Programming and Proving in Agda

Jesper Cockx

Version of March 9, 2021

Agda is both a strongly typed functional programming language with support for first-class *dependent types* and a *proof assistant* based on the Curry-Howard correspondence between propositions and types. The goal of these lecture notes is to introduce both these unique aspects of Agda to a general audience of functional programmers. It starts from basic knowledge of Haskell as taught in Hutton's book *Programming in Haskell*, and builds up to using equational reasoning to formally prove correctness of functional programs.

Contents

1 *An introduction to Agda for Haskell programmers*

Most programmers think of a type system as a set of rules that together prevent a class of basic runtime errors, such as using a string where the program expects an integer. This is indeed why type systems were first introduced to programming languages: to prevent program crashes that would follow from interpreting data in memory incorrectly.[1] However, since their invention, type systems have evolved to include a much broader range of applications:

[1] For example, a type system can prevent the floating point number 1.0 from being interpreted as a memory address.

- By making use of type information, an IDE can help the programmer during the programming by providing type information and suggestions and by generating and modifying code based on the type information. For example, the IDE can use types to generate all possible cases in a case expression.
- A type can express precise invariants of data in a program, such as the length of a list or the lower and upper bounds of a search tree. These expressive types that can depend on runtime inputs are known as *dependent types*.
- Types can even be used to express mathematical properties and proofs of these properties, which can be checked automatically by the type checker. This usage of a type system as a logic is based on a deep and fundamental result known as the *Curry-Howard correspondence*.

While strongly typed languages such as Java or Haskell have type systems that are quite expressive, due to various reasons it is hard to fully appreciate the three above points from their perspective. In these lecture notes, we will study Agda, a functional language that is similar to Haskell but is a bit more experimental and has an even more expressive type system with full support for dependent types.

On the surface level, Agda code looks quite similar to Haskell code, with some small syntactic differences. However, beneath the hood are a number of notable differences:

- Types in Agda are *first-class values*: basic types such as `Nat` and `Bool` are themselves values of another type called `Set`. Values of type `Set` can be passed around as arguments and returned just as other values.
- All functions in Agda are *total*: where evaluating a function call in Haskell could lead to a runtime error or loop forever, evaluating a function call in Agda is guaranteed to always return a value of the correct type in finite time.
- Agda has support for *dependent function types* where the type of the output can be different depending on the runtime value of the input. This allows us to assign a precise type to functions

Historical note. Agda's type system is based on *dependent type theory*, a formal language invented by the Swedish philosopher Per Martin-Löf. His original motivation was to provide a new foundation for *constructive mathematics*, a kind of mathematics where each proof of existence contains an algorithm to actually construct the object that is proven to exist. For example, a constructive proof of "there exists a prime number greater than 1000" provides an algorithm to actually produce such a number. Later on, it was discovered that Martin-Löf's type theory is also well suited to implement computer proof assistants (such as the Coq system) and programming languages with built-in support for verification (such as Agda).

that are impossible to type correctly in many other languages, such as `printf`.

- Agda also has types that correspond to (proofs of) *logical propositions*. Using these types, Agda can also be used as a *proof assistant* for writing and checking mathematical proofs.

The goal of these lecture notes is *not* to provide a full guide to using Agda as a full-featured programming language. Instead, we use Agda as a vehicle to study the basic building blocks that make up a dependently typed functional programming language. Most of the types and functions we define — as well as much more — can be found in Agda's standard library, which you can get from `github.com/agda/agda-stdlib`. In these notes, we will not make use of the standard library and instead define everything from the ground up. When using Agda for real, it is of course much easier to get these definitions from the library instead!

Installing Agda

There are several ways to install Agda, but unfortunately none are very fast and simple. Here we describe how to install Agda from source using Stack.[2] Other ways to install Agda can be found at `agda.readthedocs.io/en/latest/getting-started/installation.html`. To follow the rest of these notes, it is recommended that you use Agda version 2.6.0 or later.

[2] Instructions for installing Stack can be found on `docs.haskellstack.org/en/stable/README`.

Assuming you have Stack installed, the following command should install the latest release of Agda:

```
stack install Agda-2.6.1.2 --resolver lts-16.31
```

This step will take a long time and a lot of memory to complete, and use up 4-5 GB of disk space for a fresh install.

On Ubuntu Linux and similar systems (including WSL[3] on Windows) you can alternatively install a binary release by running '`sudo apt install agda`'.

[3] `docs.microsoft.com/en-us/windows/wsl/install-win10`

To write and edit Agda code, we recommend Visual Studio Code with the `agda-mode` extension.[4] Other IDEs with support for Agda are Emacs and Atom.

[4] You can get Visual Studio Code from `code.visualstudio.com`. To install an extension, click the Extensions button on the left, enter the name of the extension, and click 'install'.

You can test to see if you've installed Agda correctly by creating a file called `hello.agda` with these lines:

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```
(1)

Open this file in VS Code and press the key combination `Ctrl+c` followed by `Ctrl+l` (for "load file"). You should see a new frame titled Agda with the message `*All Done*`, and the code should be highlighted.[5]

[5] In this introduction to Agda we will only use the interactive mode of Agda, which acts as a type checker and interpreter similar to `ghci` for Haskell. Agda also supports several backends for compilation, including a GHC backend and a JavaScript backend. A full "Hello, world" example for creating an executable from your Agda code can be found at `agda.readthedocs.io/en/v2.6.1.2/getting-started/hello-world.html`.

Once you have loaded an Agda file, there is a number of other commands that can be used to query the typechecker. Here are two that are used very often:

Deduce type (`Ctrl+c Ctrl+d`) This command will ask you for an expression, for which it will then try to infer the type. For example, running this command and entering `hello` will return the inferred type `Greeting`.

Normalize expression (`Ctrl+c Ctrl+n`) This command will ask you for an expression, which it will evaluate as far as possible. For example, running this command and entering `greet` will return the fully evaluated term `hello`.

It can be difficult to figure out the types and normal forms, especially when you are just starting to learn Agda. When you are unsure of the type or normal form of an expression, first try to predict what it is and then use the commands above to test your prediction.

Syntactic differences with Haskell

As we noted before, the syntax of Agda is in many respects similar to that of Haskell. Here we list the most important differences:

Typing Typing is denoted by a single colon in Agda. For example, instead of writing `b :: Bool` as in Haskell, in Agda we write `b : Bool` to indicate that `b` is a boolean.

Unicode Agda allows optional use of unicode symbols in code. For example, we can write `Bool → Bool` instead of `Bool -> Bool` and $\lambda\ x\ \to\ x$ instead of `\x -> x`. It is also allowed to use unicode symbols in names of definitions and variables. For example, we can define a function that is named $\surd$.[6]

[6] Many unicode characters can be entered easily using LaTeX syntax when using the Agda mode for VS Code. For example, when you type `\to` it will be replaced by →, and `\lambda` is replaced by λ.

Naming In Agda there is no syntactic difference between an expression and a type. As a consequence, there are no restrictions on what names have to start with a small letter or a capital letter. In addition, almost all ascii and unicode characters[7] can be used as part of a name. As a convention, names of functions, constructors, and variables usually start with a small letter, while names of both type constructors and type variables usually start with a capital letter. While this naming convention is not enforced by the language, we will keep to it in these notes.

[7] The exceptions are the symbols `.;{}()"`.

Operators To refer to the name of an operator such as $+$ in isolation, Agda uses underscores (as opposed to parentheses in Haskell). For example, we have `_+_ : Nat → Nat → Nat`.

Warning. Agda will interpret all names with underscores as operators, so the use of underscores in non-operator names is strongly discouraged.

Whitespace Due to the liberal naming rules, Agda requires the use of spaces before and after each operator. For example, Agda requires you to write `1 + 1` instead of `1+1`. In fact, `1+1` is a valid name for a function or variable!

Lists Unlike Haskell, Agda does not assign a special role to lists compared to other data structures. In particular, there is no special syntax for list literals or list comprehensions in Agda. Instead, the type `List` A is simply a datatype defined with two constructors `[]` and `_::_`. The Haskell list `[1,2,3]` can then be written in Agda as 1 :: 2 :: 3 :: `[]`.

Tuples Similarly, Agda does not have a special built-in type of tuples. Instead, the type $A \times B$ is defined as a datatype with one constructor `_,_`. For example, we have 6 `, true` : `Nat` × `Bool`.

Data types and pattern matching

Like in Haskell, the core part of any Agda program consists of declarations of new *data types* and new *function definitions* by pattern matching on these data types.

Unlike Haskell, there is no automatic import of any modules from the standard library, so we are free to either load whichever library we want or define everything from scratch. In this introduction, we will do the latter.

Let's define our first datatype in Agda: the type of (unary) natural numbers:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```
(2)

Natural number literals. By default, this definition of natural numbers forces us to write out the numbers as `zero`, `suc zero`, `suc (suc zero)`, ... However, Agda also has built-in support for decimal numbers, which we can activate by writing the following pragma below the definition of `Nat`:

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

Once we have activated this pragma, Agda will interpret the numbers 0, 1, 2, ... using the constructors `zero` and `suc`.

We can immediately spot a few differences with the corresponding Haskell definition `data Nat = Zero | Suc Nat`:

- The names of the constructors start with a small letter instead of a capital letter (see *Naming* above).
- The definition spells out the full type of each constructor instead of just the types of their arguments.
- The definition also assigns a type to the symbol `Nat` itself, namely `Nat : Set`. The reason is that each defined symbol in Agda must have a type - including types themselves - and `Set` is the type of 'simple' types such as `Nat`.

Next, we define addition `_+_` on natural numbers by pattern matching as follows:

```
_+_ : Nat → Nat → Nat
zero    + y = y
(suc x) + y = suc (x + y)
```
(3)

Just like in Haskell, a definition by pattern matching consists of a list of *clauses* or equalities. Also like in Haskell, functions can make *recursive calls* to themselves on the right-hand side of a clause.

To test our definition, we first load the file in VS Code (`Ctrl+c Ctrl+l`) and then ask Agda to evaluate an expression by pressing `Ctrl+c Ctrl+n` (for "normalize expression"). A prompt will appear where you can enter the term to be evaluated. For example, if you enter `(suc zero) + (suc zero)` you should get back the response `suc (suc zero)`.

Exercise 1.1. Define the function `halve : Nat → Nat` that computes the result of dividing the given number by 2 (rounded down). Test your definition by evaluating it for several concrete inputs.

Exercise 1.2. Define the function `_*_ : Nat → Nat → Nat` for multiplication two natural numbers.

Exercise 1.3. Define the type `Bool` with constructors `true` and `false`, and define the functions for negation `not : Bool → Bool`, conjunction `_&&_ : Bool → Bool → Bool`, and disjunction `_||_ : Bool → Bool → Bool` by pattern matching.

Interactive programming in Agda

One of the unique features of Agda is its support for *interactive programming*, which is integrated closely with the typechecker. To use the interactive mode of Agda, you start by writing an incomplete definition with one or more *holes*, placeholders for code you haven't written yet. Holes are written in Agda code as a question mark `?` or the special string `{!!}`.

```
not : Bool → Bool
not b = {!!}
```
(4)

Agda can load a file even if it still contains holes. If you load the file (with `Ctrl+c Ctrl+l`), Agda will assign a number to the holes and give you some information about each one:

```
?0 : Bool
```

Once you have loaded a file with holes, there are various ways you can ask the Agda typechecker to help you through the use of shortcuts:[8]

[8] When you have edited your code in some way and you want to run one of these commands, always remember to first load the file using `Ctrl+c Ctrl+l`.

Get goal type and context (`Ctrl+c Ctrl+,`) This command will give you the type of the hole the cursor is currently in, as well as the types of all the variables that are currently in scope. For example, for the hole above you get:

```
Bool
b : Bool
```

Case (`Ctrl+c Ctrl+c`) This command will perform a case split on a variable. For example, if you put the cursor in the hole in the definition of `not` and press `Ctrl+c Ctrl+c`, Agda will prompt us for a variable to split on. You can then enter the name x of the variable and press `enter`, giving us the following result:

```
not : Bool → Bool
not true  = {!!}
not false = {!!}
```
(5)

Alternatively, you can first enter the name of a variable in the hole (between the `{!` and `!}` signs) and then press `Ctrl+c Ctrl+c` to split on that variable.

Give (`Ctrl+c Ctrl+space`) This command allows you to enter an expression into a hole. For example, if you put our cursor in the first hole in the definition of `not` and press `Ctrl+c Ctrl+space`, Agda will prompt us for an expression to give. You can then enter `false` in the prompt and press `enter`. Agda will check that the expression is of the right type (in this case `Bool`) and then (if typechecking is successful) replace the hole with the given expression.

```
not : Bool → Bool
not true  = false
not false = {!!}
```
(6)

By doing the same with the term `true` in the second hole, the definition of `not` is completed. Alternatively, you can first enter an expression into the hole and then press `Ctrl+c Ctrl+space` to replace the hole with that expression.

You can get a full overview of all available commands by pressing `Ctrl+Shift+P` to open the command palette in VS Code and then typing `Agda`.

The `Set` type and polymorphic functions

One of the fundamental features of Agda is that types such as `Nat` or `Bool` are first-class values that can be returned and passed around as arguments. The type of `Nat` and `Bool` is called `Set` by Agda. For example, we can define an alias for `Nat` as follows:

```
MyNat : Set
MyNat = Nat
```
(7)

This works just like the declaration `type MyNat = Nat` in Haskell: the type checker will treat the two types `Nat` and `MyNat` as identical for all intents and purposes.

The type `Set` is also used to implement polymorphic functions and datatypes in Agda. For example, we can define the polymorphic identity function `id` as follows:

```
id : (A : Set) → A → A
id A x = x
```
(8)

The function `id` takes two arguments: a type A of type `Set` and an element x of type A. For example, `id Bool true` has type `Bool` and evaluates to `true`, while `id Nat zero` has type `Nat` and evaluates to `zero`.

Since writing out the type arguments each time becomes boring quickly, Agda also allows them to be declared *implicit* by using curly braces {}:

```
id : {A : Set} → A → A
id x = x
```
(9)

With this definition, we can simply write `id true` or `id zero` and Agda will infer the type automatically.

The type of `Set`. Since all the types we have seen so far have type `Set`, and `Set` itself is also a type, you might be wondering whether `Set` itself also has type `Set`, i.e. whether we have `Set : Set`. The answer is no: there are deep reasons why this is not allowed in Agda. Concretely, in 1972 the logician Jean-Yves Girard discovered a paradox in type theory (which is also at the basis of Agda). If Agda would allow `Set : Set`, it would break logical soundness of Agda, which is important for using Agda as a theorem prover (see later). More concretely, with `Set : Set` it would be possible to circumvent Agda's termination checker and implement non-terminating functions. To avoid this paradox and the problems it causes, Agda introduces another type $\texttt{Set}_1$ such that $\texttt{Set} : \texttt{Set}_1$, a type $\texttt{Set}_2$ such that $\texttt{Set}_1 : \texttt{Set}_2$, et cetera.

As another example, `if/then/else` can simply be defined as a function in Agda:

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true  then x else y = x                          (10)
if false then x else y = y
```

Note that the underscores _ in the name will make Agda recognize it as an operator, so we can write terms such as `if b then false else true`.

Just as we can define polymorphic functions, we can also define polymorphic datatypes by adding an argument of type `Set`. For example, the type of lists can be defined as follows:[9]

```
data List (A : Set) : Set where
  []  : List A                                      (11)
  _::_ : A → List A → List A
```

[9] In order to write expressions such as 1 :: 2 :: 3 :: [] (as opposed to 1 :: (2 :: (3 :: []))), we also need to tell Agda that :: is right associative. We can do so as follows:

```
infixr 5 _::_
```

Similarly, we can define the type of pairs as a polymorphic datatype. In Agda, the type of pairs is usually written as $A \times B$ in analogy of the notion of the product of two sets in mathematics.[10] It is defined as follows:

[10] To write ×, type `\times`.

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B                               (12)
infixr 4 _,_
```

We can define functions on pairs by pattern matching, for example:

```
fst : {A B : Set} → A × B → A
fst (x , y) = x
                                                    (13)
snd : {A B : Set} → A × B → B
snd (x , y) = y
```

Exercise 1.4. Implement the following Agda functions:

- `length : {A : Set} → List A → Nat`
- `_++_ : {A : Set} → List A → List A → List A`
- `map : {A B : Set} → (A → B) → List A → List B`

Exercise 1.5. Implement the type `Maybe A` with two constructors `just : A → Maybe A` and `nothing : Maybe A`. Next, implement the function `lookup : {A : Set} → List A → Nat → Maybe A` that returns the element at the given position in the list.

Totality checking

If we call a function in Haskell, we know (thanks to purity) that it will not perform arbitrary side-effects like doing IO or modifying global variables. However, there are still a few things that could happen:

- There could be an error in the program due to an incomplete pattern match, or because the program contains a call to `error` or `undefined`.
- The function might not return because it gets stuck in an infinite loop.

Since functions in Haskell do not always produce a result, we call Haskell a *partial language*.

In contrast, Agda is a *total language*, i.e. Agda functions are *total functions* in the mathematical sense:

- There is no `error` or `undefined`
- Agda performs a *coverage check* to ensure all definitions by pattern matching are complete.
- Agda performs a *termination check* to ensure all recursive definitions are terminating.

Let's look at some examples. Suppose we write an incomplete definition by pattern matching:

```
foo : Bool → Bool
foo true = false
```
(14)

Then Agda highlights the definition and gives us an error:

```
Incomplete pattern matching for foo. Missing cases:
  foo false
when checking the definition of foo
```

Likewise, if we write a non-terminating definition:

```
bar : Nat → Nat
bar x = bar x
```
(15)

Then Agda also highlights the definition and gives us an error:

```
Termination checking failed for the following functions:
  bar
Problematic calls:
  bar x
```

These restrictions on coverage and termination can sometimes seem quite restrictive, but it also enables entirely new things to be done with the type system. In particular, totality of functions is crucial for working with dependent types, as function calls can appear inside types. It is also crucial for using Agda as a proof assistant, to rule out circular proofs. We will discuss both of these topics in more detail in the following sections.

Limitations of coverage and termination checking. In general, coverage checking and termination checking are undecidable problems, so it is impossible to detect all complete and terminating functions without allowing some false negatives. Agda instead errs on the side of caution and allows only a subset of all complete and terminating functions. So it could happen that you write down a complicated function that is complete and terminating, yet Agda still throws an error. If that happens, you will have to write the function in a different way to make it obvious to Agda that the function is really total. A good rule of thumb is to include use `Ctrl+c Ctrl+c` to make sure you have cases for all constructors, and to only make recursive calls on arguments that are *structurally decreasing*, i.e. that are a subterm of the pattern on the left-hand side of the clause. When in doubt, it can also pay off to split complex functions into simpler helper functions that are easier for Agda to analyze.

Exercise 1.6. Is it possible to implement a function of type $\{A : \texttt{Set}\} \to \texttt{List}\ A \to \texttt{Nat} \to A$ in Agda? If yes, do so. If no, explain why not.

2 *Dependent types*

Dependent types are types that can refer to — or *depend on* — parts of a program. With dependent types, it is possible to write much more precise types than in other non-dependent type systems. In particular, dependent types can be used to encode invariants of our programs directly into their types, so the type checker can ensure that they are never broken. In this section we will look into how we can use dependent types in Agda and why they can be useful.

Vectors: lists that know their length

The prototypical example of a dependent type is the type of *vectors* $\texttt{Vec}\ A\ n$. This type contains vectors of exactly n elements of type A. For example:

```
myVec1 : Vec Nat 5
myVec1 = 1 :: 2 :: 3 :: 4 :: 5 :: []

myVec2 : Vec (Bool → Bool) 2
myVec2 = not :: (λ x → x) :: []                    (16)

myVec3 : Vec Nat 0
myVec3 = []
```

In fact, the length n does not have to be a literal number, but it can be any expression of type `Nat`. So we might as well have written `Vec Nat` $(2 + 3)$ instead of `Vec Nat` 5.

Overloading of constructors. Agda allows us to use the same name for the constructors of different datatypes, for example `[]` can be a constructor of both `List` and `Vec`.

In a dependently typed language, the return type of a function is allowed to depend on the inputs of the function. For example, we can implement a function `zeroes` that takes as input a number n, and produces a vector of zeroes of length n:

```
zeroes : (n : Nat) → Vec Nat n
zeroes zero    = []                                 (17)
zeroes (suc n) = 0 :: zeroes n
```

The type $(n : $ `Nat`$) \to$ `Vec Nat` n is called a *dependent function type* (a.k.a. a Π type) because the type of the output depends on the input. Note that the type of the result changes depending on which clause of the definition we are in: in the first clause, the input is `zero` so the output must have type `Vec Nat zero`, while in the second clause the input is `suc` n so the output must have type `Vec Nat (suc` n`)`.

Exercise 2.1. Implement the function `downFrom` : $(n :$ `Nat`$) \to$ `Vec Nat` n that, given a number n, produces the vector $(n-1)$ `::` $(n-2)$ `::` $\ldots$ `::` 0. (You'll need to copy the definition of the `Vec` type below to test if your definition typechecks.)

We can also define functions where the type of one of the arguments depends on a previous input. For example:

```
prepend : (n : Nat) → Bool
        → Vec Bool n → Vec Bool (suc n)             (18)
prepend n b bs = b :: bs
```

In fact, this is just a special case of the dependent function type, thanks to currying: the type of `prepend` can equivalently be written as $(n :$ `Nat`$) \to ($`Bool` $\to$ `Vec Bool` $n \to$ `Vec Bool (suc` n`))`.

If an argument to a function appears in the type of a later argument, we can make it implicit by using curly braces $\{\}$:

```
prepend : {n : Nat} → Bool
        → Vec Bool n → Vec Bool (suc n)             (19)
prepend b bs = b :: bs
```

This allows us for example to write `prepend true (false :: [])` instead of `prepend 1 true (false :: [])`. Here Agda infers automatically from the length of the vector `false :: []` that n must be 1.

Note that dependent function types use the same syntax as polymorphic function types. This is no coincidence: in Agda, polymorphic functions are just a special case of dependent functions where the argument is of type `Set`.

Let us now take a closer look at the `Vec` type itself. It is defined as follows:

```
data Vec (A : Set) : Nat → Set where
  []   : Vec A 0
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
infixr 5 _::_
```
(20)

Parameters vs. indices. You might wonder why the argument $(A :$ `Set`$)$ appears *before* the colon in the definition of `Vec`, while the argument type `Nat` appears after it. The reason for this is that A is a *parameter*, while the argument of type `Nat` is an *index* of the datatype. The main difference is that a parameter is bound once for the whole definition and must occur uniformly in the return types of the constructors — i.e. the return type of each constructor must be of the form `Vec` A ... — while each constructor determines the value of the index individually.

Just like `List`, `Vec` has two constructors `[]` and `_::_`. The main difference lies in the *type of the datatype*: it is no longer `Set` but instead `Nat` → `Set`, indicating that it takes one additional argument of type `Nat`. This argument is called an *index*, and correspondingly `Vec` is called an *indexed datatype*. This is reflected in the types of the constructors: `[]` constructs a vector of length 0, while `_::_` takes arguments of type A and `Vec` A n and constructs a vector of length `suc` n, where n is an implicit argument of the constructor.

We can define functions on `Vec` just like we did for `List`, by pattern matching on the constructors. For example, we can define concatenation of vectors as follows:

```
_++Vec_ : {A : Set} {m n : Nat}
        → Vec A m → Vec A n → Vec A (m + n)
[]        ++Vec ys = ys
(x :: xs) ++Vec ys = x :: (xs ++Vec ys)
```
(21)

Pay close attention to the return type `Vec` A $(m + n)$ of this function: it depends on both m and n!

So far the length index of `Vec` has allowed us to make the types of operations on lists more precise, but it hasn't really done anything we couldn't have done in Haskell. So let's try something a bit more ambitious and implement a function `head` that *only* accepts arguments of length at least one.

```
head : {A : Set}{n : Nat} → Vec A (suc n) → A
head (x :: xs) = x
```
(22)

Note that this definition does not include a case for the empty vector `[]`. Yet it is still accepted by the coverage checker of Agda! This is a good thing: the empty vector `[]` does not have type `Vec` A $($`suc` $n)$, so any clause of the form `head []` $=$ ... would be ill-typed! By restricting the type of the input, we have avoided the need to define a case for `[]`, while in Haskell we would either end up with an incomplete definition or be forced to use `error` or `undefined`.

Exercise 2.2. Implement the function `tail` : $\{A :$ `Set`$\}\{n :$ `Nat`$\}$ → `Vec` A $($`suc` $n)$ → `Vec` A n.

Exercise 2.3. Implement the function `dotProduct` : $\{n :$ `Nat`$\}$ → `Vec Nat` n → `Vec Nat` n → `Nat` that calculates the 'dot product' (or scalar product) of two vectors. Note that the type of the function enforces the two vectors to have the same length!

Indexing vectors with the `Fin` type

One important operation on lists is looking up the element at a given position. In Haskell, this is implemented by the function

`(!!) :: [a] -> Int -> a`. However, this is a partial function: if the index is too big (or negative) the function will throw an error. Since throwing errors is not allowed in Agda, we are forced to give it the type $\{A : \texttt{Set}\} \to \texttt{List}\ A \to \texttt{Nat} \to \texttt{Maybe}\ A$, and return `nothing` in case the index is too big. However, we can do better by using `Vec` instead of `List` and requiring the index to be in range of the vector. To make this work, we introduce a new type `Fin` n of *finite sets*. This type consists of all natural numbers less than n, i.e. the numbers $0, 1, 2, \ldots, (n-1)$.

As an example, the type `Fin` 3 contains three elements:

```
zero3 : Fin 3
zero3 = zero

one3 : Fin 3
one3 = suc zero

two3 : Fin 3
two3 = suc (suc zero)
```
(23)

However, if we try to define `three3` : `Fin` 3 as `suc (suc (suc zero))`, we get an error:

```
(suc _n_112) != zero of type Nat
when checking that the expression zero has type Fin 0
```

Without immediately going into the full details of this error message, we can already infer that Agda does not allow us to define `suc (suc (suc zero))` as an element of `Fin` 3.

The type `Fin` 0 in particular contains *zero* arguments: it is an *empty type*. This makes sense as we want to use `Fin` n as an index into a vector of length n, and there are no valid indices into a vector of length 0. Empty types will take an important role once we start using Agda as a proof assistant in the next section.

The type `Fin` is defined as follows:

```
data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)
```
(24)

The constructor `zero` constructs an element of type `Fin (suc n)` for any n, while `suc` constructs a new element of type `Fin (suc n)` for each element of type `Fin` n. In particular, there is no way to construct an element of type `Fin` 0. For type `Fin` 1 the only possible constructor is `zero`, since using `suc` would require us to give an element of type `Fin` 0.

For now, let us define a safe variant of the `lookup` function using `Vec` and `Fin`. Here is the type signature:

```
lookupVec : {A : Set} {n : Nat} → Vec A n → Fin n → A
lookupVec xs i = {!!}
```
(25)

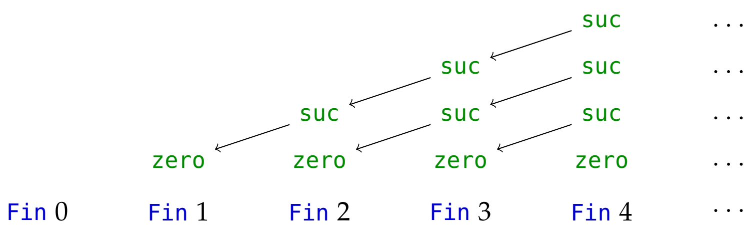

Figure 1: We can visualize the family of `Fin` types as an (infinite) triangle, where each type other than `Fin` 0 has an element `zero`, and each type also has all elements of the form `suc` x where x belongs to the *previous* type in the family.

Notice that the length n of the vector is also used as the upper bound for the index: this way we enforce that the index really is in range of the vector. We then split on the vector xs using `Ctrl+c Ctrl+c`:

```
lookupVec : {A : Set} {n : Nat} → Vec A n → Fin n → A
lookupVec []       i = {!!}                              (26)
lookupVec (x :: xs) i = {!!}
```

So far, so good. Next, we split on the index i in the first clause:

```
lookupVec : {A : Set} {n : Nat} → Vec A n → Fin n → A
lookupVec []       ()                                    (27)
lookupVec (x :: xs) i = {!!}
```

Something interesting has happened: the index has been replaced by the special syntax () and the equality sign has disappeared. This is Agda's way to indicate that there is no element of type `Fin zero`, i.e. there is no possible index into a vector of length 0. The special pattern () used to indicate this is called an *absurd pattern*, and the clause is called an *absurd clause*.

Having completed the case for the empty vector `[]`, we can also split on the index i in the second clause:

```
lookupVec : {A : Set} {n : Nat} → Vec A n → Fin n → A
lookupVec []       ()
lookupVec (x :: xs) zero    = {!!}                       (28)
lookupVec (x :: xs) (suc i) = {!!}
```

Since the length of the vector x `::` xs is `suc` n, we get cases for both constructors of `Fin`. Finally, we can complete the definition of `lookupVec` by filling in the remaining two holes the same way as we would have done for the unsafe lookup function in Haskell.

```
lookupVec : {A : Set} {n : Nat} → Vec A n → Fin n → A
lookupVec []       ()
lookupVec (x :: xs) zero    = x                          (29)
lookupVec (x :: xs) (suc i) = lookupVec xs i
```

Thanks to the power of dependent types, we have managed to implement a *safe* and *total* version of the lookup function, without having to change the return type in any way.

Exercise 2.4. Implement a function `putVec` : $\{A$: `Set`$\}\{n$: `Nat`$\} \to$ `Fin` $n \to A \to$ `Vec` $A\ n \to$ `Vec` $A\ n$ that sets the value at the given position in the vector to the given value, and leaves the rest of the vector unchanged.

The dependent pair type

Another important type for programming with dependent types is called the Σ type or the *dependent pair type*. It can be seen as a generalization of the normal pair type $A \times B$ where the type of the second component can be different depending on the value of the first component. For example, the type `Σ Nat (Vec Bool)` (or equivalently, `Σ Nat (λn → Vec Bool n)`) contains the elements `2 , (true :: false :: [])` and `0 , []` but not `2 , []` (since `[]` does not have type `Vec Bool 2`). In general, this type consists of pairs `m , xs` where `m : Nat` and `xs : Vec A m`.

We can define the type Σ of dependent pairs as follows:[11]

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```
(30)

[11] To write Σ, type `\Sigma`.

Note that the second parameter of Σ is not just a type in `Set` but a *function* of type `A → Set`, i.e. a dependent type.

We can see that Σ is indeed a generalization of the normal pair type where the type of the second component ignores its input:[12]

[12] This definition defines `A ×' B` to be a *type alias* for `Σ A (λ_ → B)`.

```
_×'_ : (A B : Set) → Set
A ×' B = Σ A (λ _ → B)
```
(31)

We have the following functions for getting the individual components of a dependent pair:

Exercise 2.5. Implement functions converting back and forth between `A × B` and `A ×' B`.

```
fstΣ : {A : Set}{B : A → Set} → Σ A B → A
fstΣ (x , y) = x

sndΣ : {A : Set}{B : A → Set} → (z : Σ A B) → B (fstΣ z)
sndΣ (x , y) = y
```
(32)

Note that return type of `sndΣ` depends on the result of the first component `fstΣ`.

An important use of the Σ type is to *hide* some of the information in a type when it is not relevant. For example, we can hide the length of a vector by pairing it up with its length:

```
List' : (A : Set) → Set
List' A = Σ Nat (Vec A)
```
(33)

Exercise 2.6. Implement functions converting back and forth between `List A` and `List' A`.

3 *The Curry-Howard correspondence*

Now as we said before, Agda is not just a programming language but also a proof assistant. This means we can use Agda to formulate theorems and prove them, and Agda will check that the proofs are correct. However, before we can use Agda as a proof assistant, we first have to take a step back and understand the connection between type systems and mathematical logic. This connection is known as the *Curry-Howard correspondence*, named so after the mathematician Haskell B. Curry (yes, that one), who discovered the correspondence between a simple type system and propositional logic in 1934, and the logician William A. Howard, who extended the isomorphism to quantifiers such as ∀ ('for all') and ∃ ('exists') in 1969.

The core idea of the Curry-Howard correspondence is that we can interpret logical propositions — such as "P and Q", "not P", "P implies Q", ... — as *types* whose elements are valid proofs of that proposition.

Propositional logic

To get a better idea of what this means, we will take a look at several logical propositions and deduce what type they correspond to. For each proposition, we ask two important questions: how do we *prove* this proposition, and what can we *deduce* from it?

Conjunction. As a first example, consider the proposition "P and Q". What do we need in order to prove it? Well, we first need a proof of P, and we also need a proof of Q. Hence a proof of "P and Q" is a *pair* (p, q) of two proofs, where p is a proof of P and q is a proof of Q. Similarly, if we are working on a proof and have an assumption that "P and Q" holds, then we can deduce that both P holds and Q holds. So given a proof r of "P and Q", we can get a proof `fst` r of P and another proof `snd` r of Q. So under the Curry-Howard correspondence, "P and Q" corresponds to the pair type $P \times Q$.

Implication. As a second example, let's look at the proposition "P implies Q". In order to prove this implication, we may assume that we have a proof x of P holds and from that we need to construct a proof q of Q. In other words, a proof of "P implies Q" is a *function* $\lambda\ x \to q$ that transforms a proof of P into a proof of Q. In addition, if we have both a proof f of "P implies Q" and a proof p of P, then we can combine the two proofs to get a new proof $f\ p$ of Q. So under the Curry-Howard correspondence, "P implies Q corresponds to the function type $P \to Q$.

Disjunction. What type corresponds to the proposition "P or Q"? In order to prove it, we need to either provide a proof p of P or a proof q of Q. To avoid confusion between whether we proved P or Q, we can label the proofs as either `left` p or `right` q. Using a proof of "P or Q" is a bit less straightforward. If we know that P holds or Q holds and we want to prove R, then we need to show two things: (1) that P implies R, and (2) that Q implies R. We can summarize this proof of R as `cases` $z\ f\ g$ where z is a proof of "P or Q", f is a proof of "P implies R", and g is a proof of "Q implies R" (i.e. $f : P \to R$ and $g : Q \to R$). In conclusion, the proposition "P or Q" corresponds to the type `Either` $P\ Q$ under the Curry-Howard correspondence.

Exercise 3.1. Define the `Either` type in Agda, and write down a definition of the function `cases` : $\{A\ B\ C$: `Set`$\} \to$ `Either` $A\ B \to (A \to C) \to (B \to C) \to C$.

Truth. A very basic proposition is "*true*", i.e. the proposition that always holds no matter what. Proving it is straightforward: we don't need to provide any assumptions. We could thus say there is a trivial proof `tt` of "*true*". On the other hand, assuming "*true*" in a proof does not provide any new information. We can thus say that "*true*" corresponds to the unit type ⊤[13], which is defined as follows:

```
data ⊤ : Set where
  tt : ⊤
```
(34)

(This is very similar to the empty tuple type `()` in Haskell, which has a single inhabitant `() :: ()`.)

[13] Unicode input for ⊤ is `\top`.

Falsity. The other basic proposition is "*false*", the proposition that is never true. There are no ways to prove it, which suggests that it should correspond to an *empty* type. In Agda, we can define the empty type ⊥[14] as a datatype with no constructors:

```
data ⊥ : Set where
  -- no constructors
```
(35)

[14] Unicode input for ⊥ is `\bot`.

On the other hand, if we assume we have a proof p of "*false*", then the principle of explosion (also known under the latin name "ex falso quodlibet", or "from falsity follows anything") tells us we can can get a proof `absurd` p of any proposition we want. In Agda, we can define `absurd` as follows:

```
absurd : {A : Set} → ⊥ → A
absurd ()
```
(36)

(Remember that the absurd pattern () indicates to Agda that there are no valid constructors.)

On empty types. Note that it is not possible to define a real empty type in Haskell: even if we define a type with no constructors, it is still inhabited by `undefined`, as well as infinitely looping programs. So in order to express false propositions, it is essential to work in a total language such as Agda.

Propositional logic versus boolean logic. On the surface, the two types ⊤ and ⊥ seem to be very similar to the booleans `true` and `false`. However, they have a very different role: `true` and `false` are *values* that our Agda programs can manipulate and return, while ⊤ and ⊥ are *types* used by Agda itself. In particular, it is not possible to write a program to check whether a given type is ⊤ or ⊥. So think carefully which one you want to use in what situation!

From these basic propositions, we can derive some other notions:

Negation. We can encode "not P" as the type $P \to \bot$.

Equivalence. We can encode "P is equivalent to Q" as $(P \to Q) \times (Q \to P)$.

The correspondences between propositions and types we have discussed so far are summarized in Table 1.

Propositional logic		**Type system**
proposition	P	type
proof of a proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	`Either` P Q	either type
implication	$P \to Q$	function type
truth	⊤	unit type
falsity	⊥	empty type
negation	$P \to \bot$	function to ⊥
equivalence	$(P \to Q) \times (Q \to P)$	pair of two functions

Table 1: The Curry-Howard correspondence between propositional logic and simple (non-dependent) types in Agda.

Thanks to the Curry-Howard correspondence, we can take any formula in propositional logic, translate it to a type in Agda, and then *prove* the formula by writing down a function of that type. Let's look at some examples:

- The proposition "P implies P" translates to the Agda type $P \to P$, which we can prove as follows:

```
proof1 : {P : Set} → P → P
proof1 p = p
```
(37)

Note that this is simply the identity function `id`.

- The proposition "If (P implies Q) and (Q implies R) then (P implies R)" translates to the Agda type $(P \to Q) \times (Q \to R) \to (P \to R)$, which we can prove as follows:

```
proof2 : {P Q R : Set} → (P → Q) × (Q → R) → (P → R)
proof2 (f , g) = λ x → g (f x)
```
(38)

Note that this is exactly (the uncurried version of) function composition.

- The proposition "If (P or Q) implies R then (P implies R) and (Q implies R)" translates to the Agda type $(\texttt{Either}\ P\ Q \to R) \to (P \to R) \times (Q \to R)$, which we can prove as follows:

```
proof3 : {P Q R : Set}
       → (Either P Q → R) → (P → R) × (Q → R)
proof3 f = (λ x → f (left x)) , (λ y → f (right y))
```
(39)

Exercise 3.2. Translate the following propositions to Agda types using the Curry-Howard correspondence, and prove them by implementing a function of that type.

- If A then (B implies A).
- If (A and *true*) then (A or *false*).
- If A implies (B implies C), then (A and B) implies C.
- If A and (B or C), then either (A and B) or (A and C).
- If A implies C and B implies D, then (A and B) implies (C and D).

Predicate logic

So far, we have used the Curry-Howard correspondence to view formulas from propositional logic as Agda types, and prove them by writing down functions. However, we would also like to write down and prove propositions that say something about a given value or function. For example, we would like to be able to prove that 6 is even, or that `length` (`map` f xs) is equal to `length` xs for all xs, or that there exists a number n such that $n + n = 12$. In order to prove such statements, we first need to answer two more questions: how can we define predicates that express concrete properties such as being even, and how can we express quantifiers such as "for all" and "there exists".

Let's start with the first question. Since — according to Curry-Howard — propositions are types, we can define new propositions by defining new data types. For example, we can define a type `IsEven` n that expresses whether n is even by pattern matching on n:

```
data IsEven : Nat → Set where
  even-zero : IsEven zero
  even-suc2 : {n : Nat} → IsEven n → IsEven (suc (suc n))
```
(40)

Note that `IsEven` is an *indexed datatype*, just like `Vec` and `Fin` We can then easily prove that 6 is even:

```
6-is-even : IsEven 6
6-is-even = even-suc2 (even-suc2 (even-suc2 even-zero))
```
(41)

Constructive mathematics. You may already have noticed that certain propositions that are typically held to be true cannot be proven when translated to Agda via the Curry-Howard correspondence. For example, the law of the excluded middle ("either P or not P") translates to the type $\{P : \texttt{Set}\} \to \texttt{Either}\ P\ (P \to \bot)$, for which we cannot in general provide an implementation without knowing anything about P. The reason is that Agda uses a *constructive logic*, as opposed to the classical logic typically used in mathematics. In particular, a constructive proof of "P or Q" requires us to have a decision procedure to determine whether P holds or Q holds. While this seems like a major limitation of Agda, experience shows it is only very rarely a problem in practice: most of the time a proof that uses the excluded middle can be rewritten in a way such that it doesn't.

For the rare cases where you want to prove a proposition that only holds in classical logic, it is possible to translate between constructive and classical logic using the *double negation translation*: a proposition P is true in classical logic if and only if "not (not P)" is true in constructive logic. So to prove a proposition P using classical logic, all we have to do is prove $(P \to \bot) \to \bot$ in the constructive logic of Agda.

Exercise 3.3. Write a function of type $\{P : \texttt{Set}\} \to (\texttt{Either}\ P\ (P \to \bot) \to \bot) \to \bot$.

On the other hand, there is no way to prove that 7 is even: `IsEven` 7 is an empty type. In fact, we can prove that `IsEven` 7 is false by defining a function from this type to ⊥:

```
7-is-not-even : IsEven 7 → ⊥
7-is-not-even (even-suc2 (even-suc2 (even-suc2 ())))
```
(42)

Here we have to case split three times on the constructor `even-suc2` (using `Ctrl+c Ctrl+c`) before Agda accepts the absurd pattern ().

One predicate that is often quite useful is the predicate stating that a given `Bool` is `true`. It can be defined as follows:

```
data IsTrue : Bool → Set where
  is-true : IsTrue true
```
(43)

The type `IsTrue` b has exactly one constructor if and only if b is `true`, and it is empty if b is false. By using this type in conjunction with a function that returns a boolean, we can easily express many properties. For example, we can define the function `_=Nat_` that checks equality of two numbers, and use it to prove that the list 1 `::` 2 `::` 3 `::` `[]` has length 3:

Defining properties as functions. In this section we define new properties as Agda data types. An alternative approach is to define new properties as *functions* that return a value of type `Set`. For example, an alternative definition of `IsTrue` could be given as:

```
IsTrue' : Bool → Set
IsTrue' true  = ⊤
IsTrue' false = ⊥
```
(44)

This approach often results that are shorter, but less readable. It is also less general, as some types (such as the identity type which we will discuss in the next section) can only be defined as a data type. Hence we prefer the approach using data types over the one using functions.

```
_=Nat_ : Nat → Nat → Bool
zero    =Nat zero    = true
(suc x) =Nat (suc y) = x =Nat y
_       =Nat _       = false

length-is-3 : IsTrue (length (1 :: 2 :: 3 :: []) =Nat 3)
length-is-3 = is-true
```
(45)

Universal and existential quantifiers

Next, we would like to express properties that quantify over a given type, using quantifiers such as ∀ ("for all") and ∃ ("there exists"). Luckily, we already have all the types we need, we just didn't realize it yet!

Universal quantification Consider the proposition "for all x of type A, $P(x)$". To prove it, we need to be able to provide a proof of $P(v)$ for each concrete value $v : A$. In other words, we need a *function* $\lambda v \to p$ that for each v produces a proof p of $P(v)$. In the opposite direction, if we assume we have a proof f of "for all x of type A, $P(x)$" and we have a concrete value $v : A$, then we can apply the proof to the case of v to get a proof $f\ v$ of $P(v)$. So under the Curry-Howard correspondence, "for all x of type A, $P(x)$" corresponds to the *dependent function type* $(x : A) \to Px$.

Existential quantification Next, consider the proposition "there exists a $x : A$ such that $P(x)$". To prove it, we need to provide a concrete example $v : A$, and provide a proof p that $P(v)$ holds, i.e. we need a *pair* (v, p) of a $v : A$ and a $p : P\ v$. Conversely, if we have a proof z of "there exists a $x : A$ such that $P(x)$" then we should be able to extract the witness `fst` $z : A$ as well as the proof `snd` $z : P$ (`fst` z). From this we deduce that "there exists a $x : A$ such that $P(x)$" corresponds to the *dependent pair type* $\Sigma\ A\ (\lambda\ x \to P\ x)$.

We extend the table summarizing the Curry-Howard correspondence with universal and existential quantification in Table 2.

Propositional logic		Type system
proposition	P	type
proof of a proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	`Either` $P\ Q$	either type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	empty type
negation	$P \to \bot$	function to $\bot$
equivalence	$(P \to Q) \times (Q \to P)$	pair of two functions
universal quantification	$(x : A) \to P\ x$	dependent function type
existential quantification	$\Sigma\ A\ (\lambda\ x \to P\ x)$	dependent pair type

Table 2: The Curry-Howard correspondence between predicate logic and dependent types in Agda.

As an example, we can prove that for any natural number n, `double` n is even:

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

double-is-even : (n : Nat) → IsEven (double n)
double-is-even zero    = even-zero
double-is-even (suc m) = even-suc2 (double-is-even m)
```
(46)

Let's take a closer look at the implementation of `double-is-even`. It makes use of two features we've used before: *pattern matching* and *recursion*. Through the lens of the Curry-Howard correspondence, we can see these two features in a new light.

- By pattern matching on the natural number n, we are doing a *proof by cases* on n, which allows us to prove the cases $n =$ `zero` and $n =$ `suc` m separately. Thanks to Agda's coverage checker, we can be sure that the proof covers all cases.
- By making a recursive call to `double-is-even` m on the right-hand side for `double-is-even` (`suc` m), we are making use of *induction* on the number n: we assume that the proposition holds for $n = m$, and from that we prove that it holds for $n =$ `suc` m. By induction, we can then conclude that it holds for all values of n. Thanks to Agda's termination checker, we can be sure that we only make use of the inductive hypothesis for smaller values of n.

So in summary, thanks to the Curry-Howard correspondence we can use familiar techniques from functional programming to write formal proofs!

As another example, we can prove that for any number n, n =Nat n is true:

```
n-equals-n : (n : Nat) → IsTrue (n =Nat n)
n-equals-n zero    = is-true
n-equals-n (suc m) = n-equals-n m
```
(47)

We can now also prove existential statements by making use of the Σ type. For example, we can prove that there exists a number n such that $n + n = 12$ by exhibiting the number 6:

```
half-a-dozen : Σ Nat (λ n → IsTrue ((n + n) =Nat 12))
half-a-dozen = 6 , is-true
```
(48)

As another example, we can prove that any number n is either 0 or the successor of another number m:

```
zero-or-suc : (n : Nat)
  → Either (IsTrue (n =Nat 0))
           (Σ Nat (λ m → IsTrue (n =Nat (suc m))))
zero-or-suc zero    = left is-true
zero-or-suc (suc m) = right (m , n-equals-n m)
```
(49)

The identity type

In the previous section, we proved that `length (xs ++ ys)` is equal to `length xs + length ys` by making use of the function `_=Nat_` together with the predicate `IsTrue`. This method works for concrete types such as natural numbers, but it has a fundamental flaw: if we want to prove something about a function with return type X, we first need to define a function `_=X_` : $X \to X \to$ `Bool`. This can be rather difficult, and moreover it doesn't work for abstract type variables: how would we state that `id` x is equal to x for variables $x : A$ of *any* type A?

In order to express equality at any type, Martin-Löf introduced a new type $x \equiv y$[15], called the *identity type*,[16] with a single constructor `refl` : $x \equiv x$ (short for 'reflexivity'). If x and y are equal, then $x \equiv y$ has a single inhabitant `refl`, so it behaves like the unit type ⊤. On the other hand, if x and y are distinct (e.g. x = `zero` and y = `suc zero`) then $x \equiv y$ has no constructors and hence it behaves like the empty type ⊥. In Agda, we can define the identity type as follows:

[15] Unicode input for ≡ is `\equiv`.

[16] Do not confuse the *identity type* $x \equiv y$ with the (type of the) *identity function* `id` : {A : `Set`} → A → A.

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x

infix 4 _≡_
```
(50)

Just like `Vec` and `Even`, `_≡_` is defined as an indexed datatype, this time with two indices of type A.

With this type, we can for example prove that, indeed, $1 + 1 = 2$:[17]

```
one-plus-one : 1 + 1 ≡ 2
one-plus-one = refl
```
(51)

[17] 20 pages is still better than the 362 pages that Whitehead and Russell needed to prove this fact!

... and that 0 is not equal to 1:[18]

[18] Exciting!

```
zero-not-one : 0 ≡ 1 → ⊥
zero-not-one ()
```
(52)

We can also prove facts about polymorphic types, for example that the `id` function always returns its input:

```
id-returns-input : {A : Set} → (x : A) → id x ≡ x
id-returns-input x = refl
```
(53)

Despite the fact that the identity type only has a single inhabitant `refl`, we can prove other properties of equality: symmetry (if $x = y$, then $y = x$), transitivity (if $x = y$ and $y = z$ then $x = z$), and congruence (if $x = y$ then $f(x) = f(y)$). To prove these properties, we have to match on an argument of type $x \equiv y$. Since `refl` is the only constructor of this type, there is always only a single case. However, pattern matching on `refl` is not useless: by matching a proof of $x \equiv y$ against the constructor `refl`, Agda learns that x and y are indeed equal. For example, in the definition of `sym` below, matching on `refl` teaches Agda that x and y must be equal, which is required for the right-hand side `refl` to be accepted at type $y \equiv x$.

Unit tests in Agda. A neat trick you can do in Agda is to write unit tests directly in your code by making use of the identity type. For example, we can write a test that `length (1 :: 2 :: [])` is 2 as follows:

```
length-test1 :
  length (1 :: 2 :: []) ≡ 2
length-test1 = refl
```
(54)

When the Agda typechecker checks that `refl : length (1 :: 2 :: []) ≡ 2`, it will evaluate both sides to make sure they are indeed equal. So if anything changes and the test no longer succeeds, you will know immediately as the file will not even typecheck any more!

```
-- symmetry of equality
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl

-- transitivity of equality
trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl

-- congruence of equality
cong : {A B : Set} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```
(55)

In general, there are two cases where we can match on a proof of $a \equiv b$:

- If a and b can be unified by instantiating some of the variables, then we can match on `refl`.
- If a and b are obviously different (e.g. are applications of different constructors) then we can match on an absurd pattern ()

In all other cases where a and b cannot easily be unified but they are not obviously distinct either, Agda will throw an error message saying it doesn't know whether there should be a case for the constructor `refl`.

4 Equational reasoning in Agda

In chapter 16 of his book *Programming in Haskell*, Hutton explains how to use *equational reasoning* to prove properties of Haskell functions. However, these proofs quickly become long and rather boring, which makes it easy for mistakes to slip through. In Agda, we can do better: thanks to the Curry-Howard correspondence, we can write proofs about Agda in Agda itself, and have the typechecker check their correctness automatically. Moreover, thanks to Agda's flexible operator syntax, we can even write these proofs in a style very similar to Hutton's.

In order to write equational reasoning proofs in Agda, we first need to define a few operators that will provide us with a nice syntax to write down these proofs. It is normal that these definitions do not make much sense on their own, but their usefulness will become apparent soon.[19]

[19] To enter the ⟨ symbol in Agda, write \<, and similarly for ⟩, write \>.

```
begin_ : {A : Set} → {x y : A} → x ≡ y → x ≡ y
begin p = p

_end : {A : Set} → (x : A) → x ≡ x
x end = refl

_=⟨_⟩_ : {A : Set} → (x : A) → {y z : A}
       → x ≡ y → y ≡ z → x ≡ z
x =⟨ p ⟩ q = trans p q                                  (56)

_=⟨⟩_ : {A : Set} → (x : A) → {y : A} → x ≡ y → x ≡ y
x =⟨⟩ q = x =⟨ refl ⟩ q

infix  1 begin_
infix  3 _end
infixr 2 _=⟨_⟩_
infixr 2 _=⟨⟩_
```

Simple examples

As a first simple example of equational reasoning in Agda, let us prove that `reverse` has no effect on singleton lists, in the sense that `reverse [ x ]` $=$ `[ x ]` for any value x, where `[ x ]` $= x$ `:: []`:

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

reverse : {A : Set} → List A → List A
reverse [] = []
reverse (x :: xs) = reverse xs ++ [ x ]

reverse-singleton : {A : Set} (x : A) → reverse [ x ] ≡ [ x ]
reverse-singleton x =
  begin
    reverse [ x ]                                              (57)
  =⟨⟩ -- definition of [_]
    reverse (x :: [])
  =⟨⟩ -- applying reverse (second clause)
    reverse [] ++ [ x ]
  =⟨⟩ -- applying reverse (first clause)
    [] ++ [ x ]
  =⟨⟩ -- applying _++_
    [ x ]
  end
```

From this, we can see the general structure of a proof that uses equational reasoning: it starts with `begin` and ends with `end`, and in between is a sequence of expressions separated by `=⟨⟩` symbols. Each of the expressions should be equal to the previous one, and the result of the whole block (starting with `begin` and ending with `end`) is a proof that the first expression is equal to the last one. This results in proofs that are very easy to read compared to ones using `refl` and `trans` directly.

Proof by cases and induction

We can combine equational reasoning with other techniques we have seen before. For example, we can prove that `not` (`not` b) $= b$ by case analysis (i.e. pattern matching) on b:

```
not-not : (b : Bool) → not (not b) ≡ b
not-not false =
  begin
    not (not false)
  =⟨⟩ -- applying the inner not                               (58)
    not true
  =⟨⟩ -- applying not
    false
  end
```

```
not-not true =
  begin
    not (not true)
  =⟨⟩ -- applying the inner not
    not false
  =⟨⟩ -- applying not
    true
  end
```
(59)

We can also prove facts about natural numbers by induction (i.e. recursion). For example, we can prove that $n + 0 = n$ for all n : `Nat` (note that this fact is not immediately obvious from the definition of `_+_`, which only says something about $0 + n$ and (`suc` m) + n):

```
add-n-zero : (n : Nat) → n + zero ≡ n
add-n-zero zero =
  begin
    zero + zero
  =⟨⟩                        -- applying +
    zero
  end
add-n-zero (suc n) =
  begin
    (suc n) + zero
  =⟨⟩                        -- applying +
    suc (n + zero)
  =⟨ cong suc (add-n-zero n) ⟩ -- using induction hypothesis
    suc n
  end
```
(60)

(61)

In order to prove that `suc` $(n + 0)$ is equal to `suc` n, we have to make use of the induction hypothesis `add-n-zero` n, which says that $n + 0 = n$. Agda cannot figure this out on its own, so we have to provide the proof manually. This is where the operator `_=⟨_⟩_` comes in: it allows us to provide an equality proof in between the angle brackets. We have to provide a proof of `suc` $(n + 0) \equiv$ `suc` n but `add-n-zero` n has type $n + 0 \equiv n$, so we apply `cong suc` to it to apply `suc` to both sides of the equation.

Exercise 4.1. Prove that m + `suc` n = `suc` $(m + n)$ for all natural numbers m and n. Next, use the previous lemma and this one to prove commutativity of addition, i.e. that $m + n = n + m$ for all natural numbers m and n.

As another example, let us show that addition of natural numbers is associative, i.e. that $x + (y + z) = (x + y) + z$. Since there are three variables, we have to choose on which one to pattern match. Since `_+_` is defined by pattern matching on its first argument, and x appears twice as the first argument to `_+_`, it is natural to try matching on x first:

```
add-assoc : (x y z : Nat) → x + (y + z) ≡ (x + y) + z
add-assoc zero y z =
  begin
    zero + (y + z)
  =⟨⟩                              -- applying the outer +
    y + z
  =⟨⟩                              -- unapplying add
    (zero + y) + z
  end
add-assoc (suc x) y z =
  begin
    (suc x) + (y + z)
  =⟨⟩                              -- applying the outer add
    suc (x + (y + z))
  =⟨ cong suc (add-assoc x y z) ⟩ -- using induction hypothesis
    suc ((x + y) + z)
  =⟨⟩                              -- unapplying the outer add
    (suc (x + y)) + z
  =⟨⟩                              -- unapplying the inner add
    ((suc x) + y) + z
  end
```
(62)

(63)

In general, each case of a proof typically starts by applying some definitions, then perhaps applying an auxiliary lemma and/or induction hypothesis, and finally unapplying some definitions. When you are stuck writing a proof, it often helps to work from both directions: 'forwards' from the starting point and 'backwards' from the final result, until it becomes clear what step is still required.

Exercise 4.2. Consider the following function:

```
replicate : {A : Set}
  → Nat → A → List A
replicate zero    x = []
replicate (suc n) x =
  x :: replicate n x
```
(64)

Prove that the length of `replicate` n x is always equal to n, by induction on the number n.

Induction on lists

Induction can be used to prove properties of any recursive datatype, not just natural numbers. As an example, we will prove that reversing a list is its own inverse, i.e. `reverse` (`reverse` xs) $=$ xs, by induction on xs.

The proof of the base case (for `[]`) is straightforward, but the inductive case (for x `::` xs) requires a bit more work. In particular, it requires us to prove that `reverse` distributes over list concatenation, swapping the two lists in the process:

$$\texttt{reverse}\ (xs \mathbin{+\!+} ys) = \texttt{reverse}\ ys \mathbin{+\!+} \texttt{reverse}\ xs$$

We can prove such auxiliary lemmas either as standalone definitions, or as local definitions in a `where`-block. Here we take the latter approach.

```
reverse-reverse : {A : Set} → (xs : List A) → reverse (reverse xs) ≡ xs
reverse-reverse [] =
  begin
    reverse (reverse [])
  =⟨⟩                                        -- applying inner reverse        (65)
    reverse []
  =⟨⟩                                        -- applying reverse
    []
  end
reverse-reverse (x :: xs) =
  begin
    reverse (reverse (x :: xs))
  =⟨⟩                                        -- applying the inner reverse
    reverse (reverse xs ++ [ x ])
  =⟨ reverse-distributivity (reverse xs) [ x ] ⟩ -- distributivity (see below)
    reverse [ x ] ++ reverse (reverse xs)
  =⟨⟩                                        -- reverse singleton list        (66)
    [ x ] ++ reverse (reverse xs)
  =⟨⟩                                        -- definition of ++
    x :: reverse (reverse xs)
  =⟨ cong (x ::_) (reverse-reverse xs) ⟩     -- using induction hypothesis
    x :: xs
  end
  where
    reverse-distributivity : {A : Set} → (xs ys : List A)
                           → reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
    reverse-distributivity [] ys =
      begin
        reverse ([] ++ ys)
      =⟨⟩                                    -- applying ++
        reverse ys
      =⟨ sym (append-[] (reverse ys)) ⟩      -- see append-[] lemma          (67)
        reverse ys ++ []
      =⟨⟩                                    -- unapplying reverse
        reverse ys ++ reverse []
      end

      where
        append-[] : {A : Set} → (xs : List A) → xs ++ [] ≡ xs
        -- definition of append-[] omitted
```

```
reverse-distributivity (x :: xs) ys =
  begin
    reverse ((x :: xs) ++ ys)
  =⟨⟩                                                -- applying ++
    reverse (x :: (xs ++ ys))
  =⟨⟩                                                -- applying reverse
    reverse (xs ++ ys) ++ reverse [ x ]
  =⟨⟩                                                -- applying reverse
    reverse (xs ++ ys) ++ [ x ]
  =⟨ cong (_++ [ x ]) (reverse-distributivity xs ys) ⟩ -- using induction hypothesis
    (reverse ys ++ reverse xs) ++ [ x ]
  =⟨ append-assoc (reverse ys) (reverse xs) [ x ] ⟩  -- using associativity of ++
    reverse ys ++ (reverse xs ++ [ x ])
  =⟨⟩                                                -- unapplying inner ++
    reverse ys ++ (reverse (x :: xs))
  end

  where
    append-assoc : {A : Set} → (xs ys zs : List A)
                 → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
    -- definition of append-assoc omitted
```
(68)

The proof of distributivity in turn requires two auxiliary lemmas: that `xs ++ [] = xs`, and the associativity of `_++_`.

Exercise 4.3. Fill in the missing proofs of `append-[]` and `append-assoc`.

As another example, we can show that the `map` function satisfies the two functor laws:

`map id = id`	(identity law)
`map (g . h) = map g . h`	(composition law)

The first law is straightforward to prove:

```
map-id : {A : Set} (xs : List A) → map id xs ≡ xs
map-id [] =
  begin
    map id []
  =⟨⟩ -- applying map
    []
  end
```
(69)

```
map-id (x :: xs) =
  begin
    map id (x :: xs)
  =⟨⟩                          -- applying map
    id x :: map id xs
  =⟨⟩                          -- applying id
    x :: map id xs
  =⟨ cong (x ::_) (map-id xs) ⟩ -- using induction hypothesis
    x :: xs
  end
```
(70)

In the final step of the proof, we make use of the *section syntax* x ::_; this is equivalent to $\lambda\ xs \to x$ :: xs.

For the second law, we first need to define function composition. The symbol . is not a valid function name, but we can instead use the symbol ∘.[20]

[20] Unicode input for ∘ is \o

```
_∘_ : {A B C : Set} → (B → C) → (A → B) → (A → C)
g ∘ h = λ x → g (h x)
```
(71)

With this definition, we can also state and prove the second functor law for map on lists:

```
map-compose : {A B C : Set} (f : B → C) (g : A → B) (xs : List A)
            → map (f ∘ g) xs ≡ map f (map g xs)
map-compose f g [] =
  begin
    map (f ∘ g) []
  =⟨⟩                                  -- applying map
    []
  =⟨⟩                                  -- unapplying map
    map f []
  =⟨⟩                                  -- unapplying map
    map f (map g [])
  end
```
(72)

```
map-compose f g (x :: xs) =
  begin
    map (f ∘ g) (x :: xs)
  =⟨⟩                                  -- applying map
    (f ∘ g) x :: map (f ∘ g) xs
  =⟨⟩                                  -- applying function composition
    f (g x) :: map (f ∘ g) xs
  =⟨ cong (f (g x) ::_) (map-compose f g xs) ⟩ -- using induction hypothesis
    f (g x) :: map f (map g xs)
  =⟨⟩                                  -- unapplying map
    map f (g x :: map g xs)
  =⟨⟩                                  -- unapplying map
    map f (map g (x :: xs))
  end
```
(73)

Exercise 4.4. Prove that length (map f xs) is equal to length xs for all xs.

Exercise 4.5. Define the functions take and drop that respectively return or remove the first n elements of the list (or all elements if the list is shorter). Prove that for any number n and any list xs, we have take n xs ++ drop n $xs = xs$.

Verifying optimizations

In section 16.6 of *Programming in Haskell*, Hutton notes that the naive implementation of reverse using concatenation _++_ is very inefficient: it needs to traverse the whole list for each application of _++_, resulting in quadratic complexity overall. A more efficient implementation of this function uses a helper function with an extra argument called an *accumulator* to store pass around the intermediate results.

```
reverse-acc : {A : Set} → List A → List A → List A
reverse-acc [] ys = ys                                          (74)
reverse-acc (x :: xs) ys = reverse-acc xs (x :: ys)
```

```
reverse' : {A : Set} → List A → List A                          (75)
reverse' xs = reverse-acc xs []
```

We can test that this function `reverse'` is indeed much faster than `reverse`: it has linear rather than quadratic complexity. However, can we be sure that the two implementations do indeed produce the same result? Let's prove it!

```
reverse'-reverse : {A : Set} → (xs : List A) → reverse' xs ≡ reverse xs
reverse'-reverse xs =
  begin
    reverse' xs
  =⟨⟩                           -- definition of reverse'
    reverse-acc xs []                                           (76)
  =⟨ reverse-acc-lemma xs [] ⟩ -- using reverse-acc-lemma
    reverse xs ++ []
  =⟨ append-[] (reverse xs) ⟩  -- using append-[]
    reverse xs
  end
```

To prove the correctness of `reverse'`, we need to prove a fact about the helper function `reverse-acc`, namely that `reverse-acc xs [] = reverse xs ++ []`. However, if we try to prove this directly, we get stuck: in the recursive case, the second argument of `reverse-acc` is no longer `[]`, so it is not possible to make use of the inductive hypothesis. Instead, we need to prove a more general result where we replace `[]` by a variable *ys*.[21]

[21] When proving something by induction, it often happens that a direct attempt fails, but we can first prove a more general statement for which the induction does work, and then derive the desired result from that. So when you are stuck on a proof, keep your eyes open for possible generalizations!

```
  where
    reverse-acc-lemma : {A : Set} → (xs ys : List A)
      → reverse-acc xs ys ≡ reverse xs ++ ys
    reverse-acc-lemma [] ys =
      begin
        reverse-acc [] ys
      =⟨⟩                                                       (77)
        ys
      =⟨⟩
        [] ++ ys
      =⟨⟩
        reverse [] ++ ys
      end
```

```
reverse-acc-lemma (x :: xs) ys =
  begin
    reverse-acc (x :: xs) ys
  =⟨⟩                                           -- definition of reverse-acc
    reverse-acc xs (x :: ys)
  =⟨ reverse-acc-lemma xs (x :: ys) ⟩           -- using induction hypothesis
    reverse xs ++ (x :: ys)
  =⟨⟩                                           -- unapplying ++
    reverse xs ++ ([ x ] ++ ys)
  =⟨ sym (append-assoc (reverse xs) [ x ] ys) ⟩ -- using associativity of append
    (reverse xs ++ [ x ]) ++ ys
  =⟨⟩                                           -- unapplying reverse
    reverse (x :: xs) ++ ys
  end
```
(78)

The proof of `reverse-acc-lemma` is mostly standard. The only notable thing is the use of the two lemmas `append-[]` and `append-assoc` which you have proved before: if you want to make them accessible here, you'll need to move them out of the `where`-block to the top level to make them globally accessible.

In his book, Hutton gives another example of using an accumulator for flattening a tree structure. In Agda, we can define binary trees as follows:

```
data Tree (A : Set) : Set where
  leaf : A → Tree A
  node : Tree A → Tree A → Tree A
```
(79)

We can then define two versions of the `flatten` function: one naive implementation that uses `_++_`, and one efficient one that uses an accumulator:

```
flatten : {A : Set} → Tree A → List A
flatten (leaf x)     = [ x ]
flatten (node t1 t2) = flatten t1 ++ flatten t2
```
(80)

```
flatten-acc : {A : Set} → Tree A → List A → List A
flatten-acc (leaf x)     xs = x :: xs
flatten-acc (node t1 t2) xs =
  flatten-acc t1 (flatten-acc t2 xs)
```
(81)

```
flatten' : {A : Set} → Tree A → List A
flatten' t = flatten-acc t []
```
(82)

Now we can prove that these two implementations are functionally equivalent, following the example of the `reverse` function above:

Exercise 4.6. Complete the proof of `flatten'-flatten`.

```
flatten-acc-flatten : {A : Set} (t : Tree A) (xs : List A) → flatten-acc t xs ≡ flatten t ++ xs
flatten-acc-flatten (leaf x) xs =
  begin
    flatten-acc (leaf x) xs
  =⟨⟩
    x :: xs
  =⟨⟩
    [ x ] ++ xs
  =⟨⟩
    flatten (leaf x) ++ xs
  end
flatten-acc-flatten (node l r) xs =
  begin
    flatten-acc (node l r) xs
  =⟨⟩
    flatten-acc l (flatten-acc r xs)
  =⟨ flatten-acc-flatten l (flatten-acc r xs) ⟩
    flatten l ++ (flatten-acc r xs)
  =⟨ cong (flatten l ++_) (flatten-acc-flatten r xs) ⟩                    (83)
    flatten l ++ (flatten r ++ xs)
  =⟨ sym (append-assoc (flatten l) (flatten r) xs) ⟩
    (flatten l ++ flatten r) ++ xs
  =⟨⟩
    (flatten (node l r)) ++ xs
  end

flatten'-flatten : {A : Set} → (t : Tree A) → flatten' t ≡ flatten t
flatten'-flatten t =
  begin
    flatten' t
  =⟨⟩
    flatten-acc t []
  =⟨ flatten-acc-flatten t [] ⟩
    flatten t ++ []
  =⟨ append-[] (flatten t) ⟩
    flatten t
  end
```

Compiler correctness

We conclude this section with the extended example from section 16.7 of *Programming in Haskell*. Rather than duplicating the full explanation here, we just show how to port the code to Agda and formally state the correctness property. Please refer to the book for a full explanation.

```
data Expr : Set where
  valE : Nat → Expr                                                         (84)
  addE : Expr → Expr → Expr
```

```
eval : Expr → Nat
eval (valE x)     = x
eval (addE e1 e2) = eval e1 + eval e2
```
(85)

```
data Op : Set where
  PUSH : Nat → Op
  ADD  : Op
```
(86)

```
Stack = List Nat
Code  = List Op
```
(87)

```
exec : Code → Stack → Stack
exec []             s             = s
exec (PUSH x :: c)  s             = exec c (x :: s)
exec (ADD :: c)     (m :: n :: s) = exec c (n + m :: s)
exec (ADD :: c)     _             = []
```
(88)

```
-- First version, inefficient and hard to reason about
-- compile :  Expr → Code
-- compile (valE x) = [ PUSH x ]
-- compile (addE e1 e2) = compile e1 ++ compile e2 ++ [ ADD ]

-- Second version, much faster and easier to
-- prove correct
compile' : Expr → Code → Code
compile' (valE x)     c = PUSH x :: c
compile' (addE e1 e2) c = compile' e1 (compile' e2 (ADD :: c))

compile : Expr → Code
compile e = compile' e []
```
(89)

```
exec-distributivity : (c d : Code) (s : Stack)
                    → exec (c ++ d) s ≡ exec d (exec c s)
exec-distributivity []            d s = {!!}
exec-distributivity (PUSH x :: c) d s = {!!}
exec-distributivity (ADD    :: c) d s = {!!}
```
(90)

Exercise 4.7. Complete the proof of `exec-distributivity`.

```
compile'-exec-eval : (e : Expr) (s : Stack) (c : Code)
  → exec (compile' e c) s ≡ exec c (eval e :: s)
compile'-exec-eval (valE x) s c =
  begin
    exec (compile' (valE x) c) s
  =⟨⟩
    exec (PUSH x :: c) s
  =⟨⟩
    exec c (x :: s)
  =⟨⟩
    exec c (eval (valE x) :: s)
  end
```
(91)

```
compile'-exec-eval (addE e1 e2) s c =
  begin
    exec (compile' (addE e1 e2) c) s
  =⟨⟩
    exec (compile' e1 (compile' e2 (ADD :: c))) s
  =⟨ compile'-exec-eval e1 s (compile' e2 (ADD :: c)) ⟩
    exec (compile' e2 (ADD :: c)) (eval e1 :: s)
  =⟨ compile'-exec-eval e2 (eval e1 :: s) (ADD :: c) ⟩          (92)
    exec (ADD :: c) (eval e2 :: eval e1 :: s)
  =⟨⟩
    exec c (eval e1 + eval e2 :: s)
  =⟨⟩
    exec c (eval (addE e1 e2) :: s)
  end
```

```
compile-exec-eval : (e : Expr) → exec (compile e) [] ≡ [ eval e ]
compile-exec-eval e =
  begin
    exec (compile e) []
  =⟨ compile'-exec-eval e [] [] ⟩
    exec [] (eval e :: [])                                      (93)
  =⟨⟩
    eval e :: []
  =⟨⟩
    [ eval e ]
  end
```

A List of unicode characters

→	\to
λ	\lambda
×	\times
Σ	\Sigma
⊤	\top
⊥	\bot
≡	\equiv
⟨	\<
⟩	\>
∘	\o

B Further reading

- The official Agda documentation.[22]
- The Agda standard library.[23]
- Philip Wadler, Wen Kokke, and Jeremy Siek: *Programming Language Foundations in Agda*[24] (online book). This book focuses mainly on using Agda to explain core ideas from programming

[22] agda.readthedocs.io/en

[23] github.com/agda/agda-stdlib

[24] plfa.github.io

languages research, but it is also generally an excellent first introduction to Agda.

- Aaron Stump: *Verified Functional Programming in Agda*[25] (physical book, first two chapters are freely available). This book also starts from the basics and builds towards using Agda as a language to implement and verify functional programs.

[25] `morganclaypoolpublishers.com/catalog_Orig/product_info.php?cPath=24&products_id=908`

- Ulf Norell and James Chapman: *Dependently Typed Programming in Agda*[26] (tutorial in `pdf` format).

[26] `www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`

- Ana Bove and Peter Dybjer: *Dependent Types at Work*[27] (tutorial in `pdf` format).

[27] `www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf`

- Andreas Abel: *An Introduction ot Dependent Types and Agda*[28] (lecture notes).

[28] `www2.tcs.ifi.lmu.de/~abel/DepTypes.pdf`

- Andreas Abel: *Agda Equality*[29] (lecture notes).

[29] `www2.tcs.ifi.lmu.de/~abel/Equality.pdf`