The Tao of Types

Thorsten Altenkirch

March 28, 2024

Contents

1 Introduction **7**

2 Simple Types **9**
- 2.1 Functions . . . 10
- 2.2 Identity and composition . . . 12
- 2.3 λ-calculus . . . 15
- 2.4 Combinatory logic . . . 17
- 2.5 Products and Sums . . . 19
  - 2.5.1 Sums . . . 20
  - 2.5.2 Products . . . 21
  - 2.5.3 Strong sums? . . . 22
  - 2.5.4 Finite Types . . . 23
- 2.6 History . . . 25
- 2.7 Exercises . . . 27

3 Propositional logic **29**
- 3.1 Boolean logic . . . 29
- 3.2 Propositions as types . . . 32
- 3.3 Classical principles . . . 36
- 3.4 The negative translation . . . 38
- 3.5 History . . . 40
- 3.6 Exercises . . . 40

4 Datatypes **45**
- 4.1 The natural numbers . . . 45
  - 4.1.1 Recursion . . . 47
  - 4.1.2 Arithmetic operations . . . 47
  - 4.1.3 The Ackermann function . . . 48
- 4.2 Iterator and recursor . . . 50
  - 4.2.1 Non primitive recursive functions . . . 53
- 4.3 Inductive Types . . . 54
  - 4.3.1 Lists . . . 54
  - 4.3.2 Syntax trees . . . 56
  - 4.3.3 Rose trees . . . 57

4.3.4 Ordinals . . . 58
4.4 Positivity . . . 61
4.5 Coinductive types . . . 63
4.5.1 Streams . . . 63
4.5.2 Conatural numbers . . . 65
4.5.3 Coiterator and corecursor . . . 66
4.6 History . . . 68
4.7 Exercises . . . 69

5 Dependent types **73**
5.1 Vectors and dependent functions . . . 73
5.2 The family of finite sets . . . 76
5.3 Π-types and Σ-types . . . 77
5.4 Relating simple and dependent type formers . . . 78
5.5 Arithmetic of types . . . 80
5.6 History . . . 83
5.7 Exercises . . . 84

6 Predicate logic **89**
6.1 Predicates and Quantifiers . . . 89
6.2 Some Tautologies . . . 91
6.3 Equality . . . 93
6.3.1 Propositional vs definitional equality . . . 96
6.4 Proving properties of addition . . . 97
6.5 Extending the negative translation . . . 101
6.6 History . . . 102
6.7 Exercises . . . 103

7 Programs and proofs **107**
7.1 Decidability of equality . . . 108
7.1.1 Church's thesis . . . 110
7.1.2 Uniqueness of equality proofs . . . 111
7.2 Inductively defined relations: less or equal . . . 111
7.3 An alternative definition of less or equal . . . 114
7.4 History . . . 116
7.5 Exercises . . . 118

8 Deduction **121**
8.1 The rules of natural deduction . . . 121
8.2 Semantics . . . 123
8.3 Equivalence with combinatory logic . . . 125
8.4 Kripke models . . . 129
8.5 Completeness . . . 132
8.6 Normalisation . . . 133
8.7 History . . . 136
8.8 Exercises . . . 136

9 Eliminators and coinduction **141**
9.1 The eliminator for natural numbers 141
9.2 Iterator and eliminators for dependent types 143
9.3 Eliminators for equality 146
9.3.1 The recursor 146
9.3.2 J and K 146
9.3.3 Paulin-Mohring's eliminator 149
9.4 Coinduction for streams 151
9.5 History 154
9.6 Exercises 154

Chapter 1

Introduction

Chapter 2

Simple Types

Before jumping straight into dependent types with all its beautiful complexities we will first have a look at something simpler, appropriately named *simple types*. We start of with the most fundamental type of all, the function type $_\to_$ (section 2.1).

One issue with simple types (which is repeated next chapter when we look at propositional logic) is that it is not clear what to start with since we haven't yet got any types to built functions with. For examples I will use types like $\mathbb{N}$ (the type of natural numbers: $0, 1, 2, \ldots$. We will also look at constructions which work for any type, called Set in Agda. That is we will be using *polymorphic* constructions without introducing a formal account for polymorphism which we will do later once we have dependent types. A particular example of polymorphic constructions are identity id and composition $_\circ_$ (section 2.2) which give us a first taste of category theory.

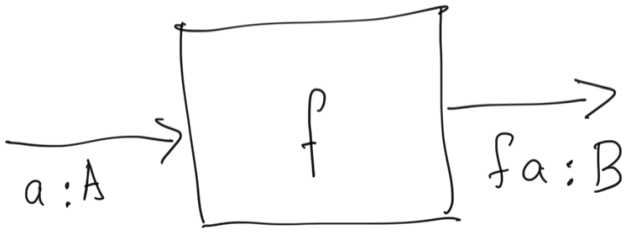
We will have a superficial look at λ-calculus (section 2.3), by which I always mean typed λ-calculus because the untyped one doesn't make any sense to me. At least enough to see that variables are causing a bit of a headache, something which can be avoided my using combinatory logic which features just two combinators (S and K), but the price we have to pay is the creation fo some completely unreadable combinator code. However, I think it is worthwhile to know about them because they are a simple example of a general pattern in Type Theory: the reduction of an intuitively appealing notation into a collection of combinators that are better suited for metatheoretic study. This is reminiscient of the reduction of complex reasoning principles to some basic axioms in set theory. We will also return to combinators later from a more formal perspective.

Functions are not all there is to simple types and hence we are going to look at sums ($_\uplus_$) and products ($_\times_$) (section 2.5 which let us encode finite types. Finally we will meet some old acquaintances from high school in the context of type theory.

2.1 Functions

In Type Theory functions are a fundamental concept. This is different from set theory where functions are viewed as a special kind of relations, namely relations that assign to each element of the domain (the input) exactly one element of the codomain (the output). In Type Theory it is the other way around: we start with functions and actually we need functions to say what a relations is, namely a relation between A and B is a function from A and B to the type of propositions. But lets not get ahead of ourselves.

Intuitively, a function between types A and B is a black box where we can input elements of $\mathsf{a} : \mathsf{A}$ and out come elements of $\mathsf{f}\,\mathsf{a} : \mathsf{B}$.



You may notice that we don't write $f(a)$ as usual in Mathematics and in many programming languages but instead save brackets and just write $\mathsf{f}\,\mathsf{a}$, which is common in functional programming languages like Haskell. We write $\mathsf{A} \to \mathsf{B}$ for the type of functions from A to B and hence we write $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ to express that f is such a function.

I was saying *black box*, even though for reasons of readability I didn't actually draw a black box. The meaning here is rather metaphorical, it means we cannot look into the box. If somebody gives you a function, all you can do it is to feed it elements and observe the output. You have no way to explore the mechanism (this is a sort of digital rights management for functions). This aspect will become important when we discuss extensionality later.

Let's look at some examples! For the purpose of illustration I will use some types like the natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ [1] [2] and the booleans $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$ even though we will only introduce them later. We will also use some basic operations such as addition $(_+_)$ on natural numbers.

Let's define a function which adds 2 to its input

$$\begin{aligned}&\mathsf{add2} : \mathbb{N} \to \mathbb{N}\\ &\mathsf{add2}\,\mathsf{x} = \mathsf{x} + 2\end{aligned}$$

[1] In computer science unlike in mathematics we start counting with 0, which enables us to answer questions like *How many elephants are in your fridge?*

[2] You may think I have succumbed to set theory already by using $\{\dots\}$. But this is not the case, this is just a convenient notation for labelled coproducts which we will introduce soon.

To define add2 we introduce a parameter x. Now we can apply add2 to a number as in 3 and observe add2 3 which is (no surprise) 5. Lets do this step by step:

$$\begin{aligned} \mathsf{add2}\ 3 &= 3 + 2 && \text{Use the definition of } \mathsf{add2} \\ &= 5 && \text{Calculating the sum.} \end{aligned}$$

In the first step we replace the parameter x with the actual argument 2 and then we use our knowledge about addition to conclude. The first step, replacing the parameter with the argument, is called β-reduction.

The definition above combines naming and defining the function. If we just want to write a function without giving it a name we can use λ-abstraction. That is we could have defined the same function as follows:

$$\begin{aligned} &\mathsf{add2} : \mathbb{N} \to \mathbb{N} \\ &\mathsf{add2} = \lambda\, \mathsf{x} \to \mathsf{x} + 2 \end{aligned}$$

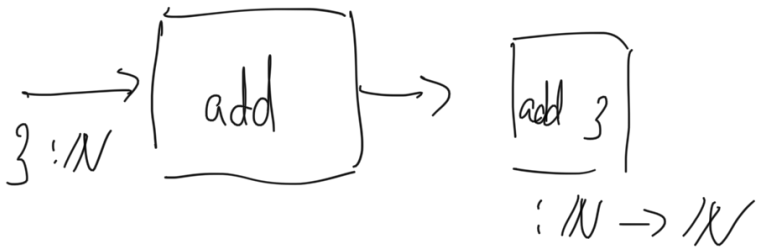
We can view the first definition as an abbreviation for the 2nd. We can now give a more detailed derivation where we differentiate the unfolding of a definition and β-reduction.

$$\begin{aligned} \mathsf{add2}\ 3 &= (\lambda\, \mathsf{x} \to \mathsf{x} + 2)\ 3 && \text{Unfolding the definition of } \mathsf{add2}. \\ &= 3 + 2 && \beta\text{-reduction} \\ &= 5 && \text{Calculating the sum.} \end{aligned}$$

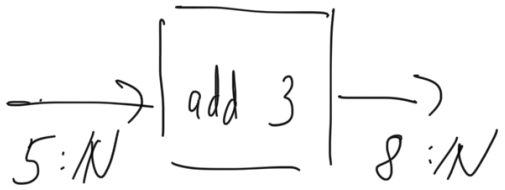
Using λ we can only define a functions with one parameter. To define a function with several parameters we use *currying*, that is a function that returns a function

$$\begin{aligned} &\mathsf{add} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \\ &\mathsf{add} = \lambda\, \mathsf{x} \to (\lambda\, \mathsf{y} \to \mathsf{x} + \mathsf{y}) \end{aligned}$$

So $\mathsf{add}\ 3 : \mathbb{N} \to \mathbb{N}$ is the function $\lambda\, \mathsf{y} \to 3 + \mathsf{y}$ that is the function that adds 3, and $(\mathsf{add}\ 3)\ 5$ reduces to $5 + 3$ and hence 8. We can illustrate this – add outputs another box add 3:



which we can then feed with the 2nd argument to get the final output:



There are some syntactic conventions to simplify the use of currying: $\to$ is *right associative*, hence we can write $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ for $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ and application is *left associative* hence we can write $\mathsf{g}\ 3\ 5$ for $(\mathsf{g}\ 3)\ 5$. Moreover we can combine several λ-abstractions and write $\lambda\ \mathsf{x}\ \mathsf{y} \to \mathsf{x} + \mathsf{y}$ for $\lambda\ \mathsf{x} \to (\lambda\ \mathsf{y} \to \mathsf{x} + \mathsf{y})$. Consequently the same convention also applies for the explicit function definition:

```
add : ℕ → ℕ → ℕ
add x y = x + y
```

Instead of returning a function we can also have functions that have functions as input (these are called *higher order functions*). An example is:

```
k : (ℕ → ℕ) → ℕ
k h = h 2 + h 3
```

Note that in this case we cannot omit the brackets in the function type. k is a function that gets a function on the natural numbers as input and returns a number. What is k add2? We can reason as follows:

$$\begin{aligned} \mathsf{k\ add2} &= \mathsf{add2\ 2} + \mathsf{add2\ 3} \\ &= (2 + 2) + (3 + 2) \\ &= 9 \end{aligned}$$

2.2 Identity and composition

Some functions work for any type, we call them *polymorphic*. An example is the identity function $\mathsf{id} : \mathsf{A} \to \mathsf{A}$ which works for every type $\mathsf{A} : \mathsf{Set}$. In Agda we can write :

```
id : {A : Set} → A → A
id x = x
```

Writing $\{\mathsf{A} : \mathsf{Set}\} \to \ldots$ indicates that the function works for every Set. Agda will automatically instantiate A that is we can just write $\mathsf{id}\ 3 : \mathbb{N}$ and Agda infers that $\mathsf{A} = \mathbb{N}$ in this case. In some cases it may be necessary to instantiate

the type variable explicitly in this case we can write id $\{\mathbb{N}\}$ or in case there are many parameters but we only want to instantiate a specific one we write id $\{A = \mathbb{N}\}$.

We will later explain types like $\{A : Set\} \to ...$ which are actually instances of dependent types. In the moment we view polymorphism as a metamathematical notion, and we will use A B C : Set to indicate variable types. In Agda this can be achieved by declaring:

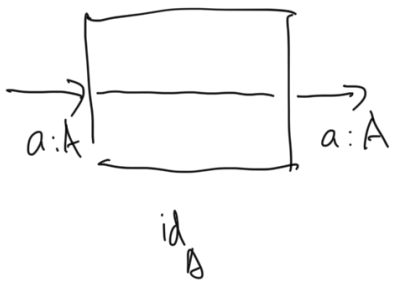
```
variable
  A B C : Set
```

which means that I can write

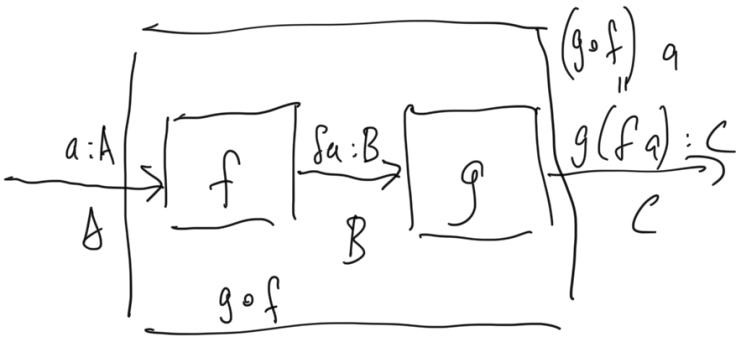
```
id : A → A
id x = x
```

and Agda will automatically translate the type into $\{A : Set\} \to A \to A$.

We can draw a picture for the identity function which just consists of a wire from the input to the output:



Another example is function composition: given $g : A \to B$ and $f : B \to C$ we can construct a new function $f \circ g : A \to C$ which feeds the output of g into the input of f:



Composition can also be defined as a polymorphic function:

```
_∘_ : (B → C) → (A → B) → (A → C)
(g ∘ f) x = g (f x)
```

In Agda we can define infix functions (and indeed more complicated syntactic schemes) by using _ to indicate where the arguments go. The price we have to pay for this flexibility is that we have to separate any syntactical component with spaces, e.g. Agda would read g∘f just as one identifier.

Let's do an example. Earlier we define the function add2 : $\mathbb{N} \to \mathbb{N}$, in the same vain we can define a squaring function:

```
square : ℕ → ℕ
square x = x * x
```

Now we can construct square ∘ add2 : $\mathbb{N} \to \mathbb{N}$, what is (square ∘ add2) 3? Let's calculate:

$$\begin{aligned}(\mathsf{square} \circ \mathsf{add2})\ 3 &= \mathsf{square}\ (\mathsf{add2}\ 3)\\ &= \mathsf{square}\ (3 + 2)\\ &= \mathsf{square}\ 5\\ &= 5 * 5\\ &= 25\end{aligned}$$

This example can also be used to show that composition is not commutative,

what is (add2 ∘ square) 3?

$$\begin{aligned}(\mathsf{add2} \circ \mathsf{square})\ 3 &= \mathsf{add2}\ (\mathsf{square}\ 3)\\ &= \mathsf{add2}\ (3 * 3)\\ &= \mathsf{add2}\ 9\\ &= 2 + 9\\ &= 11\end{aligned}$$

Hence we can see that square ∘ add2 and add2 ∘ square are different functions. In any case it isn't always possible to turn around composition. Assume as given a function IsEven : ℕ → Bool that returns true if the input is even but false otherwise (e.g. isEven 3 = false but isEven 6 = true). Now we can form isEven ∘ square : ℕ → Bool (which actually behaves the same way as isEven) but it doesn't make sense to form square ∘ isEven because the output of isEven is Bool and this doesn't match the input of square which is ℕ.

You may note that there is a strange twist in the order of arguments to _∘_ which is visible in its type (B → C) → (A → B) → (A → C). Why is it not (A → B) → (B → C) → (A → C)? And indeed when we evaluate f ∘ g we first evaluate g and then f. However, the reason is that function application is already written *the wrong way around*, that is when we evaluate square (add2 3) we first evaluate add2 3 and then square 5, which is actually counterintuitive for people who write and read from left to right. Maybe we should write function application postfix 3 add2 instead of add2 3. However, it seems to be too late to change this convention. And since (f ∘ g) x = f (g x) it seems wiser not to change the order when defining composition.

2.3 λ-calculus

I don't want to give a fully formal introduction to λ-calculus at this point. We will do this later when we have developed enough type theory to do this in Agda itself.

As we have already discussed we can view the explicit definition of a function like

```
_∘_ : (B → C) → (A → B) → (A → C)
(f ∘ g) x = f (g x)
```

is a shorthand for an explicit definition using λ-terms:

```
_∘_ : (B → C) → (A → B) → (A → C)
_∘_ = λ f → λ g → λ x → f (g x)
```

And since unfolding definitions is standard practice in Mathematics and elsewhere we concentrate on what happens to λ-terms.

When we form a λ-term we are using variables. Hence beginners often ask *What is the type of a variable?* The answer is that we assume that we are given a type of any variable, this is often called the (typing-) context.

The other cases are application and λ-abstraction:

variable A variable x has the type we have assumed for it in the context.

application Given a term $\mathsf{M} : \mathsf{A} \to \mathsf{B}$ and a term $\mathsf{N} : \mathsf{A}$ we can form a term $\mathsf{M}\,\mathsf{N} : \mathsf{B}$.

abstraction Given a variable x [3] if from assuming $\mathsf{x} : \mathsf{A}$ (here we extend the context) we can show that $\mathsf{M} : \mathsf{B}$ then we can form $\lambda\,\mathsf{x} \to \mathsf{M} : \mathsf{A} \to \mathsf{B}$. I need to add that we will only assume one type for a variable, that is by assuming $\mathsf{x} : \mathsf{A}$ we are ignoring all earlier assumptions regarding x.

This covers the *pure* λ-terms which don't use constructions specific to datatypes like $\mathbb{N}$ or Bool (but we may assume that the types of standard functions like $_+_$ or isEven are in the context.

I have already mentioned β-equality. By β-equality we mean that we can reduce an application of a λ-abstraction to an argument by substituting the variable with the argument, that is

$$(\lambda\,\mathsf{x} \to \mathsf{M})\,\mathsf{N} = \mathsf{M}[\mathsf{x}:=\mathsf{N}]$$

where $\mathsf{M}[\mathsf{x}:=\mathsf{N}]$ is M with all occurrences of x replaced by N.

Hang on, actually it is a bit more complicated. We certainly do not want to replace bound variables, that is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{x} \to \mathsf{x}))\,\mathsf{3}$ should just by $\lambda\,\mathsf{x} \to \mathsf{x}$ and not $\lambda\,\mathsf{x} \to \mathsf{3}$. We remedy this by saying that $\mathsf{M}[\mathsf{x}:=\mathsf{N}]$ is M with all *free* occurrences of x replaced by N. An occurence is free if it is not bound.

We are not done yet! Consider $\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y})$ and let's say we have a variable y lying around. What is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y}))\,\mathsf{y}$? If we mechanically replace x by y we obtain $\lambda\,\mathsf{y} \to \mathsf{y} + \mathsf{y}$. This is wrong because the y to refers to the bound variable y and not to our top-level y. This is called variable capture and it has to be avoided.

But then what is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y}))\,\mathsf{y}$? Here we introduce another equality, α-equivalence which says that bound variables can be consistently replaced, i.e. $\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y} = \lambda\,\mathsf{z} \to \mathsf{x} + \mathsf{z}$. Using this we can avoid capture by replacing the bound variable y avoiding capture, that is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y}))\,\mathsf{y} = \lambda\,\mathsf{z} \to \mathsf{y} + \mathsf{z}$. Note that any variable but y would work here.

There is yet another equation, η-equality which is motivated by the idea of extensionality, that is that two functions which are equal when applied to the same argument should be considered equal. In pure λ-calculus this is obtained by adding the following equation: assume $\mathsf{M} : \mathsf{A} \to \mathsf{B}$ and given a variable $\mathsf{x} : \mathsf{A}$ which does not appear free in M, then

$$\lambda\,\mathsf{x} \to \mathsf{M}\,\mathsf{x} = \mathsf{M}$$

[3] Note that here x is not the specific variable called x but a metavariable standing for any concrete variable.

Reading it from right to left it means that if you want to show that two functions M and N are equal it is enough to show that M x = N x where x is a variable which does not appear in M or N because

$$\begin{aligned} \mathsf{M} &= \lambda\, \mathsf{x} \to \mathsf{M}\, \mathsf{x} \\ &= \lambda\, \mathsf{x} \to \mathsf{N}\, \mathsf{x} \\ &= \mathsf{N} \end{aligned}$$

As an example we use η-equality to show that f ∘ id = f:

$$\begin{aligned} \mathsf{f} \circ \mathsf{id} &= \lambda\, \mathsf{x} \to \mathsf{f}\, (\mathsf{id}\, \mathsf{x}) \\ &= \lambda\, \mathsf{x} \to \mathsf{f}\, \mathsf{x} \\ &= \mathsf{f} \end{aligned}$$

2.4 Combinatory logic

The polymorphic functions from section 2.2, id and _∘_ are also called combinators. We are going to introduce two combinators in this section, called S and K, which are *functionally complete*. This means that every function which we can write in (pure) λ-calculus can be written just using these two combinators. This applies only to pure functions which do not refer to other datatypes (like ℕ or Bool), which would require additional combinators. We are going to show this by providing a translation from λ-terms into terms only using combinators, which means that we can eliminate all variables. In a way combinators are a form of functional machine code and they have indeed been used for compilation.

We start with K which introduces constant functions:

```
K : A → B → A
K x y = x
```

So for example K 2 : B → ℕ is the function that will always return 2. Note that this function is still polymorphic, it works for any type B.

The other combinator is S which generalizes composition, given types A B C:

```
S : (A → B → C) → (A → B) → (A → C)
S f g x = f x (g x)
```

S is similar to _∘_ but the first functions gets to see not only the result of the 2nd but also the original input. So for example: S (λ x y → x + y) (λ x → 2 * x) is the same as λ x → x + 2 * x. [4]

We can derive id (which is usually written I in combinatory logic) from S and K. The idea is that we can derive it from S by replacing the first argument f with K this obtaining S K = λ g x → K x (g x) = λ g x → x. To get the identity we can substitute g with any function of the appropriate type. The

[4] We need to instantiate A,B,C with ℕ which is done automatically by agda.

simplest choice is K, i.e. id = S K K. When implementing this in Agda we run into a bit of a technical problem, given a type A we would like to define:

```
id : A → A
id = S K K
```

Actually Agda is going to complain because it cannot infer the argument B for the 2nd K – indeed we can use any type here. The simplest choice is A and we need to tell Agda this explicitly:

```
id : A → A
id {A} = S K (K {B = A})
```

We will now show that every pure lambda term can be translated into combinatory logic. I will do this informally we will later use Agda itself to give formal proof of this theorem.

Now let's do composition as an example. We can write _∘_ as a pure λ-term

```
_∘_ = λ f g x → f (g x)
```

Actually for our purposes it is better to expand the shorthand for repeated λ-abstraction:

```
_∘_ = λ f → λ g → λ x → f (g x)
```

Our strategy is to eliminate the λs from inside out, that is first we translate λ x → f (g x) into combinators. This term will still contain the variables f and g. However, next we abstract g and then f to obtain a term using only combinators that contains no variables.

The terms which appear during the translation are terms which have no λ-abstractions but may contain variables, applications and the combinators S and K. We are going to construct λ x → M by looking at each of the possible cases:

M=x In this case clearly λ x → x = id which we have already obtained as S K K.

M=y If the variable is different from x we have a function λ x → y which is a constant function, that is K y.

M = M_1 M_2 In this case we use the assumption that we know already how to translate λ x → M_1 and λ x → M_2 Now we have

$$\begin{aligned} \lambda\, x \to M &= \lambda\, x \to M_1\, M_2 \\ &= S\, (\lambda\, x \to M_1)\, (\lambda\, x \to M_2) \end{aligned}$$

which explains the need for S.

M = K Clearly λ x → K is just a constant function returning K, hence K K.

M = **S** As above $\lambda\,\mathsf{x} \to \mathsf{S}$ is just $\mathsf{K}\,\mathsf{S}$.

Ok, let's apply the translation to our example $\lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x})$. We start with

$$\begin{aligned}
&\lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x}) \\
&= \mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{f})\,(\lambda\,\mathsf{x} \to \mathsf{g}\,\mathsf{x}) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{f})\,(\mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{g})\,(\lambda\,\mathsf{x} \to \mathsf{x})) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{f})\,(\mathsf{S}\,(\mathsf{K}\,\mathsf{g})\,\mathsf{id})
\end{aligned}$$

Before proceeding let's introduce some optimisations to avoid truly gigantic terms. So for example we can see that $\lambda\,\mathsf{x} \to \mathsf{g}\,\mathsf{x}$ is just g using η. Another optimisation is that if the variable x does not appear in M then $\lambda\,\mathsf{x} \to \mathsf{M}$ is just $\mathsf{K}\,\mathsf{M}$ and there is no need to go through M. Using these optimisations we translate $_\circ_$ in one go

$$\begin{aligned}
&\lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x}) \\
&= \lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{f})\,(\lambda\,\mathsf{x} \to \mathsf{g}\,\mathsf{x}) \\
&= \lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \mathsf{S}\,(\mathsf{K}\,\mathsf{f})\,\mathsf{g} \\
&= \lambda\,\mathsf{f} \to \mathsf{S}\,(\mathsf{K}\,\mathsf{f}) \\
&= \mathsf{S}\,(\lambda\,\mathsf{f} \to \mathsf{S})\,(\lambda\,\mathsf{f} \to \mathsf{K}\,\mathsf{f}) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}
\end{aligned}$$

While this term is comparatively small it is still hard to understand. The point of combinator is rather to show that we can avoid all the complications involving variables and just get away with S and K instead.

2.5 Products and Sums

So far we have only encountered the function type. In this section we will get to know some more basic type formers:

products Written $\mathsf{A} \times \mathsf{B}$, and also called cartesian product. In basic Mathematics we use products to represent coordinate systems, like

$$(1\ ,1) : \mathbb{R} \times \mathbb{R}.$$

sums In Agda we write $A \uplus B$, I prefer $A + B$ but this clashes with $+$ on numbers. They are also called disjoint unions and coproducts.

Actually lets start with sums which are less common in conventional Mathematics but more common in Type Theory and functional programming.

2.5.1 Sums

Sums are necessary to represent alternatives. We have one thing or another. To define sums we use the **data** constructor:

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

This says that an element of A ⊎ B is either inj₁ a where a : A or it is inj₂ b where b : B.

A simple example is that if in a form you can either fill in your order number (OrdNum) or your customer reference (CustRef) we can represent this as

$$\mathsf{OrdNum} \uplus \mathsf{CustRef}.$$

Note that even if we use the same number, lets say 1704 as an order number a customer reference we can always sure which is meant because we will use $\mathsf{inj}_1\ 1704$ if it is an order number and $\mathsf{inj}_2\ 1704$ if it is a customer reference.

The symbol for sums in Agda is a bit strange: it is a combination of the symbol for set theoretic union $\cup$ and $+$, this is related to the name for sums in set theory *disjoint union*. I prefer to just use $+$ but this clashes with the use for numbers.

How are $\cup$ and $\uplus$ related? Indeed, there is no operation $\cup$ on types. The reason is that $\cup$ in an *intensional* operator whose meaning depends on the choice of representation. For example let $A = \{\mathsf{a}, \mathsf{b}\}$ and $B = \{0, 1, 2\}$ then

$$\begin{aligned} A \cup B &= \{\mathsf{a}, \mathsf{b}, 0, 1, 2\} \\ A \uplus B &= \{\mathsf{inj}_1\ \mathsf{a}, \mathsf{inj}_1\ \mathsf{b}, \mathsf{inj}_2\ 0, \mathsf{inj}_2\ 1, \mathsf{inj}_2\ 2\} \end{aligned}$$

However, we can view A just as the one possible representation of a type with 2 elements. Another choice would be $A' = \{0, 1\}$. What happens now with $\cup$ and $\uplus$?

$$\begin{aligned} A' \cup B &= \{0, 1, 2\} \\ A' \uplus B &= \{\mathsf{inj}_1\ 0, \mathsf{inj}_1\ 1, \mathsf{inj}_2\ 0, \mathsf{inj}_2\ 1, \mathsf{inj}_2\ 2\} \end{aligned}$$

$A' \cup B$ now has only 3 elements, while $A' \uplus B$ again has 5 elements, hence is just another representation of a type with 5 elements. Thus $\uplus$ is insensitive to the representation of a type, this is a property which holds for all operations definable in Type Theory (and hence in Agda).

What can we do with $\uplus$? A useful general combinator is case that performs case analysis. we can define it using *pattern matching*:

```
case : (A → C) → (B → C) → A ⊎ B → C
case f g (inj₁ a) = f a
case f g (inj₂ b) = g b
```

case chooses between two functions, f knows how to handle A and g know how to handle B. Going back to our example with the order form (OrdNum ⊎ CustRef): if we have a function to lookup an order using the order number

```
lookupOrdNum : OrdNum → Order
```

and a function

```
lookupCustRef : CustRef → Order
```

we can use case to combine them

```
case lookupOrdNum lookupCustRef : OrdNum ⊎ CustRef → Order
```

2.5.2 Products

We define products as a special case of a *record type*:

```
record _×_ (A B : Set) : Set where
  field
    proj₁ : A
    proj₂ : B
```

that is we specify the projections, which are functions with the types:

```
proj₁ : A × B → A
proj₂ : A × B → B
```

We can construct elements of A × B by specifying what the result is when applying projections. This is called *copatternmatching*. For example we can use this to define the pairing constructor which in Agda is just the _,_, that is one doesn't have to put brackets around the tuple.

```
_,_ : A → B → A × B
proj₁ (a , b) = a
proj₂ (a , b) = b
```

We can derive the constructor automatically by adding the constructor keyword to the record definition:

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
```

What can we do with products? We derive a function curry that turns a function from products into a curried function:

```
curry : (A × B → C) → (A → B → C)
curry f = λ a b → f (a , b)
```

We can also do the reverse and translate a curried function back into one on products:

$$\begin{aligned}&\mathsf{uncurry} : (\mathsf{A} \to \mathsf{B} \to \mathsf{C}) \to (\mathsf{A} \times \mathsf{B} \to \mathsf{C})\\&\mathsf{uncurry\ g} = \lambda\,\mathsf{x} \to \mathsf{g}\,(\mathsf{proj}_1\,\mathsf{x})\,(\mathsf{proj}_2\,\mathsf{x})\end{aligned}$$

Indeed, the two functions are inverse to each other, that is $\mathsf{curry}\ (\mathsf{uncurry\ g}) = \mathsf{g}$ and $\mathsf{uncurry}\ (\mathsf{curry\ f}) = \mathsf{f}$. The first equation follows from what we already know:

$$\begin{aligned}&\mathsf{curry}\ (\mathsf{uncurry\ g})\\&= \lambda\,\mathsf{a\ b} \to \mathsf{uncurry\ g}\ (\mathsf{a}\ ,\ \mathsf{b}) && \text{Unfolding curry}\\&= \lambda\,\mathsf{a\ b} \to \mathsf{g}\ (\mathsf{proj}_1\ (\mathsf{a}\ ,\ \mathsf{b}))\ (\mathsf{proj}_2\ (\mathsf{a}\ ,\ \mathsf{b})) && \text{Unfolding uncurry}\\&= \lambda\,\mathsf{a\ b} \to \mathsf{g\ a\ b} && \text{Evaluating projections}\\&= \mathsf{g} && \text{Use } \eta \text{ twice.}\end{aligned}$$

For the 2nd equation we need an η=equality for products which says that $(\mathsf{proj}_1\,\mathsf{x}\ ,\ \mathsf{proj}_2\,\mathsf{x}) = \mathsf{x}$. This is also called *surjective pairing*:

$$\begin{aligned}&\mathsf{uncurry}\ (\mathsf{curry\ f})\\&= \lambda\,\mathsf{x} \to \mathsf{curry\ f}\ (\mathsf{proj}_1\,\mathsf{x})\ (\mathsf{proj}_2\,\mathsf{x}) && \text{Unfolding uncurry}\\&= \lambda\,\mathsf{x} \to \mathsf{f}\ (\mathsf{proj}_1\,\mathsf{x}\ ,\ \mathsf{proj}_2\,\mathsf{x}) && \text{Unfolding curry}\\&= \lambda\,\mathsf{x} \to \mathsf{f\ x} && \text{Surjective pairing}\\&= \mathsf{f} && \eta \text{ for functions.}\end{aligned}$$

This shows that the type of functions over products and curried functions are equivalent. Most functional programmers prefer to work with curried functions.

2.5.3 Strong sums?

We can try to play a similar game for coproducts. First of all using a product we can uncurry case:

$$\begin{aligned}&\mathsf{case\text{-}c} : (\mathsf{A} \to \mathsf{C}) \times (\mathsf{B} \to \mathsf{C}) \to \mathsf{A} \uplus \mathsf{B} \to \mathsf{C}\\&\mathsf{case\text{-}c} = \mathsf{uncurry\ case}\end{aligned}$$

and hence the inverse of case is:

$$\begin{aligned}&\mathsf{uncase} : (\mathsf{A} \uplus \mathsf{B} \to \mathsf{C}) \to (\mathsf{A} \to \mathsf{C}) \times (\mathsf{B} \to \mathsf{C})\\&\mathsf{uncase\ h} = (\lambda\,\mathsf{a} \to \mathsf{h}\ (\mathsf{inj}_1\ \mathsf{a}))\ ,\ (\lambda\,\mathsf{b} \to \mathsf{h}\ (\mathsf{inj}_2\ \mathsf{b}))\end{aligned}$$

we would hope that the functions are inverse to each other, that is that uncase (case-c f) = f and case-c (uncase g) = g. The first one is we can show:

uncase (case-c f)	
$= (\lambda\ a\ \rightarrow$ case-c f (inj$_1$ a)) , (λ b $\rightarrow$ case-c f (inj$_2$ b))	Unfold uncase
$= (\lambda\ a\ \rightarrow$ proj$_1$ f a , λ b $\rightarrow$ proj$_2$ f b)	Unfold case-c and case.
= (proj$_1$ f , proj$_2$ f)	η twice.
= f	Surjective pairing

However, the 2nd equality doesn't hold in Agda. Actually we can prove it using propositional equality _≡_ we are going to introduce later. The equality _=_ we have been considering here when talking *about* [5] λ-terms is called definitional equality.

Indeed the η-laws for functions and surjective pairing is already extending the notion of *definitional* equality, they are mainly introduced for convenience. Indeed, the corresponding laws for other types, eg. the natural numbers, would destroy decidability and hence not be very useful for a language like Agda. It is possible, even though not entirely straightforward, to add η-laws for _+_ but this is usually avoided, mainly because it would quickly lead to an exponential blow up of cases and hence be computationally too costly.

2.5.4 Finite Types

The operation _×_ we have introduced handles the binary case of products. We can use it several times to handle more components, as in A × B × C. In practice it is better to use records which have more meaningful field names than proj$_1$ etc.

However, one interesting case not covered by binary products is the nullary product or unit type. In Agda this is denoted as ⊤ and can be defined as a record with no fields:

```
record ⊤ : Set where
  constructor tt
```

It has one element tt : ⊤.

We can play the same game with sums and derive the empty sum, that is a sum with no injections:

```
data ⊥ : Set where
```

The empty sum is the empty type, it is a type with no elements. A version of case for the empty type is quite useful:

```
case⊥ : {A : Set} → ⊥ → A
case⊥ ()
```

[5] Within Type Theory we cannot talk about definitional equality _=_ is is part of the language itself.

We don't need any cases for $\bot$ – Agda marks this by () which basically means this case has been intentionally left empty. We observe that we get a function from the empty type into any type. This is ok, because we will never be able to run this function since there isn't an element in the empty type as the name suggests.

We can use the unit type $\top$ and sums $\uplus$ to construct some finite types. For example Two : Set is a type with 2 elements:

```
Two : Set
Two = ⊤ ⊎ ⊤
```

Here are the elements:

```
zero₂ : Two
zero₂ = inj₁ tt
one₂ : Two
one₂ = inj₂ tt
```

We can add one more element to Two to construct Three a type with 3 elements:

```
Three : Set
Three = ⊤ ⊎ Two
zero₃ : Three
zero₃ = inj₁ tt
one₃ : Three
one₃ = inj₂ zero₂
two₃ : Three
two₃ = inj₂ one₂
```

However, as already in the case of products in practice it is usually preferable to use **data** to define labelled sums: [6]

```
data Two : Set where
  zero₂ one₂ : Two
data Three : Set where
  zero₃ one₃ two₃ : Three
```

How many elements are in Two ⊎ Three? An example is $\mathsf{inj}_1\ \mathsf{one}_2$. Ok there is the choice between inj_1 and inj_2, if we choose inj_1 we then have a choice between 2 elements and if we choose inj_2 we have a choice of 3 elements, and hence we can choose out of $2 + 3 = 5$ elements.

$\mathsf{inj}_1\ \mathsf{zero}_2$	$\mathsf{inj}_1\ \mathsf{one}_2$	$\mathsf{inj}_2\ \mathsf{zero}_3$	$\mathsf{inj}_2\ \mathsf{one}_3$	$\mathsf{inj}_2\ \mathsf{two}_3$

In general if A has n elements and B has m elements then A ⊎ B has m + n elements.

[6] Note that we can combine constructors with the same type in one line.

How many elements are in Two × Three? A typical example is one_2 , two_3. Here we construct a pair and we have 2 independent choices, for the first components we have 2 options and for the second we have 3 and since these choices are independent we have 2 × 3 = 6 elements.

zero_2 , zero_3	zero_2 , one_3	zero_2 , two_3
one_2 , zero_3	one_2 , one_3	one_2 , two_3

In general if A has n elements and B has m elements then A × B are m × n elements.

What about the function type? How many elements are in Two → Three? Here is a typical example:

```
foo : Two → Three
foo zero₂ = two₃
foo one₂  = one₃
```

We can define any such function by pattern matching: we have 3 choices in each case hence $3 \times 3 = 3^2 = 9$ elements. Now unlike products and sums function types are not symmetric. How many elements are in Three → Two? Here again a typical example:

```
bar : Three → Two
bar zero₃ = zero₂
bar one₃  = one₂
bar two₃  = zero₂
```

There are 3 cases because there are 3 elements in three and in each case we have a choice of 2, hence we have $2 \times 2 \times 2 = 2^3 = 8$ elements. From these examples we can see that in general if A has n elements and B has m elements then A → B has m^n elements.

Indeed, in Mathematics the function type is often written as an exponential, that is A → B is written as B^A.

Previously, we have shown that types A × B → C and A → B → C are equivalent (via curry and uncurry). Writing this in exponential notation we find an old friend from high school:

$$C^{A \times B} = (C^B)^A$$

Similarly, with strong coproducts we would have that (A → C) × (B → C) and A ⊎ B → C are equivalent (via case-c and uncase). If we write + for ⊎ we arrive at another well known equation:

$$C^A \times C^B = C^{A+B}$$

2.6 History

λ-calculus was invented by Alonzo Church who used it in papers about formal logic in the 1930ies, e.g. [Chu32]. The story goes that the use of λ arose by

accident: Church used a graphical notation where he connected the binding and the usage of a variable. I imagine he wrote something like this:



But the typesetter (this was not only before latex but before computers) went back and said, *Sorry Professor, I can't do that. But this looks a bit like a* λ*. Is it ok if I use just this?*



I don't know wether this is actually true but it is a nice story. Church realised that one of the problem of his calculus was that he could define two many weird functions, including the so called *fixpoint combinator* which allows to computer the fixpoint of any function by general recursion. But what is the fix point of negation? To avoid this sort of problems Church also invented the typed λ-calculus. [7] Church had two primitive types, ι (the greek letter *iota)* standing for individuals and o (the greek letter *omikron* but it is the same as an o) standing for propositions, motivated by his intended application to formal logic. In my presentation I am not considering any fixed set of base types but a view them as type variables.

The untyped λ-calculus is still useful as a universal programming language which is an alternative to Turing machines. Indeed, when Alan Turing visited Church they realised that Turing machines and λ-calculus compute the same set of computable functions. The generalisation of this observation to all computational mechanism is known as the Church-Turing thesis.

In many papers about typed λ-calculus one starts with the untyped λ-calculus and then introduces a type system on top, as a way of assigning types to untyped terms. I do not follow this view which I think is too syntactic and consider typed terms as primitive. We call this *intrinsic typing*. Yes, it is true if we have to implement λ-terms we will at some point think about untyped terms but we also need to think about strings, characters bits etc and will not talk about these implementation details.

Combinatory logic was actually invented before λ-calculus by Moses Schön-

[7] A reference here is a a paper from 1940: [Chu40]

finkel in the 1920ies [8]. Again his interest was in formal logic and he realised that combinators are a nice way to avoid the complexities of variables. This was also the motivation of Haskell Curry who reinvented combinatory logic more or less in the form we know it now. [9] Some people say that the idea of *currying* a function with several arguments which is named after Curry, should have been called *schönfinkeln* but this never caught on - I wonder why? A good overview over the history of λ-calculus and combinatory logic is [CH06].

To understand my presentation of products and sums better (also called coproducts) one needs to know about category theory. Category theory is basically an abstract form of algebra which was introduced by Saunders MacLane and others. [10] Their motivation came mainly from pure Mathematics, algebraic topology, I believe. However, later it was realised that category theory is also extremely useful for theoretical computer science. One important observation was Joachim Lambek's observation that a fundamental notion in category theory, *cartesian closed categories*, corresponds exactly to typed λ-calculus. In the categorical account products are already built-in and function types are characterised by the curry-uncurry-equivalence. One of the beautiful features of category theory is the presence of a mirror (*duality*) and it turns out that the mirror image (the dual) of products are sums or coproducts as we have introduced which are also extremely useful.

At this point you may wonder what the mirror image of functions is? This is a longer story but the short answer is that there isn't one. If you assume that there is one, the theory collapses: all λ-terms are equal.

λ-calculus is important for programming, to be precise in *functional programming*. Indeed the 2nd oldest programming language, LISP (The oldest is FORTRAN) was developed by John McCarthy in the late 1950ies and was inspired by the (untyped) λ-calculus [McC60]. Later types were introduced, famously by Robin Milner in the form of ML which stands for *Metalanguage* as a component of the Edinburgh LCF project (*Logic for computable functions*) [GMM+78]. Nowadays languages like Haskell (named after Haskell curry) [11] and CAML are modern instances of typed functional programming and Agda's syntax and mechanisms borrow a lot especially from Haskell.

2.7 Exercises

1. Given

$$\begin{aligned}&\mathsf{add3} : \mathbb{N} \to \mathbb{N}\\ &\mathsf{add3}\,\mathsf{x} \;=\; \mathsf{x} + 3\end{aligned}$$

   And for any type A:

[8] If you can read german, you can check out [Sch24].
[9] An early reference is See [Cur30]
[10] MacLanes book *Categories for the working Mathematician* is still worth a look. [ML78]
[11] A good introduction to Haskell is [Hut16]

```
tw : (A → A) → A → A
tw f n = f (f n)
```

Consider tw tw add3 1.

(a) What is its type? Justify your answer.

(b) What is its value? Show every step of the evaluation.

2. Derive a term in combinatory logic (that is only using S and K) for tw using

```
tw : (A → A) → A → A
tw = λ f → λ n → f (f n)
```

3. In section 2.2 we defined identity and composition:

```
id : A → A
id x = x
_∘_ : (B → C) → (A → B) → (A → C)
(f ∘ g) x = f (g x)
```

Using β and η show that the following equations [12] hold definitionally:

$$\mathsf{f} \circ \mathsf{id} = \mathsf{f}$$
$$\mathsf{id} \circ \mathsf{f} = \mathsf{f}$$
$$\mathsf{f} \circ (\mathsf{g} \circ \mathsf{h}) = (\mathsf{f} \circ \mathsf{g}) \circ \mathsf{h}$$

4. Define an operation with the following type:

```
A× : (B → C) → A × B → A × C
```

and show that the following equations hold using β, η and surjective pairing:

$$\mathsf{A}\times\ \mathsf{id} = \mathsf{id}$$
$$\mathsf{A}\times\ (\mathsf{f} \circ \mathsf{g}) = (\mathsf{A}\times\ \mathsf{f}) \circ (\mathsf{A}\times\ \mathsf{g})$$

This shows that the operation A× _ on types is (definitionally) a functor.

5. Find elements of the following types using only pure λ-terms and the combinators for sums and products (proj_1 , proj_2 , _,_, inj_1 , inj_2, case).

```
x₀ : (A → B) → (B → C) → (A → C)
x₁ : (A → B) → ((A → C) → C) → ((B → C) → C)
x₂ : A × (B ⊎ C) → (A × B) ⊎ (A × C)
x₃ : (A × B) ⊎ (A × C) → A × (B ⊎ C)
```

[12] These are the equations of a category. We are establishing that types and functions from a category, definitionally.

Chapter 3

Propositional logic

When we study logic we usually start with propositional logic, which introduces the basic logical connectives:

conjunction The logical operation **and** which is usually denoted as _∧_ or as _&_ as an operation on Bool.

disjunction The logical operation **or** which is denoted as _∨_ or as _|_ for Bool.

negation The logical operation **not** which is denoted as ¬ or as ! for Bool.

implication The logical operation **if** ... **then** ... which is written as _⇒_ but has not standard notation as an operation on booleans.

logical equivalence We often say **if, and only if** which is abbreviated as **iff**, this is written as ⇔. For Bool this just corresponds to equality on booleans which we will write _==_.

Propositional logic is usually the first step which is later followed by predicate logic which introduces the quantifiers **for all** ∀ and **exists** ∃. While propositional logic is simpler it is somehow abstract because we cannot say anything interesting because we cannot form interesting propositions yet. Hence we use logical variables to form propositions and study logical tautologies, that is propositions that are true for every assignment of truth values.

Before we explain the type theoretic approach using propositions as types, we review the classical semantics using Bool.

3.1 Boolean logic

What is a truth value or a proposition? The classical approach is to equate propositions with Bool which is a type with exactly two values true and false. We could use the operations on types we have introduced in the last section to

define Bool as $\top \uplus \top$ but we will just use the mechanism of datatypes and define:

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

Now to define _&_ we use pattern matching:

```
_&_ : Bool → Bool → Bool
true & y = y
false & y = false
```

The idea is that if the first parameter is true then the result of true & y is just y, while if the first parameter is false then false & y == false for all y. We could have also used the 2nd parameter since the operation is commutative. Actually this is a tautology: x & y == y & x that is this expression is true for all possible values of x y : Bool. We can show this by drawing a truth table:

x	y	x & y	y & x	x & y == y & x
false	false	false	false	true
false	true	false	false	true
true	false	false	false	true
true	true	true	true	true

Actually I haven't yet defined the function _==_ but I leave this as an exercise.

Next we define the operation boolean or:

```
_|_ : Bool → Bool → Bool
true | y = true
false | y = y
```

The idea is similar as for &: if the first parameter is true we already know that true | y ≡ true but of the first parameter is false then false | y ≡ y. We leave the verification that _|_ is commutative as an exercise.

Already with _&_ and _|_ we can explore a number of laws, which are the laws of *boolean algebra*. As in arithmetic we have the law of distributivity x* (y+z) ≡ x*y + x*z which now becomes x & (y | z) ≡ x & y | x & z. When writing the law we already exploited the convention that _&_ binds stronger that _|_.

As above we can verify the law by drawing the truth table:

x	y	z	x & (y | z)	x & y | x & z	x & (y | z) == x & y | x & z
false	false	false	false	false	true
false	false	true	false	false	true
false	true	false	false	false	true
false	true	true	false	false	true
true	false	false	false	false	true
true	false	true	true	true	true
true	true	false	true	true	true
true	true	true	true	true	true

Actually we don't have to look at the whole truth table but we can show directly that x & (y | z) and x & y | x & z are equal. We only look at x we have two case x = true and x = false. In the case x = true we can reason for the left hand side:

$$\begin{aligned} \mathsf{x}\ \&\ (\mathsf{y} \mid \mathsf{z}) &= \mathsf{true}\ \&\ (\mathsf{y} \mid \mathsf{z}) && \text{using the definition of _\&_} \\ &= \mathsf{y} \mid \mathsf{z} \end{aligned}$$

and for the right hand side:

$$\begin{aligned} (\mathsf{x} \mid \mathsf{y})\ \&\ (\mathsf{x} \mid \mathsf{z}) &= (\mathsf{true}\ \&\ \mathsf{y}) \mid (\mathsf{true}\ \&\ \mathsf{z}) && \text{using the definition of _\&_ twice} \\ &= \mathsf{y}\ \&\ \mathsf{z} \end{aligned}$$

Hence both sides are identical. In the case x = false we can also show that both sides are identical:

$$\begin{aligned} \mathsf{x}\ \&\ (\mathsf{y} \mid \mathsf{z}) &= \mathsf{false}\ \&\ (\mathsf{y} \mid \mathsf{z}) && \text{using the definition of _\&_} \\ &= \mathsf{false} \end{aligned}$$

and for the right hand side:

$$\begin{aligned} (\mathsf{x} \mid \mathsf{y})\ \&\ (\mathsf{x} \mid \mathsf{z}) &= (\mathsf{false}\ \&\ \mathsf{y}) \mid (\mathsf{false}\ \&\ \mathsf{z}) && \text{using the definition of \& twice} \\ &= \mathsf{false} \mid \mathsf{false} && \text{using the definition of \mid} \\ &= \mathsf{false} \end{aligned}$$

Other important equations are the de Morgan laws, which are basically the observation that the truth tables of _&_ and _|_ can be obtained from each other by switching true and false. To express this we need negation:

```
!_ : Bool → Bool
! true  = false
! false = true
```

The de Morgan laws now are:

x	y	! (x | y)	! x & ! y	! (x | y) == ! x & ! y
false	false	true	true	true
false	true	true	true	true
true	false	true	true	true
true	true	false	false	true

x	y	! (x & y)	! x | ! y	! (x & y) == ! x | ! y
false	false	true	true	true
false	true	false	false	true
true	false	false	false	true
true	true	false	false	true

Let us verify the 2nd one again by equational reasoning, again we only need to consider x.

x=true

$$\begin{aligned} \text{! (x \& y)} &= \text{! (true \& y)} \\ &= \text{! y} \end{aligned}$$

$$\begin{aligned} \text{! x | ! y} &= \text{! true | ! y} \\ &= \text{false | ! y} \\ &= \text{!y} \end{aligned}$$

x=false

$$\begin{aligned} \text{! (x \& y)} &= \text{! (false \& y)} \\ &= \text{! false} \\ &= \text{true} \end{aligned}$$

$$\begin{aligned} \text{! x | ! y} &= \text{! false | ! y} \\ &= \text{true | ! y} \\ &= \text{true} \end{aligned}$$

We will soon see how paper proofs like this can be done in Agda itself (see section 6.7 exercise 3).

3.2 Propositions as types

While the boolean semantics is quite standard it has some serious shortcomings. Can we really identify truth values with Bool? There are many propositions of which we don't know wether they are true or false hence we cannot assign a boolean value to them. The problem becomes obvious once we move on to predicate logic: a predicate over the natural number would be just a function $\mathsf{P} : \mathbb{N} \to \mathsf{Bool}$ but what is the truth value of the statement that P is true for all natural numbers $\forall\, \mathsf{x} : \mathbb{N} \to \mathsf{P\ x} : \mathsf{Bool}$? We would have to check P for infinitely many inputs which is clearly not possible.

In Type Theory we use another definition of truth values, instead of saying that a proposition is either true or false, we say that a proposition is something

we can have evidence for. A proposition holds if we can find evidence. It isn't always clear wether we can find evidence for a proposition hence this is different from the boolean explanation. Indeed this interpretation is associated with *intuitionistic logic*, which has different laws than boolean algebra (it is called Heyting algebra).

To be precise we identify propositions with Set that is we assign to any proposition the type of evidence for it. [1]

```
prop = Set
```

We will use P Q R to stand for arbitrary propositions (i.e. elements of prop). What is evidence for P **and** Q that is P ∧ Q? Clearly it is a pair p , q of evidence p : P and evidence q : Q. That is:

```
_∧_ : prop → prop → prop
P ∧ Q = P × Q
```

Similarly, evidence for P **or** Q is the sum of evidence for P and Q:

```
_∨_ : prop → prop → prop
P ∨ Q = P ⊎ Q
```

Implication which doesn't play a central role in boolean logic is central in intuitionistic logic. What is evidence for P ⇒ Q? It is a function that transforms evidence for P to evidence for Q. Hence:

```
_⇒_ : prop → prop → prop
P ⇒ Q = P → Q
```

We also need to define the logical constants: True has a trivial proof, while False is the empty type, there is no evidence for it.

```
True : prop
True = ⊤

False : prop
False = ⊥
```

Having all these we can define negation by saying that ¬ P holds if P entails impossible:

```
¬ : prop → prop
¬ P = P ⇒ False
```

We also define logical equivalence as implications going in both direction, which is somehow obvious from the symbol used:

[1] Don't confuse this with Agda's proof-irrelevant Prop, which identifies all elements of a proposition. We will discuss proof-irrelevance later.

```
_⇔_ : prop → prop → prop
P ⇔ Q = (P ⇒ Q) ∧ (Q ⇒ P)
```

In the boolean setting we were able to use equality _≡_ for logical equivalence. This is not possible in the propositions as types explanation presented here. [2]

We use the usual assumptions how to read logical formulas:

- _∧_ binds stronger than _∨_.
- _∨_ binds stronger than _⇒_.
- _⇒_ binds stronger than _⇔_.
- _⇒_ is right associative (like _→_).

The following table summarizes our definitions of logical connectives:

latin	*english*	*logic*	*types / definitions*
Conjunction	And	P ∧ Q	P × Q
Disjunction	Or	P ∨ Q	P ⊎ Q
Implication	if-then	P ⇒ Q	P → Q
Verum	true	True	⊤
Falsum	false	False	⊥
Negation	not	¬ P	P ⇒ False
Equivalence	if-and-only-if	P ⇔ Q	(P ⇒ Q) ∧ (Q ⇒ P)

Let's revisit the tautologies we have shown using the boolean interpretation earlier. Certainly _∧_ is commutative, because _×_ is. We only need to show one direction because the other one follows by symmetry. We can show it easily by copatternmatching:

```
∧-comm→ : P ∧ Q ⇒ Q ∧ P
proj₁ (∧-comm→ x) = proj₂ x
proj₂ (∧-comm→ x) = proj₁ x
∧-comm : P ∧ Q ⇔ Q ∧ P
proj₁ ∧-comm = ∧-comm→
proj₂ ∧-comm = ∧-comm→
```

Since we have declared a constructor for _×_, namely _,_ we could have proven ∧-comm→ also by pattern matching:

```
∧-comm→ : P ∧ Q ⇒ Q ∧ P
∧-comm→ (p , q) = q , p
```

which is sometimes more convenient as the next example shows.

Let's prove that _∧_ distributes over _∨_, that is:

[2] However, we will recover this identification when we introduce Homotopy Type Theory later, which also uses a small modification of the translation presented here.

$$\wedge\vee\text{-distr} : P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$

Let's use the Agda system to construct the proof / program interactively. We start with

$$\wedge\vee\text{-distr} = ?$$

Our first step is to use copattern matching to split $_\Leftrightarrow_$ which is defined as a conjunction in its two parts:

$$\begin{aligned}\text{proj}_1\ \wedge\vee\text{-distr} &= ?\\ \text{proj}_2\ \wedge\vee\text{-distr} &= ?\end{aligned}$$

We first focus on the $_\Rightarrow_$ direction. This is a function hence let us introduce an assumption / parameter:

$$\text{proj}_1\ \wedge\vee\text{-distr}\ x = ?$$

Our goal is to prove $P \wedge Q \vee P \wedge R$ and we can use $x : P \wedge (Q \vee R)$. Our next step is to analyze our assumption using pattern matching:

$$\text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ qr) = ?$$

We now have $p : P$ and $qr : Q \vee R$. We analyze qr further which means we get two cases:

$$\begin{aligned}\text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_1\ q) &= ?\\ \text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_2\ r) &= ?\end{aligned}$$

Now we are cooking with gas. In the first case we have $q : Q$ hence we can prove $P \wedge Q \vee P \wedge R$. But first of all we have to decide wether we use inj_1 or inj_2. No proces for guessung which one:

$$\text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_1\ q) = \text{inj}_1\ ?$$

Now we are left to prove $P \wedge Q$ which is no problem with the assumptions lying around:

$$\text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_1\ q) = \text{inj}_1\ (p\ ,\ q)$$

The next line works the same way:

$$\text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_2\ r) = \text{inj}_2\ (p\ ,\ r)$$

At this point I am getting bored and just complete the program:

$$\begin{aligned}\text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_1\ q) &= \text{inj}_1\ (p\ ,\ q)\\ \text{proj}_1\ \wedge\vee\text{-distr}\ (p\ ,\ \text{inj}_2\ r) &= \text{inj}_2\ (p\ ,\ r)\\ \text{proj}_2\ \wedge\vee\text{-distr}\ (\text{inj}_1\ (p\ ,\ q)) &= p\ ,\ \text{inj}_1\ q\\ \text{proj}_2\ \wedge\vee\text{-distr}\ (\text{inj}_2\ (p\ ,\ r)) &= p\ ,\ \text{inj}_2\ r\end{aligned}$$

I hope you have been getting bored too. Indeed it shouldn't be difficult to write this little program without much interaction but I wanted to demonstrate how to do it step by step because in some cases this may be necessary.

In this instance the two functions we define to show a logical equivalence are actually inverse to each other, but this is not always the case. So for example:

$$\begin{aligned}&\wedge\text{-dbl} : P \Leftrightarrow P \wedge P\\ &\text{proj}_1\ \wedge\text{-dbl}\ p = p , p\\ &\text{proj}_2\ \wedge\text{-dbl}\ (p , _) = p\end{aligned}$$

These two functions are not inverse, e.g. if we start with $2 , 3$ and go back and forth we end up with $2 , 2$.

Next we revisit the de Morgan laws, the first one is no problem:

$$\begin{aligned}&\text{dm}_1 : \neg (P \vee Q) \Leftrightarrow \neg P \wedge \neg Q\\ &\text{proj}_1\ \text{dm}_1\ f = (\lambda\, p \to f\ (\text{inj}_1\ p)) , \lambda\, p \to f\ (\text{inj}_2\ p)\\ &\text{proj}_2\ \text{dm}_1\ (g , h)\ (\text{inj}_1\ p) = g\ p\\ &\text{proj}_2\ \text{dm}_1\ (g , h)\ (\text{inj}_2\ q) = h\ q\end{aligned}$$

But for the 2nd one we run into a problem in one direction:

$$\begin{aligned}&\text{dm}_2 : \neg (P \wedge Q) \Leftrightarrow \neg P \vee \neg Q\\ &\text{proj}_1\ \text{dm}_2\ f = ?\\ &\text{proj}_2\ \text{dm}_2\ (\text{inj}_1\ f)\ (p , q) = f\ p\\ &\text{proj}_2\ \text{dm}_2\ (\text{inj}_2\ g)\ (p , q) = g\ q\end{aligned}$$

We cannot show $\neg (P \wedge Q) \Rightarrow \neg P \vee \neg Q$. When we attempt to show this we assume $f : \neg (P \wedge Q)$, and now we need to establish $\neg P \vee \neg Q$. The problem is that we have no information which case we should choose. E.g. if we choose to show $\neg P$ we assume $p : P$ and we need to derive False. The only possibility is to use f but this requires $P \wedge Q$ as the input and we only have P.

After all this should not be surprising: We may know that not both propositions can be false but we don't know which one it is. E.g. if somebody tells us that it is not true that both it rains and we go to the zoo, then we still don't know wether it doesn't rain or we don't go to the zoo or both.

So indeed while the 2nd de Morgan law holds in the boolean semantics it doesn't hold intuitionistically (it is not a law of Heyting algebra).

3.3 Classical principles

So what is the exact difference between the boolean logic which is also called *classical logic*, and the evidence based logic which is called intuitionistic logic. It seems that certain propositions, or actually propositional schemes since they involved propositional variables, which are provable classically are not provably intuitionistically. But what exactly is the difference?

It turns out that the extra power of classical logic is exactly the assumption that every proposition is either true or false. This is expressed as the following

scheme which is called the *law of the excluded middle* or in latin *tertium non datur* (the third is not given). We define

```
TND : prop → prop
TND P = P ∨ ¬ P
```

This means that we write TND P to express that excluded middle holds for P. Classical logic corresponds to assuming that TND holds for all propositions.

Let's go back to the instance of the de Morgan law which we couldn't prove. It turns out that we can prove it using TND:

```
dm₂₁tnd : TND P → ¬ (P ∧ Q) ⇒ ¬ P ∨ ¬ Q
dm₂₁tnd (inj₁ p) f = inj₂ (λ q → f (p , q))
dm₂₁tnd (inj₂ np) f = inj₁ np
```

The idea is that If P holds then we can prove ¬ Q because from assuming Q we can derive a contradiction using the premise ¬ (P ∧ Q). On the other hand if ¬ P holds we are done using just this.

We can see that using TND we can simulate truth tables: this proof corresponds to the algebraic proof for booleans we have done in section 3.1.

There is an alternative principle we can use, which we will see is equivalent to TND. This is the *principle of indirect proof*, also called *reductio ad absurdo* (reduction to the absurd) in latin. This you see in many mathematical proofs: to prove a proposition P we assume ¬ P and derive a contradiction. Since deriving a contradiction is the same as negation this just means that ¬ (¬ P) implies P:

```
RAA : prop → prop
RAA P = ¬ (¬ P) → P
```

We can show that we can derive RAA from TND using the same technqiue we have used for the de Morgan law above:

```
tnd→raa : TND P → RAA P
tnd→raa (inj₁ p) nnp = p
tnd→raa (inj₂ np) nnp = case⊥ (nnp np)
```

We assume nnp : ¬ (¬ P). Now we analyze P using tnd: if it is p : P we are done. On the other hand if we have np : ¬ P we can derive a contradiction nnp np : False and then use case⊥ to conclude.

The other direction is somehow more interesting: while we cannot prove P ∨ ¬ P in general, we can show that it cannot be false. That is we can prove ¬ (¬ (P ∨ ¬ P)) [3]

```
nntnd : ¬ (¬ (P ∨ ¬ P))
nntnd npnp = npnp (inj₂ (λ p → npnp (inj₁ p)))
```

[3] This should be called the principle of the excluded middle: it states that it is impossible that a proposition is neither true nor false.

It is worthwhile to step through this proof. Ok we assume npnp : ¬ (P ∨ ¬ P) to derive False. The only option now is just to use npnp which means we need to show P ∨ ¬ P. There is no chance that we could prove P hence lets try ¬ P. Ok we assume p : P and have to derive False again and again there is no choice but to use npnp and what dejavu we are back proving P ∨ ¬ P. But this time we actually know p : P hence a we have to do is to use this assumption.

Now we can just use RAA (P ∨ ¬ P) to derive TND P from nntnd:

```
raa→tnd : RAA (P ∨ ¬ P) → TND P
raa→tnd raa = raa nntnd
```

So if we assume that RAA P for all propositions P then we can also show TND Q for all propositions, even though it is not the same one. In this sense the two principles are equivalent.

3.4 The negative translation

We may wonder when exactly is a classical tautology provable intuitionistically, i.e. with the propositions as types interpretation, and when not. We have seen that one of the de Morgan rules ¬ (P ∨ Q) ⇔ ¬ P ∧ ¬ Q is provable intuitionistically but for the other one only one direction ¬ P ∨ ¬ Q ⇒ ¬ (P ∧ Q) is provable but the other direction ¬ (P ∧ Q) ⇒ ¬ P ∨ ¬ Q required classical reasoning.

The difference is evidence for an intuitionistic disjunction P ∨ Q contains information, namely which of the P and Q actually holds. However, in the 2nd de Morgan rule ¬ (P ∧ Q) merely states that it cannot be that both P and Q hold while ¬ P ∨ ¬ Q tells us which of the two actually doesn't hold and we cannot create this information from nothing.

We call propositions that contain information *positive* and the ones that do not *negative*. We will see that we can understand classical logic as the logic of negative propositions and that negative propositions are the one for which RAA holds.

First of all we show that any negated proposition is negative

```
neg-is-neg : RAA (¬ P)
```

which expands to showing that ¬ (¬ (¬ P)) → ¬ P which may be slightly surprising because in general we don't have ¬ (¬ P) → P in general. But it makes now sense because both sides are negative, while in the later case we can replace P by a positive proposition.

We can show this directly, but maybe it is more instructive to do this a bit more slowly. We can show that every proposition implies its double negation:

```
nn : P → ¬ (¬ P)
nn p np = np p
```

Indeed this is just simple application. We can also show the following principle:

```
neg-impl : (P → Q) → ¬ Q → ¬ P
```

It is always good to do a little intuitive check: If we know *If the sun shines we go to the zoo* then *If we don't go to the zoo the sun doesn't shine*. Here is the proof:

```
neg-impl f nq p = nq (f p)
```

Now we can show neg-is-neg by just combining the two results:

```
neg-is-neg = neg-impl nn
```

We can also show that conjunction and implication preserve negativity (for implication we only need that the conclusion is negative):

```
and-neg : RAA P → RAA Q → RAA (P ∧ Q)
impl-neg : RAA Q → RAA (P → Q)
```

I leave this as an exercise.

The only connective that is not negative is disjunction. Classically we can show that $P \vee Q$ is equivalent to the negative $\neg\,(\neg\,P \wedge \neg\,Q)$ because $P \vee Q \Leftrightarrow \neg\,(\neg\,(P \vee Q))$ and using de Morgan $\neg\,(P \vee Q) \Leftrightarrow \neg\,P \wedge \neg\,Q$. Hence we can define a classical disjunction:

```
_ ∨ᶜ _ : prop → prop → prop
P ∨ᶜ Q = ¬ (¬ P ∧ ¬ Q)
```

What can we do with $_ \vee^c _$? We can actually prove the injections easily:

```
inj₁ᶜ : P → P ∨ᶜ Q
inj₂ᶜ : Q → P ∨ᶜ Q
```

But what about $\text{case}^c : (P \to R) \to (Q \to R) \to P \vee^c Q \to R$? It turns out that we can derive it, if we assume that R is negative:

```
caseᶜ : RAA R → (P → R) → (Q → R) → P ∨ᶜ Q → R
```

Now using classical disjunction we can actually prove excluded middle:

```
tndᶜ : P ∨ᶜ ¬ P
```

Having inj_1 , inj_2, case^c and tnd^c together with all the standard principles for the remaining negative connectives we can simulate any classical proof. Any propositional variable quantifies only over negative propositions, that is when we assume P : prop we need to add raap : RAA P. So basically the negative translation provides an alternative to the boolean explanation of classical logic. However, it has a big advantage, because as we will see it also works for predicate logic where the boolean explanation fails.

Hence classical logic can be understood as the logic of negative propositions. Hence we can summarise that *a classical logician is somebody who just can't say anything positive.*

3.5 History

Boolean algebra was invented by George Boole in the 19th century who also gave his name to the type of booleans or short Bool. However, the laws of boolean algebra don't play a big role in our presentation. Truth tables were already used in the 19th century first by Peirce and they were popularized by Wittgenstein and Post in the early 20th century.

Intuitionistic logic was introduced by Brouwer in the 1920ies who also had a big fight with Hilbert about the use of the excluded middle. This was called the *Grundlagenstreit* (Argument about foundations).[4] Hilbert complained:

> Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. To prohibit existence statements and the principle of excluded middle is tantamount to relinquishing the science of mathematics altogether.

It is fair to say that Hilbert won, at least as far as mainstream Mathematics is concerned. But don't forget this was before the invention of computers.

It was Brouwer's student Heyting who made the formal rules of intuitionistic logic precise and who also came up with a constructive semantics for it, the Brouwer-Heyting-Kolmogorov interpretation which is closely related to the propositions as types explanation but it uses untyped *realizers*.

The propositions as types explanation is also called the *Curry-Howard Isomorphism*: [5] they observed an analogy between the rules of logic and the rules of typed λ-calculus. I prefer calling it the propositions as types explanation because where does logic come form in the first place? I think we should use the idea that propositions can be identified with the type of their evidence to justify the rules of logic. In this way the semantics comes first and the formal rules are just a reflection of the semantics. While this is not historically correct, I think this is a better way to look at this.

The negative translation which is also called the double negation translation goes back to Gödel and Gentzen. In the context of functional programming it is also known as the continuation passing style transformation (CPS transform) which provides a translation for control operators like call-cc (*call with current continuation*).

3.6 Exercises

1. Define the operations

```
implb : Bool → Bool → Bool
_==_ : Bool → Bool → Bool
```

[4] I recommend van Dalen's biography of Brouwer which also includes lots of details about this famous disagreement. [VD99]

[5] It was first observed by Curry in a letter in 1934 and later refined by Howard in 1969 but this was only published in 1980 [How80].

Here implb is the boolean version of implication. There are given by the following truth tables:

P	Q	implb P Q	P == Q
false	false	true	true
false	true	true	false
true	false	false	false
true	true	true	true

2. if_then_else_ is a basic combinator on Bool. It is defined as follows

```
if_then_else : Bool → A → A → A
if true then y else z = y
if false then y else z = z
```

Define all the logical operations

```
_&_ : Bool → Bool → Bool
_|_ : Bool → Bool → Bool
!_ : Bool → Bool
implb : Bool → Bool → Bool
_==_ : Bool → Bool → Bool
```

in agda using only if_then_else (i..e without using pattern matching).

3. Prove the following equivalences. It is helpful to use some auxilliary statements to be able to use pattern matching more easily.

   All propositions in this part are provable.

```
e1 : (P → Q ∧ R) ⇔ ((P → Q) ∧ (P → R))
e2 : (P ∨ Q → R) ⇔ ((P → R) ∧ (Q → R))
```

4. We are playing *logic poker* today.

   Try to prove the following propositions using Agda.

```
P1  = {P Q : prop} → (P ⇒ Q) ⇒ ¬ P ⇒ ¬ Q
P2  = {P Q : prop} → (P ⇒ Q) ⇒ ¬ Q ⇒ ¬ P
P3  = {P Q : prop} → (P ⇒ Q) ⇒ ¬ P ∨ Q
P4  = {P Q : prop} → ¬ P ∨ Q ⇒ (P ⇒ Q)
P5  = {P : prop} → ¬ (P ⇔ ¬ P)
P6  = {P : prop} → ¬ (P ∨ ¬ P)
P7  = {P : prop} → ¬ (¬ (¬ P)) ⇒ ¬ P
P8  = {P Q : prop} → ((P ⇒ Q) ⇒ P) ⇒ P
P9  = {P : prop} → (¬ (¬ P) ⇒ P) ⇒ P ∨ ¬ P
P10 = {P : prop} → P ∨ ¬ P ⇒ (¬ (¬ P) ⇒ P)
```

   Consider the following cases:

(a) You can prove it directly (intuitionistically). E.g. you have

```
Pa = {P : prop} → P → P
```

in this case you just provide a proof:

```
pa : Pa
pa x = x
```

(b) You cannot prove it with intuitionistic logic but you can prove it with classical logic.
E.g. if I ask you to prove

```
Pb = {P Q : prop} → ¬ (P ∧ Q) → ¬ P ∨ ¬ Q
```

and you realize that you cannot prove this intutionistically but you can prove it with classical logic that is using RAA P for some proposition P : prop. I define

```
CLASS = {P : prop} → RAA P
```

and you prove

```
pb : CLASS → Pb
```

When you prove pa you get an extra parameter raa which you can use in the proof:

```
pb raa h =
  raa (λ x → x (inj₁ (λ p → x (inj₂ (λ q → h (p , q))))))
```

If you need TND P you can use raa→tnd.

(c) It isn't even true in classical logic. In this case you should find a counterexample. That is you should be able to instantiate the propositional variables with ⊤ and ⊥ and prove the negation.

```
Pc = {P Q : prop} → P → Q
```

but you realize that this is false by instantiating P = ⊤ and Q = ⊥. Ok, in this case you prove

```
pc : Pc → ⊥
pc h = h {⊤} {⊥} tt
```

5. Complete the missing proofs from the negative translation (section 3.4):

```
and-neg : RAA P → RAA Q → RAA (P ∧ Q)
impl-neg : RAA Q → RAA (P → Q)
```

$$\begin{array}{l}
\mathrm{inj}_1^c \ : \ \mathsf{P} \ \to \ \mathsf{P} \ \vee^c \ \mathsf{Q} \\
\mathrm{inj}_2^c \ : \ \mathsf{Q} \ \to \ \mathsf{P} \ \vee^c \ \mathsf{Q} \\
\mathrm{case}^c \ : \ \mathsf{RAA}\,\mathsf{R} \ \to \ (\mathsf{P} \ \to \ \mathsf{R}) \ \to \ (\mathsf{Q} \ \to \ \mathsf{R}) \ \to \ \mathsf{P} \ \vee^c \ \mathsf{Q} \ \to \ \mathsf{R} \\
\mathrm{tnd}^c \ : \ \mathsf{P} \ \vee^c \ \neg \, \mathsf{P}
\end{array}$$

Hint: It is useful to prove that double negation is monotone:

$$(\mathsf{P} \ \to \ \mathsf{Q}) \ \to \ \neg \, (\neg \, \mathsf{P}) \ \to \ \neg \, (\neg \, \mathsf{Q})$$

Chapter 4

Datatypes

The types we have seen so far, that is functions (_→_), products (_×_,⊤) and sums (_+_, ⊥) are finite or preserve finiteness. In computation but also in Mathematics we often use infinite types, such as the natural numbers (ℕ), which I have already used for illustration. In this chapter we will look at such *datatypes*, starting with the natural numbers as the canonical example. We discuss primitive recursion first in a schematic way and then given by combinators: the recursor which can be reduced to the iterator (also called fold). We then look at various examples of inductive types such as lists, syntax trees and infinite trees representing ordinal notations. We investigate the limits of what constitutes a reasonable inductive type and what not — this is related to the notion of positivity. Using the notion of categorical mirror we also investigate the dual of inductive types: coinductive types. Examples here are streams and coinductive natural numbers, which can also be infinite.

4.1 The natural numbers

One of the most versatile datatypes we consider is the type of natural numbers, i.e. the counting numbers $0, 1, 2, 3, 4, \ldots$. Indeed, because we are computer scientists we are starting counting with 0, which is useful when calculating off-sets in datastructures but it is also the answer to the question *How many elephants do you have in your fridge* (I hope).

It was Guiseppe Peano from Torino who wrote down the precise laws of the natural numbers in predicate logic at the end of the 19th century, this is nowadays known as Peano Arithmetic. We aren't going to study these laws in detail but use one important idea: according to Peano a natural number is either zero or it is the successor of another natural number. We can implement this in Agda:

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

This is the definition of a sum, but the novelty is that it is recursive, that is we are using $\mathbb{N}$ which is just being defined in the argument of the constructor suc.

Given this we can derive the usual notation: [1]

$$
\begin{array}{lll}
0 = & \mathsf{zero} & \\
1 = & \mathsf{suc}\ 0 & = \mathsf{suc}\ \mathsf{zero} \\
2 = & \mathsf{suc}\ 1 & = \mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}) \\
3 = & \mathsf{suc}\ 2 & = \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) \\
\vdots & & \vdots
\end{array}
$$

Let's define a function on the natural numbers, for example the predecessor:

```
pred : ℕ → ℕ
pred zero    = zero
pred (suc n) = n
```

Here we use pattern matching to analyse the constructor as we have seen already. Actually this function is a bit of a cheat because there isn't really a predecessor of 0. A better version uses Maybe where Maybe A is just a shorthand for $\top \uplus$ A:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just :    A → Maybe A
```

Using Maybe we can now define a version of pred that signals an error on 0 by returning nothing:

```
pred : ℕ → Maybe ℕ
pred zero    = nothing
pred (suc n) = just n
```

This version of pred indicates an error on 0 by returning nothing but for any other number returns just the predecessor.

This predecessor is the exact inverse of the constructors which can be combined into one function:

```
zerosuc : Maybe ℕ → ℕ
zerosuc nothing  = zero
zerosuc (just n) = suc n
```

It is not hard to see that the functions pred and zerosuc are inverse to each other.

[1] Luckily we can tell Agda to use the standard notation for numbers simply by invoking a *pragma*:

```
{-# BUILTIN NATURAL ℕ #-}
```

4.1.1 Recursion

To define more interesting functions on the natural numbers we need to use recursion, for example let's define the doubling function:

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

That is the double of 0 is 0 and the double of a number of the form suc n is 2 + double n, which as we haven't yet defined _+_ is just suc (suc (double n).

For example double 3 = 6. Let's go through this step by step:

```
double 3 = double (suc (suc (suc zero)))
         = suc (suc (double (suc (suc zero))))
         = suc (suc (suc (suc (double (suc zero)))))
         = suc (suc (suc (suc (suc (suc (double zero))))))
         = suc (suc (suc (suc (suc (suc zero)))))
         = 6
```

We can justify the use of recursion: all natural numbers are constructed via zero and suc. To define a function f over the natural numbers we just have to give a case for zero and one for suc n. Since the number n has to be constructed *before* we suc n we can construct the answer to the function f in the same way and compute the answer to f n before we compute the answer to f (suc n) hence we can use f n when constructing f (suc n).

Not all functions are exactly of this form, already the inverse of double, that is half which forgets the remainder, uses a slightly more general scheme

```
half : ℕ → ℕ
half zero          = zero
half (suc zero)    = zero
half (suc (suc n)) = suc (half n)
```

That is half of 0 and 1 is zero and half orf 2 + n is 1 + half n. The recurrence from suc (suc n) can be justified in the same way as before because certainly n is constructed before suc (suc n).

4.1.2 Arithmetic operations

Let's define the standard functions of arithmetic starting with addition:

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

The idea of this definition of addition is that for example 3 + 5 is the 3rd successor of 5 that suc (suc (suc 5)). This means that 0 + n is just n and suc m + n is one more than m + n.

Next we define multiplication _*_ which is repeated addition in the same way as _+_ is repeated suc.

```
_*_ : ℕ → ℕ → ℕ
zero  * n = 0
suc m * n = n + m * n
```

That is for example 3 * 5 is 5 + 5 + 5 + 0. Recursively we express this by saying that 0 * n is just 0 and suc n * m is n + (m * n).

We can go further and define exponentiation as repeated multiplication. In the previous cases it didn't matter over which argument we recursed because they were commutative. But exponentiation certainly isn't, e.g. $2^3 = 8$ but $3^2 = 9$. If we want to mimic the usual order of arguments for exponentiation we need to recur on the 2nd argument:

```
_^_ : ℕ → ℕ → ℕ
m ↑ zero  = 1
m ↑ suc n = m * m ↑ n
```

For example 3 ↑ 5 is 3 * 3 * 3 * 3 * 3 * 1. Hence m ↑ 0 = 1 and m ↑ suc n = m * m ↑ n.

4.1.3 The Ackermann function

Each instance of recursion corresponds to using a for-loop in conventional programming. While addition is just defined using one for-loop, multiplication calls addition inside its loop, hence we have a nested for-loop. And exponentiation uses a double nested for-loop. Historically, functions that only use for-loops are called primitive recursive, and all the examples we have seen are examples of primitive recursion. Hilbert, a famous german Mathematician, asked wether all total recursive functions can be defined using primitive recursion. His student Ackermann showed that this is not the case by exhibiting a function that is not primitive recursive because it grows faster than any primitive recursive function. The idea of the Ackermann function is to have one extra parameter which determines the nesting of for loops, that is he defined a function ack : ℕ → ℕ → ℕ such that

```
ack 0 m n = suc n
ack 1 m n = m + n
ack 2 m n = m * n
ack 3 m n = m ↑ n
    ⋮   ⋮
```

The next function, hyperexponentiation that is repeated exponentiation doesn't really have a standard notation anymore. We can define ack using recursion over the natural numbers:

```
ack : ℕ → ℕ → ℕ → ℕ
ack zero                  m n       = suc n
ack (suc zero)            m zero    = m
ack (suc (suc zero))      m zero    = 0
ack (suc (suc (suc zero))) m zero   = 1
ack (suc (suc (suc (suc l)))) m zero = m
ack (suc l)               m (suc n) = ack l m (ack (suc l) m n)
```

To stay consistent with exponentiation we define the functions by recursion over the 2nd argument. The first line defines ack 0 to be just the successor of the 2nd argument because then we can obtain addition by repeating it. The next three lines define the 0 cases for addition m + 0 = m, multiplication m * 0 = 0 and exponentiation m ↑ 0 = 1. The idea for multiplication and exponentiation is to use the neutral element of the previous operation, corresponding to 0 iterations. However, since the is no neutral element for exponentiation, that is no number such that $x^n = n$, we just use the first argument as for addition. The last line uniformly expresses that the function for ack (suc l) is defined by iterating the previous function ack l n times.

Actually the definition above is a bit complicated due to all the different cases for 0, hence often a simplified version of the Ackermann function is used where the levels don't exactly correspond to the usual definitions of the arithmetic operations but have the same nesting of for loops:

```
acks : ℕ → ℕ → ℕ → ℕ
acks zero    m n       = suc n
acks (suc l) m zero    = m
acks (suc l) m (suc n) = acks l m (acks (suc l) m n)
```

Using either ack or acks we can define the function fast:

```
fast : ℕ → ℕ
fast n = ack n n n
```

We can only evaluate the first few instances of fast

```
fast 1 = 1 + 1                    = 2
fast 2 = 2 * 2                    = 4
fast 3 = 3 ↑ 3                    = 27
fast 4 = 4 ↑ (4 ↑ (4 ↑ 4))        = ?
```

Due to the fact that our implementation of numbers using a unary representation we can't even compute fast 4 in a reasonable time. But even using a very efficient implementation of numbers will not save us much, the incredible fast

growing behaviour of fast will spoil any attempt to compute many more results. And indeed, fast grows faster than any primitive recursive function that is any function definable only with for-loops.

4.2 Iterator and recursor

We intuitively explained and justified recursive definitions over the natural numbers but now we shall be more precise and say exactly what recursive definitions are permissable.

In general we define a function $f : \mathbb{N} \to M$ [2] by recursion over the natural numbers

```
f 0 = z
f (suc n) = s (f n)
```

where

```
z : M
s : M → M
```

We can turn this into a higher order function, the iterator [3]:

```
It𝕟 : M → (M → M) → 𝕟 → M
It𝕟 z s zero    = z
It𝕟 z s (suc n) = s (It𝕟 z s n)
```

[4] We call M the *motive* and z,s methods. Our claim is that we can reduce recursion over the natural number to the iterator. Ok, we haven't actually made precise what we mean by recursion over the natural numbers and introducing the iterator is exactly one way to do this.

We can derive dbl using only the iterator:

```
double-it : 𝕟 → 𝕟
double-it = It𝕟 0 (λ double-n → suc (suc double-n))
```

The instances for the methods z and s can be read off the recursive definition of double:

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

The reduction of half is less obvious - here again the recursive definition:

[2] In the following I use A B M : Set as variables as in chapter 2.

[3] In category theory $It\mathbb{N}\ z\ s : \mathbb{N} \to M$ is the universal morphism from the initial algebra. It is also called a *fold* in functional programming, or a *catamorphism*.

[4] Remember that M is one of the names we have declared to be used as implicit parameters for Set.

```
half : ℕ → ℕ
half zero          = zero
half (suc zero)    = zero
half (suc (suc n)) = suc (half n)
```

half doesn't follow exactly the pattern we have stated before but it is reducible to it. The idea is that we define an auxiliary function half-aux which computes the results of half n and half (suc n) simultaneously using a product.

```
half-aux : ℕ → ℕ × ℕ
half-aux zero = zero , zero
half-aux (suc n) = proj₂ (half-aux n) , suc (proj₁ (half-aux n))
```

[5] The first line is justified by the observation that half 0 , half 1 = 0 , 0. The recursive call follows form the fact that we have already half (suc n) as the 2nd component of half-aux n and by the recursive defintion half (suc (suc n)) = half n which explains the 2nd component. Now half-aux is an instance of our scheme and we can define

```
half-aux-it : ℕ → ℕ × ℕ
half-aux-it = Itℕ (zero , zero) (λ p → proj₂ p , suc (proj₁ p))
half-it : ℕ → ℕ
half-it n = proj₁ (half-aux-it n)
```

The definition of _+_ is straightforward, here the recursive version:

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

We can transform it into an instance of our scheme using higher order functions:

```
zero  +_ = λ n → n
suc m +_ = λ n → suc (m + n)
```

And hence we define

```
_+it_ : ℕ → ℕ → ℕ
_+it_ = Itℕ (λ n → n) (λ m+ n → suc (m+ n))
```

[5] The 2nd line can be more nicely written using a local definition:

```
half-aux (suc n) = proj₂ hn, suc (proj₁ hn)
  where hn = half-aux n
```

or even better using **with** which enables pattern matching on an intermediate result:

```
half-aux (suc n) with half-aux n
half-aux (suc n) | half-n , half-sn = half-sn , (suc half-n)
```

The same technique works for multiplication:

```
_*-it_ : ℕ → ℕ → ℕ
_*-it_ = Itℕ (λ n → 0) λ m* n → n + (m* n)
```

A more interesting case is a function which doesn't only use the recursive result but also the input. An example is the factorial function which can be recursively defined we follows:

```
_! : ℕ → ℕ
zero ! = 1
suc n ! = suc n * n !
```

n ! is the number of ways we can put n people on n chairs. If there are no people and no chairs we have to do nothing hence the answer is 1 (this seems to confuse some people). Otherwise in the case (suc n) ! there are suc n ways to place the first person and then n ! to distribute n people on n chairs.

The factorial function is an instance of a slightly modified scheme, to define f : ℕ → A we use:

```
f 0 = z
f (suc n) = s n (f n)
```

where

```
z : A
s : ℕ → A → A
```

As above we can turn this into a general higher order function, called the recursor [6]:

```
Rℕ : M → (ℕ → M → M) → ℕ → M
Rℕ z s zero = z
Rℕ z s (suc n) = s n (Rℕ z s n)
```

We can define the factorial function using only the recursor:

```
_!-r : ℕ → ℕ
_!-r = Rℕ 1 (λ n n! → suc n * n!)
```

We can actually derive Rℕ from Itℕ by (re-)computing the input and the output together:

```
Rℕ-it : M → (ℕ → M → M) → ℕ → M
Rℕ-it {M} z s n = proj₂ (Itℕ z' s' n)
  where
    z' : ℕ × M
```

[6] Called *paramorphism* by Meertens.

```
z' = 0 , z
s' : ℕ × M → ℕ × M
s' (n , m) = suc n , s n m
```

That is we use $\mathbb{N} \times \mathsf{M}$ as the motive, the first component is identical to the input. Hence in the 0-case we use 0 , z and in the successor case given n , m we compute suc n , s n m, hence we are able to supply the extra argument. In the end we have only to extract the 2nd component to obtain the result.

Finally let's have a look at the simplified Ackermann function:

```
acks : ℕ → ℕ → ℕ → ℕ
acks zero    m n        = suc n
acks (suc l) m zero     = m
acks (suc l) m (suc n)  = acks l m (acks (suc l) m n)
```

Can we compute the Ackermann function using the iterator or the recursor? It seems that our scheme exactly captures primitive recursion. However, the traditional notion of primitive recursion doesn't include the use of higher order functions which leads to a much more powerful scheme. And indeed the Ackermann function is definable:

```
acks-it : ℕ → ℕ → ℕ → ℕ
acks-it = Itℕ (λ m n → suc n)
  (λ ack-l m → Itℕ m λ ack-sl-m-n → ack-l m ack-sl-m-n)
```

The main idea here is to translate the pattern using higher order functions as in the case for addition and multiplication, and then use the iterator again to translate the nested pattern.

Indeed, the power of primitive recursion with higher order function is much more than ordinary primitive recursion. We can define all recursive functions that can be proven total in Peano Arithmetic. Indeed a lambda calculus with a type of natural numbers and the recursor is called *System T* and was introduced by Gödel as a functional system corresponding to Peano Arithmetic.

4.2.1 Non primitive recursive functions

There are recursive functions over the natural numbers that cannot be defined directly using primitive recursion. An example is the computation fo the *greatest common divisor* using subtraction. First we define *cut-off subtraction* which stops at 0:

```
_-_ : ℕ → ℕ → ℕ
zero - n = zero
suc m - zero = suc m
suc m - suc n = m - n
```

That is 3 - 2 = 1 while 1 - 2 = 0. Another ingredient is a comparison function comp which gets two numbers m,n and returns different values depending on wether m < n , m = n or m > n:

```
comp : ℕ → ℕ → A → A → A → A
comp zero zero m<n m=n m>n          = m=n
comp zero (suc n) m<n m=n m>n       = m<n
comp (suc m) zero m<n m=n m>n       = m>n
comp (suc m) (suc n) m<n m=n m>n    = comp m n m<n m=n m>n
```

Now we can define gcd where we always subtract the smaller number from the larger one until they are equal:

```
gcd : ℕ → ℕ → ℕ
gcd m n  =  comp m n (gcd m (n - m)) m (gcd (m - n) m)
```

e.g. gcd 56 84 = 27.

However, Agda complains *Termination checking failed* because it cannot see that the arguments were reduced during recursion. [7] And indeed, there is no obvious way to translate this function into one using the iterator or the recursor.

I this case we can save the day by noticing that the recursion is bounded by the sum of the numbers and define:

```
gcd-aux : ℕ → ℕ → ℕ → ℕ
gcd-aux zero m n  =  zero
gcd-aux (suc fuel) m n  =
  comp m n (gcd-aux fuel m (n - m)) m (gcd-aux fuel (m - n) m)
gcd' : ℕ → ℕ → ℕ
gcd' m n  =  gcd-aux (m + n) m n
```

The first argument is often called fuel because it acts like the fuel for a car, we have to pay one unit for each recursive call. However, we have set up our definition so that we can never run out of fuel. Hence the first line in the definition of gcd-aux will never be executed and we could have put in any vaule instead of 0. There are better ways to deal with this situation, especially once we have dependent types.

However, there are recursive and terminating functions which we cannot define using higher order primitive recursion.

4.3 Inductive Types

4.3.1 Lists

The type of natural numbers

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

[7] We can override this warning by using the pragma

{-# TERMINATING #-}

before the declaration of the function.

of an *inductive datatype* which we can view as a recursively defined sum. Another example of an *inductive datatype* is the type of lists, representing finite sequences of a given type:

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

Constructors in this case are the empty list [] (pronounced *nil*) and given a : A and another list l : Lis A we can construct a new list a :: l using the constructor _::_ [8] (pronounced *cons*). All lists are made from [] and _::_, for example 1 :: 2 :: 3 :: [] : List ℕ.

Recursive definitions on lists follow the same idea as for natural numbers. Examples are the length function:

```
length : List A → ℕ
length [] = 0
length (x :: l) = 1 + length l
```

which computes the length of a list; a function that sums the numbers in a list:

```
sumList : List ℕ → ℕ
sumList [] = 0
sumList (n :: ns) = n + sumList ns
```

and the append function which concatenates two lists:

```
_++_ : List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

So for example

```
length (1 :: 2 :: 3 :: []) = 3
sumList (1 :: 2 :: 3 :: []) = 6
(1 :: 2 :: 3 :: []) ++ (4 :: 5 :: []) = 1 :: 2 :: 3 :: 4 :: 5 :: []
```

Append is very similar to addition, indeed addition can be viewed as a special case of append if we identify ℕ with List ⊤. However, unlike _+_, _++_ is not commutative in general, since the order of elements matter.

Lists are a functor, that is an operation that can not only applied to elements but also to functions. This is witnessed by the map-function, that applies a function to every element of a list, producing a new list:

[8] In the language Haskell a single : is used for cons, and a double :: for typing. Type Theoreticians put more emphasis on typing hence they prefer to have the shorter notation for this.

```
map : (A → B) → List A → List B
map f [] = []
map f (x :: xs) = f x :: map f xs
```

So for example

```
map suc (1 :: 2 :: 3 :: []) = 2 :: 3 :: 4 :: []
```

As for natural numbers recursive definitions over lists can be reduced to one combinator, which is usually called fold but which we consequently call ItList:

```
ItList : M → (A → M → M) → List A → M
ItList n c [] = n
ItList n c (a :: l) = c a (ItList n c l)
```

All the recursive functions over lists we have defined can be reduced to ItList using the ideas we have established in the previous section:

```
length-it : List A → ℕ
length-it = ItList 0 (λ a length-l → suc length-l)
sumList-it : List ℕ → ℕ
sumList-it = ItList 0 _+_
_++it_ : {A : Set} → List A → List A → List A
_++it_ = ItList (λ l → l) (λ a m++ l → a :: (m++ l))
map-it : {A B : Set} → (A → B) → List A → List B
map-it f = ItList [] (λ a map-f-l → f a :: map-f-l)
```

We can also derive a recursor for lists and other inductive datatypes which allows to use the input list in the recursive call but we leave the details as an exercise.

4.3.2 Syntax trees

A common application of inductive datatypes is the representation of syntax. An expression like 2 * (3 + 4) can be viewed as a tree generated by constructors for constants and for each of the operators:

```
data Expr : Set where
  const : ℕ → Expr
  _[+]_ : Expr → Expr → Expr
  _[*]_ : Expr → Expr → Expr
```

So our example expression can be represented as follows:

```
const 2 [*] (const 3 [+] const 4) : Expr
```

An obvious function we want to define is the evaluator which computes the numeric value of an expression:

```
eval : Expr → ℕ
eval (const x) = x
eval (e₁ [+] e₂) = eval e₁ * eval e₂
eval (e₁ [*] e₂) = eval e₁ + eval e₂
```

So for example

```
eval (const 2 [*] (const 3 [+] const 4)) = 14
```

As for other inductive types we introduce an iterator for Expr:

```
ItExpr : (ℕ → M) → (M → M → M)
  → (M → M → M)
  → Expr → M
ItExpr c p m (const x) = c x
ItExpr c p m (e₁ [+] e₂) = p (ItExpr c p m e₁) (ItExpr c p m e₂)
ItExpr c p m (e₁ [*] e₂) = m (ItExpr c p m e₁) (ItExpr c p m e₂)
```

I hope that the general scheme for deriving iterators is now evident. The methods correspond to the constructors where every occurence of the datatype is replaced with the motive M. The function which is computed by the iterator recursively replaces each constructor with the method.

We can define the evaluator using the iterator:

```
eval-it : Expr → ℕ
eval-it = ItExpr
  (λ n → n)
  (λ eval-e₁ eval-e₂ → eval-e₁ + eval-e₂)
  (λ eval-e₁ eval-e₂ → eval-e₁ * eval-e₂)
```

4.3.3 Rose trees

The syntax trees from the previous section were binary trees, the nodes created by _[+]_ and _[*]_ each have two subtrees. Using lists of subtrees we can define trees with an arbitrary number of subtrees which are called *rose trees*.

```
data RoseTree : Set where
  node : List RoseTree → RoseTree
```

We don't need a special constructor for leaves, because we can just have a node with no subtrees node [].

As an example we define the following rose tree:

```
rt : RoseTree
rt = node (node []
  :: node (node (node [] :: []) :: node [] :: [])
```

```
    :: node (node [] :: [])
    :: [])
```

As an example for a recursive function on rose trees we count the number of nodes:

```
countNodes : RoseTree → ℕ
countNodes (node ts) = suc (countNodesList ts)
  where countNodesList : List RoseTree → ℕ
    countNodesList [] = 0
    countNodesList (t :: ts) = countNodes t + countNodesList ts
```

Here we use an auxiliary function which counts the nodes in a list of rose trees. We use **where** to do this. [9]

So for example countNodes rt evaluates to 8.

What is the iterator for Rose Trees? We just apply the scheme and obtain the type of methods by replacing RoseTree with the motive. We end up with

```
ItRoseTree : ((List M) → M) → RoseTree → M
```

But how can we recursively apply the iterator to the arguments which are inside a list. A good way to address this is to use map:

```
ItRoseTree m (node ts) = m (map (ItRoseTree m) ts)
```

However, Agda's termination checker isn't clever enough to see that recursion through map will terminate and we have to use the following more elaborate version where we implement an instance of map locally to the iterator:

```
ItRoseTree {M} m (node ts) = m (mapIt ts)
  where mapIt : List RoseTree → List M
    mapIt [] = []
    mapIt (t :: ts) = ItRoseTree m t :: mapIt ts
```

Now we can derive our count function from the iterator. The method is the function sumList which, as we have seen, can be defined using only the iterator for lists:

```
countNodes-it : RoseTree → ℕ
countNodes-it = ItRoseTree (λ ns → suc (sumList ns))
```

4.3.4 Ordinals

All the trees we have seen so far were *finitely branching* that is any node only had finitely many subtrees. We can also consider infinitely branching trees and a nice application of those is a notation for ordinal numbers.

[9] Agda has both **let** and where, but for reasons I don't understand only **where** supports the definition of functions by pattern matching.

The idea of ordinal numbers is that we also consider infinite numbers. That is we have the sequence

$$0, 1, 2, 3, 4, \ldots$$

but now we add an element on the top which we call ω:

$$0, 1, 2, 3, 4, \ldots \omega$$

The idea is that ω is greater than any of the numbers in the sequence $0, 1, 2, 3, 4, \ldots$ and it is the smallest such ordinal. We can continue this and create a new infinite sequence:

$$0, 1, 2, 3, 4, \ldots \omega, \omega + 1, \omega + 2, \omega + 3, \omega + 4, \ldots$$

And again we have an element on the top:

$$0, 1, 2, 3, 4, \ldots \omega, \omega + 1, \omega + 2, \omega + 3, \omega + 4, \ldots \omega + \omega$$

We can continue this game infinitely and produce yet another ordinal $\omega * \omega$ and so on. The main idea is that every ordinal number is associated with the corresponding _<_ relation. In each case this relation is well-founded, that is you can only go down a finite number of steps before reaching 0. Note that this also holds for ω, in the first step we can go down to any finite number and after this it is clear that we will end up at 0. However, there is no bound to the length of this sequence.

Here is the definition of our notation for ordinals. It starts like the definition for natural numbers but then we add another constructor lim which allows us to construct ordinals from infinite sequences:

```
data Ord : Set where
  zero : Ord
  suc : Ord → Ord
  lim : (ℕ → Ord) → Ord
```

We can embed the natural numbers into the ordinals:

```
emb : ℕ → Ord
emb zero = zero
emb (suc x) = suc (emb x)
```

Using emb we can derive ω:

```
ω : Ord
ω = lim emb
```

We can lift the arithmetic operations to ordinals to derive ordinals like $\omega + \omega$ and ω * ω. Before we define addition it is a good idea to note that ordinal addition is not commutative, that is $\alpha + \beta$ is not the same as $\beta + \alpha$. We can understand addition by just putting the ordinals next to each other. So for example $1 + \omega$ looks like this:

$$*, 0, 1, 2, 3, 4, \ldots$$

Here we write $*$ for the 0 in the ordinal 1. On the other hand $\omega + 1$ looks like this

$$0, 1, 2, 3, 4, \cdots *$$

In the first case we have just shifted the ordinals by 1, but the order we obtain is equivalent to the one of ω, that is in the sense of orders the ordinals $1 + \omega$ and ω are equivalent. This is not the case for $\omega + 1$ because this time there is an infinite gap between the finite elements and the new one, which is not equivalent to ω.

While when defining _+_ for natural numbers it didn't matter over which argument we did the recursion for the ordinals it does. And the correct choice is to do recursion over the 2nd argument:

```
_+ord_ : Ord → Ord → Ord
m +ord zero = m
m +ord suc n = suc (m +ord n)
m +ord lim f = lim (λ i → m +ord f i)
```

We define addition for a limit ordinal, that is an ordinal of the form lim f by adding the same element to each element of the sequence and then taking the limit of the sequence. So for example $\omega + \omega$ is obtained by taking the limit of the sequence $\omega, \omega + 1, \omega + 2, \ldots$ matching our intuitive explanation.

Ordinals can be used to quantify the recursive strength of a recursion scheme but also the strength of a logic. A famous example is related to the fight of Hercules against the Hydra. The Hydra is a monsters with many heads which we view as a rose tree. Hercules can chop of one head of the Hydra (that is remove a subtree of the form node []) but as a response the Hydra can grow an arbitrary number of heads at lower levels. However, Hercules will always win after a finite number of steps and be able to chop off all the heads.

The fact that Hercules will win is an instance of a particular ordinal called ϵ_0 which we can obtain by defining multiplication and exponentiation and then take the limit of ω-towers of the form

$$\omega, \omega^{\omega}, \omega^{(\omega^{\omega})}, \ldots$$

Any Hydra configuration can be viewed as an ordinal in this sequence and any move by Hercules chooses a smaller ordinal. However, one can show that the ordinal ϵ_0 corresponds to the logical strength of Arithmetic and hence we cannot prove that the game terminate using only induction and natural numbers.

Before we finish let's derive the iterator for Ord:

```
ItOrd : {M : Set} → M → (M → M) → ((ℕ → M) → M)
  → Ord → M
ItOrd z s l zero = z
ItOrd z s l (suc x) = s (ItOrd z s l x)
ItOrd z s l (lim f) = l (λ i → ItOrd z s l (f i))
```

and implement ordinal addition as an instance:

```
_+ord-it_ : Ord → Ord → Ord
α +ord-it β = ItOrd α suc lim β
```

Note that we explicitly do recursion over the 2nd argument.

4.4 Positivity

Introductions to λ-calculus often start with the untyped λ-calculus which presents the syntax of the λ-calculus without types. We can try to simulate this idea by defining a strange datatype Λ:

```
data Λ : Set where
  lam : (Λ → Λ) → Λ
```

Agda won't accept this definition (for good reasons) but for the moment we can switch this off by saying `NO_POSITIVITY_CHECK`. The idea is that any function $\Lambda \to \Lambda$ gives rise to another untyped λ-term which corresponds to its λ-abstraction. The simplest example is the identity function $\lambda\, x \to x$ which is represented as $\mathsf{lam}\ (\lambda\, x \to x) : \Lambda$. To be able to form more interesting terms we need to use application. However, this is easy to define using pattern matching:

```
app : Λ → Λ → Λ
app (lam f) x = f x
```

Since every untyped λ-term is produced by an abstraction, we can just use this to obtain a function and hence we have application. We can obtain some interesting terms such as self-apply which is

$$\lambda\, x \to x\, x$$

Hence using Λ we define:

```
self-apply : Λ
self-apply = lam (λ x → app x x)
```

What happens if we apply self-apply to itself? This term is called Ω:

$$\Omega = (\lambda\, x \to x\, x)\ (\lambda\, x \to x\, x)$$

we translate this to

```
Ω : Λ
Ω = app self-apply self-apply
```

Now the strange thing about Ω is that when we β-reduce it we get back to Ω:

$$\begin{aligned}\Omega &= (\lambda\, x \to x\, x)\ (\lambda\, x \to x\, x)\\ &= (x\, x)\ [\ x \mapsto (\lambda\, x \to x\, x)\ ]\\ &= (\lambda\, x \to x\, x)\ (\lambda\, x \to x\, x)\\ &= \Omega\end{aligned}$$

This means that we have constructed an element of Λ which is not generated by lam. And indeed if we try to evaluate Ω Agda will hang.

It is the negative occurence of the type we are defining which causes this. Another example of a type which only has a negative occurence is the the following (and again we have to switch off the positivity checker).

```
data Weird₁ : Set where
  foo : (Weird₁ → ⊥) → Weird₁
```

It is hard to explain Weird_1 but that is the point. We can show that Weird_1 is empty using pattern matching. The idea is that if there is an element of Weird_1 then we have a function $\mathsf{Wierd}_1 \to \bot$ and hence we can derive $\bot$:

```
¬weird₁ : Weird₁ → ⊥
¬weird₁ (foo x) = x (foo x)
```

However, now using $\neg\mathsf{weird}_1$ we can actually construct an element of Wierd_1, namely foo ¬weird. But since we have shown that Weird_1 is empty we can derive an element of the empty type:

```
bad : ⊥
bad = ¬weird₁ (foo ¬weird₁)
```

This is certainly bad, because the empty type was supposed to be empty.

These examples illustrate that we need to have some restrictions on the types of constructors when defining an inductive datatype. The question to answer first is what do we actually mean by an inductive datatype? All the elements of an inductive datatype are generated from the constructors. In the case that an argument of the constructor is the type we are defining means that this should be an element which we know already. That is we create the elements in stages, e.g. with the natural numbers in the first step we can only create zero : $\mathbb{N}$, or for the expressions we can only create the constants const 1 : Expr. But in the next step we can create suc n : $\mathbb{N}$ or in the case of Expr we can create for example const 1 [+] const 2 and so on. This also justifies the recursion principles expressed via the iterators. This line of thought explains inductive datatypes where the type appears as an argument to a constructor.

This also works for rose trees but here we have to create a whole list of trees before we can go on creating a new tree.

An interesting example is

```
data T₁ : Set where
  foo : T₁ → T₁
```

What are the elements of this type? Since we have no leaves we can't even get started to create elements. Hence the type is empty. We can actually show this by defining a function into the empty type by recursion:

```
¬t₁ : T₁ → ⊥
¬t₁ (foo x) = ¬t₁ x
```

Understanding infinitely branching trees like Ord is a bit more involved. We first create all the trees we can build in finitely many stages. These include some of the lim-trees but only the one which only use the trees we have constructed before, hence which have only a finite number of different subtrees. Once we have created all of them we have reached a stage ω. Now we can continue to create trees using all the infinitely many we have created up to ω. And so on, we can repeat this process to get more and more trees.

Hence we can allow constructor arguments which are of the form $A \to X$ where X is the type we are defining just now and A is a type we already know. This is called a *strictly positive type* and this is the condition Agda is using to decide wether a datatype declaration is acceptable.

But what about the following type?

```
data Weird₂ : Set where
  foo : ((Weird₂ → Bool) → Bool) → Weird₂
```

In this example the occurence of Weird_2 in the argument is not strictly positive but still positive, i.e. double negative.

Should we allow such types? Agda doesn't think so, but why? First of all our semantic explanation is out of the window, we cannot generate Weird_2 in stages. But is it inconsistent? This doesn't seem to be the case even though the proof needs quite strong logical principles. However, this type is inconsistent with classical logic.

If we assume TND or RAA then the type Bool is the type of propositions and Weird_2 constructs a fixpoint of the double power set and we can use well known constructions (Cantor's diagonalisation) to derive a contradiction.

Hence, while positive datatypes on their own seem to be consistent, they rule out certain assumptions like the assumption of classical logic. Moreover they are not justified semantically. These seem to be good reasons to rule them out.

4.5 Coinductive types

One of the nice features of category theory is the availability of a mirror: for any construction X there is a dual one, which is usually called co-X. For example from the point of category products are dual to sums and indeed sums are sometimes called coproducts. We have seen that recursive sums aka inductive types are very useful, hence now we wonder what can we do with recursive products or coinductive types.

4.5.1 Streams

An example of a coinductive type is the type of streams, or infinite sequences. Given a stream over A we can obtain its head which is an element of A and its tail which is another stream. Hence we define:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

Unlike in the case for sums we have to indicate that we want to define a recursive record by invoking the keyword coinductive. [10]

To construct a stream we just have to present a head and a tail. We can derive a constructor for streams using copattern matching which we have already seen for products:

```
_::_ : A → Stream A → Stream A
head (a :: as) = a
tail (a :: as) = as
```

Indeed we can automatically derive the constructor by saying:

```
record Stream (A : Set) : Set where
  constructor _::_
  coinductive
  field
    head : A
    tail : Stream A
```

We can use copatternmatching which we have already seen for products to define the function

```
from : ℕ → Stream ℕ
```

which for any number n returns the stream of numbers staring with n that is n, n+1, n+2, ...

```
head (from n) = n
tail (from n) = from (suc n)
```

Clearly the head of from n is just n, while the tail is the stream starting with the next number, hence from (suc n).

While we understood inductive types by construction, that is all elements are constructed using the constructors, we understand coinductive types by the way they are destructed. E.g. a stream is anything of which we can take a head and a tail.

The recursive definition of from n is reasonable, because we can always compute the head and the tail of the result of from n, we never get stuck.

Clearly we cannot define from for lists

[10] The reason is that Agda uses the η-law for records which is undecidable for recursive records. Since agda doesn't implement the η-law for sums there is no need to flag recursive sums.

```
fromList : ℕ → List ℕ
fromList n = n :: fromList (suc n)
```

because the recursive definition is not justified by descending along an inductively defined type eg ℕ.

On the other hand while we can define a function that filters out elements from a list given a boolean function:

```
filterL : {A : Set} → (A → Bool) → List A → List A
filterL f [] = []
filterL f (x :: xs) with f x
filterL f (x :: xs) | false = filterL f xs
filterL f (x :: xs) | true = x :: filterL f xs
```

However, Agda will complain if we try to do the same for streams:

```
filterS : {A : Set} → (A → Bool) → Stream A → Stream A
filterS f xs with f (head xs)
filterS f xs | false = filterS f (tail xs)
filterS f xs | true = (head xs) :: filterS f (tail xs)
```

Indeed, we cannot answer the question what is the head of filterS f xs. And indeed there may be none, we can define a stream of false:

```
falses : Stream Bool
head falses = false
tail falses = falses
```

but what is head (filterS (λ x → x) falses) : Bool?

4.5.2 Conatural numbers

Actually streams are not the dual of lists because they do not allow empty streams. We can define a type of colists which contain both finite and infinite sequences but instead we are going to study the dual of natural numbers, the conatural numbers. As we have observed previously the destructor for natural numbers is pred : ℕ → Maybe ℕ. Hence we define conatural numbers as objects that have a predecessor:

```
record ℕ∞ : Set where
  coinductive
  field
    pred∞ : Maybe ℕ∞
```

As we have derived the predecssor for the natural numbers we can now derive 0 and suc for conatural numbers:

```
zero∞ : ℕ∞
pred∞ zero∞ = nothing
```

```
suc∞ : ℕ∞ → ℕ∞
pred∞ (suc∞ n) = just n
```

The rationale behind these definitions should be clear: zero∞ is the conatural number on which pred∞ returns an error, while the sucessor of a conatural number is the one whose predecessor is just that number.

Apart from those finite numbers we can also create an infinite number:

```
∞ : ℕ∞
pred∞ ∞ = just ∞
```

The definition is straightforward: what is the predecessor of ∞? It is certainly not 0 but it is just ∞.

Let's define addition for conatural numbers. It is a good idea to have the definition of addition of natural numbers in mind:

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

In contrast to define _+∞_ we have to determine what is the predeccessor of m +∞ n?

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
pred∞ (m +∞ n) with pred∞ m
pred∞ (m +∞ n) | nothing = pred∞ n
pred∞ (m +∞ n) | just m' = just (m' +∞ n)
```

To find this out we query the first argument. If it was 0, that is pred∞ m returns nothing then it is the predecessor of the 2nd argument. However, if the predecessor is just m' then the result is just (m' +∞ n).

_+∞_ is the infinite extension of _+_. To make this statement a bit more precise, we can define an embedding

```
ℕ→ℕ∞ : ℕ → ℕ∞
ℕ→ℕ∞ zero = zero∞
ℕ→ℕ∞ (suc n) = suc∞ (ℕ→ℕ∞ n)
```

just using recursion. Now the definng property of _+∞_ wrt _+_ is

```
ℕ→ℕ∞ (m + n) ≡ (ℕ→∞ m) +∞ (ℕ→∞ n)
```

We haven't yet developed the machinery to prove this and we havent explained when two coinductive numbers are equal.

4.5.3 Coiterator and corecursor

As for recursion over an inductive type we can derive a general combinator to do corecursion over a coinductive type. To define a function f : W → Stream A corecursively we are using copattern patching:

```
head (f x) = h x
tail (f x) = f (t x)
```

where

```
h : A → W
t : W → W
```

This scheme of *coiteration* can be expressed as one higher order function:

```
ColtStream : {A W : Set} → (W → A) → (W → W)
                         → W → Stream A
head (ColtStream h t x) = h x
tail (ColtStream h t x) = ColtStream h t (t x)
```

As an example the from function can be defined using only the coiterator:

```
from-coit : ℕ → Stream ℕ
from-coit = ColtStream (λ n → n) (λ n → suc n)
```

Similarily in the case of the conatural numbers we can define:

```
Colt ℕ∞ : {W : Set} → (W → Maybe W) → W → ℕ∞
pred∞ (Colt ℕ∞ p x) with p x
pred∞ (Colt ℕ∞ p x) | nothing = nothing
pred∞ (Colt ℕ∞ p x) | just x' = just (Colt ℕ∞ p x')
```

As an example we can derive

```
∞-coit : ℕ∞
∞-coit = Colt ℕ∞ just tt
```

However, it turns deriving _+∞_ is a bit more difficult. Here is the code we have written using copatternmatching:

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
pred∞ (m +∞ n) with pred∞ m
pred∞ (m +∞ n) | nothing = pred∞ n
pred∞ (m +∞ n) | just m' = just (m' +∞ n)
```

We clearly do corecursion on the first argument. However, if the first argument is 0, that is of pred∞ m is nothing we immediately return pred∞ n. This sort of behaviour cannot be directly achieved with the coiterator, but requires a *corecursion*. The corecursor can stop corecursion and just return an answer, hence the method returns a sum:

```
CoR ℕ∞ : {W : Set} → (W → ℕ∞ ⊎ Maybe W) → W → ℕ∞
pred∞ (CoR ℕ∞ p x) with p x
pred∞ (CoR ℕ∞ p x) | inj₁ n = pred∞ n
```

```
pred∞ (CoRℕ∞ p x) | inj₂ nothing = nothing
pred∞ (CoRℕ∞ p x) | inj₂ (just x') = just (CoRℕ∞ p x')
```

Using the corecursor we can define _+∞_. We are also using a case combinator for Maybe

```
caseMaybe : {A M : Set} → M → (A → M) → Maybe A → M
caseMaybe n j nothing = n
caseMaybe n j (just x) = j x


_+∞-corec_ : ℕ∞ → ℕ∞ → ℕ∞
m +∞-corec n =
  CoRℕ∞ (λ x → caseMaybe (inj₁ n) (λ m' → inj₂ (just m')) (pred∞ x)) m
```

The corecursor is dual to the recursor, which can be also formulated using a product but this is curried in the usual formulation. And as for the recursor, the corecursor can be derived from the coiterator.

4.6 History

Guiseppe Peano lived in Turino in the late 19th century. He wrote down the rules of reasoning for the natural numbers which is now in his honour called *Peano Arithmetic*. A proper account of Peano Arithmetic uses predicate logic which we haven't yet introduced. Peano only used addition and multiplication in his presentation but it can be shown that all primitive recursive functions can be derived. Interestingly, this ceases to be the case if you leave out multiplication which leads to something called Pressburger Arithmetic.

By primitive recursive functions one means traditionally functions which can be derived using no function types, that is with Rℕ but only products and ℕ can be used. This intuitively corresponds to for-loops in programming. Ackermann formulated his function to show that there are computable functions which cannot be defined with primitive recursion.

It was Gödel who formulated a system which can express all functions that are definable in Peano arithmetic. This is called System T [11] and it is basically what we have presented, i.e. you have Rℕ and function types. There are still functions which are not definable in this system, corresponding to the problem of Archilles and the hydra which we have mentioned already in the section on ordinals.

Our presentation of inductive types goes back to algebraic datatyes and pattern matching invented by my PhD supervisor Rod Burstall which were first implemented in a language called Hope [BMS80]. Subsequently algebraic datatypes and pattern matching was used in many functional programming languages including Miranda and Haskell. Coinductive types were investigated by Coquand [Coq93] who also formulated the guarded corecursion principle we

[11] Gödel introduced System T in [Göd58] (in german).

are using. Copatternmatching was introduced much later in the context of Type Theory by Andreas Abel and and Brigitte Pientka [AP13].

We only briefly touched upon the untyped λ-calculus which, together with untyped combinatory logic, is a basic computational mechanism indeed an alternative to Turing machines. A good overview over the untyped λ-calculus is Barendregt's book [B+84].

4.7 Exercises

1. Define the following functions using pattern matching and structural recursion on the natural numbers.

   (a) Define a function even that determines wether its input is even.

   $$\mathsf{even?} : \mathbb{N} \to \mathsf{Bool}$$

   Examples :

   $$\begin{aligned} \mathsf{even?}\ 3 &= \mathsf{false} \\ \mathsf{even?}\ 6 &= \mathsf{true} \end{aligned}$$

   (b) Define a function sum that sums the numbers from 0 until n-1.

   $$\mathsf{sum} : \mathbb{N} \to \mathbb{N}$$

   Examples:

   $$\begin{aligned} \mathsf{sum}\ 2 &= 3 \\ \mathsf{sum}\ 10 &= 55 \end{aligned}$$

   (c) Define a function max that calculates the maximum of 2 numbers.

   $$\mathsf{max} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

   Examples:

   $$\begin{aligned} \mathsf{max}\ 2\ 3 &= 3 \\ \mathsf{max}\ 10\ 1 &= 10 \end{aligned}$$

   (d) Define a function fib which calculates the nth item of the Fibonacci sequence: $1, 1, 2, 3, 5, 8, 13, \ldots$ (each number is the sum of the two previous ones).

   $$\mathsf{fib} : \mathbb{N} \to \mathbb{N}$$

   Examples:

   $$\begin{aligned} \mathsf{fib}\ 0 &= 1 \\ \mathsf{fib}\ 6 &= 13 \end{aligned}$$

(e) Define a function eq that determines wether two numbers are equal.

```
eq? : ℕ → ℕ → Bool
```

Examples:

```
eq? 4 3 = false
eq? 6 6 = true
```

(f) Define a function rem such that rem m n returns the remainder when dividing m by suc n (this way we avoid division by 0).

```
rem : ℕ → ℕ → ℕ
```

Examples:

```
rem 5 1 = 1
rem 11 2 = 2
```

(g) Define a function div such that div m n returns the result of dividing m by suc n (ignoring the remainder).

```
div : ℕ → ℕ → ℕ
```

Examples:

```
div 5 1 = 2
div 11 2 = 3
```

2. Define all the functions of the previous question but this time only use the iterator It$\mathbb{N}$ and/or the recursor R$\mathbb{N}$.

   Use the following naming convention: if the function was called f then call it f-i if you only use the iterator, but f-r if you use the recusor (and possibly the iterator).

3. In this exercise we implement tree-sort to sort a list of natural numbers.

   We define the type of trees with labels on the nodes.

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

   Here is an example tree:

```
node (node leaf 1 (node leaf 2 leaf)) 5 leaf
```

   Define a function tree2list which collects all the leaves from left to right in a list.

```
tree2list : {A : Set} → Tree A → List A
```

Here is a simple unit test

```
tree2list t                    = 1 :: 2 :: 5 :: []
```

We are going to produce sorted trees. A sorted tree is one where for every node the leaves on the left have smaller values and the leaves on the right have larger values.

I give you a comparison function on $\mathbb{N}$.

```
_⩽?_ : ℕ → ℕ → Bool
zero ⩽? m = true
suc n ⩽? zero = false
suc n ⩽? suc m = n ⩽? m
```

Write a function that transforms a list of natural number into a sorted tree.

```
list2tree : List ℕ → Tree ℕ
```

Here is a test:

```
list2tree (10 :: 2 :: 1 :: 5 :: [])
  = node (node leaf 1 (node leaf 2 leaf)) 5 (node leaf 10 leaf)
```

Hint: it may be a good idea to write a function

```
insert : ℕ → Tree ℕ → Tree ℕ
```

that inserts one value into a sorted tree maintaining its sortedness.

Using the previous function it is now easy to define tree-sort a function that sorts lists.

```
tree-sort : List ℕ → List ℕ
```

Here is a unit test for treesort.

```
tree-sort (10 :: 2 :: 1 :: 5 :: [])        = 1 :: 2 :: 5 :: 10 :: []
```

4. Reimplement tree-sort using only the iterators for lists and trees (also called folds).

   The functions should **not** use pattern matching on lists or trees or any recursion but only use the combinators defined here:

```
foldList : {A : Set} {M : Set} →
  M → (A → M → M) → List A → M
foldList mnil mcons [] = mnil
foldList mnil mcons (x :: xs) = mcons x (foldList mnil mcons xs)
foldTree : {A : Set} {M : Set} →
  M → (M → A → M → M) → Tree A → M
foldTree mleaf mnode leaf = mleaf
foldTree mleaf mnode (node l x r) =
  mnode (foldTree mleaf mnode l) x (foldTree mleaf mnode r)
```

If the original function is called x then the function with folds only is called x-f.

5. We have already defined addition for conatural numbers:

```
_+∞_ : ℕ∞ → ℕ∞ → ℕ∞
prd∞ (m +∞ n) with prd∞ m
prd∞ (m +∞ n) | just m' = just (m' +∞ n)
prd∞ (m +∞ n) | nothing = prd∞ n
```

Our goal is now to define multiplication:

```
_*∞_ : ℕ∞ → ℕ∞ → ℕ∞
```

This is not so easy because we need to make sure that the recursion is syntactically guarded, i.e. the recursive call has to appear directly at the right hand side not inside another function.

Hint: Define an auxilliary function first which solves a more general problem.

Now here is a simple unit test:

```
ℕ∞→ℕ (ℕ→ℕ∞ 3 *∞ ℕ→ℕ∞ 5) = 15
```

6. Define the map operations for all the examples of functors:

```
T-List-map : (A → B) → T-List C A → T-List C B
T-Expr-map : (A → B) → T-Expr A → T-Expr B
T-RoseTree-map : (A → B) → T-RoseTree A → T-RoseTree B
T-Ord-map : (A → B) → T-Ord A → T-Ord B
```

Chapter 5

Dependent types

We are now getting to one of the main innovations of Type Theory: dependent types. In general a dependent type is a function with codomain Set. Examples are vectors Vec A : ℕ → Set where Vec A n is a sequence elements of A of length n, and Fin : ℕ → Set where Fin n corresponds to numbers less then n.

Actually we have already seen dependent types. For example List : Set → Set is a dependent type indexed by Set. Hence *Polymorphism* is actually subsumed by dependent types! Dependent types can be indexed by several indices, and indeed Vec : Set → ℕ → Set if we spell out all the dependencies.

We will discuss Π-types which generalize function types and Σ-types which generalize products. There are a number of interesting relations between dependent type formers and simply typed ones and we will also extend the *type arithmetic* from 2.5.4 to dependent types.

5.1 Vectors and dependent functions

We are going to look at some examples for dependent functions using vectors. I will postpone the actual definition of vectors until I have explained some syntactic conventions. Vec A n is the type of sequences of elements of A of length n, it is a refinement of the type of lists and we use the same constructors. For example 1 :: 2 :: 3 :: [] : Vec ℕ 3.

We can define a function that produces a list of 0s of length n a list:

```
zeroes-l : ℕ → List ℕ
zeroes-l zero    = []
zeroes-l (suc n) = 0 :: zeroes-l n
```

E.g. zeroes-l 3 = 0 :: 0 :: 0 :: []. Using vectors we can define an analogous function with a more precise type because we know that zeroes-l n will produce a list of length n. So instead we output a vector of the length given by the input.

```
zeroes-v : (n : ℕ) → Vec ℕ n
```

```
zeroes-v zero     = []
zeroes-v (suc n)  = 0 :: (zeroes-v n)
```

We see that now the function type is *dependent* and the function type constructor actually introduces a variable.

In this example we can also see that we can often turn non-dependent functions into dependent ones with a more precise type. Let's look at another example: append. We have already seen the append function on lists which was defined infix but for the purpose of the current discussion I am going to redefine it in prefix form:

```
append-l : List A → List A → List A
append-l []       ys = ys
append-l (x :: xs) ys = x :: (append-l xs ys)
```

We can define a very similar operation on vectors, which we call append-v which appends two vectors and records in its type that the length of the result is the sum of its inputs:

```
append-v : (i j : ℕ) → Vec A i → Vec A j → Vec A (i + j)
append-v zero    j [] w       = w
append-v (suc i) j (x :: v) w = x :: (append-v i j v w)
```

Here the type (i j : ℕ) → ... is just a shorthand for (i : ℕ) → (j : ℕ) → .... Let's define some vectors

```
v1 : Vec ℕ 2
v1 = 1 :: 2 :: []
v2 : Vec ℕ 3
v2 = 3 :: 4 :: 5 :: []
```

and append them using append-v:

```
append-v 2 3 v1 v2 : Vec ℕ 5
```

using that 2 + 3 = 5. And this term will evaluate to

```
1 :: 2 :: 3 :: 4 :: 5 :: [] : Vec ℕ 5
```

Already at this point we note that always having to supply the indices for an operation like append-v will clutter up the code and potentially make it unreadable. However, it is also completely unnecessary because we can deduce them from the subsequent arguments.

To avoid this clutter dependently typed languages have introduced the notion of *implicit arguments* which are arguments which are filled in automatically by the type checker. To make an argument implicit we replace (x : A) → ... by {x : A} → .... Hence we can write

```
append-v : {i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
```

and when applying append-v we just write

```
append-v v1 v2 : Vec ℕ 5
```

and the type checker will insert the hidden arguments 2 and 3 which it can infer from the later arguments. We can also make implicit arguments explicit again by using curly brackets again:

```
append-v {2} {3} v1 v2 : Vec ℕ 5
```

Actually using implicit parameters we can also switch to the more readable infix definition of append for vectors resembling our original definition for lists:

```
_++v_ : {i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
[]       ++v w = w
(x :: v) ++v w = x :: (v ++v w)
```

We have already seen implicit arguments when looking at polymorphic functions like length : List A → ℕ which exploited our convention that A is a set parameter. Agda will expand this to

```
length : {A : Set} → List A → ℕ
```

and in an application like length (1 :: 2 :: 3 :: []) the implicit parameter ℕ is inserted, i.e. this expression *elaborates* to length {ℕ} (1 :: 2 :: 3 :: []).

We see that not only are datatypes like List : Set → Set are actually dependent types but also the mystery of polymorphic functions can be subsumed by dependent function types without much notational overhead.

And indeed the constructor _::_ for lists already has an implicit parameter namely:

```
_::_ : {A : Set} → A → List A → List A
```

and hiding it enables us to use the customary infix notation.

After having explained this syntactic convention, I can reveal the actual definition of vectors:

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

This is an inductive definition, very similar to the inductive definition of lists. As before A : Set is a parameter but we define not a Set but a dependent type ℕ → Set. The constructors create elements for different instances of Vec A.

We can also see that the overloaded [1] _::_ for vectors has actually two implicit arguments:

[1] Agda allows the overloading of constructors since it can determine which actual constructor is meant from the type.

```
_::_ : {A : Set} {n : ℕ} → A → Vec A n → Vec A (suc n)
```

You may notice that the dependency on Set and on the ℕ in the declaration of Vec looks different. This is partly convenience since A is the same in the whole declaration but in this case it also has an impact on keeping Vec A n in Set but this is something which we will discuss later in more detail.

5.2 The family of finite sets

Using dependent types we can avoid run-time errors such as the typical *index out of range error* we get when accessing an array. As a simple example lets look at the indexing operation for lists. We would like to implement an operation:

```
_!!_ : List A → ℕ → A
```

but while we can fill in the cases where the list is a cons we get stuck on the case where it is empty:

```
_!!_ : List A → ℕ → A
[] !! n       = ?
(x :: xs) !! zero  = x
(x :: xs) !! suc n = xs !! n
```

Indeed there is no way to complete this program since A could be empty.

We can handle the error explicitly by using Maybe:

```
_!!_ : {A : Set} → List A → ℕ → Maybe A
[] !! n       = nothing
(x :: as) !! zero  = just x
(x :: as) !! suc n = as !! n
```

This is ok and we can use the fact that Maybe is a monad to write programs with error handling. However, we still may get an error anywhere in the middle of a computation.

Wouldn't it better, if we could show statically that no such error can occur? And indeed we can use vectors for this purpose. The other ingredient is Fin which provides us with the allowed index-range of a vector. As Vec was a refinement of List, Fin is a refinement of ℕ:

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

The idea is that every non-empty finite type has a zero and that the elements of Fin n get embedded into Fin (suc n) using suc for Fin. That is the first levels

of Fin look like this

$$\begin{aligned}
&\mathsf{Fin}\ 0 = \{\}\\
&\mathsf{Fin}\ 1 = \{\mathsf{zero}\ \{0\}\}\\
&\mathsf{Fin}\ 2 = \{\mathsf{zero}\ \{1\}, \mathsf{suc}\ \{1\}\ (\mathsf{zero}\ \{0\})\}\\
&\mathsf{Fin}\ 3 = \{\mathsf{zero}\ \{2\}, \mathsf{suc}\ \{2\}\ (\mathsf{zero}\ \{1\}), \mathsf{suc}\ \{2\}\ (\mathsf{suc}\ \{1\}\ (\mathsf{zero}\ \{0\}))\}\\
&\quad\vdots\ \ \vdots
\end{aligned}$$

Here it is actually helpful to use the convention to make the implicit arguments explicit.

Now we can use Vec and Fin to write a safe version of the lookup function:

```
_!!v_ : {A : Set} {n : ℕ} → Vec A n → Fin n → A
(x :: v) !!v zero  = x
(x :: v) !!v suc i = v !!v i
```

Using _!!v_ we can access

```
v1 !!v (suc zero) = 2
```

but

```
v1 !!v (suc (suc (suc zero)))
```

isn't well-typed.

5.3 Π-types and Σ-types

In the previous section we have encountered dependent function types which are also called Π-types. [2] In general we have a type A : Set and a dependent type B : A → Set and we can form (x : A) → B x : Set. We can understand the simple function type A → B as a special case where the 2nd type doesn't actually depends on the first and hence we omit the binding.

```
A → B = (_ : A) → B
```

In all other aspects dependent function types behave very much like simple function types. The application rule now reads that given f : (x : A) → B and a : A we obtain f a : B a. We can define dependent function either by an explicit definition or using λ-abstraction.

Previously, after introducing _→_ we defined products _×_. And indeed as Π types generalize → we also have a dependent generalisation of ×, the Σ-type. As for _×_ this is just an instance of a record type:

[2] I will explain later in this chapter where the Π comes from.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

Elements of Σ A B are pairs a , b where the 1st component is a : A and the 2nd component is b : B a, that is the type of the 2nd component depends on the 1st. Correspondingly we have the projections

```
proj₁ : Σ A B → A
proj₂ : (a : Σ A B) → B (proj₁ a)
```

Note that the type of the second projection is a dependent function type. To improve readability I am introducing a special syntax for Σ-types and write Σ[x ∈ A] P for Σ A (λ x → P). [3]

A simple example would be a type of flexible vectors:

```
FlexVec : Set → Set
FlexVec A = Σ[ n ∈ ℕ ] Vec A n
```

An element of FlexVec A is a pair (n , v) where n : ℕ and v : Vec A n. It isn't hard to see that FlexVec A is equivalent to List A and indeed we can implement the standard functions on lists:

```
[]fv : FlexVec A
[]fv = 0 , []
_::fv_ : A → FlexVec A → FlexVec A
a ::fv (n , v) = suc n , (a :: v)
```

I leave it as an exercise to derive ItList for FlexVec.

As before for function types products are a special case of Σ-types:

```
A × B = Σ[ _ ∈ A ] B
```

However, there is also a different relation we shall explore.

5.4 Relating simple and dependent type formers

You may have noticed that we haven't got a dependent version of sums _⊎_. However, doesn't the use of the symbol Σ indicate that Σ-types should be related to sums and not products?

And indeed we can derive a binary sums from Σ-types and if_then_else_. [4]

[3] I am using ∈ here because Agda doesn't allow me to use : in abbrevations.

[4] if_then_else_ was defined in section 3.6, but only for Set. Here I need a *level polymorphic* one to return sets. I will explain this later.

```
_⊎'_ : Set → Set → Set
A ⊎' B = Σ[ x ∈ Bool ] if x then A else B
```

The idea is that an element of $A \uplus' B$ is a pair (b , x) consisting of a boolean b representing the choice between the two alternatives and then depending on this choice an element of either A or B. Given this idea it is straightforward to derive the injections:

```
inj₁' : A → A ⊎' B
inj₁' a = true , a
inj₂' : B → A ⊎' B
inj₂' b = false , b
```

We can also derive case using pattern matching:

```
case' : (A → C) → (B → C) → A ⊎' B → C
case' f g (false , b) = g b
case' f g (true , a)  = f a
```

In the same vain we can derive _×_ from Π:

```
_×'_ : Set → Set → Set
A ×' B = (x : Bool) → if x then A else B
```

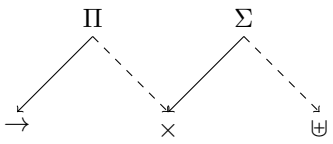
An element of A ×' B is simply a dependent function from booleans to either A or B depending on the input. Hence, the projections are realised by application:

```
proj₁' : A ×' B → A
proj₁' f = f true
proj₂' : A ×' B → B
proj₂' f = f false
```

The pairing constructor introduces a dependent function by pattern matching:

```
_,'_ : A → B → A ×' B
(a ,' b) false = b
(a ,' b) true  = a
```

We can relate all the basic type formers in one picture:



Here → goes from the dependent to the non-dependent version while -› goes from the infinite version to the binary one as discussed above.

You may notice that there are two ways to derive ×: either is the non-dependent version of Σ or as the binary case of Π. This corresponds to the

fact that $\times$ can be viewed inductively – if derived from Σ – or coinductively, if derived from Π.

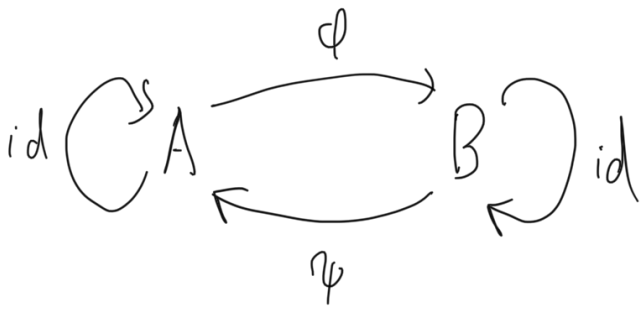
This also explains the use of symbols: Π-types are really infinite products and Σ-types are infinite sums.

5.5 Arithmetic of types

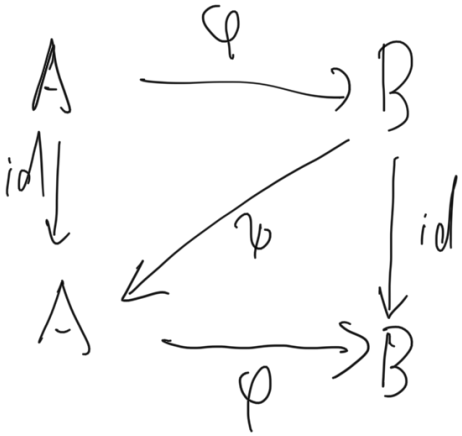
In section 2.5.4 we observed that there is a nice connection between operation on finite types and operations on numbers. Using Fin we can make this relation more precise:

$$\begin{aligned} \mathsf{Fin\ m} \uplus \mathsf{Fin\ n} &\cong \mathsf{Fin\ (m + n)} \\ \mathsf{Fin\ m} \times \mathsf{Fin\ n} &\cong \mathsf{Fin\ (m * n)} \\ \mathsf{Fin\ m} \to \mathsf{Fin\ n} &\cong \mathsf{Fin\ (n \uparrow m)} \end{aligned}$$

By $\mathsf{A} \cong \mathsf{B}$ we mean *A and B have the same number of elements.* This can be made precises by saying that we have functions $\phi : \mathsf{A} \to \mathsf{B}$ and $\psi : \mathsf{B} \to \mathsf{A}$ which are inverse to each other, that is $\phi \circ \psi = \mathsf{id}$ and $\phi \circ \psi = \mathsf{id}$. I usually draw the following categorical diagram to depict this situation:



But I realized that it is not clear what compositions should commute. A better diagram is:



In any case since I still haven't introduced equality types [5] we will limit ourselves to just talking about the finctions in both direction. Hence I define:

```
record _~_ (A B : Set) : Set where
  field
    φ : A → B
    ψ : B → A
```

As an example we can give translations for Bool ∼ Fin 2.

```
bool~2 : Bool ∼ Fin 2
φ bool~2 false = zero
φ bool~2 true = suc zero
ψ bool~2 zero = false
ψ bool~2 (suc zero) = true
```

It should be clear that these two functions are indeed inverse to each other.

As an example I construct the first equivalence:

```
sum-eq : {m n : ℕ} → (Fin m ⊎ Fin n) ∼ Fin (m + n)
```

The idea is that we implement the operations for Fin m ⊎ Fin n on Fin (m + n) and then use this for the translation.

The first injection is an embedding which keeps the value the same but changes the index:

```
inj₁f : {m n : ℕ} → Fin m → Fin (m + n)
inj₁f zero = zero
inj₁f (suc i) = suc (inj₁f i)
```

[5] They are coming soon, but the equivalences involving functions realy on functional extensionality which is only provable in cubical type theory.

The second injection has to shift the elements over the first m that is it applies m successors:

```
inj₂f : {m n : ℕ} → Fin n → Fin (m + n)
inj₂f {zero} i = i
inj₂f {suc m} x = suc (inj₂f {m} x)
```

This is also an example where we make a hidden parameter explicit. We didn't really have to supply {m} in the recursive call but writing it here actually increases readability.

We can now use the defined injections as the translations of the actual injections and hence implement the first translation:

```
φ sum-eq (inj₁ x) = inj₁f x
φ sum-eq (inj₂ y) = inj₂f y
ψ sum-eq = ?
```

To implement the inverse direction we need to implement case on Fin (m + n) and then use the original injections depending on the interval we are in.

```
casef : {m n : ℕ} {C : Set} → (Fin m → C) → (Fin n → C)
   → Fin (m + n) → C
casef {zero} f g x = g x
casef {suc m} f g zero = f zero
casef {suc m} f g (suc i) =
  casef {m} (λ j → f (suc j)) g i
```

To achieve this we recur both over the first index m and the input x : Fin (m + n). If m = 0 we know that we are in the 2nd case and we can just apply g. Otherwise suc m we check the input: if it is zero we are certainly in the first case and we just use f zero. Otherwise we recur on m and i but we need to adjust f by composing it with the successor.

Using casef we can complete our translation:

```
φ sum-eq (inj₁ x) = inj₁f x
φ sum-eq (inj₂ y) = inj₂f y
ψ sum-eq = casef inj₁ inj₂
```

We can now extend the equivalence to Π- and Σ-types. In Mathematics we use them to represent products and sums over a finite series (actually also infinite but we won't cover this here):

$$\Sigma_{i=0}^{i<n} a_i = a_0 + a_1 + \cdots + a_{n-1}$$
$$\Pi_{i=0}^{i<n} a_i = a_0 * a_1 * \cdots * a_{n-1}$$

Using finite types we can implement these operations. The sequence a is simply represented as a function Fin n → ℕ. We define

```
Σℕ : (n : ℕ) (a : Fin n → ℕ) → ℕ
Σℕ zero vs    = 0
Σℕ (suc n) a  = a zero + Σℕ n (λ i → a (suc i))
```

and using the same pattern:

```
Πℕ : (n : ℕ) (a : Fin n → ℕ) → ℕ
Πℕ zero a  = 1
Πℕ (suc n) a  = a zero * Πℕ n (λ i → a (suc i))
```

We now have the following equivalences:

Σ[i ∈ Fin n] Fin (a i))	∼	Fin (Σℕ n a)
((i : Fin n) → Fin (f i))	∼	Fin (Πℕ n f)

which is one explanation of the use of Π and Σ for the operations on types.

Here is an example to illustrate the equivalence for Σ-types. Let

```
i : ℕ
i = 3
a : Fin i → ℕ
a zero = 3
a (suc zero) = 1
a (suc (suc n)) = 2
```

Now clearly

```
Σℕ i a = 3 + 1 + 2
       = 6
```

The following table shows the equivalence:

0 ↦ (0 , 0)	1 ↦ (0 , 1)	2 ↦ (0 , 2)
3 ↦ (1 , 0)		
4 ↦ (2 , 0)	5 ↦ (2 , 1)	

I leave it as an exercise to construct the equivalences. The idea is the same as I have shown in the example of for _⊎_ we just have to implement the basic operations on the type for the finite representation.

5.6 History

Dependent types were the main innovation which Per Martin-Löf introduced when developing his Type Theory in the early 1970ies [ML75] . Hence it is sometimes called *dependent type theory*. One motivation for dependent types is to extend the proposition as types explanation to predicate logic, which we will study in the next chapter.

Martin-Löf already mentioned the possibility of implementing his theory and this challenge was taken up in the 1980ies by Bob Constable and his group who implemented the NuPRL system, where NuPRL is short for *Nearly ultimate Proof Representation Language* [G+95].

A comprehensive overview over Martin-Löf's Type Theory is given in [MLS84] which was coauthored with Giovanni Sambin. This book presents what we now call *extensional type theory* a presentation of *intensional type theory* which is closer to what is implemented in Agda can be found in [NPS90].

My own first encounter with dependent types was Randy Pollack's LEGO system [LP92] which was based on Thierry Coquand's calculus of constructions [LP92] which combined dependent types and Girard's System F [Gir72]. The calculus of constructions was also the basis of the Coq system [dt20] which has been in development since the 1990ies and which is still very popular.

Apart from using it for proof assistants dependent types are very useful for programming. A relatively early language with dependent types is Cayenne by Lennart Augustsson [Aug98]. The Agda system we are using was developed by Ulf Norell [Nor08] at Gothenburg. It was influenced by earlier implementations of dependently typed languages in Gothenburg (like the ALF system [AGNvS94]) and got its name from a language developed by Catarina Coquand. It was also influenced by Conor's McBride's Epigram system which famously featured a 2-dimensional syntax [McB04]. Edwin Brady developed Idris emphasising the programming aspect of dependent types [Bra17] using type-driven program development.

5.7 Exercises

1. We implement some basic operations on vectors and matrices of natural numbers.

   We define vectors using Vec:

   ```
   Vector : ℕ → Set
   Vector m = Vec ℕ m
   ```

   Here Vector n is an n-dimensional vector. Here are some examples:

   ```
   v1 : Vector 3
   v1 = 1 :: 2 :: 3 :: []
   v2 : Vector 3
   v2 = 2 :: 3 :: 0 :: []
   ```

   Implement the multiplication of a skalar with a vector:

   ```
   _*v_ : {n : ℕ} → ℕ → Vector n → Vector n
   ```

   Here is an example:

   ```
   2 *v v1 = 2 :: 4 :: 6 :: []
   ```

Next we implement vector addition:

```
_+v_ : {n : ℕ} → Vector n → Vector n → Vector n
```

Here is an example:

```
v1 +v v2 = 3 :: 5 :: 3 :: []
```

Next we introduce matrices. A matrix is simply a Vec of row vectors:

```
Matrix : ℕ → ℕ → Set
Matrix m n = Vec (Vector n) m
```

Hence Matrix m n is an m×n-Matrix.

Here are some examples:

```
id3 : Matrix 3 3
id3 = (1 :: 0 :: 0 :: [])
  :: (0 :: 1 :: 0 :: [])
  :: (0 :: 0 :: 1 :: [])
  :: []
inv3 : Matrix 3 3
inv3 = (0 :: 0 :: 1 :: [])
  :: (0 :: 1 :: 0 :: [])
  :: (1 :: 0 :: 0 :: [])
  :: []
m3 : Matrix 3 3
m3 = (1 :: 2 :: 3 :: [])
  :: (4 :: 5 :: 6 :: [])
  :: (7 :: 8 :: 9 :: [])
  :: []
m4 : Matrix 3 2
m4 = (1 :: 2 :: [])
  :: (4 :: 5 :: [])
  :: (7 :: 8 :: [])
  :: []
```

The next task is to define the multiplication of a vector with a matrix:

```
_*vm_ : {m n : ℕ} → Vector m → Matrix m n → Vector n
```

The vector is now viewed as a column vector. Here are some examples:

```
v1 *vm m4 = 30 :: 36 :: []
v1 *vm m3 = 30 :: 36 :: 42 :: []
```

If the input vector is empty you have to produce a vector of 0s with the size given by the matrix. This calls for an axilliary function.

Finally, the task is to implement matrix multiplication. It is recommended to make use of the function we have just defined previously.

```
_*mm_ : {l m n : ℕ} → Matrix l m → Matrix m n → Matrix l n
```

Here are several examples:

```
inv3 *mm m3 = (7 :: 8 :: 9 :: []) ::
              (4 :: 5 :: 6 :: []) ::
              (1 :: 2 :: 3 :: []) :: []
m3 *mm m4 = (30 :: 36 :: []) ::
            (66 :: 81 :: []) ::
            (102 :: 126 :: []) :: []
m3 *mm m3 = (30 :: 36 :: 42 :: []) ::
            (66 :: 81 :: 96 :: []) ::
            (102 :: 126 :: 150 :: []) :: []
```

2. We continue the topic of matrices. The task is now to define the transpose of a matrix:

   ```
   transpose : {m n : ℕ} → Matrix m n → Matrix n m
   ```

   There are certainly several ways to solve this exercise but a nice way is to first establish combinators to establish that `F A = Vec A n` is a *applicative* functor. That is we have operations

   ```
   return : A → F A
   app : F (A → B) → F A → F B
   ```

   which are subject to some laws which we will ignore just now.

3. Define the following operations on Fin:

   ```
   max : {n : ℕ} → Fin (suc n)
   emb : {n : ℕ} → Fin n → Fin (suc n)
   rev : {n : ℕ} → Fin n → Fin n
   ```

   where max computes the maximal element of a finite set, e.g. `max {2} = suc (suc zero)`; emb embeds `Fin n` into `Fin (suc n)` without changing the numerical value, e.g. `emb {2} (suc zero) = suc zero`; while rev reverts the order, e.g. `rev {4} (suc zero) = suc (suc zero)` and vice versa `rev {4} (suc (suc zero)) = suc zero`.

4. Derive the translations for _×_ and _→_ and their numerical counterparts. That is derive:

   ```
   prod-eq : {m n : ℕ} → (Fin m × Fin n) ∼ Fin (m * n)
   exp-eq  : {m n : ℕ} → (Fin m → Fin n) ∼ Fin (n ↑ m)
   ```

5. Derive the translations for Σ- and Π-types and their numerical counterparts. That is derive:

$$\begin{array}{l}\Sigma{-}\mathsf{eq} : \{\mathsf{m} : \mathbb{N}\}\,\{\mathsf{f} : \mathsf{Fin\ m} \to \mathbb{N}\} \\ \qquad \to (\Sigma[\ \mathsf{i} \in \mathsf{Fin\ m}\ ]\ \mathsf{Fin}\ (\mathsf{f\ i})) \quad \sim \mathsf{Fin}\ (\Sigma\mathbb{N}\ \mathsf{m\ f}) \\ \Sigma{-}\mathsf{eq} : \{\mathsf{m} : \mathbb{N}\}\,\{\mathsf{f} : \mathsf{Fin\ m} \to \mathbb{N}\} \\ \qquad \to ((\mathsf{i} : \mathsf{Fin\ m}) \to \mathsf{Fin}\ (\mathsf{f\ i})) \sim \mathsf{Fin}\ (\Pi\mathbb{N}\ \mathsf{m\ f})\end{array}$$

Chapter 6

Predicate logic

Predicate logic is the language of Mathematics. Unlike propositional logic which on its own isn't good for anything we can express any mathematical concept in predicate logic. This is why predicate logic has been used to by Peano to present the theory of natural numbers and later by Zermelo and Fraenkel to write down the axioms of set theory. However, from the type-theoretic perspective we don't have to present the formalism of predicate logic but as for propositional logic we just need to explain how we can represent the idioms of predicate logic in Type Theory using the propositions as types translation.

In chapter 3 we introduced propositional logic with the connectives $_\wedge_$ (and), $_\vee_$ (or), implication $_\Rightarrow_$, negation $\neg_$, logical equivalence $_\Leftrightarrow_$ and the logical constants True and False. Now we are going to extend this to predicate logic and we are going to introduce the quantifiers $\forall$ (for all) and $\exists$ (exists) and the polymorphic relation $_\equiv_$ (equality). [1]

6.1 Predicates and Quantifiers

What is a predicate? Actually predicate is just another word for property. An example is the predicate $\mathsf{Even} : \mathbb{N} \to \mathsf{prop}$, a predicate is a function from some domain to propositions. Since we assume $\mathsf{prop} = \mathsf{Set}$ a predicate is nothing but a dependent type. So for example $\mathsf{Even}\ 2 : \mathsf{prop}$ is the proposition that 2 is even, while $\mathsf{Even}\ 3 : \mathsf{prop}$ is the proposition that 3 is even. We expect that the former is non-empty, i.e. has evidence, while the should be no evidence for the latter.

Predicates with several inputs are called relations. Using currying we can represent relations without using products ($_\times_$) so for example

$$_\leqslant_ : \mathbb{N} \to \mathbb{N} \to \mathsf{prop}$$

[1] I would prefer to use $_=_$ for equality and but this would use with agda using it for definitional equality (which we cannot talk about from within Type Theory).

is the less or equal relation. This is not limited to binary relations, an example for a ternary once is congruence modulo $_\equiv_\,\mathsf{mod}\,_$ e.g. $3 \equiv 7 \,\mathsf{mod}\, 2 : \mathsf{prop}$ expresses the (true) proposition that 3 and 7 have the same remainder when divided by 2.

To avoid repetition we just assume as given $\mathsf{A} : \mathsf{Set}$ predicates $\mathsf{P}\ \mathsf{Q} : \mathsf{A} \to \mathsf{prop}$ and a proposition $\mathsf{R} : \mathsf{prop}$.

We need to define the following operations:

$\forall[\ \mathsf{x} \in \mathbf{A}\ ]\ \mathbf{P\ x}$ For all x:A , P x holds.

$\exists[\ \mathsf{x} \in \mathbf{A}\ ]\ \mathbf{P\ x}$ There is an $\mathsf{x} : \mathsf{A}$, such that P x holds.

So for example $\forall[\ \mathsf{x} \in \mathbb{N}\ ]\ \mathsf{Even\ x} \vee \mathsf{Odd\ x}$ expresses that all natural are even or odd, and $\exists[\ \mathsf{x} \in \mathbb{N}\ ]\ \mathsf{Even\ x}$ expresses that there is an even natural number.

What is evidence for $\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$? It is a function that assigns to every $\mathsf{x} : \mathsf{A}$ evidence for P x. Hence we use the dependent function type and define

$$\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x} = (\mathsf{x} : \mathsf{A}) \to \mathsf{P\ x}$$

What is evidence for $\exists \mathsf{P\ x} \in \mathsf{A}$, P x? It is a pair of an element $\mathsf{a} : \mathsf{A}$ and evidence for P a. Hence we use a dependent pair type or Σ-type to interpret exists:

$$\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x} = \Sigma[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$$

We can use function application to instantiate a universally quantified assumption. That is given $\mathsf{h} : \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ and $\mathsf{a} : \mathsf{A}$ we obtain $\mathsf{f\ a} : \mathsf{P\ a}$. To prove a universally quantified proposition $\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ we assume as given an arbitrary $\mathsf{x} : \mathsf{A}$ and prove $\mathsf{p} : \mathsf{P\ x}$ and now $\lambda\, \mathsf{x} \to \mathsf{p} : \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$.

To prove an existential statement $\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ we need to find a witness $\mathsf{a} : \mathsf{A}$ and show that the witness satisfies the property $\mathsf{p} : \mathsf{P\ a}$ then $(\mathsf{a}\ ,\ \mathsf{p}) : \exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$. On the other hand we can show the elimination principle for existential quantification: that is if we assume $\mathsf{h} : \exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ and we can show that a given any $\mathsf{x} : \mathsf{A}$ which that satisfies $\mathsf{p\ x}$ we can prove a proposition R then we know R. This can be expressed as follows:

$$\mathsf{ex\text{-}e} : (\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}) \to (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ (\mathsf{P\ x} \Rightarrow \mathsf{R})) \to \mathsf{R}$$

This is easy to prove using pattern matching:

$$\mathsf{ex\text{-}e}\ (\mathsf{a}\ ,\ \mathsf{p})\ \mathsf{f} = \mathsf{f\ a\ p}$$

I noticed that students often struggle with this principle because it is a bit hard to verbalize. Here is my attempt using a concrete example" I use A to be the students in my class, I use P x to mean that x is clever and R means that I m happy. Now we know it is true that if any student is clever then I am happy this corresponds to $\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x} \to \mathsf{R}$.

We can now complete the table showing the propositions as types explanation for all logical connectives:

latin	*english*	*logic*	*types / definitions*
Conjunction	and	$P \wedge Q$	$P \times Q$
Disjunction	or	$P \vee Q$	$P \uplus Q$
Implication	if-then	$P \Rightarrow Q$	$P \to Q$
Verum	true	True	$\top$
Falsum	false	False	$\bot$
Negation	not	$\neg P$	$P \Rightarrow \mathsf{False}$
Equivalence	if-and-only-if	$P \Leftrightarrow Q$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
Universal quantification	for all	$\forall[\, x \in A \,]\, P\, x$	$(x : A) \to P\, x$
Existential quantification	exists	$\exists[\, x \in A \,]\, P\, x$	$\Sigma[\, x \in A \,]\, P\, x$

Abstract explanations like this are either obvious or incomprehensible. Hence the best is to look at some examples which is what we are going to do in the rest of this chapter.

6.2 Some Tautologies

To illustrate what we can do with our definition of quantifiers, we shall prove some tautologies, that is statements in predicate logic which are true in general.

The first tautology I want to prove is that $\forall$ commutes with $\wedge$, that is

$$\mathsf{taut}_1 \;:\; (\forall[\, x \in A \,]\, P\, x \wedge Q\, x) \Leftrightarrow (\forall[\, x \in A \,]\, P\, x) \wedge (\forall[\, x \in A \,]\, Q\, x)$$

I think it is alway helpful to think of a concrete example: so let's say A is the set of students in the lecture, $P\, x$ means that student x has yellow laces, and $Q\, x$ means that x wears a black shirt. Now it is clearly the same to say *Everybody has yellow laces and wears a black shirt* and *Everybody has yellow laces and everybody wears a black shirt*, and this has nothing to do with students, laces or shirt colors.

Ok two prove the statement we need to prove two implications:

$$\mathsf{proj}_1\ \mathsf{taut}_1 \;=\; ?$$
$$\mathsf{proj}_2\ \mathsf{taut}_1 \;=\; ?$$

Each of them is an implication, hence we may as well introduce parameters.

$$\mathsf{proj}_1\ \mathsf{taut}_1\ h \;=\; ?$$
$$\mathsf{proj}_2\ \mathsf{taut}_1\ h \;=\; ?$$

In the first case we have an assumption $h : \forall[\, x \in A \,]\, P\, x \wedge Q\, x)$, i.e. a dependent function and we need to prove $(\forall\, [\, x \in A \,]\, P\, x) \wedge (\forall\, [\, x \in A \,]\, Q\, x)$, that is a pair of functions $?\,, ?$. In the first component we need to prove $\forall[\, x \in A \,]\, P\, x)$ that is a function $\lambda\, x \to ?$ where $? : P\, x$. Now we can derive $h\, x : P\, x \wedge Q\, x$ and hence obtain $\mathsf{proj}_1\ (h\, x) : P\, x$. Similarly, we use $\lambda\, x \to \mathsf{proj}_2\ (h\, x)$ for the 2nd component.

$$\mathsf{proj}_1\ \mathsf{taut}_1\ \mathsf{h}\ =\ (\lambda\,\mathsf{x}\ \rightarrow\ \mathsf{proj}_1\ (\mathsf{h}\,\mathsf{x}))\ ,\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{proj}_2\ (\mathsf{h}\,\mathsf{x})$$

For the other direction

$$\mathsf{proj}_2\ \mathsf{taut}_1\ \mathsf{h}\ =\ ?$$

We have that $\mathsf{h}\ :\ (\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})\wedge(\forall\,[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})$ and we need to show $?\ :\ \forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}\wedge\mathsf{Q}\,\mathsf{x}$. We first split our assumption h into two

$$\mathsf{proj}_2\ \mathsf{taut}_1\ (\mathsf{hp}\ ,\ \mathsf{hq})\ =\ ?$$

With $\mathsf{hp}\ :\ \forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})$ and $\mathsf{hq}\ :\ \forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x}$. Now to prove $?\ :\ \forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}\wedge\mathsf{Q}\,\mathsf{x}$ we need to derive a function, hence we use $\lambda\,\mathsf{x}\ \rightarrow\ ?$ where $?\ :\ \mathsf{P}\,\mathsf{x}\wedge\mathsf{Q}\,\mathsf{x}$. Hence we need to produce a pair $?\ ,\ ?\ :\ \mathsf{P}\,\mathsf{x}\wedge\mathsf{Q}\,\mathsf{x}$ which we can do using $\mathsf{hp}\,\mathsf{x}\ :\ \mathsf{P}\,\mathsf{x}$ and $\mathsf{hq}\,\mathsf{x}\ :\ \mathsf{Q}\,\mathsf{x}$. Hence we have

$$\mathsf{proj}_2\ \mathsf{taut}_1\ (\mathsf{hp}\ ,\ \mathsf{hq})\ =\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{hp}\,\mathsf{x}\ ,\ \mathsf{hq}\,\mathsf{x}$$

What happens if we replace $_\wedge_$ by $_\vee_$ in the previous statement. Can we prove

$$(\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}\vee\mathsf{Q}\,\mathsf{x})\Leftrightarrow(\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})\vee(\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})$$

A quick analysis using common sense shows that this doesn't hold. It maybe that all students have yellow laces or black shirts but this doesn't mean that all students have yellow laces or all students have black shirts. I.e. the left to right implication doesn't hold. However, the right to left one is actually valid and can be proven:

$$\begin{array}{l}\mathsf{taut}_2\ :\ (\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})\vee(\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})\Rightarrow(\forall[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}\vee\mathsf{Q}\,\mathsf{x})\\ \mathsf{taut}_2\ (\mathsf{inj}_1\ \mathsf{p})\ =\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{inj}_1\ (\mathsf{p}\,\mathsf{x})\\ \mathsf{taut}_2\ (\mathsf{inj}_2\ \mathsf{q})\ =\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{inj}_2\ (\mathsf{q}\,\mathsf{x})\end{array}$$

Ok, what happens if we start agan with the first tautology but this time we replace $\forall\mathsf{P}$ with $\exists\mathsf{P}$? Does exists commute with and?

$$(\exists[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}\wedge\mathsf{Q}\,\mathsf{x})\Leftrightarrow(\exists[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})\wedge(\exists[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})$$

Again common sense shows that this doesn't hold, it may well be that there is a student who has yellow laces and another one with a black shirt but this doesn't mean that there is one student with yellow laces and a black shirt. This time the implication from right to left fails but we can prove the one from left to right:

$$\begin{array}{l}\mathsf{taut}_3\ :\ (\exists[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}\wedge\mathsf{Q}\,\mathsf{x})\Rightarrow(\exists[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})\wedge(\exists[\,\mathsf{x}\in\mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})\\ \mathsf{taut}_3\ (\mathsf{a}\ ,\ (\mathsf{p}\ ,\ \mathsf{q}))\ =\ (\mathsf{a}\ ,\ \mathsf{p})\ ,\ (\mathsf{a}\ ,\ \mathsf{q})\end{array}$$

The code shows clearly why we can't turn it around. We use the same a twice but this won't work in the other direction when both witnesses may be different.

What happens if we do both replacements at the same time. Can we prove:

$$(\exists[\, x \in A \,]\; P\, x \vee Q\, x) \Leftrightarrow (\exists[\, x \in A \,]\; P\, x) \vee (\exists[\, x \in A \,]\; Q\, x)$$

And indeed we can:

$$\begin{aligned}
&\mathsf{taut}_4 \;:\; (\exists[\, x \in A \,]\; P\, x \vee Q\, x) \Leftrightarrow (\exists[\, x \in A \,]\; P\, x) \vee (\exists[\, x \in A \,]\; Q\, x)\\
&\mathsf{proj}_1\ \mathsf{taut}_4\ (a\ ,\ \mathsf{inj}_1\ p) \;=\; \mathsf{inj}_1\ (a\ ,\ p)\\
&\mathsf{proj}_1\ \mathsf{taut}_4\ (a\ ,\ \mathsf{inj}_2\ q) \;=\; \mathsf{inj}_2\ (a\ ,\ q)\\
&\mathsf{proj}_2\ \mathsf{taut}_4\ (\mathsf{inj}_1\ (a\ ,\ p)) \;=\; a\ ,\ \mathsf{inj}_1\ p\\
&\mathsf{proj}_2\ \mathsf{taut}_4\ (\mathsf{inj}_2\ (a\ ,\ q)) \;=\; a\ ,\ \mathsf{inj}_2\ q
\end{aligned}$$

I leave it to you to go through the code line by line.

To summarize we can show:

$$\begin{aligned}
&(\forall[\, x \in A \,]\; P\, x \wedge Q\, x) \Leftrightarrow (\forall[\, x \in A \,]\; P\, x) \wedge (\forall[\, x \in A \,]\; Q\, x)\\
&(\exists[\, x \in A \,]\; P\, x \vee Q\, x) \Leftrightarrow (\exists[\, x \in A \,]\; P\, x) \vee (\exists[\, x \in A \,]\; Q\, x)
\end{aligned}$$

what is the pattern? $\forall$ is represented as a Π-type which is an infinite product and $\wedge$ is represented as $\times$ which is binary product. Now it should be no surprise that products commute with products. In the same vain $\exists$ was represented as a Σ-type which is an infinite sum and $\vee$ was represented using $\uplus$ which is a binary sum and again sums commute with sums.

Finally, here is a tautology which I find difficult to express in english:

$$\mathsf{taut}_5 \;:\; ((\exists[\, x \in A \,]\; P\, x) \Rightarrow R) \Leftrightarrow (\forall[\, x \in A \,]\; P\, x \Rightarrow R)$$

This time I use $P\ x$ to mean that x is clever and R means that the lecturer is happy. Now is it equivalent to say *If there is a student who is clever then the lecturer is happy* and *For all students it holds that if the student is clever then the lecturer is happy*? I think we have reached the limits of natural language here and we better just prove it:

$$\begin{aligned}
&\mathsf{proj}_1\ \mathsf{taut}_5\ h \;=\; \lambda\, a\ p \;\rightarrow\; h\ (a\ ,\ p)\\
&\mathsf{proj}_2\ \mathsf{taut}_5\ h \;=\; \lambda\, g \;\rightarrow\; h\ (\mathsf{proj}_1\ g)\ (\mathsf{proj}_2\ g)
\end{aligned}$$

Not only is the proof very simple, we have seen in before! Yes, this is curry and uncurry from section 2.5.2 And indeed what we do here is to curry a dependent pair and a function.

6.3 Equality

While we usually define relations that are specific to a set (e.g. $_\leqslant_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathsf{prop}$), there is one relation which is used for all sets: equality. We write

$$_\equiv_ \;:\; A \;\rightarrow\; A \;\rightarrow\; \mathsf{prop}$$

That is given a b : A we write a ≡ b : prop for the proposition that a is equal to b. So for example we write 1 + 2 ≡ 1 + 2 : prop which is actually inhabited, while 0 ≡ 1 : prop is empty.

Equality is defined as an inductive type with one constructor: reflexivity.

```
data _≡_ : A → A → prop where
  refl : {a : A} → a ≡ a
```

Indeed, this is not fundamentally different to the dependent types we have already seen e.g. Vec and Fin. However, the special thing about equality is that the same index is repeated in the codomain of refl.

We can understand the equality type as saying that identical things are equal. This sounds trivial but before we weren't able to talk about identity. The equality type reflects the notion of identity as a type and hence we can talk about it, i.e. use it in a proposition.

Now that we have defined equality, we should verify that basic properties hold. What are the basic properties of equality? First of all it should be an equivalence relation. A relation _~_ : A → A → prop is an equivalence relation, if it is reflexive, symmetric and transitive. We can express this using a record: [2]

```
record isEqRel (_~_ : A → A → prop) : prop where
  field
    reflexive  : {a : A}     → a ∼ a
    symmetric  : {a b : A}   → a ∼ b → b ∼ a
    transitive : {a b c : A} → a ∼ b → b ∼ c → a ∼ c
```

Ok let's show that _≡_ {A} : A → A → prop is an equivalence relation. The first property, reflexivity, is obvious because it is exactly what the constructor refl says. What about symmetry?

```
sym : {a b : A} → a ≡ b → b ≡ a


sym h = ?
```

We have assumed a b : A and h : a ≡ b and our goal is to prove b ≡ a. We are going to use the fact that the only proof of a ≡ b is refl and pattern match on h. We end up with

```
sym refl = ?
```

And now we only have one assumption a : A and our goal is to prove a ≡ a because once we matched h with refl we also need to identify a and b. Now this is easy to complete:

[2] Ok, at this point I am giving up on using the special ∀-syntax which translates into a dependent function type because it is more convenient to use {..} → here because we don't always want to fill in these parameters when using sym or any of the other combinators for equality.

```
sym refl = refl
```

To summarise: we can show that equality is symmetric because the only way we can prove a ≡ b is by refl and in this case a and b are identical and we can use refl to prove b ≡ a which is just a ≡ a.

Transitivity is also straightforward using basically the same idea:

```
trans : {a b c : A} → a ≡ b → b ≡ c → a ≡ c
```

Ok we start with:

```
trans p q = ?
```

where a b c : A, p : a ≡ b and q : b ≡ c and we need to prove a ≡ c. An obvious move is to pattern match one of the equality proofs. But which one? It turns out it doesn't matter. Let's just go with p.

```
trans refl q = ?
```

Matching p with refl has identified a and b. Hence we now only have a c : A and q : a ≡ c and our goal is still a ≡ c which just follow by assumption:

```
trans refl q = q
```

To summarise: we have shown that _≡_ {A} is an equivalence relation:

```
≡-is-eqRel : isEqRel (_≡_ {A})
≡-is-eqRel = record {
  reflexive  = refl;
  symmetric  = sym;
  transitive = trans}
```

There are other properties of equality which are useful in proofs. Every function preserves equality, we say equality is a congruence.

```
cong : (f : A → B) → {a b : A} → a ≡ b → f a ≡ f b
```

Again this is easy to prove by pattern matching a ≡ b:

```
cong f refl = refl
```

At this point we may wonder why we can't prove the inverse of cong

```
uncong : (f : A → B) → {a b : A} → f a ≡ f b → a ≡ b
```

However, we cannot pattern match f a ≡ f b. Even though we know that it has to be proven by refl we don't know that the argument would have to be f a, and indeed f may be a constant function and there is no reason why we should assume that both sides need to be f a. Actually uncong states that every function is injective which clearly isn't true.

We can only pattern match an equality assumption if at least one side is a variable which can be identified with the other side — which implies it cannot contain this variable as a subterm.

Another important property of equality is that we can replace equals by equals. That is if we have a goal $\mathsf{P\ b}$ and we know that $\mathsf{a \equiv b}$ then we can prove $\mathsf{P\ a}$ instead. This is expressed by the following function:

$$\mathsf{subst : (P : A \to prop) \to \{a\ b : A\} \to a \equiv b \to P\ a \to P\ b}$$

which again is easy to prove using pattern matching:

$$\mathsf{subst\ P\ refl\ p = p}$$

Indeed, all the principles we have proved before can be easily derived from subst. I'll leave this as an exercise.

6.3.1 Propositional vs definitional equality

Sometimes Type Theory is criticised because we have two equalities. And indeed, maybe you have noticed that I sometimes use _=_ to talk about equalities which follow directly from the definitions of functions and now we have introduced _≡_ as *propositional equality*. Isn't this rather confusing.

I don't think so. There is only one equality we can talk about *in* Type Theory and this is exactly propositional equality _≡_, which captures precisely what we mean by equality in Mathematics. However, when we talk *about* Type Theory then Type Theory is indeed a bit more involved then conventional logic, in that we also have a static equality, i.e. definitional equality. And we cannot talk about this in Type Theory as we cannot talk about the inhabitance _ : _. Both are called *judgements* and not propositions. They are static properties of the language, so to say rules of the grammar but not the content we are talking about.

Once we accept that $\mathsf{a : A}$ is a judgement we also need to be able to talk about a judgemental equality. For example let's say I define

$$\mathsf{n : \mathbb{N}}$$
$$\mathsf{n = 3}$$

and later I construct:

$$\mathsf{v : Vec\ \mathbb{N}\ n}$$
$$\mathsf{v = 1 :: 2 :: 3 :: []}$$

Then to see that the definition of v is well formed, i.e. is statically an acceptable text we need to know statically that $\mathsf{n = 3}$. Which means that equality is a part of the judgement that a certain text is grammatically correct.

Now when I present Type Theory, i.e. in a book about Type Theory or in a lecture I am often switching between two modes: I am talking about Type Theory and then I am talking in Type Theory. Hence I will sometimes mention

that certain expressions are definitionally equal using _=_ even though this is nothing I can talk about in Type Theory. And certainly if two expressions are definitionally equal then the values they denote are indeed propositionally equal just using refl.

6.4 Proving properties of addition

Ok, now we have the tools, let's do something with them. Actually let's build more tools. When reasoning about numbers we need to use algebra and what are the most basic laws of algebra? Here are two basic laws about addition on natural numbers: adding zero doesn't change anything. Why two laws? Since we haven't yet proven that $x + y = y + x$ we can add zero on the left $0 + n \equiv n$ or on the right $n + 0 \equiv n$.

Before we look at them let's recall the definition of addition from section 4.1.2:

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

Now, let's prove the laws. Starting with adding zero on the left:

```
lneutr+ : {n : ℕ} → 0 + n ≡ n
```

It turns out that this law is trivial, because this follows directly from our definition of _+_. Hence we just need to say

```
lneutr+ = refl
```

and we are done.

The other direction is less straightforward but more interesting.

```
rneutr+ : {n : ℕ} → n + 0 ≡ n
```

We cannot say rneutr+ = refl because the two sides are not identical or compute to the same expression. However, we can still prove it by using recursion. For this purpose first of all lets make the parameter visible

```
rneutr+ {n} = ?
```

Ok, we have assumed n : ℕ what can we do with it? Exactly, we can pattern match on it:

```
rneutr+ {zero} = ?
rneutr+ {suc n} = ?
```

The first goal is zero + zero ≡ zero and this is easy to prove again, because we know from the definition of _+_ that zero + n = n, hence

```
rneutr+ {zero} = refl
```

In the suc n case our goal is suc n + zero ≡ suc n. While here we can't reduce to identical expressions we can compute a bit exploiting the second line in the definition of _+_ which tells us that suc m + n ≡ suc (m + n). Applying this to the left side of our goal it simplifies to suc (n + zero) ≡ suc n. This suggest we could recursively prove n + zero ≡ n but we need to get rid of the pesky suc. But we can, we just need to use cong suc:

```
rneutr+ {suc n} = cong suc ?
```

And now indeed the goal is n + zero ≡ n and we can use a recursive call:

```
rneutr+ {suc n} = cong suc (rneutr+ {n})
```

What has actually happened? We just used recursion in a proof! And it meant that to prove that some properties holds for all natural numbers it is enough to prove it for zero and prove it for suc n using that it holds for n. This proof principle has a name, it is called *the principle of induction*. This is a nice consequence of propositions as types: induction is recursion.

Let's move on. Another very useful property of addition is that it doesn't matter how we bracket several additions, that is:

```
assoc+ : {l m n : ℕ} → (l + m) + n ≡ l + (m + n)
```

This property is called *associativity*. Again our strategy is to use pattern matching and recursion but this time we have three numbers to choose from: l, m and n. It turns out that the first choice l is the right one, but why? Looking at the definition of addition we notice that an expression of the form e_1 + e_2 computes if e_1 starts with a constructor. Hence it is a good idea to turn expressions on the left into constructors and l is best positioned to achieve this.

```
assoc+ {zero} = ?
assoc+ {suc l} = ?
```

The first goal is to show (zero + m) + n ≡ zero + (m + n) and using the definition of _+_ we can see

```
(zero + m) + n = m + n
zero + (m + n) = m + n
```

Hence we just need to say:

```
assoc+ {zero} = refl
```

Ok, what about the suc l case? We need to prove (suc l + m) + n ≡ suc l + (m + n). Again we can use the definition of _+_:

```
(suc l + m) + n = suc (l + m) + n
                = suc ((l + m) + n
(suc l) + (m + n) = suc (l + (m + n)
```

Hence all we need to do is to use cong suc and then we can use recursion:

```
assoc+ {suc l} = cong suc (assoc+ {l})
```

In algebra there is a name for an operation (like _+_) and an element which behaves like 0: it is called *a monoid*:

```
record IsMonoid
  (e : A) (_•_ : A → A → A) : prop where
  field
    lneutr : {x : A} → e • x ≡ x
    rneutr : {x : A} → x • e ≡ x
    assoc : {x y z : A} → (x • y) • z ≡ x • (y • z)
```

and we can summarize our results so far:

```
+isMonoid : IsMonoid zero _+_
+isMonoid = record {
  lneutr = lneutr+;
  rneutr = rneutr+;
  assoc = λ {x} {y} {z} → assoc+ {x} {y} {z}}
```

However, addition is one more useful property: it is commutative m + n ≡ n + m. How can we prove this?

```
comm+ : {m n : ℕ} → m + n ≡ n + m
```

Again the question is over which argument to do the induction (aka recursion). However,due to the symmetry of the equation it doesn't matter - hence let's use the first.

```
comm+ {zero} {n} = ?
comm+ {suc m} {n} = ?
```

We get two goals:

```
zero + n ≡ n + zero
suc m + n ≡ n + suc m
```

In both cases we can simplify the left hand side, but there is no simplification possible for the righthand sides, because they are stuck on n.

```
n ≡ n + zero
suc (m + n) ≡ n + suc m
```

The first goal can be handled with rneutr+ but it is not clear how to make progress with the 2nd. We need a way to move the successor form the 2nd argument to the front - while the definition only provides a way to move the

successor from inside the first argument. Hence we suspend this proof and first prove a lemma:

```
suc+ : {m n : ℕ} → suc (n + m) ≡ n + suc m
```

In this case it is clear that the induction should use n. The rest is straight forward:

```
suc+ {m} {zero} = refl
suc+ {m} {suc n} = cong suc (suc+ {m} {n})
```

Back to our original problem:

```
comm+ : {m n : ℕ} → m + n ≡ n + m
```

While as remarked the zero case can be handled with renutr+ there is a slight inconvenience that the equation is the wrong way around: rneutr proves n + zero ≡ n but we need n ≡ n + zero. This can be easily dealt with using sym:

```
comm+ {zero} {n} = sym rneutr+
```

For the second case we have to combine the lemma suc+ we have shown and the recursive call. That is to show

```
suc m + n ≡ n + suc m
```

We use

```
suc m + n = suc (m + n)        The definition of +
          = suc (n + m)        The induction hypothesis for m
          = n + suc m          The lemma suc+
```

Using trans and cong we can translate this into agda code

```
comm+ {suc m} {n} = trans (cong suc (comm+ {m} {n})) suc+
```

It is clear that this presentation of equational proofs will become very unreadable very quickly. The standard agda library defines some syntactic sugar which provides a nicer presentation which shows the intermediate goals and enables us to layout the derivation nicely but which actually reduces to the same proof.

```
comm+ {suc m} {n} = begin
  suc m + n
    ≡⟨ refl ⟩
  suc (m + n)
    ≡⟨ cong suc (comm+ {m} {n}) ⟩
  suc (n + m)
    ≡⟨ suc+ ⟩
```

```
  n + suc m
  ■
```

However, we still have to spend a lot of space for mathematically trivial steps. This is can be addressed using a solver which automatically computes solutions.

To summarise we have shown that _+_ forms a *commutative monoid*:

```
record IsCommMonoid
  (e : A) (_•_ : A → A → A) : prop where
  field
    isMonoid : IsMonoid e _•_
    isComm : {x y : A} → x • y ≡ y • x


+isCommMonoid : IsCommMonoid zero _+_
+isCommMonoid = record {
  isMonoid = +isMonoid;
  isComm = λ {x} {y} → comm+ {x} {y}}
```

6.5 Extending the negative translation

We have introduced the negative translation in section 3.4 as an alternative to the boolean interpretation of classical logic. Now moving to predicate logic we cannot actually use the boolean interpretation because given P : A → Bool how can we determine a boolean to ∀[x : A] P x. In particular it may be that A is infinite, e.g. A = ℕ.

However, the negative translation which translates any proposition P so that RAA P = ¬ (¬ P) → P holds still works as we will see. First of all we need to show that RAA holds for ∀[x ∈ A] P x if for all x : A we have that RAA (P x) holds.

```
all-neg : ((x : A) → RAA (P x)) → RAA (∀[ x ∈ A ] P x)
all-neg raa h x = raa x (λ g → h (λ k → g (k x)))
```

What about ∃[x ∈ A] P x? Like P ∨ Q this is a positive statement, i.e. it contains information, namely the choice of witness. Hence as for _∨_ we translate ∃ using the classical equivalence that ∃[x ∈ A] P x is equivalent to saying that it is not the case the property isn't false for all x : A.

```
Ex-c : (A : Set) (P : A → prop) → prop
Ex-c A P = ¬ (∀[ x ∈ A ] ¬ (P x))
```

I am going to write $\exists^c[\, x \in A \,]\, P$ for classical existence. As for $_ \vee^c _$. It should be clear that we have:

```
neg-ex-c : RAA (∃ᶜ[ x ∈ A ] P)
```

because we have already shown that all negated propositions satisfy RAA.

We need to show that the basic principles for existentials are provable, that is we need to show:

$$\mathrm{ex}^c\text{−i} : (a : A) \to P\ a \to \exists c[\, x \in A \,]\ P\ x$$
$$\mathrm{ex}^c\text{−e} : \mathsf{RAA}\ R \to ((x : A) \to P\ x \to R) \to \exists^c[\, x \in A \,]\ P\ x \to R$$

What about equality? In turns out that most equalities we encounter are already negative, for example $_ \equiv _\ \{\mathbb{N}\}$ satisfies RAA because it is decidable as we a re going to show in the next chapter. However, we don't need to assume this but we can define classical equality as the double negation of the usual equality:

$$_ \equiv^c _ : A \to A \to \mathsf{prop}$$
$$a \equiv^c b = \neg\,(\neg\,(a \equiv b))$$

and as before we have:

$$\mathsf{ne\text{-}eq\text{-}c} : \{a\ b : A\} \to \mathsf{RAA}\,(a \equiv^c b)$$

We can derive some the basic principles associated with equality for $_ \equiv^c _$:

$$\mathrm{refl}^c : \{a : A\} \to a \equiv\text{-c}\ a$$
$$\mathrm{subst}^c : (P : A \to \mathsf{prop}) \to (\{x : A\} \to \mathsf{RAA}\,(P\ x)) \to \{a\ b : A\} \to a \equiv\text{-c}\ b \to P\ a \to P\ b$$

I am leaving the verification of these properties as an exercise.

Hence it seems that classical logic is just the logic of the negative fragment of intuitionistic logic where we use negative, i.e. classical versions, of disjunction and existence. Almost, but there is another important aspect of classical Mathematics: the axiom of choice which we are going to discuss later. The negative translation is not going to help us here because the negative translation of the axiom of choice is not provable intuitionistically.

6.6 History

An early form of predicate logic was introduced by Frege in the late 19th century [Fre79]. His basic laws allowed quantification over all sets and hence was inconsistent as Russell pointed out. However, his attempt to make the rules of reasoning precise lead to the formal system of predicate logic. This is also often called first order logic, because we don't allow quantification over propositions themselves.

While Frege tried to capture set theory and logic at the same time, later it became clear that it is a good idea to separate the system of reasoning and the particular basic assumptions, i.e. axioms. Hence for example Peano Arithmetic and Zermelo-Fraenkel set theory can be formulated in the same framework.

It was Hilbert and Ackermann who clearly formulated first order logic in the 1930ies, (there is a recent english translation: [HA99]).

Gödel proved a completeness theorem for classical predicate logic which shows that the formal system is complete for the classical notion of a model, i.e. if a proposition holds in all models it is also provable [Kur29]. However, this is done for classical logic in a classical metatheory and in this form it is not relevant in our context.

Unlike set theory type theory doesn't use the framework of predicate logic it stands on its own. However, many of the idioms of predicate logic are used and we can understand them via their translation into predicate logic as we have shown. However, the interpretation of disjunction $\vee$ and existence $\exists$ we have represented are debatable since the types don't really behave like propositions. We will return to this issue later and present an alternative view what is a proposition.

6.7 Exercises

1. We are playing logic poker again. But this time with predicate logic!

   Consider the following propositions:

$$
\begin{array}{l}
\mathsf{P01} = \{\mathsf{A\ B} : \mathsf{Set}\}\,\{\mathsf{R} : \mathsf{A} \to \mathsf{B} \to \mathsf{prop}\} \to \\
\quad ((\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\exists[\,\mathsf{y} \in \mathsf{B}\,]\,\mathsf{R\ x\ y}) \to \\
\quad (\exists[\,\mathsf{y} \in \mathsf{B}\,]\,\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{R\ x\ y}) \\
\mathsf{P02} = \{\mathsf{A\ B} : \mathsf{Set}\}\,\{\mathsf{R} : \mathsf{A} \to \mathsf{B} \to \mathsf{prop}\} \to \\
\quad (\exists[\,\mathsf{y} \in \mathsf{B}\,]\,(\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{R\ x\ y})) \\
\quad \to (\forall[\,\mathsf{x} \in \mathsf{A}\,]\,(\exists[\,\mathsf{y} \in \mathsf{B}\,]\,\mathsf{R\ x\ y})) \\
\mathsf{P03} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad \neg\,(\exists[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x}) \to \forall[\,\mathsf{x} \in \mathsf{A}\,]\,\neg\,(\mathsf{P\ x}) \\
\mathsf{P04} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\neg\,(\mathsf{P\ x})) \to \neg\,(\exists[\,\mathsf{x} \in \ \mathsf{A}\,]\,\mathsf{P\ x}) \\
\mathsf{P05} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\neg\,(\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x})) \to \exists[\,\mathsf{x} \in \mathsf{A}\,]\,\neg\,(\mathsf{P\ x}) \\
\mathsf{P06} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\exists[\,\mathsf{x} \in \mathsf{A}\,]\,\neg\,(\mathsf{P\ x})) \to (\neg\,(\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x})) \\
\mathsf{P07} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\neg\,(\neg\,(\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x}))) \to \forall[\,\mathsf{x} \in \mathsf{A}\,]\,\neg\,(\neg\,(\mathsf{P\ x})) \\
\mathsf{P08} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\neg\,(\neg\,(\mathsf{P\ x}))) \to (\neg\,(\neg\,(\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x}))) \\
\mathsf{P09} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\exists[\,\mathsf{x} \in \mathsf{A}\,]\,\top) \to (\exists[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x}) \to \forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x} \\
\mathsf{P10} = \{\mathsf{A} : \mathsf{Set}\}\,\{\mathsf{P} : \mathsf{A} \to \mathsf{prop}\} \to \\
\quad (\exists[\,\mathsf{x} \in \mathsf{A}\,]\,\top) \to (\exists[\,\mathsf{y} \in \mathsf{A}\,]\,(\mathsf{P\ y} \to \forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P\ x}))
\end{array}
$$

   Here is a quick summary of the rules of logic poker:

   (a) You can prove it directly (intuitionistically) - just provide pi : Pi.

(b) You cannot prove it with intuitionistic logic but you can prove it with classical logic. In this case prove pi : CLASS → Pi where

```
CLASS = {P : prop} → RAA P
```

(c) It isn't even true in classical logic. In this case prove pi : Pi → ⊥. Note that you will have to come up with predicates and relations to refute the proposition.

2. Show that all the properties of equality we have shown are derivable from subst. That is prove

```
sym-s : {a b : A} → a ≡ b → b ≡ a
trans-s : {a b c : A} → a ≡ b → b ≡ c → a ≡ c
cong-s : (f : A → B) → {a b : A} → a ≡ b → f a ≡ f b
```

using only:

```
subst : (P : A → prop) → {a b : A} → a ≡ b → P a → P b
```

instead of pattern matching.

3. Verify the laws of boolean algebra we discussed in section 3.1.

```
com-& : {x y : Bool} → x & y ≡ y & x
distr-&| : {x y z : Bool} → x & (y | z) ≡ x & y | x & z
dm1 : {x y : Bool} → ! (x | y) ≡ ! x & ! y
dm2 : {x y : Bool} → ! (x & y) ≡ ! x | ! y
```

4. Show that the natural numbers ℕ with addition and multiplication form a commutative semiring. We have already shown that addition forms a commutative monoid, hence it is left to show:

```
*isCommMonoid : IsCommMonoid 1 _*_
null-l : {n : ℕ} → 0 * n ≡ 0
distr-l : {l m n : ℕ} → (l + m) * n ≡ l * n + m * n
```

Note that I have mitted the symmetric forms of the distributivity laws:

```
null-r : {n : ℕ} → n * 0 ≡ 0
distr-r : {l m n : ℕ} → l * (m + n) ≡ l * m + l * n
```

since they follow from commutativity but it may be useful to prove them before establishing commutativity.

5. Continuing from the previous question: show that the natural numbers form an exponential semi-ring, that is prove the laws of high school algebra:

```
exp-plus : {l m n : ℕ} → l ↑ (m + n) ≡ l ↑ m * l ↑ n
exp-times : {l m n : ℕ} → l ↑ (m * n) ≡ (l ↑ m) ↑ n
```

6. Prove the missing lemmas about the negative translation from section 6.5.

```
all-neg : ((x : A) → RAA (P x)) → RAA (∀[ x ∈ A ] P x)
ex^c−i : (a : A) → P a → ∃^c[ x ∈ A ] P x
ex^c−e : RAA R → ((x : A) → P x → R) → ∃^c[ x ∈ A ] P x → R
refl^c : {a : A} → a ≡^c a
subst^c : (P : A → prop) → ({x : A} → RAA (P x))
  → {a b : A} → a ≡^c b → P a → P b
```

Chapter 7

Programs and proofs

While in the last chapter I have spent some effort to introduce a nice syntax for predicate logic resembling the commonly used symbols actually in type theory we usually stick to Π-types and Σ-types where the latter are usually replaced by more generic record types. One of the reasons for this is that the proofs and programs or logic and types are not always clearly separable.

Let's for example consider a program that sorts natural numbers. It should have the type

```
sort : List ℕ → List ℕ
```

Let's say we want to be more specific and express in the type that the program does actually sorts its input. Let's assume we have defined a predicated Sorted : List ℕ → prop which expressed that a list is sorted and a relation _~_ : List ℕ → List ℕ → prop where l ∼ l' means that both lists have the same elements but in a different order (we say one is a permutation of the other). Now we can define a more refined output type as a record:

```
record SortSpec (inp : List ℕ) : Set where
  out : List ℕ
  sorted : Sorted out
  perm : inp ∼ out
```

and our sorting program now has the type

```
sort : (inp : List ℕ) → SortSpec inp
```

By the way this shows how we can also use Σ-types or record types to express what is called *comprehension* in set theory, that is:

```
SortSpec out = {out : List ℕ | Sorted out ∧ inp ∼ out}
```

However, the point I want to make is that we think or the list out as data and the other two components sorted and perm as proofs of propositions. However, if

we want to use the sort-function to externally sort a sequence maybe using some robot then it is perm which tells us how the items should be rearranged! That is what has been a proof becomes data and we should have said: $_\tilde{}_ : \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N} \to \mathsf{Set}$. Hence the distinction between Set and prop we have made was a bit artificial and maybe we should have stuck to Set anyway.

7.1 Decidability of equality

We are going to look at another program that combines logic and types. We can define a function that decides wether two natural numbers are equal:

```
eqb : ℕ → ℕ → Bool

eqb zero    zero    = true
eqb zero    (suc n) = false
eqb (suc m) zero    = false
eqb (suc m) (suc n) = eqb m n
```

While this is a perfectly fine program we have to read its code to see that it is doing the right thing. Ok, this is maybe easy in this case but wouldn't it be nicer if the type says what the program is doing (as in our improved version of sort)? And not only nicer but also potentially more useful because this sort of information may come in handy when we want to prove something about some program that uses eqb. That is we would like to express that eqb decides $_\equiv_\ \{\mathbb{N}\}$.

Here we can exploit an advantage of working in a logic where $\mathsf{P} \vee \neg\,\mathsf{P}$ is not a tautology because we can use $\mathsf{P} \vee \neg\,\mathsf{P}$ to express that P has been decided, i.e. we know wether it is true or false. Since this construction is used frequently, there is an explicit definition as a sum:

```
data Dec (P : Set) : Set where
  yes : P → Dec P
  no  : ¬ P → Dec P
```

That is Dec P is equivalent to $\mathsf{P} \vee \neg\,\mathsf{P}$ and expresses that P has been decided. Examples for propositions which can be decided easily are True and False:

```
decTrue : Dec True
decTrue = yes tt

decFalse : Dec False
decFalse = no case⊥
```

More interesting cases of decided propositions include *Fermat's last theorem* but not for example P=NP, i.e. wether the class of problems solvable in deterministic and non-deterministic polynomial time are the same. While most people think that the answer is no, nobody has got a proof so we cannot prove Dec P=NP.

There is some hope that one day P=NP will be decided some day and we may wonder wether there are intrinsically undecidable propositions? The answer is no , because we proved

```
nntnd : {P : prop} → ¬ (¬ (P ∨ ¬ P))
```

in section 3.3.

We can use the notion of decided to express what it means for a predicate `P : A → prop` to be *decidable*, it means that we can decide each instance:

```
(x : A) → Dec (P x)
```

It is not worth to define a predicate decidable because it would only work for predicates but not for relations and our first application is exactly the deciability of `_≡_ {ℕ}`:

```
_≡ℕ?_ : (m n : ℕ) → Dec (m ≡ n)
```

The first case is easy:

```
zero ≡ℕ? zero = yes refl
```

We say yes and we give the reason. In the next case we want to say no:

```
zero ≡ℕ? suc n = no ?
```

We need to show `¬ (zero ≡ suc n)`. It turns out that this can be done just by local pattern matching because agda *knows* that different constructors cannot be equal:

```
zero ≡ℕ? suc n = no λ ()
```

The nest case is symmetric and we use the same trick:

```
suc m ≡ℕ? zero = no λ ()
```

Finally if both inputs start with a succesor we have to inveestigate the result of the recursive call:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc n) | yes p = ?
(suc m ≡ℕ? suc n) | no np = ?
```

In the yes case we can just match x:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n)  | no np    = ?
```

The negative case is a bit more interesting. Certainly if `¬ (m ≡ n)` then also `¬ (suc m ≡ suc n)`:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n)  | no np   = no (λ p → np ?)
```

At this point we need to show m ≡ n and we know p : suc m ≡ suc n hence we need to know that suc is injective. This is easy to prove using (local) pattern matching:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n)  | no np   = no (λ {refl → np refl})
```

If we look at the whole function definition again:

```
_≡ℕ?_ : (m n : ℕ) → Dec (m ≡ n)
zero ≡ℕ? zero   = yes refl
zero ≡ℕ? suc n  = no λ ()
suc m ≡ℕ? zero  = no λ ()
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n)  | no np   = no (λ {refl → np refl})
```

we see that it has exactly the same structure as eqb but instead of returning a boolean we return an element of Dec (m ≡ n) which is basically a boolean with the reason that we are allowed to return it.

Hence decidability is an example where we can refine a simply typed program with a non very expressive type into a type which tells us exactly what the function is doing. Again we see that the border between reasoning and programming is rather fluid.

7.1.1 Church's thesis

While we cannot have a proposition which cannot be decided, it is possible to have a predicate or a relation which is not decidable. Indeed the Halting problem provides an example, which we can view as a predicate

```
Halts : ℕ → prop
```

assuming that we use some encoding of programs (e.g. Turing machines) as natural numbers. Can we show:

```
undecHalt : ((n : ℕ) → Dec (Halts n)) → ⊥
```

The answer is no: Type Theory still allows us to assume the excluded middle for all propositions

```
tnd : {P : prop} → Dec P
```

and using this there is a trivial proof of $(\mathsf{n} : \mathbb{N}) \to \mathsf{Dec}\ (\mathsf{Halts}\ \mathsf{n})$ and hence it is inconsistent to have both undecHalt and tnd. We can assume a principle which says that all functions on the natural numbers are computable (this is called *Church's thesis*) and using this we can prove undecHalt. Hence, clearly Church's thesis is inconsistent with classical logic, it is anti-classical.

7.1.2 Uniqueness of equality proofs

In Type Theory we can ask questions that don't make sense in conventional reasoning. For example we may wonder whether there is at most one way to prove that two numbers are equal

$$\mathsf{uip}\mathbb{N} : \{\mathsf{m}\ \mathsf{n} : \mathbb{N}\}\ (\mathsf{p}\ \mathsf{q} : \mathsf{m} \equiv \mathsf{n}) \to \mathsf{p} \equiv \mathsf{q}$$

This is interesting because it shows that $_\equiv_\ \{\mathbb{N}\}$ is *propositional*. Ok, previously we have said that $\mathsf{prop} = \mathsf{Set}$ and I have argued that the distinction is not very precise. However, we may say that a type is a proposition if it contains no other information but that it is inhabited. That is all elements are equal.

It seems that we can prove uip$\mathbb{N}$ in a generic way using pattern matching. We match the first element:

$$\mathsf{uip}\mathbb{N}\ \mathsf{refl}\ \mathsf{q} = ?$$

Now we have $\mathsf{q} : \mathsf{m} \equiv \mathsf{m}$. Can we pattern match this? Before we were instantiating at least one of the arguments, but this time they are already equal. One may think this doesn't matter and:

$$\mathsf{uip}\mathbb{N}\ \mathsf{refl}\ \mathsf{refl} = ?$$

Now we just have to show $\mathsf{refl} \equiv \mathsf{ref}$ which is easy:

$$\mathsf{uip}\mathbb{N}\ \mathsf{refl}\ \mathsf{refl} = \mathsf{refl}$$

However, the last pattern matching isn't completely uncontroversial and in particular it is inconsistent with Homotopy Type Theory which we will be considering later. We can rule out this proof by enabling the option `--without-K` which will reject the pattern matching on q.

However, it turns out that in this particular case we can still prove uip$\mathbb{N}$. This is an instance of Hedberg's theorem which tells us that uip holds for any type which has a decidable equality.

7.2 Inductively defined relations: less or equal

The equality relation has been defined as an inductive type in a particular simple way: we only needed one constructor. Inductively defined relations and predicates are quite useful in general and we shall use the less or equal relation on natural numbers $_\leqslant_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Set}$ as an example.

There are various ways to define $_\leqslant_$, e.g. we can use addition $\mathsf{m} \leqslant \mathsf{n} = \exists[\ \mathsf{k} \in \mathbb{N}\ ]\ \mathsf{k} + \mathsf{m} \equiv \mathsf{n}$. However, here we shall go another way and define the relation as given by some deduction rules:

$$\frac{}{0 \leqslant \mathsf{n}}\ \mathsf{le0} \qquad \frac{\mathsf{m} \leqslant \mathsf{n}}{\mathsf{suc\ m} \leqslant \mathsf{suc\ n}}\ \mathsf{leS}$$

We can use these rules to derive for example that $1 \leqslant 3$:

$$\frac{\dfrac{}{0 \leqslant 2}\ \mathsf{le0}}{1 \leqslant 3}\ \mathsf{leS}$$

On the other hand there cannot be a derivation of $3 \leqslant 1$. The only rule directly applicable is leS and after this we are stuck because no rule applies:

$$\frac{\dfrac{\text{impossible}}{2 \leqslant 0}}{3 \leqslant 1}\ \mathsf{leS}$$

We can turn this idea into an inductive definition with $\mathsf{le0}$ and leS as constructors:

```
data _⩽_ : ℕ → ℕ → Set where
  le0 : {n : ℕ} → zero ⩽ n
  leS : {m n : ℕ} → m ⩽ n → suc m ⩽ suc n
```

The derivation of $1 \leqslant 3$ can be expressed:

```
1⩽3 : 1 ⩽ 3
1⩽3 = leS le0
```

Using pattern matching we can show $\neg\,(3 \leqslant 1)$:

```
¬3⩽1 : ¬ (3 ⩽ 1)
¬3⩽1 x = ?
```

We pattern match $\mathsf{x} : 3 \leqslant 1$, only leS is applicable:

```
¬3⩽1 : ¬ (3 ⩽ 1)
¬3⩽1 (leS x) = ?
```

Now $\mathsf{x} : 2 \leqslant 0$ and no constructor is applicable, hence we get the empty pattern:

```
¬3⩽1 : ¬ (3 ⩽ 1)
¬3⩽1 (leS ())
```

and we are done.

Let's establish some basic properties of $_\leqslant_$, it is reflexive, transitive and antisymmetric. A relation with these properties is called a *partial order*. We

have already seen reflexivity and transitivity from equality which was an equivalence relation. Antisymmetry is not exactly the opposite of symmetry but it says that $\mathsf{m} \leqslant \mathsf{n}$ and $\mathsf{m} \leqslant \mathsf{n}$ can only both hold, if $\mathsf{m} \equiv \mathsf{n}$.

We start with reflexivity which we can prove by induction over natural numbers:

```
refl⩽ : {n : ℕ} → n ⩽ n
refl⩽ {zero} = le0
refl⩽ {suc n} = leS (refl⩽ {n})
```

Transitivity is maybe more interesting.

```
trans⩽ : {l m n : ℕ} → l ⩽ m → m ⩽ n → l ⩽ n


trans⩽ p q = ?
```

Here we need to do an induction over derivations, which just corresponds to pattern matching

```
trans⩽ le0 q = ?
trans⩽ (leS p) q = ?
```

The first case $\mathsf{zero} \leqslant \mathsf{n}$ is easily dealt with:

```
trans⩽ le0 q = le0
```

While the 2nd needs a further induction / pattern matching over the 2nd derivation $\mathsf{q} : \mathsf{suc\ m} \leqslant \mathsf{n}$ but luckily onle one rule applies:

```
trans⩽ (leS p) (leS q) = ?
```

And now to show $\mathsf{suc\ l} \leqslant \mathsf{suc\ n}$ we only need to use leS and recursion, aka appeal to the induction hypothesis:

```
trans⩽ (leS p) (leS q) = leS (trans⩽ p q)
```

The proof of antisymmetry:

```
antisym : {m n : ℕ} → m ⩽ n → n ⩽ m → m ≡ n
```

works in a similar way: we do induction over both derivations and nicely our first choice limits the 2nd:

```
antisym le0 le0 = refl
antisym (leS p) (leS q) = cong suc (antisym p q)
```

In the 2nd line we reduce showing $\mathsf{suc\ m} \equiv \mathsf{suc\ n}$ to $\mathsf{m} \equiv \mathsf{n}$ using cong suc and this we can prove recursively.

Indeed, _⩽_ is decidable, which is a refinement of the boolean function:

```
leb : ℕ → ℕ → Bool
leb zero n = true
leb (suc m) zero = false
leb (suc m) (suc n) = leb m n
```

This is similar to our construction of _≡?_:

```
_⩽?_ : (m n : ℕ) → Dec (m ⩽ n)
zero ⩽? n = yes le0
suc m ⩽? zero = no (λ ())
suc m ⩽? suc n with m ⩽? n
(suc m ⩽? suc n) | yes x = yes (leS x)
(suc m ⩽? suc n) | no x = no (λ {(leS p) → x p})
```

In the last case we are exploiting that leS is invertible, i.e. that suc m ⩽ suc n implies m ⩽ n. This follows by pattern matching since leS is the only constructor that can prove suc m ⩽ suc n.

Finally, we may ask wether m ⩽ n is a proposition in the sense explained in the last section. Indeed, it should be clear that it is, since we never had a choice between derivations. However, this time there are no issues and the construction is permitted even without K.

```
isProp⩽ : {m n : ℕ} (p q : m ⩽ n) → p ≡ q
isProp⩽ le0 le0 = refl
isProp⩽ (leS p) (leS q) = cong leS (isProp⩽ p q)
```

In the last line we actually use cong on a proof rule: we reduce showing leS p ≡ leS q to p ≡ q which can be done by recursion.

7.3 An alternative definition of less or equal

The definition of _⩽_ in the previous section is not the only possibility. Here is an alternative:

```
data _⩽'_ : ℕ → ℕ → Set where
  le'refl : {m : ℕ} → m ⩽' m
  le'S : {m n : ℕ} → m ⩽' n → m ⩽' suc n
```

We use le'refl to show that n ⩽ n and then we can raise the second argument using le'S. So for example:

```
1 ⩽'3 : 1 ⩽' 3
1 ⩽'3 = le'S (le'S le'refl)
```

And using pattern matching we can still show negative results:

```
¬3⩽'1 : ¬ (3 ⩽' 1)
¬3⩽'1 (le'S ())
```

I hope it is intuitively clear that $_\leqslant_$ and $_\leqslant'_$ define the same relation. Let's show this in agda:

```
⩽⇔⩽' : {m n : ℕ} → m ⩽ n ⇔ m ⩽' n
```

We have to define functions in both directions, mapping one sort of derivations into the other and vice versa.

```
proj₁ ⩽⇔⩽' p = ?
proj₂ ⩽⇔⩽' p = ?
```

Let's look at the first line. We analyse the derivation of $m \leqslant n$

```
proj₁ ⩽⇔⩽' le0     = ?
proj₁ ⩽⇔⩽' (leS p) = ?
```

We observe that we need to prove that $_\leqslant'_$ is closed under the rules defining $_\leqslant_$. That is we need to establish

```
le0⩽' : {n : ℕ} → zero ⩽' n
suc'⩽' : {m n : ℕ} → m ⩽' n → suc m ⩽' suc n
```

For the first one we use recursion over $n : \mathbb{N}$:

```
le0⩽' {zero}  = le'refl
le0⩽' {suc n} = le'S (le0⩽' {n})
```

and for the second we do an induction over the derivation of $m \leqslant' n$:

```
suc'⩽' le'refl   = le'refl
suc'⩽' (le'S p) = le'S (suc'⩽' p)
```

Using these we can recursively substitute the rules used to derive $m \leqslant n$ to prove $m \leqslant' n$.

```
proj₁ ⩽⇔⩽' le0     = le0⩽'
proj₁ ⩽⇔⩽' (leS p) = suc'⩽' (proj₁ ⩽⇔⩽' p)
```

We apply the same strategy for the other direction. We now need to show that $_\leqslant_$ is closed under the rules defining $_\leqslant'_$. The first one we have already done, when we showed that $_\leqslant_$ is reflexive, but we still need to show the 2nd which we derive again by recursion over derivations:

```
leS⩽ : {m n : ℕ} → m ⩽ n → m ⩽ suc n
leS⩽ le0     = le0
leS⩽ (leS p) = leS (leS⩽ p)
```

Here is the complete proof:

```
⩽⇔⩽' : {m n : ℕ} → m ⩽ n ⇔ m ⩽' n
proj₁ ⩽⇔⩽' le0     = le0⩽'
proj₁ ⩽⇔⩽' (leS p) = suc'⩽' (proj₁ ⩽⇔⩽' p)
proj₂ ⩽⇔⩽' le'refl = refl⩽
proj₂ ⩽⇔⩽' (le'S p) = leS⩽ (proj₂ ⩽⇔⩽' p)
```

In the usual jargon we could say that _⩽_ is the least relation closed under the rules defining it and once we show that _⩽'_ is closed under the same rules we know that m ⩽ n implies m ⩽' n. However, this seems a bit ambitious, do we really need to think about the least relation among all relations? The reasoning we are using here is more concrete, we just mapping trees to trees. There is no need to think about the *least relation* ....

7.4 History

The idea that we can use dependent types to support a programming style where programs are intrinsically correct is quite recent. It goes along with the evolution of implementations such as the ALF system [AGNvS94] in Gothenburg, Conor McBride's OLEG [McB00a] which grew out of the LEGO system [LP92]. [1] OLEG later evolved to Epigram [McB04] which together with ALF provided much input into Agda.

I made the case for *progification* as a synthesis of programming and verification in an unpublished paper [Alt94]. This was also the base of my course at the ESSLI summer school in Prague in 1996 [Alt96] on *Integrated verification inType Theory* which also featured an introduction to ALF.

Conor McBride [2] is the most influential contributor to the this style of writing correct programs using dependent types (see his PhD thesis [McB00a]): he has explained this in many of his *gigs* which involved animations made with plastic slides in a time when we still used overhead projectors. Most famous is his talk *Winging It* [McB09] where he explains dependent types by an analogy with a proletarian revolution: the terms are the workers on the left of the colon whereas the types are the bourgeoisie on the right hand side of the colon who have no idea about work.

[1] Conor said that OLEGO was a rearrangemengt of LEGO which referred to the treatment of holes. However, it was also necessary to rearrange the name due tothreats by lawyers of the LEGO compnay now that the internet was growing beyond academia.

[2] For full disclosure: Conor was like me supervised by Rod Burstall in Edinburgh.



However, the established social oder is overcome and the terms march through the colon. Are the types now running away in panic?



No, they are actually happy about their new expressivity:



Jointly with James McKinna many of the ideas are explained in *The view*

from the left [MM04] and later in our joint and also our unpublished paper [AMM05] on *why dependent types matter?*.

7.5 Exercises

1. Show that equality for the type Expr as defined in section 4.3.2 is decidable.

   ```
   data Expr : Set where
     const : ℕ → Expr
     _[+]_ : Expr → Expr → Expr
     _[*]_ : Expr → Expr → Expr
   ```

   That is construct a program of the following type:

   ```
   _≡Expr?_ : (e f : Expr) → Dec (e ≡ f)
   ```

   You may use _≡ℕ?_.

2. Prove uniqueness of equality proofs for _≡_ {ℕ} without using K, that is with the flag `--without-K` enabled.

   ```
   uipℕ : {m n : ℕ} (p q : m ≡ n) → p ≡ q
   ```

   Remark: While this is a consequence of Hedberg's theorem which we will introduce later — it is also possible to prove it directly. For this purpose it may be helpful to define equality for the natural numbers inductively:

   ```
   data _~ℕ_ : ℕ → ℕ → Set where
     zero : zero ~ℕ zero
     suc : {m n : ℕ} → m ~ℕ n → suc m ~ℕ suc n
   ```

   and then showing that the _≡_ {ℕ} and _~ℕ_ are equivalent (Actually it is enough to establish a retraction, that is a function and with an inverse). It should be straightforward to show uniqueness for _~ℕ_.

3. A common definition of $\leqslant$ avoiding the use of an inductively defined family is

   ```
   _⩽+_ : ℕ → ℕ → prop
   m ⩽+ n = ∃[ k ∈ ℕ ] k + m ≡ n
   ```

   or equivalently one could also use:

   ```
   _+⩽_ : ℕ → ℕ → prop
   m +⩽ n = ∃[ k ∈ ℕ ] m + k ≡ n
   ```

Show that they are logically equivalent to $_\leqslant_$ or $_\leqslant'_$ without having to use commutativity of $_+_$. That is show

$$\mathsf{R}\Leftrightarrow\mathsf{S} : \{\mathsf{m}\ \mathsf{n} : \mathbb{N}\} \to \mathsf{m}\ \mathsf{R}\ \mathsf{n} \Leftrightarrow \mathsf{m}\ \mathsf{S}\ \mathsf{n}$$

for R being $\leqslant+$ or $+\leqslant$ and depending on the choice for R, S is either $\leqslant$ or $\leqslant'$.

Chapter 8

Deduction

So far I have introduced the rules of Type Theory informally and have explained logical principles by the proposition as types explanation. But in the end we want to make the rules of reasoning precise. This can be done by presenting a formal system defining the logic or the type theory.

In this chapter we will look at a very modest subset, basically the logic corresponding to propositional logic using only implication, which corresponds to section 2.1. We are not going to model β-reduction but only present a notion of provability which is called *natural deduction*. We will also study the relationship to combinatory logic and give a formal account of the translation presented in section 2.4.

Another import aspect of a logical system is its semantics. We will use the semantics to prove non-derivability results, i.e. that something is not provable. First we use the naive semantics which allows us to disprove statements by giving a counterexample but then we will explore a more precise semantics, Kripke models, which allows us to disprove also classically valid statements like Pierce's formula $((\mathsf{P} \to \mathsf{Q}) \to \mathsf{P}) \to \mathsf{P}$. We prove completeness of Kripke models (i.e. everything which holds semantically is also derivable) and then use a modified from of the completeness result to prove normalisation — this is inspired by a technique called *normalisation by evaluation*.

This chapter also serves a different purpose in giving more examples how the technology we have introduced in the previous chapters can be used, in particular how we can use inductive families to present deductive systems.

8.1 The rules of natural deduction

The first step is to formally define the syntax of propositions, i.e. formulas: no surprises here they will be trees. The trees are built from implications which we will write as _[⇒]_ but what is at the leaves of the trees? OK we need to introduce propositional variables which I am going to represent as strings.

```
data Form : Set where
  Var : String → Form
  _[⇒]_ : Form → Form → Form
```

So for example we can represent the proposition

$$(\mathsf{Q} \to \mathsf{R}) \to (\mathsf{P} \to \mathsf{Q}) \to \mathsf{P} \to \mathsf{R} : \mathsf{prop}$$

as

```
((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
  [⇒] (Var "P") [⇒] (Var "R")
```

Using strings to represent propositional variables is not the best choice but we will go for it in the moment. However, we will not use strings for the variables used in terms or derivations as we will see in a moment.

We want to define a predicate what propositional formulas are derivable, that is $\vdash : \mathsf{Form} \to \mathsf{Set}$ but this is not sufficient because we need to take care of assumptions as well. That is we are going to introduce a type $\mathsf{Con} : \mathsf{Set}$ of contexts and then define derivations as $_\vdash_ : \mathsf{Con} \to \mathsf{Form} \to \mathsf{Set}$.

How to represent contexts? As I said I am not going to use names to represent variables but only pointers, or actually a form of *de Bruijn indices*. Hence a context is just a sequence of assumptions, i.e. a list of formulas:

```
Con = List Form
```

We are going to use $\Gamma\ \Delta\ \Theta : \mathsf{Con}$ to quantify over contexts and $\mathsf{A}\ \mathsf{B}\ \mathsf{C} : \mathsf{Form}$ for formulas.

Ok, what are the rules of natural deduction. There are two rules to deal with implication: the first one corresponds to λ-abstraction and it conveys the idea: to prove $A \Rightarrow B$ we assume A and prove B:

$$\frac{(\mathsf{A} :: \Gamma) \vdash \mathsf{B}}{\Gamma \vdash \mathsf{A}\ [\Rightarrow]\ \mathsf{B}}\ \mathsf{lam}$$

The other rule corresponds to application: If we know $\mathsf{A} \Rightarrow \mathsf{B}$ and also A then we can derive B. The context stays the same, i.e. the number of assumptions doesn't change.

$$\frac{\Gamma \vdash \mathsf{A}\ [\Rightarrow]\ \mathsf{B} \quad \Gamma \vdash \mathsf{A}}{\Gamma \vdash \mathsf{B}}\ \mathsf{app}$$

This rule is also called *modus ponens*.

We still need some structural rules to access the context of assumptions. In the usual syntax of λ-calculus we use named variables but really these variables are just pointers into the context. Hence we can replace variables by numbers representing the offset within the context. This was introduced by Niklas de Bruijn and hence we call this representation of variables *de Bruijn indices*. E.g. the λ-term $\lambda\ \mathsf{f} \to \lambda\ \mathsf{x} \to \mathsf{f}\ \mathsf{x}\ \mathsf{x}$ would be written as $\lambda\ \lambda\ 1\ 0\ 0$, because x is

represented by 0 since it was the last variable introduced and f is represented by the index 1 because it is one step down in our list of assumptions.

Hence we introduce the two rules:

$$\frac{}{(A :: \Gamma) \vdash A}\ \mathsf{zero} \qquad \frac{\Gamma \vdash A}{(B :: \Gamma) \vdash A}\ \mathsf{suc}$$

zero allows us to access the last item in the context, and suc discards the last item and allows us to access previous items. So for example in the context

```
(Var "P") :: (Var "Q") :: []
```

We can access

$$\frac{}{(\mathsf{Var}\ \texttt{"P"}) :: (\mathsf{Var}\ \texttt{"Q"}) :: [] \vdash \mathsf{Var}\ \texttt{"P"}}\ \mathsf{zero}$$

and

$$\frac{\dfrac{}{(\mathsf{Var}\ \texttt{"Q"}) :: [] \vdash \mathsf{Var}\ \texttt{"Q"}}\ \mathsf{zero}}{(\mathsf{Var}\ \texttt{"P"}) :: (\mathsf{Var}\ \texttt{"Q"}) :: [] \vdash \mathsf{Var}\ \texttt{"Q"}}\ \mathsf{suc}$$

Actually since we don't have a special type of *assumptions* (corresponding to variables) we can apply suc to any derivation. This rule is usually called weakening. In terms of de Bruijn indices it means that we can replace 1 2 by suc (0 1) that is we *weaken* all free variables, i.e. add a successor.

As before for _⩽_, derivations are an indexed inductive definition with constructors corresponding to rules:

```
data _⊢_ : Con → Form → Set where
  zero : (A :: Γ) ⊢ A
  suc  : Γ ⊢ A → (B :: Γ) ⊢ A
  lam  : (A :: Γ) ⊢ B → Γ ⊢ (A [⇒] B)
  app  : Γ ⊢ (A [⇒] B) → Γ ⊢ A → Γ ⊢ B
```

As an example we derive (Q → R) → (P → Q) → P → R. That is we translate the meta level Agda construction

```
λ f g x → f (g x) : (Q → R) → (P → Q) → P → R
```

into

```
d-C : [] ⊢ ((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
           [⇒] (Var "P") [⇒] (Var "R")
d-C = lam (lam (lam (app (suc (suc zero)) (app (suc zero) zero))))
```

8.2 Semantics

Ok, we can translate λ-terms into derivations in our calculus, using de Bruijn indices to represent variables. However, certain formulas should not be derivable, for example we should not be able to derive

```
[] ⊢ ((Var "P") ⇒ (Var "Q")) ⇒ (Var "P").
```

Why not. Because we can actually show that this proposition is false for instances of the variables:

```
cex : ({P Q : prop} → (P ⇒ Q) ⇒ P) → ⊥
cex h = h {⊥} {⊤} (λ x → tt)
```

Hence if our system is reasonable, the technical term is *sound*, we should expect that this formula is not derivable.

When we needed to show that certain instances of _⩽_ are not derivable, e.g. ¬ (3 ⩽ 2) we could just use pattern matching which ended with impossible branches. Can we do the same here to prove:

```
([] ⊢ (((Var "P") [⇒] (Var "Q")) [⇒] (Var "P"))) → ⊥
```

We quickly notice that this doesn't lead anywhere. In particular the app-rule allows us to introduce more and more complicated formulas whose unprovability doesn't lead to a situation where no rules are applicable.

However, the solution is to actually prove soundness, that is to show that any formula we can derive is actually true for any instances of the propositional variables. This provides more than only just a tool to prove non-derivability it also shows us that our system is sensible.

We define semantics of formulas, contexts and derivations. The semantics of a formula is a proposition, but we need to know an interpretation of the propositional variables. I call this an *environment*:

```
Env = String → prop
```

Now we can define the semantics of formulas:

```
⟦_⟧F : Form → Env → prop
⟦ Var x ⟧F ρ     = ρ x
⟦ A [⇒] B ⟧F ρ = ⟦ A ⟧F ρ ⇒ ⟦ B ⟧F ρ
```

I am using here Scott brackets ⟦ .. ⟧, named after Dana Scott, to signal the demarcation between syntax and semantics.

The definition is very straightforward, variables we look up in the context and _[⇒]_ is translated as _⇒_. Contexts are just the conjunction of all the assumptions, hence:

```
⟦_⟧C : Con → Env → prop
⟦ [] ⟧C ρ       = ⊤
⟦ A :: Γ ⟧C ρ = ⟦ A ⟧F ρ ∧ ⟦ Γ ⟧C ρ
```

Finally, a derivation is translated into *the assumptions imply the conclusions:*

```
⟦_⟧d : Γ ⊢ A → {ρ : Env} → ⟦ Γ ⟧C ρ → ⟦ A ⟧F ρ
⟦ zero ⟧d (p , _)   = p
```

```
⟦ suc d ⟧d (_ , γ)  =  ⟦ d ⟧d γ
⟦ lam d ⟧d γ        =  λ p → ⟦ d ⟧d (p , γ)
⟦ app d e ⟧d γ      =  ⟦ d ⟧d γ (⟦ e ⟧d γ)
```

The rules for zero and suc translate indices into the access to the context (similar to our definition of _!!_ earlier), and lam is translated using λ-abstraction and app using application.

We can now reap the fruits of our efforts and apart from having the happy feeling that our calculus actually makes sense we can prove the non-derivability example. First we need to cook up an environment:

```
ρ₀ : Env
ρ₀ "P" = ⊥
ρ₀ "Q" = ⊤
ρ₀ _   = ⊥
```

Here we also see why I don't like to use strings for variables: we need to assign a meaning to every string and not just to the ones we used in our formula. Here our solution is to interpret all other variables with $\bot$ as a default.

We are now ready to show the non-derivability result:

```
cex-d : ([] ⊢ (((Var "P") [⇒] (Var "Q")) [⇒] (Var "P"))) → ⊥
cex-d d = ⟦ d ⟧d {ρ₀} tt (λ ())
```

The method we have used exploits that we could refute the proposition semantically whose non-derivability we wanted to show. However, for a classically true formula we cannot do this, as we cannot refute excluded middle as we have seen previously. Hence the naive semantics we have shown here isn't good enough. While it is *sound*, it is not *complete*.

8.3 Equivalence with combinatory logic

In section 2.4 we informally showed that pure functions in simply typed λ-calculus can be represented using just the combinators S and K. We will do this formally now and show that provability in combinatory logic, i.e. a logic which uses only S, K and application but no lam is equivalent to provability in natural deduction.

One of the main advantages of combinatory logic is that we don't need to use variables, aka assumptions. That means we don't have to use contexts but can use a one sided version of $\vdash$:

```
data ⊢sk : Form → Set where
  SS : ⊢sk ((A [⇒] B [⇒] C)
            [⇒] (A [⇒] B) [⇒] A [⇒] C)
  KK : ⊢sk (A [⇒] B [⇒] A)
  app : ⊢sk (A [⇒] B) → ⊢sk A → ⊢sk B
```

Our goal is to show that propositions provable from no assumptions are equivalent to propositions provable in propositional logic.

```
thm : ([] ⊢ A) ⇔ ⊢sk A
```

The direction from right to left is the easy one. All we need to show is that the combinators S and K are provable in natural deduction:

```
K-d : [] ⊢ (A [⇒] B [⇒] A)
K-d = lam (lam (suc zero))
S-d : [] ⊢ ((A [⇒] B [⇒] C)
  [⇒] (A [⇒] B) [⇒] A [⇒] C)
S-d =
  lam (lam (lam (app (app (suc (suc zero)) zero)
    (app (suc zero) zero))))
```

We only need them in the empty context here but we could actually prove them in any context. We can now show one implication $\vdash \mathsf{sk}\ A \Rightarrow [] \vdash A$ by induction over the derivation of $\vdash \mathsf{sk}$:

```
proj₂ thm SS = S-d
proj₂ thm KK = K-d
proj₂ thm (app d e) = app (proj₂ thm d) (proj₂ thm e)
```

The other direction is a bit more difficult. We cannot hope to prove anything by induction over $[] \vdash A$ because as soon as we encounter lam we need to talk about non-empty contexts.

To be able to compare natural deduction with combinatory logic we introduce a two-sided version of natural deduction with assumptions. Here we are using the same rules as for natural deduction:

```
data _⊢skC_ : Con → Form → Set where
  zero : (A :: Γ) ⊢skC A
  suc  : Γ ⊢skC A → (B :: Γ) ⊢skC A
  SS   : Γ ⊢skC ((A [⇒] B [⇒] C)
                 [⇒] (A [⇒] B) [⇒] A [⇒] C)
  KK   : Γ ⊢skC (A [⇒] B [⇒] A)
  app  : Γ ⊢skC (A [⇒] B) → Γ ⊢skC A → Γ ⊢skC B
```

We can show that derivability in the empty context for $\vdash \mathsf{skC}$ coincides with provability in $\vdash \mathsf{sk}$. Actually for our proof we only need one direction:

```
skC→sk : [] ⊢skC A → ⊢sk A
skC→sk SS       = SS
skC→sk KK       = KK
skC→sk (app d e) = app (skC→sk d) (skC→sk e)
```

Unlike for natural deduction we can prove this by derivation over $[] \vdash \mathsf{skC}\ A$ for the empty context because there is no lam and the variable rules do not apply in the empty context.

We can now try to show a more general implication relating natural deduction (_⊢_) and the two-sided combinatory logic: (_⊢skC_):

```
⊢→⊢skC : Γ ⊢ A → Γ ⊢skC A
```

It is clear that we are going to prove this by induction over Γ ⊢ A, i.e. we pattern match on the derivation:

```
⊢→⊢skC zero      = ?
⊢→⊢skC (suc d)   = ?
⊢→⊢skC (lam d)   = ?
⊢→⊢skC (app d e) = ?
```

The cases for zero, suc and app are straightforward because we have corresponding constructors for ⊢skC:

```
⊢→⊢skC zero      = zero
⊢→⊢skC (suc d)   = suc (⊢→⊢skC d)
⊢→⊢skC (app d e) = app (⊢→⊢skC d) (⊢→⊢skC e)
```

It remains the case for lam which is the core of the proof, namely that we can simulate lam using just S and K. Hence we need to prove a lemma, that _⊢skC_ is closed under lam:

```
lam-sk : (A :: Γ) ⊢skC B → Γ ⊢skC (A [⇒] B)
```

We will do this by induction over derivations again:

```
lam-sk zero      = ?
lam-sk (suc d)   = ?
lam-sk SS        = ?
lam-sk KK        = ?
lam-sk (app d e) = ?
```

The first line, the case for zero we used zero to prove (A :: Γ) ⊢skC A hence we need to show Γ ⊢skC (A [⇒] A). This is exactly the point where we needed I. Hence we better derive it:

```
II : Γ ⊢skC (A [⇒] A)
II {A = A} = app (app SS KK) (KK {B = A})
```

Here we encounter the same problem as before, we need to instantiate one of the propositional variables explicitly. Using II we can cover the zero-case:

```
lam-sk zero = II
```

The next case is suc, given a derivation d : Γ ⊢skC A we get suc d : (B :: Γ) ⊢skC A. Hence we need to derive Γ ⊢skC B ⇒ A. In this case we

used suc aka weakening to ignore the last variable and this is clearly the time for K:

```
lam-sk (suc d)  =  app KK d
```

The cases where we used SS or KK to derive $(A :: \Gamma) \vdash \mathsf{skC}\ B$ are similar, because we certainly ignored the last variable. Hence we are also using KK for those cases together with the same combinator:

```
lam-sk SS  =  app KK SS
lam-sk KK  =  app KK KK
```

The final and interesting case is app, we are abstracting from an application. That is exactly the point which justifies S which distributes abstractions over an application. This is the only case where we actually need to refer to the recursive result in the proof:

```
lam-sk (app d e)  =  app (app SS (lam-sk d)) (lam-sk e)
```

We have now covered all the cases:

```
lam-sk : (A :: Γ) ⊢skC B → Γ ⊢skC (A [⇒] B)
lam-sk zero       = II
lam-sk (suc d)    = app KK d
lam-sk SS         = app KK SS
lam-sk KK         = app KK KK
lam-sk (app d e)  = app (app SS (lam-sk d)) (lam-sk e)
```

Now we can complete the proof of $\vdash\!\!\rightarrow\vdash\mathsf{skC}$ by filling in the case for lam:

```
⊢→⊢skC : Γ ⊢ A → Γ ⊢skC A
⊢→⊢skC zero       = zero
⊢→⊢skC (suc d)    = suc (⊢→⊢skC d)
⊢→⊢skC (lam d)    = lam-sk (⊢→⊢skC d)
⊢→⊢skC (app d e)  = app (⊢→⊢skC d) (⊢→⊢skC e)
```

We can now prove the other direction of thm, namely that $[] \vdash A \Rightarrow \vdash\mathsf{sk}\ A$ by using $\vdash\!\!\rightarrow\vdash\mathsf{skC}$ in the special case of the empty context and combining it with $\mathsf{skC}\!\rightarrow\!\mathsf{sk}$:

```
thm : ([] ⊢ A) ⇔ ⊢sk A
proj₁ thm d          = skC→sk (⊢→⊢skC d)
proj₂ thm SS         = S-d
proj₂ thm KK         = K-d
proj₂ thm (app d e)  = app (proj₂ thm d) (proj₂ thm e)
```

We can apply the left to right direction to our example to derive a combinatory proof of the example derivation:

```
d-C-sk : ⊢sk (((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
              [⇒] (Var "P") [⇒] (Var "R"))
d-C-sk = proj₁ thm d-C
```

The derivation we obtain is truly huge:

```
app
(app SS
(app (app SS (app KK SS))
  (app
    (app SS
      (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK SS))))
    (app
      (app SS
        (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK KK))))
      (app (app SS (app KK KK)) (app (app SS KK) KK))))))
(app
(app SS
  (app (app SS (app KK SS))
    (app
      (app SS
        (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK SS))))
      (app
        (app SS
          (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK KK))))
        (app (app SS (app (app SS (app KK SS)) (app KK KK)))
          (app KK KK))))))
(app
  (app SS
    (app (app SS (app KK SS))
      (app
        (app SS
          (app (app SS (app KK SS)) (app (app SS (app KK KK)) (app KK SS))))
        (app (app SS (app KK KK)) (app KK KK)))))
  (app (app SS (app KK KK)) (app KK KK))))
```

Indeed, we have implemented only the inefficient algorithm which is unsuitable when translating by hand. In section 2.4 we informally described a more efficient version which is more useful when doing the translation by hand. I leave it as an exercise to implement a modified version of lam-sk which implements the more efficient translation.

8.4 Kripke models

The naive semantics we introduced in section 8.2 isn't good enough to prove the non-derivability of propositions which are classically true such as the Pierce formula $((P \to Q) \to P) \to P$, this can be easily seen by checking the truth table.

It is less obvious that we cannot derive it. We can show that it entails the excluded middle

```
Pierce = {P Q : prop} → ((P → Q) → P) → P
```

```
pierce→tnd : Pierce → {P : prop} → TND P
pierce→tnd pierce {P} = pierce {TND P} {⊥} (λ p → case⊥ (nntnd p))
```

which strongly suggests that we cannot prove it but this doesn't amount to a proof.

To address this problem we need to come up with a better semantics, one which is actually complete that is for any underivable proposition we can construct a semantic counterexample. Or more positively: if a formula is true in the semantics then it is also derivable. This is the inverse of soundness which just stated that derivable propositions are true in the semantics.

A semantics which is complete for the fragment of intuitionistic propositional logic we have presented are Kripke models. The basic idea is that we have different worlds which correspond to stages of knowledge. In each world we know that certain atomic propositions hold and if we expand our knowledge we may learn that additional propositions are true but one which was true will remain so. The interesting point is how to interpret implication. We say that $P \to Q$ holds in a world if whenever in the future I will find out P then at this point Q will also hold.

Using this idea we can present a countermodel using just two worlds: lets call them w_0 and w_1. w_1 is a future of w_0, we write $w_1 \sqsubseteq w_0$ and while we know nothing in w_0, we know P in w_1, we write $w_1 \Vdash P$. Now we observe that $(P \to Q) \to P$ holds in w_0, we write $w_0 \Vdash (P \to Q) \to P$ because in w_0 Q doesn't hold hence the implication holds trivially and in the future w_1 we actually know P hence the formula holds. However, since w_0 doesn't force P, the Pierce formula doesn't hold in w_0. Hence we have found a countermodel.

Ok, lets do this a bit more formally: A Kripke model is given by

```
record Kripke : Set₁ where
  field
    Worlds : Set
    _⊑_ : Worlds → Worlds → prop
    refl⊑ : {w : Worlds} → w ⊑ w
    trans⊑ : {w₀ w₁ w₂ : Worlds} → w₀ ⊑ w₁ → w₁ ⊑ w₂ → w₀ ⊑ w₂
```

Here $_\sqsubseteq_$ is the future relation, we write $w_1 \sqsubseteq w_0$ to mean that w_1 is a future of of w_0. We assume that the future relation is reflexive and transitive, i.e. it is a preorder.

```
_⊩_ : Worlds → String → prop
mon⊩ : {w₀ w₁ : Worlds}
   → w₁ ⊑ w₀ → {x : String} → w₀ ⊩ x → w₁ ⊩ x
```

The relation $w \Vdash x$ (often called *forcing*) expresses that the propositional variable x presented by a string holds in world w. This should be monotone, any variable we know holds now will still be valid in the future.

This completes the definition of a Kripke model. Now we assume as given a Kripke model `K : Kripke` and we are going to extend $_\Vdash_$ to formulas and

context and uses this to express what it means that a sequent $\Gamma \vdash A$ is valid in K. We define _⊩F_ using the interpretation of _[⇒]_ suggested earlier:

```
_⊩F_ : Worlds → Form → prop
w ⊩F Var x = w ⊩ x
w ⊩F A [⇒] B = {w' : Worlds} → w' ⊑ w → w' ⊩F A → w' ⊩F B
```

We extend _⊩_ to contexts, which simply says that it should hold for all formulas in the context. As in the naive semantics we are interpreting a context as the conjunction of all the formulas it contains.

```
_⊩C_ : Worlds → Con → prop
w ⊩C [] = True
w ⊩C (A :: Γ) = w ⊩F A ∧ w ⊩C Γ
```

We write $\Gamma \models A$ to express that semantically Γ entails A, i.e. every world that forces Γ also forces A:

```
_ ⊨ _ : Con → Form → prop
Γ ⊨ A = {w : Worlds} → w ⊩C Γ → w ⊩F A
```

We can now show soundness by induction over $\Gamma \vdash A$:

```
⟦_⟧ : Γ ⊢ A → Γ ⊨ A
```

I leave the details as an exercise. To show soundness you first need to show that _⊩F_ and _⊩C_ are monotone.

Now to complete the proof of the non-derivability of the Pierce formula we have to formally define the Kripke model we sketched earlier.

```
module Pierce where
  data W : Set where
    w0 w1 : W
  data _⊑P_ : W → W → prop where
    refl : {w : W} → w ⊑P w
    w10 : w1 ⊑P w0
  data _⊩P_ : W → String → prop where
    w1p : w1 ⊩P "P"
```

I leave it as an exercise to complete the definition of the Kripke model. We can then show that the Pierce formula is not forced in w_0:

```
not⊩Pierce : ¬ (w0 ⊩F (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

and hence using soundness it is not provable:

```
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

8.5 Completeness

Ok, Kripke models are good enough to refute the Pierce formula but are they good enough to refute any formula that is not derivable? This means are they complete? Indeed we can construct one Kripke model, the universal model, which models exactly derivability. Hence if a formula is holds in all Kripke models then it will also hold in this one and hence it is derivable.

We construct this model from out derivation relation, that is we turn derivability into a Kripke model. The future relation we obtain be extending derivability on contexts, we define $\Gamma \vdash\mathsf{C}\ \Delta$, if all the formulas in Δ are derivable from Γ. Formally this can be defined as follows:

```
data _⊢C_ : Con → Con → Set where
  [] : Γ ⊢C []
  _::_ : Γ ⊢ A → Γ ⊢C Δ → Γ ⊢C (A :: Δ)
```

The idea is that $_\vdash\mathsf{C}_$ corresponds to parallel substitutions. That is an element $\Gamma \vdash\mathsf{C}\ \mathsf{A}_0\ ,\ \mathsf{A}_1\ ,\ ..\ ,\ \mathsf{An}$ is a sequence of derivations $\mathsf{a}_0 :: \mathsf{a}_1 :: .. :: \mathsf{an}$ with $\Gamma \vdash \mathsf{ai} : \mathsf{Ai}$. We have an identity substitution:

```
idC : Γ ⊢C Γ
```

and we can *cut* a term and a substitution:

```
cut : Γ ⊢C Δ → Δ ⊢ A → Γ ⊢ A
```

Iterating this operation provides us with a cut for $_\vdash\mathsf{C}_$ corresponding to the transitivity of the future relation:

```
cutC : Γ ⊢C Δ → Δ ⊢C Θ → Γ ⊢C Θ
```

The forcing relation for our universal model is just derivability applied to variables and monotonicity is just an instance of cut. This completes the construction of the universal Kripke model - the details are again left as an exercise:

```
UM : Kripke
UM = record
  {Worlds = Con
  ; _⊑_ = _⊢C_
  ; refl⊑ = idC
  ; trans⊑ = cutC
  ; _⊩_ = λ Γ x → Γ ⊢ Var x
  ; mon⊩ = λ γ a → cut γ a
  }
```

The derived forcing relation $_\Vdash\mathsf{F}_$for this model has the same type as the derivation relation. Hence the key lemma is to establish that forcing and derivability coincide. This can be shown by mutual induction over the formula A:

```
q : Γ ⊩F A → Γ ⊢ A
u : Γ ⊢ A → Γ ⊩F A
```

I use q for *quote* sind it goes from the semantics into the syntax, and u for *unquote*. Using these lemmas it is not hard to derive completeness:

```
compl : Γ ⊢ A → Γ ⊨ A
```

8.6 Normalisation

There is an alternative way to establish non-provability which corresponds more to our naive reasoning. We only consider derivations which are normal, which means that they avoid derivations which can be β-reduced, i.e. that correspond to λ-terms of the form $(\lambda\ x \to M)\ N$ and which always use λ to prove an implication. The latter corresponds to always η-expanding and is called the long η normal form.

Now when we try to prove the Pierce formula

```
p : ((P → Q) → P) → P
p = ?
```

the first step has to be a λ:

```
p = λ x → ?
```

with x : (P → Q) → P. Now our goal is P and the only assumption that produces something in P is x. Hence we have to refine by x:

```
p = λ x → x ?
```

Ok our goal is now to prove P → Q. Since this is an implication we need to use a λ:

```
p = λ x → x (λ y → ?)
```

We now have a new assumption y : P and our goal is Q. But now we are stuck because none of our assumptions produces Qs. Hence there is no normal proof.

It is clear that to use this idea to show that there is no derivation of ((P → Q) → P) → P in the original system we need to prove that for any derivation Γ ⊢ A there is a normal derivation Γ ⊢nf A.

Let's start with defining normal derivations inductively. First of all we need to recover variables, which correspond to using an assumption from the context.

```
data _⊢v_ : Con → Form → Set where
  zero : (A :: Γ) ⊢v A
  suc : Γ ⊢v A → (B :: Γ) ⊢v A
```

This derivation correspond to numbers, indeed they are de Bruijn indices which we have discussed earlier.

We now mutually define three kinds of derivations:

```
data _⊢nf_ : Con → Form → Set
data _⊢ne_ : Con → Form → Set
data _⊢sp_,_ : Con → Form → Form → Set
```

Here _⊢nf_ are the normal derivations, _⊢ne_ are *neutral derivations* this is they are derivations which are a variable applied to a sequence of normal terms, and _⊢sp_,_ is the *spine* of a neutral derivation, i.e. the sequence of normal terms to which the variable is applies. The spine derivation has two formula parameters: the first one is the assumption we start with and the 2nd one is the goal.

So let's look at _⊢sp_,_ first:

```
data _⊢sp_,_ where
  []  : Γ ⊢sp C , C
  _::_ : Γ ⊢nf A → Γ ⊢sp B , C → Γ ⊢sp (A [⇒] B) , C
```

Once we have reached the end of the sequence the goal has to agree with the premise. The _::_ constructor shortens the distance from the premise A [⇒] B to B by providing Γ ⊢nf A.

The neutral derivations are simply given by a variable applied to a spine:

```
data _⊢ne_ where
  ne : Γ ⊢v A → Γ ⊢sp A , B → Γ ⊢ne B
```

Finally we can define normal derivations:

```
data _⊢nf_ where
  ne→nf : Γ ⊢ne Var x → Γ ⊢nf (Var x)
  lam : (A :: Γ) ⊢nf B → Γ ⊢nf (A [⇒] B)
```

A normal derivation of an implication is always given by lam but when our goal is a variable we have to resort to neutral derivations.

We can now show that there is no normal derivation of the Pierce formula just by using pattern matching:

```
no-Pierce-nf-deriv :
  ¬ ([] ⊢nf (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

But as usual I leave this as an exercise.

Now let's prove normalisation. Actually we have already done it more or less in the previous section when we proved completeness. The basic idea is to combine soundness and completeness. That is using soundness we go from a derivation to the semantics in the universal Kripke model. Using completeness we can go back to a derivation but this wouldn't tell us anything new. However,

we can observe that the completeness proof always returns a normal derivation! Hence by combining soundness and completeness we can prove normalisation.

The Kripke model we are using is just a variation of the universal model from the last section. The future relation just uses variables since normal derivations are not closed under general substitutions but only under renamings

```
data _⊢vC_ : Con → Con → Set where
  [] : Γ ⊢vC []
  _::_ : Γ ⊢v A → Γ ⊢vC Δ → Γ ⊢vC (A :: Δ)
```

We need to show that we can establish identity and cut operations for variable substitutions:

```
idvC : Γ ⊢vC Γ
cutvC : Γ ⊢vC Δ → Δ ⊢vC Θ → Γ ⊢vC Θ
cutne : Γ ⊢vC Δ → Δ ⊢ne A → Γ ⊢ne A
```

the proofs are the same as before but now applied to the refined derivation forms. Hence we get another Kripke model:

```
NM : Kripke
NM = record
  {Worlds = Con
  ; _⊑_ = _⊢vC_
  ; refl⊑ = idvC
  ; trans⊑ = cutvC
  ; _⊩⊢_ = λ Γ x → Γ ⊢ne Var x
  ; mon⊩⊢ = λ γ a → cutne γ a
  }
```

We also need to refine the types of q and u:

```
q : Γ ⊩⊢F A → Γ ⊢nf A
u : Γ ⊢ne A → Γ ⊩⊢F A
```

And using q and soundness ([[_]]) we can prove normalisation:

```
norm : Γ ⊢ A → Γ ⊢nf A
```

Putting norm and no-Pierce-nf-deriv together we derive an alternative proof of the non-derivability of the Pierce formula"

```
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
no-Pierce-deriv p = no-Pierce-nf-deriv (norm p)
```

So in a way normalisation is a canned completeness proof where the semantic component is replaced by a purely syntactic definition of normal forms.

8.7 History

Natural deduction was introduced by Gentzen in the 1930ies [Gen35, Gen35]. In his presentation he didn't use explicit contexts as we did but marked assumptions in derivations and linked them to the position where they are discharged. He also presented the system for predicate logic and covered all logical connectives. Gentzen also introduced a different calculus, the sequent calculus, as a tool to prove his *Hauptsatz* which corresponds to the normalisation theorem we have discussed.

We have already discussed the history of combinatory logic in section 2.6. The completeness proof was first sketched by Schönfinkel [Sch24] it can be found in [Cur30] and in the form we have presented it was first given by Rosenbloom [Ros50] (according to [CH06]).

Kripke modes were first introduced for modal logic [Kri59]. The application to intuitionistic logic is discussed by Schütte in [Sch68]. The completeness theorem in the form we have presented is is only provable for the fragment including only $\rightarrow$, $\wedge$, True and $\forall$. To include the remaining connectives, i.e. False, $\vee$, $\exists$ one has to use Beth models [Bet48] (as described in [TvU99]) or resort to a classical metatheory. In general one finds a good overview over this kind of material in the books by Troelstra and van Dalen [TvD88b, TvD88a].

My presentation of normalisation is inspired by *Normalisation by Evaluation* which was discovered by Schwichtenberg and Berger [BS91]. The relation between this algorithm and the completeness proof for Kripke models was noted in [AHS95].

8.8 Exercises

1. Implement the efficient algorithm for translating λ-terms into combinatory logic which was described in section 2.4. That is

   - If the variable x does not appear in M then $\lambda\,x\ \rightarrow\ M$ is just K M.
   - $\lambda\,x\ \rightarrow\ g\,x$ is just g.

   You need to reimplement:

   ```
   lam-sk-x : (A :: Γ) ⊢skC B → Γ ⊢skC (A [⇒] B)
   ```

   The rest stays basically the same — we only replace one direction of thm:

   ```
   ⊢→⊢skC-x : Γ ⊢ A → Γ ⊢skC A
   ⊢→⊢skC-x zero      = zero
   ⊢→⊢skC-x (suc d)   = suc (⊢→⊢skC-x d)
   ⊢→⊢skC-x (lam d)   = lam-sk-x (⊢→⊢skC-x d)
   ⊢→⊢skC-x (app d e) = app (⊢→⊢skC-x d) (⊢→⊢skC-x e)
   thm-x : ([] ⊢ A) → ⊢sk A
   thm-x d = skC→sk (⊢→⊢skC-x d)
   ```

Try the improved version on

```
d-C-sk-x : ⊢sk (((Var "Q") [⇒] (Var "R")) [⇒] ((Var "P") [⇒] (Var "Q"))
                [⇒] ((Var "P") [⇒] (Var "R")))
d-C-sk-x = thm-x d-C
```

the result should be:

```
app (app SS (app KK SS)) KK
```

2. Complete the soundness proof for Kripke models from section 8.4. That is prove the following propositions:

```
mon-F : w₁ ⊑ w₀ → {A : Form} → w₀ ⊩F A → w₁ ⊩F A
mon-C : w₁ ⊑ w₀ → {Γ : Con} → w₀ ⊩C Γ → w₁ ⊩C Γ
⟦_⟧ : Γ ⊢ A → Γ ⊨ A
```

3. Complete the proof that the Pierce formula is not derivable using the soundness result and by completing the definition of the Kripke counter-model.

   Complete the definition of the Kripke model KP:

```
trans⊑P : {w₀ w₁ w₂ : W} → w₀ ⊑P w₁ → w₁ ⊑P w₂ → w₀ ⊑P w₂
mon⊩P : {w₀ w₁ : W}
  → w₁ ⊑P w₀ → {x : String} → w₀ ⊩P x → w₁ ⊩P x
KP : Kripke
KP = record
  {Worlds = W
  ; _⊑_ = _⊑P_
  ; refl⊑ = refl
  ; trans⊑ = trans⊑P
  ; _⊩_ = _⊩P_
  ; mon⊩ = mon⊩P
  }
```

   And then using KP show that Pierce is not derivable:

```
not⊩Pierce : ¬ (w0 ⊩F (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

4. Complete the completeness proof from section 8.5, that is complete the definition of the universal Kripke model:

```
idC : Γ ⊢C Γ
cut : Γ ⊢C Δ → Δ ⊢ A → Γ ⊢ A
```

```
cutC : Γ ⊢C Δ → Δ ⊢C Θ → Γ ⊢C Θ

UM : Kripke
UM = record
  {Worlds = Con
  ; _⊑_ = _⊢C_
  ;refl⊑ = idC
  ;trans⊑ = cutC
  ; _⊩_ = λ Γ x → Γ ⊢ Var x
  ;mon⊩ = λ γ a → cut γ a
  }
```

Hint: It is useful to establish the following lemmas generalising weakening:

```
sucC : Γ ⊢C Δ → (B :: Γ) ⊢C Δ
extC : Γ ⊢C Δ → (A :: Γ) ⊢C (A :: Δ)


q : Γ ⊩F A → Γ ⊢ A
u : Γ ⊢ A → Γ ⊩F A

qC : Γ ⊩C Δ → Γ ⊢C Δ

compl : Γ ⊢ A → Γ ⊨ A
```

5. Show that there is no normal derivation of the Pierce formula:

```
no-Pierce-nf-deriv : ¬ ([] ⊢nf (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
```

6. Complete the construction of the Kripke model using normal forms as described in section 8.6. That is complete the following definitions by adapting the construction of the previous question:

```
idvC : Γ ⊢vC Γ
cutvC : Γ ⊢vC Δ → Δ ⊢vC Θ → Γ ⊢vC Θ
cutne : Γ ⊢vC Δ → Δ ⊢ne A → Γ ⊢ne A

NM : Kripke
NM = record
  {Worlds = Con
  ; _⊑_ = _⊢vC_
  ;refl⊑ = idvC
  ;trans⊑ = cutvC
  ; _⊩_ = λ Γ x → Γ ⊢ne Var x
  ;mon⊩ = λ γ a → cutne γ a
  }
```

Once this is done we can derive the variations of q and u:

```
q : Γ ⊩F A → Γ ⊢nf A
u : Γ ⊢ne A → Γ ⊩F A
```

Finally use this to derive the normalisation function:

```
norm : Γ ⊢ A → Γ ⊢nf A
```

which can be use to provide an alternative proof of the non derivability of the Pierce formula:

```
open NormPierce
no-Pierce-deriv : ¬ ([] ⊢ (((Var "P" [⇒] Var "Q") [⇒] Var "P") [⇒] Var "P"))
no-Pierce-deriv p = no-Pierce-nf-deriv (norm p)
```

Chapter 9

Eliminators and coinduction

Definitions by pattern matching and recursion are a convenient way to define functions out of inductive types but it is hard to make precise exactly what is allowed and what isn't. Also to define pattern matching precisely is not so easy and this makes it hard to use this for metatheoretic study along the lines of what we have done in the last chapter.

Previously, in the simply typed setting we have introduced the iterator and the recursor as combinators which cover recursive definitions (see section 4.2). In this chapter we will look at the extension to dependent functions using a more general combinator called the *eliminator*. We will also look at iterators and eliminators for inductive families, especially for equality.

Finally we will look at coinductive definitions but notice that the mirror in the type theory we have introduced so far is rather imperfect. To prove properties of coinductive definition we need *coinduction* which doesn't generalize coiteration but worse which cannot be given a computational semantics in the type theory we are using, which is *intensional type theory*. We will fix this later when we are introducing cubical type theory.

9.1 The eliminator for natural numbers

Ok, the point of the eliminator is to explain how we can define functions out of an inductive type, which is called elimination in contrast to introduction which is achieved by the constructors. Let's take the natural numbers as an example. We construct them using zero and suc.

When defining a dependent function out of the natural numbers we assume as given a family $\mathsf{M} : \mathbb{N} \to \mathsf{Set}$ which we call the motive. We want to define

$$\mathsf{f} : (\mathsf{n} : \mathbb{N}) \to \mathsf{M}\ \mathsf{n}$$

We do this by pattern matching and recursion:

$$\begin{aligned} \mathsf{f}\ \mathsf{zero} &= \mathsf{z} \\ \mathsf{f}\ (\mathsf{suc}\ \mathsf{n}) &= \mathsf{s}\ \mathsf{n}\ (\mathsf{f}\ \mathsf{n}) \end{aligned}$$

where

```
z : M zero
s : (n : ℕ) → M n → M (suc n)
```

Note that s needs to see n because it is the index to M. Hence the eliminator behaves like the recursor in that we have to supply the natural number and not only the recursively computed result.

We can put this together into one constant, the eliminator:

```
Eℕ : (M : ℕ → Set)
   → M 0
   → ((n : ℕ) → M n → M (suc n))
   → (n : ℕ) → M n
Eℕ M z s zero    = z
Eℕ M z s (suc n) = s n (Eℕ M z s n)
```

You may notice that while the type is different the equations defining the eliminator are exactly the same as the one for the recursor.

Let's revisit some examples and translate them from pattern matching to using Eℕ, e.g. there is rneutr+

```
rneutr+ : (n : ℕ) → n + 0 ≡ n
rneutr+ zero    = refl
rneutr+ (suc n) = cong suc (rneutr+ n)
```

It is straightforward to turn this into an application of the eliminator:

```
rneutr+-e : (n : ℕ) → n + 0 ≡ n
rneutr+-e = Eℕ (λ n → n + 0 ≡ n) refl (λ n ih → cong suc ih)
```

I have called the recursive result ih short for *induction hypothesis* and indeed the type of Eℕ indeed corresponds to the induction principle for natural numbers.

However, not all uses of Eℕ are of a logical nature, there are completely computational uses of it. One example we already looked it was:

```
zeroes-v : (n : ℕ) → Vec ℕ n
zeroes-v zero    = []
zeroes-v (suc n) = 0 :: (zeroes-v n)
```

And again we can read of the parameters for the eliminator from the recursive definition:

```
zeroes-v-e : (n : ℕ) → Vec ℕ n
zeroes-v-e = Eℕ (Vec ℕ) [] (λ n v → 0 :: v)
```

The recursor can be derived from the eliminator, all we need to do is to use a constant family as the motive:

```
Rℕ-E : {M : Set} → M → (ℕ → M → M) → ℕ → M
Rℕ-E {M} = Eℕ (λ _ → M)
```

and again since the iterator is a special case of the recursor we get this as well:

```
Itℕ-E : {M : Set} → M → (M → M) → ℕ → M
Itℕ-E z s = Rℕ-E z (λ _ → s)
```

9.2 Iterator and eliminators for dependent types

OK, we have seen a dependent eliminator for a non-dependent type like the natural numbers. I'll leave it as an exercise to derive eliminators for the inductive types we discussed in 4.3 , e.g. List, Expr,RoseTree and Ord. To have a bit more excitement we shall look at a dependent inductive type like Vec. As a reminder Vec was defined as follows:

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

However, before looking at the eliminator for Vec, maybe we should consider the iterator? But hang on, since Vec is a dependent type how can it have a non-dependent iterator? The answer is that the iterator will depend on the indices for vectors but not on the vector itself. That is we have a motive M : ℕ → Set and we want to define a function:

```
f : {n : ℕ} → Vec A n → M n
```

That is the function does depend on n but not on the vector itself. When we define this function by pattern matching we end up in the following situation:

```
f {n} []          = m-nil
f {suc n} (a :: v) = m-cons {n} a (f {n} v)
```

with

```
m-nil  : M 0
m-cons : {n : ℕ} → A → M n → M (suc n)
```

That is as before we obtain the types of the methods by replacing the inductive type Vec by the motive M. And also as we are now used to we can turn this into the definition of the iterator:

```
ItVec : (M : ℕ → Set) → M 0
  → ({n : ℕ} → A → M n → M (suc n))
  → {n : ℕ} → Vec A n → M n
```

```
ItVec M m-nil m-cons []       = m-nil
ItVec M m-nil m-cons (a :: v) = m-cons a (ItVec M m-nil m-cons v)
```

An example of a non-dependent function over vectors is the _++v_.

```
_++v_ : {i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
[] ++v w       = w
(x :: v) ++v w = x :: (v ++v w)
```

Ok,_++v_ is a dependent function but it doesn't depend on vectors but only on the indices. An indeed, we can define _++v_ using only ItVec:

```
_++v-it_ : {i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
_++v-it_ {A = A} {i} {j} v w = ItVec (λ i → Vec A (i + j)) w
                                     (λ a v++w → a :: v++w) v
```

Ok, but when do we actually need dependent functions over vectors? I haven't yet given an example for such a thing but I can easily make one up. We may want to show that Vec with [] and _++v_ form an indexed monoid.

In particular we may want to show that [] is a right inverse:

```
{n : ℕ} (v : Vec A n) → v ++v [] ≡ v
```

He we alreay encounter a problem: this statement doesn't type check! The left hand side has type Vec A (n + 1) and the right hand side has type Vec A n. However, we have shown that they are equal rneutr+ : (n : ℕ) → n + 0 ≡ n (in section 6.4) and we can use subst to fix this issue:

```
rneutr++ : {n : ℕ} (v : Vec A n)
  → (subst (Vec A) (rneutr+ n) (v ++v [])) ≡ v
```

and then just use pattern matching:

```
rneutr++ {n}     []       = refl
rneutr++ {suc n} (x :: v) = ?
```

The second case is not completely straightforward because now the subst is on our way. There are more principled ways to deal with this situation but for the sake of brevity, I am just using a very special lemma that allows us to commute _::_ and subst:

```
lem : {m n : ℕ} (p : m ≡ n) (x : A) (v : Vec A m)
  → subst (Vec A) (cong suc p) (x :: v) ≡ x :: (subst (Vec A) p v)
lem refl x v = refl
```

This allows us to complete the proof:

```
rneutr++ [] = refl
rneutr++ {A = A} {n = suc n} (x :: v) =
```

```
begin
  subst (Vec A) (cong suc (rneutr+ n)) (x :: (v ++v []))
      ≡⟨ lem (rneutr+ n) x (v ++v []) ⟩
  x :: subst (Vec A) (rneutr+ n) (v ++v [])
      ≡⟨ cong (λ w → x :: w) (rneutr++ v) ⟩
    x :: v
      ■
```

We got a bit distracted on the way: the point was to derive an eliminator for Vec. In general our problem is that we want to derive a function that depends on vectors:

```
f : {n : ℕ} (v : Vec A n) → M {n} v
```

where

```
M : {n : ℕ} (v : Vec A n) → Set
```

We can achieve this by pattern matching:

```
f {0} [] = m-nil
f {suc n} (a :: v) = m-cons {n} a v (f {n} v)
```

where

```
m-nil  : M {0} []
m-cons : {n : ℕ} (a : A) (v : Vec A n) → M {n} v → M {suc n} (a :: v)
```

We turn the handle and derive an eliminator for Vec:

```
EVec : (M : {n : ℕ} → Vec A n → Set)
  → M {0} []
  → ({n : ℕ} (a : A) (v : Vec A n)
    → M {n} v → M {suc n} (a :: v))
  → {n : ℕ} (v : Vec A n) → M {n} v
EVec M m-nil m-cons [] = m-nil
EVec M m-nil m-cons (a :: v) = m-cons a v (EVec M m-nil m-cons v)
```

And now we can use our example to cook up a use case for EVec:

```
rneutr++-e : {n : ℕ} (v : Vec A n)
  → (subst (Vec A) (rneutr+ n) (v ++v [])) ≡ v
rneutr++-e {A = A} =
  EVec (λ {n} v → (subst (Vec A) (rneutr+ n) (v ++v [])) ≡ v)
    refl
    λ {n} x v h → trans (lem (rneutr+ n) x (v ++v []))
                        (cong (λ w → x :: w) (rneutr++ v))
```

9.3 Eliminators for equality

9.3.1 The recursor

The previous section was mainly a built-up for this one where we actually deal with a much simpler dependent inductive type : equality (_≡_). Here is how we defined equality as an inductive type in section 6.3:

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

We have only one constructor refl and it isn't even recursive. Without further ado we can write down the iterator:

```
It≡ : (M : A → A → Set)
 → ({a : A} → M a a)
 → {a b : A} → a ≡ b → M a b
It≡ M r refl = r
```

We can verify the properties of equality we have shown in 6.3 using only the iterator. As an example here is the proof of symmetry using pattern matching:

```
sym : {a b : A} → a ≡ b → b ≡ a
sym refl = refl
```

And it is easy to translate this into a proof using It≡:

```
sym-it : {a b : A} → a ≡ b → b ≡ a
sym-it = It≡ (λ a b → b ≡ a) refl
```

We can also prove trans, cong and subst using only the iterator because they don't depend on equality proofs.

9.3.2 J and K

The eliminator for equality has a name, it was called J by Martin-Löf and hence we shall use J instead of E≡. Here we go:

```
J : (M : {a b : A} → a ≡ b → Set)
   → ({a : A} → M {a} {a} refl)
   → {a b : A} (p : a ≡ b) → M {a} {b} p
J M m-refl refl = m-refl
```

But what can we prove with J what we couldn't already prove with It≡? We need some type which depends on equality proofs. So far we have only seen one, uniqueness of equality proofs [1]

[1] Which is traditionally called *uip* for uniqueness of identity proofs.

```
uip : {x y : A} (p q : x ≡ y) → p ≡ q
uip refl refl = refl
```

Now we should hope that we can derive uip from J, actually two instances since we pattern match twice. Ok let's start with replacing matching p:

```
uip-e : {x y : A} (p q : x ≡ y) → p ≡ q
uip-e = J (λ {x} {y} p → (q : x ≡ y) → p ≡ q) ?
```

We are left with having to show $\{a : A\}\ (q : a \equiv a) \to \mathsf{refl} \equiv q$ but this turns out tricky because it is not an instance of J. J is only applicable to define functions of the form $\{x\ y : A\}\ (p : x \equiv y) \to M\ \{x\}\ \{y\}\ p$ but not goals of the form $\{x : A\}\ (p : x \equiv x) \to M\ \{x\}\ p$. We may hope that there is a way to massage the goal so that we can use J to prove it. When I was doing my PhD we tried and failed and it was Martin Hofmann and Thomas Streicher who proved that it is indeed impossible. But before talking about this, let's notice that there is an obvious and cheap solution. We simply need another eliminator which handles the case I have just described, and this is called K. I hasten to add that this has nothing to do with the K from combinatory logic and it was just chosen because it is the next letter after J.

```
K : (M : {a : A} → a ≡ a → Set)
  → ({a : A} → M {a} refl)
  → {a : A} (p : a ≡ a) → M {a} p
K M m-refl refl = m-refl
```

Using K it is easy to complete uip-e:

```
uip-e : {x y : A} (p q : x ≡ y) → p ≡ q
uip-e = J (λ {x} {y} p → (q : x ≡ y) → p ≡ q)
  (K (λ {a} q → refl ≡ q) refl)
```

On the other hand we can derive K from uip using only subst which is derivable from J:

```
K-from-uip : (M : {a : A} → a ≡ a → Set)
  → ({a : A} → M {a} refl)
  → {a : A} (p : a ≡ a) → M {a} p
K-from-uip M m-refl {a} p = subst (M {a}) (uip refl p) m-refl
```

Indeed we need only an instance of uip namely uip refl which says that any equality proof showing $a = a$ is equal to refl: $\{x : A\}\ (q : x \equiv x) \to \mathsf{refl} \equiv q$.

At the time I thought this was the deal: we just add K and everything is fine. Indeed Conor McBride showed in his PhD that all instances of pattern matching are derivable from the eliminators and K. Hence we have a combinatory way to say what we mean by pattern matching (ok we still need to address the question what is a good inductive type but this is another issue).

However, then came Voevodsky — actually this is not completely true this was already pointed out by Martin Hofmann much earlier. The univalence principle characterises equality of types and an important consequence is that equality of types is given by equivalence which for most types boils down to having a bijection or an isomorphism (an invertible function). However, there is usually more then one bijection between types, e.g. there are two bijections showing Bool ≅ Bool, namely identity and negation. In general there are n ! bijections between a finite type with n elements and itself.

So here is a choice: we either can have K or equivalently uip or we can have univalence but we can't have both. We will discuss univalence in more detail later but let's see what are the consequences of not having K which also provides some insight how Hofmann and Streicher managed to show that it is not derivable.

Maybe we should ask the question: if we cannot prove uip what can we actually do with J. Just using the recursor we can prove that equality is an equivalence relation, that is we have:

```
refl : {a : A} → a ≡ a
sym : {a b : A} → a ≡ b → b ≡ a
trans : {a b c : A} → a ≡ b → b ≡ c → a ≡ c
```

However, maybe we should use the word *relation* carefully because there is more to equality then just being inhabited, if we don't have uip.

However, in this case we have more structure, we can show some algebraic laws. We have already seen monoids (e.g. the natural numbers with 0 and addition), and another important structure are groups, where we also have inverses (e.g. the integers with addition and unary negation). However, we don't have a group here because we have to take the indices into account but the structure we get is called a *groupoid*. Here are the laws of a groupoid in the specific case of equality:

```
lneutr≡ : {a b : A} (p : a ≡ b) → trans refl p ≡ p
rneutr≡ : {a b : A} (p : a ≡ b) → trans p refl ≡ p
assoc≡ : {a b c d : A} (p : a ≡ b) (q : b ≡ c) (r : c ≡ d)
  → trans p (trans q r) ≡ trans (trans p q) r
linv≡ : {a b : A} (p : a ≡ b) → trans (sym p) p ≡ refl
rinv≡ : {a b : A} (p : a ≡ b) → trans p (sym p) ≡ refl
```

Now lneutr≡ is just refl because we defined trans this way but the remaining ones are less trivial but provable using J. For example rneutr≡:

```
rneutr≡ : {a b : A} (p : a ≡ b) → trans p refl ≡ p
rneutr≡ = J (λ {a} {b} p → trans p refl ≡ p) refl
```

I leave it as an exercise to prove the remaining ones. Actually it is easy to see since it is sufficient to replace p with refl which we can do using J since both ends of the equality p are variables, namely a and b.

Now Hofmann's and Streicher's main idea was that we can turn this around: not only does the equality type with only J gives rise to a groupoid but actually the laws of a groupoid are sufficient to derive J. That is we can model each type as an arbitrary groupoid and with this interpretation we can derive J. Naturally there are some technical details one needs to work out, in particular what is a model of type theory in terms of groupoids.

However, there is a further complication: the equality within the groupoid of equality types itself, that is equations like trans p refl ≡ p themselves do not in general satisfy uip. So indeed, we don't have just a groupoid but a much more complicated beast called a ω-groupoid. Hence, Hofmann's and Streicher's groupoid model while sound is not complete in the sense that it entails uip for equalities of equality proofs and this is not provable in Type Theory.

9.3.3 Paulin-Mohring's eliminator

The eliminator J can only by used to define functions of a type of the form

```
{x y : A} (p : x ≡ y) → C {x} {y} p
```

It doesn't work when any side of the equality is fixed. We cannot do much if both sides are fixed because this would means we can derive K but the situation is different if only one side is fixed as in

```
{y : A} (p : a ≡ y) → C {y} p
```

for some fixed a : A.

Indeed, we can understand this as a different definition of equality: we fix a : A and we define the predicate of being equal to a. We can write this down as follows

```
data _≡'_ (a : A) : A → Set where
  refl : a ≡' a
```

Here a : A is a parameter we could as well have assumed it globally.

We note that _≡'_ looks very similar to _≡_: it has the same type A → A → Set and the constructor refl has the same type a ≡ a and a ≡' a. However, the eliminator is different:

```
J' : (a : A)
  (M : {y : A} → a ≡' y → Set)
  → (M {a} refl)
  → {y : A} (p : a ≡' y) → M {y} p
J' a M m-refl refl = m-refl
```

If we want to compare the two types it makes sense to check wether their respective eliminators are derivable. One direction is easy, we can derive J for _≡'_ from J'.

```
J-from-J' : (M : {a b : A} → a ≡' b → Set)
  → ({a : A} → M {a} {a} refl)
  → {a b : A} (p : a ≡' b) → M {a} {b} p
J-from-J' M m-refl {a} = J' a (M {a}) (m-refl {a})
```

All we needed to do is to specialise the motive M and the method m-refl. What about the other direction?

```
J'-from-J : (a : A)
  (M : {y : A} → a ≡ y → Set)
  → (M {a} refl)
  → {y : A} (p : a ≡ y) → M {y} p
```

This is less obvious because we cannot just cook up motive for J' from a motive for M. But what about the iterator? Actually we have already seen the recursor for _≡'_

```
It≡' : (a : A) (M : A → Set) → M a → {y : A} → (a ≡ y) → M y
```

Indeed, if we just commute the parameters, we get a well known aquaintance

```
subst-it : (P : A → Set) → {a b : A} → a ≡ b → P a → P b
```

which is indeed derivable from It≡

```
subst-it P = It≡ (λ x y → P x → P y) (λ q → q)
```

The other ingredient in the derivation of J' is an important principle which is called *contractibility of singletons*. Given $a : A$ the singleton represents the set of elements that is equal to a, we could write $\{y : A \mid a \equiv y\}$ that is using Σ: Σ A (λ y → a ≡ y). Now there is always at least one element in the singleton type, namely (a , refl), but the important point is that this is the only one.

```
contr-sgl : (a : A) → (p : Σ A (λ y → a ≡ y))
  → (a , refl) ≡ p
```

That this is provable by J becomes obvious once we uncurry the Σ-type

```
contr-sgl : {a y : A} → (q : a ≡ y)
  → _≡_ {A = Σ A (λ y → a ≡ y)} (a , refl) (y , q)
contr-sgl {A} =
J (λ {a} {y} q → _≡_ {A = Σ A (λ z → a ≡ z)} (a , refl) (y , q))
  refl
```

How does this help us to derive J'? Now if you look at the type of J' it says that once we know M {a} refl we know M {y} p for any y : A and p : a ≡ y. Hell yes, all we need to do is to combine subst and contr-sgl.

```
J'-from-J {A} a M m-refl p =
  subst M' (contr-sgl p) m-refl
```

```
  where M' : (Σ A (λ y → a ≡ y)) → Set
        M' (y , q) = M {y} q
```

Hence the two versions of equality are indeed equivalent, but the eliminator which comes from _≡'_ is more versatile because it allows one side of the equation to be fixed. This was noticed first by Christine Paulin-Mohring and is the standard definition of equality in the Coq system.

9.4 Coinduction for streams

We would hope that we could just hold up the categorical mirror and derive a dual version of the eliminator as we have done with coiterators and corecursors. However, the story is not as nice as we may have hoped but reveal a deep asymmetry in the kind of Type Theory we have studied so far.

Let's look at the streams again. Here is the definition from section 4.5.1:

```
record Stream (A : Set) : Set where
  constructor _::_
  coinductive
  field
    head : A
    tail : Stream A
```

The coiterator is defined as follows:

```
ColtStream : {W : Set} → (W → A) → (W → W)
  → W → Stream A
head (ColtStream h t x) = h x
tail (ColtStream h t x) = ColtStream h t (t x)
```

While the eliminator simply arose as a dependent version of the iterator / recursor this will not work in this case because the coinductive type is the output of the iterator.

The eliminator expressed the fact that all elements of an inductive type are created from the constructors. In the dual situation whatever we can do with a stream has to happen through the destructors, we cannot observe the actual definition of the stream using the coiterator. Hence we going to define identify two streams which are *bisimilar*, i.e. which behave the same wrt the destructors.

We use the notion of a *bisimulation* which is a relation, in this case on streams, that expresses that we can identify elements. A relation

```
_~_ : Stream A → Stream A → Set
```

is a bisimulation if related streams s ∼ t have equal heads head s ≡ head t and related tails tail s ∼ tail t. The principle of coinduction says that elements that are bisimlar, that is there is a bisimulation that relates them are actually equal. We can make this precise as follows:

```
postulate
  CoIndStream : (_~_ : Stream A → Stream A → Set)
    → ({s t : Stream A} → s ∼ t → head s ≡ head t)
    → ({s t : Stream A} → s ∼ t → tail s ∼ tail t)
    → {s t : Stream A} → s ∼ t → s ≡ t
```

We can use CoIndStream to prove equations about streams. We had previously defined

```
from : ℕ → Stream ℕ
head (from n) = n
tail (from n) = from (suc n)
```

and we now want to prove the following theorem: if we map suc to the stream of numbers starting with n, i.e. from n than this is the same stream as the stream starting with suc n, i.e. from (suc n)

To formally state the theorem, we need first that Stream is functorial, i.e. there is a map function. This is easy to define using copattern matching:

```
mapStream : (A → B) → Stream A → Stream B
head (mapStream f s) = f (head s)
tail (mapStream f s) = mapStream f (tail s)
```

I leave it as an exercise to derive mapStream from the coiterator. Now we can state the theorem formally:

```
thm : (n : ℕ) → from (suc n) ≡ mapStream suc (from n)
```

We are going to use coinduction to prove thm. In this instance we can read off the relation directly from the theorem:

```
data _~_ : Stream ℕ → Stream ℕ → Set where
  ~-in : (n : ℕ) → from (suc n) ∼ mapStream suc (from n)
```

We need to show that _~_ is indeed a bisimulation. This turns out to be very straightforward. First of all we note that the heads agree, that is

```
hd (from (suc n)) = hd (suc n :: from (suc (suc n)))
                  = suc n
hd (mapStream suc (from n)) = hd (mapStream suc (n :: from (suc n)))
                            = hd (suc n :: mapStream (from (suc n))))
                            = suc n
```

Hence the agda proof is just

```
hd~ : {s t : Stream ℕ} → s ∼ t → head s ≡ head t
hd~ (~-in n) = refl
```

Next we need to show that if elements are related then their tails are also related. Ok, let's analyze the tails:

```
tl (from (suc n))          = tl (suc n :: from (suc (suc n)))
                           = from (suc (suc n))
tl (mapStream suc (from n)) = tl (mapStream suc (n :: from (suc n)))
                           = tl (suc n :: mapStream (from (suc n))))
                           = mapStream (from (suc n))
```

Hence as we see the tails are an instance of the _~_ just by replacing n with suc n. Here is the agda code:

```
tl~ : {s t : Stream ℕ} → s ∼ t → tail s ∼ tail t
tl~ (~-in n) = ~-in (suc n)
```

Hence putting everything together we can now prove thm:

```
thm n = CoIndStream _~_ hd~ tl~ (~-in n)
```

I have added CoIndStream as a postulate because we cannot prove it. The reason is that we can only prove equalities for objects which are actually identical. However, the theorem entails that for example

$$\mathsf{from}\ 1 \equiv \mathsf{mapStream}\ \mathsf{suc}\ (\mathsf{from}\ 0)$$

and given the definition of _≡_ we should be able to prove this using refl. However, we cannot because from 1 and mapStream suc (from 0) are not identical, they do not compute to the same thing — indeed they do not compute at all.

This is a defect of the Type Theory we have introduced so far, which is also called *Intensional Type Theory*. So while the streams given by the two expressions are *extensionally* equal (they are the same infinite sequences), they are not *intensionally* equal, i.e. they are given by different definitions.

Another instance of this phenomena is the lack of functional extensionality, which is the following principle:

```
postulate
  funext : {f g : A → B} → (x : A) → f x ≡ g x → f ≡ g
```

funext is basically coinduction for functions and it suffers from the same problem: consider f,g : ℕ → ℕ with f x = x + 0 and g x = 0 + x. Using rneutr+ we can show that (x : ℕ) → f x ≡ g x but we cannot prove f ≡ g because we cannot show this using refl which is the only constructor for equality.

Yes, we can just postulate Coind and funext but this is not only unsatisfying it also destroys the computational behaviour of type theory. We are thinking of natural numbers as generated from 0 and suc but with postulates present we may have a number which cannot be reduced to a numeral because somewhere it contains an instance of one of the postulates.

We will fix this problem later, when we are going to use a different definition of equality in cubical type theory.

9.5 History

Eliminators were used by Per Martin-Löf, examples can be found in the book with Sambin [MLS84]. However, the equality eliminator is missing from this book because Martin-Löf used equality reflection at this point — this is referred to as extensional type theory. J is presented under the name eqpeel in [NPS90]. Early examples of eliminators can be found in [BCMS89] and a systematic account of eliminators for inductive families is in [Dyb94].

The combinator K as an extension of J was introduced by Thomas Streicher in a letter to Per Martin-Löf in 1996 raising the question wether K is derivable from J. This was answered to the negative by the construction of the groupoid model [HS98].

A systematic account of dependent pattern matching and the fact that it can be derived from the eliminators including K can be found in Conor McBride's PhD thesis [McB00b].

Coinduction and bisimulation are well known techniques using in the context of labelled transtion systems [San11]. The coinduction principle for streams we have presented can be found in [JR97].

9.6 Exercises

1. Implement decidability of equality for natural numbers using only E$\mathbb{N}$.

   ```
   _≡ℕ?-e_ : (m n : ℕ) → Dec (m ≡ n)
   ```

   You also need the recursor for Dec:

   ```
   RDec : (A → M) → (¬ A → M) → Dec A → M
   RDec m-yes m-no (yes x) = m-yes x
   RDec m-yes m-no (no x) = m-no x
   ```

2. Derive eliminators for the inductive datatypes from section 4.3: List , Expr and Ord. I am intentionally not including RoseTree.

3. Prove that equality satisfies all the laws of a groupoid using only J

   ```
   lneutr≡ : {a b : A} (p : a ≡ b) → trans refl p ≡ p
   rneutr≡ : {a b : A} (p : a ≡ b) → trans p refl ≡ p
   assoc≡ : {a b c d : A} (p : a ≡ b) (q : b ≡ c) (r : c ≡ d)
     → trans p (trans q r) ≡ trans (trans p q) r
   linv≡ : {a b : A} (p : a ≡ b) → trans (sym p) p ≡ refl
   rinv≡ : {a b : A} (p : a ≡ b) → trans p (sym p) ≡ refl
   ```

4. Show that $\mathbb{N}\infty$ with _+∞_ and zero∞ is a commutative monad, i.e. show

   ```
   rneutr+∞ : {m : ℕ∞} → m +∞ zero∞ ≡ m
   lneutr+∞ : {m : ℕ∞} → zero∞ +∞ m ≡ m
   ```

```
assoc+∞ : {l m n : ℕ∞} → (l +∞ m) +∞ n ≡ l +∞ (m +∞ n)
comm+∞ : {m n : ℕ∞} → m +∞ n ≡ n +∞ m
```

using coinduction for natural numbers. To formulate CoInd$\mathbb{N}\infty$ we need an operation to lift a relation over Maybe

```
data MaybeR (_~_ : A → A → prop) :
  Maybe A → Maybe A → prop where
  nothing : MaybeR _~_ nothing nothing
  just : {m' n' : A} → m' ∼ n'
    → MaybeR _~_ (just m') (just n')
```

Now a bisimulation is a relation which is stable under the destsructor using MaybeR. Hence we arrive at the following coinduction principle which has to be postulated as usual.

```
postulate
  CoInd ℕ∞ : (_~_ : ℕ∞ → ℕ∞ → Set)
    → ({m n : ℕ∞} → m ∼ n
      → MaybeR _~_ (pred∞ m) (pred∞ n))
    → {m n : ℕ∞} → m ∼ n → m ≡ n
```

To solve this exercise you need some agda tools we haven't yet introduced:

The inspect pattern When we pattern match over an expression is in

```
f ... with pred∞ n
f ... | nothing = ?
f ... | just n' = ?
```

we don't know that pred∞ n ≡ nothing respectively pred∞ ≡ just n' which we will need in the proof. To address this we can use the inspect-pattern which uses the following type:

```
data Singleton {A : Set} (x : A) : Set where
  _with≡_ : (y : A) → x ≡ y → Singleton x
```

and inspect is a function which produces elements of Singleton:

```
inspect : ∀ {A : Set} (x : A) → Singleton x
inspect x = x with≡ refl
```

Now we can use inspect in the **with**-clause above:

```
f ... with inspect (pred∞ n)
f ... | nothing with≡ nothing-eq = ?
f ... | just m' with≡ just-eq    = ?
```

Now nothing-eq : pred∞ n ≡ nothing and just-eq : pred∞ ≡ just n' provide the missing equations.

The rewrite clause To get expressions using with unstuck you need to use rewrite. An example is to derive the defining equations for _+∞_:

```
eq+∞-nothing : {x y : ℕ∞} → pred∞ x ≡ nothing
  → pred∞ (x +∞ y) ≡ pred∞ y
eq+∞-nothing p rewrite p = refl
eq+∞-just : {x y x' : ℕ∞} → pred∞ x ≡ just x'
  → pred∞ (x +∞ y) ≡ just (x' +∞ y)
eq+∞-just p rewrite p = refl
```

This is also handy in conjunction with the inspect-pattern, e.g.

```
f ... with inspect (pred∞ n)
f ... | nothing with≡ nothing-eq rewrite nothing-eq = ?
f ... | just m' with≡ just-eq    rewrite just-eq    = ?
```

Chapter 10

Universes and paradoxes

We have introduced the type of sets Set (should really be called Type), it's elements are types like $\mathbb{N}$: Set but what is the type of Set itself? The obvious naive answer is Set that is Set : Set but this rings a bell. Wasn't there a problem with this?

Indeed, when Frege was developing the foundations of Mathematics based on naive set theory he also assumed that there is a set of all sets. The british mathematican and philosopher Russell saw this and sent him a letter explaining that this assumption is inconsistent. This is what we now call *Russell's paradox*.

Do we have Russell's paradox in Type Theory? One may first think maybe not since we cannot talk about elements in Type Theory and this seems to be an essential part of Russell's paradox. Maybe this was the reason that Per Martin-Löf included this assumption in an early version of Type Theory. In his case it was the French logician Jean-Yves Girard who pointed out that this is inconsistent because he had studied a type-theoretic system, System U, that was also inconsistent and which was actually weaker than Type : Type. This paradox is now known as Girard's paradox.

However, it turns out that using inductive types we can directly encoded a variant of Russell's paradox in Type Theory, this was called the paradox of trees by Thierry Coquand.

10.1 The paradox of trees

Sets in the sense of set theory are trees where the order and multiplicity of subtrees doesn't matter. It turns out that to implement Russell's paradox we can just use ordinary trees we don't need to quotient them.

Hence we come up with the following definition of trees:

```
data Tree : Set where
  node : (I : Set) → (I → Tree) → Tree
```

Now, agda rejects this definitions because it knows that we are doing something naughty. It's size checker says that we cannot define a Set whose constructor has a Set as an argument. As we will see it is perfectly correct but since our plan is to derive a paradox we override this by adding the pragma NO_UNIVERSE_CHECK indicating that we are intentionally naughty.

We can play set theory with Tree. First of all the empty tree ([]), corresponding to $\{\}$

```
[] : Tree
[] = node ⊥ λ ()
```

Now we can define the tree whose only subtree is the empty tree ([[]]) corresponding to $\{\{\}\}$:

```
[[]] : Tree
[[]] = node ⊤ (λ _ → [])
```

and the tree containing the two previous trees [[],[[]]] corresponding to $\{\{\}, \{\{\}\}$:

```
[[],[[]]] : Tree
[[],[[]]] = node Bool (λ x → if x then [] else [[]])
```

We can also encode von Neumann's definition of natural numbers, where $n = 0, 1, \ldots, n-1$:

```
ℕ-tree : Tree
ℕ-tree = node ℕ λ n → node (Fin n) (fin-trees {n})
  where fin-trees : {n : ℕ} → Fin n → Tree
    fin-trees {zero} ()
    fin-trees {suc n} zero = node (Fin n) (fin-trees {n})
    fin-trees {suc n} (suc j) = fin-trees {n} j
```

Indeed we can define the _∈_ relation on trees, where t ∈ u expresses that t is a direct subtree of u:

```
data _∈_ : Tree → Tree → Set where
  ∈-in : {I : Set} (f : I → Tree) (i : I) → f i ∈ node I f
```

The main idea of Russell's paradox is that we define a good tree to be a tree that doesn't contain itseelf and then show that the tree of all good trees is good iff it is not good:

```
Bad : Tree → Set
Bad t = t ∈ t

Good : Tree → Set
Good t = ¬ (Bad t)
```

We call the tree of all good trees Russell:

```
Russell : Tree
Russell = node (Σ[ t ∈ Tree ] Good t) proj₁
```

Now we can show that Russell is good iff it is bad

```
strange : Good Russell ⇔ Bad Russell
proj₁ strange h = ∈-in proj₁ (Russell , h)
proj₂ strange (∈-in ∘ (proj₁) (.Russell , good)) = good
```

Because if Russell is good then it must contain itself hence it is bad but if it is bad, then it contains itself and hence it must be good, because Russell only contains good trees.

Now we have a situation that P ⇔ ¬ P and indeed this was an exercise of propositional logic to show that this is inconsistent:

```
ex : ¬ (¬ P ⇔ P)
ex {P} (npp , pnp) = pnp p p
  where np : ¬ P
        np p = pnp p p
        p : P
        p = npp np
```

Combining strange and ex we can prove ⊥:

```
paradox : ⊥
paradox = ex strange
```

Clearly, this shows that our assumption that Set : Set is unsound.

Bibliography

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. A user's guide to alf. *Chalmers University of Technology, Sweden*, 1994.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[Alt94] Thorsten Altenkirch. Programming + verification = progification. available from `http://www.cs.nott.ac.uk/~txa/publ/progify94.pdf`, 1994.

[Alt96] Thorsten Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. available from `http://www.cs.nott.ac.uk/~txa/publ/esslli96.pdf`.

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[AP13] Andreas M Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. *ACM SIGPLAN Notices*, 48(9):185–196, 2013.

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *International School on Advanced Functional Programming*, pages 240–267. Springer, 1998.

[B$^+$84] Hendrik P Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[BCMS89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.

[Bet48] Evert W Beth. Semantical considerations on intuitionistic mathematics. 1948.

[BMS80] Rod M Burstall, David B MacQueen, and Donald T Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143, 1980.

[Bra17] Edwin Brady. *Type-driven development with Idris*. Manning Publications Company, 2017.

[BS91] U Berger and H Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE, 1991.

[CH06] Felice Cardone and J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817, 2006.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[Coq93] Thierry Coquand. Infinite objects in type theory. In *International Workshop on Types for Proofs and Programs*, pages 62–78. Springer, 1993.

[Cur30] Haskell Brooks Curry. Grundlagen der kombinatorischen logik. *American journal of mathematics*, 52(4):789–834, 1930.

[dt20] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2020+.

[Dyb94] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

[Fre79] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Nebert, 1879.

[G+95] PRL Group et al. Implementing mathematics with the nuprl proof development system. *Computer Science Department, Cornell University*, 1995.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de lâĂŹarithmétique dâĂŹordre supérieur*. PhD thesis, PhD thesis, PhD thesis, Universit e Paris VII, 1972.

[GMM+78] Mike Gordon, Robin Milner, Lockwood Morris, Malcolm Newey, and Christopher Wadsworth. A metalanguage for interactive proof in lcf. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130, 1978.

[Göd58] Von Kurt Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *dialectica*, 12(3-4):280–287, 1958.

[HA99] David Hilbert and Wilhelm Ackermann. *Principles of mathematical logic*, volume 69. American Mathematical Soc., 1999.

[How80] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 1998.

[Hut16] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2016. 2nd edition.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

[Kri59] Saul A Kripke. A completeness theorem in modal logic. *The journal of symbolic logic*, 24(1):1–14, 1959.

[Kur29] Gödel Kurt. Über die vollständigkeit des logikkalküls. *PhD thesis*, 1929.

[LP92] Zhaohui Luo and Robert Pollack. *LEGO proof development system: User's manual*. LFCS, Department of Computer Science, University of Edinburgh, 1992.

[McB00a] Conor McBride. Dependently typed functional programs and their proofs. 2000.

[McB00b] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, UK, 2000.

[McB04] Conor McBride. Epigram: Practical programming with dependent types. In *International School on Advanced Functional Programming*, pages 130–170. Springer, 2004.

[McB09] Conor McBride. Winging it. slides from a talk at WG2.8 #26, Frauenchiemsee, 2009. the orginal slides are about 10 years older.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[ML78] S Mac Lane. Categories for the working mathematician. *Springer*, 54:55–81, 1978.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of functional programming*, 14(1):69–111, 2004.

[Nor08] Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[Ros50] PC Rosenbloom. The elements of mathematical logic, new york, 1950.

[San11] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[Sch24] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische annalen*, 92(3-4):305–316, 1924.

[Sch68] Kurt Schütte. *Vollständige Systeme modaler und intuitionistischer Logik*, volume 42. Springer-Verlag, 1968.

[TvD88a] Anne Sjerp Troelstra and Dirk van Dalen. Constructivism in mathematics: An introduction. vol. ii, vol. 123 of. *Studies in Logic and the Foundations of Mathematics*, 1988.

[TvD88b] Anne Sjerp Troelstra and Dirk van Dalen. Constructivism in mathematics. vol. i, volume 121 of. *Studies in Logic and the Foundations of Mathematics*, 26, 1988.

[TvU99] AS Troelstra and P van Ulsen. The discovery of ew beth's semantics for intuitionistic logic. *Essays dedicated to Johan van Benthem on the occasion of his 50th birthday. University of Amsterdam, Amsterdam*, 1999.

[VD99] Dirk Van Dalen. *Mystic, geometer, and intuitionist: The life of LEJ Brouwer*. Oxford: Clarendon Press, 1999.