# An Introduction to Programming and Proving in Agda (incomplete draft)

Peter Dybjer

January 29, 2018

## 1 A first Agda module

Your first Agda-file is called `BoolModule.agda`. Its contents are

```
module BoolModule where

data Bool : Set where
  true : Bool
  false : Bool

not : Bool -> Bool
not true = false
not false = true
```

It declares a module `BoolModule` which contains

- the definition of the data type `Bool` with the two constructors `true` and `false`
- the definition of the function `not` by case analysis (or pattern matching) on the constructors for `Bool`.

Note the following:

- Each Agda file `file.agda` is a module with the same name `file`. It is nevertheless compulsory to explicitly declare it by writing

  ```
  module file where
  ```

- The data type `Bool` is an element of the type `Set`. We call `Set` a ”large” type (since it is a type of types) and its elements ”small” types.
- We must first declare the type of the function `not` before writing the pattern matching equations.

**Standard Library.** Agda has a standard library where many common types and functions are defined (including booleans and negation).

### 1.1 Comparison to Haskell

We can write analogous definitions of `Bool` and `not` in Haskell, where `Bool` is part of the standard library for Haskell. If it were a user defined data type its definition would be

```
data Bool = True | False
```

where the Haskell convention is that constructor names begin with capital letters. The definition of `not` in Haskell is

```
not True = False
not False = True
```

Note that we do not need to declare its type because Haskell infers it.

Haskell's module system is also different from Agda's, but we will not discuss this here.

## 2 The Agda mode of Emacs

### 2.1 Parsing and type-checking an Agda-file

Usually, Agda-files are written using the agda-mode of the Emacs editor. In this mode you have access to a number of commands. First and foremost, the command "load" parses and type-checks your file. You execute by either typing `^C^L` or by choosing it under the Agda-menu in Emacs. If successful, a coloured version of your Agda file is then displayed.

For example, the file `BoolModule.agda` type-checks correctly and is displayed as follows:

```
module BoolModule where

data Bool : Set where
 true : Bool
 false : Bool

not : Bool -> Bool
not true = false
not false = true
```

The keywords module, where, and data have become brown, the module name BoolModule purple, the types Bool and Set and the defined function not blue, and the constructors true and false green. The colon (:) and the function arrow (->) have remained black.

## 2.2 Running an Agda-program

An Agda-program is a well-typed expression. To run an Agda-program means to evaluate such an expression to a value (normal form). You can do this by executing the command `^C^N` ("normalize") and then write the expression you want to evaluate in the lower window (after Expression:). For example, if you write the expression

```
  not true
```

the system will return

```
  false
```

in the same window.

Remark. Agda's evaluator is more general than Haskell's, because it also evaluates under binders. More about this later.

## 2.3 Unicode

We can use unicode in Agda, and for example write → instead of -> for the function arrow. To enter this we write the latex code for the unicode symbol, e g, `\to` for →. Similarly, we can write the latex code `\neg` if we want to use the negation symbol ¬ for not.

## 2.4 Writing Agda-programs by gradual refinement of partial expressions

Type-checking is a complex matter in dependently typed programming. It is therefore very useful to write ones programs by gradual refinement of "partial expressions", that is, expressions with "holes" denoted by "?". Examples of well-typed partial expressions with respect to the module BoolModule.agda are

```
  ? : Bool
  b : Bool
  true : Bool
  not ? : Bool
  not (not ?) : Bool
  not (not b) : Bool
  ? : Set
  ? -> ? : Set
  Bool -> ? : Set
  Bool : Set
```

Note that partial expressions can also contain variables, provided they have been declared, such as `b : Bool` above.

We can write an Agda program by gradually refining a well-typed partial expression into an ordinary well-typed total expression. For example, we can write the function

```
¬¬ : Bool → Bool
¬¬ b = not (not b)
```

by gradually refining the right hand side of the equation. (Note that ¬¬ is a single identifier. Consult the Agda wiki for more information.) First write

```
¬¬ b = ?
```

indicating that the right hand side is completely undefined. If we load the file `^C^L` we can check that we have a well-typed partial expression and indeed ”?” is a well-typed partial expression of type `Bool`. Moreover, the Agda system replaces our ”?”-mark with a ”hole”:

```
¬¬ b = { }0
```

The idea is that you should now fill the hole by writing an expression (or partial expression) in it.

### 2.4.1 Complete refinement (”give”)

If you are ready to fill the hole with the complete expression you first type it inside the hole

```
¬¬ b = {not (not b)}0
```

and then you execute the command `^C^SPC` (”give”) while keeping the cursor inside the hole. Agda will now replace the hole by the complete expression and type-check it.

### 2.4.2 Partial refinement (”refine”)

You can also partially refine the expression in the hole. For example, if you have only decided that the top most function symbol is `not`, but want to leave the argument undefined, you first write

```
¬¬ b = {not ?}0
```

and then do the command ”refine” `^C^R` while keeping the cursor inside the hole. You will get

```
¬¬ b = not { }1
```

Again, you can choose either complete or partial instantiation of the new hole. Say that this time we choose to again partially instantiate it with the top most function symbol `not`:

```
¬¬ b = not ({not ?}1)
```

After doing ”refine” you will get

```
¬¬ b = not (not { }2)
```

Finally you fill the hole by the variable `b`:

```
¬¬ b = not (not {b }2)
```

and execute either ”give” or ”refine”, to finish the definition.

In the important special case when the partial expression is a function symbol followed by one or more ?-marks it suffices to write the function symbol in the hole. For example, writing

```
¬¬ b = {not}0
```

is equivalent to writing

```
¬¬ b = {not ?}0
```

## 2.5 Mixfix operations

You can declare function symbols with mixfix syntax using underscores to mark the argument positions. An example is the following definition of the conditional (the special case where the return type is `Bool`):

```
if_then_else_ : Bool → Bool → Bool → Bool
if true then y else z = y
if false then y else z = z
```

Note that in the declaration line there are no spaces around the underscores, but when applied there must be spaces around the arguments.

Infix syntax is a special case of mixfix. For example, here is a definition of ”and” for booleans:

```
_&&_ : Bool → Bool → Bool
b && true = b
b && false = false
```

## 2.6 Precedences

You can declare the precedence of an operation and also the associativity of infix operations. For example,

```
infixl 50 _&&_
```

means that `_&&_` is a left associative operation with precedence 50.

## 2.7 Generating patterns

It is common to define functions by pattern matching on the constructors of a data type. When defining such a function by gradual refinement, Agda has a special command `^C^C` ("case split") for generating the clauses for pattern matching.

Let us consider the gradual refinement of the definition of && and above. We begin by writing

```
_&&_ : Bool → Bool → Bool
b && c = ?
```

After type-checking (loading) the definition line becomes

```
b && c = { }0
```

We now want to case split on the variable `c : Bool`. To do this we write `c` in the hole and then write `^C^C` which instructs Agda to do case analysis on `c`. We get two pattern matching equations, one for each of the constructors for `Bool`:

```
b && true  = { }1
b && false = { }2
```

We can now fill the two holes by `b` and `false` respectively.

**Exercise:** Define the operation `not` by gradual refinement and pattern matching!

**Exercise:** Define logical or, logical implication, and logical equivalence

```
_||_ : Bool → Bool → Bool
_=>_ : Bool → Bool → Bool
_<=>_ : Bool → Bool → Bool
```

by gradual refinement and pattern matching using `^C^C, ^C^R`, and `^C^SPC`.

## 2.8 Comments

Comments are either lines which begin with "–" or text `...` which is enclosed by braces and dashes as follows: "`{- ... -}`":

# 3 Natural numbers

Most programming languages provide a type of positive and negative integers and the basic arithmetic operations on them. For example, Java has the type `int` of 32-bit signed 2-complement integers (and also the types `short` of 16-bit and `long` of 64-bit versions). Haskell provides the type `Int` of fixed precision integers and `Integer` of arbitrary precision integers.

## 3.1 Unary natural numbers

In Agda, neither integers nor natural numbers are primitive concepts; they have to be defined. From the point of view of logic the most basic form of natural numbers are the unary natural numbers, that is, the numbers generated by the two constructors zero for the number 0 and succ for the successor function which adds 1 to a number. In Agda:

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

All natural numbers can now be generated by applying `succ` a finite number of times to `zero`:

```
one = succ zero
two = succ one
```

etc. Note that in this case Agda lets us skip the type declarations although they are usually compulsory.

## 3.2 Arithmetic operations

Similarly, the arithmetic operations are not primitive in Agda, but are defined by the user in terms of pattern matching and recursion:

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

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

These are example of a particularly simple and important kind of recursive definition, called *primitive* recursion, where the recursive call defines the value of a function f on succ $n$ in terms of the value of f on `n`. One also says that such a definition is by induction on `n`: there is a base case for zero and a step case for succ `n`. Functions defined by primitive recursion always terminate.

We can declare both + and $*$ to be left associative infix operations such that $*$ binds harder than +:

```
infixl 60 _+_
infixl 70 _*_
```

Thus `x + y + z` parses as `(x + y) + z` in the definition below:

```
f : Nat → Nat → Nat → Nat
f x y z = x + y + z
```

## 3.3 Decimal notation and machine arithmetic

It is inconvenient to write unary natural numbers and inefficient to compute the arithmetic operations using their primitive recursive definitions as simplification rules. Therefore, Agda has a ”pragma”

```
{-# BUILTIN NATURAL Nat #-}
```

which tells Agda that the type `Nat` can be compiled as the built-in machine integers. Moreover, it tells Agda that decimal notation can be used instead of unary, so that you for example can write 1 for succzero.

Moreover, if you write

```
{-# BUILTIN NATPLUS _+_ #-}
{-# BUILTIN NATTIMES _*_ #-}
```

the Agda compiler (see Agda wiki) will use binary machine arithmetic for computing + and *.

There are many other built in types: integers, floats, strings, and characters. See the Agda wiki for more information.

Exercise: Define cut-off subtraction:

```
_-_ : Nat → Nat → Nat
```

This is defined like ordinary subtraction on integers when the result is non-negative, but returns `zero` when ordinary subtraction would have returned a negative integer.

Exercise: Define the less than function:

```
_<_ : Nat → Nat → Bool
```

Remark. We might want to make a special module for natural numbers called `NatModule`. If we want to access definitions from `BoolModule` we need to explicitly import and open it:

```
module NatModule where

open import BoolModule
```

See the AgdaWiki for more information.

Exercise: Define a version of the conditional which returns natural numbers (rather than booleans as in section 1):

```
ifn_then_else_ : Bool → Nat → Nat → Nat
```

Exercise. Define the power function

```
power : Nat → Nat → Nat
```

Exercise. Define the factorial function

```
factorial : Nat → Nat
```

Compute the result of factorial for some arguments by normalizing with `^C^N`. How large factorials can Agda compute?

Exercise. Define the Ackerman function. (See wikipedia.)

Exercise. Define a data type of binary numbers in Agda and then define addition of binary numbers. Note that there are several issues: should you avoid leading zeros, for example? Can you define translation maps back and forth to the unary numbers? Note that if you do not allow leading zeros, it may be easier to translate back and forth to strictly positive unary numbers.

Exercise. How would you define integers as a data type in Agda? How would you define addition of integers? Subtraction?

## 3.4 Termination checking

In Agda you are only allowed to define total functions, that is, functions which terminate and return a value when applied to an arbitrary well-typed argument. For example, division is a partial function on natural numbers, since the result of dividing by 0 is undefined, so it should not be accepted.

Agda has a termination checker that will check that your functions terminate. All the functions defined so far are defined by primitive recursion and are easily recognized as terminating by Agda. On the other hand, if you define division by repeated subtraction as follows

```
_÷_ : Nat → Nat → Nat
m ÷ n = ifn m < n then zero else succ ((m - n) ÷ n)
```

then Agda will report that the termination checking failed:

```
Termination checking failed for the following functions:
  _÷_
Problematic calls:
  (m - n) ÷ n
```

The termination checker analyses the recursive calls of a function and tries to establish that these calls are on ”smaller” arguments. In the case of the division it checks whether the arguments in the recursive call `(m - n) ÷ n` are smaller than the respective arguments of the calling instance `m ÷ n`. It notices that it is not clear that `(m - n)` is strictly smaller than `m`. Indeed, if `n` is zero the division function will call itself with the same arguments and fail to terminate.

So how do we define division in Agda? We have to turn it into a total function. This can be done in several ways, either by returning an error value when dividing by 0, or by making sure that the input domain of the denominator does not include zero. We will show later how this can be done.

Assume now for simplicity that we use `zero` as the error value.

```
_÷_ : Nat → Nat → Nat
m ÷ n = ifn (iszero n || m < n) then zero else succ ((m - n) ÷ n)
```

Now, we have defined division as a total function. However, Agda's termination checker will still return the same error message. The reason is that it fails to recognize that we have ruled out the problematic case when `n` is zero in the recursive call. This requires more advanced reasoing than the termination checker is capable of. This example shows that the termination checker is not complete. There are terminating functions which are not accepted by it.

Exercise. It is of course unsatisfactory to use zero as an error value. Define a modified type of natural numbers with a special error value `error`. Then try to redefine division by repeated subtraction so that

```
m ÷ 0 = error
```

and note that the termination checker will not accept this definition.

It is possible to turn off the termination checker by including either of the following two pragmas in your file

```
{-# TERMINATING #-}
{-# NON-TERMINATING #-}
```

In the first case you declare a function to be terminating although it has not been checked by the termination checker. In the second case you allow the possibility of writing non-terminating functions, and in this way Agda can be used for writing general partial recursive functions in much the same way as Haskell. However, note that using non-terminating functions in types may break type-checking.

Note. In Martin-Löf type theory, the logical language which Agda grew out of, the only recursion which is allowed is primitive recursion (and analogous notions of structural recursion on other data types than natural numbers). Agda's termination checker has relaxed this condition and accepts a more general class of terminating recursive definitions. For example, it accepts the following definition of equality of natural numbers

```
_==_ : Nat → Nat → Bool
zero == zero = true
zero == succ n = false
succ m == zero = false
succ m == succ n = m == n
```

This is not a primitive recursive definition, since it does induction in both arguments simultaneously. Primitive recursion requires that induction is in one argument at a time.

Exercise. Define equality of natural numbers using only official primitive recursion. (See wikipedia for a precise definition.) Hint. Use cut-off subtraction.

### 3.5 Records

Discuss simply typed version of records: including examples of counters and finite sets. Polymorphism and axioms will be discussed later.

## 4 Dependent Types

So far all our Agda-programs have been written in the simply typed subset of Agda. We are now ready to move to the key feature: dependent types. An example is the type `Vect A n` of vectors of length (dimension) `n` where the elements have type `A`. We say that the type `Vect A n` called a dependent type, since `Vect A n` depends on `A : Set` and `n : Nat`. We have the following typing:

```
  Vect : Set → Nat → Set
```

We will however, postpone the definition of `Vect` for a bit, and instead show how to use of dependent types for defining polymorphic types and functions.

### 4.1 The logical framework

The core of Agda is a typed lambda calculus with dependent types, often called the ”Logical Framework” (”LF”). We already saw that it has a type `Set` and you can declare data types to be members of this type. So `Set` is a type of data types. (The types in `Set` are also called small types). We do not have

```
  Set : Set
```

because this would result in a logical paradox, similar to Russell's paradox for naive set theory.

If `A : Set`, then `A` is a type, that is, it can stand to the right of the ”:”-sign.

Moreover, we do not only have ordinary function types A → B, but also ”dependent function types” (x : A) → B, where the type B may depend on the variable `x : A`. For example, the function which maps a number `n : Nat` to the vector `[ 0, ..., n-1 ]` has type `(n : Nat) → Vect Nat n`.

## 4.2 Polymorphism

We shall now see how dependent function types can be used to express the types of polymorphic functions. A very simple example is the identity function (or ”combinator”):

```
id : (X : Set) → X → X
id X x = x
```

Informally, this says ”for any type X, we have an identity function id X : X → X”. Note that this is different from polymorphism in Haskell where we have a function id of type X → X, for a type variable X, and that X does not appear as an argument of id. Note also that the type expression

```
(X : Set) → X → X
```

is well-formed and that X → X is a type which ”depends” on the variable X : Set.

Remark: we can use λ-notation in Agda and instead write

```
id′ : (X : Set) → X → X
id′ =  X x → x
```

## 4.3 Implicit arguments

It would be tedious to always have to write out all type-variable arguments. For this purpose Agda has facility for declaring arguments to be ”implicit” by using the ”implicit dependent function type” `{x : A} → B`. The typing rule is that if `b : {x : A} → B`, and `a : A`, then `b : B[x := a]`, that is, that b is a term of type B where the term `a` has been substituted for the variable `x`. In this way we can recover Haskell’s notation for polymorphism, e g, the type of the identity function with implicit polymorphism is

```
id−implicit : {X : Set} → X → X
id−implicit x = x
```

Exercise: write id−implicit using λ-notation!

We can define many polymorphic combinators with implicit arguments. For example, the infix composition combinator can be defined by

```
_∘_ : {X Y Z : Set} → (Y → Z) → (X → Y) → X → Z
(g ∘ f) x = g (f x)
```

Exercise: Define the K and S-combinators! (See wikipedia if you don't know their definitions.)

In combinatory logic one shows that the two combinators K and S suffice for defining all other combinators. In particular one can show that the identity combinator can be defined as I = S K K. Check what happens if you let Agda normalize S K K! Explain.

## 4.4 Polymorphic lists

Data types can be polymorphic too. For example, in Haskell we have the type `[a]` of lists where the elements belong to the type `a` for an arbitrary type `a`. In Agda we declare the polymorphic data type of lists as follows:

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

This declaration states that the type of `List` is `Set → Set`, that is, if `A : Set` then `List A : Set`. Moreover, the types of the constructors have an implicit argument, which is not written out in the declaration above. Their proper types are:

```
 [] : {A : Set} → List A
_::_ : {A : Set} → A → List A → List A
```

Let us now define some list functions, for example map is defined as follows:

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (a :: as) = f a :: map f as
```

What about head? Let us try to define

```
head : {A : Set} → List A → A
head []         = ?
head (a :: as)  = a
```

In Haskell we can simply define head as a partial function, which is not defined for []. This is not possible in Agda, where all functions are total. The way out would be to define head [] as an explicit error element, for example ”Nothing” in Haskell's Maybe type defined by

```
data Maybe a = Just a | Nothing
```

Exercise: Define the Maybe type in Agda and the function

```
head : {A : Set} → List A → Maybe A
```

Note that we can make tail total without using Maybe by conventionally defining tail [] = [].

## 4.5 Cartesian products

### 4.5.1 Cartesian product as a polymorphic data type

The Cartesian product of two types (the type of pairs, cf Pierce, figure 11-5 p 126) is a polymorphic data type with one constructor:

```
data _×_ (A B : Set) : Set where
 <_,_> : A → B → A × B
```

Exercise: write out the full types of `_×_` and `<_,_>`. Hint: look at the definition of the data type List. Pierce uses `{_,_}`.

We can now define the first and the second projection:

```
fst : {A B : Set} → A × B → A
fst < a , b > = a

snd : {A B : Set} → A × B → B
snd < a , b > = b
```

### 4.5.2 Cartesian product as polymorphic record

Alternatively, we can define the cartesian product as a polymorphic record as follows:

```
record _×'_ (A B : Set) : Set where
 field
  l1 : A
  l2 : B
```

This is like A × B but with labels l1 and l2 for the two components. Then we can write `_×'_.l1 _×'_.l2` for the projections (cf fst and snd):

```
fst' : {A B : Set} -> A ×' B -> A
fst' = _×'_.l1

snd' : {A B : Set} -> A ×' B -> B
snd' = _×'_.l2
```

We can now define pairing of elements as a record

```
<_,_>' : {A B : Set} -> A -> B -> A ×' B
< a , b >' = record { l2 = b
 ; l1 = a
 }
```

Since we often want to have access to the constructor of a record, Agda allows us to declare it directly in the record definition

```
record _×''_ (A B : Set) : Set where
 constructor <_,_>''
 field
  l1 : A
  l2 : B
```

Remark. Pierce uses t.1 for fst t, the ”first projection” and t.2 for snd t, the ”second projection”.

Pierce’s evaluation rules are for ”strict” pairs, evaluating the first component to a value, then the second component. When projections are computed we first compute both components of the pair, then throw one away

In general we can define types of n-tuples for arbitrary n (pairs, triples, quadruples, ...). The type of 0-tuples is the unit type defined as follows:

```
data Unit : Set where
 <> : Unit
```

Alternatively it can be defined as a record with no fields:

```
record Unit' : Set where
 constructor <>'
```

Remark. An important difference between products defined as data types and products defined as record, is that the latter enjoys the property of ”surjective pairing”. For the unit type this states that `x = <>` for any `x : Unit'`. This is not the case for the data type `x : Unit`. For binary products it states that `x = < fst' x, snd' x >` for any `x : A ×' B`.

## 5 Inductive families

We are now ready to define the vectors as a dependent data type (inductive family):

```
data Vect (A : Set) : Nat → Set where
 [] : Vect A zero
 _::_ : {n : Nat} → A → Vect A n → Vect A (succ n)
```

Note that this difference is similar to the definition of `List` above. The difference is that `Vect A : Nat → Set` is a family of types for each `A`, whereas `List A` is one type. Moreover, the constructors of `Vect` contain size information: `[]` is a vector of dimension zero, whereas the dimension of the vector `a :: as` is one greater than the dimension of `as`.

Given the size information we can now make `head` a total function by expressing that its argument of must be a vector of length strictly greater than zero:

```
head : {A : Set} → {n : Nat} → Vect A (succ n) → A
head (a :: as) = a
```

## 6 Proving in Agda

We have now come to the point where we that Agda is not only a programming language but also a proof assistant, that is, a system where you can write machine-checked proofs. In traditional logic, a proof is a sequence of formulas, such that each formula is either an axiom or follows from a number of previous formulas by a rule of inference. (In computer science it's actually more natural to see such a proof of a theorem as a tree, where the root is the theorem, the leaves are the axioms, and the inner nodes follow from their parents by a rule of inference.) In Agda and other systems based on the Curry-Howard correspondence there is another notion of proof, where proofs are programs, formulas are types, and a proof is a correct proof of a certain theorem provided the corresponding program has the type of the corresponding formula. Proofs in this sense are often called proof objects, to emphasize the difference with proofs in the usual sense of logic.

This might all be somewhat mysterious to begin with, so let us start with something simple and again use our Booleans and try to prove some properties about Boolean operations. We shall prove that the type Bool with the operations `not`, `_&&_` and `||` form a Boolean algebra. (See wikipedia.)

### 6.1 The identity type

Say that we want to prove the following law of Boolean algebra, the law of double negation:

```
not (not b) = b
```

for an arbitrary `b : Bool`. We can do this by proving the law for b = true and b = false:

```
not (not true) = true
not (not false) = false
```

Both those equations follow by equational reasoning from the two defining equations for `not`:

```
not (not true) = not false = true
not (not false) = not true = false
```

How can we express this reasoning in Agda? Since we shall identify formulas (propositions) and types, the first question is: what type corresponds to the proposition `not (not b) = b`? To this end we use the identity type `Id A a a'`, which contains proofs that `a, a' : A` are identical. Thus the law of double negation for Bool can be written as the Agda-type:

```
(b : Bool) → Id Bool (not (not b)) b
```

that is, we need to write a function which for any `b : Bool` returns an element (proof object) of `Id Bool (not (not b)) b`. The idea behind the identity type is that there is only one way to construct a proof of an identity: a and a' must be identical (after simplification). Hence we define it as a data type with one constructor:

```
data Id (A : Set) (a : A) : A → Set where
 refl : Id A a a
```

This constructor is called refl for "reflexivity of identity", that is, the law that any element `a : A` is identical to itself.

We get a more familiar looking notation if we use the infix sign ≡ and make the type A implicit:

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

Note that there is an important difference between `a ≡ a'`, which is a type, and `a = a'` which is not - it's a relation between terms. The latter expresses that a and a' are equal by definition (definitionally equal).

We can now for example prove that 1 + 1 is equal to 2:

```
oneplusoneistwo : 1 + 1 ≡ 2
oneplusoneistwo = refl
```

Agda accepts that refl : 1+1 ≡ 2 because it first simplifies (normalizes) $1 + 1$ (using the defining equations for $+$) and returns 2. Hence it looks at the type of refl and realizes that we need to instantiate `A` by `Nat` and `a` by 2.

When we case-split (C-c C-c) a proof c : I A a a', Agda replaces c by the only possible constructor refl, and unifies a and a'.

Let's return to the proof of the law of double negation.

```
doublenegation : (b : Bool) → not (not b) ≡ b
doublenegation true = refl
doublenegation false = refl
```

Note that the types of the right hand sides of two respective equations are

```
 not (not true)  ≡  true
not (not false)  ≡  false
```

After simplification (normalization) they become

```
 true  ≡  true
false  ≡  false
```

and hence they are instances of the type of `refl`.

Since `doublenegation` always returns `refl` it is tempting to try to skip the case analysis and simply write

```
doublenegation b = refl
```

However, this does not work (try it!) since `refl` does not have the type `not (not b) ≡ b`. We cannot simplify `not (not b)` when `b` is a variable.

Exercise. Prove some other laws of Boolean algebra. (See wikipedia for the definition.)

## 6.2 Pattern matching on the proof of an identity

We can also prove that propositional identity is a symmetric relation:

```
sym : {A : Set} → (a a′ : A) → a ≡ a′ → a′ ≡ a
sym a .a refl = refl
```

Here the proof is a bit more sophisticated, because now we pattern match on a proof object for identity. As usual we begin with putting a ? on the right hand side:

```
sym a a' p = ?
```

Now we pattern match on `p`. The only constructor for ≡ is `refl` and Agda tries to unify its type with the type of the goal. This forces `a` and `a'` to be identical (definitionally). We get

```
sym a .a refl = { }0
```

This is because when we pattern match on the proof that `a ≡ a'` we force them to be equal. The dot in front of the second `a` indicates that it is forced to be `a` by the unification. The type of the hole is now `a≡ a` so we can fill it with`refl`.

Exercise: prove transitivity by doing pattern matching in both arguments:

```
trans : {A : Set} → {a₁ a₂ a₃ : A} → a₁ ≡ a₂ → a₂ ≡ a₃ → a₁ ≡ a₃
trans refl refl = refl
```

We can also prove the general rule of identity elimination, the rule that states that we can substitute identical elements for each other. If a property is true for a□, then it's also true for any a□ equal to a□

```
subst : {A : Set} → {P : A → Set} → {a₁ a₂ : A} → a₁ ≡ a₂ → P a₂ → P a₁
subst refl q = q
```

A special case of this is that functions map equal inputs to equal outputs

```
cong : {A B : Set} → {a₁ a₂ : A} → (f : A → B) → a₁ ≡ a₂ → f a₁ ≡ f a₂
cong f refl = refl
```

Exercise: prove symmetry and transitivity using subst but without using pattern matching! Similarly, construct `cong` without pattern matching but using subst.

## 7 Proof by induction

We shall now show how to prove a property (associativity of addition) of natural numbers by induction. We first show how to write the proof by pattern matching:

```
associativity−plus : (m n p : Nat) → ((m + n) + p) ≡ (m + (n + p))
associativity−plus m n zero = refl
associativity−plus m n (succ p) = cong succ (associativity−plus m n p)
```

Using propositions as types we can prove the general principle of mathematical induction:

```
natind : {P : Nat → Set}
 → P zero
 → ((m : Nat) → P m → P (succ m))
 → (n : Nat) → P n
natind base step zero = base
natind base step (succ n) = step n (natind base step n)
```

We can use this principle to prove associativity of addition without pattern matching:

```
associativity−plus−ind : (m n p : Nat) → (m + n) + p ≡ m + (n + p)
associativity−plus−ind m n p
 = natind { p → ((m + n) + p) ≡ (m + (n + p))}
  refl
  (( p r → cong succ r))
  p
```

Note that the two proofs are essentially the same - they are just two different ways of expressing the same idea. Pattern matching provides ”syntactic sugar” for definitions by primitive recursion, but it also opens the possibility for more general recursion schemes than primitive recursion.

Exercise. Prove that

```
postulate 0−neutral−right : (m : Nat) → m + 0 ≡ m
postulate 0−neutral−left : (n : Nat) → 0 + n ≡ n
postulate succ−+−right : (m n : Nat) → m + succ n ≡ succ (m + n)
postulate succ−+−left : (m n : Nat) → succ m + n ≡ succ (m + n)
```

Exercise. Prove commutativity of addition:

```
postulate commutativity−plus : (m n : Nat) → m + n ≡ n + m
```

## 8 Propositional logic

We shall now show the correspondence between propositions and sets for propositional logic. This correspondence was discovered by Curry, who noticed that there is a correspondence between some basic combinators in the lambda calculus and some axioms for implication.

For example, the type of the identity combinator

```
I : {A : Set} → A → A
I x = x
```

corresponds to the axiom $A \supset A$, where $\supset$ is implication. The type of the composition combinator

```
B : {A B C : Set} → (B → C) → (A → B) → A → C
B g f x = g (f x)
```

corresponds to the axiom $(B \supset C) \supset (A \supset B) \supset A \supset C$. Similarly the type of the constant combinator

```
K : {A B : Set} → A → B → A
K x y = x
```

corresponds to the axiom $A \supset B \supset A$. And finally, the type of

```
S : {A B C : Set} → (A → B → C) → (A → B) → A → C
S g f x = g x (f x)
```

corresponds to the axiom $(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$.

Finally modus ponens (the rule that says that from $A \supset B$ and $A$ deduce $B$) corresponds to the typing rule for application (from $f : A \supset B$ and $a : A$ deduce $f\ a : B$).

The intuitive idea is that a (constructive) proof $f$ of an implication $A \supset B$ is a function which maps proofs of $A$ to proofs of $B$. In constructive mathematics functions are computable, that is, they are ”programs”.

Furthermore, conjunction corresponds to Cartesian product. The type of the constructor corresponds to the ”introduction rule” for conjunction:

```
data _&_ (A B : Set) : Set where
  <_,_> : A → B → A & B
```

Moreover, the projections correspond to ”elimination rules”

```
fst-& : {A B : Set} → A & B → A
```

```
fst-& < x , y > = x

snd-& : {A B : Set} → A & B → B
snd-& < x , y > = y
```

Similarly disjoint union (sum) corresponds to disjunction and the types of the constructors correspond to introduction rules for disjunction:

```
data _∨_ (A B : Set) : Set where
 inl : A → A ∨ B
 inr : B → A ∨ B
```

Moreover, definition by cases corresponds to proof by cases (the disjunction elimination rule):

```
case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
case f g (inl x) = f x
case f g (inr y) = g y
```

Let us now construct a simple proof, the proof of commutativity of conjunction:

```
comm-∨ : (A B : Set) → A ∨ B → B ∨ A
comm-∨ A B (inl a) = inr a
comm-∨ A B (inr b) = inl b
```

Exercise: Prove commutativity of conjunction!

Finally, we point out that the the unit set corresponds to a trivially true proposition with a single proof object:

```
data ⊤ : Set where
 <> : ⊤
```

Similarly, the empty set is defined as the set with no constructors and corresponds to a trivially false proposition. We write

```
data ⊥ : Set where
```

The elimination rule for ⊥ corresponds to the rule of proof by no case; ⊥ implies any proposition. If we pattern match on the constructors for ⊥ we get no cases. However, rather than simply not writing out any cases at all, Agda writes one line

```
nocase : {C : Set} → ⊥ → C
nocase ()
```

where () indicates the argument which cannot be instantiated in a type-correct way.

Note that there is a general pattern for all the logical connectives, except ⊃, stating that the types of the constructors correspond to the introduction rules. If we want to have this pattern for ⊃ as well, we can alternatively define it as follows:

```
data _⊃_ (A B : Set) : Set where
 ⊃−intro : (A → B) → A ⊃ B
```

Modus ponens is now defined as follows

```
mp : {A B : Set} → A ⊃ B → A → B
mp (⊃−intro g) a = g a
```

Negation is defined as implying the absurd:

```
¬ : Set → Set
¬ A = A → ⊥
```

With this definition neither the law of double negation

$$(A : \mathsf{Set}) \to \neg\,(\neg\, A) \to A$$

nor the law of excluded middle

$$(A\colon \mathsf{Set}) \to A \vee \neg\, A$$

are valid.

However, the inverse of double negation is valid:

```
inverse−dn : (A : Set) → A → ¬ (¬ A)
inverse−dn A a = f → f a
```

This is called the BHK (Brouwer-Heyting-Kolmogorov) interpretation of logic or the Curry-Howard correspondence between propositions as types (sets). Howard showed how to generalize this to predicate logic by introducing dependent types.

Exercise: Prove the following three laws.

(a) the law of triple negation:

$$(A\colon \mathsf{Set}) \to \neg(\neg(\neg A)) \to \neg A$$

(b) excluded middle implies double negation:

$$(A : \mathsf{Set}) \to (A \vee \neg A) \to \neg(\neg A) \to A$$

(c) double negation implies excluded middle:

$$((X\colon \mathsf{Set}) \to \neg(\neg X) \to X) \to (A : \mathsf{Set}) \to (A \vee \neg A)$$

Note the strengthening of the assumption in (c): to prove excluded middle for $A$ it does not suffice to know double negation for $A$ itself, but you must know it for any proposition $X$.

## 9 Predicate logic

Howard introduced dependent types and could thus extend Curry's correspondence between propositions as types (sets) to the quantifiers.

Since a proposition corresponds to a set, a predicate $P$ over a set $A$, corresponds to a "propositional function" $P : A \to \mathsf{Set}$. The universally quantified proposition $\forall x : A.Px$ corresponds to the set $(x : A) \to Px$. Intuitively, a proof of $\forall x : A.Px$ is a function which maps an arbitrary element $a : A$ to a proof of $Pa$.

To emphasize the correspondence between dependent function spaces and universally quantified propositions, Agda allows the notation $\forall (x : A) \to Px$ as an alternative to $(x : A) \to Px$.

You can think of universal quantification (over an infinite set) as infinite conjunction and existential quantification as infinite disjunction.

A proof of an existentially quantified proposition $\exists x : A.Px$ is a pair $<a, p>$ consisting of an element $a : A$ and a proof $p : Pa$. We can define it as a data type with one constructor:

```
data ∃ (A : Set) (P : A → Set) : Set where
 <_,_> : (a : A) → P a → ∃ A P
```

The first projection gives access to the "witness" $a\colon A$.

```
witness : {A : Set} {P : A → Set} → ∃ A P → A
witness < a , p > = a
```

The second projection gives access to the proof that $P$ holds for the witness $a$ (that is, $p : P a$).

```
proof : {A : Set} → {P : A → Set} → (c : ∃ A P) → P (witness c)
proof < a , p > = p
```

The rule of existence elimination is as follows:

```
∃−elim : {A : Set} → {P : A → Set} → {Q : Set} → ((a : A) → P a → Q) → ∃ A P → Q
∃−elim f < a , p > = f a p
```

Exercise: Define uncurrying!

uncurry : A B C : Set → (A → B → C) → A × B → C

and note the similarity with ∃-elim!

Just as there was an alternative version of implication, there is a universal quantifier which is defined as a datatype with a single constructor:

```
data ∀‘ (A : Set) (P : A → Set) : Set where
 ∀′−intro : ((x : A) → P x) → ∀‘ A P
```

Note that this gives variable free-notation ∀‘ $AP$= ∀‘ $A$(λ $x$→ $Px$) for ∀ $x$: $A$. $Px$.

We can also define the rule of universal elimination:

```
∀′−elim : {A : Set} {P : A → Set} → ∀‘ A P → (a : A) → P a
∀′−elim (∀′−intro f) a = f a
```

# 10 Arithmetic

## 10.1 Peano's axioms

The induction axiom is one of Peano's axioms for arithmetic (see wikipedia). Another is the following:

```
peano−4 : (n : Nat) → zero ≡ succ n → ⊥
peano−4 n ()
```

We can in fact prove all Peano's axioms (see wikipedia) from our definition of Nat. Do this as an exercise!