbitrary assumption).
- Any function that takes as input a correct object outputs a correct object.

If the type of the data structure ensures the invariants we want, both these statement hold due to Curry-Howard's isomorphism.

Sorting Lists. Consider, for example, implementing a sorting procedure for lists. The input is a list and the output should be a sorted permutation of the input. I provide a definition of sorted lists with elements from a given set (expressed as a dependent type):

```
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 $xs \equiv ys$, then I can insert x somewhere in xs to obtain ys. The three constructors correspond to the these principles:

- The empty list is sorted.
- A singleton list is sorted.
- Given a sorted list, we obtain another sorted list by attaching an element smaller than all the elements in the list.

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. I have implemented, as an example, the selection sort which is present in the Appendix 6.5. Although other sorting procedures could be implemented, the definition of the SortedList is perfect for the approach taken. Implementing, for example, merge sort would be more difficult using this definition of sorting, than using some equivalent definition based on e.g. splitting lists.

This example shows the expressiveness that dependent typing makes available, as well as its capacity for abstraction. In implementing the FingerTrees, I aim for a similar verification method.

2.5.1 Nested Types

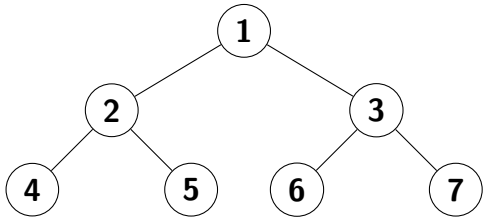
Through their type, FingerTrees also maintain structural invariants, without having to rely on dependent types. This is an instance where nested types are a useful concept.

Nested types[4], also called irregular types or polymorphic recursions, can aid in enforcing structure in data types such as full binary trees, cyclic structures [**?**] or square matrices [17]. 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 one to declare functions on nested types, in contrast to SML and older versions of Haskell.

[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.

Figure 2.1: Full Binary Tree



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[4]:

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


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

The same principles apply in the case of FingerTrees, which is based on a full 2-3 Tree.

I believe that the presence of these nested types makes termination checking an even harder task in many cases. The problems that occur in the implementation of the FingerTree are not limited to this context. I have provided the implementation of yet another data structure that suffers from the same limitations in Appendix 6.3.

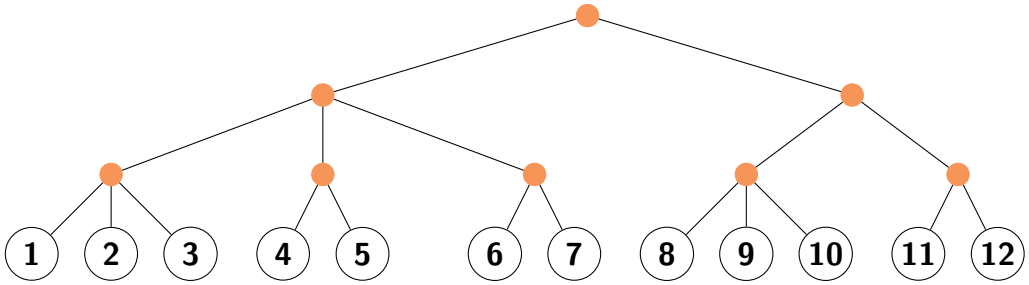
2.6 2-3 Trees

2-3 Tree are at the basis of FingerTrees. They are rooted trees where each node has either two or three children and all leaves occur on the same level. Using nested types,as presented in the previous section, we can encode this constraint into the type of the data structure. [4]

Instead of using the pair (_ × _) as above, we declare a new data type that can contain two as well as three elements.

```
data Node (A : Set) : Set where
  Node2 : A → A → Node A
```

Figure 2.2: Example 2-3 Tree




```
Node3 : A → A → A → Node A
```

To declare the tree, we use a nested type. The constructors are called *Zero* and *Succ* to signal the origin of the idea in Okasaki's numerical representations (see Appendix 6.2)

```
data Tree (A : Set) : Set where
  Zero : A → Tree A
  Succ : Tree (Node A) → Tree A
```

As an example, Figure 2.2 could be written in Agda as follows (indented in an attempt to clarity):

```
test−tree : Tree ℕ
test−tree = Succ(Succ(Succ(Zero
  (Node2
    (Node3
      (Node3 1 2 3)
      (Node2 4 5)
      (Node2 6 7))
    (Node2
      (Node3 8 9 10)
      (Node2 11 12))))))
```

The number of Succ nestings indicates the tree height. Further discussions about similar data structures can be found in [7]. Another example is present in Appendix 6.3, with an emphasis on the practical difficulties of implementation in Agda.

2.7 FingerTrees

FingerTrees are a data structure introduced by Ralph Hinze and Robert Patinsson[9], based on Okasaki's work in amortisation.
Initially designed as double-ended queues with constant amortised time append, their structure, together with the cached measurements, allow specialisation 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 leaves. For efficient appending, the tree is surrounded by buffers (digits) at each level. Furthermore, the data structure is accompanied by a measurement function and a binary operator, such that the reduced measures of all nodes in a sub-tree are cached in its root. These are necessary for efficient splitting (see Table 2.3).

2.7.1 Measurements and the Monoid

The versatility of the FingerTree as a data structure comes from its association with a set of measurements (V). The *measure* of the tree $\|ft\|$ is a map between a FingerTree and an element of the set V.

The construction of this map requires two building blocks. First, we need a function that maps the elements of the FingerTree to elements of the set V, referred to as the **measurement function**. Secondly, we need the set V to be the carrier of a **Monoid**.

That is, we pick an **operator** (in infix notation):

$$\_ \cdot \_ : (V \times V) \to V$$

and an element in V, which will be called the **neutral element**, such that the next axioms hold:

$$\forall x \in V, \epsilon \cdot x = x \qquad \text{(neutral-left)}$$
$$\forall x \in V, x \cdot \epsilon = x \qquad \text{(neutral-right)}$$
$$\forall x, y, z \in V. x \cdot (y \cdot z) = (x \cdot y) \cdot z \qquad \text{(assoc)}$$

We define the measure of the tree to be the result of recursively reducing each branch and accumulating the result using the operator ($\_ \cdot \_$). At the base case, an **Empty** tree should be mapped to the **neutral element** and the **leaves** in the tree should be mapped using the **measurement function**.

A further consideration is that of efficiency. Recomputing these values would incur a linear time cost for every operation. We can amortise this cost by keeping cached measurement tags in all nodes of the tree.

2.7.2 Invariants

Achieving efficiency requires keeping two invariants on the data structure:

- The tree is full and all the leaves occur on the last level.
- The measurement tags are correct.

Working in Agda allows keeping these invariants solely in the type signature of the FingerTree. More specifically:

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

2.7.3 Previous Work

FingerTrees have been previously implemented and proved correct. I will outline some previous results, as well as their limitations, providing more incentives for this dissertation.

- Basic Implementation in Agda.
  This version can be found on GitHub.[7]. Its mentioned intention is to closely follow the original paper. It also uses the idea of *Sizing*, although only in the type declaration (and constructors). Since the constraints do not affect functions that modify the data type, they do not really aid correctness proofs. This implementation has no proofs associated with it and didn't type check on my machine. Nonetheless, I have used it as a starting point.
- Implementation in Coq.
  This implementation is provided by Matthieu Souzeau[18] as a proof of concept for Russell, a Coq extension. I have drawn great inspiration from that paper, and I was particularly intrigued by a small caveat related to termination checking.

  My dissertation proceeds in a similar manner, implementing the FingerTree and tackling the invariants in the same way. Additionally, I am proving further properties of the operations and presenting a working solution to the termination issue.
- Implementation in Isabelle.
  Another working implementation has been done in Isabelle[13]. However, this implementation diverges from the original specification of the data structure, removing the nesting. The two invariants that I have mentioned must be maintained explicitly, due to the lack of dependent types in Isabelle.

The fact that this data structure in both Coq and Isabelle, two established theorem provers, illustrates the complexity involved as well as the interesting nature of its particularities.

[7] https://github.com/fkettelhoit/agda-fingertrees

2.7.4 Abstract Operations

Table 2.3: Summary of operations

Operation	Short Description	Properties
_◁_(cons)	Appending an element at the left of the FingerTree	$\|\|x \triangleleft ft\|\| = \|\|x\|\| \cdot \|\|ft\|\|$ $toList(x \triangleleft ft) = x :: toList ft$
snoc	Appending an element at the right of the FingerTree	$\|\|x \triangleright ft\|\| = \|\|ft\|\| \cdot \|\|x\|\|$ $toList(x \triangleright ft) = toList ft + +[x]$
viewL	Deconstructing a FingerTree into its first element and the rest of the elements reorganised as a FingerTree	$\|\|viewL ft\|\| = \|\|ft\|\|$ $toList(viewL ft) = toList ft$
viewR	Deconstructing a FingerTree into its last element and the rest of the elements reorganised as a FingerTree	$\|\|viewR ft\|\| = \|\|ft\|\|$ $toList(viewR ft) = toList ft$
foldL-ft	Similar to the same operation on Lists, equivalent to mentally replacing all the nodes with a call to a given function	$foldL - ft(i, f, ft) = foldl(i, f, toList ft)$ $foldL - ft(\epsilon, foldfun, ft) = \|\|ft\|\|$ where $foldfun(a, b) = \|\|a\|\| \cdot b$
split	Extract an arbitrary element, given by a predicate function P and reconstruct the left and right remaining element into two FingerTrees	$\|\|split(i, P, ft)\|\| = \|\|ft\|\|$

Some of the properties on the rightmost column are presented in the implementation. There are properties which could not be proven due to problems with the with operator, which I present in section 3.1.8.

The properties related to size (||_||) are proven 'internally', as part of the implementation. The others are proven 'externally', by defining terms of types that express those properties.

2.8 Starting Point

At the beginning of this project, I had no previous knowledge of:

- Agda or Haskell - although I did have experience with functional programming languages from the Foundations of Computer Science course.

- Formal verification of computer programs.
- FingerTrees and other advanced functional data structures.

Some relevant material was present in *Logic and Proof* (Larry Paulson) and *Types* (Andrew Pitts), as well as some parts from Discrete Mathematics (Marcelo Fiore) and Computation Theory (Andrew Pitts).

Chapter 3

Implementation

3.1 FingerTrees - Implementation

The previous section has provided an abstract representation of FingerTrees, as well as introduced the prerequisites for writing an implementation in Agda. In the following, I will provide implementations of the constructors and operations described in Table 2.3 and present proofs of correctness.

3.1.1 Data type declaration

The FingerTree is originally polymorphic in two types:

- **A**: the type of elements that are contained in the FingerTree.
- **V**: the type of measures.

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

- **Monoid**[1] **V**: which contains a neutral element(**ε**), a binary operator(•), 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 for 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)
```

[1] see AlgebraStructures.agda

```
      → (v ≡ ‖ x ‖ • ‖ y ‖)
      → Node A V
    Node3 : (v : V)
      → (x : A) → (y : A) → (z : A)
      → (v ≡ ‖ x ‖ • ‖ y ‖ • ‖ z ‖)
      → Node A V
```

Digits were presented in the original paper as lists, but this definition limits them to having 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 μ. The measurement's correctness is enforced in all constructors. Note the nested type and the universal quantification over possible sizes for the recursive call. Apart from the measurement indexing, the rest corresponds to the original paper [9].

```
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, when 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
```

Considering Figure, 1.1, I have colour-coded the nodes as follows:

Symbol	Constructor
	Deep
	Node
	Digit (of various lengths)
	Empty

3.1.2 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 a *Sizing* that depends on all elements in the FingerTree.

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

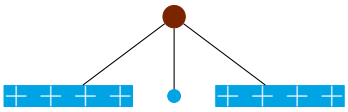


Figure 3.1: Bigger FingerTree

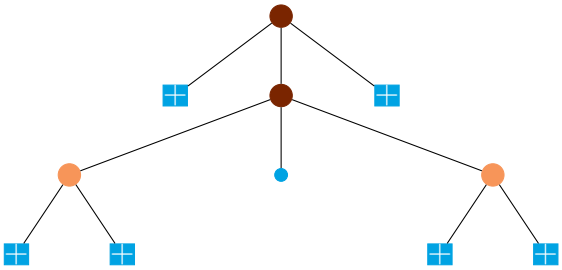


Figure 3.2: Smaller FingerTree

In Figures 3.1 and 3.3, it is ambiguous 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. I will show in section 3.1.6 how using *Sized Types* can impede the implementation of the viewL operation.

As suggested by Matthew Souzeau [18], a sizing that reflects the number of elements is ideal. We can use the already existing measurement index to achieve this goal.

Having completed the declaration of the data type, I will next present the implementation of the operations in Table 2.3, with associated proofs and discussions.

3.1.3 _◁_ (Cons) and *Snoc*

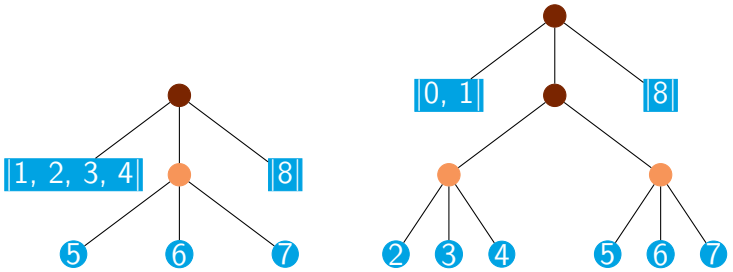
_◁_ (cons) is the operator that appends an element to the left of the FingerTree.

The implementation is straightforward if there is room in the leftmost digit. Otherwise, we have to recursively insert and reform parts of the FingerTree.

Ultimately, for correctness, we are concerned with whether the output tree is a correct FingerTree that has a correct measurement.

That is, by inserting an element x,

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



(a) Before Cons (b) After Cons

Figure 3.3: 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 FingerTree is correct with respect to the definition. The result of the function is presented after the equal sign (=), and the rewrite clause contains evidence for the type-checker that the declaration respects the type signature.

Also note that the first two cases had to be written in prefix notation, damaging readability. The reasons for this are that the bare type information was not sufficient for performing inference, and I had to manually tag the operations and make the implicit arguments explicit.

```
_◁_ {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 FingerTree operations are symmetric with respect to the middle, so the construction of the snoc operator is exactly dual. Its implementation is provided in the source code.

3.1.4 Proving correctness of the _◁_(cons) operator

Proving properties of FingerTrees requires reasoning about their elements and their order. It is therefore convenient to transform them into lists, as they encode these properties. Although this conversion is an instantiation of foldl (Section 3.1.9), this form makes reasoning easier:

```
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 :: []
toList−ft (Deep x₁ ft x₂) = (toList−dig x₁) ++
    (flatten−list (toList−ft ft)) ++
    (toList−dig x₂)
```

In the previous, toList−dig is a straightforward conversion, while flatten−list transforms a list of Nodes into a list of As. (See Appendix 6.1 for full implementation)

Assuming that the implementation of list is correct, I define the correctness of the _◁_(cons) operator as the distributivity of toList over _◁_(cons).

```
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)
```

This ensures that not only the last *cons*ed element is the first element in the FingerTree, but also that the recursive case of _◁_(cons) preserves the order of the elements.

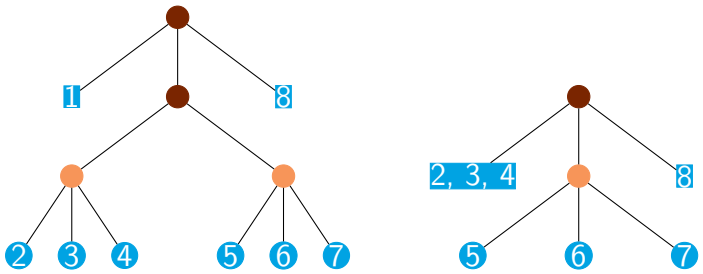
3.1.5 View from the Left/Right

As suggested in the original paper [9], the structure of the FingerTree is complicated and users may benefit from a higher level of abstraction. Furthermore, we have no mechanism of *deconstructing* a sequence.

In this case, we will *view*[22] each FingerTree as the product between an element and the remaining FingerTree.

```
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 is indexed in the same way as the FingerTree, enforcing the correctness of the measurement. We need to implement a procedure that transforms between the two representations. As in the implementation of the _◁_(cons) operator, most cases are superfluous. The nontrivial case arises when the leftmost digit contains a single entry:



(a) Before ViewL (b) After ViewL

Figure 3.4: ViewL operation (only including the tails)

Unfortunately, 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 (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 x  x ) x  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 x  x  x ) x  sf
```

The previous listing represents a pair of mutual recursive functions. deepL is a variant of the Deep constructor, which also allows the construction of a FingerTree in the case when the first Digit is empty (encoded by nothing). The proof that the result has measure "measure-maybe-digit pr • s • measure-digit sf" could only be obtained because of the dependently typed implementation of Node. A non-dependently typed version would have required postulating correctness of the Node constructors.

3.1.6 *ViewL* using *Sized Types*

An alternative definition of the FingerTree can be indexed by *Size*, as explained in Section 2.4.1. In order to simplify things in the following, we are no longer concerned with the correctness of the measurement function, discarding it completely. We only analyse termination checking.

```
data FingerTree {a : Level} (A : Set a) : {size : Size} → Set a where
  Empty  : {i : Size} → FingerTree A {↑ i}
  Single : {i : Size} → A → FingerTree A {↑ i}
  Deep   : {i : Size} → Digit A → FingerTree (Node A) {i} → Digit A →
    FingerTree A {↑ i}
```

This implementation is analogous to the one presented in Section 3.1.1, after being stripped of all the components related to the measurement.

The implementation of viewL illustrates the difficulty of sizing the FingerTrees using Agda's inbuilt sizes [1], as above.

Although the _◁_(cons) operation type-checks,[2]

```
_◁_ : ∀ {i a} {A : Set a} → A → FingerTree A {i} → FingerTree A {↑ i}
```

implementing the *view* operation is not straightforward. Declaring the analogous data structure (below) stops the implementation of the viewL method.

```
data ViewL {a}(A : Set a) : {i : Size} → Set a where
  nilL : ∀ {i} → ViewL A {↑ i}
  consL : ∀ {i} → A → FingerTree A {i} → ViewL A {↑ i}


viewL : ∀ {i : Size} {a} {A : Set a} → FingerTree A {i} → ViewL {a} A {i}
viewL Empty = nilL
viewL (Single x) = consL x {!!} -- consL x (Empty)
viewL (Deep pr m sf) = {!!} -- consL (head pr) (deepL (tail pr) m sf)
```

Two cases are left unsolved. The second one is due to the observation that with *Sized types*, we can only specify a preorder between the argument and the outputs. There is no way to relate the outputs of polymorphic functions in a meaningful way. In this case, we would not be able to decide if *cons*ing an element of type *A* yields a smaller tree than *cons*ing an element of type Node *A*, which should hold in this case. The error thrown in this case is related to the presence of a negative cycle in the sizing graph. A more detailed example of this issue is in Appendix 6.3. The first issue, however, is because of a design choice in the definition of *Sized Types* that forbids pattern matching[1].

The difficulty of satisfying the termination checker using *Sized Types* is a real bottleneck in Agda programming and should be studied further.

[2] I believe that this signals another issue, as after performing a very similar implementation in Appendix 6.3, I was expecting a failure in the trivial cases. There is more work to be done in this area.

In the next sections, I will continue the implementation of the FingerTree as presented initially, reintroducing the measurement and the indexing.

3.1.7 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 a simple property that appears difficult to prove in a dependently typed setting. The following issue also occurs in the Coq implementation [18], but appears trivial in the Isabelle implementation [13].

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 to $\forall$ *ft : FingerTree A V* $\{\epsilon\}$ $viewL(ft) \equiv NilL \iff ft \equiv Empty$ allows the definition. However, the type-checker gets stuck trying to case split *ft*. The reason is that it cannot find terms in *V* to satisfy the constrained indexing (e.g. a Deep constructor returning a FingerTree A V $\{\epsilon\}$).

Indeed, if we try to prove this statement on a simpler version of *FingerTree* that is indexed by Size [3], 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[2], using an alternate implemenation of equality – *Heterogeneous Equality*, which unifies the types at a later stage.

Using heterogeneous equality, we can write the original statement in Agda, as well as pattern-match on the FingerTree. However, some problems related to the with construct resurface.

3.1.8 with and rewrite

The implementation abounds in use of with and rewrite statements. with, introduced in the works of McBride and McKinna [12], allows pattern matching on intermediate computations. rewrite, on the other hand, replaces an expression on the left hand side, by making use of a supplied equality relation.

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

The use of these statements 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. With limited support from Agda's proof searching tools, the programmer has to provide proofs that ensure correct typing. Due to the nature of the *Builtin.Equality* relation, there is no trivial refactoring of the code, as suggested in the documentation. [4].

Discussion about with. There are various 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 behind a *with abstraction*, the definition of that function cannot makes use of further with statements containing the abstracted expression of the argument. This is discussed in the documentation: Ill-typed with-abstraction [5]. This is the reason that proving the correctness of ViewL seemed problematic.

- 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 an example in which I try to append an element at the end of a list, which does not type-check.

  ```
  snoc : ∀ {A} → A → List A → List A
  ```

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

```
snoc x xs with xs
snoc x xs | [] = x :: []
snoc x xs | y :: ys = y :: (snoc x ys)
```

Taking these 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 FingerTree.

I should note here that the proofs that I am making are still providing a powerful verification, since the FingerTree maintains all invariants and the measurement semantics is preserved and used sanely.

3.1.9 Folding

Next, I implement the fold operation and show its correctness. I will only present the fold-left implementation, since fold-right is analogous. [6]

Defining folds on **Node** and **Digit** is trivial, since they are just length constrained lists. I have presented them in the Appendix. Consider its implementation on the FingerTree:

```
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
```

The operation of folding over nested types has been extensively studied [7]. Further exemplifying implementations can be found in .... works [8].

3.1.10 Proving correctness of Fold Left

Next, I will show that the previous implementation is sane, by seeing whether folding over a FingerTree 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 }}
```

[6] To exemplify the semantics of foldl on a list, consider *foldl f s [x, y, z] = f (f (f s x) y) z*.

```
→ {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, we can prove that if we fold over the FingerTree using the **Monoid** and the **Measure**[7] over which it is instantiated, we obtain the same result as the measure.

```
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)
```

This result is important as a sanity check of the measure semantics we are trying to preserve throughout the implementation, since this fact can only be true if we preserve the order of calls to the monoid operator.

3.1.11 Splitting

Splitting is an extension of the ViewL paradigm, which allows extraction of elements arbitrarily far in the FingerTree. It consists of a left side, a middle element, and a right side.[8]

The same issues occur, as in the case of viewL. I have tried in this implementation to keep the usage of with to 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 returns when the value of the predicate turns true $p(i) = True$, giving the element at which this change occurs as 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
```

[7] foldfun v x = v • ‖ x ‖

[8] the sides are correct FingerTrees

```
  → (FingerTree A V {μ₂}) -- right side
  → Split-d A V {μ₁ • ‖ x ‖ • μ₂}
```

Since this implementation is long and 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 FingerTree 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 initial 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 on where the predicate changes to *True*. This could happen during the prefix **pr**, during the nested FingerTree **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. I will provide an explanation of why this application was more difficult in section 4.2.2.

3.2 Proving correctness

As in the case with viewL, proving statements about results with of a split application requires type checking within with abstractions. This feature is not available yet in Agda (throws the error: "Feature not Implemented") and I could not find a way around it.

However, as suggested in [18], the aim of the proof, namely that: For all ft, left, right : FingerTree, p : Boolean predicate, i : value and m : element of the finger tree, it must be the case that:

if split $p\ i\ ft$ = split−d $left\ m\ right$
then toList−ft ft = (toList−ft $left$) ++ [m] ++ (toList−ft $right$)

However, the type signature of the split function already ensures that:

$\| ft \|$ = $\|$split $p\ i\ ft$ $\|$

For the specific instance of the FingerTree using an arbitrary type A for elements and having values of type A List:

ft : FingerTree A (A List)

instantiated with the measurement which turns elements to singleton lists and the list monoid [9], we would find that:

$\| ft \|$ = foldl−ft _++_ [] ft = toList−ft ft

Given the fact that the size of split is given by $\mu_1 \bullet \| x \| \bullet \mu_2$ (see definition of Split−d), it would follow too that:

$\|$split $p\ i\ ft$ $\|$ = toList−ft ft = (toList−ft $left$) ++ [m] ++ (toList−ft $right$).

Unfortunately, I could not find a way to integrate such an argument in Agda, since it is not universally quantified. An intuitive idea is that this statement could hold for all A's and V's in an instantiation of the FingerTree combining an arbitrary measurement and monoid with the list instantiation.

Roughly, let f be the first measurement function, such that $f(x) = v \in V$, and a monoid over V with the usual notation. We can define the pair-measurement, g, such that

$g(x) = (f(x), [x]) \in V \times (ListA)$

[9] nil is the neuter element, and list concatenation (++) is an associative operation

and, furthermore, we can find a monoid in $(Monoid(V \times (ListA))$ by using the following definition:

```
pair−monoid : ∀ {a} {A : Set a} {B : Set a}
            → Monoid A
            → Monoid B
            → Monoid (A × B)
pair−monoid m₁ m₂ = monoid
  (ε₁ , ε₂) -- neutral element
  (λ p₁ p₂ → zip _•₁_ _•₂_ p₁ p₂) -- binary operation
... plus the associated proofs
  where
    ε₁ = Monoid.ε m₁
    ε₂ = Monoid.ε m₂
    _•₁_ = Monoid._•_ m₁
    _•₂_ = Monoid._•_ m₂
```

Unfortunately, I have not figured out how to formalise the fact that the set of all FingerTree A V is included in the set of FingerTree A $(V \times (ListA))$. Such a result will prove that the FingerTree is fully verified from the point of view of the elements it contains and their order.

3.3 Other recursive definitions

The example of implementing a reverse method brings to surface the difficulty of writing recursive functions, in particular those that output values whose type depends on one of the arguments.

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

```
reverse ft = foldl _◁_ Empty ft
```

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

- _◁_ is a dependent function, incompatible with our definition of foldl [6]

  It is possible to implement an analogous dependently typed version of foldl. However this would yield a specific solution only applicable to this particular case. Another strategy for achieving the same result is converting the FingerTree to a list and then performing the folding.

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

```
measure−ft (reverse ft) = foldl foldfun ϵ (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.

3.3.1 Reverse

The non-dependently typed interface is constructed through a dependent pair. The function pack−ft 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 relates the result to the argument, so it is not formally verified.

3.4 Random Access Sequences

By proper instantiation of the measurement function and monoid, we can specialise the FingerTree to various data structures. The first suggested one 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 in special types, aiding the use of instance arguments.

3.4.1 SizeW and Entry

The SizeW is simply a wrapper around Nat[10]

```
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 toSizeW, so that we can populate a MonoidSizeW

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

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

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
```

Having described both the monoid and the measurement function, we have essentially defined an instance of the FingerTree.

3.4.2 Seq Instance

The Sequence instance is simply an alias for a FingerTree, 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}
```

The main operations expected on a Random Access Sequence (Seq) are getting and setting an element at an arbitrary position.

Getting the nth element

The naive implementation would simply call viewL n times, yielding an amortised linear cost. However, we can find an $\mathcal{O}(log(n))$ implementation by using split. The predicate checks whether the argument value is smaller than n, whereas the argument is initialised to 0 and increases by the number of elements skipped in every iteration.

```
_!_ : ∀ {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 FingerTree concatenation. [11]

```
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)
```

[11] I am referring to a linear time concatenation. The logarithmic implementation presented in [9] is a suggestion for future work

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

This works by extracting the *nth* element and concatenating the *left* and *right* FingerTrees around the new value.

3.5 Writing recursive definitions

As mentioned in section 3.1.6, the use of with can sometimes confuse the termination checker. This, combined with the nature of the viewL, makes it hard to make use of the abstraction brought by deconstructing FingerTrees as we would deconstruct normal Linked Lists.

In fact, Agda's termination checker cannot prove that, for a FingerTree *ft*, viewL (x ◁ 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 Matthieu Souzeau's paper [18] that to overcome this problem, one must find a suitable indexing that reflects the number of elements in the FingerTree. This is exactly what the Random Access Sequence achieves.

3.5.1 Wellfounded 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 [12], 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 recognises.

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 [13] to construct a Well-Founded relation on Sequences using a Well-Founded relation on the Sizes.

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

[12] http://adam.chlipala.net/cpdt/html/GeneralRec.html

[13] also implemented in the standard library

```
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-foundedness, we can proceed by creating a recursor object and writing a recursive definition.

3.5.3 Reversing

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 debatable 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 FingerTree.

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 properties. As a concrete example, 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. [14]

```
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

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))
```

The problem that remains is proving the inductive step. Because of the limitations of the *view* operation and the difficulty of pattern matching on a size-constrained FingerTree (Seq A ((size 1) • s)), I have not been able to find terms of this type.
Finding a general method of proving such properties, in the context of views[22], would solve many of the problems encountered throughout this dissertation, suggesting potential for further work.

3.6 Summary

In this section, I have presented an implementation of the FingerTree [9], using Agda [15] as the programming language. The implementation is formally verified to the extend discussed above, ensuring the correct measure of the FingerTree, as well as the correctness of the operations with respect to the order of the elements. I am showing how the FingerTrees can be specialised to RandomAccessSequences, suggesting that properties proven about the FingerTrees carry through to the data structure they generalise. Finally, I am showing limitations of the Agda systems, and arguing further directions as necessary.

[14] At the time of implementation, I had not yet realised this trick. It would be an interesting extension to change the mechanism employed by the rewrite to use something like that, allowing the use of 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 dependently typed 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 a supportive argument towards the use of dependent types, and understanding what causes their 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.

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.

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

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

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. Since Agda has lazy evaluation, I was required to force its evaluation by a constant time *get* operation[3]. The output is done via the Haskell-like Monad way.

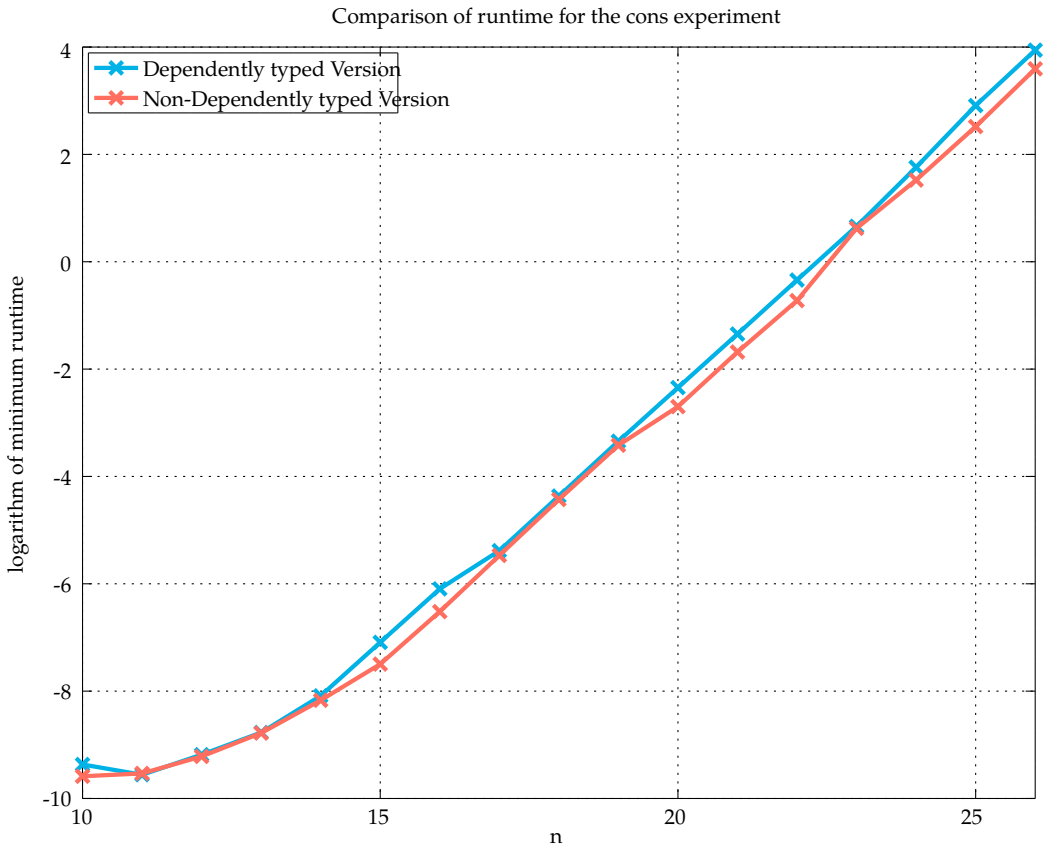
```
big−seq : (n : ℕ) → Seq ℕ (size n)
big−seq zero = Empty
big−seq (suc n) = (entry n) ◁ (big−seq n)


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

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

[3]Getting the first element is guaranteed to have a negligeable run-time. Since it resides in the leftmost digit, it triggers only three function calls.

Figure 4.1: Consing experiment

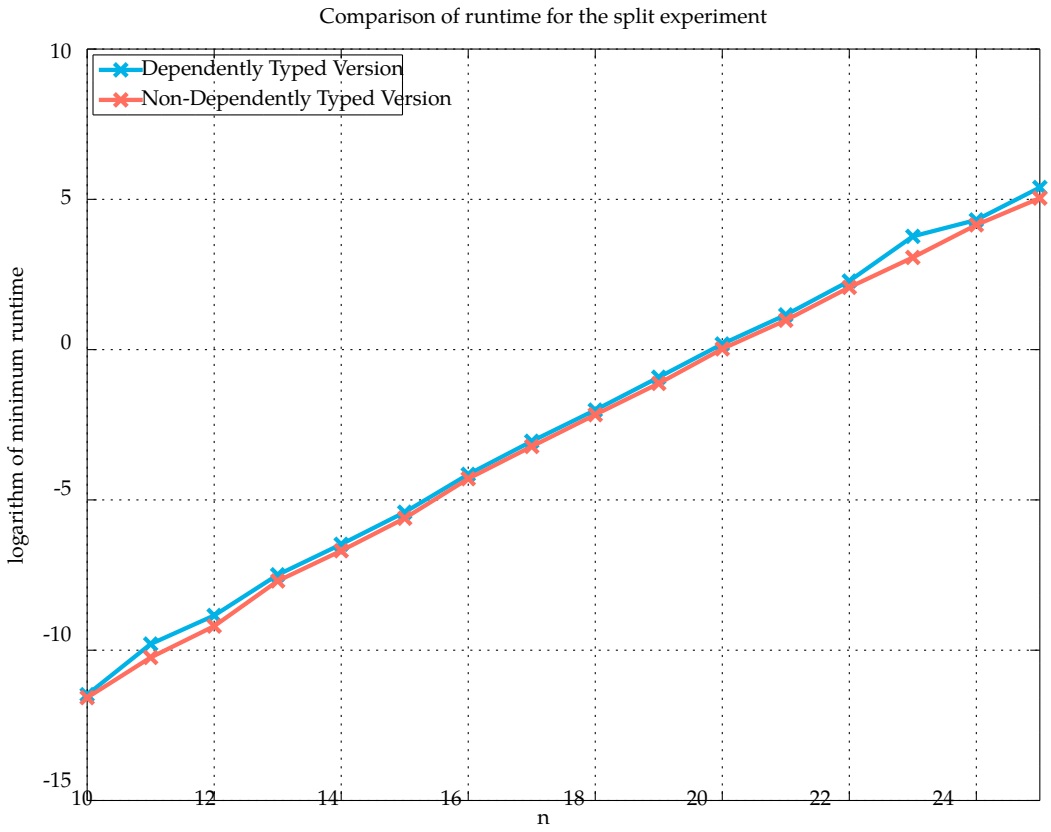


4.1.2 split

This experiment is an extension of the previous one. In addition to constructing a big−sequence, we also invoke the split function in the form of *get*(_!_). In order to obtain a non-trivial result, I am computing the time required to extract the middle element.

Since performing the *get* operation requires creating a big−sequence in the same way as above, I have subtracted the results from the previous experiment, so that values reflect only the minimum time taken for the extraction of the middle element from increasingly long sequences.

Figure 4.2: Splitting Experiment



It is clear from both these experiments that the dependently typed versions incur 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 dependently typed 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 emphasising 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[4], 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[5].
- I have been consistent with jumping to new lines.

I will make a clear distinction between *internal* verification and *external* verification [19]. 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 categorise the lines in four categories. **Type annotations** (declaring the function about to be defined as well as data declarations), **Implementation** (definitions of functions that are relevant to the implementation), **Proofs** (lines related to proofs and the calls to them), **Other** (importing modules, *where* clauses etc.).

Selection Sort. The selection sort procedure, presented in section 2.5, 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.

Table 4.2: SLOC for Sorting

Content	Verified	Not Verified
Implementation	18	14
Type annotations	24	3
Proofs	124	0
Other	53	50
Total	219	67

[4] Source Lines of Code

[5] It is debatable whether the metrics will show something about using dependent types or just about myself as a programmer.

The non-verified 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 FingerTree example.

Table 4.3: SLOC for FingerTree

Content	Measured Version	Size Version
Implementation	185	162
Type annotations	195	100
Proofs	350	30
Other	30	22
Total	760	314

FingerTree. 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.

In the dependently typed setting, the Implementation and Type annotation carry almost equal weight, in contrast to a 3:1 ratio in the non-dependently typed one.

However, 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.

At the beginning of the project, I was expecting that the automatic proof search tool would make a great difference [10]. Unfortunately, the cases in which it returned results were very few in most of the files, and nonexistent in the implementation of the dependent FingerTree.

Some proofs were not included here, due to the fact that to derive them I used a reimplementation of the MonoidSolver (required because of incompatibilities with the standard library). This automates the proof through the use of reflection [21]. Such proofs are constructed at compile time and I could observe them using normalization. Unfortunately, they are considerably longer than the other proofs I have written, so including them would have made this analysis inconsistent.

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*, ϵ is *neutral element*
- about List: *associativity* of ++ and *[]* is *neutral element*

I have performed an analysis of which 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 a potential starting point for refactoring. A summary of results is 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

This graph suggests two things. First of all, due to the declaration of the monoid operator as **infixl**, the operations that have a right-associative structure tend to be easier to prove. This creates a contrast between cons and viewL on one side and snoc and viewR on the other.

I believe that the split operation is even more difficult because apart from taking over the difficulties of both viewL and viewR, it also combines two different types of logical statement.

Although we only rely on the Curry Howard Isomorphism, the alternative reasoning also incorporates the Boolean data type and its associated operations (and, or, negation). Instead of proceeding as before, using:

```
proof−istrue : ∀ a → property a
proof−isfalse : ∀ a → (property a → ⊥)
```

We now also make use of:

```
pred−istrue : ∀ a → (predicate a ≡ true)
pred−isfalse : ∀ a → (predicate a ≡ false)
```

This inconvenience became more noticeable when I tried to port the Isabelle implementation [13], which is not based on dependent types. Although they can be fully encoded, the proofs become long due to the lack of type checking support. Some examples of this are in the Appendix 6.5. This would perhaps be a starting point for some future work.

4.3 Discussion

As part of the introduction, I posed a question related to how far Agda's expressivity can take us in the implementation of a functional data structure.

In relation to Matthieu Souzeau's paper [18], I have completed the implementations of all the operations he suggested. Furthermore, I have presented an Agda way to deal with the termination checking problem, discussed in his section 4.4.1 Dependence Hell. It is interesting to notice that this problem is not strictly related to Agda, but to a wider family of dependently typed languages.

In addition, I have also explicitly proven axioms that relate FingerTrees to their list representation.

Chapter 5

Conclusion

5.1 Accomplishments

In this dissertation, I have successfully implemented the FingerTree data structure [9] in Agda [15] and provided verified definitions for the data structure described in Table 2.3. The results were similar to those obtained by Mattheiu Souzeau [18], as both implementations suffered from the same limitations. From this, I draw the conclusion that even though Coq is a richer platform, it does not necessarily achieve more, especially in cases when simplicity is important. The inferiority of proof automation could become a blessing in disguise in Agda, since the development of the FingerTree almost lead me to rediscover the proof by reflection employed in the MonoidSolver.
I managed to outline and give extensive examples to the problem *Sized Types* can cause when combined with functions on nested types. The source of the difficulty has been pointed out as being firstly because of the lack of pattern matching support and secondly due to the creation of negative cycles in the sizing graph. I have discussed the termination-checking limitation of dependently-typed theorem provers, and also presented an Agda implementation of the solution suggested in [18] in Section 3.5.3.
In order to achieve these results, I have provided a concise introduction to Agda and programming with dependent types, as well as examples of proofs and other similar data structures where the same issues occur (included in the Appendix due to size constraints). I have evaluated the implementations, as well as provided some insight into the difficulty of developing proofs in Agda. Furthermore, I have managed to compile a substantial fragment of the relevant literature and provide satisfying examples that connect the essential concepts.

5.2 Further work

In the course of developing this dissertation, I have noticed some Agda limitations. I have illustrated them throughout the previous sections. They are all potential starting points for future work:

- Type-checking within `with` abstracted statements.

- Use of *Sized Types* for nested types.
- Compilation limitations.

Considering the implementation itself, I can make the following suggestions:

- With regard to my original proposition of packaging the implementation in a library – the outcome of this project could have easily been included in the standard library, had I used the already implemented Monoid record and universe level conventions. This is a straightforward extension that could be carried out and would benefit the community.
- Auxiliary constraints of the measuring function would allow writing recursive definitions using the approach presented in Section 3.5.3. One could enforce that:

  $$\mathsf{mpos} : \forall\ x \to \varepsilon > \| x \|$$

  This would ensure that adding a new element to the FingerTree yields one with a bigger size (where bigger is given by the _>_ relation), formally by ensuring that: $\forall$ s : V, x : A. s $\bullet \|x\| >$ s. This allows writing arbitrary recursive definitions using the paradigm from Section 3.5.3.
- Related to the discussion about the Isabelle implementation [13] and porting it fully to Agda, I believe that a module similar to the Propositional Equality can be implemented to facilitate doing proofs of the type:

  $$\mathsf{predicate}\ a \equiv \mathsf{false}$$

5.3 Lessons learned

During this project, I have familiarised myself with Agda, and gained sufficient insight into type theory to be able to read and understand cutting edge research. I have acquired invaluable experience in formal verification and its application to testing or prototyping data structures. Successful verification requires some practice in order to get used to the environment's quirks, as in many cases a well chosen definitions for theorems is key to the process.

The field of formal verification using dependent types is slowly entering the industry, through the advent of more accessible programming languages like Agda, Idris or Dependent Haskell. Although still young, such tools show potential and are surrounded by active communities. I have shown, through implementing a non trivial data structure, the trade-off between expressivity and limitations in Agda, and I hope that I have inspired further work and study in this area.

Chapter 6

Appendix

6.1 Helper Functions

- Flattening a list of Node A V to a list of A, by repeatedly transforming nodes to lists and aggregating the result.

```
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)
```

6.2 Numerical Representations

The treatment of containers as natural numbers has been studied in depth[16]. The basic idea is that simple numerical operations correspond naturally to operations on containers. For example:

increasing a number	corresponds to	adding an element
decreasing a number	corresponds to	removing an element
adding two numbers	corresponds to	merging to containers

This treatment of numbers, represented in various numerical basis, allows the constructions the obey the implicit recursive slowdown, presented by okasaki. This allows in lazy languages like Haskell, implementation of operations such as insertion and deletion in ammortised O(1) cost – which represented a breakthrough in functional programming languages.

6.3 Further example of the termination checking limitation

In this section, I will present a data structure as implemented by Ralf Hinze[7] and show the issues that could arise because of the termination checker in more detail.

Consider the trivial implementation of a binary tree in a functional programming language:

```
data Bush (A : Set) : Set where
  Leaf : A → Bush A
  Fork : Bush A → Bush A → Bush A
```

In order to stay consistent with the original implementation, the data structure above will be split in two different types that represent the constructors [**?**].

```
data Leaf (A : Set) : Set where
  LEAF : A → Leaf A

data Fork (B : Set → Set)(A : Set) : Set where
  FORK : (B A) → (B A) → Fork B A
```

We can now refer to the Random Access Sequence implementation. They are a numerical representation based on base two of natural numbers, however, rather than the 0-1 system, the author prefers to use the 1-2 system for a number of effiency reasons.

$$\begin{aligned} inc(\epsilon) &= 1 \\ inc(1a) &= 2a \\ inc(2a) &= 1inc(a) \end{aligned}$$

This inc operator should correspond analogously to the 'Cons' operators in the data structure:

```
data RandomAccessList (B : Set → Set) (A : Set) : Set where
  Nil : RandomAccessList B A
  One : (B A) → (RandomAccessList (Fork B) A) → RandomAccessList B A
  Two : (Fork B A) → (RandomAccessList (Fork B) A) → RandomAccessList B A
```

Now, by implementing the function $incr$, we can see the similarity between the adding an element to the left and the number representation

```
incr : {B : Set → Set} {A : Set} → (B A)
  → RandomAccessList B A
  → RandomAccessList B A
incr b Nil = One b Nil
incr b (One b₂ ds) = Two (FORK b b₂) ds
incr b (Two b₂ ds) = One b (incr b₂ ds)
```

Following, I declare the sequence by using the definition of Leaf as a layer of abstraction.

```
IxSequence : Set → Set
IxSequence = RandomAccessList Leaf

cons : {A : Set} → A → IxSequence A → IxSequence A
cons a s = incr (LEAF a) s
```

6.3.1 Defining a view

We can then implement the $front$ method, which returns a view of the list in terms of the first element and a continutation. Our goal is to abstract away the intricacy of the type declaration, so we can implement methods easily. First, we need to declare the return type, wrapped in a view data structure.

```
data View (A : Set) : Set where
  Vnil : View A
  VCns : A × IxSequence A → View A

front : {A : Set} → IxSequence A → View A
front Nil = Vnil
front (One (LEAF x) ds) = VCns (x , zero ds)
front (Two (FORK (LEAF a) b) ds) = VCns (a , One b ds)
```

The zero method is a restructuring method, as we will find in the FingerTree implementation.

```
zero : {B : Set → Set} {A : Set} →
  RandomAccessList (Fork B) A →
  RandomAccessList B A
zero Nil = Nil
zero (One b ds) = Two b (zero ds)
zero (Two (FORK b₁ b₂) ds) = Two b₁ (One b₂ ds)
```

6.3.2 Example termination failure

Here, Agda termination checker will fail. We will try to implement an append function, which is a straightforward process given the methods previously declared:

```
append : {A : Set} → A → IxSequence A → IxSequence A
append x seq with front seq
append x seq | Vnil = cons x Nil
append x seq | VCns (head , tail) = cons head (append x tail)
```

6.3.3 Using sized types

Sized types are agda's response to fixing such issues. However, trying to come up with an implementation that type checks, even in this relatively simple case seems to be very difficult. The intuition in this case is that we need to convince agda that FORK a b is bigger than any individual a b in the context of the RAL constructors. However, sized types are only relative, not on an absolute scale.

```
data RandomAccessList (B : Set → Set)(A : Set) : {i : Size} → Set where
  Nil : ∀ {i} → RandomAccessList B A {i}
  One : ∀ {i} → (B A) → (RandomAccessList (Fork B) A {i})
    → RandomAccessList B A {↑ i}
  Two : ∀ {i} → (Fork B A) → (RandomAccessList (Fork B) A {i})
    → RandomAccessList B A {↑ ↑ i}

incr : {B : Set → Set} {A : Set} {i : Size} → (B A)
  → RandomAccessList B A {i}
  → RandomAccessList B A {↑ i}
incr b Nil = One b Nil
incr b (One b₂ ds) = Two (FORK b b₂) ds
incr b (Two b₂ ds) = {!!} -- One b (incr b  ds)
```

Consider the implementation of incr. The problem arises when we are recursively calling *incr b ds* . This is where the complication of nested types arose in the first place. incr is a polymorphic function, so in the second interation it would be instantiated with

$$B' = FORK\,B$$
$$A' = A$$

Now, it is obvious that inserting an element of type Fork B A should increase the size of the container by more than inserting an element of type A would. Under this polymorphism however, the two operations are equivalent. The solution to this problem would require a size scaled on the type, so that *size(Fork B A) > size(A)*. However, this needs to be hardcoded for specific type, as Agda has no way of differentiating between different types of type Set, so no general method is available.

6.4 Other Agda Syntax and Terminology

6.4.1 Agda's interactive help

Before writing the implementation of a function, as you stumble upon the equals(=) sign, you can tell agda to place a hole (! !) instead of an implementation. Here, you can perform a number of operations:

- See the types and values of variables in the scope

- Case-split
  For example, consider the addition of natural numbers.

  ```
  _+_ : ℕ → ℕ → ℕ
  m + n = ?
  ```

  Performing a case-split on the variable n shows me all the possible ways in which a natural number can be constructed.

  ```
  _+_ : ℕ → ℕ → ℕ
  zero + n = ?
  suc m + n = ?
  ```

- Refine and Auto
  These provide the automated and interactive ways of theorem-proving. Essentially, Agda looks throughout the environment to find an inhabitant (a variable) that has the type of the hole.
  They are definitely not as powerfull as any functionality given by Coq of Isabelle, but it can save some typing.

6.4.2 Implicit Arguments

Agda introduces some syntax for various types of arguments you can provide to functions. As you probably saw, there is a difference in handling the polymorphic types (in the case of List) and the values given as arguments to type constructors (in the case of Vec).

In the declaration of Vec:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

The first (A : Set) is the type argument for instantiating a polymorphic type, before the :, while the is the type of the value argument for the dependently typed instantiation.

Another think to notice here is the curly brackets ({n : }) in the declaration of the _::_ constructor. This is called an implicit argument. Agda will bind n to a value it sees fit in the scope. If there are more possibilities, it will take a guess.

6.4.3 Instance Arguments

Throughout this dissertation, we will be using some properties of certain types, for example of having a monoid operation associated with them. In Haskell, you would accomplish that with the use of type classes **[?]**

In order to mimic this behaviour we will use instance arguments. They are declared by using double square brackets, {{ }} or the unicode equivalent.
What Agda does in this case, it looks for a possible instantiation of that type in the current scope, following some predefined rules. [**?**] It is important there is only one available possibility, otherwise it will fail to type check.

The use will become obvious in the Implementation section.

6.5 Sorting – full code

In the following, I present the sorting algorithm of which I referred in Section 6.5.

```
open import Data.List using (List; []; _∷_)
open import Data.Maybe
open import Data.Nat renaming (_≤_ to _≤n_)
open import Data.Product
open import Data.Bool
open import Data.Sum
open import Relation.Binary.PropositionalEquality
open import Data.Vec using (Vec; []; _∷_)

record PartialOrder (A : Set) : Set where
  constructor poset
  field
    _≤_ : A → A → Bool
    ≤refl : ∀ a → (a ≤ a ≡ true)
    ≤trans : ∀ a b c
      → (a ≤ b ≡ true)
      → (b ≤ c ≡ true)
      → (a ≤ c ≡ true)
    ≤neg : ∀ a b → (a ≤ b ≡ false) → (b ≤ a ≡ true)

and-left : ∀ a b → (a ∧ b ≡ true) → (a ≡ true)
and-left false b p = p
and-left true b p = refl

and-right : ∀ a b → (a ∧ b ≡ true) → (b ≡ true)
and-right false false p = p
and-right true false p = p
and-right a true p = refl

and-combine : ∀ a b → (a ≡ true) → (b ≡ true) → (a ∧ b ≡ true)
and-combine .true b refl q = q

module sorting (A : Set) (pos : PartialOrder A) where

  _≤_ = PartialOrder._≤_ pos
  ≤refl = PartialOrder.≤refl pos
  ≤trans = PartialOrder.≤trans pos
  ≤neg = PartialOrder.≤neg pos
```

```
data _∈_ {A : Set} : {n : ℕ} → (x : A) → (xs : Vec A n) → Set where
  found : ∀ {n} → (x : A) → (xs : Vec A n) → x ∈ (x :: xs)
  skip : ∀ {n} (x : A) → (y : A) → (xs : Vec A n) → (x ∈ xs) → x ∈ (y :: xs)

all : ∀ {n : ℕ} → (p : A → Bool) → (ys : Vec A n) → Bool
all p [] = true
all p (x :: ys) = (p x) ∧ (all p ys)

data _ins_≡_ {A : Set} : {n : ℕ} → A → Vec A n → Vec A (suc n) → Set where
  stop : ∀ {n : ℕ} {x : A} {xs : Vec A n}
      → x ins xs ≡ (x :: xs)
  go : ∀ {n : ℕ} {x y} {xs : Vec A n} {ys : Vec A (suc n)}
      → x ins xs ≡ ys
      → x ins (y :: xs) ≡ (y :: ys)

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)

incl−prop0 : ∀ {A : Set} {n} → (x : A) → (y : A) → (xs : Vec A n) → (x ∈ xs)
  → (x ∈ (y :: xs))
incl−prop0 x y xs prop = skip x y xs prop

incl−prop1 : ∀ {A : Set} {n} → (x : A) → (y : A) → (z : A) → (xs : Vec A n)
  → (x ∈ (z :: xs)) → (x ∈ (z :: y :: xs))
incl−prop1 x y .x [] (found .x .[]) = found x (y :: [])
incl−prop1 x y₁ y [] (skip .x .y .[] prop) =
  skip x y (y₁ :: []) (skip x y₁ [] prop)
incl−prop1 x y .x (x₁ :: xs) (found .x .(x₁ :: xs)) =
  found x (y :: x₁ :: xs)
incl−prop1 x y z (x₁ :: xs) (skip .x .z .(x₁ :: xs) prop) =
  skip x z (y :: x₁ :: xs) (skip x y (x₁ :: xs) prop)

get−min : ∀ {n : ℕ} → (i : A) → (xs : Vec A n) → A
get−min i [] = i
get−min i (x :: xs) = if (i ≤ x) then get−min i xs else get−min x xs


min−is−min1 : ∀ {n : ℕ}
  → (i : A)
  → (xs : Vec A n)
  → (min : A)
  → (min ≡ get−min i xs)
  → (min ≤ i ≡ true)
min−is−min1 i [] .i refl = ≤refl i
```

```
min-is-min1 i (x :: xs) _ refl with i ≤ x | inspect (i ≤_) x
min-is-min1 i (x :: xs) _ refl | false | re = ≤trans (get-min x xs) x i rec neg
  where
    rec : (get-min x xs ≤ x) ≡ true
    rec = min-is-min1 x xs (get-min x xs) refl
    --
    neg : (x ≤ i ≡ true)
    neg = ≤neg i x (Reveal_·_is_.eq re)
min-is-min1 i (x :: xs) _ refl | true | r = min-is-min1 i xs (get-min i xs) refl

min-is-min : ∀ {n : ℕ}
  → (i : A)
  → (xs : Vec A n)
  → (min : A)
  → (min ≡ get-min i xs)
  → (min ≤ i) ∧ (all (min ≤_) xs) ≡ true
min-is-min i [] .i refl = and-combine (i ≤ i) true (≤refl i) refl
min-is-min i (x :: xs) min rf with i ≤ x | inspect (i ≤_) x
min-is-min i (x :: xs) .(get-min x xs) refl | false | re =
  and-combine ((get-min x xs) ≤ i)
              ((get-min x xs ≤ x) ∧ all (_≤_ (get-min x xs)) xs) term0 term2
  where
    neg : (x ≤ i ≡ true)
    neg = ≤neg i x (Reveal_·_is_.eq re)
    term1 = min-is-min1 i xs (get-min i xs) refl
    term2 = min-is-min x xs (get-min x xs) refl
    term0 : (get-min x xs) ≤ i ≡ true
    term0 = ≤trans (get-min x xs) x i
      (and-left ((get-min x xs ≤ x)) (all (_≤_ (get-min x xs)) xs) term2) neg
min-is-min i (x :: xs) .(get-min i xs) refl | true | re =
  and-combine-3 ((get-min i xs ≤ i))
    ((get-min i xs ≤ x))
    (all (_≤_ (get-min i xs)) xs) term1 term2 term4
  where
    term0 : (i ≤ x ≡ true)
    term0 = Reveal_·_is_.eq re

    term1 : (get-min i xs ≤ i) ≡ true
    term1 = min-is-min1 i xs (get-min i xs) refl

    term2 : (get-min i xs ≤ x) ≡ true
    term2 = ≤trans (get-min i xs) i x term1 term0

    term3 : (get-min i xs ≤ i) ∧ all (_≤_ (get-min i xs)) xs ≡ true
    term3 = min-is-min i xs (get-min i xs) refl

    term4 : all (_≤_ (get-min i xs)) xs ≡ true
    term4 = and-right (get-min i xs ≤ i) (all (_≤_ (get-min i xs)) xs) term3

    and-combine-3 : ∀ a b c → (a ≡ true) → (b ≡ true) → (c ≡ true)
      → (a ∧ b ∧ c ≡ true)
```

```
    and-combine-3 .true .true .true refl refl refl = refl


get-min-incl : ∀ {n : ℕ} → (i : A) → (xs : Vec A n)
  → (get-min i xs ∈ (i :: xs)) ⊎ (get-min i xs ∈ xs)
get-min-incl  i [] = inj₁ (found i [])
get-min-incl  i (x :: xs) with i ≤ x
get-min-incl  i (x :: xs) | false with (get-min-incl x xs)
get-min-incl  i (x :: xs) | false | inj₁ x₁ =
  inj₂ x₁
get-min-incl  i (x :: xs) | false | inj₂ y =
  inj₂ (skip (get-min x xs) x xs y)
get-min-incl  i (x :: xs) | true with (get-min-incl  i xs)
get-min-incl  i (x :: xs) | true | inj₁ x₁ =
  inj₁ (incl-prop1 (get-min i xs) x i xs x₁)
get-min-incl  i (x :: xs) | true | inj₂ y =
  inj₁ (skip (get-min i xs) i (x :: xs) (skip (get-min i xs) x xs y))

incl-prop3 : ∀ {A : Set}{n : ℕ} → (a : A) → (x : A)
  → (xs : Vec A n) → (x ∈ xs) → (a ∈ (x :: xs)) → (a ∈ xs)
incl-prop3 a .a xs pr1 (found .a .xs) = pr1
incl-prop3 a x xs pr1 (skip .a .x .xs pr2) = pr2

incl-prop2 : ∀ {A : Set}{n : ℕ} → (a : A) → (x : A)
  → (xs : Vec A n) → (x ∈ xs) → ((a ∈ (x :: xs)) ⊎ a ∈ xs)) → (a ∈ xs)
incl-prop2 a x xs prf (inj₁ x₁) = incl-prop3 a x xs prf x₁
incl-prop2 a x xs prf (inj₂ y) = y

extract-element : ∀ {A : Set}{n : ℕ}
  → (x : A)
  → (xs : Vec A (suc n))
  → (x ∈ xs)
  → Vec A n
extract-element x .(x :: xs) (found .x xs) = xs
extract-element {n = zero}
    x
    .(y :: [])
    (skip .x y [] prf) = []
extract-element {n = suc n}
    x
    .(y :: xs)
    (skip .x y xs prf) = y :: extract-element x xs prf

extract-insert : ∀ {A : Set}{n : ℕ}
    → (x : A)
    → (xs : Vec A (suc n))
    → (prf : x ∈ xs)
    → (ys : Vec A n)
    → (ys ≡ extract-element x xs prf)
    → (x ins ys ≡ xs)
extract-insert x (.x :: ys) (found .x .ys) .ys refl = stop
```

```
extract-insert x (y :: []) (skip .x .y .[] ()) .[] refl
extract-insert x₁
               (y :: x :: xs)
               (skip .x₁ .y .(x :: xs) prf)
               .(y :: extract-element x₁ (x :: xs) prf)
               refl = go (extract-insert x₁ (x :: xs)
                 prf
                 (extract-element x₁ (x :: xs) prf)
                 refl
)

extract-element-min : ∀ {n : ℕ}
      → (a : A)
      → (x : A)
      → (xs : Vec A (suc n))
      → (prf : x ∈ xs)
      → (all (a ≤_) xs ≡ true)
      → (all (a ≤_) (extract-element x xs prf) ≡ true)
extract-element-min  a x .(x :: xs) (found .x xs) min =
  and-right (a ≤ x) (all (_≤_ a) xs) min
extract-element-min  a x .(y :: []) (skip .x y [] prf) min = refl
extract-element-min  a x₁ .(y :: x :: xs) (skip .x₁ y (x :: xs) prf) min =
  and-combine
    (a ≤ y)
    (all (_≤_ a) (extract-element x₁ (x :: xs) prf))
    (and-left (a ≤ y) ((a ≤ x) ∧ all (_≤_ a) xs) min)
      (extract-element-min  a x₁ (x :: xs) prf (
      and-right true ((a ≤ x) ∧ all (_≤_ a) xs) (
      and-right (a ≤ y) ((a ≤ x) ∧ all (_≤_ a) xs) min)))


min-start-prop0 : ∀ {n : ℕ} → (x : A) → (xs : Vec A n)
      → (get-min x (x :: xs) ≡ get-min x xs)
min-start-prop0 x [] rewrite ≤refl x = refl
min-start-prop0 x (y :: xs) with x ≤ x |  ≤refl x | x ≤ y
min-start-prop0 x (y :: xs) | .true | refl | false = refl
min-start-prop0 x (y :: xs) | .true | refl | true = refl

sort : ∀ {n : ℕ} → (xs : Vec A n) → (SortedList xs)
sort [] = []
sort (x :: []) = [ x ]
sort (x :: xs) = (min :: (sort rest)) min-sublist reconstruct
  where
    min = get-min x (x :: xs)

    min-in-list : min ∈ (x :: xs)
    min-in-list = (incl-prop2 min x (x :: xs)
      (found x xs)
      (get-min-incl x (x :: xs)))

    rest = extract-element min (x :: xs) min-in-list
```

```
    reconstruct : min ins rest ≡ (x :: xs)
    reconstruct = extract-insert min (x :: xs) min-in-list rest refl

    min-start : min ≡ get-min x xs
    min-start = min-start-prop0 x xs

    min-sublist : all (min ≤_) rest ≡ true
    min-sublist = extract-element-min
      min
      min
      (x :: xs)
      min-in-list
      (min-is-min x xs min min-start)


_≤nat_ : ℕ → ℕ → Bool
zero ≤nat zero = true
zero ≤nat suc m = true
suc n ≤nat zero = false
suc n ≤nat suc m = n ≤nat m

≤reflnat : (n : ℕ) → (n ≤nat n ≡ true)
≤reflnat zero = refl
≤reflnat (suc n) = ≤reflnat n

≤transnat : (n : ℕ) → (m : ℕ) → (p : ℕ)
  → (n ≤nat m ≡ true)
  → (m ≤nat p ≡ true)
  → (n ≤nat p ≡ true)
≤transnat zero zero zero p1 p2 = refl
≤transnat zero zero (suc p) p1 p2 = refl
≤transnat zero (suc m) zero p1 p2 = refl
≤transnat zero (suc m) (suc p) p1 p2 = refl
≤transnat (suc n) zero p () p2
≤transnat (suc n) (suc m) zero p1 ()
≤transnat (suc n) (suc m) (suc p) p1 p2 = ≤transnat n m p p1 p2

≤neg : (n : ℕ) → (m : ℕ) → (n ≤nat m ≡ false) → (m ≤nat n ≡ true)
≤neg zero zero p = refl
≤neg zero (suc m) ()
≤neg (suc n) zero p = refl
≤neg (suc n) (suc m) p = ≤neg n m p

-- open sorting { }

NatOrder : PartialOrder ℕ
NatOrder = poset _≤nat_ ≤reflnat ≤transnat ≤neg

open sorting ℕ NatOrder using (sort)
```

```
a = sort (2 :: 3 :: 1 :: 0 :: [])
```

I have included this example in the appendix because it is relatively small complete example that illustrates some key facts of formal verification in Agda.

First, posing the problem in way compatible with the implementation can aid the proof section. Refer the Section 6.5.

Secondly, I illustrates the difficulty of integrating the type of reasoning illustrated in Section 4.2.2. This implementation became more lengthy because of having to define helper functions like and−combine−3, and−combine, and−left, and−right, which require explicit definitions of terms, although the inference algorithm should be able to identify them. Moreover, I have included proofs which are also present in the standard library for completeness (nat, reflnat, transnat, neg) to be able to instantiate and test the implementation.

Bibliography

[1] Andreas Abel. Miniagda: Integrating sized and dependent types, 2010.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.

[3] THORSTEN ALTENKIRCH ANDREAS ABEL. A predicative analysis of structural recursion, 19999.

[4] Richard Bird and Lambert Meertens. Nested datatypes. *LNCS*, 1422:52–67, 1998.

[5] Adam Chipala. *Certified programming with dependent types*. 2016.

[6] Nicolas Pouillard Daniel Gustafsson. Foldable containers and dependent types.

[7] Ralf Hinze. Numerical representations as higher-order nested datatypes, 1998.

[8] Ralf Hinze and Ross Paterson. De bruijn notation as a nested datatype. *Journal of Functional Programming*, 1999.

[9] Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

[10] Pepijn Kokke and Wouter Swierstra. *Auto in Agda*, pages 276–301. Springer International Publishing, Cham, 2015.

[11] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[12] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.

[13] Benedikt Nordhoff, Stefan Körner, and Peter Lammich. Finger trees. *Archive of Formal Proofs*, October 2010.

[14] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[15] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[16] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[17] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, page 28. Pearson Education, 1999.

[18] Matthieu Sozeau. Program-ing finger trees in coq. *SIGPLAN Not.*, 42(9):13–24, October 2007.

[19] Aaron Stump. *Verified Functional Programming in Agda*. Morgan and Claypool, 2016.

[20] David Thibodeau. Termination checking: Comparing structural recursion and sized types by examples, 2011.

[21] Paul van der Walt and Wouter Swierstra. *Engineering Proof by Reflection in Agda*, pages 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[22] Phil Wadler. *Views: A way for pattern matching to cohabit with data abstraction*, pages 307–313. ACM, 1987.