Euclid's theorem in Agda

February 29, 2012

This document describes a proof in Agda that there are infinitely many primes, or more precisely, that for any natural number n there exists a prime p such that $p > n$. We have explained the code in great detail, as an introduction to Agda for mathematicians.

As Agda is based on intuitionistic logic, we can only prove the existence of p by constructing it, in some sense or other. Our approach is as follows:

(a) We prove that for any natural number $m > 1$ there is a smallest natural number $p > 1$ such that p divides m, and that this p is prime.

(b) Given an arbitrary natural number n we show that the number $m = n! + 1$ is larger than 1, so we can apply (a) to get a prime p dividing m.

(c) We then prove that this p is larger than n. In more detail, we suppose that $p \leq n$, and then note that p divides both $n!$ and $n! + 1$, so p divides 1, so $p = 1$, which is a contradiction because 1 is not prime. This proves the double negation of the desired inequality $p > n$, which is enough even in Agda's constructive framework because the order relation on $\mathbb{N}$ is decidable.

The proof relies on a number of modules in the Agda standard library, and some new modules written for this project. The basic definitions of Peano arithmetic are set up in the Agda standard library (specifically, in the module `Data.Nat` and its submodules). Facts about divisibility and primes are proved in `Data.Nat.Divisibility` and `Data.Nat.Primality`. The new modules are as follows:

(1) The module `Extra.Data.Nat.LargerPrime` defined in this file assembles the ingredients mentioned below to prove the main result.

(2) The module `Extra.Data.Nat.LeastDivisor` implements step (a) in the above outline. It relies on a more general framework for finding least elements defined in the module `Extra.Data.Nat.Minimiser`.

(3) The module `Extra.Data.Nat.Primality` proves some basic facts about primes that are not covered by the standard library module `Data.Nat.Primality`.

(4) The module `Extra.Data.Nat.Factorial` defines the factorial function and proves some basic properties.

We start by declaring the module:

```
module Extra.Data.Nat.LargerPrime where
```

Modules provide a framework for packaging together related definitions. In ordinary mathematical writing it is common to quote results from different sources with incompatible notational conventions; one then has to redefine some of the notation in the sources to avoid ambiguity or clashes. Agda modules provide reasonably good support for maneuvers of this kind.

We next list some standard library modules whose results we need to use. Note that we use the keyword `open` as well as `import`. If we just say `import Data.Nat.Primality` then we can use the definition `Prime`

given in the module `Data.Nat.Primality` but we need to refer to it as `Data.Nat.Primality.Prime`. If we then say `open Data.Nat.Primality` then we can use the short form `Prime` rather than `Data.Nat.Primality.Prime`. The line `open import Data.Nat.Primality` is equivalent to `import` followed by `open`. The code for the module `Data.Nat.Primality` is in the file `src/Data/Nat/Primality.agda` in the standard library. The location of the standard library depends on the details of your installation. If you already have Agda working under emacs then the command `C-h v agda2-include-dirs` should tell you where to look.

```
open import Data.Empty
open import Data.Nat
open import Data.Nat.Properties
open import Data.Nat.Divisibility
open import Data.Nat.Primality
open import Relation.Nullary.Core
open import Relation.Binary.PropositionalEquality
```

The module `Data.Empty` defines the empty set. The next four modules should be self-explanatory. The module `Relation.Nullary.Core` contains the negation operator and some auxiliary definitions for decidable relations that will be discussed later. The module `Relation.Binary.PropositionalEquality` contains some code that makes it easier to formalise proofs that use chains of equalities, like

$$\begin{aligned} a &= b && \text{by Lemma 1} \\ &= c && \text{by Lemma 2} \\ &= d && \text{by Lemma 3.} \end{aligned}$$

It also contains many things that will not concern us for the moment.

Next, we will need to use the fact that addition of natural numbers is commutative. In order to keep the standard library tidy, this is not represented as an isolated fact, but is packaged with various other facts in a proof that $\mathbb{N}$ is a commutative semiring. We therefore need to import the definition of a semiring to gain access to the fact that we need. We use the following code:

```
open import Algebra.Structures
open IsCommutativeSemiring isCommutativeSemiring using ( +-comm )
```

The effect is that the name `+-comm` now refers to a proof that $n + m = m + n$ for all $n, m \in \mathbb{N}$. (Note that names in Agda can use almost any character other than a space. It is common for names to include mathematical symbols or other punctuation.)

The details take some explanation, which could be skipped by the impatient. Firstly, `IsCommutativeSemigroup` (with a capital `I`) is a function of eight arguments that is defined in the module `Algebra.Structures`. The first three arguments are given in curly brackets in the function definition, which means that in most circumstances these arguments need not be given explicitly because they can be deduced from the remaining arguments. The first two arguments are levels, which are used by Agda to avoid Russell-type paradoxes. One can usually ignore these completely. The third (implicit) argument is a set A. The remaining (explicit) arguments are a relation on A (written as $\approx$), two binary operations on A (written as addition and multiplication) and two elements $0, 1 \in A$. The value of the function `IsCommutativeSemigroup` at these arguments is, roughly speaking, the set of proofs that $\approx$ is an equivalence relation and that $A/\approx$ is a commutative semiring under the given operations. Of course this set could be empty, if the operations do not in fact make $A/\approx$ into a commutative semiring. (For reasons that we will not discuss here, algebraic structures are always represented in Agda in this fashion, with operations defined on a set A, and axioms that hold only modulo a specified equivalence relation on A.)

Next, the natural numbers zero and one, and the operations of addition and multiplication of natural numbers, are defined in `Data.Nat`. We also have the relation of equality on $\mathbb{N}$. Feeding these ingredients into the function `IsCommutativeSemiring` we get the set P of proofs that $\mathbb{N}$ is a commutative semiring. The module `Data.Nat.Properties` constructs such a proof, which is a particular element called `Data.Nat.Properties.isCommutativeSemiring` (with a lower case `i`) of the set P. As we

have imported and opened the module `Data.Nat.Properties`, we can omit the prefix and use the name `isCommutativeSemiring`.

We described `IsCommutativeSemiring` as a set-valued function, which is essentially correct, but in more technical terms it is a parametrised record type. Agda automatically creates an associated module for each parametrised record type. The statement

```
open IsCommutativeSemiring isCommutativeSemiring using ( +-comm )
```

is a variant of the operation of opening a module, which we have seen before. The proof `isCommutativeSemigroup` is a package of proofs of all the different axioms for a commutative semiring, and the effect of the above `open` statement is to enable us to refer to the part of the package called `+-comm`, which is a proof that addition is commutative. This process of opening a record is similar to the usual practice of saying "consider a semiring R" and then taking all semiring-related notation to refer to R unless otherwise specified.

We need one more module from the standard library, called `∣-Reasoning`. This is defined as a submodule of `Data.Nat.Divisibility`, and it makes it easier to reason with chains of divisibility statements, analogous to the chains of equalities that we described previously.

```
open ∣-Reasoning
```

We also import three new modules that we mentioned previously:

```
open import Extra.Data.Nat.LeastDivisor
open import Extra.Data.Nat.Primality
open import Extra.Data.Nat.Factorial
```

We next specify what we hope to construct. In Agda, whenever we construct an object with certain properties, we need to package it together with a collection of proofs that it does indeed have those properties. In our case we construct a prime p that is larger than n, and we need to remember a proof of that inequality. In Agda, the proposition that p is greater than n is denoted by `p > n`, with spaces between the symbols. Like all propositions in Agda, this is a type, whose terms (if any) are the proofs that $p > n$. The string `p>n` (with no space) is a valid variable name in Agda, and it is a standard idiom to use this name to refer to an arbitrary term of type `p > n`. With this notation, we hope to construct a record consisting of a natural number p, a proof (denoted by `p-prime`) that p is prime, and a proof (denoted by `p>n`) that $p > n$. The next block of code defines `LargerPrime n` (with a capital L) to be the set of such records.

```
record LargerPrime (n : ℕ) : Set where
  field
    p : ℕ
    p-prime : Prime p
    p>n : p > n
```

We now want to define a function `largerPrime` (with a lower case l) which constructs an element `largerPrime n` of `LargerPrime n` for each natural number n. In Agda's notation, this means that `largerPrime` is a term of type `(n : ℕ) → LargerPrime n`. Agda's type checking system is strong enough that to ensure that any term that it accepts as having the above type gives a valid proof of Euclid's theorem.

The next few lines are housekeeping:

```
largerPrime : (n : ℕ) → LargerPrime n
largerPrime n = record {
    p = p ;
    p-prime = p-prime ;
    p>n = p>n
  } where
```

The rest of the code is indented at least as much as the keyword `where`, which means that it is treated as a single block. The above lines tell Agda that this block will define `p`, `p-prime` and `p>n` with the advertised types.

We next invoke the `leastDivisor` function defined in the module `Extra.Data.Nat.LeastDivisor`.

```
L = leastDivisor (1 + (n !)) (s≤s (n!≥1 n))
```

The `leastDivisor` function takes two arguments: a natural number and a proof that that number is at least two. For the number we take $1 + (n!)$. The module `Extra.Data.Nat.Factorial` defines a function `n!≥1` such that `n!≥1 n` is a proof that `1 ≤ n!`, and `Data.Nat` defines a function `s≤s` that converts proofs of `a ≤ b` to proofs of `1 + a ≤ 1 + b`. We can thus use `s≤s (n!≥1 n)` as the second argument to `leastDivisor`, and we define `L` to be the value returned by this function. Now `L` is a record with seven fields. There is a field called `p` whose value is a natural number, a field called `p-prime` whose value is a proof that `p` is prime, and a field called `p-least` whose value is a proof that no number with $2 \le k < p$ divides $1 + (n!)$.

There is also a field called `p|n`. The name of this field is designed to make sense when the first argument to `leastDivisor` is called `n`. Its value in our context is in fact a proof that p divides $1 + (n!)$, not a proof that p divides n.

We now open the record `L`:

```
open LeastDivisor L renaming ( p≤n to p≤1+n! ; p|n to p|1+n! )
```

This allows us to refer to the fields in `L` by their short names, except that we have given two of the fields new names that make more sense in our context. In particular, this defines `p` and `p-prime` as required, so we just need to define `p>n`.

We first declare the type of `p>n`:

```
p>n : p > n
```

We next have a `with` clause:

```
p>n with (suc n) ≤? p
```

Here `suc` is the successor function, taking n to $1 + n$. Next, there is a function called `_≤?_` defined in `Data.Nat`. This is an example of a useful notational mechanism in Agda: when the name of a function involves underscores, the position of those underscores indicates how the arguments are placed relative to the function symbol. Because the name `_≤?_` starts and ends with an underscore, we can write `n ≤? m` for the value of the function at a pair of natural numbers `n` and `m`. That value will either have the form `yes r` (where `r` is a proof that $n \le m$) or `no q` (where `q` is a proof that $n \not\le m$). (These are terms of the type `Dec (n ≤ m)`, which is defined in `Relation.Nullary.Core`.) The clause `p>n with (suc n) ≤? p` indicates that we will define `p>n` using two separate cases, depending on the value of `(suc n) ≤? p`. If this value has the form `yes r` then `r` must be a proof that `p > n` and we can just take `p>n` to be `r`. (This relies on the fact that `p > n` is *by definition* the same as `(suc n) ≤ p`, as we see by reading the code for `Data.Nat`.) This is implemented by the following line:

```
p>n | yes r = r
```

Now what if `(suc n) ≤? p` has the form `no q` for some `q`? Here `q` would have to be a proof that `p ≤ n`, which cannot actually happen. Nonetheless, Agda forces use to deal with this hypothetical case, and prove that it is vacuous. This works as follows. Agda notation for the empty type is `⊥`. For any set `W` there is a term `⊥-elim` of type `(⊥ → W)`. Logically, this corresponds to the principle that a falsity implies everything; set-theoretically, it corresponds to the fact that the empty set is initial in the category of sets. (The set `W` is an implicit argument to the `⊥-elim` function, given in curly brackets when that function is defined in `Data.Empty`.) Next, the function `Prime` defined in `Data.Nat.Primality` sends a natural number n to the set of proofs that n is prime. When n is given as a double successor, the set `(Prime n)` is defined in an obvious way in terms of divisibility properties. However, the two initial cases $n = 0$ and $n = 1$ are handled

separately, with `Prime 0 = Prime 1 = ⊥` by definition. We handle the `no q` case by constructing from `q` a proof that 1 is prime, or in other words a term `1-prime` of type `Prime 1 = ⊥`. We then apply `⊥-elim` to `1-prime` to get a term of whatever type we want; in this case, a term of type `(p > n)`. The line

```
p>n | no  q = ⊥-elim 1-prime where
```

indicates that we will handle the `no q` case by the general approach discussed above, and promises that the following block of code will eventually define `1-prime`. Note that the type checker knows what type `p>n` is supposed to have, and supplies this type as the implicit argument to `⊥-elim`. It also knows that `1-prime` must have type `⊥` in order to be an acceptable explicit argument to `⊥-elim`, and when we get to the definition of `1-prime` it will check that we have indeed provided a term of type `⊥`.

The rest of the code forms another `where` block nested inside the `where` block that we introduced earlier, so it is indented by two more spaces.

We now construct a proof that `p` divides `n !`. The module `Data.Nat.Factorial` defines a function `k|n!` that takes a pair of inequalities `1 ≤ k` and `k ≤ n`, and returns a proof that `k` divides `n !`. We have a proof called `p-prime` that `p` is prime, and the function `prime≥1` defined in `Extra.Data.Nat.Primality` can be used to convert this to a proof that `1 ≤ p`. By hypothesis we have a proof called `q` that `p > n` is false, and using the functions `≰⇒>` and `≤-pred` from `Data.Nat` we convert this to a proof that `p ≤ n` is true. After feeding these ingredients to the function `k|n!` we obtain the required proof that `p` divides `n !`. We denote this proof by `p|n!`.

```
p|n! : p ∣ n !
p|n! = k|n! (prime≥1 p-prime) (≤-pred (≰⇒> q))
```

We now need a proof that `p` divides `n ! + 1`. We have already extracted from the record `L` a proof (denoted by `p|1+n!`) that `p` divides `1 + n !`. Although `n ! + 1` is provably equal to `1 + n !`, these two terms have a different internal representation so we cannot just use `p|1+n!` as a proof that `p ∣ n ! + 1`. However, we also extracted a function called `+-comm` from `Data.Nat.Properties.isCommutativeSemiring`, and `(+-comm 1 (n !))` is a proof that `1 + n ! ≡ n ! + 1`. Using notation established in the module `∣-Reasoning` we can string these proofs together:

```
p|n!+1 : p ∣ n ! + 1
p|n!+1 = begin p ∣⟨ p|1+n! ⟩ 1 + (n !) ≡⟨ +-comm 1 (n !) ⟩ (n !) + 1 ■
```

We next use the function `∣-∸` defined in `Data.Nat.Divisibility`. Given proofs that `i` divides `n + m` and `n`, this returns a proof that `i` divides `m`. Thus, we can pass in our proofs that `p` divides `n !` and `n ! + 1`, and get back a proof that `p` divides `1`.

```
p|1 : p ∣ 1
p|1 = ∣-∸ p|n!+1 p|n!
```

The module `Data.Nat.Divisibility` also defines a function called `∣1⇒≡1`, which can be used to convert our proof that p divides 1 to a proof that $p = 1$.

```
p≡1 : p ≡ 1
p≡1 = ∣1⇒≡1 p|1
```

We conclude by using the function `subst` defined in `Relation.Binary.PropositionalEquality.Core`. Roughly speaking, this works as follows. The first argument is a set `A` depending on a parameter `x`. The second argument is a proof of `u ≡ v`, where `u` and `v` are possible values of `x`. The third argument is an element of the set `(A u)`, and the return value is the same thing regarded as an element of `(A v)`. We have a proof that `p ≡ 1` and a proof that `p` is prime, so we can use `subst` to produce a proof that `1` is prime.

```
1-prime : Prime 1
1-prime = subst Prime p≡1 p-prime
```

As discussed previously, this element `1-prime` can be fed into `⊥-elim` to give a tautological proof of our theorem in the vacuous case where `p ≤ n`.