THE AGDA STANDARD LIBRARY

N. P. STRICKLAND

1. Introduction

In this document we give a survey of the structure, organisation and contents of the Agda standard library.

We will mostly ignore foundational issues and use the traditional language of sets and functions rather than talking about type theory. This glosses over some important distinctions, but we feel that they are best discussed separately.

We will generally use Agda notation, which differs from ordinary mathematical notation in several ways.

- We write `X → Y` for the set of functions from `X` to `Y`.
- We write `x : X` to indicate that `x` is an element of `X`. Note that `f : X → Y` means that `f` is a map from `X` to `Y`, as in traditional notation.
- Given `f : X → Y` and `x : X` we write `f x` or `(f x)` (not `f(x)`) for the value of `f` at `x`.
- We interpret `X → Y → Z` as `X → (Y → Z)`. Thus, functions `f : X → (Y → Z)` biject with functions `g : (X × Y) → Z` by a rule that can be written in traditional notation as $g(x, y) = f(x)(y)$. In Agda functions like `f` are easier to handle than functions like `g`, and are almost always preferred.
- In Agda, names of variables and functions can contain punctuation characters and mathematical symbols. The notation `n > m` (with spaces) refers to the set of proofs that `n` is larger than `m`. It is a common idiom to use a name like `n>m` (without spaces) for an arbitrary element of that set.
- Function names often contain underscore characters, which are treated specially: they indicate the placement of the arguments. For example, the addition function $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ is officially called `_+_`, so we can write `(_+_ n m)` to refer to the sum of `n` and `m`. However, because of the placement of the underscores we can also write `n + m` for the same thing.

2. Natural numbers

Natural numbers are introduced in the module `Data.Nat`. There is a standard correspondence between module names and file names, so the relevant code is in the file `Nat.agda` in the `Data` subdirectory of the main directory for the library.

Natural numbers are defined in Peano style. The key block of code is as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ
```

Roughly speaking, this says that zero is a natural number, and that there is a successor function from $\mathbb{N}$ to itself. We call `zero` and `suc` the *constructors* for $\mathbb{N}$.

Note that the full name of the set of natural numbers is `Data.Nat.ℕ`. We can use the short form $\mathbb{N}$ when the module `Data.Nat` has been imported and opened. This will be discussed in more detail in Section 9.

In Peano's original approach to $\mathbb{N}$ we also have induction principles; the simplest version says that if `p zero` is true and `p n` implies `p (suc n)` then `p n` holds for all `n`. It is built in to Agda that we have induction and recursion principles for any type introduced using the `data` keyword and a list of constructors. We therefore do not need to do anything special for $\mathbb{N}$.

2.1. Algebraic operations.

For example, the file `Data.Nat` defines addition of natural numbers as follows:

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

See the Introduction for comments on the notation used here.

Other algebraic operations defined in `Data.Nat` include the following.

- The truncated predecessor map `pred : ℕ → ℕ`, which is $\mathrm{pred}(n) = \max(n-1, 0)$ in traditional notation.
- The truncated subtraction operation `n ∸ m`, which is $\max(n-m, 0)$ in traditional notation.
- Multiplication, written as `n * m`.
- Maximum and minimum, written as `n ⊔ m` and `n ⊓ m`.

Properties of these operations (such as commutativity and associativity of addition) are not proved in `Data.Nat`, but in the module `Data.Nat.Properties` which is defined in the file `Data/Nat/Properties.agda`. That module defines terms such as

```
Data.Nat.Properties.isCommutativeSemiring.+-assoc
```

which is a proof that addition is associative. The definition of what it means for an operation to be associative is in `Algebra.FunctionProperties`, and the definition of a commutative semiring is in `Algebra.Structures`. There is quite a lot to be said about how all this information is organised and how one can refer to it efficiently. This is discussed in more detail in Section 7.

The module `Data.Nat.Properties` also proves that `(ℕ,⊔,⊓)` is a distributive lattice and a commutative semiring without identity.

2.2. **Order.** The module `Data.Nat` also contains information about the ordering on ℕ. As usual in Agda, the notation `m ≤ n` (which is a synonym for `(_≤_ m n)`) refers to a set rather than just a truth value; the elements of the set can be thought of as proofs that `m` is less than or equal to `n`. These sets are defined recursively as follows.

(a) For any natural number `n` we can construct a proof that `zero ≤ n`;
(b) Whenever we have a proof that `m ≤ n`, we can construct from it a proof that `suc m ≤ suc n`.

To explain (a) in more detail, there is a function called `z≤n` that takes any natural number `n` to a proof that `zero ≤ n`. [1] Here `n` is an *implicit argument* to the function `z≤n`. The point is that whenever we use the function `z≤n`, the Agda typechecker will already know what type the value of the function needs to have, and that determines `n`. Thus, the parameter `n` is given in curly brackets when `z≤n` is declared, which means that we can just write `z≤n` with no argument.

Similarly, in (b) we actually have a function called `s≤s`, which takes three arguments: a natural number `m`, a natural number `n`, and a proof that `m ≤ n`. In accordance with the standard Agda idiom, the third argument is called `m≤n` (with no spaces, so it counts as a single symbol). The first two arguments can be deduced from the context so they are declared to be implicit; only the last argument needs to be given explicitly.

The set `m ≤ n` is defined using the `data` keyword and constructors `z≤n` and `s≤s`. We therefore automatically have a recursion principle: we can define a function `f` from `m ≤ n` to another set by specifying the composite of `f` with the constructors. Some restrictions are necessary to ensure that this recursion terminates, but Agda will automatically reject any definition where it is unable to convince itself that these restrictions are satisfied.

The module `Data.Nat` also defines various the relations `m ≥ n`, `m < n`, `m ≰ n` and so on in terms of `m ≤ n`. Agda implicitly uses a kind of constructive logic in which the law of the excluded middle does not automatically hold. Because of this, it is not immediate that `m ≰ n` is equivalent to `m > n`. However, this does in fact hold, and is partially proved in `Data.Nat.Properties`. In more detail, that module defines a map named `≰⇒>` from `m ≰ n` to `m > n`, which can be interpreted as a proof that the former relation implies the latter. It would not be hard to prove the converse, but I think that it is not done in the standard library.

Another manifestation of the same idea is the function `_≤?_` defined in `Data.Nat`. For natural numbers `m` and `n`, the value `m ≤? n` will either have the form `(yes p)` (where `p` is a proof that `m ≤ n`) or `(no q)` (where `q` is a proof that `m ≰ n`). This is useful for defining functions by cases depending on the order. All

[1] Alternatively, we can say that `z≤n` takes any natural number `k` to a proof that `zero ≤ k`; the letter `n` in the name `z≤n` is only suggestive. In some contexts this can cause confusion, but any confusion can usually be cured using Agda's renaming mechanism, which will be discussed later.

this relies on the framework of decidable binary relations established in the module `Relation.Binary`. The same module also gives basic definitions about partial and total orders.

The module `Data.Nat.Properties` also proves a long list of other small lemmas about the algebraic and order theoretic properties of ℕ, such as the following:

- If `m ≤ n` then `m ≤ k + n`.
- `m ≤ m + n` and `n ≤ m + n`. (Because the proof of commutativity of addition needs to invoked explicitly whenever used, it is helpful to have both of these statements proved separately.)
- `m ⊓ n ≤ m` and `m ≤ m ⊔ n`.
- If `i + j ≡ i + k` then `j ≡ k`. (Here and elsewhere in Agda, the notation `x ≡ y` is used for the proposition that `x` equals `y`, or equivalently the set of proofs that `x` equals `y`, whereas we write `x = y` when defining `x` to be `y`.)

2.3. **Divisibility.** Various aspects of elementary number theory are treated in the modules `Data.Nat.Coprimality`, `Data.Nat.DivMod`, `Data.Nat.Divisibility`, `Data.Nat.GCD`, `Data.Nat.LCM` and `Data.Nat.Primality`.

The module `Data.Nat.Divisibility` defines the divisibility relation `m ∣ n`. This is an inductive type with a single constructor called `divides`. If `q` is a natural number and `eq` is a proof that `n ≡ q * m` then `(divides q eq)` is an element of `m ∣ n`. Here `m` and `n` are implicit arguments to the constructor. It is proved that the divisibility relation gives a decidable partial order on ℕ (using the general framework of partially ordered sets from `Relation.Binary`), and there are various lemmas about the relationship between divisibility and the algebraic operations.

The module `Data.Nat.DivMod` is about division with remainder. When `n` and `d` are natural numbers with `d` nonzero, it defines an element +(n divMod d)+. Roughly speaking, this is the unique pair `(q , r)` where `0 ≤ r < d` and `r + q * d ≡ n`. More precisely, the module defines a set `(DivMod n d)` whose purpose is to act as a codomain for the function `_divMod_`. This set has a single constructor called `result`. If `q` and `r` are natural numbers and `r < n` then `(result q r)` is by definition an element of `(DivMod n d)`, where `d` is an implicit argument and `n` is defined to be `r + q * d`.

As a slight wrinkle in this picture, the element `r` is not actually an element of ℕ, but instead an element of the set `(Fin d)` discussed in Section 4.

The module also defines a variant `(n divMod' d)` which encapsulates the same information in a slightly different form. We will not spell out the difference here except to say that this variant is often more convenient.

If we just want the quotient `q` or the remainder `r` separately, we can use the functions `(n div d)` and `(n mod d)`, which are defined in terms of `(n divMod' d)`.

We also mention that the constraint `d > 0` in the definition of `(n divMod d)` is enforced by declaring an implicit third parameter for the `_divMod_` function, like this:

```
_divMod_ : (n : ℕ)  (d : ℕ)  {≢0 : False (d ≟ 0)} → DivMod n d
```

The word `False` here is not a primitive part of the language but is defined in `Relation.Nullary.Decidable`. The mechanism used here will be explained in more detail **where?**.

The modules `Data.Nat.GCD`, `Data.Nat.LCM` and `Data.Nat.Coprimality` contain material related to greatest common divisors and least common multiples. Most of the real work is in the GCD module, and the other two modules merely shuffle information around. The main job is to define a function that computes (for each `m` and `n`)

- A natural number `d`.
- A proof that `d` is a common divisor of `m` and `n`.
- A proof that every common divisor of `m` and `n` also divides `d`.
- Natural numbers `x` and `y`.
- Either a proof that `d + y * n ≡ x * m`, or a proof that `d + x * m ≡ y * n`.

(For the last two points, it would be more usual to ask for integers `x` and `y` with a proof that `d ≡ x * n + y * m`, but we need to work around the fact that negative numbers have not yet been introduced.) The file `GCD.agda` defines a set `(Lemma m n)`, such that an element of this set is essentially a system of data as above. This is actually done in a separate module, so the fully qualified name is `Data.Nat.Bézout.Lemma` rather than just `Lemma`. (**Should that be** `Data.Nat.GCD.Bézout.Lemma`? **I am a bit confused about the scoping**

rules here.) The problem is thus to define an element `(lemma m n)` in `(Lemma m n)` for all `m` and `n`. This is done recursively by a variant of Euclid's algorithm. As ingredients, there are functions `base`, `refl`, `sym`, $\texttt{step}^{\texttt{l}}$ and $\texttt{step}^{\texttt{r}}$ which can be described loosely as follows:

- `base` constructs an element of `(Lemma 0 n)`, showing that `n` is the gcd of `0` and `n`.
- `refl` constructs an element of `(Lemma n n)`, showing that `n` is the gdc of `n` and `n`.
- `sym` maps `(Lemma m n)` to `(Lemma n m)`.
- $\texttt{step}^{\texttt{l}}$ maps `(Lemma m n)` to `(Lemma m (m + n))` (provided that `n > 0`). The function $\texttt{step}^{\texttt{r}}$ is similar.

We can define `(lemma m n)` by applying these functions repeatedly. Officially, this is a definition by recursion over the set $\mathbb{N} \times \mathbb{N}$ with its lexicographic order. This is handled as a special case of a very general framework for recursive definitions that is set up in the modules `Induction`, `Induction.Lexicographic` and `Induction.Nat`. The details are somewhat complicated and will be discussed in Section 8.

The module `Data.Nat.Primality` defines the set `(Prime n)` of proofs that `n` is prime. It also defines a function `(prime? n)` which returns either an element of the form `(yes a)` (where `a` is a proof that `n` is prime) or an element of the form `(no b)` (where `b` is a proof that `n` is not prime). In other words, this is a decision procedure for primality, using the framework of `Relation.Unary.Decidable`.

2.4. **Binary representation.** The module `Data.Bin` defines a set `Bin` which is essentially the same as $\mathbb{N}$ but with elements represented essentially as lists of binary digits. The module defines addition, multiplication and an order relation on `Bin`, together with conversion functions relating it to $\mathbb{N}$.

3. Other numbers

3.1. **Integers.** The set $\mathbb{Z}$ is defined in `Data.Integer`. It has two constructors, corresponding to the inclusion of $\mathbb{N}$ in $\mathbb{Z}$ and the map $\mathbb{N} \to \mathbb{Z}$ given by $n \mapsto -n - 1$.

The same module defines algebraic operations on integers, the order relation, decision procedures for ordering and equality, and miscellaneous functions relating $\mathbb{Z}$ to $\mathbb{N}$ and the set `Sign = {+ , -}` of signs (defined in `Data.Sign`). The module `Data.Integer.Properties` constructs an element

```
Data.Integer.Properties.isCommutativeRing
```

which is a package of proofs of all the commutative ring axioms for $\mathbb{Z}$. For example, the element

```
Data.Integer.Properties.isCommutativeRing.*-comm
```

is a proof that multiplication is commutative. This relies on the framework established in the module `Algebra.Structures`. Some of the work is delegated to `Data.Integer.Addition.Properties` and `Data.Integer.Multiplication.Properties`. There are also proofs of a few additional properties (such as cancellation laws) that have not been incorporated in `Algebra.Structures`.

3.2. **Rational numbers.** The module `Data.Rational` defines the set $\mathbb{Q}$ of rational numbers (which are essentially represented in the form $a/(b+1)$, with $a \in \mathbb{Z}$ and $b \in \mathbb{N}$ with $\gcd(|a|, b+1) = 1$). This must be regarded as a stub: no algebraic or order operations are defined or analysed.

3.3. **Boolean values.** The module `Data.Bool` defines the set `Bool`. It has two constructors with no arguments, denoted `true` and `false`. Thus, in traditional terms, it corresponds to the set `{ true, false }`. The module defines the usual logical operations (denoted by `not a`, `a ∧ b`, `a ∨ b` and `a xor b`), and a decision procedure for equality (written $\texttt{a} \stackrel{?}{=} \texttt{b}$). There is also a function called `if_then_else_`, which involves a more elaborate use than we have seen previously of the rules for underscores in function names. The expression `(if a then x else y)` is a synonym for `(if_then_else_ a x y)`; it evaluates to `x` if `a` is true, and to `y` if `a` is false.

It is a general feature of Agda that propositions are not usually represented by elements of `Bool`, but instead by sets, with the elements of the set being thought of as proofs of the proposition. Because of this it is useful to have a function `T : Bool → Set` which sends `false` to the empty set and `true` to a set with one element. The empty set is denoted by $\bot$ and is introduced in `Data.Empty`. The singleton set is denoted by $\top$ and is introduced in `Data.Unit`. Note that $\top$ is a special symbol, entered as `\top` in LaTeX, not the letter `T`. The unique element of $\top$ is denoted by `tt`.

The module `Data.Bool.Properties` shows how `Bool` can be regarded as a commutative semiring or a commutative ring or a Boolean algebra under various combinations of the usual operations. It also proves miscellaneous other easy properties.

3.4. **Etc.** There is currently no implementation of real or complex numbers, or the rings $\mathbb{Z}/n\mathbb{Z}$.

4. Finite sets

Given a natural number `n`, we often want to consider the set `Fin n` of natural numbers that are smaller than `n`. One way to encode this in Agda would involve something like the set of pairs `(k , p)`, where `k` is a natural number and `p` is an element of the set `k < n`, or in other words a proof that `k` is less than `n`. This would generalise easly to cover other subsets of $\mathbb{N}$. However, it turns out to be convenient to use a more condensed model for `Fin n`, which is defined in `Data.Fin`.

This is a convenient point to explain a little about Agda's facilities for import and renaming. The module `Data.Fin` needs to use various things coming from `Data.Nat`. Normally we would just include a line in `Fin.agda` saying

```
open import Data.Nat
```

which would then allow us to use all notation from `Data.Nat`. However, in the present context that would cause notational clashes. The point is that we will want to introduce an order relation on the set `Fin n`. Although there are functions that convert between elements of `Fin n` and natural numbers, they are not exactly the same thing, so we cannot just use the order relation on $\mathbb{N}$ from `Data.Nat`. In order to distinguish between the two order relations, we have the lines

```
open import Data.Nat as Nat
  using (ℕ; zero; suc; z≤n; s≤s)
  renaming ( _+_ to _N+_; _∸_ to _N∸_
           ; _≤_ to _N≤_; _≥_ to _N≥_; _<_ to _N<_; _≤?_ to _N≤?_)
```

This allows us to use the notation `x N≤ y` for the order relation on $\mathbb{N}$, and reserve the symbol `x ≤ y` for the new order relation on `Fin n`.

5. Data structures

5.1. **Maybe.** The set `Maybe A` is essentially the set `A` with an extra element adjoined. It is typically used as the return type for functions that usually produce an element of `A` but may fail to do so in exceptional cases, when the extra element is returned instead. For any element `a : A`, the corresponding element of `Maybe A` is denoted by `just a`; the extra element is denoted by `nothing`. Thus, the set `Maybe A` has two constructors, `just : A → Maybe A` and `nothing : Maybe A`.

For technical reasons the set `Maybe A` is defined in `Data.Maybe.Core`, but most of the associated functions and properties are in `Data.Maybe`.

Another technicality here is that we need to take account of levels. These are a mechanism used by Agda to avoid Russell-type paradoxes, which will be discussed in Section 10. Every set has a level, which is an element of a set called `Level` introduced in a file called `Level.agda` in the top directory of the standard library. If `A` is a set of level `a` then `Maybe A` also has level `a`. The definition of `Maybe` takes the level `a` as an implicit argument.

Most other definitions of data structures discussed in this section also depend on one or more sets of various different levels, and the relevant levels are treated as implicit arguments. One needs a minimal understanding of this issue to *read* the definitions of the data structures, but when *using* the definitions all the levels become invisible.

- Discuss `Maybe` as a monad.
- Discuss functions related to `Maybe`.

5.2. **Products and coproducts.** Given two sets `A` and `B`, the module `Data.Sum` defines a new set `A ⊎ B` with two constructors: a map $\mathtt{inj}_1$ `: A → A ⊎ B` and a map $\mathtt{inj}_2$ `: B → A ⊎ B`. It is built into the foundations of Agda that a map out of `A ⊎ B` is freely and uniquely determined by the composite with the two constructors, so `A ⊎ B` can be thought of as the disjoint union (or coproduct) of `A` and `B`.

If we have maps `f : A → C` and `g : B → C` then the combined map `A ⊎ B → C` is denoted by `[f , g]′`. The unprimed notation `[f , g]` is used for a slightly more general concept in which we do not have a single set `C` but instead a family of sets depending on a point in `A ⊎ B`.

Next, as well as the above binary coproduct, we also have a coproduct construction for indexed families. In more detail, suppose we have a set `A`, and a family of sets `B x` for all `x : A`. We then have a set `Σ A B`, whose elements are pairs `(x , y)` with `x` in `A` and `y` in `B x`. In other words `Σ A B` is the disjoint union of all the sets `B x`. The definition of `Σ` in `Data.Product` includes the line

```
constructor _,_
```

which allows us to use traditional ordered pair notation to construct elements of `Σ A B`. The projections sending `(x , y)` to `x` and `y` are denoted by $\texttt{proj}_1$ and $\texttt{proj}_1$, respectively.

If `B` is just a single set then we can apply `Σ` to the corresponding constant family, and the result is just the set of pairs `(x , y)` with `x` in `A` and `y` in `B`, or in other words `A × B`. In Agda, the product `A × B` is actually *defined* by this mechanism.

The module `Data.Product` also introduces the notation `∃ B` for `Σ A B` (with `A` treated as an implicit argument, along with the levels of the sets `A` and `B` which we have not bothered to mention.) This makes sense in terms of the interpretation of propositions as sets, as we will discuss in more detail later.

There are various natural maps involving sets of the form `Σ A B` that are also introduced in `Data.Product`. We will not attempt to summarise them here.

5.3. **Lists.**

5.4. **AVL trees.**