Table of Contents

Introduction

About this tutorial

Welcome to *Learn You an Agda and Achieve Enlightenment!* If you’re reading this, you’re probably curious as to what Agda is, why you want to learn it, and in general what the big deal is about dependently typed, purely functional programming.

Inspired by BONUS, the writer of Learn You a Haskell, I decided that I should write an approachable Agda tutorial that would introduce dependently typed programming to ordinary people rather than Ivory Tower Academics. Of course, seeing as *I* am one of those Ivory Tower Academics, this might not be easy. I am, however, prepared to give it a try. Learning Agda was a very rewarding but very difficult process for me. It is my hope that, by writing this tutorial, it will become a little bit easier for everyone else.

Step One: Learn Haskell

(The original tutorial suggested that Haskell should be learned before Agda. However, if one already knows some functional programming, then Agda should not be very hard to learn. This should be especially true for those with some background in logic and experience with other dependently typed languages.)

This tutorial is not aimed at those who are completely new to functional programming. Agda is similar on a basic level to typed functional languages such as Haskell and ML, and so knowing a language in the ML family will certainly make learning Agda a great deal easier.

If you don't know a statically typed functional language, I recommend that you learn Haskell, as Agda has a close relationship with the Haskell ecosystem. If you're looking for a good Haskell tutorial, look no further than this book's companion, Learn You a Haskell.

If you don't know how purely functional programming works, learn a little of it before trying to tackle Agda.

Understanding of imperative and object oriented programming (C, Java, Ruby..) isn't necessary. In fact, trying to apply skills learned from these languages might even be harmful when you're trying to learn Agda.

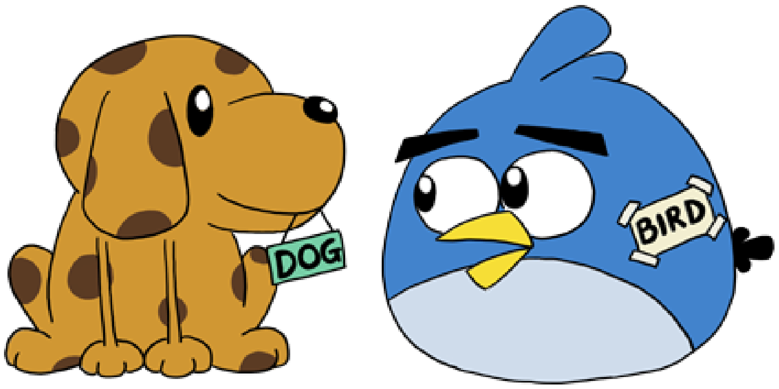
The moral of the story is: keep an open mind. A lot of Agda's power comes from features that are at first difficult to understand. It took a long time for everything in Agda to fall into place in my head. Agda is *hard*. After some time, though, Agda's inherent awesomeness comes to the fore, and it all just clicks. If you encounter obstacles in your Agda learning, don't be discouraged! Keep working, and eventually you will be a master of Agda fu.

What is Agda, anyway?

Agda is a programming language, but not a programming language like Java. It's not even very much like Haskell, although it's a lot more like Haskell than Java.

Agda is a programming language that uses *dependent types*. Many of you would be familiar with types from imperative languages such as Java or C++, and if you're reading up to this point, you should also have a familiarity with types from Haskell.

Types in these languages essentially annotate expressions with a tag. At a simple level, an expression's type might just be a concrete type, like `Bool` or `Int`. Java (through generics), C++ (through templates) and Haskell all support polymorphic types as well, such as `List a` or `Map k v`.



But, if `List a` is a type, then what exactly *is* just `List` (without the parameter)? Haskell calls it a "type constructor", but really it's a *function* at the type level. `List` takes in a type, say `Int`, and returns a new type, `List Int`. Haskell (with appropriate extensions) even supports arbitrary functions on the type level, that don't necessarily have to construct a type term, and instead can simply refer to existing ones.

So, Haskell has type-level functions, even type-level types (kinds). It almost seems like an entirely new language, overlaid over Haskell, that operates at compile time, manipulating type terms.

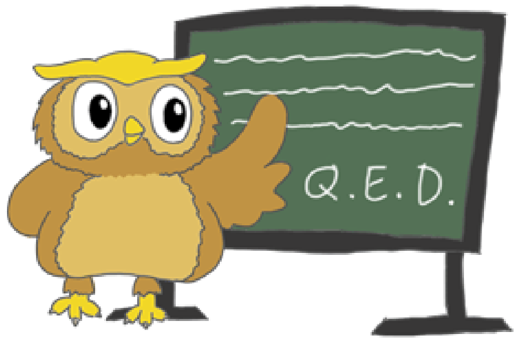
In fact, you could think of any type system this way. In C++, people exploit the Turing-completeness of their type system to perform compile-time analysis and computation. While such type level work is very powerful, I fear that such type machinery is very often difficult to understand and manipulate. Even in Haskell, applications that make extensive use of type-level computation are very often substantially harder to comprehend. The type-level "language" is almost always substantially more complicated to work with than the value-level "language."

In Agda, the distinction between types and values does not exist. Instead, the language you use to manipulate type terms is exactly the same language that you use to manipulate values.

This means that you can actually include values *inside* a type. For example, the `List` type constructor can be parameterized by both the type of its contents *and* the length of the list in question (we'll be doing this later). This allows the compiler to check for you to make sure there are no cases where you attempt to call `head` on a potentially empty list, for example. Being able to include values inside a type, and use all the same value-level operations on them, is what makes Agda *dependently typed* - Not only can values have a type, but types can have a value.

In fact, seeing as the language of values and the language of types are the same, *any property* that you can express about a value can be expressed statically in its type, and machine checked by Agda. We can statically eliminate any error scenario from our program.

Types are Proofs



If I can come up with a function of type `Foo -> Bar` (and Agda says that it's type correct) that means that I've written not only a program, but also a proof by construction that, assuming some premise `Foo`, the judgment `Bar` holds. (We'll touch more on proofs later; I don't want to get bogged down in details just yet.)

Seeing as our `Foo` and `Bar` can be as expressive as we like, this lets us prove *anything we want* about our program simply by exploiting this correspondence between proofs and programs - called the Curry-Howard Correspondence, discovered by two brilliant logicians in the sixties.

Why prove when you can just test?

The validity of formal verification of software is often hotly contested by programmers who usually have no experience in formal verification. Often testing methodologies are presented as a more viable alternative.

While formal verification is excessive in some situations where bugs are acceptable, I hardly think that testing could replace formal verification completely. Here are three reasons for this:

- **Proofs work in concurrent scenarios**. You can't reliably unit test against race conditions, starvation or deadlock. All of these things can be eliminated via formal methods.
- **Proofs, like programs, are compositional**. Tests are not. In testing scenarios, one typically has to write both unit tests and integration tests: unit tests for testing small components individually, and integration tests for testing the interaction between those small components. If I have proofs of the behavior of those small components, I can simply use those proof results to satisfy a proof obligation about their interaction – there is no need to reinvent everything for both testing scenarios.
- **Proofs are fool-proof**. If I have a suite of tests to show some property, it's possible that that property does not actually hold - I simply have not been thorough enough in my tests. With formal verification, it's impossible for violations of your properties to slip through the cracks like that.

Of course, proofs are not for every scenario, but I think they should be far more widely used than they currently are.

Thanks to Curry-Howard, Agda can also be used as a *proof* language, as opposed to a *programming* language. You can construct a proof not just about your program, but about anything you like.

In fact, Curry-Howard shows us that the fundamentals of functional programming (Lambda Calculus), and the fundamentals of mathematical proof (Logic) are in fact the same thing (*isomorphic*). This means that we can structure mathematical proofs in Agda as *programs*, and have Agda check them for us. It's just as valid as a standard pen-and-paper mathematical proof (probably more so, seeing as Agda doesn't let us leave anything as "an exercise for the reader" - and Agda can check our proof's correctness automatically for us. We'll be doing this later by proving some basic mathematical properties on Peano natural numbers.

So, Agda is a language that really lives the dream of the Curry-Howard correspondence. An Agda program is also a proof of the formula represented in its type.

How do I get started?

At the time of writing, it is only really feasible to edit Agda code using Emacs. GNU Emacs or XEmacs are both fine. However, you don't need a great deal of Emacs proficiency to edit Agda code.

Installing Agda

(If you are using Ubuntu Linux, you may wish to skip to the next section, Installing on Ubuntu Linux.)

You'll need GHC, a Haskell compiler, and an assortment of tools and libraries that make up the Haskell Platform. It is the best way to get started using Haskell, and it's also the easiest way to get Agda.

Once you have Haskell and Emacs, there are three things you still need to do:

- Install Agda. Linux users may have Agda packages available from their package manager (search for "agda" to find out). If not or otherwise, simply use the Haskell platform's `cabal-install` tool to download, compile, and set up Agda.

```
$ cabal install agda
```

- Install Agda mode for emacs. Simply type in a command prompt (where Agda is in your `PATH`):

```
$ agda-mode setup
```

- Compile Agda mode as well (you'll need to do this again if you update Agda):

```
$ agda-mode compile
```

By then you should be all set. To find out if everything went as well as expected, head on over to the next section.

Installing on Ubuntu Linux

On a Ubuntu Linux system, instead of installing Agda using cabal as above, one could alternatively use the following two commands, which take a few minutes to run:

```
sudo apt-get install agda-mode
sudo apt-get install agda-stdlib
```

That's it! Now, when you launch Emacs and edit a file with the .agda extension, it should switch to `agda-mode` or `agda2-mode`. If not, you can switch manually by invoking one of the following (in Emacs, of course): `M-x agda-mode` or `M-x agda2-mode` or `Esc-x agda-mode`. (An easy way to find out which agda modes are available is to type `M-x agda` and then hit tab a couple of times to see the possible completions.)

Hello, Peano

Definitions, Definitions

Unlike the previous section, this section will actually involve some coding in Agda.

Most language tutorials start with the typical “Hello, World” example, but this is not really appropriate for a first example in Agda. Unlike other languages, which rely on a whole lot of primitive operations and special cases for basic constructs, Agda is very minimal - most of the “language constructs” are actually defined in libraries.

Agda doesn’t even have numbers built in, so the first thing we’re going to do is define them—specifically *natural numbers*. Natural numbers are nonnegative integers, that is, the whole numbers starting with zero and going up. Mathematics uses the symbol $\mathbb{N}$ to represent natural numbers, so we’re going to borrow that for our example (Another thing that sets Agda apart from other languages is its extensive use of unicode to make mathematical constructs more natural). To enter ℕ into emacs, type `\bn`. To enter the unicode arrow (→), type `\->`. I’m going to demonstrate this line by line, so bear with me.

First, open a file named `LearnYouAn.agda` in Emacs and type the following:

```
module LearnYouAn where
  data ℕ : Set where
```

The `data` keyword means we’re defining a type—in this case, `ℕ`. In this example, we’re specifying that the type `ℕ` is of type `Set` (that’s what the colon means).

Hold on a second, types have types?

If you recall the introduction, I mentioned that in Agda, types and values are treated the same way. Since values are given types, types are given types as well. Types are merely a special group of language terms, and in Agda, all terms have types.

Even `Set` (the type of our type `ℕ`) has a type: `Set₁`, which has a type `Set₂`, going on all the way up to infinity. We’ll touch more on what these `Set` types mean later, but for now you can think of `Set` as the type we give to all the data types we use in our program.

This infinite hierarchy of types provides an elegant solution to Russell’s Paradox. Since, for any ν∈ ℕ, `Set ν` contains only values “smaller” than ν, (for example, `Set₁` cannot contain `Set₁` or `Set₂`, only `Set`), Russell’s problematic set (which contains itself) cannot exist and is not admissible.

Structural Induction

Okay, so, we’ve defined our type, but now we need to fill the type with values. While a type with no values does have its uses, a natural numbers type with no values is categorically wrong. So, the first natural number we’ll define is zero:

```
    zero : ℕ
```

Here we are simply declaring the term `zero` to be a member of our new type `ℕ`. We could continue to define more numbers this way:

```
    zero : ℕ
    one : ℕ
    two : ℕ
```

But we’d quickly find our text editor full of definitions and we’d be no closer to defining all the natural numbers than when we started. So, we should instead refer to a strict mathematical definition.

- Zero is a natural number ($0 \in \mathbb{N}$).
- For any natural number n, $n + 1$ is also a natural number. For convenience, We shall refer to $n + 1$ as $\mathtt{suc}\ n$.[1] ($\forall n \in \mathbb{N}.\ \mathtt{suc}\ n \in \mathbb{N}$).

(The notation I’m using here should be familiar to anyone who knows set theory and/or first-order logic. Don’t panic if you don’t know these things, we’ll be developing models for similar things in Agda later, so you will be able to pick it up as we go along.)

This is called an *inductive definition* of natural numbers. We call it *inductive* because it consists of a *base* rule, where we define a fixed starting point, and an *inductive* rule that, when applied to an element of the set, *induces* the next element of the set. This is a very elegant way to define infinitely large sets. This way of defining natural numbers was developed by a mathematician named Giuseppe Peano, and so they're called the Peano numbers.

We will look at inductive *proof* in the coming sections, which shares a similar structure.

For the base case, we've already defined zero to be in $\mathbb{N}$ by saying: `zero : ℕ`. To define the natural numbers inductively, let's recall the induction step of first order logic. This can be written as follows:

$$\forall n \in \mathbb{N}.\ \texttt{suc}\ n \in \mathbb{N}$$

Given a natural number `n`, the constructor `suc` will return another natural number. In other words, `suc` could be considered a *function* that, when given a natural number, produces the next natural number. In Agda, we define the constructor `suc` like so:

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

Now we can express the number one as `suc zero`, and the number two as `suc (suc zero)`, and the number three as `suc (suc (suc zero))`, and so on.

Aside: Haskell

Incidentally, this definition of natural numbers corresponds to the Haskell data type:

```
data Nat = Zero | Suc Nat
```

If you load that into GHCi and ask it what the type of `Suc` is, it (unsurprisingly) will tell you: `Nat -> Nat`. This is a good way to get an intuition for how to define constructors in Agda.

Also, GHC supports an extension, Generalized Algebraic Data Types or GADTs, which allows you to define data types Agda style:

```
data Nat :: * where
  Zero :: Nat
  Suc  :: Nat -> Nat
```

It's worth noting that GADTs are not exactly the same as Agda data definitions, and Haskell is still not dependently typed, so much of what you learn in this book won't carry over directly to extended Haskell.

One, Two, …Five!

Now we're going to define some arithmetic operations on our natural numbers. Let's try addition, first.

```
_+_ : ℕ → ℕ → ℕ
```

Here I'm declaring a function. To start with, I give it a type[2]—it takes two natural numbers, and returns a natural number.

Aside: What do those underscores mean?

Unlike Haskell which has only prefix functions (ordinary functions) and infix functions (operators), Agda supports *mixfix* syntax. This allows you to declare functions where the arguments can appear anywhere within a term. You use underscores to refer to the "holes" where the arguments are meant to go.

So, an if-then-else construct in Agda can be declared with:[3]

```
  if_then_else_ : ∀ { a } → Bool → a → a → a
```

This can be used with great flexibility: You can call this function with `if a then b else c`, which Agda interprets as `if_then_else_ a b c`. This syntactic flexibility delivers great expressive power, but be careful about using it too much, as it can get very confusing!

Our First Check

Now, let's implement and check the sum function by structural recursion.[4]

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

Normally we'd run the program at this point to verify that it works, but in Agda we *check* our code. This checks that all our *proof obligations* have been met:

- It checks your types. Types are how you encode proofs in Agda (although we haven't done any non-trivial proofs yet), so this is important.
- It checks that your program provably terminates.

Proof obligations of a program can only be machine-checked if the program terminates, but checking that any program terminates is in general undecidable (see The Halting Problem). To circumvent this dilemma, Agda runs its checker only on *structural* recursion with finite data structures, and warns that it can't check proof obligations in which non-structural recursion is used. We will discuss this more in later sections, but all of the examples in the early part of this tutorial can be proved by Agda to terminate.

At this point, the contents of the LearnYouAn.agda file should be as follows:

```
module LearnYouAn where

  data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ

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

Make sure you get the indentation right! In particular, the constructors for `zero` and `suc` start in the 5th column (below the t in `data`), whereas all three lines in the definition of `_+_` begin in column 3.

To check the program, type `C-c C-l` in Emacs (or choose Load from the Agda menu). If your program checks correctly, there will be no error messages, no hole markers (yellow highlighting) and no orange-highlighted non-terminating sections. Also, the words `(Agda: Checked)` should appear in the Emacs mode line, and you should notice that the colors of characters have changed.

Right now, our checks aren't all that meaningful—the only thing they prove is that our addition function does indeed take any natural number and produce a natural number, as the type suggests. Later on, when we encode more information in our types, our checks can mean a lot more—even more than running and testing the program.

"I Have Merely Proven It Correct"

To evaluate an expression (just to verify that it truly does work), we can type `C-c C-n` into emacs, or select "Evaluate term to normal form" from the Agda menu. Then, in the minibuffer, we can type an expression for 3 + 2:

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

This produces the following representation of the number 5:

```
(suc (suc (suc (suc (suc zero)))))
```

In this section we have examined the Peano natural numbers, and defined some basic functions and data types in Agda. In the next section, we'll look at propositional logic, and how to encode logical proofs in Agda using this system.

Troubleshooting

If you are having trouble getting the First program to work, make sure you have the indentation right. It might help to download the LearnYouAn.agda file, just to be sure.

Propositions and Predicates

"Logic is the art of going wrong with confidence"

Now that we've defined the natural numbers, we're going to do some simple example proofs of some basic mathematical properties. We'll first discuss logic and logic specification in *natural deduction*, and as we go, we'll discuss the application to Agda.

At a fundamental level, a logic is a system of *judgments*. Judgments are statements in a mathematical *language* that may be proven, or unproven. A *language* is usually described as a set of strings, which make up every *term* in the language, but this is a simplification: a language can be made up of arbitrary data structures. We just use strings to represent these structures because any data structure can be represented in some string form.

For our example, we will define a very simple logic based on the language of *natural numbers* $\mathbb{N}$ we used earlier.

We're going to have just one type of judgment, of the form $\mathbb{N}\ \mathbf{even}$, which is provable only when the given number is even.

A logic consists of a set of *axioms* and a set of *rules*. Axioms are the foundation of the logic: they're the basic, simple statements that are assumed to be true. *Rules* describe how to produce new *theorems* from existing ones. A theorem is a proposition that has been proved. Thus, the rules tell us how to construct proofs of statements using proofs of other statements. We can formally specify these axioms and rules in a *meta-logic* called *natural deduction*, by writing them in the form of *inference rules*, which look like this:

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_n}{C}\ (\text{name of rule})$$

This says that if we can prove all of the *premises* $P_1 \cdots P_n$, then we can prove the *conclusion* C.

For our purposes, we have just one axiom, that the number zero is even. Axioms are written as inference rules with no premises:

$$\frac{}{\mathtt{zero}\ \mathbf{even}}\ \textsc{Zero}$$

Then, based on the inductive reasoning we used earlier, the *rules* for our logic should express that if some number m is even, then $m + 2$ is also even. We do this by writing an *inference rule schema*, which describes a *set* of rules, by including one or more *metavariables* in an inference rule.

If we have some metavariable x in a rule schema, we can substitute *any* term in the language for x, and the result is a valid rule. For example,

$$\frac{x\ \mathbf{even}}{\mathtt{suc}\ (\mathtt{suc}\ x)\ \mathbf{even}}\ \textsc{Step}$$

If we want to show that four is even, we apply this rule twice (once where x is two, and once when x is zero), leaving the obligation $\mathbf{zero\ even}$ which is shown by the axiom $\textsc{Zero}$.

We can write this proof using natural deduction in a "proof tree" format:

$$\dfrac{\dfrac{\dfrac{}{\mathbf{zero\ even}}\ \textsc{Zero}}{\mathbf{suc\ (suc\ zero)\ even}}\ \textsc{Step}}{\mathbf{suc\ (suc\ (suc\ (suc\ zero)))\ even}}\ \textsc{Step}$$

When proving a theorem, we work from the bottom of this tree upwards, applying rules that fit the form of the goal as we go. When reading the proof, we work downwards, reasoning from known axioms to the theorem that we want.

How does it all relate to Agda?

Agda's types correspond to judgments. If we can construct a value, or "inhabitant," of a certain type, we have simultaneously constructed a *proof* that the theorem encoded by that type holds.

As types are judgments, and values are theorems, *data constructors* for a type correspond to *inference rules* for the corresponding proposition.

Let's encode the judgment $\mathbf{even}$ in Agda, based on our definition in natural deduction. We'll use the mix-fix name `_even` here rather than just `even` so that we can use the judgment in post-fix form. As our judgment is over the language of natural numbers, we *index* the type constructor for our judgment by the type ℕ.

```
data _even : ℕ → Set where
```

Then we can define the axiom $\textsc{Zero}$ as a constructor for the type `zero even` as follows:

```
  ZERO : zero even
```

The $\textsc{Step}$ axiom is a little more complicated, due to the presence of the metavariable x. If we just write the rule as-is,

```
  STEP : x even → suc (suc x) even
```

Agda responds with the following:

```
/home/username/LearnYouAn.agda:14,12-13
Not in scope:
  x
  at /home/username/LearnYouAn.agda:14,12-13
when scope checking x
```

To resolve this, we let `STEP` depend on a parameter, or variable. We name this variable x, and specify that x must be of type ℕ, as follows:

```
  STEP : (x : ℕ) → x even → suc (suc x) even
```

This is an example of a *dependent type*, which provides a means of defining a *rule schema* (a collection of rules parameterized by a variable). Here, for example, `STEP zero` refers to the rule `zero even → suc (suc zero) even`. Thus `STEP` has the same substitution semantics for metavariables that we described above.

At this point, our full program looks as follows:

```
module LearnYouAn where

  data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ

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

  data _even : ℕ → Set where
    ZERO : zero even
    STEP : (x : ℕ) → x even → suc (suc x) even
```

You should type this program into your Emacs buffer, save it as the file LearnYouAn.agda, and check it with `C-c C-l`

Before we go on to use our program to actually prove something, we make one more observation. In the current version of our program, the type of `x` can be inferred from its context, since we use it with `_even`, which takes a instance of type `ℕ` as an argument. Therefore, so we can use the special `∀` symbol to introduce `x` and omit the type. So, our final definition of `_even` is

```
data _even : ℕ → Set where
   ZERO : zero even
   STEP : ∀ x → x even → suc (suc x) even
```

(This change will appear in the file LearnYouAn2.agda below.)

Four is an even number

Now we use our Agda program to prove that four is even. Add the following lines to your LearnYouAn.agda file, and then type `C-c C-l` (use `\_1` to type the subscript symbol `₁`):

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = ?
```

Agda will convert the question mark into the symbols `{ }0`, which is Agda's notation for a *hole* in the program (i.e., a hole in the proof).

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = { }0
```

The following will appear in a separate Emacs buffer:

```
?0 : suc (suc (suc (suc zero))) even
```

This tells us that our proof obligation at hole `?0` is `suc (suc (suc (suc zero))) even`.

Next, put your cursor into the hole, and type `STEP ? ?`, and then type `C-c C-space`. This splits the hole in two more holes.

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = STEP { }1 { }2

?1 : ℕ
?2 : suc (suc zero) even
```

The obligation `?1`, corresponding to hole `{ }1`, is the number we must provide for `x` when applying the `STEP` constructor. Hole `{ }2` corresponds to proof obligation `?2`, which is the obligation to show that two is even. In this case, there is only one constructor that fits the obligation—namely, `STEP`

Move the cursor inside hole `{ }2` and type `C-c C-r`. Agda splits the hole into another `STEP` call for us, resulting in two more holes, `{ }3` and `{ }4`.

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = STEP { }1 (STEP { }3 { }4)

?1 : ℕ
?3 : ℕ
?4 : zero even
```

Another `C-c C-r` in hole `{ }4` will fill it with `ZERO`.

```
  proof₁ : suc (suc (suc (suc zero))) even
  proof₁ = STEP { } (STEP { } ZERO)

?1 : ℕ
?3 : ℕ
```

The remaining obligations, `?1` and `?3`, can also be fulfilled automatically based on the surrounding context. We can see what *constraints* on the holes are known to Agda by using the "Show Constraints" option, or by typing `C-c C-=`. For the present example, this prints the following:

```
?1 := suc (suc zero)
?3 := zero
```

Agda will fill in the holes for us if we use the "Solve Constraints" option, or `C-c C-s`, and our final proof becomes

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = STEP (suc (suc zero)) (STEP zero ZERO)
```

Another feature worth mentioning is Agsy, an automatic proof searcher for Agda. Simply type `C-c C-a` in any hole and Agsy will search for an appropriate term to fill it. It's not guaranteed to find anything, but it can be useful. (In the example above, it works well.)

Implicits

It can be annoying, though, to have to pass in those numbers to `STEP` explicitly, when Agda already knows from surrounding context exactly what they are. In these situations, you can use a single underscore (_) to indicate that you wish Agda to infer the value in this position during type-checking.

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = STEP _ (STEP _ ZERO)
```

If Agda cannot infer the value of the implicit underscore, an "unsolved metavariable" will be shown in the goals window, and the underscore will be highlighted in yellow.

For this particular judgment, however, it is almost always obvious from known constraints what the value of those numbers should be. In these cases, it's common to use an *implicit parameter* when declaring the type:

```
data _even : ℕ → Set where
   ZERO : zero even
   STEP : ∀ {x} → x even → suc (suc x) even    -- long form { x : ℕ } is also fine
```

Note that here we have added braces around the variable `x` in our definition of `STEP`. This lets us omit the number entirely when writing our proof:

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = STEP (STEP ZERO)
```

If Agda cannot, for whatever reason, infer the value of the implicit parameter, yellow highlighting and an unsolved metavariable will be added, as before. In those scenarios, you can manually specify the value of the implicit parameter by using braces:

```
proof₁ : suc (suc (suc (suc zero))) even
proof₁ = STEP {suc (suc zero)} (STEP {zero} ZERO)
```

For `_even`, this is not usually necessary, but if you find yourself specifying implicit parameters frequently, you may wish to consider making it explicit.

Implication

In Agda, an implication proposition corresponds to a *function type*.

When we prove an implication, say `A ⇒ B`, we assume the premise `A`, and derive the conclusion `B`. In terms of proof by construction, this is the same as implementing a function—a function that, when given an argument of type `A`, constructs an return value of type `B`.

In other words, given a proof (by construction) of some proposition `A`, our function produces a proof of proposition `B`. By writing such a function (and having Agda check it), we have constructed a proof of `A ⇒ B`.

Consider the simple tautology `A ⇒ A`. Let's prove this in Agda!

First, we prove it for the proposition that natural numbers exist, that is, "if natural numbers exist, then natural numbers exist."

```
proof₂ : ℕ → ℕ
proof₂ ν = ν      -- type \nu for ν.
```

So, proof of this tautology corresponds to the identity function. The reason for this is fairly clear: Given a proof of `A`, there's one obvious way to produce a proof of `A`: present the same proof you were just given!

Universal Quantification

It would be nice though, to make the above proof about *all* propositions, not merely the proposition that natural numbers exist—after all, the proof is the same regardless of the proposition involved!

To do this, we have to exploit Agda's flexible type system a little. We make our identity function take an additional parameter—a type. Given a type, we then return an identity function, instantiated for that type.

```
proof₂′ : (A : Set) → A → A
proof₂′ _ x = x
```

The new type signature here means: Given some value of type `Set` (i.e., a type), called `A`, return a function from `A` to `A`. We take this to be equivalent to the logical statement, "For any proposition `A`, `A ⇒ A`."

In logic, the *universal quantifier* symbol ∀ means "for any" or "for all". So, the above type signature could be stated as, $(\forall A)(A \Rightarrow A)$. Making propositions about *all* members of a set (or universe) is called *universal quantification*, and it corresponds to *parametric polymorphism* (including Java generics and C++ templates) in type system lingo.

Now we can implement our special case proof, `proof₂`, in terms of the more general `proof₂′`, as follows:

```
proof₂ : ℕ → ℕ
proof₂ = proof₂′ ℕ
```

In other words, "to prove ℕ → ℕ, apply `proof₂′` to type ℕ."

In the next section we will start a new Agda program so, before moving on, we list the full contents of our first Agda program, which resides in the file LearnYouAn2.agda.

```
module LearnYouAn2 where

  data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ

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

  data _even : ℕ → Set where
    ZERO : zero even
    STEP : ∀ x → x even → suc (suc x) even

  proof₁ : suc (suc (suc (suc zero))) even
  proof₁ = STEP (suc (suc zero)) (STEP zero ZERO)

  proof₂ : ℕ → ℕ
  proof₂ ν = ν
  proof₂′ : (A : Set) → A → A
  proof₂′ _ x = x
```

Conjunction

Unlike universal quantification or implication, conjunction and disjunction do not correspond to built-in types in Agda, however they are fairly straightforward to define.

The *conjunction* of the propositions P and Q is denoted by $P \wedge Q$. Informally, to prove the conjunction $P \wedge Q$, we prove each of its components. If we have a proof each component, then we automatically have a proof of their conjunction. Thus conjunction corresponds to a *pair* or a *tuple* (more formally known as a *product type*) in Agda.

More formally, to give meaning to conjunction, we must say how to introduce the judgment $P \wedge Q\ \mathsf{true}$. A verification of $P \wedge Q$ requires a proof of P and a proof of Q. Thus, the introduction rule for conjunction is,

$$\frac{P\ \mathsf{true} \quad Q\ \mathsf{true}}{P \wedge Q\ \mathsf{true}} \quad \wedge \mathrm{I}$$

Let's introduce conjunction in Agda. Create a file called `IPL.agda` containing the following (and check it with `C-c C-l`):

```
module IPL where

  data _∧_ (P : Set) (Q : Set) : Set where
    ∧-intro : P → Q → (P ∧ Q)
```

(The symbol `∧` is produced by typing `\and`.)

Here we've defined a new data type, this time it is *parameterized* by two propositions, which make up the components of the conjunction. Conjunction itself is also a proposition, and we give it the type Set.

Notice how the `∧-intro` constructor can only produce a proof of `P ∧ Q` if it is passed both a proof of `P` and a proof of `Q`. This is how conjunction is demonstrated *by construction*—it is impossible to construct a conjunction that is not supported by proofs of each component.

We now prove some simple properties about conjunctions, such as `P ∧ Q ⇒ P`.

```
  proof₁ : {P Q : Set} → (P ∧ Q) → P
  proof₁ (∧-intro p q) = p
```

This is the elimination rule sometimes called "and-elim-1" or "and-elim-left."
If we define a similar rule for "and-elim-2", we have the following program:

```
module IPL where

  data _∧_ (P : Set) (Q : Set) : Set where
    ∧-intro : P → Q → (P ∧ Q)

  proof₁ : {P Q : Set} → (P ∧ Q) → P
  proof₁ (∧-intro p q) = p

  proof₂ : {P Q : Set} → (P ∧ Q) → Q
  proof₂ (∧-intro p q) = q
```

Bijection

Now that we have defined conjunction and implication, we can define a notion of logical *equivalence*. Two propositions are *equivalent* if both propositions can be considered to be the same. This is defined as: if one is true, the other is also true. In logic, this is called *bijection* and is written as `A ⇔ B`. Bijection can be expressed simply as a conjunction of two implications: If A is true then B is true, and if B is true then A is true.

```
_⇔_ : (P : Set) → (Q : Set) → Set
a ⇔ b = (a → b) ∧ (b → a)
```

(The symbol ⇔ is produced by typing <=>.)

Proving conjunction properties

We can write programs (i.e. proofs) of the algebraic properties of conjunction. The commutative property says that $A \wedge B \Leftrightarrow B \wedge A$. Let's prove it:

```
∧-comm′ : {P Q : Set} → (P ∧ Q) → (Q ∧ P)
∧-comm′ (∧-intro p q) = ∧-intro q p

∧-comm : {P Q : Set} → (P ∧ Q) ⇔ (Q ∧ P)
∧-comm = ∧-intro (∧-comm′ {P} {Q}) (∧-comm′ {Q} {P}) -- implicits provided for clarity only.
```

Remove the implicits and have Agda check the following:

```
∧-comm′ : {P Q : Set} → (P ∧ Q) → (Q ∧ P)
∧-comm′ (∧-intro p q) = ∧-intro q p

∧-comm : {P Q : Set} → (P ∧ Q) ⇔ (Q ∧ P)
∧-comm = ∧-intro ∧-comm′ ∧-comm′
```

Let's also prove associativity.

```
∧-assoc₁ : { P Q R : Set } → ((P ∧ Q) ∧ R) → (P ∧ (Q ∧ R))
∧-assoc₁ (∧-intro (∧-intro p q) r) = ∧-intro p (∧-intro q r)

∧-assoc₂ : { P Q R : Set } → (P ∧ (Q ∧ R)) → ((P ∧ Q) ∧ R)
∧-assoc₂ (∧-intro p (∧-intro q r)) = ∧-intro (∧-intro p q) r

∧-assoc : { P Q R : Set } → ((P ∧ Q) ∧ R) ⇔  (P ∧ (Q ∧ R))
∧-assoc = ∧-intro ∧-assoc₁ ∧-assoc₂
```

Disjunction

If conjunction is a *pair*, because it requires *both* proofs to hold, then disjunction is a *sum type* (also known as an `Either` type), because it only requires one proof in order to hold. In order to model this in Agda, we add *two* constructors to the type, one for each possible component of the disjunction.

```
data _∨_ (P Q : Set) : Set where
   ∨-intro₁ : P → P ∨ Q
   ∨-intro₂ : Q → P ∨ Q
```

Using this, we can come up with some interesting proofs. The simplest one to prove is *disjunction elimination*, which is the rule `∀A B C ⇒ ((A ⇒ C) ∧ (B ⇒ C) ∧ (A ∨ B))⇒ C`. In other symbols,

$$\frac{A \vee B \quad A \Rightarrow C \quad B \Rightarrow C}{C}$$

In words, if A implies C and B implies C, and A is true or B is true, then if follows that C is true.

```
∨-elim : {A B C : Set} → (A → C) → (B → C) → (A ∨ B) → C
∨-elim ac bc (∨-intro₁ a) = ac a
∨-elim ac bc (∨-intro₂ b) = bc b
```

We can also prove the algebraic properties of disjunction, such as commutativity:

```
∨-comm′ : {P Q : Set} → (P ∨ Q) → (Q ∨ P)
```

```
∨-comm′ (∨-intro₁ p) = ∨-intro₂ p
∨-comm′ (∨-intro₂ q) = ∨-intro₁ q

∨-comm : {P Q : Set} → (P ∨ Q) ⇔ (Q ∨ P)
∨-comm = ∧-intro ∨-comm′ ∨-comm′
```

The associativity proof is left as an exercise for the reader.

Negation

You have probably noticed if you're familiar with boolean logic that I've avoided mentioning *false* throughout this entire section. Unlike boolean logic, Agda's *intuitionistic* logic does not have a well-defined notion of "false". In *classical* and boolean logics, all propositions are considered to be either true or false. Intuitionistic logic, by contrast, is purely *constructive*. You can either construct a proof for a proposition, making it true, or you can fail to construct a proof, making you feel bad.

The only "false" values that exist in intuitionistic logic, therefore, are values for which *there can exist no proof*. In Agda, this corresponds to a type that contains no values. We call this type `⊥`, pronounced "bottom". We define it like so:

```
data ⊥ : Set where
```

That's right. No, it's not a mistake. There are no constructors for `⊥`. It is a type for which it is *impossible* to produce a value. Having such a value allows us to define negation (`¬A`) as true if `A` being true would mean bottom is true (which is impossible). Or, in more formal terms: `¬A ⇔ (A ⇒ ⊥)`

```
¬ : Set → Set -- for ¬ type \neg
¬ A = A → ⊥
```

The Curry Howard Correspondence

This section has taught you how to encode propositional logic into Agda's type system. The correspondences discussed here between disjunction and sum types, conjunction and product types, functions and implication, and propositions and types are the fundamentals behind the Curry-Howard Correspondence. Using these tools, you can encode any constructive proof in Agda, which covers a vast range of possible proofs, including the vast majority of proofs encountered in program verification.

Future sections will introduce relational equality, and begin proving some theorems about the Peano numbers we introduced in the previous section.

Hole filling

(This section did not appear in the original version of the tutorial. It is adapted from Stephen Diehl's tutorial, is essentially independent of the rest of the tutorial, and could be read immediately after Section 1.)

Let's get some practice creating some "much needed" gaps in our proofs and then filling them in. As we learned above, Agda calls such gaps "holes".

Open a file called HoleFilling.agda and put the following code in it:

```
module HoleFilling where

data Bool : Set where
  false : Bool
  true : Bool
```

Above we saw how to implement a general conjunction, but here we will implement a simpler (post-fix) conjunction for the `Bool` type in order to demonstrate the creation and removal of holes.

To implement conjunction for `Bool`, we merely have to give the value of the conjunction for each of the four possible pairs of `Bool` values. We begin by entering the following (put this outside the indentation block of `data Bool`):

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

Note that `Bool → Bool → Bool` indicates the types of arguments `∧` should expect (namely, `∧` is a function from `Bool` type to `Bool → Bool` type). Since Agda knows what to expect, it can write much of the function for us. As usual, use `C-c C-l` to type-check the program, and Agda creates a hole where we had a question mark. Our program should now look like this

```
module HoleFilling where

data Bool : Set where
  false : Bool
  true : Bool

∧ : Bool → Bool → Bool
∧ a b = {!!}
```

Now, with the cursor in the hole (i.e., at the braces { }), and with the hole highlighted, type `C-c C-,` and Agda should respond with a list of goals that we must accomplish in order to give a valid definition of `∧` for type `Bool`.

```
Goal: Bool
————————————————————————————————————————————————————————————
b : Bool
a : Bool
```

With the cursor still in the hole, hit `C-c C-c` and Agda responds with “pattern variables to case”, which prompts us to introduce a case for the first variable. Next to the phrase enter the variable `b`, to which Agda responds with

```
∧ : Bool → Bool → Bool
∧ a false = {!!}
∧ a true = {!!}
```

Put the cursor in the first of the two new holes and again type `C-c C-c` but this time enter a in response to the “pattern variables to case” prompt.

```
∧ : Bool → Bool → Bool
∧ false false = {!!}
∧ true false = {!!}
∧ a true = {!!}
```

With the cursor in the hole, type `C-c C-c` and again enter a.

```
∧ : Bool → Bool → Bool
∧ false false = {!!}
∧ true false = {!!}
∧ false true = {!!}
∧ true true = {!!}
```

Finally, we fill in the right hand sides to comport with conjunction:

```
∧ : Bool → Bool → Bool
∧ false false = false
∧ true false = false
∧ false true = false
∧ true true = true
```

Instead of filling in the holes in the end manually, you could place the cursor in each of the holes and hit `C-c C-a`. The result will be four `false`’s, which wouldn’t be a very useful definition of conjunction… so change the last `false` to `true`!

Emacs agda-mode key bindings

Below is a list of the Emacs commands we encountered above. For a more complete list, visit the Agda Wiki.

Command	Key binding
Load	C-c C-l
Evaluate term	C-c C-n
Show goals	C-c C-?
Next goal	C-c C-f
Split obligation	C-c C-space
Split again	C-c C-r
Show Constraints	C-c C-=
Solve Constraints	C-c C-s
Agsy find proof term	C-c C-a

Copyright

Notes

1. `suc` standing for successor. ↩
2. Unlike Haskell, type declarations are mandatory. ↩
3. Don’t worry if you’re scared by that ∀ sign, all will be explained in time. ↩
4. Don’t be scared by the term - structural recursion is when a recursive function follows the structure of a recursive data type - it occurs very frequently in functional programs.↩

by Liam O'Connor-Davis (with a few additions by wjd) - -

g+1 6

« Isotopy