Razvan Kusztos

Verified functional datastructures in Agda

Diploma in Computer Science

Girton College

January 21, 2017

Proforma

Name:	**Razvan Kusztos**
College:	**Girton College**
Project Title:	**Verified functional datasturcures and algo**
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 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.

1.2 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)

1.3 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. [**?**] 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 in Bird and Meertens [**?**] as well, but slightly modified so I can build a pathological example on top of it later on:

```
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.

1.4 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.

1.4.1 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.

1.4.2 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_, [_])
```

1.4.3 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.

1.4.4 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.

1.4.5 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 Foodnote)

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 p
```

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 can see how this will become particularly nice with Views [**?**].
-example needed

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 : ∀(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
```

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.

-show the syntax mostly
-reason behind inspect

2.1.5 instance arguments

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

2.2 Views

-introduction via wadler http://www.cs.tufts.edu/ nr/cs257/archive/phil-wadler/views.pdf
-emphasise the point in section 10 induction
-give nat example
-show use in nested types

2.3 Termination checking

-correctness requires termination checking
-how termination checking works in agda - 'structurally smaller'
-how views with nested types fail

2.4 Well-foundedness

-what is well-founded induction
-tb to larry paulson's paper: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10
-agda's standard library provides the tools

Chapter 3

Implementation

3.1 FingerTrees in general

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

3.2 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 get solve the issue by defining a well-founded induction
-use the measurement info, since it’s there

3.3 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

3.4 Dependently typed FingerTrees

3.5 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
}
```

3.6 Tables

Here is a simple example[1] 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.

[1] A footnote

3.7 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.