Software Foundations

December 16, 2013

Preface

0.1 Welcome

This electronic book is a Agda version of Pierce's Coq book named Software Foundations.

0.2 Overview

...

Chapter 1

Basic Functional Programming in Agda

1.1 Introduction

The functional programming style brings programming closer to mathematics: If a procedure or method has no side effects, then pretty much all you need to understand about it is how it maps inputs to outputs — that is, you can think of its behavior as just computing a mathematical function. This is one reason for the word "functional" in "functional programming." This direct connection between programs and simple mathematical objects supports both sound informal reasoning and formal proofs of correctness.

The other sense in which functional programming is "functional" is that it emphasizes the use of functions (or methods) as *first-class values* — i.e., values that can be passed as arguments to other functions, returned as results, stored in data structures, etc. The recognition that functions can be treated as data in this way enables a host of useful idioms, as we will see. Other common features of functional languages include *algebraic data types* and *pattern matching*, which make it easy to construct and manipulate rich data structures, and sophisticated *polymorphic type systems* that support abstraction and code reuse. Agda shares all of these features.

1.2 Enumerated Types

One unusual aspect of Coq is that its set of built-in features is extremely small. For example, instead of providing the usual palette of atomic data types (booleans, integers, strings, etc.), Agda offers an extremely powerful mechanism for defining new data types from scratch — so powerful that all these familiar types arise as instances.

Agda has a standard library that comes with definitions of booleans, numbers, and many common data structures like lists. But there is nothing magic or primitive about these library definitions: they are ordinary user code. To see how this works, let's start with a very simple example.

1.2.1 Days of Week

The following declaration tells Agda that we are defining a new set of data values — a type.

```
data Day : Set where
  Monday    : Day
  Tuesday   : Day
  Wednesday : Day
  Thursday  : Day
  Friday    : Day
  Saturday  : Day
  Sunday    : Day
```

The type is called Day, and its members are Monday, Tuesday, etc. The second through eighth lines of the definition can be read "monday is a day, tuesday is a day, etc."

Having defined Day, we can write functions that operate on days.

```
nextDay : Day → Day
nextDay Monday    = Tuesday
nextDay Tuesday   = Wednesday
nextDay Wednesday = Thursday
nextDay Thursday  = Friday
nextDay Friday    = Saturday
nextDay Saturday  = Sunday
nextDay Sunday    = Monday
```

One thing to note is that the argument and return types of this function are explicitly declared. Like most functional programming languages, Agda can often work out these types even if they are not given explicitly — i.e., it performs some type inference — but we'll always include them to make reading easier.

Having defined a function, we should check that it works on some examples. There are actually two different ways to do this in Agda. One uses the notion of equality and another uses the interactive emacs mode. To test the new defined nextDay function, just type C-c C-n and type nextDay *Friday* on the emacs buffer in order to Agda evaluate the expression to *Saturday*.

1.2.2 Booleans

In a similar way, we can define the type *Bool* of booleans, with members true and false.

```
data Bool : Set where
  True  : Bool
  False : Bool
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Agda does, of course, provide a default implementation of the booleans in its standard library, together with a multitude of useful functions and lemmas. Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
¬ : Bool → Bool
¬ True  = False
¬ False = True

and : Bool → Bool → Bool
and True t  = t
and False _ = False
```

```
or : Bool → Bool → Bool
or True _ = True
or False t = t
```

Agda supports the so called mixfix operator syntax and unicode identifiers that are extensively used in the standard library. We believe that using these only serves as an additional barrier for newbies learning Agda. So, we will avoid it.

Again, we can test these definitions using the emacs mode through the command C-c C-n.

Exercise 1.1 (The nand logic operator)**.** *Define the following function to represent the nand logic operator. Nand connective is defined using the following truth table:*

A	B	nand A B
False	*False*	*True*
False	*True*	*True*
True	*False*	*True*
True	*True*	*False*

```
nand : Bool → Bool → Bool
nand a b = ?
```

Exercise 1.2 (The and3 function)**.** *Implement the* and3 *function that returns the conjunction of 3 boolean values.*

```
and3 : Bool → Bool → Bool → Bool
and3 a b c = ?
```

1.2.3 Function Types

We can use the emacs interactive mode to deduce an expression types. Just type C-c C-d and enter a expression in the emacs buffer to Agda give this expression type.

As an example, entering the expression and *True*, Agda will return the type *Bool* → *Bool*. Entering ¬, will also return *Bool* → *Bool*.

1.2.4 Numbers

The types we have defined so far are examples of "enumerated types": their definitions explicitly enumerate a finite set of elements. A more interesting way of defining a type is to give a collection of "inductive rules" describing its elements. For example, we can define the natural numbers as follows:

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

The clauses of this definition can be read:

- zero is a natural number.
- suc is a "constructor" that takes a natural number and yields another one — that is, if n is a natural number, then suc n is too.

Agda compiler provides some pragmas to enable numeric literals, in order to avoid the verbose notation of n-aries sucs.

```
{-# BUILTIN NATURAL Nat #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

Let's look at this in a little more detail.

Every inductively defined set (Day, Nat, Bool, etc.) is actually a set of expressions. The definition of *Nat* says how expressions in the set *Nat* can be constructed:

- the expression zero belongs to the set *Nat*;
- if n is an expression belonging to the set *Nat*, then suc n is also an expression belonging to the set *Nat*; and expressions formed in these two ways are the only ones belonging to the set *Nat*.

The same rules apply for our definitions of *Day* and *Bool*. The annotations we used for their constructors are analogous to the one for the zero constructor, and indicate that each of those constructors doesn't take any arguments.

These three conditions are the precise force of the inductive declaration. They imply that the expression zero, the expression suc zero, the expression suc (suc zero), the expression suc (suc (suc zero)), and so on all belong to the set *Nat*, while other expressions like *True*, and *True False*, and suc (suc *False*) do not.

We can write simple functions that pattern match on natural numbers just as we did above — for example, the predecessor function:

```
pred : Nat → Nat
pred zero = zero
pred (suc n) = n
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference: functions like pred and minustwo come with computation rules — e.g., the definition of pred says that pred 2 can be simplified to 1 — while the definition of suc has no such behavior attached. Although it is like a function in the sense that it can be applied to an argument, it does not do anything at all!

For most function definitions over numbers, pure pattern matching is not enough: we also need recursion. For example, to check that a number n is even, we may need to recursively check whether $n - 2$ is even.

```
evenb : Nat → Bool
evenb zero = True
evenb (suc zero) = False
evenb (suc (suc n)) = evenb n
```

We can define oddb, a function that tests if a natural number is odd, similarly or using evenb.

Exercise 1.3. *Defining* oddb *Define the function* oddb *that tests if a number is odd, without recursion.*

```
oddb : Nat → Bool
oddb n = ?
```

Naturally, we can also define multi-argument functions by recursion.

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

Adding three to two now gives us five, as we'd expect — You can test it using C-c C-n, as you know.

The simplification that Coq performs to reach this conclusion can be visualized as follows:

```
suc (suc (suc zero)) + suc (suc zero)        ⇒
suc (suc zero) + suc (suc (suc zero))        ⇒
suc zero + suc (suc (suc (suc zero)))        ⇒
zero + suc (suc (suc (suc (suc zero))))      ⇒
suc (suc (suc (suc (suc zero))))
```

Multiplication and subtraction over naturals are defined straightforwardly, by pattern matching. The underscore represents a *wildcard* pattern. Writing underscores in a pattern is the same as writing some variable that doesn't get used on the right-hand side. This avoids the need to invent a bogus variable name.

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

_ − _ : Nat → Nat → Nat
zero − _ = zero
(suc n) − zero = suc n
suc n − suc m = n − m
```

Exercise 1.4 (factorial function)**.** *Define a function to compute the factorial of a given natural number.*

```
factorial : Nat → Nat
factorial n = ?
```

When we say that Agda comes with nothing built-in, we really mean it: even equality testing for numbers is a user-defined operation! The beq_nat function tests natural numbers for equality, yielding a boolean.

```
beqNat : Nat → Nat → Bool
beqNat zero zero        = True
beqNat zero (suc _)     = False
beqNat (suc _) zero     = False
beqNat (suc n) (suc m)  = beqNat n m
```

Similarly, the ble_nat function tests natural numbers for less-or-equal, yielding a boolean.

```
bleNat : Nat → Nat → Bool
bleNat zero _           = True
bleNat (suc n) zero     = False
bleNat (suc n) (suc m)  = bleNat n m
```

Exercise 1.5 (Definition of blt_nat)**.** *The* blt_nat *function tests natural numbers for less-than, yielding a boolean.*

```
bltNat : Nat → Nat → Bool
bltNat n m = ?
```

1.3 Proof by Simplification

Little digression: In type theory based proof assistants, like Agda and Coq, there are (at least) two notions of equality: the definitional equality and the propositional equality. I need to put here some explanation about propositional equality. **End of little digression**.

Now that we've defined a few datatypes and functions, let's turn to the question of how to state and prove properties of their behavior. Actually, in a sense, we've already started doing this: each time we use C-c C-n in the previous sections makes a precise claim about the behavior of some function on some particular inputs. The proofs of these claims were always the same: use refl — that has type x ≡ x, for each x — to check that both sides of the = simplify to identical values.

The same sort of "proof by simplification" can be used to prove more interesting properties as well. For example, the fact that 0 is a "neutral element" for + on the left can be proved just by observing that 0 + n reduces to n no matter what n is, a fact that can be read directly off the definition of +.

```
lemmaPlus0N : forall (n : Nat) → 0 + n ≡ n
lemmaPlus0N n = refl
```

Agda emacs mode uses some unicode notation for symbols. More information about unicode notation can be found at Agda wiki.

Agda follows the so-called *Curry-Howard isomorphism* where types denote logical formulas and terms proofs. So, a proof of a theorem just correspond to a function whose type denotes the formula being proved by the term defining the function.

Note that we've added the quantifier **forall** (n : *Nat*), so that our theorem talks about all natural numbers n. In order to prove theorems of this form, we need to to be able to reason by assuming the existence of an arbitrary natural number n. This is achieved in the proof by considering the quantified variable n as a function parameter, putting it on the context as an hypothesis. With this, we can start the proof by saying "OK, suppose n is some arbitrary number."

Agda does not have tactics like Coq, so in order to prove theorems we need to fully construct terms with the type of the theorem being proved. The next definitions are simple examples of proofs:

```
lemmaPlus1L : forall (n : Nat) → 1 + n ≡ suc n
lemmaPlus1L n = refl
lemmaMult0L : forall (n : Nat) → 0 * n ≡ 0
lemmaMult0L n = refl
```

1.4 Proof by Rewriting

Here is a slightly more interesting theorem:

```
plusIdExample : forall (n m : Nat) → n ≡ m → n + n ≡ m + m
```

Instead of making a completely universal claim about all numbers n and m, this theorem talks about a more specialized property that only holds when n ≡ m. The arrow symbol is pronounced "implies." As before, we need to be able to reason by assuming the existence of some numbers n and m. We also need to assume the hypothesis n ≡ m.

Before we show the real proof of this little theorem, we need to learn how to use Agda emacs mode in order to interactively construct proofs. When building a term, we can use holes, ?, as parts of a term that will need to be filled with a type correct term in order to finish the definition. Loading the file with $[C - c\ C - l]$, we find that Agda checks the unfinished program, turning the ? into labelled braces,

```
plusIdExample : forall (n m : Nat) → n ≡ m → n + n ≡ m + m
plusIdExample n m = { }0
```

and tells us, in the information window,

```
?0 : n ≡ m → n + n ≡ m + m
```

that the type of the 'hole' corresponds to the return type we wrote. In order to prove this, we need to put the equality n ≡ m in context to use it as an hypothesis. To this we can add another parameter to plusIdExample to represent the hypothesis of type n ≡ m. This can be expessed as:

```
plusIdExample : forall (n m : Nat) → n ≡ m → n + n ≡ m + m
plusIdExample n m prf = { }0
```

leaving the following hole:

```
?0 : n + n ≡ m + m
```

To finish this proof we need to use the equality constructor refl, but the hole doesn't have the shape x ≡ x. In order to make the hole type fit the refl type, we need to *pattern match* on the proof that n ≡ m. When the pattern matches occurs, the equality n ≡ m is rewritten in the hole type, making it equal to m + m ≡ m + m that matches refl type.

Agda mode can automate the generatiion of total pattern matching in definitions. Putting the variable on which we want to pattern match on the hole and pressing C-c C-c.

In this proof, we want to pattern match on the equality n ≡ m, so we put the variable prf on the hole and trigger Agda mode case spliting:

```
plusIdExample : forall (n m : Nat) → n ≡ m → n + n ≡ m + m
plusIdExample n m prf = {!prf!}
```

Emacs will change the definition of plusIdExample to:

```
plusIdExample : forall (n m : Nat) → n ≡ m → n + n ≡ m + m
plusIdExample .m m refl = {!!}
```

in which the hole has the following type

```
?0 : m + m ≡ m + m
```

that matches refl type. A important part of this definition is that the left-hand side of the definition has what we call a dotted pattern. Dotted patterns specify equality constraints to be used by the type checker of Agda, when verifying a term. In this example, the dotted pattern is used to specify that the first and the second argument of plusIdExample must be the same, in order to refl be valid.

To finish the proof, we can just type refl in the hole and press C-c C-g or use Agda mode proof search search, by pressing C-c C-a.

```
plusIdExample : forall (n m : Nat) → n ≡ m → n + n ≡ m + m
plusIdExample .m m refl = refl
```

Exercise 1.6 (A simple equality proof)**.** *Prove the following simple equality:*

```
plusIdExercice : forall (n m o : Nat) → n ≡ m → m ≡ o → n + m ≡ m + o
plusIdExercice = { }0
```

Holes can be used whenever we want to skip trying to prove a theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary facts that we believe will be useful for making some larger argument, use holes to accept them on faith for the moment, and continue thinking about the larger argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, when we leave unfinished terms we leave a door open for total nonsense to enter Agda's nice, rigorous, formally checked world!

As another example of equality proof, consider the following simple code piece:

```
mult0Plus : forall (n m : Nat) → (0 + n) ∗ m ≡ n ∗ m
mult0Plus n m = refl
```

Agda is able to determine that (0 + n) ∗ m is definitionally equal to n ∗ m, so we can just prove mult0Plus using refl.

1.5 Proof by Case Analysis

Of course, not everything can be proved by simple calculation: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can block the calculation. For example, if we try to prove the following fact using reduction as above, we get stuck.

```
beqNatN + 1 = 0 : forall (n : Nat) → beqNat (n + 1) 0 ≡ False
beqNatN + 1 = 0 n = { }0
```

The reason for this is that the definitions of both beqNat and + begin by performing a match on their first argument. But here, the first argument to + is the unknown number n and the argument to beqNat is the compound expression n + 1; neither can be simplified.

What we need is to be able to consider the possible forms of n separately. If n is zero, then we can calculate the final result of beqNat (n + 1) 0 and check that it is, indeed, false. And if n ≡ suc n′ for some n′, then, although we don't know exactly what number n + 1 yields, we can calculate that, at least, it will begin with one suc, and this is enough to calculate that, again, beqNat (n + 1) 0 will yield *False*.

To consider, separately, the cases where n ≡ 0 and where n ≡ *S* n′ we can just pattern match on n, getting:

```
beqNatN + 1 = 0 : forall (n : Nat) → beqNat (n + 1) 0 ≡ False
beqNatN + 1 = 0 zero = refl
beqNatN + 1 = 0 (suc n) = refl
```

Proofs by case analysis (pattern matching) works for any data type, like *Bool*:

```
notInvolutive : forall (b : Bool) → ¬ (¬ b) ≡ b
notInvolutive False = refl
notInvolutive True = refl
```

Exercise 1.7 (Proof by case analysis)**.** *Prove the following simple fact:*

```
zeroNBeq + 1 : forall (n : Nat) → beqNat 0 (n + 1) ≡ False
zeroNBeq + 1 n = { }0
```

1.6 More Exercices

Use what you have learned so far to prove the following theorems.

Exercise 1.8. *Prove the following:*

```
identityFnAppliedTwice : forall (f : Bool → Bool)
  (forall (b : Bool) → f x ≡ x) → forall (b : Bool) → f (f b) ≡ b
identityFnAppliedTwice = { }0
```

Now state and prove a theorem negationFnAppliedTwice *similar to the previous one but where the second hypothesis says that the function f has the property that f x = not x.*

Exercise 1.9. *Prove the following lemma. (You may want to first prove a subsidiary lemma or two.)*

```
lemma : forall (b c : Bool) → (and b c ≡ or b c) → b ≡ c
lemma = { }0
```

Exercise 1.10 (Binary Numbers)**.** *Consider a different, more efficient representation of natural numbers using a binary rather than unary system. That is, instead of saying that each natural number is either zero or the successor of a natural number, we can say that each binary number is either zero, twice a binary number, or one more than twice a binary number.*

1. *First, write an inductive definition of the type bin corresponding to this description of binary numbers.*
2. *Next, write an increment function for binary numbers, and a function to convert binary numbers to unary numbers.*

Chapter 2

Proof by Induction

The next line imports all definitions from the previous chapter.

```
open import Basics
```

By processing this file with Agda using C-c C-l, the file *Basics* is loaded automatically.

2.1 Naming Cases

Unlike Coq, Agda does not have a way to define tactics for naming cases in proofs by case analysis. Agda takes very seriously the Curry-Howard isomorphism in which case analysis is understood as pattern matching in function definitions. Consider the following example, that show a property of and function:

```
andElim1 : forall (b c : Bool) → and b c ≡ True → b ≡ True
andElim1 b c prf = { }0
```

We can prove this by case analysis on b, since and is defined by analysis on the first argument (see Section 1.2.2). So, we put b on the hole and trigger the case analysis using Agda mode to produce the following:

```
andElim1 : forall (b c : Bool) → and b c ≡ True → b ≡ True
andElim1 True c prf = { }0
andElim1 False c prf = { }1
```

Now, Agda type checker is able to reduce the hypothesis prf and the goals hold definitionally using refl.

```
andElim1 : forall (b c : Bool) → and b c ≡ True → b ≡ True
andElim1 True c prf = refl
andElim1 False c prf = prf
```

Exercise 2.1 (Simple case analysis proof.)**.** *Prove* andElim2*:*

```
andElim2 : forall (b c : Bool) → and b c ≡ True → c ≡ True
andElim2 b c prf = { }0
```

2.2 Proof by Induction

We proved in the last chapter that 0 is a neutral element for + on the left using a simple argument. The fact that it is also a neutral element on the right...

```
plus0R : forall (n : Nat) → n + 0 ≡ n
plus0R n = refl   -- does not type check!
```

... cannot be proved in the same simple way. Just using refl doesn't work: the n in n + 0 is an arbitrary unknown number, so Agda cannot reduce the definition of + can't be simplified for check that n + 0 ≡ n.

And reasoning by cases doesn't get us much further: the branch of the case analysis where we assume n = 0 goes through, but in the branch where n = S n′ for some n′ we get stuck in exactly the same way. We could pattern match on n′ to get one step further, but since n can be arbitrarily large, if we try to keep on like this we'll never be done.

```
plus0R : forall (n : Nat) → n + 0 ≡ n
plus0R zero = refl
plus0R (suc n′) = { }0   -- stuck here...
```

To prove such facts — indeed, to prove most interesting facts about numbers, lists, and other inductively defined sets — we need a more powerful reasoning principle: **induction**.

Recall (from high school) the principle of induction over natural numbers: If P (n) is some proposition involving a natural number n and we want to show that P holds for all numbers n, we can reason like this:

- show that P (zero) holds;
- show that, for any n′, if P (n′) holds, then so does P (suc n′); conclude that P (n) holds for all n.

Note that the induction hypothesis can be seen as a "recursive call" of the property being proved over n′. Taking this view, induction proofs are just recursive functions! Curry-Howard isomorphism, strikes again! We need to use the induction hypothesis plus0R n′, which has type n′ + 0 ≡ n′, but the hole has type ?0 : (suc n′ + 0) ≡ suc n′, that can be simplified by Agda to ?0 : suc (n′ + 0) ≡ suc n′, according to the definition of +. Note that, that only difference between the type suc (n′ + 0) ≡ suc n′ and the type of induction hypothesis n′ + 0 ≡ n′ is the application of suc constructor in both sides of the equality, showing a *congruence* property. Function cong allows us to reason in this way by applying a function on both sides of a given equality. For now, we will only use cong and other equality related functions as "black-boxes". Latter, we will see how these are defined in Agda.

```
plus0R : forall (n : Nat) → n + 0 ≡ n
plus0R zero = refl
plus0R (suc n′) = cong suc (plus0R n′)
```

As another example of an inductive proof over natural numbers, consider the following theorem:

```
minusDiag : forall (n : Nat) → n − n ≡ 0
minusDiag zero = refl
minusDiag (suc n′) = minusDiag n′
```

Exercise 2.2 (Simple induction proofs)**.** *Prove the following lemmas using induction. You might need previously proven results.*

```
mult0R : forall (n : Nat) → n * 0 ≡ 0
mult0R = { }0
plusNSucM : forall (n m : Nat) → suc (n + m) ≡ n + suc m
plusNSucM = { }1
plusComm : forall (n m : Nat) → n + m ≡ m + n
plusComm = { }2
plusAssoc : forall (n m p : Nat) → n + (m + p) ≡ (n + m) + p
plusAssoc = { }3
```

Exercise 2.3. *doublePlus Consider the following function, which doubles its argument:*

```
double : Nat → Nat
double zero = zero
double (suc n') = suc (suc (double n'))
```

Use induction to prove this simple fact about double:

```
doublePlus : forall (n : Nat) → double n ≡ n + n
doublePlus = { }0
```

2.3 Proofs within Proofs

In informal mathematics, large proofs are very often broken into a sequence of theorems, with later proofs referring to earlier theorems. Occasionally, however, a proof will need some miscellaneous fact that is too trivial (and of too little general interest) to bother giving it its own top-level name. In such cases, it is convenient to be able to simply state and prove the needed “sub-theorem” right at the point where it is used. In Agda, we can do this by simply stating this little theorem as a local definition using **where** reserved word. For people used to Haskell, local definitions are just as in Haskell. In fact, Agda syntax is heavily based on Haskell's. Next example shows how to use local definitions in a proof.

```
mult0Plus' : forall (n m : Nat) → (0 + n) * m ≡ n * m
mult0Plus' n m = cong (\n → n * m) h where
  h : 0 + n ≡ n
  h = refl
```

Here we use h as a sub-proof of the fact 0 + n ≡ n and a anonymous function — denoted by the — to represent the greek letter λ, that is used in λ-calculus to represent a function binding symbol.

The next is a more elaborate example of local definitions in proofs. Note that we need to make a local proof of the fact that n + m ≡ m + n, using the previous fact (proved in an exercice) that addition is commutative.

```
plusRearrange : forall (n m p q : Nat) → (n + m) + (p + q) ≡ (m + n) + (p + q)
plusRearrange n m p q = cong (\n → n + p + q) nmComm where
  nmComm : n + m ≡ m + n
  nmComm = plusComm n m
```

Exercise 2.4 (Commutativity of Multiplication). *Use local definitions to help prove this theorem. You shouldn't need to use induction.*

```
plusSwap : forall (n m p : Nat) → n + (m + p) ≡ m + (n + p)
plusSwap = { }0
```

Now prove commutativity of multiplication. (You will probably need to define and prove a separate subsidiary theorem to be used in the proof of this one.) You may find that plusSwap *comes in handy.*

```
multComm : forall (n m : Nat) → n ∗ m ≡ m ∗ n
multComm = { }0
```

Exercise 2.5 (even n implies odd suc n). *Prove the following simple fact:*

```
evenNoddSucN : forall (n : Nat) → evenb n ≡ ¬ (oddb (suc n))
evenNoddSucN = { }0
```

2.4 More Exercices

Exercise 2.6. *Take a piece of paper. For each of the following theorems, first think about whether (a) it can be proved using only simplification and rewriting, (b) it also requires case analysis, or (c) it also requires induction. Write down your prediction. Then fill in the proof. (There is no need to turn in your piece of paper; this is just to encourage you to reflect before hacking!)*

```
bleNatRefl : forall (n : Nat) → True ≡ bleNat n n
bleNatRefl = { }0
zeroNbeqSuc : forall (n : Nat) → beqNat 0 (suc n) ≡ False
zeroNbeqSuc = { }1
andFalseR : forall (b : Bool) → and b False ≡ False
andFalseR = { }2
plusBleCompatL : forall (n m p : Nat) → bleNat n m ≡ True →
  bleNat (p + n) (p + m) ≡ True
plusBleCompatL = { }3
sucNBeq0 : forall (n : Nat) → beqNat (suc n) zero ≡ False
sucNBeq0 = { }4
mult1L : forall (n : Nat) → 1 ∗ n ≡ n
mult1L = { }5
all3Spec : forall (b c : Bool) → or (and b c) (or (¬ b) (¬ c)) ≡ True
all3Spec =  { }6
multPlusDistrR : forall (n m p : Nat) → (n + m) ∗ p ≡ (n ∗ p) + (m ∗ p)
multPlusDistrR =  { }7
multAssoc : forall (n m p : Nat) → n ∗ (m ∗ p) ≡ (n ∗ m) ∗ p
multAssoc = { }8
beqNatRefl : forall (n : Nat) → True ≡ beqNat n n
beqNatRefl = { }9
```

2.5 Equational Reasoning

Agda's supports for mixfix operators offers an excellent oportunity for creating operators that can resamble pencil-and-paper style of reasoning. In this section we will see how to use support for equational reasoning to construct a proof for commutativity of addition for natural numbers.

NOTE: I belive that here would be nice to talk a bit about the usage of equational reasoning proofs. Latter, in another chapter, talk about propositional equality and some functions over it.

Chapter 3

Lists — Working with Structured Data

3.1 Pairs of Numbers

In a data type definition, each constructor can take any number of arguments — none (as with *True* and zero), one (as with suc), or more than one, as in this definition:

```
data NatProd : Set where
  _,_ : Nat → Nat → NatProd
```

This declaration can be read: "There is just one way to construct a pair of numbers: by applying the constructor , to two arguments of type *Nat*."

Here are two simple function definitions for extracting the first and second components of a pair. (The definitions also illustrate how to do pattern matching on two-argument constructors.)

```
fst : NatProd → Nat
fst (n, _) = n
snd : NatProd → Nat
snd (_, n) = n
```

The *NatProd* ilustrates that, in Agda, we can define infix constructors naturally by marking argument positions with underscores.

Let's try and prove a few simple facts about pairs. If we state the lemmas in a particular (and slightly peculiar) way, we can prove them with just refl (and its built-in simplification):

```
surjectivePairing : forall (n m : Nat) → (n, m) ≡ (fst (n, m), snd (n, m))
surjectivePairing n m = refl
```

Another way to state and prove this simple lemma is using pattern matching (case analysis):

```
surjectivePairing′ : forall (p : NatProd) → p ≡ (fst p, snd p)
surjectivePairing′ (n, m) = refl
```

Here, in order to be able to state the definitional equality between p and (fst p, snd p) we need to pattern match on p in order to Agda be able to reduce functions fst and snd.

Exercise 3.1 (Projections and Swap)**.** *First, define a function* swap *which swaps the first and second element of a given pair:*

```
swap : NatProd → NatProd
swap = { }0
```

Next prove this property:

```
fstSndSwap : forall (p : NatProd) → (snd p, fst p) ≡ swap p
fstSndSwap = { }0
```

3.2 List of Numbers

Generalizing the definition of pairs a little, we can describe the type of lists of numbers like this: "A list is either the empty list or else a pair of a number and another list."

```
data NList : Set where
  nil : NList
  _,_ : Nat → NList → NList
infixr 4 _,_
```

As an example, here we have a simple 3-element list:

```
sample : NList
sample = 1, 2, 3, nil
```

As you might be an alert reader, Agda supports overloading of data constructors. The context of a given expression is used to determine of which we are considering.

A number of functions are useful for manipulating lists. For example, the repeat function takes a number n and a count and returns a list of length count where every element is n.

```
repeat : Nat → Nat → NList
repeat n zero = nil
repeat n (suc m) = n, repeat n m
```

The length function calculates the number of elements of a given list.

```
length : NList → Nat
length nil = zero
length (x, xs) = suc (length xs)
```

The ++ ("append") function concatenate two lists:

```
infixr 4 _++_
_++_ : NList → NList → NList
nil ++ ys = ys
(x, xs) ++ ys = x, (xs ++ ys)
```

Here are two smaller examples of programming with lists. The head function returns the first element (the "head") of the list, while tail returns everything but the first element (the "tail"). Of course, the empty list has no first element, so we must pass a default value to be returned in that case.

```
head : NList → (default : Nat) → Nat
head nil d = d
head (x, _) d = x

tail : NList → NList
tail nil = nil
tail (_, xs) = xs
```

Exercise 3.2 (Definition of nonZeros)**.** *Define the function* nonZeros *that remove all* zero *values from a* NList*. Implement your function in such a way that* testNonZeros *be a type correct term.*

```
nonZeros : NList → NList
nonZeros = { }0

testNonZeros : (0, 1, 0, 2, 0, nil) ≡ (1, 2, nil)
testNonZeros = refl
```

Exercise 3.3 (Definition of oddMembers)**.** *Define the function* oddMembers *that remove all even values from a* NList*. Implement your function in such a way that* testOddMembers *be a type correct term.*

```
oddMembers : NList → NList
oddMembers = { }0

testOddMembers : (0, 1, 0, 3, 0, nil) ≡ (1, 3, nil)
testOddMembers = refl
```

Exercise 3.4 (Definition of alternate)**.** *Implement* alternate *that alternates two given* NLists *into one.*

```
alternate : NList → NList → NList
alternate = { }0
```

3.3 Bag via Lists

A bag (or multiset) is like a set, but each element can appear multiple times instead of just once. One reasonable implementation of bags is to represent a bag of numbers as a list.

```
Bag : Set
Bag = NList
```

Note that *Bag* is explicitly annotated with type Set, that is the type of "types".

Exercise 3.5 (Functions over bags)**.** *Complete the following definitions for the functions* count*,* sum*,* add*, and* member *for bags. Again, implement your functions in a way that the test cases are typeable by Agda.*

```
count : Bag → Bag
count = { }0

testCount1 : count 1 (1, 2, 1, nil) ≡ 2
testCount1 = refl

testCount2 : count 3 (1, 2, 1, nil) ≡ 0
testCount2 = refl
```

```
sum : Bag → Nat
sum = { }0
testSum : sum (1, 2, 1, nil) ≡ 6
testSum = refl
add : Nat → Bag → Bag
add = { }0
testAdd : count 1 (add 1 (1, 2, 1, nil)) ≡ 3
testAdd = refl
member : Nat → Bag → Bool
member = { }0
testMember1 : member 1 (1, 2, 1, nil) ≡ True
testMember1 = refl
testMember2 : member 3 (1, 2, 1, nil) ≡ False
testMember2 = refl
```

Exercise 3.6 (More bag functions)**.** *Here are some more bag functions for you to practice with.*

```
removeOne : Nat → Bag → Bag
removeOne = { }0
testRemoveOne1 : count 1 (removeOne 1 (1, 2, 1, nil)) ≡ 1
testRemoveOne1 = refl
testRemoveOne2 : count 1 (removeOne 1 (1, 2, 1, nil)) ≡ 2
testRemoveOne2 = refl
removeAll : Nat → Bag → Bag
removeAll = { }0
testRemoveAll : count 1 (removeAll 1 (1, 2, 1, nil)) ≡ 0
testRemoveAll = refl
subset : Bag → Bag → Bool
subset = { }0
testSubset1 : subset (1, 2, nil) (3, 1, 2, nil) ≡ True
testSubset1 = refl
testSubset2 : subset (2, 3, nil) (1, 2, 4, nil) ≡ False
testSubset2 = refl
```

Exercise 3.7. *Write down an interesting theorem about bags involving the functions* count *and* add*, and prove it. Note that, since this problem is somewhat open-ended, it's possible that you may come up with a theorem which is true, but whose proof requires techniques you haven't learned yet. Feel free to ask for help if you get stuck!*

3.4 Reasoning about Lists

Just as with numbers, simple facts about list-processing functions can sometimes be proved entirely by simplification. For example, the simplification performed by refl is enough for this theorem...

```
nilAppL : forall (n : NList) → nil ++ n ≡ n
nilAppL n = refl
```

... because the nil is substituted into the match position in the definition of ++, allowing the match itself to be simplified.

Also, as with numbers, it is sometimes helpful to perform case analysis on the possible shapes (empty or non-empty) of an unknown list, as shown in this little theorem:

```
tailLengthPred : forall (n : NList) → pred (length n) ≡ length (tail n)
tailLengthPred nil = refl
tailLengthPred (x, n) = refl
```

Usually, though, interesting theorems about lists require induction for their proofs.

3.4.1 Micro-sermon

Simply reading example proofs will not get you very far! It is very important to work through the details of each one, using Agda and thinking about what each step of the proof achieves. Otherwise it is more or less guaranteed that the exercises will make no sense.

3.4.2 Induction on Lists

Proofs by induction over datatypes like *NList* are perhaps a little less familiar than standard natural number induction, but the basic idea is equally simple. Each data type declaration defines a set of data values that can be built up from the declared constructors: a boolean can be either *True* or *False*; a number can be either zero or suc applied to a number; a list can be either nil or , applied to a number and a list.

Moreover, applications of the declared constructors to one another are the only possible shapes that elements of a inductively defined set can have, and this fact directly gives rise to a way of reasoning about inductively defined sets: a number is either zero or else it is suc applied to some smaller number; a list is either nil or else it is , applied to some number and some smaller list; etc. So, if we have in mind some proposition P that mentions a list l and we want to argue that P holds for all lists, we can reason as follows:

- First, show that P is true of l when l is nil.
- Then show that P is true of l when l is n, l′ for some number n and some smaller list l′, assuming that P is true for l′.

Since larger lists can only be built up from smaller ones, eventually reaching nil, these two things together establish the truth of P for all lists l. Here's a concrete example:

```
appAssoc : forall (l1 l2 l3 : NList) → l1 ++ (l2 ++ l3) ≡ (l1 ++ l2) ++ l3
appAssoc nil l2 l3 = refl
appAssoc (x, l1) l2 l3 =
  ((x, l1) ++ (l2 ++ l3)) ≡ [refl]
  (x, (l1 ++ (l2 ++ l3))) ≡ [cong (\p → x, p) (appAssoc l1 l2 l3)]
  (x, ((l1 ++ l2) ++ l3))
  □
```

Here we use Agda infix operators to do some equational reasoning in proofs, that allow us to argue in a pencil and paper fashion. The operator ≡ [?] acts like a equality that uses the term between brackets to rewrite the current term and □ finish the proof.

The same proof can be done in the following way, in a paper:

Theorem 1. *For all NLists* l1, l2 *and* l3, *we have* l1 ++ (l2 ++ l3) ≡ (l1 ++ l2) ++ l3.

Proof. We will proceed by induction on l1.

1. Case l1 = nil: In this case we have: nil $++(l2++l3) \equiv l2++l3 \equiv$ (nil $++l2)++l3$, as required.

2. Case l1 = x, l1′: We have that:

$$\begin{array}{lcl}
(x\,,\,l1') \mathbin{++} (l2 \mathbin{++} l3) & \equiv & \{\text{by def.}\} \\
x\,,\,(l1' \mathbin{++} (l2 \mathbin{++} l3)) & \equiv & \{\text{by I.H.}\} \\
x\,,\,((l1' \mathbin{++} l2) \mathbin{++} l3 & \equiv & \{\text{by def.}\} \\
((x\,,\,l1') \mathbin{++} l2) \mathbin{++} l3 & \square &
\end{array}$$

   as required.

□

Here's another simple example:

```
appLength : forall (n n′ : NList) → length (n ++ n′) ≡ length n + length n′
appLength nil n′ = refl
appLength (x, xs) n′ =
  length ((x, xs) ++ n′)     ≡ [refl]
  length (x, (xs ++ n′))     ≡ [refl]
  suc (length (xs ++ n′))  ≡ [cong suc (appLength xs n′)]
  suc (length xs + length n′)  ≡ [refl]
  length (x, xs) + length n′
  □
```

Exercise 3.8 (Practice informal proof)**.** *Prove* appLength *theorem using a informal style, like the proof for* appAssoc*.*

For a slightly more involved example of an inductive proof over lists, suppose we define a "cons on the right" function snoc like this...

```
snoc : Nat → NList → NList
snoc n nil = n, nil
snoc n (x, xs) = x, (snoc n xs)
```

... and use it to define a list-reversing function rev like this:

```
rev : NList → NList
rev nil = nil
rev (x, xs) = snoc x (rev xs)
```

Now let's prove some more list theorems using our newly defined snoc and rev. For something a little more challenging than the inductive proofs we've seen so far, let's prove that reversing a list does not change its length. Our first attempt at this proof gets stuck in the successor case...

```
revLength : forall (n : NList) → length (rev n) ≡ length n
revLength nil = refl
revLength (x, xs) =
  length (rev (x, xs))  ≡ [refl]
  length (snoc x (rev xs))  ≡ [{ }0]
  suc (length (rev xs))  ≡ [{ }1]
  suc (length xs)        ≡ [{ }2]
```

```
  length (x, xs)
  □
```

So let's take the equation about snoc that would have enabled us to make progress and prove it as a separate lemma.

```
lengthSnoc : forall (n : Nat) (l : NList) → length (snoc n l) ≡ suc (length l)
lengthSnoc n nil = refl
lengthSnoc n (x, xs) =
  length (snoc n (x, xs)) ≡ [refl]
  length (x, (snoc n xs)) ≡ [refl]
  suc (length (snoc n xs)) ≡ [cong suc (lengthSnoc n xs)]
  suc (length (x, xs))
  □
```

Note that we make the lemma as general as possible: in particular, we quantify over all natlists, not just those that result from an application of rev. This should seem natural, because the truth of the goal clearly doesn't depend on the list having been reversed. Moreover, it is much easier to prove the more general property.

Now we can complete the original proof.

```
revLength : forall (l : NList) → length (rev l) ≡ length l
revLength nil = refl
revLength (x, xs) =
  length (rev (x, xs))      ≡ [refl]
  length (snoc x (rev xs)) ≡ [lengthSnoc x (rev xs)]
  suc (length (rev xs))    ≡ [cong suc (revLength xs)]
  suc (length xs)          ≡ [refl]
  length (x, xs)
  □
```

Theorem 2. *For all numbers n and lists l, length (snoc n l) = suc (length l).*

Proof. We will proceed by induction l.

- Caso l = nil. We have that:

$$
\begin{array}{lcl}
\text{length (snoc n nil)} & \equiv & \{\text{by def.}\} \\
\text{length (n, nil)} & \equiv & \{\text{by def.}\} \\
\text{suc zero} & \equiv & \{\text{by def.}\} \\
\text{suc (length nil)} & &
\end{array}
$$

  as required.

- Caso l = x, xs. We have that:

$$
\begin{array}{lcl}
\text{length (snoc n (x, xs))} & \equiv & \{\text{by def.}\} \\
\text{length (x, snoc n xs)} & \equiv & \{\text{by def.}\} \\
\text{suc (length (snoc n xs))} & \equiv & \{\text{by def.}\} \\
\text{suc (suc (length xs))} & \equiv & \{\text{by I.H.}\} \\
\text{suc (length (x, xs))} & &
\end{array}
$$

  as required.

□

Theorem 3. *Theorem: For all lists l, length (rev l) = length l.*

Proof. We will proceed by induction l.

- Case l = nil. We have that:

$$\begin{array}{lll} \text{length (rev nil)} & \equiv & \{\text{by def.}\} \\ \text{length nil} & \equiv & \{\text{by def.}\} \\ \text{zero} & \equiv & \{\text{by def.}\} \\ \text{length nil} & & \end{array}$$

  as required.

- Case l = x, xs. We have that:

$$\begin{array}{lll} \text{length (rev (x , xs))} & \equiv & \{\text{by def.}\} \\ \text{length (snoc x (rev xs))} & \equiv & \{\text{by lengthSnoc}\} \\ \text{suc (length (rev xs))} & \equiv & \{\text{by I.H.}\} \\ \text{suc (length xs)} & & \end{array}$$

  as required.

□

3.4.3 List Exercices, Part 1

More practice with lists.

```
appNilEnd : forall (l : NList) → l ++ nil ≡ l
appNilEnd = { }0
revInvolutive : forall (l : NList) → rev (rev l) ≡ l
revInvolutive = { }1
```

There is a short solution to the next exercise. If you find yourself getting tangled up, step back and try to look for a simpler way.

```
appAss4 : forall (l1 l2 l3 l4 : NList) → l1 ++ (l2 ++ (l3 ++ l4)) ≡ ((l1 ++ l2) ++ l3) ++ l4
appAss4 = { }0
snocApp : forall (l : NList) → snoc n l ≡ l ++ (n, nil)
snocApp = { }1
distrRev : forall (l1 l2 : NList) → rev (l1 ++ l2) ≡ rev l2 ++ rev l1
distrRev = { }2
revInjective : forall (l l′ : NList) → rev l ≡ rev l′ → l ≡ l′
revInjective = { }3
```

3.5 Maybe

Here is another type definition that is often useful in day-to-day programming:

```
data NatMaybe : Set where
  Just : Nat → NatMaybe
  Nothing : NatMaybe
```

One use of natoption is as a way of returning "error codes" from functions. For example, suppose we want to write a function that returns the nth element of some list. If we give it type $Nat \to NList \to Nat$, then we'll have to return some number when the list is too short!

```
indexBad : Nat → NList → Nat
indexBad _ nil = 42   -- arbitrary...
indexBad zero (x, xs) = x
indexBad (suc n) (_, xs) = indexBad n xs
```

On the other hand, if we give it type $Nat \to NList \to NatMaybe$, then we can return *Nothing* when the list is too short and *Just* a when the list has enough members and a appears at position n.

```
index : Nat → NList → NatMaybe
index _ nil = Nothing
index zero (x, _) = Just x
index (suc n) (_, xs) = index n xs
```

Exercise 3.9 (Head)**.** *Implement the function* head *from earlier so we don't have to pass a default element for the* nil *case.*

Exercise 3.10 (Equality of NLists)**.** *Define a function* beqNList *that tests the equality of two given* NLists *and prove the following property:*

```
beqNList : forall (l : NList) → beqNList l l
beqNList = { }0
```