Introduction to Agda

Lecture at the AFP summer school in Utrecht

Jesper Cockx

7 July 2023

Technical University Delft

Lecture plan

- A brief overview of formal verification, dependent types, and Agda
- Differences between Agda and Haskell
- Types as first-class values
- Dependent data types
- Dependent function types
- The Curry-Howard correspondence
- Equational reasoning in Agda

"Program testing can be used to show the presence of bugs, but never to show their absence!"

– Edsger W. Dijkstra

When testing is just not enough

Question. In what situations might testing not be enough to ensure software works correctly?

When testing is just not enough

Question. In what situations might testing not be enough to ensure software works correctly?

- … failure is very costly (e.g. spacecraft, medical equipment, self-driving cars)
- … the software is difficult to update (e.g. embedded software)
- … it is security-sensitive (e.g. banking, your private chats)
- … errors are hard to detect or not apparent until much later (e.g. compilers, concurrent systems)

Formal verification

Formal verification is a collection of techniques for proving correctness of programs with respect to a certain formal specification.

These techniques often rely on ideas from formal logic and mathematics to ensure a very high degree of trustworthiness.

Why dependent types?

Dependent types are a form of formal verification that is embedded in the programming language.

Advantages.

- No different syntax to learn or tools to install
- Tight integration between IDE and type system
- Express invariants of programs in their types
- Use same syntax for programming and proving

Formally verifying a program should not be more difficult than writing the program in the first place!

The Agda language

Agda is a purely functional programming language similar to Haskell.

Unlike Haskell, it has full support for dependent types.

It also supports interactive programming with help from the type checker.

Installing Agda

VS Code plugin.
Install the `agda-mode` plugin and enable the Agda Language Server in the settings.

Binary release. (Linux/WSL)
`sudo apt install agda`

From source. (Cabal/Stack)
`cabal install Agda` or
`stack install Agda`

Installing an editor for Agda

The following editors have support for Agda:

- **VS Code**: Install the `agda-mode` plugin
- **Emacs**: Plugin is distributed with Agda (run `agda-mode setup`)
- **Atom**: `https://atom.io/packages/agda-mode`
- **Vim**: `https://github.com/derekelkins/agda-vim`

A first Agda program

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

This program:

- Defines a datatype Greeting with one constructor hello.
- Defines a function greet of type Greeting that returns hello.

Loading an Agda file

You can load an Agda file by pressing `Ctrl+c` followed by `Ctrl+l`.

Once the file is loaded (and there are no errors), other commands become available:

`Ctrl+c Ctrl+d`	Infer type of an expression.
`Ctrl+c Ctrl+n`	Evaluate an expression.

Agda vs. Haskell

Basic syntax differences

Typing uses a single colon:
b : Bool instead of b :: Bool.

Naming has fewer restrictions: any name can start with small or capital letter, and symbols can occur in names.

Whitespace is required more often: 1+1 is a valid function name, so you need to write 1 + 1 instead.

Infix operators are indicated by underscores: _+_ instead of (+)

Unicode syntax

Agda allows unicode characters in its syntax:

- $\to$ can be used instead of `->`
- λ can be used instead of `\`
- Other symbols can also be used as (parts of) names of functions, variables, or types: $\times$, Σ, $\top$, $\bot$, $\equiv$, $\langle$, $\rangle$, $\circ$, …

Entering unicode

Editors with Agda support will replace LaTeX-like syntax (e.g. `\to`) with unicode:

→	`\to`
λ	`\lambda`
×	`\times`
Σ	`\Sigma`
⊤	`\top`
⊥	`\bot`
≡	`\equiv`
...	

Quiz question

Question. Which is NOT a valid name for an Agda function?

1. `1+1=2`
2. `foo bar`
3. $\lambda\!\rightarrow\!\times$Σ
4. `if_then_else_`

Declaring new datatypes

To declare a datatype in Agda, we need to give the full type of each constructor:

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

We also need to specify that Bool itself has type Set (see later).

Defining functions by pattern matching

Just as in Haskell, we can define new functions by pattern matching:

$$
\begin{aligned}
&\mathsf{not} : \mathsf{Bool} \to \mathsf{Bool} \\
&\mathsf{not}\ \mathsf{true}\ \ = \mathsf{false} \\
&\mathsf{not}\ \mathsf{false} = \mathsf{true}
\end{aligned}
$$

The type of natural numbers

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
{-# BUILTIN NATURAL Nat #-}

one   = 1 -- = suc zero
two   = 2 -- = suc one
three = 3 -- = suc two
four  = 4 -- = suc three
```

Functions on natural numbers

```
isEven : Nat → Bool
isEven zero          = true
isEven (suc zero)    = false
isEven (suc (suc x)) = isEven x

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

Holes in programs

A hole is a part of a program that is not yet complete. A hole can be created by writing `?` or `{!!}` and loading the file (`Ctrl+c Ctrl+l`).

New commands for files with holes:

`Ctrl+c Ctrl+,`	Give information about the hole
`Ctrl+c Ctrl+c`	Case split on a variable
`Ctrl+c Ctrl+space`	Give a solution for the hole

Exercise. Use these to define the function maximum : Nat → Nat → Nat.

Total functional programming

In contrast to Haskell, Agda is a total language:

- **NO** runtime errors
- **NO** incomplete pattern matches
- **NO** non-terminating functions

So functions are true functions in the mathematical sense: evaluating a function call always returns a result in finite time.

Why should we care about totality?

Some reasons to write total programs:

- Better guarantees of correctness
- Spend less time debugging infinite loops
- Easier to refactor without introducing bugs
- Less need to document valid inputs

Totality is also crucial for working with dependent types and using Agda as a proof assistant (see later).

Coverage checking

Agda performs a coverage check to ensure all definitions by pattern matching are complete:

```
pred : Nat → Nat
pred (suc x) = x
```

Incomplete pattern matching for pred.
Missing cases: pred zero

Termination checking

Agda performs a termination check to ensure all recursive definitions are terminating:

$$\mathsf{inf} : \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathsf{inf}\ x = 1 + \mathsf{inf}\ x$$

Termination checking failed for the following functions: inf
Problematic calls: inf x

To solve or not to solve the halting problem

Question. Isn't it impossible to determine whether a function is terminating? Or does Agda solve the halting problem?

To solve or not to solve the halting problem

Question. Isn't it impossible to determine whether a function is terminating? Or does Agda solve the halting problem?

Answer. No, Agda only accepts functions that are 'obviously terminating', and rejects all other functions.

Structural recursion

Agda only accepts functions that are structurally recursive: the argument of each recursive call must be a subterm of the argument on the left of the clause.

For example, this definition is rejected:

```
f : Nat → Nat
f (suc (suc x)) = f zero
f (suc x)       = f (suc (suc x))
f zero          = zero
```

Types as first-class values

The type `Set`

In Agda, types such as Nat and (Bool → Bool) are themselves expressions of type Set.

We can pass around and return values of type Set just like values of any other type.

Example. Defining a type alias as a function:

```
MyNat : Set
MyNat = Nat

myFour : MyNat
myFour = 4
```

Polymorphic functions in Agda

We can define polymorphic functions as functions that take an argument of type Set:

$$\mathsf{id} : (A : \mathsf{Set}) \to A \to A$$
$$\mathsf{id}\ A\ x = x$$

For example, we have $\mathsf{id}\ \mathsf{Nat}\ \mathsf{zero} : \mathsf{Nat}$ and $\mathsf{id}\ \mathsf{Bool}\ \mathsf{true} : \mathsf{Bool}$.

Hidden arguments

To avoid repeating the type at which we apply a polymorphic function, we can declare it as a hidden argument using curly braces:

$$\mathsf{id} : \{A : \mathsf{Set}\} \to A \to A$$
$$\mathsf{id}\ x = x$$

Now we have $\mathsf{id\ zero} : \mathsf{Nat}$ and $\mathsf{id\ true} : \mathsf{Bool}$.

If/then/else as a function

We can define if/then/else in Agda as follows:

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

This is an example of a mixfix operator.

Example usage.

```
test : Nat → Nat
test x = if (x ≤ 9000) then 0 else 42
```

Polymorphic datatypes

Just like we can define polymorphic functions, we can also define polymorphic datatypes by adding a parameter (A : Set):

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A
infixr 5 _::_
```

Note. Agda does not have built-in support for list syntax [1, 2, 3]. Instead, we have to write 1 :: 2 :: 3 :: [].

A tuple type in Agda

Agda does not have a builtin type of tuples (x, y), but we can define the product type $A \times B$:

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

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

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

No pattern matching on `Set`

It is not allowed to pattern match on arguments of type Set:

```
-- Not valid code:
sneakyType : Set → Set
sneakyType Bool = Nat
sneakyType Nat  = Bool
```

One reason for this is that Agda (like Haskell) erases all types during compilation.

Quiz question

Is it possible to implement a function of type $\{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{Nat} \to A$ in Agda?

Dependent types

Cooking with dependent types (1/3)

Suppose we are implementing a cooking assistant that can help with preparing three kinds of food:

```
data Food : Set where
  pizza  : Food
  cake   : Food
  bread : Food
```

We want to implement a function amountOfCheese : Food → Nat that computes how much cheese is needed.

Problem: How can we make sure this function is never called with argument cake?

Cooking with dependent types (2/3)

Solution. We can make the type Food more precise making it into an indexed datatype:

```
data Flavour : Set where
  cheesy     : Flavour
  chocolatey : Flavour

data Food : Flavour → Set where
  pizza : Food cheesy
  cake  : Food chocolatey
  bread : {f : Flavour} → Food f
```

This defines two types Food cheesy and Food chocolatey.

Cooking with dependent types (3/3)

We can now rule out invalid inputs by using the more precise type Food cheesy:

```
amountOfCheese : Food cheesy → Nat
amountOfCheese pizza  = 100
amountOfCheese bread = 20
```

The coverage checker of Agda knows that cake is not a valid input!

Dependent type theory (1972)

Per
Martin-Löf

A dependent type is a family of types, depending on a term of a base type.

Dependent type theory (1972)

Per Martin-Löf

A dependent type is a family of types, depending on a term of a base type.

Example (not by Martin-Löf). Food is a dependent type indexed over the base type Flavour.

Vectors: lists that know their length

Vec A n is the type of vectors with exactly n arguments of type A:

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

myVec2 : Vec Nat 0
myVec2 = []

myVec3 : Vec (Bool → Bool) 2
myVec3 = not :: id :: []
```

Definition of the `Vec` type

Vec *A* *n* is a dependent type indexed over the base type Nat:

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

This has two constructors [] and _::_ like List, but the constructors specify the length in their types.

Parameters vs. indices

The argument ($A : Set$) in the definition of Vec is a parameter, and has to be *the same in the type of each constructor*.

The argument of type Nat in the definition of Vec is an index, and must be *determined individually for each constructor*.

Quiz question

Question. How many elements are there in the type Vec Bool 3?

Quiz question

Question. How many elements are there in the type Vec Bool 3?

Answer. 8 elements:

- true :: true :: true :: []
- true :: true :: false :: []
- true :: false :: true :: []
- true :: false :: false :: []
- false :: true :: true :: []
- false :: true :: false :: []
- false :: false :: true :: []
- false :: false :: false :: []

Type-level computation

During type-checking, Agda will evaluate expressions in types:

```
myVec4 : Vec Nat (2 + 2)
myVec4 = 1 :: 2 :: 3 :: 4 :: []
```

Since Agda is a total language, any expression can appear inside a type.

(A non-total language with dependent types would only allow a few ‘safe’ expressions.)

Checking the length of a vector

Constructing a vector of the wrong length in any way is a type error:

```
myVec5 : Vec Nat 0
myVec5 = 1 :: 2 :: []
```

```
suc _n_46 != zero of type Nat
when checking that the inferred
type of an application
Vec Nat (suc _n_46)
matches the expected type
Vec Nat 0
```

Dependent functions

Dependent function types

A dependent function type is a type of the form $(x : A) \to B\ x$ where the *type* of the output depends on the *value* of the input.

Example.

$$\begin{aligned}&\mathsf{zeroes} : (n : \mathsf{Nat}) \to \mathsf{Vec}\ \mathsf{Nat}\ n\\&\mathsf{zeroes}\ \mathsf{zero} \qquad = []\\&\mathsf{zeroes}\ (\mathsf{suc}\ n) = 0 :: \mathsf{zeroes}\ n\end{aligned}$$

E.g. $\mathsf{zeroes}\ 3$ has type $\mathsf{Vec}\ \mathsf{Nat}\ 3$ and evaluates to $0 :: 0 :: 0 :: []$.

Concatenation of vectors

We can pattern match on Vec just like on List:

```
mapVec : {A B : Set} {n : Nat} →
  (A → B) → Vec A n → Vec B n
mapVec f []        = []
mapVec f (x :: xs) = f x :: mapVec f xs
```

Note. The type of mapVec specifies that the output has the same length as the input.

A safe `head` function

By making the input type of a function more precise, we can rule out certain cases statically (= during type checking):

$$\mathsf{head} : \{A : \mathsf{Set}\}\{n : \mathsf{Nat}\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A$$
$$\mathsf{head}\ (x :: xs) = x$$

Agda knows the case for head [] is impossible!
(just like for amountOfCheese cake)

A safe `tail` function

Question. What should be the type of tail on vectors with the following definition?

$$\text{tail}\ (x :: xs) = xs$$

A safe `tail` function

Question. What should be the type of tail on vectors with the following definition?

$$\text{tail}\ (x :: xs) = xs$$

Answer.

$$\text{tail} : \{A : \text{Set}\}\ \{n : \text{Nat}\} \to \text{Vec}\ A\ (\text{suc}\ n) \to \text{Vec}\ A\ n$$
$$\text{tail}\ (x :: xs) = xs$$

Exercise

Define a function zipVec that only accepts vectors of the same length.

A safe lookup

By combining head and tail, we can get the 1st, 2nd, 3rd,…element of a vector with at least that many elements.

How can we define a function lookupVec that get the element at position i of a Vec A n where $i < n$?

Note. We want to get an element of A, *not* of Maybe A!

The `Fin` type

We need a type of indices that are *safe* for a vector of length n, i.e. numbers between 0 and $n - 1$.

This is the type Fin n of finite numbers:

```
zero3 one3 two3 : Fin 3
zero3 = zero
one3  = suc zero
two3  = suc (suc zero)
```

Definition of the `Fin` type

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

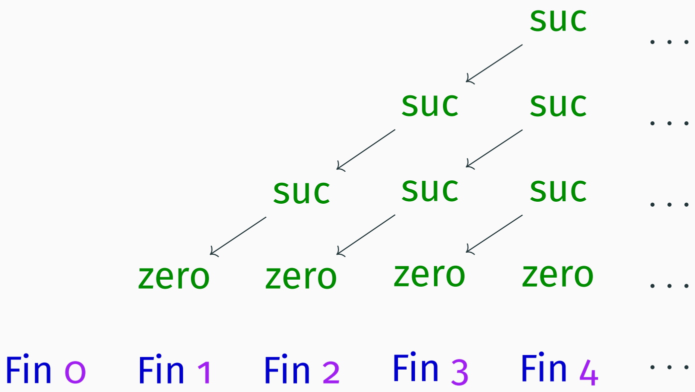
An empty type

Fin *n* has *n* elements, so in particular Fin 0 has *zero* elements: it is an empty type.

This means there are *no valid indices* for a vector of length 0.

Note. Unlike in Haskell, we cannot even construct an expression of Fin 0 using `undefined` or an infinite loop.

The family of `Fin` types



A safe lookup (1/5)

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

A safe lookup (2/5)

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

A safe lookup (3/5)

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

A safe lookup (4/5)

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

A safe lookup (5/5)

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

We now have a safe and total version of the Haskell `(!!)` function, without having to change the return type in any way.

Exercise (1/2)

Define a datatype Expr of expressions of a small programming language with:

- Number literals $0, 1, 2, \dots$
- Arithmetic expressions $e_1 + e_2$ and $e_1 * e_2$
- Booleans true and false
- Comparisons $e_1 < e_2$ and $e_1 == e_2$
- Conditionals `if` e_1 `then` e_2 `else` e_3

Expr should be a *dependent type* indexed over the type Ty of possible types of this language:

```
data Ty : Set where
  tInt  : Ty
  tBool : Ty
```

Exercise (2/2)

Next, write a function $\mathsf{El} : \mathsf{Ty} \to \mathsf{Set}$ that interprets a type of this language as an Agda type.

Finally, define $\mathsf{eval} : \{t : \mathsf{Ty}\} \to \mathsf{Expr}\ t \to \mathsf{El}\ t$ that evaluates a given expression to an Agda value.

Dependent types: Summary

A dependent type is a type that *depends on* a value of some base type.

With dependent types, we can specify the allowed inputs of a function more precisely, ruling out invalid inputs at compile time.

Examples of dependent types.

- Food f, indexed over f : Flavour
- Vec A n, indexed over n : Nat
- Fin n, indexed over n : Nat
- Expr t, indexed over t : Ty

The Curry-Howard Correspondence

"Every good idea will be discovered twice: once by a logician and once by a computer scientist."

– Philip Wadler

Formal verification with dependent types

Agda is not just a programming language but also a proof assistant for verifying properties:

- For any x : Nat, $x + x$ is an even number.
- length (map f xs) = length xs
- foldr (λ x xs $\rightarrow$ xs ++ x) [] xs
  = foldl (λ xs x $\rightarrow$ x :: xs) [] xs

To do this, we first need to answer the question: what exactly is a proof?

What even is a proof? (1/3)

In mathematics, a proof is a sequence of statements where each statement is a direct consequence of previous statements.

Example. A proof that if (1) $A \Rightarrow B$ and (2) $A \wedge C$, then $B \wedge C$:

(3)	A	(follows from 2)
(4)	B	(modus ponens with 1 and 3)
(5)	C	(follows from 2)
(6)	$B \wedge C$	(follows from 4 and 5)

What even is a proof? (2/3)

We can make the dependencies of a proof more explicit by writing it down as a proof tree.

Example. Here is the same proof that if (1) $A \Rightarrow B$ and (2) $A \wedge C$, then $B \wedge C$:

$$\dfrac{\dfrac{A \Rightarrow B^{(1)} \qquad \dfrac{A \wedge C^{(2)}}{A}}{B} \qquad \dfrac{A \wedge C^{(2)}}{C}}{B \wedge C}$$

What even is a proof? (3/3)

To represent these proofs in a programming language, we can annotate each node of the tree with a proof term:

$$\dfrac{p : A \Rightarrow B \qquad \dfrac{q : A \wedge C}{\mathsf{fst}\ q : A}}{p\ (\mathsf{fst}\ q) : B} \qquad \dfrac{q : A \wedge C}{\mathsf{snd}\ q : C}$$

$$\dfrac{p\ (\mathsf{fst}\ q) : B \qquad \mathsf{snd}\ q : C}{(p\ (\mathsf{fst}\ q), \mathsf{snd}\ q) : B \wedge C}$$

What even is a proof? (3/3)

To represent these proofs in a programming language, we can annotate each node of the tree with a proof term:

$$\dfrac{\dfrac{p : A \Rightarrow B \qquad \dfrac{q : A \wedge C}{\mathsf{fst}\ q : A}}{p\ (\mathsf{fst}\ q) : B} \qquad \dfrac{q : A \wedge C}{\mathsf{snd}\ q : C}}{(p\ (\mathsf{fst}\ q), \mathsf{snd}\ q) : B \wedge C}$$

Hmm, these proof terms start to look a lot like functional programs...

The Curry-Howard correspondence

Haskell B. Curry

We can interpret logical propositions ($A \wedge B$, $\neg A$, $A \Rightarrow B, \ldots$) as the types of all their possible proofs.

In particular: A false proposition has no proofs, so it corresponds to an empty type.

What is conjunction $A \wedge B$?

What do we know about the proposition $A \wedge B$ (A and B)?

- To prove $A \wedge B$, we need to provide a proof of A and a proof of B.
- Given a proof of $A \wedge B$, we can get proofs of A and B

What is conjunction $A \wedge B$?

What do we know about the proposition $A \wedge B$ (A and B)?

- To prove $A \wedge B$, we need to provide a proof of A and a proof of B.
- Given a proof of $A \wedge B$, we can get proofs of A and B

$\Rightarrow$ The type of proofs of $A \wedge B$ is the type of pairs $A \times B$

What is implication $A \Rightarrow B$?

What do we know about the proposition $A \Rightarrow B$ (A implies B)?

- To prove $A \Rightarrow B$, we can assume we have a proof of A and have to provide a proof of B
- From a proof of $A \Rightarrow B$ and a proof of A, we can get a proof of B

What is implication $A \Rightarrow B$?

What do we know about the proposition $A \Rightarrow B$ (A implies B)?

- To prove $A \Rightarrow B$, we can assume we have a proof of A and have to provide a proof of B
- From a proof of $A \Rightarrow B$ and a proof of A, we can get a proof of B

$\Rightarrow$ The type of proofs of $A \Rightarrow B$ is the function type $A \to B$

Proof by implication (Modus ponens)

Modus ponens says that if P implies Q and P is true, then Q is true.

Question. How can we prove this in Agda?

Proof by implication (Modus ponens)

Modus ponens says that if P implies Q and P is true, then Q is true.

Question. How can we prove this in Agda?

Answer.

```
modusPonens : {P Q : Set} → (P → Q) × P → Q
modusPonens (f , x) = f x
```

What is disjunction $A \vee B$?

What do we know about the proposition $A \vee B$ (A or B)?

- To prove $A \vee B$ we need to provide a proof of A or a proof of B.
- If we have:
  - a proof of $A \vee B$
  - a proof of C assuming a proof of A
  - a proof of C assuming a proof of B

  then we have a proof of C.

What is disjunction $A \vee B$?

What do we know about the proposition $A \vee B$ (A or B)?

- To prove $A \vee B$ we need to provide a proof of A or a proof of B.
- If we have:
  - a proof of $A \vee B$
  - a proof of C assuming a proof of A
  - a proof of C assuming a proof of B

  then we have a proof of C.

$\Rightarrow$ The type of proofs of $A \vee B$ is the sum type Either A B

Proof by cases

Proof by cases says that if $P \vee Q$ is true and we can prove R from P and also prove R from Q, then we can prove R.

Question. How can we prove this in Agda?

Proof by cases

Proof by cases says that if $P \vee Q$ is true and we can prove R from P and also prove R from Q, then we can prove R.

Question. How can we prove this in Agda?

Answer.

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

Quiz question

Question. Which Agda type represents the proposition *"If (P implies Q) then (P or R) implies (Q or R)"*?

1. $(\text{Either}\ P\ Q) \to \text{Either}\ (P \to R)\ (Q \to R)$
2. $(P \to Q) \to \text{Either}\ P\ R \to \text{Either}\ Q\ R$
3. $(P \to Q) \to \text{Either}\ (P \times R)\ (Q \times R)$
4. $(P \times Q) \to \text{Either}\ P\ R \to \text{Either}\ Q\ R$

What is truth?

What do we know about the proposition ‘true’?

- To prove ‘true’, we don’t need to provide anything
- From ‘true’, we can deduce nothing

What is truth?

What do we know about the proposition ‘true’?

- To prove ‘true’, we don’t need to provide anything
- From ‘true’, we can deduce nothing

$\Rightarrow$ The type of proofs of truth is the *unit type* $\top$ with one constructor tt:

```
data ⊤ : Set where
  tt : ⊤
```

What is falsity?

What do we know about the proposition ‘false’?

- There is no way to prove ‘false’
- From a proof t of ‘false’, we get a proof `absurd` t of any proposition A

What is falsity?

What do we know about the proposition ‘false’?

- There is no way to prove ‘false’
- From a proof t of ‘false’, we get a proof `absurd` t of any proposition A

$\Rightarrow$ The type of proofs of falsity is the empty type $\bot$ with no constructors:

```
data ⊥ : Set where
```

Principle of explosion

The principle of explosion[1] says that if we assume a false statement, we can prove any proposition P.

Question. How can we prove this in Agda?

[1] Also known as *ex falso quodlibet* = *from falsity follows anything*.

Principle of explosion

The principle of explosion[1] says that if we assume a false statement, we can prove any proposition P.

Question. How can we prove this in Agda?

Answer.

```
absurd : {P : Set} → ⊥ → P
absurd ()
```

[1] Also known as *ex falso quodlibet* = *from falsity follows anything*.

Curry-Howard for propositional logic

We can translate from the language of logic to the language of types according to this table:

Propositional logic		Type system
proposition	P	type
proof of a proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	$\text{Either}\ P\ Q$	either type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	empty type

Derived notions

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

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

Exercise

Translate the following statements to types in Agda, and prove them by constructing a program of that type:

1. If P implies Q and Q implies R, then P implies R
2. If P is false and Q is false, then (either P or Q) is false.
3. If P is both true and false, then any proposition Q is true.

Constructive logic

In classical logic we can prove certain 'non-constructive' statements:

- $P \vee (\neg P)$ (excluded middle)
- $\neg\neg P \Rightarrow P$ (double negation elimination)

However, Agda uses a constructive logic: a proof of $A \vee B$ gives us a decision procedure to tell whether A or B holds.

When P is unknown, it's impossible to decide whether P or $\neg P$ holds, so the excluded middle is unprovable in Agda.

From classical to constructive logic

Consider the proposition P ("P is true") vs. $\neg\neg P$ ("It would be absurd if P were false").

Classical logic can't tell the difference between the two, but constructive logic can.

Theorem (Gödel and Gentzen). P is provable in classical logic if and only if $\neg\neg P$ is provable in constructive logic.

Curry-Howard beyond simple types

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Curry-Howard beyond simple types

- Classical logic corresponds to continuations (e.g. Lisp)

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Curry-Howard beyond simple types

- Classical logic corresponds to continuations (e.g. Lisp)
- Linear logic corresponds to linear types (e.g. Rust)

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Curry-Howard beyond simple types

- Classical logic corresponds to continuations (e.g. Lisp)
- Linear logic corresponds to linear types (e.g. Rust)
- Predicate logic corresponds to dependent types (e.g. Agda)

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Defining predicates

Question. How would you define a type that expresses that a given number n is even?

Defining predicates

Question. How would you define a type that expresses that a given number *n* is even?

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

6-is-even : IsEven 6
6-is-even = e-suc2 (e-suc2 (e-suc2 e-zero))

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

Defining predicates

To define a predicate *P* on elements of type *A*, we can define *P* as a dependent datatype with base type *A*:

```
data P : A → Set where
  c₁ : ··· → P a₁
  c₂ : ··· → P a₂
  – ...
```

Universal quantification

What do we know about the proposition $\forall(x \in A).\ P(x)$ ('for all x in A, $P(x)$ holds')?

- To prove $\forall(x \in A).\ P(x)$, we assume we have an unknown $x \in A$ and prove that $P(x)$ holds.
- If we have a proof of $\forall(x \in A).\ P(x)$ and a concrete $a \in A$, then we know $P(a)$.

Universal quantification

What do we know about the proposition $\forall(x \in A).\ P(x)$ ('for all x in A, $P(x)$ holds')?

- To prove $\forall(x \in A).\ P(x)$, we assume we have an unknown $x \in A$ and prove that $P(x)$ holds.
- If we have a proof of $\forall(x \in A).\ P(x)$ and a concrete $a \in A$, then we know $P(a)$.

$\Rightarrow \forall(x \in A).\ P(x)$ corresponds to the dependent function type $(x : A) \to P\ x$.

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

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

double-even : (n : Nat) → IsEven (double n)
double-even n = {!!}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

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

double-even : (n : Nat) → IsEven (double n)
double-even zero    = {!!}
double-even (suc m) = {!!}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

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

double-even : (n : Nat) → IsEven (double n)
double-even zero    = e-zero
double-even (suc m) = {!!}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

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

double-even : (n : Nat) → IsEven (double n)
double-even zero     = e-zero
double-even (suc m) = e-suc2 {!!}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

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

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

Induction in Agda

In general, a proof by induction on natural numbers in Agda looks like this:

```
proof : (n : Nat) → P n
proof zero    = ···
proof (suc n) = ···
```

- proof zero is the base case
- proof (suc n) is the inductive case

When proving the inductive case, we can make use of the induction hypothesis proof $n : P\ n$.

Proving things about programs

General rule of thumb: A proof about a function often follows the same structure as that function:

- To prove something about a function by pattern matching, the proof will also use pattern matching (= proof by cases)
- To prove something about a recursive function, the proof will also be recursive (= proof by induction)

On the need for totality

To ensure the proofs we write are correct, we rely on the totality of Agda:

- The coverage checker ensures that a proof by cases covers all cases.
- The termination checker ensures that inductive proofs are well-founded.

The identity type and equational reasoning

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

– Donald Knuth

The identity type

The identity type $x \equiv y$ says x and y are equal:

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

The constructor refl proves that two terms are equal if they have the same normal form:

```
one-plus-one : 1 + 1 ≡ 2
one-plus-one = refl
```

Application of the identity type: Writing test cases

One use case of the identity type is for writing test cases:

$$\mathrm{test}_1 : \mathrm{length}\ (42 :: []) \equiv 1$$
$$\mathrm{test}_1 = \mathrm{refl}$$

$$\mathrm{test}_2 : \mathrm{length}\ (\mathrm{map}\ (1 +\_)\ (0 :: 1 :: 2 :: [])) \equiv 3$$
$$\mathrm{test}_2 = \mathrm{refl}$$

The test cases are run each time the file is loaded!

Proving correctness of functions

We can use the identity type to prove the correctness of functional programs.

Example. Prove that not (not b) $\equiv$ b for all b : Bool:

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

Exercise

Write down the Agda type expressing the statement that for any function *f* and list *xs*, length (map *f* *xs*) is equal to length *xs*.

Then, prove it by implementing a function of that type.

Quiz question

Question. What is the type of the Agda expression $\lambda\, b \to (b \equiv \text{true})$?

1. $\text{Bool} \to \text{Bool}$
2. $\text{Bool} \to \text{Set}$
3. $(b : \text{Bool}) \to b \equiv \text{true}$
4. It is not a well-typed expression

Pattern matching on refl

If we have a proof of $x \equiv y$ as input, we can pattern match on the constructor refl to show Agda that x and y are equal:

```
castVec : {A : Set} {m n : Nat} →
  m ≡ n → Vec A m → Vec A n
castVec refl xs = xs
```

When you pattern match on refl, Agda applies unification to the two sides of the equality.

Symmetry of equality

Symmetry states that if x is equal to y, then y is equal to x:

$$\text{sym} : \{A : \text{Set}\}\ \{x\ y : A\} \to x \equiv y \to y \equiv x$$
$$\text{sym refl} = \text{refl}$$

Congruence

Congruence states that if f : $A \to B$ is a function and x is equal to y, then f x is equal to f y:

$$\text{cong} : \{A\ B : \text{Set}\}\ \{x\ y : A\} \to$$
$$(f : A \to B) \to x \equiv y \to f\ x \equiv f\ y$$
$$\text{cong}\ f\ \text{refl} = \text{refl}$$

Equational reasoning

In school, we learned how to prove equations by chaining basic equalities:

```
  (a + b) (a + b)
= a (a + b) + b (a + b)
= a^2 + ab + ba + b^2
= a^2 + ab + ab + b^2
= a^2 + 2ab + b^2
```

This style of proving is called equational reasoning.

Equational reasoning about functional programs

Equational reasoning is well suited for proving things about pure functions:

```
  head (replicate 100 "spam")
= head ("spam" : replicate 99 "spam")
= "spam"
```

Because there are no side effects, everything is explicit in the program itself.

Equational reasoning in Agda

Consider the following definitions:

$$
\begin{aligned}
&[\_] : \{A : \mathsf{Set}\} \to A \to \mathsf{List}\ A \\
&[\ x\ ] = x :: []
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reverse} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A \\
&\mathsf{reverse}\ [] = [] \\
&\mathsf{reverse}\ (x :: xs) = \mathsf{reverse}\ xs\ \mathbin{++}\ [\ x\ ]
\end{aligned}
$$

Goal. Prove that $\mathsf{reverse}\ [\ x\ ] = [\ x\ ]$.

Example ‘on paper’

```
  reverse [ x ]
=     { definition of [_] }
  reverse (x :: [])
=     { applying reverse (second clause) }
  reverse [] ++ [ x ]
=     { applying reverse (first clause) }
  [] ++ [ x ]
=     { applying _++_ }
  [ x ]
```

Example in Agda

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

Equational reasoning in Agda

We can write down an equality proof in equational reasoning style in Agda:

- The proof starts with begin and ends with end.
- In between is a sequence of expressions separated by =⟨⟩, where each expression is equal to the previous one.

Unlike the proof on paper, here the typechecker of Agda guarantees that each step of the proof is correct!

Behind the scenes

Each proof by equational reasoning can be desugared to refl (and trans).

Example.

```
reverse-singleton : {A : Set} (x : A) →
  reverse [ x ] ≡ [ x ]
reverse-singleton x = refl
```

However, proofs by equational reasoning are much easier to read and debug.

Equational reasoning + case analysis

We can use equational reasoning in a proof by case analysis (i.e. pattern matching):

```
not-not : (b : Bool) → not (not b) ≡ b
not-not false =
  begin
    not (not false)
  =⟨⟩           -- applying the inner not
    not true
  =⟨⟩           -- applying not
    false
  end
not-not true = {!!} -- similar to above
```

Equational reasoning + induction

We can use equational reasoning in a proof by induction:

```
add-n-zero : (n : Nat) → n + zero ≡ n
add-n-zero zero    = {!!}         -- easy exercise
add-n-zero (suc n) =
  begin
    (suc n) + zero
  =⟨⟩                             -- applying +
    suc (n + zero)
  =⟨ cong suc (add-n-zero n) ⟩    -- using IH
    suc n
  end
```

Here we have to provide an explicit proof that suc (n + zero) = suc n (between the =⟨ and ⟩).

Exercise

State and prove associativity of addition on natural numbers: $x + (y + z) = (x + y) + z$

Hint. If you get stuck, try to work instead backwards from the goal you want to reach!

Application 1: Proving type class laws

Reminder: functor laws

Remember the two functor laws from Haskell:

- fmap id = id
- fmap $(f \,.\, g)$ = fmap f . fmap g

In Haskell we could only verify these laws by hand for each instance, but in Agda we can prove that they hold.

First functor law for List (base case)

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

First functor law for List (inductive case)

```
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 IH
    x :: xs
  end
```

Exercise

Prove the second functor law for List.

First, we need to define function composition:[2]

$$\_\circ\_ : \{A\ B\ C : \mathsf{Set}\} \to$$
$$(B \to C) \to (A \to B) \to (A \to C)$$
$$f \circ g = \lambda\ x \to f\ (g\ x)$$

Now we can prove that
$\mathsf{map}\ (f \circ g)\ x = (\mathsf{map}\ f \circ \mathsf{map}\ g)\ x$.

[2] Unicode input for ∘: `\circ`

Application 2: Verifying optimizations

Reminder: working with accumulators

A slow version of reverse in $O(n^2)$:

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

A faster version of reverse in $O(n)$:

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

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

How can we be sure they are equivalent? By proving it!

Equivalence of reverse and reverse'

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

Proving the lemma (base case)

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

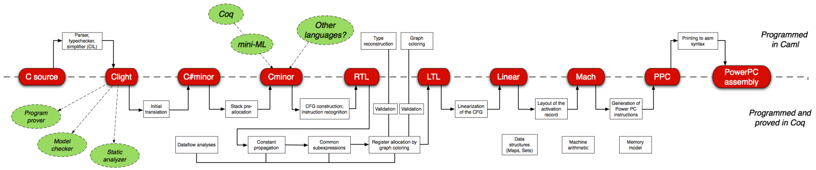
Proving the lemma (inductive case)

```
reverse-acc-lemma (x :: xs) ys =
  begin
    reverse-acc (x :: xs) ys
  =⟨⟩                              -- def of reverse-acc
    reverse-acc xs (x :: ys)
  =⟨ reverse-acc-lemma xs (x :: ys) ⟩
    reverse xs ++ (x :: ys)        -- ^ using IH
  =⟨⟩                              -- unapplying ++
    reverse xs ++ ([ x ] ++ ys)
  =⟨ sym (append-assoc (reverse xs) [ x ] ys) ⟩
    (reverse xs ++ [ x ]) ++ ys    -- ^ associativity of ++
  =⟨⟩                              -- unapplying reverse
    reverse (x :: xs) ++ ys
  end
```

Application 3: Proving compiler correctness

Real-world application: The CompCert C compiler

CompCert is an optimizing compiler for C code, which is formally proven to be correct according to the semantics of the C language, using the dependently typed language Coq.



To learn more: `https://compcert.org/`

A simple expression language

```
data Expr : Set where
  valE  : Nat → Expr
  addE : Expr → Expr → Expr

-- Example expr: (2 + 3) + 4
expr : Expr
expr = addE (addE (valE 2) (valE 3)) (valE 4)

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

Evaluating expressions using a stack

```
data Op : Set where
  PUSH  : Nat → Op
  ADD   : Op

Stack = List Nat
Code  = List Op

-- Example code for (2 + 3) + 4
code : Code
code = PUSH 2 :: PUSH 3 :: ADD
          :: PUSH 4 :: ADD :: []
```

Executing compiled code

Given a list of instructions and an initial stack, we can execute the code:

$$
\begin{aligned}
&\mathsf{exec} : \mathsf{Code} \to \mathsf{Stack} \to \mathsf{Stack} \\
&\mathsf{exec}\ [\,]\ s = s \\
&\mathsf{exec}\ (\mathsf{PUSH}\ x :: c)\ s = \mathsf{exec}\ c\ (x :: s) \\
&\mathsf{exec}\ (\mathsf{ADD} :: c)\ (m :: n :: s) = \mathsf{exec}\ c\ (n + m :: s) \\
&\mathsf{exec}\ (\mathsf{ADD} :: c)\ \_ = [\,]
\end{aligned}
$$

Compiling expressions

Goal. Compile an expression to a list of stack instructions.

A first attempt.

```
comp : Expr → Code
comp (valE x)       = [ PUSH x ]
comp (addE e1 e2) =
  comp e1 ++ comp e2 ++ [ ADD ]
```

Problem. This is very inefficient ($O(n^2)$) due to the repeated use of _++_!

Compiling with an accumulator

Problem. This is very inefficient ($O(n^2)$) due to the repeated use of `_++_`!

Instead, we can use an accumulator for the already generated code:

```
comp' : Expr → Code → Code
comp' (valE x)       c = PUSH x :: c
comp' (addE e1 e2) c =
  comp' e1 (comp' e2 (ADD :: c))

comp : Expr → Code
comp e = comp' e []
```

Proving correctness of comp

We want to prove that executing the compiled code has the same result as evaluating the expression directly:

```
comp-exec-eval : (e : Expr) → exec (comp e) [] ≡ [ eval e ]
comp-exec-eval e =
  begin
    exec (comp e) []
  =⟨ comp'-exec-eval e [] [] ⟩ -- (see next slide)
    exec [] (eval e :: [])
  =⟨⟩                          -- applying exec for []
    eval e :: []
  =⟨⟩                          -- unapplying [_]
    [ eval e ]
  end
```

Proving correctness of comp' (valE case)

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

Proving correctness of comp' (addE case)

```
comp'-exec-eval (addE e1 e2) s c =
  begin
    exec (comp' (addE e1 e2) c) s
  =⟨⟩ -- def of comp'
    exec (comp' e1 (comp' e2 (ADD :: c))) s
  =⟨ comp'-exec-eval e1 s (comp' e2 (ADD :: c)) ⟩ -- IH
    exec (comp' e2 (ADD :: c)) (eval e1 :: s)
  =⟨ comp'-exec-eval e2 (eval e1 :: s) (ADD :: c) ⟩ -- IH
    exec (ADD :: c) (eval e2 :: eval e1 :: s)
  =⟨⟩ -- applying exec for ADD
    exec c (eval e1 + eval e2 :: s)
  =⟨⟩ -- unapplying eval for addE
    exec c (eval (addE e1 e2) :: s)
  end
```