Razvan Kusztos

Verified functional datastructures in Agda

Diploma in Computer Science

Girton College

April 29, 2017

Proforma

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

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

Declaration

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

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

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

Chapter 1

Introduction

1.1 Verified Programming

Formal verification is a way of proving correctness of program, by using a formal mathematical method. This can come in many flavours, from taking code that is already written and showing that it preserves some invariants in another language (as in Hoare logic) to writing mathematical proofs. The former process can greatly benefit from developing computer software that can aid and verify the proving process. There are a lot of such systems, form which one can mention Coq, Isabelle/HOL or Agda.

These environments differ in a number of ways, and for the purpose of this project I have decided to use Agda.

Firstly, Isabelle is limited to performing 'external' verification, which refers to a clear separation between the code - subject of the proof and the proof itself. This provides a great advantage for readability and code extraction. However, for this project I was particularly interested in the way in which the typing constraints can enforce correctness.

On the other hand, Coq and Agda are both examples of programming languages with dependent types. They are based on Martin Lof intuitionistic type theory. Coq was originally designed as a theorem prover, and is much older than Agda. My initial argument for picking Agda is its relative simplicity and steeper learning curve. By using it, I have also discovered some other advantages, such as allowing mutual definitions, mixfix syntax, and the overall results of building it while regarding interactivity. Coq is however a more powerful environment, as it allows the use of tactics, while in Agda they have to be implemented (I provide an example in Section - - -)

1.2 Functional Datastructures

There has always been an imballance between the use of functional programming versus imperative programming both in industry, as well as in terms of available resource. Functional programming has often been ruled out in the past because of it running slow, or simply because it was stigmatised as belonging into academia, regardless of its properties [**?**]. However, this paradigm is being introduced now at the forefront of business development. This is because of the persistency[0] of data-structures, useful for the ever-present multicore environment. The reliablity aspect, formal verification and keeping runtime errors to a minimum have also been relevant. The issue of the runtime speed has been addressed gracefully by Okasaki [**?**], which, introduced a reusable concept that can help designers build efficient data structures: the implicit recursive slowdown.

1.3 Dependent Typing

An important breakthrough in writing verifiably correct code is the introduction of dependent types.[**?**] In this setting the distinction between types and values becomes blurry, allowing us to define types that depend on values.

An immediate practical motivation is performing the sum of two vectors. The usual programming paradigm would be (in pseudocode):

```
def sum (l1, l2):
  if (l1.length != l2.length)
    raise ListsNotEqualException;
  ...
```

This can cause a runtime error and arguably disrupts the logical flow of the program. In a program that supports dependent types, we can construct lists that are both parametrized by a variable (as in the usual polymorphic programming), but also ’indexed’.

The usual definiton of lists would be (in Agda - but easily any functional language)

```
data List (A : Set) : Set where
  nil : List A
  _∷_ : A → List A → List A
```

Compare this with the dependent definition:

```
data Vec (A : Set) : ℕ → Set where
  nil : Vec A 0
  _∷_ : ∀ {n : ℕ} → A → Vec A n → Vec A (n + 1)
```

This allows us to write functions that require a ’proof’ that the two arguments are of equal length.

```
sum : ∀ {n : ℕ} → Vec ℕ n → Vec ℕ n → Vec ℕ n
sum xs ys = ?
```

If this is not obvious from the context, the program will not type check. The developer is forced to only write correct programs.

Further details about agda syntax will be provided in section (Introduction to Agda)

1.4 Nested Types

Another way of maintaing invariants throughout the program is the trick or ’irregular’ or ’nested’ datatypes. They allow forcing strong structural invariants on the datastructure and have gained interest because of their practical implications, allowing definitions of circular datastructures [**?**] or de bruijin indexes [**?**]. Some difficulties come up in recursive calls or inductive proofs, fact which will be covered and discussed in section (TODO implementation/section_nest_example). For example, consider the next data structure, introduced by Bird and Meertens [**?**] , with a slight modification that makes future examples easier to understand.

```
data Nest {a : Level} (A : Set a) : Set a where
  nilN : Nest A
  consN : (A × A) → Nest (A × A) → Nest A
```

This can be thought of as the levels of a full binary tree (with the exeption of the binary tree with only one element, which I am ruling out for simplicity)

Another example, with a more interesting application is:

```
data BinTree (A : Set) : Set where
  empty : BinTree A
  single : A → BinTree A
  deep :  BinTree (Node A) → BinTree A
```

Where Node is simply:

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

It can be seen from the declaration that the structure will be forced to be a sequence of deep constructors, followed by either a single (Node^n A) or an empty constructor. The number of elements stored in it has to be a power of two (2^n) making it equivalent to the leafs of a full binary tree.

This dissertation is mostly concerned with 2-3 trees, the basis for the FingerTrees, which will be studied in detail in the Implementation section. They are an example to show arising problems when proving properties of nested and dependent typed structures, what limits are imposed and how some of them can be overcome.

1.5 Introduction to Agda

Agda is a dependently typed programming language, developed in the spirit of Haskell, kept as simple as possible [**?**]. All the previous examples are written in Agda, and their syntax and the newly introduced syntax will be described as we go on. Along other programming languages like Coq [**?**] or Isabelle [**?**], Agda is used as an interactive (or automatic) theorem prover. What makes it different from the two previously mentioned system is the ability to write the code and the proofs in the same environment. Its relative simplicity also motivated its use in this project.

1.5.1 Types as Values

As said before, a dependently typed environment allows types to be not only arguments to functions, as it is the case in generic data structures, but also returned values.

In Agda, the base for all types is called Set, which can be simplistically though of as the type of types.

I will use, as a running example throughout this section, the construction of natural numbers and lists. They should be sufficient for introducing most of the concepts that will be needed throughout this dissertation.

1.5.2 Declaring Data Structures

Data structures in agda follow the ADT (Algebraic Data Types) paradigm. They group together constructors that can introduce the given structure. Each constructor should be thought of as a function which returns an instance of the data structure.

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

We declare the type of natural numbers, which is of type Set. It has only two constructors; zero, which takes no argument, and suc (successor) which, provided a natural number, can construct the next natural number.

As you probably noticed, Agda has full unicode support. This makes writing proofs about mathematical objects nice and readable since you can use the conventional symbols.

Another piece of elegant syntax is the presence of mixfix operators. Most unicode characters can be used in the name of the operator, and by _ you tell agda that’s where you want to put an argument

```
(Example: _+_, if_then_else_, [_])
```

1.5.3 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.

  $$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
  $$n + m = \{!!\}$$

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

  $$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
  $$zero + m = \{!!\}$$
  $$suc\ n + m = \{!!\}$$

  A consequence of this is that all functions in Agda must be total. If a possible constructor is not present in the definition, it will not type-check.
- 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.

1.5.4 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
  nil : Vec A zero
  _::_ : ∀ {n : 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 ℕ is the type of the value argument for the dependently typed instantiation.

Another think to notice here is the curly brackets ({n : 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.

1.5.5 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.

Chapter 2

Preparation

2.1 Programs as proofs

Agda is a depedently typed programming language, based on the intuitionistic type theory developed by Per Martin Lof [**?**]. The power of the typing system is sufficient to express propositions as types. Finding an inhabitant of each type becomes equivalent to constructing a proof of the embedded proposition.

2.1.1 Some Agda Examples

Consider for example the proposition describing equality (taken from the standard libray)

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

(An explanation of the Level and universe polymorphism will be found at the end of the section – TODO Footnote)

The first thing to notice is that this is a declaration of a dependent data type. It can only be constructed by calling refl, which tells us that all elements are equal to themselves (reflexivity).

Therefore, constructing an elements of type (a ≡ b) for some a and b becomes a proof that a and b are equal. This, of course, relates elements by their structural equality and is the version most used in the standard library. This doesn't stop the developer from defining their own form of equality.

For example, this equality over integers (which is in this case equivalent to ≡)

```
infix 4 _ == _
data _ == _:ℕ → ℕ → Set where
  z : zero ≡ zero
  s : ∀{n m} → n ≡ m → (suc n) ≡ (suc m)
```

We can already start building up proofs, for example, of the property of zero to be the neutre element to addition:

```
0 − left : ∀(n : ℕ) → (zero + n ≡ n)
0 − left zero = z
0 − left (suc n) = s (0 − left n)
```

We can see that the term zero-left is a proof of the proposition embedded in the type, since it shows us how the proposition was constructed (by using z or s constructors respectively)

Consider, for example, defining a less then equal operator and the proof of the antisymmetry property

```
infix 4 _ ≤_
data _ ≤_ :ℕ → ℕ → Set where
  leq − zero : ∀{n : ℕ} → zero ≤ n
  leq − suc : ∀{n : ℕ} {m : ℕ} → m ≤ n → suc m ≤ suc n

leq − antisym : ∀(n : ℕ) (m : ℕ) → (n ≤ m) → (m ≤ n) → (n ≡ m)
leq − antisym zero zero leq − zero leq − zero = z
leq − antisym zero (suc m) leq − zero ()
leq − antisym (suc n) (suc m) (leq − suc p1) (leq − suc p2) = s (leq − antisym n m
```

This example is important because it shows two important points.

Defining relations properly can allow meaningful case split on the proof object. In the first line, we can see how, if n = 0 and m = 0, then both (n ≤ m) and (m ≤ n) can only be constructed by leq-zero. Although this case is trivial, one can imagine a case where knowing exactly what constructor was used provides extra information about the inputs.

Another import point is the absurd pattern (). Agda doesn't allow the definition of partial functions, therefore all the cases of the input must be analyzed. However, if n = 0 and m > 0, one cannot possibly construct the fourth argument to the function (of type m ≤ n). This case, therefore cannot be reached in the program.

During most of the project we will use the structural equality. Agda standard library provides a nice, readable way of constructing such proofs (further down)

2.1.2 Curry-Howard Isomorphism

Although the definitions above seem intuitive, they have strong theoretical underpinnings in Martin Lof's theory [**?**], as well as Curry-Howard isomorphism. The original result by Howard [**?**] gives the link between natural deduction in the intutionistic logic theory and the simply typed lambda calculus.
-insert some table here

2.1.3 with, rewrite

Agda provides some in-built syntax for making writing proofs more intuitive. The first such keyword is with. It allows, inspired by the work of McBride and McKinna [**?**], to pattern match on an intermediate computation. This is more or less equivalent to adding an extra argument to the left of the function, although, that would make the code objectively more unreadable by the never ending chain or arguments in the type declaration.

We will see how this become particularly nice with Views [**?**].

Another piece of usefull syntax is rewrite. This is meant to be read as 'rewrite the left hand side by using the equation on the right hand side'. This makes necessarily use of the builtin equality.

Consider the task of proving the associativity property of natural numbers:

```
open import Relation.Binary.PropositionalEquality

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

The hole here has the type: suc (x + (y + z)) ≡ suc ((x + y) + z). It is therefore suficient to rewrite suc (x + (y + z)) by using the associativity property of x y and z in order to obtain refl.

open **import** *Relation.Binary.PropositionalEquality*

$+ assoc : \forall (x\ y\ z : \mathbb{N}) \rightarrow (x + (y + z)) \equiv ((x + y) + z)$
$+ assoc\ zero\ y\ z = refl$
$+ assoc\ (suc\ x)\ y\ z\ rewrite + assoc\ x\ y\ z = refl$

In fact, rewrite is just syntactic sugar for two nested with statements, as described in the official documentation [**?**]:

```
If eqn : a ≡ b, where _≡_ is the builtin equality you can write
  f ps rewrite eqn = rhs
instead of
  f ps with a | eqn
  … | ._ | refl = rhs
```

-syntax
-unimplemented feature of with – will fit better at difficulties
-rewrite explained as a sequence of with statements
-rewrite example with +-com

2.1.4 Equivalence-Reasoning, inspect

In order to obtain a proof of a more complex nature, you need to chain quite o few rewrite statements, and checking goal types in their presence is not always as illuminating as one would imagine, because it is quite hard to keep track of how the previous rewrites affect the current goal.

This is why the standard library provides a way of chaining such rewriting in a more mathematically friendly manner. The documentation is freely available online [**?**]. To show how it will be used in this project, I will give an example of a proof done in both ways. The proof is the commutativity property of natural numbers.

open **import** *Data.Nat*

This imports the standard library version of natural numbers, which is declared in the exactly same way. It allows us to use actual digits for their representations, which can enhance readability.

Further down are some further proofs we need in order to show natural numbers are commutative.

```
+0 : (m : ℕ) → (m + 0) ≡ m
+0 zero = refl
+0 (suc m) rewrite +0 m = refl

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

Finally, the main proof:

```
+comm : ∀(x y : ℕ) → (x + y) ≡ (y + x)
+comm zero y rewrite +0 y = refl
+comm (suc x) y rewrite +suc x y |
  +suc y x |
  +comm x y = refl
```

```
open ≡-Reasoning
```

This is the module where all the equivalence reasoning primitives are declared. Now, the proof which uses this module.

```
+comm2 : ∀(x y : ℕ) → (x + y) ≡ (y + x)
+comm2 zero y =
    begin
      zero + y
     ≡⟨ sym (+0 y) ⟩
      y + zero
     ∎
+comm2 (suc x) y =
  begin
    suc (x + y)
   ≡⟨ cong suc (+comm2 x y) ⟩
    suc (y + x)
   ≡⟨ sym (+suc y x) ⟩
    y + suc x
   ∎
```

Although it is longer, due to the extra syntax, it allows reading off intermediate results, so that proofs become more readable.
It can also aid the proving process. There are cases when you know it's possible to prove an equivalence from A to B inside of a bigger

proof, but you want to leave that out for the moment and come back to it later. In this case, you can just introduce a hole in the ≡⟨ ! ! ⟩ operator, and continue the main proof, filling in side lemmas at the end.

2.1.5 instance arguments

-type classes in haskell
-the monoid record
-how we use it

2.2 Views

The idea of a view was first introduced by Phil Wadler [**?**] http://www.cs.tufts.edu/ nr/cs257/archive/phil-wadler/views.pdf . Originally it is meant as a way of reconcilling the conflict between data abstraction and pattern matching. The essence of views is representing an arbitrary data type as a newly defined and computationally meaningful algebraic data type.
The cannonical example is 'viewing' a natural number in terms of it's peano representation.
– Example in haskell maybe?

2.2.1 Example - Natural Numberss

A more meaningful example is seeing a natural number as an even or odd number in terms of it's representation. That is, a natural number is even if it can be represented as n = 2 * k for some natural k and it is odd if it can be represented as n = 2 * k + 1. I will present some agda code that does exactly this, providing a functional way of checking if a number is even or odd and also performe floored division.

```
data Repr : ℕ → Set where
  z : Repr 0
  2 ∗_ : ∀ {n : ℕ} → Repr n → Repr (n ∗ 2)
  2 ∗_+ 1 : ∀{n : ℕ} → Repr n → Repr (suc (n ∗ 2))
```

A few helper functions could help one realise how convenient this representation is in case recursive calls over the binary structure of natural number are needed (e.g. when searching in functional array).

```
_ + 1 : ∀{n : ℕ} → Repr n → Repr (suc n)
z + 1 = 2 ∗ z + 1
(2 ∗ m) + 1 = 2 ∗ m + 1
2 ∗ m + 1 + 1 = 2 ∗ (m + 1)
repr : (n : ℕ) → Repr n
repr zero = z
repr (suc n) = (repr n) + 1
```

2.2.2 Induction with views

An important point made [?] is that the design of such views should allow inductive programs to be written on them.

Views become especially usefull when the data type is very hard itself to reason about, such as the case of nested data-type or irregular datatypes.

A simple example of such a view is an implementation of the cyclic list [?].
This is a cannonical example of a great structural need satisfied through typing only.

```
data Clist (A : Set) : Set where
  Var : A → Clist A
  Nil : Clist A
  RCons : ℕ → Clist (Maybe A) → Clist A
```

This is a nested datatype, because each recursive invocation of the Rcons constructor will have type $CList(Maybe^n A)$ where n is the recursion depth. The implementations of the functions that operate on such datatype are presented in the Appendix. One would understand the need to view this data structure as a $A \times CListA$ rather than $A \times Clist(MaybeA)$, as unfolding in the latter way would cause unenecessary nestings of the $Maybe$ constructor.

```
data View (A : Set) : Set where
  NilV : View A
  ConsV : ℕ → Clist A → View A
```

This is an implementation of a function that iterates through the cyclic list for a given depth which could possibly be used in a simulation. As said before, it is a lot more straight-forward to see the data structure in a flattened way, while keeping strong invariants underneath by the nested type.

$$
\begin{aligned}
&\mathit{unwind} : \{A : \mathit{Set}\} \to \mathbb{N} \to \mathit{Clist}\ A \to \mathit{List}\ \mathbb{N}\\
&\mathit{unwind}\ \mathbb{N}.\mathit{zero}\ c = [\,]\\
&\mathit{unwind}\ (\mathbb{N}.\mathit{suc}\ n)\ c\ \mathit{with}\ \mathit{view}\ c\\
&\mathit{unwind}\ (\mathbb{N}.\mathit{suc}\ n)\ c \mid \mathit{NilV} = [\,]\\
&\mathit{unwind}\ (\mathbb{N}.\mathit{suc}\ n)\ c \mid \mathit{ConsV}\ x\ x_1 = x :: \mathit{unwind}\ n\ x_1
\end{aligned}
$$

2.3 Termination checking

Checking whether a program terminates is, by a well-known result, undecidable. Hence, Agda and other theorem provers must rely on heuristics that can rule out all the programs that do not terminate. Due to the undecidability of the halting problem, all such heuristics will also reject programs that are correct. This dissertation was inspired by noticing how certain constructions in Agda, although usefull from an implementation point of view, do not pass this termination check. We will be exploring ways to get arround it.

Since we are interested not only in computing things, but also in checking that they are correct, termination is a mandatory aspect. One can imagine very easily (false) proofs in terms of their type, but with no meaning.

```
--
  bad-proof : forall a b -> P a b
  bad-proof a b = bad-proof a b
--
```

The heuristic used in Agda revolves around the concept of a structurally smaller argument. That is, every recursive call must have a structurally smaller argument (has less layers of constructors wrapped around eachother) in order to pass the termination check.

2.3.1 Example - Regular Data Type

Consider, for example, this simple pathological case:

```
append : ∀{a} {A : Set a} → A → List A → List A
append x xs with xs
append x xs | [] = Data.List ∘ [x]
append x xs | y ∷ ys = y ∷ append x ys
```

One could argue that this fails because ’with’ forgets where ys originally came from, as we have seen before. However, this is not the case:

```
open import Relation.Binary.PropositionalEquality
append2 : ∀{a} {A : Set a} → A → List A → List A
append2 x xs with xs | inspect (λ x → x) xs
append2 x xs | [] | [eq] = Data.List ∘ [x]
append2 x xs | y ∷ ys | Reveal_·_is_ ∘ [refl] = y ∷ (append2 x ys)
```

This is a proof that the ’with’ construct in agda doens’t interact well with Agda’s termination checker. However, Having a structurally recursive dependent typing scheme actually works for this instance - in this case we will use Nat indexing, but this scheme will become more general in the implementation phase. We can extend the List data type and index it on the length, as seen before when we introduced dependent types:

```
open import Data.Vec
append3 : ∀{a} {A : Set a} {n : ℕ} → A → Vec A n → Vec A (suc n)
append3 x xs with xs
append3 x xs | [] = x ∷ []
append3 x xs | y ∷ ys = y ∷ (append3 x ys)
```

2.3.2 Example - Nested Data Type

As stated by Adam Chipala [**?**], there is no deep theoretical reason for which the termination checking should fail in the presence of such nested datatype, other that the termination checker is incomplete (refering to Coq, but is applicable in this context as well). As seen above, a simple solution to it is simply transfigurating the typing to a dependent one.

This is the declaration of Nest, as seen before.

```
data Nest {a : Level} (A : Set a) : Set a where
  nilN : Nest A
  consN : (A × A) → Nest (A × A) → Nest A
```

In this case, I am artificially creating a function with the sole purpose of creating a smaller datatype. It essentially reduces the tree by performing a given operation which takes as argument the pairs present in the leaves of the tree.

```
compact : ∀{a : Level} {A : Set a} → Nest (A × A) → (A × A → A) → Nest A
compact nilN op = nilN
compact (consN p ns) op = consN (lift − op op p) (compact ns (lift − op op))
```

The lift operation is just a map operation on the Product type:

```
lift − op : ∀{a : Level} {A : Set a} → (A × A → A) → ((A × A) × (A × A)) → (A × A)
lift − op op (proj₁, proj₂) = (op proj₁, op proj₂)
```

We introduce, as we did in the case of CList, a flattened view, together with a function that can convert between the two:

```
data View {a : Level} (A : Set a) : Set a where
  nilL : View A
  consL : A → Nest A → View A

view : {A : Set} → Clist A → View A
view (Var x) = NilV
view Nil = NilV
view (RCons x cl) = ConsV x (csnoc x cl)
```

It is not hard to infer, given the previous examples, that such a declaration would fail to pass the termination checker in case there is a recursive call. Therefore, we are modifying the code the same way we did with List, by declaring an index

```
data dNest {a : Level} (A : Set a) : ℕ → Set a where
  dnilN : dNest A zero
  dconsN : ∀{n : ℕ} → (A × A) → dNest (A × A) n → dNest A (suc n)

data dView {a : Level} (A : Set a) : ℕ → Set a where
  dnilV : dView A zero
  dconsV : ∀{n : ℕ} → A → dNest A n → dView A (suc n)
```

However, in the presence of a nested type, the structural recursive indexing fails to satisfy the type-checker.

2.4 Well-Founded relations and Induction

A well-founded relation is a relation R, ($R \subset X \times X$) such that there is no infinite sequence x0, x1, x2, ... of elements of X such that xn+1 R xn. A less than relation with this property can be used to implement a terminating induction definition, since we know that we cannot ever have an infinite sequence of decreasing argument sizes in the call stack. Tools for proving that a relation is Well-Founded and helper functions for defining recursive definitions are present in the standard library.

Chapter 3

Implementation

Data structure design in functional programming has been greately improved by the publications of Okasaki [**?**] and those of Ross Patterson and Ralph Hinze. These papers introduce a number of concepts which has made possible a great efficiency speed-up in functional algorithms. The observation that the construction of numbers is equivalent to the construction of containers that hold that number of elements [**?**] [**?**] has also provided developers with a general way of designing such constructors. However, implementing such data structures and proving correctness at the same time introduce some difficulties

In this section, I will present a dependently typed implementation of the Finger Tree[**?**] and prove correctness of some methods. I will then instantiate the data structure to allow more specific uses. Next, I will point out the issues caused by the incomplete termination checker and provide a solution. Finally, I will bring Finger Trees into a larger context and show how these problems are not specific.

3.1 Finger Trees - Introduction

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

Queues by simple instantiation.

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

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

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

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

Moreover, the measurement as a type index greatly simplifies proofs when the measurement function is chosen carefully.

3.2 Previous work on Finger Trees

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

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

[1] •

[2] •

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

- Implementation in Isabelle.
  Another working implementation has been done in Isabelle. However, this implementation diverges from the original specification of the data structure, removing the nesting. The two invariants that I have mentioned are maintained explicitly. The paper follows an external verification paradigm, whereas I aimed for an internally verified one.
  The implementation of this data structure in both Coq and Isabelle, two established theorem provers might argue both for the complexity involved, and for its interesting particularities.

3.3 Finger Trees - Implementation

This is the main section, where implementation of key points is presented and discussed[3]

3.3.1 Data type declaration

The Finger Tree is originally polymorphic in two types:

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

- **V** : this is the type of the measures of the elements.

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

[3]For a full implementation refer to the Appendix

- **Monoid**[4] **V**: which contains a neutral element(**ε**), a binary operator(**·**), and the monoid axioms, and a comparison operator.
- **Measured A V** : which consists of a norm function : $\|\| : A \to V$ I am also constraining this measurement function, so that $\forall x \in A, \|x\| \geqslant \varepsilon$. This is to further ensure that, for some Finger Tree ft, if $\|ft\| \geqslant \varepsilon$, then $ft \neq Empty$ In the section 3.whatever I will show how this can be abstracted away from the user.

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

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

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

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

Finally, the **FingerTree** is a family of types, indexed by a measurement μ. The measurement's correctness is enforced in all the constructors. Note the nested type and the universal quantification over possible sizes for the recursive call. Apart from the measurement addition, the rest corresponds to the original paper.

```
data FingerTree {a} (A : Set a) (V : Set a)
   ⦃ mo : Monoid V ⦄ ⦃m : Measured A V ⦄ :{μ : V} → Set a where
  Empty : FingerTree A V {ε}
  Single : (e : A) → FingerTree A V { ‖e‖ }
```

[4]see AlgebraStructures.agda

```
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}
```

3.3.2 *Cons* and *Snoc*

The dependent typing allows a fine intertwining between coding and proofs, and this is an example where Agda's simplicity is best seen.

Implementing a cons(◁)[5] operator is also accompanied by a proof that the result's measure will be equal to the sum of the argument's measure.

I will give the type declaration and some necessary axioms[6]

```
infixr 5 _◁_
_◁_ : ∀ {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}
```

As in the Vec example, the type is indicative of what the function is doing, and a function of this signature ensures that the output will have the correct measure. We can further prove that it also contains the expected elements, and I will do this in the Evaluation as an external verification procedure.

By performing a case split on the FingerTree, and further on the prefix Digit (in the Deep constructor), we have to implement the function for six cases. I will show the implementation of a simple case and that of the recursive case in detail.

– cons-deep-one

```
a ◁ Deep (One b) ft sf rewrite
   • − assoc (‖a‖) (‖b‖) (measure − tree ft • measure − digit sf)
     = Deep (Two a b) ft sf
```

The result of this computation is identical to the one in the original paper. However, in order for it to type check, I had to include a proof of correctness, in the form of a rewrite statement, by using the associativity property of the monoid.

[5]appends an element to the left of the Finger Tree
[6]the full implementation is in the Appendix

– cons–deep-four

```
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
```

This case is the recursive case, and the proof in this case turned out to be more verbose.

```
assoc − lemma2 : ∀{a} {A : Set a} {V : Set a}
   ⦃ mo : Monoid V ⦄ ⦃m : Measured A V ⦄ →
   (a : A) → (b : A) → (c : A) → (d : A) → (e : A) → (s : V) → (f : V) →
      (mo Monoid.• Measured.‖m ‖ a)
         ((mo Monoid.•
               (mo Monoid.• Measured.‖m ‖ b)
               ((mo Monoid.• Measured.‖m ‖ c)
         ((mo Monoid.• Measured.‖m ‖ d) (Measured.‖m ‖ e))))
         ((mo Monoid.• s) f))
         ≡
      (mo Monoid.• (mo Monoid.• Measured.‖m ‖ a) (Measured.‖m ‖ b))
         ((mo Monoid.•
            (mo Monoid.•
               (mo Monoid.• Measured.‖m ‖ c)
               ((mo Monoid.• Measured.‖m ‖ d) (Measured.‖m ‖ e)))
            s)
         f)
```

A lot of the textual complexity can be reduced by using infix notation. The reason I have left it like this is because the type signature has been generated by Agda. If left without the rewrite statement, Agda prompts us with an error, which is also what remains to be proved.

The **Snoc** operator has the same structure, but it appends the new element to the right. Its type is analogous.

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

3.3.3 Deconstructing sequences – Views

Our goal is to extract the first element of the Finger Tree (from the Left here), while also returning a correct tail. This is equivalent to defining a convenient View[**?**] on the tree. Some extra work has to be done in order to maintain the invariant of the measurement. Specifically, the Finger Tree and its view should have the same measure.

$$\begin{array}{l}\textbf{data}\ \mathit{ViewL}\ \{a\}\ (A : \mathit{Set}\ a)\ (V : \mathit{Set}\ a)\ \{\!\{\ \mathit{mo} : \mathit{Monoid}\ V\ \}\!\}\ \{\!\{\mathit{m} : \mathit{Measured}\ A\ V\ \}\!\} :\\ \quad \{s : V\} \to \mathit{Set}\ a\ \textbf{where}\\ \quad \mathit{NilL} : \mathit{ViewL}\ A\ V\ \{\varepsilon\}\\ \quad \mathit{ConsL} : \forall\{z : V\}\ (x : A) \to (\mathit{xs} : \mathit{FingerTree}\ A\ V\ \{z\}) \to\\ \qquad\qquad \mathit{ViewL}\ A\ V\ \{\,\|x\,\|\ \bullet z\}\end{array}$$

The ConsL constructor is similar to the Cons operator declared before, and this similarity can be seen in the measurement. Ensuring the the finger tree and it view will have the same measurement is done with the correct instatiation of the viewL function:

$$\begin{array}{l}\mathit{viewL} : \forall\{a\}\ \{A : \mathit{Set}\ a\}\ \{V : \mathit{Set}\ a\}\ \{\!\{\ \mathit{mo} : \mathit{Monoid}\ V\ \}\!\}\ \{\!\{\mathit{m} : \mathit{Measured}\ A\ V\ \}\!\}\\ \quad \{i : V\} \to \mathit{FingerTree}\ A\ V\ \{i\} \to \mathit{ViewL}\ A\ V\ \{i\}\end{array}$$

The cases where the prefix has length greater than one are trivial. Deconstructing the tree is just a matter of returning the head of the Digit and calling the Deep constructor with the tail of the digit. However, for the case when the tail is empty, a special constructor needs to be implemented.

$$\begin{array}{l}\mathit{deepL} : \forall\{a\}\ \{A : \mathit{Set}\ a\}\ \{V : \mathit{Set}\ a\}\ \{\!\{\ \mathit{mo} : \mathit{Monoid}\ V\ \}\!\}\ \{\!\{\mathit{m} : \mathit{Measured}\ A\ V\ \}\!\}\\ \quad \{s : V\} \to\\ \quad (\mathit{pr} : \mathit{Maybe}\ (\mathit{Digit}\ A)) \to\\ \quad (\mathit{ft} : \mathit{FingerTree}\ (\mathit{Node}\ A\ V)\ V\ \{s\}) \to\\ \quad (\mathit{sf} : \mathit{Digit}\ A) \to\\ \quad \mathit{FingerTree}\ A\ V\ \{\mathit{measure} - \mathit{maybe} - \mathit{digit}\ \mathit{pr} \bullet s \bullet \mathit{measure} - \mathit{digit}\ \mathit{sf}\}\end{array}$$

3.3.4 Splitting

The **Split** data type is a more generalized version of the previously declared ViewL. In this case the Finger Tree is deconstructed somewhere in the middle, and packed in a container that keeps the left sequence, the middle element and the right sequence.

We need to make sure to preserve the measurement of this data type. As before, we will continue to use V-indexing.

One can see from this declaration, together with the constraint on the norm operator, that there cannot be a split of size ε

```
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}) →
    (x : A) →
    (FingerTree A V {μ₂}) →
    Split − d A V {μ₁ • ‖x ‖ •μ₂}
```

The splitting function is supplied with a boolean predicate **p** and a starting value **i**, which indicate where the split occurs.

We begin to iterate through the finger tree, by cummulating the norm of elements as we go along. The procedure continues until the accumulated value turns the predicate true, in which case the element we needed has been found.

Although it seems to be an expensive operation, the cached measurements, together with the associativity property, make this operation logarithmic. We do not need to go through all the leaves in a **Digit** or **Node** to find what their cummulated measure is.

As an example, we can find the ViewL construction by setting $p(x) = true$ for all x.

Next, I will present the split operation which is a bit lengthy. The implementation could have been written more concisely by using the 'with' construction. However, a small shortcoming of Agda's type-checker in its presence can cause some important information to be lost on the way. This makes the measurement correctness unprovable, and thus the whole split function, since one requires the other.

I have therefore created a new function for each case and passed the 'current information' as Equivalence relations in the argument, similar to the **Node** constructors.

```
split − Tree : ∀{a} {A : Set a} {V : Set a}
   ⦃ mo : Monoid V ⦄ ⦃m : Measured A V ⦄
     -- type class information
  {μ : V} →
  (p : V → Bool) → (i : V) →
```

```
        -- predicate and inital value
      (ft : FingerTree A V {μ}) → Maybe (Split − d A V {μ})
        -- argument and proof that the split has the same size
    split − Tree p i Empty = nothing   -- cannot split an empty tree
    split − Tree ⦃ mo ⦄ p i (Single e) = just (split − Tree − single p i e)
    split − Tree p i (Deep pr ft sf) = just (split − Tree − if p i pr ft sf vpr refl vft refl)
      where
        vpr = p (i • (measure − digit pr))
        vft = p ((i • measure − digit pr) • measure − tree ft)
```

The interesting case here is the Deep one. The **split-tree-if** function does the case analysis for the cummulated measurements. We need to pass it the what would be cummulated at the end of the prefix (**vpr**), as well as what is cummulated and the beginning of the suffix (**vft**). We also need to show the called method that their values represent what we intended.

```
    split − Tree − if : ∀{a} {A : Set a} {V : Set a}
       ⦃ mo : Monoid V ⦄ ⦃m : Measured A V ⦄
        -- type class information
      {μ : V} →
      (p : V → Bool) → (i : V) →
        -- predicate and initial value
      (pr : Digit A) →
      (ft : FingerTree (Node A V) V {μ}) →
      (sf : Digit A) →
        -- flattened deep constructor
      (vpr : Bool) → (vpr ≡ p (i • measure − digit pr)) →
        -- passing the cummulated value at the prefix + proof of not cheating
      (vft : Bool) → (vft ≡ p ((i • measure − digit pr) • (measure − tree ft))) →
        -- passing the cummulated value at the suffix + proof of not cheating
      Split − d A V {(measure − digit pr) • μ • (measure − digit sf)}
        -- giving back the correct-sized split
    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 it doesn't become true at al
    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

I have numbered the cases in the same way they were numbered in the original paper.

– SHOULD I GO THROUGH EACH CASE?

3.4 Numerical Representations – Move these guys at the end of implementation – since they are extra stuff anyway

The treatment of containers as natural numbers has been studied in depth[**?**]. 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.

3.5 Random Access Sequences – move this lower

In this section, I will present a data structure as implemented by Ralph Hinze[**?**] 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:

```
module bush where
  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 [**?**].

```
module ral where
  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.

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

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 implemending 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)
```

We can finally declare a sequence, by using the definition of Leaf as a layer of abstraction.

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

3.5.1 Example Method

Here is the implementations of a method that illustrates the sequences' use.

```
open import Data.List

fromList : {A : Set} → List A → IxSequence A
fromList [] = Nil
fromList (x ∷ xs) = cons x (fromList xs)
```

3.5.2 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.

```
open import Data.Product

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 Finger Tree 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)
```

3.5.3 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)
```

3.5.4 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.

```
module ral − sized where

  open import Size
  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


  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}
  IxSequence : Set → {i : Size} → Set
  IxSequence = RandomAccessList Leaf
  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 b₂ 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

$$B' = FORKB$$
$$A' = A$$

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. We will implement a custom-made size function by enhancing the type with a measurement, in the context of Finger Trees.

3.6 FingerTrees in general

-what are they
-previous work: isabelle, coq, agda.

3.7 FingerTrees in Agda

-without measurement - quite boring
-with measurment
-view from the left is not working because of termination check (pointer to coq paper)
-trying to solve the issue by defining a well-founded induction
-use the measurement info, since it's already there

3.8 Random Access Sequences as an easier example

-size and entry
-trying to prove the well-foundedness
-boom, we fail because we didn't follow a dependently typed implementation

3.9 Dependently typed FingerTrees

- implementing dependently typed fingertrees ensures that there cannot be malformed trees in the program. - I will use the measurement V for the index, since it's mostly been implemented in the previous section.

- defining a well founded induction on V will allow ordering trees as well, an therefore ensure termination checking on this recursion.

- (all is well until node)

- A full dependent implemenatation seemed harder to envision. By full, I mean that I move the measure labels from the nodes to the type as well. However, I don't see how we allow trees to have differently sized nodes since their constructor only takes an instance of Node A V size, size which must be specified in advanced.

- I am cheating this using postulates. We can guarantee no bad nodes are added in the tree since the user never actually needs to call the constructor of the node, it should be a completely hidden concept.

- The approach with the view doesn't work as expected, even when Agda can see that the types are becoming smaller. I am providing the examples in node2.agda

- Complain about the with operator

- All this to simplify recursion and inductive definitions on FingerTrees.

3.10 Random Access Sequences with dependently typed FingerTrees

3.11 Verbatim text

Verbatim text can be included using `\begin{verbatim}` and `\end{verbatim}`. I normally use a slightly smaller font and often squeeze the lines a little closer together, as in:

```
GET "libhdr"

GLOBAL { count:200; all  }

LET try(ld, row, rd) BE TEST row=all
                        THEN count := count + 1
                        ELSE { LET poss = all & ~(ld | row | rd)
                               UNTIL poss=0 DO
                               { LET p = poss & -poss
                                 poss := poss - p
                                 try(ld+p << 1, row+p, rd+p >> 1)
                               }
                             }
LET start() = VALOF
{ all := 1
  FOR i = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("Number of solutions to %i2-queens is %i5*n", i, count)
    all := 2*all + 1
  }
  RESULTIS 0
}
```

3.12 Tables

Here is a simple example[7] of a table.

Left Justified	Centred	Right Justified
First	A	XXX
Second	AA	XX
Last	AAA	X

There is another example table in the proforma.

[7] A footnote

3.13 Simple diagrams

Chapter 4

Evaluation

4.1 Printing and binding

If you have access to a laser printer that can print on two sides, you can use it to print two copies of your dissertation and then get them bound by the Computer Laboratory Bookshop. Otherwise, print your dissertation single sided and get the Bookshop to copy and bind it double sided.

Better printing quality can sometimes be obtained by giving the Bookshop an MSDOS 1.44 Mbyte 3.5” floppy disc containing the Postscript form of your dissertation. If the file is too large a compressed version with `zip` but not `gnuzip` nor `compress` is acceptable. However they prefer the uncompressed form if possible. From my experience I do not recommend this method.

4.1.1 Things to note

- Ensure that there are the correct number of blank pages inserted so that each double sided page has a front and a back. So, for example, the title page must be followed by an absolutely blank page (not even a page number).

- Submitted postscript introduces more potential problems. Therefore you must either allow two iterations of the binding process (once in a digital form, falling back to a second, paper, submission if necessary) or submit both paper and electronic versions.

- There may be unexpected problems with fonts.

4.2 Further information

See the Computer Lab’s world wide web pages at URL:
`http://www.cl.cam.ac.uk/TeXdoc/TeXdocs.html`

Chapter 5

Conclusion

I hope that this rough guide to writing a dissertation is LaTeX has been helpful and saved you time.