Mathematics for Computer Scientists (G51MCS)
Lecture notes
Autumn 2005

Thorsten Altenkirch

October 26, 2005

Contents

1 Types and sets

What is a **set**? Wikipedia [1] says

> Informally, a set is just a collection of objects considered as a whole.

And what is a **type**? Wikipedia offers different definitions for different subject areas, but if we look up datatype in Computer Science we learn:

> In computer science, a datatype (often simply type) is a name or label for a set of values and some operations which can be performed on that set of values.

Here the meaning of type refers back to the meaning of sets, but it is hard to see a substantial difference. We may conclude that what is called a set in Mathematics is called a type in Computer Science.

Here, we will adopt the computer science terminology and use the term *type*. But remember to translate this to *set* when talking to a Mathematician.

There is a good reason for this choice: I am going to use the type system of the programming language Haskell to illustrate many important constructions on types and to implement mathematical constructions on a computer. Please note that this course is **not** an introduction to Haskell, this is done in the 2nd semester in G51FUN. If you want to find out more about Haskell **now**, I refer to Graham Hutton's excellent, forthcoming book, whose first seven chapters are available from his web page [2] . The book is going to be published soon and is the course text for G51FUN. Another good source for information about Haskell is the Haskell home page [3]

While we are going to use Haskell to illustrate mathematical concepts, Haskell is also a very useful language for lots of other things, from designing user interfaces to controlling robots.

1.1 Examples of types

I am sure there are some types you already know since primary school. E.g. the type **Nat** of natural numbers, which are the numbers we use for counting. In Computer Science we include the 0 to allow for the case that there is nothing to count. We can start enumerating the elements of **Nat**

$$\mathbf{Nat} = \{0, 1, 2, 3, 4, 5, \ldots\}$$

We follow the mathematical tradition to use curly brackets when enumerating the elements of a type. Clearly, this is not a precise definition because we have used ... to indicate that we can go on forever. We will fix this later. Most Mathematics books will use $\mathbb{N}$ for **Nat** but in Computer Science we prefer using names made up from several letters (because we can type them in on the keyboard) while the mathematicians like single letters but use different alphabets (that's the reason that they use so many Greek symbols).

[1] http://en.wikipedia.org/ is a free, online, user extensible encyclopedia.
[2] http://www.cs.nott.ac.uk/˜gmh/book.html
[3] http://www.haskell.org/

We write $2 \in$ **Nat** to express that 2 is an **element** of **Nat**, we may also say that 2 has type **Nat**. [4]

If you have a bank account or a credit card you will also know the type of integers, **Int** which extends **Nat** by negative numbers - in Maths one writes $\mathbb{Z}$. Using the suggestive ... again we write

$$\mathbf{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

Other types of numbers are the rational numbers **Rat** ($\mathbb{Q}$) which is the type of fractions (e.g. $2 \,/\, 3 \in$ **Rat**), the type of real numbers **Real** ($\mathbb{R}$) (e.g. $\sqrt{2} \in$ **Real** and $\pi \in$ **Real**) and the type of complex numbers **Compl** ($\mathbb{C}$) (e.g. $\sqrt{-1} \in$ **Compl**). [5] While the latter two (**Real**,**Compl**) are a central topic in Mathematics they are less essential in Theoretical Computer Science, but relevant in many application areas.

We say a type is **finite**, if we can enumerate it without using "...". An important finite type is the type **Bool** of booleans or truth values:

$$\mathbf{Bool} = \{\mathrm{True}, \mathrm{False}\}$$

We can define new finite types by using a collection of names, different from each other (these are called *constructors*). To define a type of basic colours in Haskell [6] we would write

```
data Colour = Red | Green | Yellow
```

We can enumerate **Colour**

$$\mathbf{Colour} = \{\mathrm{Red}, \mathrm{Green}, \mathrm{Yellow}\}$$

In an an enumeration the order doesn't matter, e.g. the following are alternative enumerations of **Colour**

$$\mathbf{Colour} = \{\mathrm{Green}, \mathrm{Yellow}, \mathrm{Red}\}$$
$$\mathbf{Colour} = \{\mathrm{Yellow}, \mathrm{Red}, \mathrm{Green}\}$$

we may also repeat elements

$$\mathbf{Colour} = \{\mathrm{Red}, \mathrm{Green}, \mathrm{Yellow}, \mathrm{Red}\}$$

but we usually try to avoid this. You may have already noticed that we use capitalized words for names (of types or constructors) — this is a Haskell convention. We shall use names starting with small letters for variables.

There is also the extreme case of a finite type, the empty type $\emptyset$, which has no elements. It is easy to enumerate $\emptyset$:

$$\emptyset = \{\,\}$$

[4]In Haskell we type **::** instead of $\in$, I am using lhs2Tex and latex to pretty-print Haskell programs.

[5]As most programming languages Haskell isn't very good with numbers. There is no predefined **Nat** and the type **Int** actually stands for 32 bit integers, however, there is a type **Integer** of integers of arbitrary size (if you have got enough memory). The standard prelude doesn't define **Rat**,**Real** or **Compl** but **Float** and **Double** which are actually an approximation of **Rat**. However there are standard libraries for rational numbers (introducing a type **Rational**) and complex numbers (featuring the types **Complex Float** and **Complex Double**).

[6]Actual Haskell code is displayed in a framed box. On the course page you find a link notes.hs, which contains all the Haskell code from these notes.

1.2 Making new types from old

We can construct a new type by combining information, e.g. a card in a card game may be described by combining it's face and it's suit. E.g. we define

```
data Suit = Diamonds | Hearts | Spades | Clubs
data Face = Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten
          | Jack | Queen | King | Ace
```

and now we say that a card is a **pair** (*face*, *suit*) (also called tuple) with *face* $\in$ **Face** and *suit* $\in$ **Suit**, we write

```
type Card = Suit × Face
```

We can enumerate **Card**:

$$\textbf{Card} = \{\,(\text{Diamonds}, \text{Two}), (\text{Diamonds}, \text{Three}), (\text{Diamonds}, \text{Four}), (\text{Diamonds}, \text{Five}), (\text{Diamonds}, \text{Six}), (\text{Diamonds}, \text{Seven}), (\text{Diamonds}, \text{Eight}), (\text{Diamonds}, \text{Nine}), (\text{Diamonds}, \text{Ten}), (\text{Diamonds}, \text{Jack}), (\text{Diamonds}, \text{Queen}), (\text{Diamonds}, \text{King}), (\text{Diamonds}, \text{Ace}), (\text{Hearts}, \text{Two}), (\text{Hearts}, \text{Three}), (\text{Hearts}, \text{Four}), (\text{Hearts}, \text{Five}), (\text{Hearts}, \text{Six}), (\text{Hearts}, \text{Seven}), (\text{Hearts}, \text{Eight}), (\text{Hearts}, \text{Nine}), (\text{Hearts}, \text{Ten}), (\text{Hearts}, \text{Jack}), (\text{Hearts}, \text{Queen}), (\text{Hearts}, \text{King}), (\text{Hearts}, \text{Ace}), (\text{Spades}, \text{Two}), (\text{Spades}, \text{Three}), (\text{Spades}, \text{Four}), (\text{Spades}, \text{Five}), (\text{Spades}, \text{Six}), (\text{Spades}, \text{Seven}), (\text{Spades}, \text{Eight}), (\text{Spades}, \text{Nine}), (\text{Spades}, \text{Ten}), (\text{Spades}, \text{Jack}), (\text{Spades}, \text{Queen}), (\text{Spades}, \text{King}), (\text{Spades}, \text{Ace}), (\text{Clubs}, \text{Two}), (\text{Clubs}, \text{Three}), (\text{Clubs}, \text{Four}), (\text{Clubs}, \text{Five}), (\text{Clubs}, \text{Six}), (\text{Clubs}, \text{Seven}), (\text{Clubs}, \text{Eight}), (\text{Clubs}, \text{Nine}), (\text{Clubs}, \text{Ten}), (\text{Clubs}, \text{Jack}), (\text{Clubs}, \text{Queen}), (\text{Clubs}, \text{King}), (\text{Clubs}, \text{Ace})\,\}$$

We call **Face** $\times$ **Suit** the **cartesian product** of **Face** and **Suit**. Pairs are ordered, e.g. (Queen, Diamonds) is not the same as (Diamonds, Queen). Indeed, they have different types (Queen, Diamonds) $\in$ **Face**$\times$**Suit** while (Diamonds, Queen) $\in$ **Suit** $\times$ **Face**. But even if the types are the same as in **Bool** $\times$ **Bool** we make a difference between (True, False) and (False, True).

Have you noticed? If a is a finite type with n elements and b is a finite type with m elements, how many elements does (a, b) (or $a \times b$) have? Precisely!

Is there also + for types? Yes there is, and indeed we write $a + b$, where a and b stand for types, not numbers. An element of $a + b$ is either Left x where $x \in a$ or Right y where $y \in b$. This can be defined by

```
data a + b = Left a | Right b
```

Here Left,Right are parametrized constructors, a and b are *type variables.* This operation is called the *disjoint union* and is often written as $\uplus$.

As an example for the use of $\cdot + \cdot$, imagine there is shop which sells socks which are either small, medium or large and ties which are striped or plain. We define

```
data Sock = Small | Medium | Large
data Tie = Striped | Plain
```

Then the type of articles is

```
type Article = Sock + Tie
```

We can enumerate **Article**

$$\begin{aligned}\textbf{Article} = \{&\text{Left Small}, \text{Left Medium}, \text{Left Large},\\ &\text{Right Striped}, \text{Right Plain}\}\end{aligned}$$

Given types a and b we write $a \to b$ for the type of functions, which assign an element of b to every element of a. We call a the *domain* and b the *codomain* or range of the function. A simple example of a function is *isRed*, which returns True if its input is Red and False otherwise.

$$\boxed{\begin{aligned}&\mathit{isRed} \in \textbf{Colour} \to \textbf{Bool}\\ &\mathit{isRed}\ \text{Red} = \text{True}\\ &\mathit{isRed}\ \text{Green} = \text{False}\\ &\mathit{isRed}\ \text{Yellow} = \text{False}\end{aligned}}$$

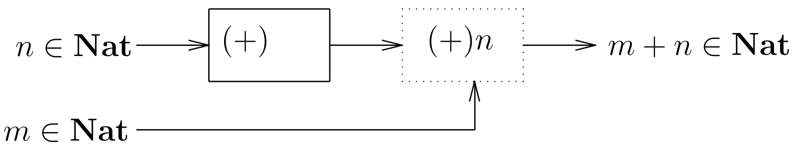
To use a function we apply it to an element of the domain, e.g. we just write *isRed* Green, that is we are not using extra parentheses as it is common in Mathematics.

We may view a function as a machine, which we feed elements of the domain and which spits out elements of the codomain. Our view of functions is *extensional*, i.e. we are not allowed to look into the machine to see how it produces its answer. While I amusing the idea of a machine, don't forget that a function is a methematical object, like a number it is independent of space and time, the output of a function only depends on its input. Unlike *procedures* or *methods* in imperative languages, like Java or Basic, functions don't have a state.

$$\begin{tikzcd}\textbf{Colour} \arrow[r] & \boxed{\text{isRed}} \arrow[r] & \textbf{Bool}\end{tikzcd}$$

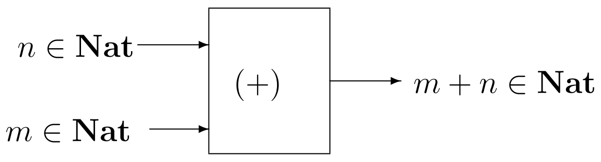
What is the type of a function with two arguments, e.g. $+$ in arithmetic? One possibility is to say that the domain of $+$ are pairs of numbers, i.e. that it has the type $(+) \in (\textbf{Nat}, \textbf{Nat}) \to \textbf{Nat}$ [7]. However, an alternative, which is adopted in Haskell, is to use the *curried* [8] form of the function, i.e. $(+) \in \textbf{Nat} \to (\textbf{Nat} \to \textbf{Nat})$, that is $(+)$ applied to one argument, e.g. 5, returns a function $(+)\ 5 : \textbf{Nat} \to \textbf{Nat}$ — the function which adds 5 to its argument. We can apply this function to another number, e.g. $((+)\ 5)\ 3$ which returns 8. To save parentheses we adopt the convention that $\to$ *associates to the right*, that is that $\textbf{Nat} \to \textbf{Nat} \to \textbf{Nat}$ stands for $\textbf{Nat} \to (\textbf{Nat} \to \textbf{Nat})$ and that function application associates to the left, that is that $(+)\ 5\ 3$ stands for $((+)\ 5)\ 3$ and finally that since $+$ is an infix operator that $5 + 3$ stands for $(+)\ 5\ 3$.

Here is a picture trying to illustrate this idea:



[7] We write $(+)$ because $+$ is usually used in an infix position, i.e. between its arguments.
[8] Curried functions are named after the logician Haskell Curry. The programming language Haskell is also named after him.

I have drawn the 2nd box with a dashed frame because this box is actually the output of the first box. Once, we have understood that this is how functions with seveal arguments are defined, we can simplify the picture by just drawing:



To summarize: the type of functions with two arguments one from type a and one from type b and with results in c can be either given as $(a, b) \to c$ (*uncurried form*) ar as $a \to b \to c$ (*curried form*).

If a is a finite type with n elements and b is a finite type with m elements, how many elements does $a \to b$ have? A good clue may come from the fact that in Mathematics one often uses an exponential notation b^a to denote funtion types.

1.3 Functions on Bool

Haskell's prelude [9] already defines a number of useful functions on **Bool** such as **and**:

$$
\begin{aligned}
&(\&\&) \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}\\
&\text{True}\,\&\& b = b\\
&\text{False}\&\& b = \text{False}
\end{aligned}
$$

How does this work? If the first argument, a, of $a\&\&b$ is True, then the value of $a\&\&b$ is equal to the second argument. If the first argument is False, then $a\&\&b$ is False, no matter what b is.

A function on finite types can be characterized by its value table (also called graph). In the case of && the value table is:

a	b	$a\&\&b$
False	False	False
False	True	False
True	False	False
True	True	True

Looking at the value table, we notice that && is symmetric but our definition was asymmetric. Indeed, we could instead define &&∗:

$$
\begin{aligned}
&(\&\&_1) \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}\\
&b\&\&_1\text{True} \;= b\\
&b\&\&_1\text{False} = \text{False}
\end{aligned}
$$

The value table of $\&\&_1$:

[9] This is the standard library which is automatically preloaded. Note that some of the definitions in this section below are not included in the appendix, because the functions are already defined in the prelude.

a	b	$a \&\&_1 b$
False	False	False
False	True	False
True	False	False
True	True	True

is the same as the one of &&. Since we cannot look into the definition of a function, because this is hidden in the black box,, to us && and $\&\&_1$ define the same function.

Here is the definition of *or*:

$$\boxed{\begin{array}{l} (||) \in \textbf{Bool} \rightarrow \textbf{Bool} \rightarrow \textbf{Bool} \\ \text{True} \, || b = \text{True} \\ \text{False} || b = b \end{array}}$$

and here its value table:

a	b	$a \, || \, b$
False	False	False
False	True	True
True	False	True
True	True	True

There is also an important function with one argument (*unary function*) **not**:

$$\boxed{\begin{array}{l} not \in \textbf{Bool} \rightarrow \textbf{Bool} \\ not \ \text{True} = \text{False} \\ not \ \text{False} = \text{True} \end{array}}$$

and its value table:

a	$not\ a$
False	True
True	False

We can combine existing functions directly to define new ones. An example is boolean equality:

$$\boxed{\begin{array}{l} (\equiv) \in \textbf{Bool} \rightarrow \textbf{Bool} \rightarrow \textbf{Bool} \\ a \equiv b = a \&\& b \mid not\ a \&\& not\ b \end{array}}$$

Note that Haskell uses different precedences for boolean operators, *not* binds more than & which binds more than ||. Hence Haskell (and we) read a & b || $not\ a$ & $not\ b$ as $(a$ & $b)$ || $((not\ a)$ & $(not\ b))$.

It is straightforward to calculate ≡'s value table:

a	b	$a \equiv b$
False	False	True
False	True	False
True	False	False
True	True	True

We have seen that ≡ is a function which can be defined by combining other functions. Could we have done this, with some other function, e.g. || ? Yes, consider the following definition of $||_1$ using && and *not*:

$$(||_1) \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$$
$$a ||_1 b = \mathit{not}\ ((\mathit{not}\ a) \&\& (\mathit{not}\ b))$$

Let's draw the value table of $||_1$

a	b	$a\ ||_1\ b$
False	False	False
False	True	True
True	False	True
True	True	True

and indeed it is identical to the one of $||$ and hence $(||) = (||_1)$.

Similarily we can define && using || and *not*:

$$(\&\&_2) \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$$
$$a \&\&_2 b = \mathit{not}\ ((\mathit{not}\ a) || (\mathit{not}\ b))$$

These definitions are based on the so called de Morgan laws, which say

$$\begin{aligned} \mathit{not}\ (a \&\& b) &= (\mathit{not}\ a)\ ||\ (\mathit{not}\ b) \\ \mathit{not}\ (a\ ||\ b) &= (\mathit{not}\ a) \&\& (\mathit{not}\ b) \end{aligned}$$

and the equation $\mathit{not}\ (\mathit{not}\ a) = a$.

Frequently, we want to analyze a boolean value somewhere in the middle of a function definition and calculate different results depending on whether the boolean is True or False. For this purpose we can define:

$$\mathit{ite} \in \mathbf{Bool} \to a \to a \to a$$
$$\mathit{ite}\ \mathrm{True}\ \ t\ e = t$$
$$\mathit{ite}\ \mathrm{False}\ t\ e = e$$

ite stands for *if-then-else*. Note that the type of *ite* is *polymorphic*, i.e. it works for any type replacing the type variable a.

Since *ite* is used so often, there is a special syntax for it:

$$\mathbf{if}\ c\ \mathbf{then}\ t\ \mathbf{else}\ e = \mathit{ite}\ c\ t\ e$$

As an example, we could have defined & using if-then-else:

$$(\&\&_3) \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$$
$$b \&\&_3 c = \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ \mathrm{False}$$

2 Basic logic

When we are expressing properties of mathematical objects or programs and reason about them we use the language of logic. We introduce here the symbols, which can be given a precise meaning, but explain them for the moment by their translation into English. Later in the course we will be more precise and introduce the rules of reasoning (called *natural deduction*) along with a computer program (called tutch) which can check that you have followed them.

A proposition is a precise statement which may be proven (then we believe it to be true) or disproven (then we believe it to be false). We assume some basic propositions as given, e.g. if $a, b\lambda\mathbf{in}\ A$ we can say $a = b$, meaning that a and b is equal. E.g. we know that $1 = 1$ is true but $1 = 2$ is false. Indeed, we accept this without further proof, both judgements are abvious.

There are propositions of which we don't know wether they are true or false, such as the proposition that there are infinitely many twin primes, that is pairs of prime numbers whose difference is 2 or in computer science whether the complexity classes P and NP are different (the $P = NP$ question).

2.1 Propositional connectives

A **logical connective** is a means to construct new propositions from given ones. We start with the basic propositional connectives, given propositions P and Q we can construct:

- $P \wedge Q$, stands for P **and** Q. We accept $P \wedge Q$ to be true, if we can show that both P and Q are true.
- $P \vee Q$, stands for P **or** Q. We accept that $P \vee Q$ to be true, if we can show at least one of P and Q. Note, that different to some uses of the word *or* in everyday language, $\vee$ is non-exclusive, e.g. $0 = 0 \vee 1 = 1$ is true.
- $P \implies Q$, stands for **If** P **then** Q, which we can also write as Q, **if** P. We accept $P \implies Q$ if we can show that Q is true from assuming that P is true.

We also introduce the propositional constants **True**, for an obviously true proposition (such as $0 = 0$) and **False** for an obviously false proposition (such as $0 = 1$). **True** and **False** are not really used in normal conversations but we may translate **True** by *fish can swim* or another blatantly obvious statement, and **False** by *pigs can fly* or another obviously false statement.

We can define new propositional connectives from the basic ones introduced above:

- $\neg P$, stands for **not** P. We define $\neg P$ as $P \implies$ **False**. Intuitively, we could say $\neg P$ means that if you believe P then you also believe that pigs can fly.
- $P \iff Q$, stands for P,**if and only if** Q. We define $P \iff Q$ as $(P \implies Q) \wedge (Q \implies P)$. $\iff$ introduces a notion of logical

equivalence and indeed if we know $P \iff Q$ we can always replace P by Q and vice versa.

To avoid having to write lots of brackets we introduce some conventions:

- $\neg$ binds stronger than $\wedge$, i.e. we read $\neg P \wedge Q$ as $(\neg P) \wedge Q$.
- $\wedge, \vee$ bind stronger than $\implies$, $\iff$, i.e. we read $P \wedge Q \implies R$ as $(P \wedge Q) \implies R$.
- $\implies$ associates to the right, i.e. we read $P \implies Q \implies R$ as $P \implies (Q \implies R)$

We could have introduced more conventions, e.g. that $\wedge$ binds stronger than $\vee$, but refrain from doing this and instead disambiguate by using brackets.

2.2 Tautologies and proof

A **propositional scheme** is a proposition containing propositional variables, e.g. $A \implies A$ and $A \implies A \wedge B$ are propositional scheme, where A, B stand for any proposition.

A **tautology** is a propositional scheme which is always provable no matter how we replace the variables by actual propositions. E.g. $A \implies A$ is a tautology but $A \implies A \wedge B$ isn't. We can see this by proving $A \implies A$: if we assume A then we know A hence $A \implies A$ holds. We can see that $A \implies A \wedge B$ is not a tautology by replacing A with **True** and B with **False** but **True** $\implies$ **True** $\wedge$ **False** cannot be provable, because we know **True** hence we would know **True** $\wedge$ **False** but this is only true, if **False** is, which it isn't (pigs can't fly).

Here are some examples of propositional tautologies:

1. **False** $\implies A$,
   this could be called the principle of *naivety*, if you believe that pigs can fly you believe everything.

2. $A \wedge B \implies A$,
   note that the converse $A \implies A \wedge B$ is not a tautology.

3. $A \implies A \vee B$,
   again note that the converse $A \vee B \implies A$ is not a tautology.

4. $(A \implies B) \implies (\neg B \implies \neg A)$.
   Let's prove this tautology using the explanation of the connectives given above:

   Proof: To show $(A \implies B) \implies (\neg B \implies \neg A)$, let's assume $A \implies B$ (1) to show $\neg B \implies \neg A$. To show this we assume $\neg B$ (2) (which just stands for $B \implies$ **False**) to show $\neg A$ which stands for $A \implies$ **False**, hence we also assume A (3) and it remains to show **False** from the assumptions (1),(2) and (3). Indeed, because we know A (3) and

$A \implies B$ (1) we know B and now we use (2) $B \implies$ **False** to show **False**. □

We notice that pigs can fly given the appropriate assumptions.

Many important tautologies have the form of logical equivalences:

1. $A \wedge B \iff B \wedge A$,
   we say that $\wedge$ is **commutative**. The same holds for $\vee$.

2. $A \wedge (B \wedge C) \iff (A \wedge B) \wedge C$,
   we say that $\wedge$ is **associative**. The same holds for $\vee$. Associativity allows us to omit brackets when we nest the same operator, i.e. we can read $A \wedge B \wedge C$ as $A \wedge (B \wedge C)$ or $(A \wedge B) \wedge C$ — it desn't matter.

3. $(A \vee B) \wedge C \iff (A \wedge C) \vee (B \wedge C)$,
   we say that $\vee$ **distributes** over $\wedge$. We prove this tautology:

   Proof: To show $\iff$ we have to show both directions:

   $(A \vee B) \wedge C \implies (A \wedge C) \vee (B \wedge C)$ We assume $(A \vee B) \wedge C$ (1) and try to show $(A \wedge C) \vee (B \wedge C)$. From (1) we know $(A \vee B)$ (2) and C (3). How could we have shown $A \vee B$? There are two cases:

   by showing A (4) From (3) and (4) we know $A \wedge C$ and hence $(A \wedge C) \vee (B \wedge C)$.

   by showing B $(4')$ From (3') and (4') we know $B \wedge C$ and hence $(A \wedge C) \vee (B \wedge C)$.

   Since in both cases we were able to show $(A \wedge C) \vee (B \wedge C)$ we have finished this part of the proof.

   $(A \wedge C) \vee (B \wedge C) \implies (A \vee B) \wedge C$ We assume $(A \wedge C) \vee (B \wedge C)$ (1) to show $(A \vee B) \wedge C$: How could we ve shown (1)? There are two possibilities:

   by showing $A \wedge C \wedge$ (2) Hence we know A (3) and C (34. From (3) we can derive $A \vee B$ and combining this with (4) we have $(A \vee B) \wedge C$.

   by showing $B \wedge C$ $(2')$ Hence we know B $(3')$ and C $(4')$. From (3') we can derive $A \vee B$ and combining this with (4') we have $(A \vee B) \wedge C$.

   In either case we have shown $(A \vee B) \wedge C$.

   □

2.3 Classical tautologies

Is $A \vee \neg A$ a tautology? Actually, it depends. Given the explanation of the connectives given above, to prove $A \vee \neg A$ we need to prove either A, or disprove it by establishing $\neg A$. However, there are propositions which we can't prove or disprove. On the other hand we are unable to exhibit an instance of $A \vee \neg A$, which will lead to a definitively false statement. We may adopt the point of view that even if we cannot prove a certain statement it is still either true and false. This point of view is accepted in classical logic, which is accepted by most modern Mathematicians, even though its philosophical foundations are rather problematic.

We say a tautology is a classical tautology, if it is provable from assuming that $A \vee \neg A$, the **principle of the excluded middle**, holds. An example is $\neg\neg B \implies B$, which can be proven as follows:

Proof: To show $\neg\neg B \implies B$ we assume $\neg\neg B$ (1) to show B. Using the principle of excluded middle we know $B \vee \neg B$. That is there are two possibilities:

B **holds.** we are done.

$\neg B$ **holds.** We note that (1) means $\neg B \implies$ **False**, hence we can use it to show **False** and since **False** implies anything, we can show B,

In either case we have shown B. $\Box$

However, at least for propositional logic there is a very mechanical way to find out whether a propositional scheme is a classical tautology: Since every proposition is either **True** or **False** we can identify propositions with **Bool** and replace the propositional connectives by the corresponding operations on **Bool**:

Propositions	**Bool**
$\wedge$	&&
$\vee$	||
$\neg$	*not*
$\iff$	$\equiv$
$\implies$	*implies*

where we define *implies* as follows:

$$\begin{aligned}&\mathit{implies} \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}\\ &\mathit{implies}\ \text{False}\ x = \text{True}\\ &\mathit{implies}\ \text{True}\ x = x\end{aligned}$$

To find out whether a propositional scheme is a classical tautology all we have to do is to draw the value table of its translation as a function on **Bool** (called its **truth table**) and check that it allways evaluates to **True**. E.g. to check that $\neg\neg A \implies A$ is a tautology, we just calculate:

A	$\neg A$	$\neg\neg A$	$\neg\neg A \implies A$
False	**True**	**False**	**True**
True	**False**	**True**	**True**

Let's verify that $(A \vee B) \wedge C \iff (A \wedge C) \vee (B \wedge C)$ is indeed a classical tautology:

A	B	C	$A \vee B \wedge C$	$A \wedge C \vee B \wedge C$	$A \vee B \wedge C \iff A \wedge C \vee B \wedge C$
False	**False**	**False**	**False**	**False**	**True**
False	**False**	**True**	**False**	**False**	**True**
False	**True**	**False**	**False**	**False**	**True**
False	**True**	**True**	**True**	**True**	**True**
True	**False**	**False**	**False**	**False**	**True**
True	**False**	**True**	**True**	**True**	**True**
True	**True**	**False**	**False**	**False**	**True**
True	**True**	**True**	**True**	**True**	**True**

2.4 Predicate logic

So far we cannot say express any interesting propositions, because we cannot make use of variables. To change this we will use predicate logic. A *predicate* over a type expresses property of elements of this type. An example is the predicate **Even** over the natural numbers, which expresses the property of a number being even. If $n \in \mathbf{Nat}$ then **Even** n is a proposition. We write $\mathbf{Even} \in \mathbf{Nat} \to \mathbf{Prop}$ to say that **Even** is a predicate over the natural numbers. Predicates can have more than one argument, then they are also **relations**. An example of a relation is $\leqslant$ which is a relation over the natural numbers (actually also for other number types): that is idf $m, n \in \mathbf{Nat}$ then $m \leqslant n$ is a proposition. We write $(\leqslant) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$. An important relation which exists for all types is equality: if $(=) \in a \to a \to \mathbf{Prop}$ where a is any type.

Variables are introduced by *quantifiers*. Let P be a proposition which contains a variable, lets say x and assume as given a set A. Then we can construct:

- $\forall x \in A.P$, we say **for all** x **in** A, P **holds**. We accept $\forall x \in A.P$ if we can show P where x is replaced by any given element of $a \in A$ (we write this as $P[x = a]$), e.g. we accept $\forall x \in \mathbf{Bool}.x = \text{True} \vee x = \text{False}$.
- $\exists x \in A.P$, we say **there exists** x **in** A, **such that** P. We accept $\exists x \in A.P$ if we can show P where x is replaced by a specific element of A, e.g. we accept $\exists x \in \mathbf{Bool}.x = \text{True}$.

The reading conventions for quantifiers are that they bind weaker than any of the propositional connectives, e.g. we read

$$\forall x \in A.P \iff Q$$

as

$$\forall x \in A.(P \iff Q)$$

It is instructive to illustrate the use of quantifiers by showing how English sentences can be translated into predicate logic. Let's assume that we have a type **Party** of names of people present at a party and a relation $\mathit{knows} \in \mathbf{Party} \to \mathbf{Party} \to \mathbf{Prop}$ where $\mathit{knows}\ a\ b$ expresses that a knows b. Here are some examples:

- *John knows Mary.*

$$\textit{knows}\ \text{John Mary}$$

- *Everybody knows somebody.*

$$\forall x \in \textbf{Party}.\exists y \in \textbf{Party}.\textit{knows}\ x\ y$$

- *Somebody is known by everybody.*

$$\exists y \in \textbf{Party}.\forall x \in \textbf{Party}.\textit{knows}\ x\ y$$

  Note that exchanging different quatifiers changes the meaning.

- *Somebody knows everybody.*

$$\exists x \in \textbf{Party}.\forall y \in \textbf{Party}.\textit{knows}\ x\ y$$

- *Everybody who Mary knows, knows her.*

$$\forall x \in \textbf{Party}.\textit{knows}\ \text{Mary}\ x \implies \textit{knows}\ x\ \text{Mary}$$

- *There are at least two different people who know John.*

$$\exists x \in \textbf{Party}.\exists y \in \textbf{Party}.\neg(x = y) \land \textit{knows}\ x\ \text{John} \land \textit{knows}\ y\ \text{John}$$

  It is customary to combine quantifiers of the same sort and write $x \neq y$ for $\neg(x = y)$, i.e. we may write:

$$\exists x, y \in \textbf{Party}.x \neq y \land \textit{knows}\ x\ \text{John} \land \textit{knows}\ y\ \text{John}$$

- *There is exactly one person who George knows.*

$$\exists x \in \textbf{Party}.\textit{knows George}\ x \land \forall y \in \textbf{Party}.\textit{knows George}\ y \implies x = y$$

  We express *exactly one* by saying that everybody who George knows is the same person.

As for propositional logic we have tautologies, which are propositions which are provable for any predicate. Given a type a and predicates $P, Q \in a \to \textbf{Prop}$ and a proposition P here are examples for tautologies:

- $(\forall x \in a.P\ x \land Q\ x) \iff (\forall y \in a.P\ y) \land (\forall z \in a.Q\ z)$ **Proof:**

  We show both direction seperately:

  $(\forall x \in a.P\ x \land Q\ x) \implies (\forall y \in a.P\ y) \land (\forall z \in a.Q\ z)$ We assume $\forall x \in A.P\ x \land Q\ x \quad (1)$. To show $(\forall y \in a.P\ y) \land (\forall z \in a.Q\ z)$ we have to show both $\forall y \in a.P\ y \quad (2)$ and $\forall z \in a.Q\ y \quad (3)$. To show (2) we assume $y \in a \quad (4)$, we have to show $P\ y$. Using (1) and (4) we know $P\ y \land Q\ y$ and hence $Q\ y$. The proof of (3) follows the same idea.

  $(\forall y \in a.P\ y) \land (\forall z \in a.Q\ z) \implies \forall x \in a.P\ x \land Q\ x$ We assume $(\forall y \in a.P\ y) \land (\forall z \in a.Q\ z) \quad (1)$ to prove $\forall x \in a.P\ x \land Q\ x$. Having assumed (1) we know $\forall y \in a.P\ y \quad (2)$ and $\forall z \in a.P\ z \quad (3)$ To show this we assume $x \in a \quad (4)$, to show $P\ x$ we use (2) and (4) and to derie $Q\ x$ we use (3) and (4), hence we have shown $P\ x \land Q\ x$.

$\square$

- $(\exists x \in a.P\ x \vee Q\ x) \iff (\exists y \in a.P\ y) \vee (\exists z \in a.Q\ z)$
- $(\exists x \in A.P \implies R) \iff \forall x \in A.P \implies R$

Instead only proving generic tautologies we can now prove specific statements, e.g. that *not* hasn't got a fixpoint:

$$\forall x \in \mathbf{Bool}.\mathit{not}\ x \neq x$$

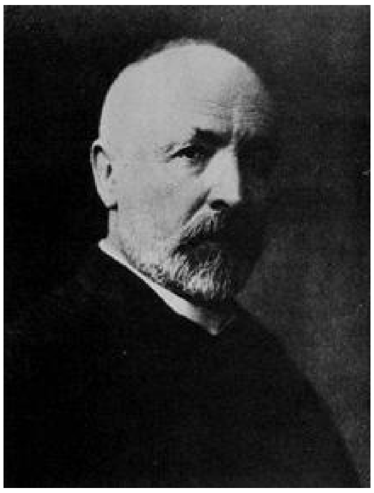
Proof: We have to show $\mathit{not}\ x \neq x$ for any $x \in \mathbf{Bool}$. There are only two possibilities:

$x =$ True We have that *not* True $=$ False by definition of *not* and it is obvious that True $\neq$ False.

$x =$ False We have that *not* False $=$ True by definition of *not* and it is obvious that False $\neq$ True.

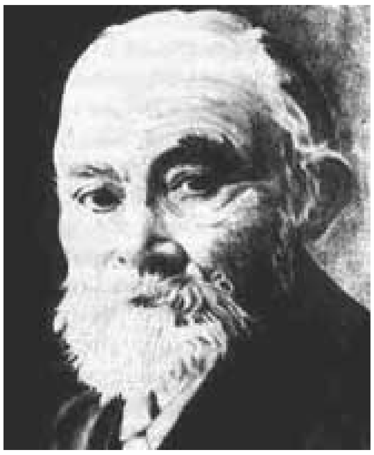
$\square$

3 Bits of history

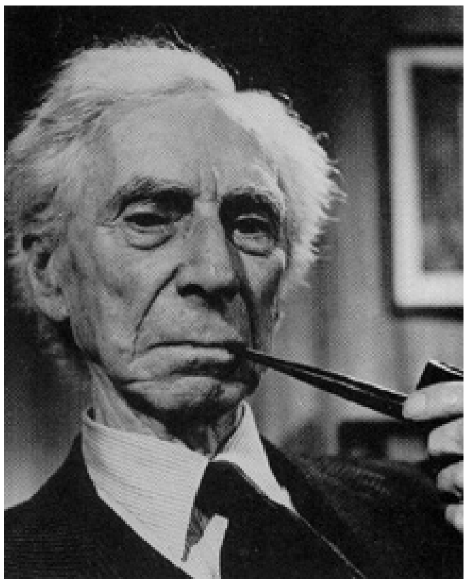


Georg Cantor, a German mathematician, was the first one to define the notion of a set (german *Menge*) in 1895:

> By a set we are to understand any collection into a whole of definite and separate objects of our intuition or our thought.

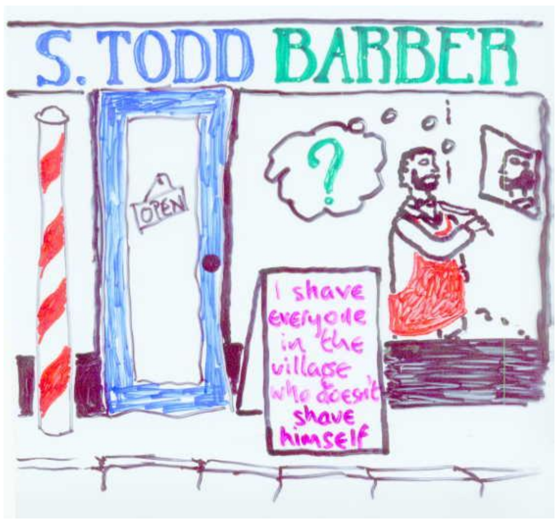


Another German, Friedrich Ludwig Gottlob Frege (1848-1925) wanted to put mathematics on a firm foundation, and for this purpose he developed *predicate logic*. He published a book. called *Begriffsschrift*, which also used Cantor's naive perception of a set, in particular the principle of comprehension, i.e. every property defines a set. If P is a property (or predicate) using a variable x, we write $\{x \mid P\}$ for the set of all objects with property P. The Begriffsschrift was also read by an English logician, Bertrand Russell (1872-1970):



Russell found a fundamental flaw in Frege's system which made it inconsistent. This means you could prove anything in it no matter whether it is true or false. Naturally, this renders it useless as a foundation of mathematics. Russell wrote a letter to Frege explaining the problem — it seems that Frege never recovered from the blow. He certainly didn't know a good way to fix it.

To explain Russell's paradox, let's have a look at a little story which involves a barber (let's call him Sweeny Todd) who lives in a little village. Since he is the only barber in the village he has put a note in his window saying: *I shave everybody in the village who doesn't shave himself*



by Conor McBride

Now, the barber has got a dilemma: If he shaves himself then he shouldn't shave himself because he is shaving precisely those people who do not shave himself. On the other hand if he does not shave himself then he should shave himself according to the sign.

The story with the barber may not seem to be so dramatic because we are quite used to false advertising. However, if we translate the same idea into set theory it becomes much more serious: Let us consider the **set R of all sets which do not contain themselves**, we write

$$R = \{X \mid X \notin X)\}$$

where $X \notin X$ is short for $\neg(X \in X)$. E.g. **Nat** $\notin R$ because the set of natural numbers does not contain itself.

However, what about R itself? If R does contain itself then it should not. If on the other hand R does not contain itself then by definition it should contain itself. Hence Frege's system was inconsistent!

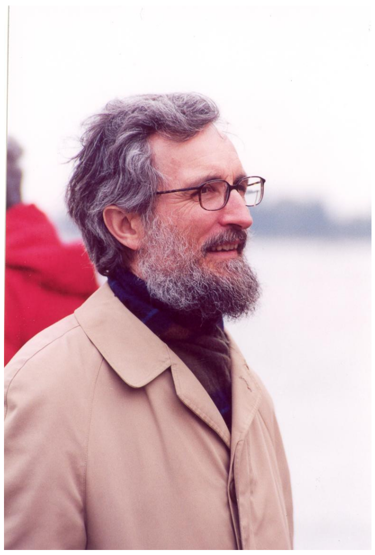
However, set theory was saved by two Germans:

Ernst Friedrich Ferdinand Zermelo (1871-1853)

Adolf Abraham Halevi Fraenkel (1891-1965)

Zermelo identified the unlimited comprehension principle, i.e. that every property gives rise to a set, as the source of the problems and introduced *the restricted comprehension principle* instead. I.e. we are only allowed to construct subsets of already existing sets, i.e. $\{x \in A \mid P\}$, where A already exists. The set R clearly does not fall in this category [10] and hence is outlawed. However, now we have to provide some ways to construct basic sets, i.e. the empty set, finite sets, the set of natural numbers and the power set (the set of all subsets of a set). Zermelo founded axiomatic set theory, in which these principles are incorporated in some basic propositions about $\in$ and $=$ called axioms. Later Fraenkel extended Zermelo's system by adding an important axiom, the axiom of replacement. The resulting system is usually called *Zermelo Fraenkel set theory* or ZF set theory for short and is widely accepted as a foundation of modern mathematics.

For many people (certainly for most mathematicans) the story ends here. However, ZF set theory is not very useful as a programming language (for example it allows to define functions which cannot be run on a computer).



Per Martin-Löf, a Swedish Mathematician and Philosopher started to develop *Type Theory* as a constructive foundation of Mathematics in the early 70's. This language is interesting for Computer Science because it is at the same time a language to reason about mathematical objects and a programming language. There are a number of computer programs which implement different variations of Martin-Löf's Type Theory: NuPRL (for *Nearly Ultimate Proof Representation Language*) developed by Robert Constable and his team in the USA; LEGO developed by Randy Pollack in Edinburgh; the Swedish systems ALF and AGDA; the French system COQ [11] , which is now one of the most successful computer proof systems, and very recently Conor McBride's Epigram system [12] on which my 2nd/3rd year module *Computer Aided Formal Reasoning* (G5BCFR) is based.

[10]Unless we introduce a set of all sets, which would make the theory inconsistent, due to Russell's paradox.

[11]`http://coq.inria.fr/`

[12]`http://www.dur.ac.uk/CARG/epigram/`

4 Natural numbers

How can we understand an infinite type like **Nat**?

$$\mathbf{Nat} = \{0, 1, 2, 3, \ldots\}$$

And how can we define operations like *isEven* $\in$ **Nat** $\to$ **Bool** which should return True if its input is an even number and False otherwise? We can hardly define *isEven* by listing all the cases:

$$\begin{aligned}&\mathit{isEven}\ 0 = \mathrm{True}\\ &\mathit{isEven}\ 1 = \mathrm{False}\\ &\mathit{isEven}\ 2 = \mathrm{True}\\ &\mathit{isEven}\ 3 = \mathrm{False}\\ &\mathit{isEven}\ 4 = \mathrm{True}\\ &\mathit{isEven}\ 5 = \mathrm{False}\\ &\ldots\end{aligned}$$

And how can we reason about *isEven* and other functions on **Nat** or other infinite types?

Our understanding of the natural numbers is that they are related to counting: 0 is a natural number and if $n \in$ **Nat** then its successor, i.e. the number $n+1$ is a natuiral number, we write Succ $n \in$ **Nat**. Now 0 and Succ are the constructors of **Nat**, i.e. we define [13]

$$\boxed{\textbf{data Nat} = 0 \mid \mathrm{Succ}\ \mathbf{Nat}}$$

now **Nat** looks a bit different than what we are used to

$$\mathbf{Nat} = \{0, \mathrm{Succ}\ 0, \mathrm{Succ}\ (\mathrm{Succ}\ 0), \mathrm{Succ}\ (\mathrm{Succ}\ (\mathrm{Succ}\ 0)), \ldots\}$$

and conveniently we define

$$\begin{aligned}1 &= \mathrm{Succ}\ 0\\ 2 &= \mathrm{Succ}\ 1\\ 3 &= \mathrm{Succ}\ 2\\ 4 &= \mathrm{Succ}\ 3\\ 5 &= \mathrm{Succ}\ 4\end{aligned}$$

However, we should be able to look beyond the superficial notation using the decimal system. The above definition of the natural number is in some way the simplest and mathematically most elegant. It was suggested by Italian logician Guiseppe Peano at the beginning of the 20th century who codified the basic rules of arithmetic by his famous axioms (called Peano's axioms). We shall not study his formulation of the axioms here in detail, but our presentation is certainly inspired by them.

[13] This is a beautified version of the real Haskell code. Haskell wouldn't accept using the symbol 0 as a constructor, hence we are actually writing Zero.

Guiseppe Peano (1858 - 1932)

We shall later say how we can express other notational systems for natural numbers by similar type definitions, in particular we shall have a look at the binary representation. Peano's representation, which we could call *unary* is practically useless, because the numbers grow far to long quickly. However, it is the easiest way to understand the natural numbers and to reason about them. Indeed, this is precisely what Peano intended. We will later introduce binary numbers and define translations between binary numbers adn Peano's numbers.

4.1 Primitive recursion

How can *isEven* be defined using Peano's definitions of the natural numbers? We observe that $\mathit{isEven}\ 0 = \text{True}$ and $\mathit{isEven}\ (n+1)$ is *the opposite* of $\mathit{isEven}\ n$, that is $\mathit{isEven}\ (n+1) = \mathit{not}\ (\mathit{isEven}\ n)$. Hence we write:

$$\boxed{\begin{aligned}&\mathit{isEven} \in \mathbf{Nat} \to \mathbf{Bool}\\ &\mathit{isEven}\ 0 = \text{True}\\ &\mathit{isEven}\ (\text{Succ}\ n) = \mathit{not}\ (\mathit{isEven}\ n)\end{aligned}}$$

We say that *isEven* is defined by *primitive recursion*, that is $\mathit{isEven}\ (\text{Succ}\ n)$ is defined using $\mathit{isEven}\ n$. This is reasonable, because if we construct the natural number Succ n we must have constructed the natural number n before. In the same way we can calculate the answer $\mathit{isEven}\ n$ *before* we calculate $\mathit{isEven}\ (\text{Succ}\ n)$ using our previously calculated answer.

Actually, this is not an accurate description of how is recursion is executed on a computer, the computer will only calculate $\mathit{isEven}\ n$ when it has to to calculate $\mathit{isEven}\ (\text{Succ}\ n)$. However, the explanation above justifies why this process always terminates and why a function constructed by primitive recursion will always produce an answer.

Hence we can calculate $\mathit{isEven}\ 3$ as follows:

$$\begin{aligned}&\mathit{isEven}\ 3\\ &=\quad \{\ \text{definition of } 3\ \}\\ &\mathit{isEven}\ (\text{Succ}\ 2)\\ &=\quad \{\ \text{definition of } \mathit{isEven}\ \}\end{aligned}$$

$$\begin{array}{l}
\mathit{not}\ (\mathit{isEven}\ 2) \\
\ =\quad \{\ \text{definition of } 2\ \} \\
\mathit{not}\ (\mathit{isEven}\ (\text{Succ}\ 1)) \\
\ =\quad \{\ \text{definition of } \mathit{isEven}\ \} \\
\mathit{not}\ (\mathit{not}\ (\mathit{isEven}\ 1)) \\
\ =\quad \{\ \text{definition of } 1\ \} \\
\mathit{not}\ (\mathit{not}\ (\mathit{isEven}\ (\text{Succ}\ 0))) \\
\ =\quad \{\ \text{definition of } \mathit{isEven}\ \} \\
\mathit{not}\ (\mathit{not}\ (\mathit{not}\ (\mathit{isEven}\ 0))) \\
\ =\quad \{\ \text{definition of } \mathit{isEven}\ \} \\
\mathit{not}\ (\mathit{not}\ (\mathit{not}\ \text{True})) \\
\ =\quad \{\ \text{definition of } \mathit{not}\ \} \\
\mathit{not}\ (\mathit{not}\ \text{False}) \\
\ =\quad \{\ \text{definition of } \mathit{not}\ \} \\
\mathit{not}\ \text{True} \\
\ =\quad \{\ \text{definition of } \mathit{not}\ \} \\
\text{False}
\end{array}$$

When displaying an equational derivation, we always say what principles we have applied to get from one line to the next. The derivation above is a simple calculation, all we are doing is expanding definitions.

Primitive recursion can be contrasted with *general recursion*, which is permitted in Haskell. A general recursive function is defined recursively without insisting that the recursive call is on a *previous* value. General recursion may not terminate and while it is easy to use for programming, the mathematical theory is more involved. Hence we shall not consider it here, but only means of recursion which can be justified like primitive recursion.

4.1.1 Addition

Using primitive recursion, we can now define addition:

$$\boxed{\begin{array}{l}
(+) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat} \\
0 + n = n \\
(\text{Succ}\ m) + n = \text{Succ}\ (m + n)
\end{array}}$$

Let us spell out what we are doing: $(+)$ is defined by recursion over the first argument. $(+)\ 0$ gives rise to the function which just returns its argument (the identity function). $(+)\ (\text{Succ}\ m)$ uses $(+)\ m$ to calculate $(+)\ m\ n$ and then takes Succ of that calculation.

We can calculate $2 + 3$:

$$\begin{array}{l}
2 + 3 \\
\ =\quad \{\ \text{definition of } 2\ \} \\
(\text{Succ}\ 1) + 3 \\
\ =\quad \{\ \text{definition of } +\ \} \\
\text{Succ}\ (1 + 3) \\
\ =\quad \{\ \text{definition of } 1\ \} \\
\text{Succ}\ ((\text{Succ}\ 0) + 3) \\
\ =\quad \{\ \text{definition of } +\ \}
\end{array}$$

$$
\begin{aligned}
&\text{Succ}\ (\text{Succ}\ (0+3)) \\
&= \quad \{ \text{ definition of } + \} \\
&\text{Succ}\ (\text{Succ}\ 3) \\
&= \quad \{ \text{ definition of } 5 \} \\
&5
\end{aligned}
$$

4.1.2 Multiplication

Let's go on and define some more functions, e.g. multiplication:

$$
\boxed{
\begin{aligned}
&(*) \in \textbf{Nat} \to \textbf{Nat} \to \textbf{Nat} \\
&0 * n = 0 \\
&(\text{Succ}\ m) * n = m + (m * n)
\end{aligned}
}
$$

Multiplication is defined by repeated addition in the same way as addition is defined by repeated Succ. As an example we calculate $2 * 3$

$$
\begin{aligned}
&2 * 3 \\
&= \quad \{ \text{ definition of } 2 \} \\
&(\text{Succ}\ 1) * 3 \\
&= \quad \{ \text{ definition of } * \} \\
&3 + 1 * 3 \\
&= \quad \{ \text{ definition of } 1 \} \\
&3 + (\text{Succ}\ 0) * 3 \\
&= \quad \{ \text{ definition of } * \} \\
&3 + 3 + 0 * 3 \\
&= \quad \{ \text{ definition of } * \} \\
&3 + 3 \\
&= \quad \{ \text{ calculate } + \} \\
&6
\end{aligned}
$$

4.1.3 Equality and less-or-equal

We define the function $(\equiv) \in \textbf{Nat} \to \textbf{Nat} \to \textbf{Bool}$ which decides whether two natural numbers are equal. As in the case of $(+)$ we exploit the curried view of a function in the definition. Let's analyze the first argument: $(\equiv)\ 0 \in \textbf{Nat} \to \textbf{Bool}$ should return the function which recognizes 0, i.e. which returns True precisely if its argument is 0, hence we write

$$
\begin{aligned}
&0 \equiv 0 &&= \text{True} \\
&0 \equiv (\text{Succ}\ n) &&= \text{False}
\end{aligned}
$$

The other case is $(\equiv)\ (\text{Succ}\ m) \in \textbf{Nat} \to \textbf{Bool}$, this function should return False if presented with 0, but what to do, if it is applied to Succ m? The answer is that we recursively call $(\equiv)\ m$ and apply it to n, because Succ $m \equiv$ Succ n has the same value as $m \equiv n$. Hence we define

$$
\begin{aligned}
&\text{Succ}\ m \equiv 0 &&= \text{False} \\
&\text{Succ}\ m \equiv \text{Succ}\ n &&= m \equiv n
\end{aligned}
$$

Putting everything together we arrive at the following definition:

$$\begin{array}{llll}
(\equiv) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Bool} \\
0 & \equiv 0 & = \text{True} \\
0 & \equiv (\text{Succ } n) & = \text{False} \\
(\text{Succ } m) & \equiv 0 & = \text{False} \\
(\text{Succ } m) & \equiv (\text{Succ } n) & = m \equiv n
\end{array}$$

By a slight modification, i.e. by returning True instead of False in the second line we obtain $\leqslant$ instead of $\equiv$.

$$\begin{array}{llll}
(\leqslant) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Bool} \\
0 & \leqslant 0 & = \text{True} \\
0 & \leqslant (\text{Succ } n) & = \text{True} \\
(\text{Succ } m) & \leqslant 0 & = \text{False} \\
(\text{Succ } m) & \leqslant (\text{Succ } n) & = m \leqslant n
\end{array}$$

4.1.4 Factorial

When we set up musical chairs, we may wonder: *How many ways are there to place 6 children on 6 chairs?* The answer is: there 6 possibilities for the first child, 5 for the second and so on, and there will be only one place left for the last one. Hence there are $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$. We write *fac* n (mathematical notation $n!$) for the number of permutations of n elements (e.g. the number of ways to arrange n children on n chairs).

We observe that there is one way to arrange no children on no chairs (hence *fac* $0 = 1$), there are $n + 1$ ways to place the first child and *fac* n ways to distribute the remaining n children, hence $\mathit{fac}\ (n + 1) = (n + 1) * (\mathit{fac}\ n)$. This leads to the following definition of *fac* by primitive recursion:

$$\begin{array}{l}
\mathit{fac} \in \mathbf{Nat} \to \mathbf{Nat} \\
\mathit{fac}\ 0 = 1 \\
\mathit{fac}\ (\text{Succ } n) = (\text{Succ } n) * (\mathit{fac}\ n)
\end{array}$$

4.1.5 Fibonacci

Another interesting function is the famous Fibonacci function $\mathit{fib} : \mathbf{Nat} \to \mathbf{Nat}$, which is usually motivated by the problem to calculate the number of rabbits after n months, if we assume that the rabbits have a child two months after they are born (ignoring among others the fact that we need actually two rabbits). We start with no rabbits in month 0 and one rabbit in month 1, consequently we have still one rabbit in month 2 and 2 rabbits in month 3, because our first rabbit got its first child. In general we have that the rabbits in month $n+2$ are the sum of the number of the rabbits in the previous months and the rabbits from the month before, who will have children now: $\mathit{fib}\ (n + 2) = (\mathit{fib}\ (n + 1)) + (\mathit{fib}\ n)$. I.e. we get the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765 \ldots$$

We define *fib*:

$$\begin{aligned}
&\mathit{fib} \in \mathbf{Nat} \to \mathbf{Nat}\\
&\mathit{fib}\ 0 = 0\\
&\mathit{fib}\ (\mathrm{Succ}\ 0) = 1\\
&\mathit{fib}\ (\mathrm{Succ}\ (\mathrm{Succ}\ m)) = (\mathit{fib}\ m) + (\mathit{fib}\ (\mathrm{Succ}\ m))
\end{aligned}$$

Different from the functions we have seen so far, we are not just using the value of a function at n to calculate the value at Succ n, but here we actually use two previous values. Using smaller values in general is called *course of value recursion* and is a modest generalisation of primitive recursion. Indeed, it has got the same justification as primitive recursion, e.g. we adopt the view that the function values for smaller values have been calculated earlier.

4.2 Induction

How can we prove something for *all* natural numbers? This is a problem very similar to the one we have discussed in the previous section and indeed the solution is very closely related to primitive recursion. Let's consider a very simple example, we want to prove $\forall n \in \mathbf{Nat}.n + \mathrm{Zero} = n$. Recall the definition of $+$

$$\begin{aligned}
&(+) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat}\\
&0 + n = n\\
&(\mathrm{Succ}\ m) + n = \mathrm{Succ}\ (m + n)
\end{aligned}$$

We can *see* by looking at the definition that $\forall\ n \in \mathbf{Nat}.0 + n = n$, but what about the equally true $\forall\ n \in \mathbf{Nat}.n + 0 = n$. Sure, we will remember the equality $\forall\ m, n \in \mathbf{Nat}.m + n = n + m$ (called *commutativity of addition*) but here we shall actually establish this law from first principles and we will see that actually $\forall\ n \in \mathbf{Nat}.n + 0 = n$ is a useful auxilliary property when showing commutativity.

We can start showing this property for the first few natural numbers and observe that a certain pattern emerges:

0 $0 + 0 = 0$ by definition of $+$.

1 To show $(\mathrm{Succ}\ 0) + 0 = \mathrm{Succ}\ 0$ we first use the definition of $+$ to see $(\mathrm{Succ}\ 0) + 0 = \mathrm{Succ}\ (0 + 0)$ and then we can exploit the previous line to see that $\mathrm{Succ}\ (0 + 0) = \mathrm{Succ}\ 0$

2 To show $(\mathrm{Succ}\ (\mathrm{Succ}\ 0)) + 0 = \mathrm{Succ}\ (\mathrm{Succ}\ 0)$ we first use the definition of $+$ to see $(\mathrm{Succ}\ (\mathrm{Succ}\ 0)) + 0 = \mathrm{Succ}\ ((\mathrm{Succ}\ 0) + 0)$ and then we can exploit the previous line to see that $\mathrm{Succ}\ ((\mathrm{Succ}\ 0) + 0) = \mathrm{Succ}\ (\mathrm{Succ}\ 0)$

...

Each of the subsequent line is simply a consequence of the previous one. E.g. to show $(\mathrm{Succ}\ n){+}0 = \mathrm{Succ}\ n$ we first apply the definition of $+$ to see $(\mathrm{Succ}\ n){+}0 = \mathrm{Succ}\ (n + 0)$ and then we apply the *previous line*, i.e. the proof for n to see that $\mathrm{Succ}\ (n + 0) = \mathrm{Succ}\ n$.

We can see that this shows that the property holds for all natural numbers, the idea is the same as for primitive recursion: because the number n is constructed before Succ n, we can establish the property for n before we show it for Succ n.

The **principle of induction** can be summarized as follows: to prove a property P for all natural numbers, i.e. to show $\forall\, n \in \mathbf{Nat}.P\ n$, we prove it for 0, i.e. we show $P\ 0$ and assuming that it holds for $n \in \mathbf{Nat}$, i.e. $P\ n$ we can show that it holds for P (Succ n). The assumption $P\ n$ which can be used when showing P (Succ n) we call the induction hypothesis (IH).

4.2.1 Commutativity of $(+)$

Let's now prove $\forall\, m, n \in \mathbf{Nat}.m+n = n+m$ by induction over m. If $m = 0$ this reduces to showing $0+n = n+0$, we know that $0+n = n$ by the definition of $+$ and we have just shown $n = n+0$ above. Let's assume that $\forall\, n \in \mathbf{Nat}.m+n = n+m$ to prove $\forall\, n \in \mathbf{Nat}\,.\,(\text{Succ}\ m) + n = n + (\text{Succ}\ m)$. Now given $n \in \mathbf{Nat}$ we know from the definition of $+$ that $(\text{Succ}\ m) + n = \text{Succ}\ (m + n)$, we can apply the induction hypthesis to deduce that $\text{Succ}\ (m+n) = \text{Succ}\ (n+m)$. All that is left to show is that $\text{Succ}\ (n + m) = n + (\text{Succ}\ m)$, showing this requires a separate proof by induction.

It is time to tidy up our argument and also follow standard mathematical practice and present proofs forward i.e. starting from the assumptions and basic lemmas (auxilliary theorems) to the goal, the final theorem. No human being constructs a proof like this, everybody does it the other way around! Some people believe that this is part of a conspiracy by Mathematicians to make their knowledge inaccessible to the rest of the world...

Lemma 4.1 $\forall\, n \in \mathbf{Nat}.n + 0 = n$

Proof: By induction over $n \in \mathbf{Nat}$:

Case $n = 0$ We show $0 + 0 = 0$:

$$\begin{aligned}&0 + 0\\ &\ = \quad \{\text{ definition of } + \}\\ &0\end{aligned}$$

Case $n = \text{Succ}\ n'$ We assume $n' + 0 = n'$ (IH) to show $(\text{Succ}\ n') + 0 = \text{Succ}\ n'$:

$$\begin{aligned}&(\text{Succ}\ n') + 0\\ &\ = \quad \{\text{ definition of } + \}\\ &\text{Succ}\ (n' + 0)\\ &\ = \quad \{\text{ (IH) }\}\\ &\text{Succ}\ n'\end{aligned}$$

□

Lemma 4.2 $\forall\, m, n \in \mathbf{Nat}.m + (\text{Succ}\ n) = \text{Succ}\ (m + n)$

Proof: By induction over $m \in \mathbf{Nat}$:

Case $m = 0$ We show $\forall\ n \in \mathbf{Nat}. 0 + (\text{Succ}\ n) = \text{Succ}\ (0 + n)$:

$$\begin{array}{l} 0 + (\text{Succ}\ n) \\ \ = \quad \{ \text{ definition of } + \} \\ \text{Succ}\ n \\ \ = \quad \{ \text{ definition of } + \} \\ \text{Succ}\ (0 + n) \end{array}$$

Case $m = \text{Succ}\ m'$ We assume $\forall\ n \in \mathbf{Nat}. m + (\text{Succ}\ n) = \text{Succ}\ (m + n) \quad (IH)$ to show $(\text{Succ}\ m) + (\text{Succ}\ n) = \text{Succ}\ (m + n)$

$$\begin{array}{l} (\text{Succ}\ m) + (\text{Succ}\ n) \\ \ = \quad \{ \text{ definition of } + \} \\ \text{Succ}\ (m + (\text{Succ}\ n)) \\ \ = \quad \{ \text{ IH } \} \\ \text{Succ}\ (\text{Succ}\ (m + n)) \\ \ = \quad \{ \text{ definition of } + \} \\ \text{Succ}\ ((\text{Succ}\ m) + n) \end{array}$$

□

Theorem 4.3 $\forall\ m, n \in \mathbf{Nat}. m + n = n + m$

Proof: By induction over $m \in \mathbf{Nat}$.

Case $m = 0$ We show $\forall\ n \in \mathbf{Nat}. 0 + n = n + 0$:

$$\begin{array}{l} 0 + n \\ \ = \quad \{ \text{ definition of } + \} \\ n \\ \ = \quad \{ \text{ lemma } \mathbf{??} \} \\ n + 0 \end{array}$$

Case $m = \text{Succ}\ m'$ We assume $\forall\ n \in \mathbf{Nat}. m' + n = n + m' \quad (IH)$ to show $(\text{Succ}\ m') + n = n + (\text{Succ}\ m')$:

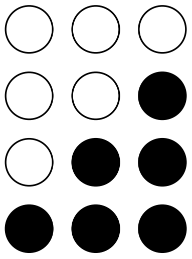
$$\begin{array}{l} (\text{Succ}\ m') + n \\ \ = \quad \{ \text{ definition of } + \} \\ \text{Succ}\ (m' + n) \\ \ = \quad \{ \text{ IH } \} \\ \text{Succ}\ (n + m') \\ \ = \quad \{ \text{ lemma } \mathbf{??} \} \\ n + (\text{Succ}\ m') \end{array}$$

□

Isn't this a bit much effort for a theorem, whose truth can be easily seen intuitively? Every child knows that if you put two heaps of sweeties together that the order in which you do this does not affect the number of sweeties you have in the end. However, apart from the point that we used a simple example to illustrate a method, it is sometimes worthwhile to prove the obvious. Maybe you were only thinking that the function you defined was addition but as soon as you are going to prove something about it you discover that there is a bug. This is different from Mathematics: Computer Scientists are interested in proving obvious things. It is also a good idea to enlist the help of a computer to make sure that we don't accidently cheat. We will get back to this later.

4.2.2 Gauss' theorem

Let's consider a less obvious proposition we can prove by induction. Carl Friederich Gauss was a famous German Mathematician in the 19th century; one of the theorems he proved is the fundamental theorem of algebra, which says that every polynom over the complex numbers has a solution. However, here we are just concerned with a little theorem he allegedly discovered when he was still a school boy. Apprently he was naughty and as a punishement teacher set him the task to add the numbers until 100. But the teacher hadn't accounted for young Gauss' genius, who very quickly came up with the right answer: 5050. Instead of doing all the stupid calculations, Gauss had realized that there is a simple formula, namely that the sum of the numbers upto n is given by $(n * (n+1)) \,/\, 2$. The following picture illustrates the basic idea in the case $n = 3$:



I.e. the sum of the $1 + 2 + 3$ depicted by the area covered by the black balls is precisely half the area of the $3 * 4$ rectangle.

First, for reference lets define by primitive recursion the function which calculates the numbers upto n, which we are going to call *sum* (Mathematicans would write $\Sigma_{i=0}^{n} i$):

$$\boxed{\begin{array}{l} \mathit{sum} \in \mathbf{Nat} \to \mathbf{Nat} \\ \mathit{sum}\ 0 = 0 \\ \mathit{sum}\ (\mathrm{Succ}\ n) = (\mathrm{Succ}\ n) * (\mathit{sum}\ n) \end{array}}$$

We want to show $\mathit{sum}\ n = (n * (n+1)) \,/\, 2$ to avoid having to define division and to justify that the division in this case stays always withing the natural numbers, we show instead $2 * (\mathit{sum}\ n) = n * (\mathrm{Succ}\ n)$ by induction:

Theorem 4.4 $\forall\, n \in \mathbf{Nat} \,.\, 2 * (\mathit{sum}\ n) = n * (\mathrm{Succ}\ n)$

Proof: By induction over $n \in \mathbf{Nat}$.

Case $n = 0$ We show $2 * (\mathit{sum}\ 0) = 0 * (\mathrm{Succ}\ 0)$

$$\begin{array}{l} 2 * (\mathit{sum}\ 0) \\ \quad = \quad \{ \text{ definition of } *,\ \mathit{sum} \ \} \\ 0 \\ \quad = \quad \{ \text{ definition of } * \ \} \\ 0 * (\mathrm{Succ}\ 0) \end{array}$$

Case $n = \mathrm{Succ}\ n'$ We assume

$2 * (\mathit{sum}\ n') = n' * (\text{Succ}\ n') \quad (\mathit{IH})$

to show $2 * (\mathit{sum}\ (\text{Succ}\ n')) = (\text{Succ}\ n') * (\text{Succ}\ (\text{Succ}\ n'))$

$$\begin{array}{l}
2 * (\mathit{sum}\ (\text{Succ}\ n')) \\
\ =\quad \{ \text{ definition of sum } \} \\
2 * ((\text{Succ}\ n') + (\mathit{sum}\ n')) \\
\ =\quad \{ \text{ distributivity } \} \\
2 * (\text{Succ}\ n') + 2 * (\mathit{sum}\ n') \\
\ =\quad \{ \text{ IH } \} \\
2 * (\text{Succ}\ n') + n * (\text{Succ}\ n') \\
\ =\quad \{ \text{ distributivity } \} \\
(2 + n) * (\text{Succ}\ n') \\
\ =\quad \{ \text{ commutativity, definition of } + \} \\
(\text{Succ}\ n') * (\text{Succ}\ (\text{Succ}\ n'))
\end{array}$$

$\square$

In the proof above we have used distributivity, i.e.

$\forall\ i, j, k \in \mathbf{Nat}\,.\ i * (j + k) = (i * j) + (i * k).$

Sure, we could have proven this here (by induction) but I will leave this as an exercise.

5 Inductive types

The idea behind the definition of the type of natural numbers, can be applied to many other types using the **data** keyword in Haskell. We are calling those types *inductive types* and indeed they come with their own primitive recursion and induction principles.

5.1 Lists

One of the most versatile types, especially but not only, in functional programming is the type of lists. Indeed, the name of the first functional language, LISP, introduced in 1958 as the 2nd high level programming language, stands for LIst Processing.

Given a type a we write $[a]$ for the type of lists or sequences of elements of a. E.g. $[23, -1, 1, 1, 0] \in [\mathbf{Int}]$, similar as for (a, b) we use the same symbols $[\ldots]$ for the operator on types and on elements. Actually the $[a_1, a_2, ..., a_n]$ notation is just a shorthand for $a_1 : (a_2 : \ldots (a_n : [])$. The basic constructors for lists are $[] \in [a]$ for the empty list, and given $x \in a$ and a list $xs \in [a]$ we can construct the new list $x : xs \in [a]$ (for historic reasons : is called *cons*). I.e. $[23, -1, 1, 1, 0] = 23 : (-1 : (1 : (1 : (0 : []))))$.

In Haskell lists are predefined because they use some special syntactic sugar, but apart from this we could have just defined:

$$\mathbf{data}\,[a] = [] \mid a : [a]$$

Strings are a special case of lists, we use the type of characters **Char** which is a finite type whose elements are the characters which can be used in computing (e.g. one of the 2^{16} unicode characters). We write elements of **Char** using single quotes, e.g. `'a'` $\in$ **Char**. The type of strings is defined as lists of characters:

$$\mathbf{type\ String} = [\mathbf{Char}]$$

The common syntax for strings using double quotes, i.e. `"Hello"` is just syntactiv sugar for [`'H'`, `'e'`, `'l'`, `'l'`, `'o'`] which in turn is just short for `'H'` : (`'e'` : (`'l'` : (`'l'` : (`'o'` : []))))).

One of the basic functions on lists is append, $+\!\!+$, which combines two lists, i.e. $[1, 2] +\!\!+ [3, 4] = [1, 2, 3, 4]$. To be able to define this function we have to extend the principle of primitive recursion to lists. We define $xs +\!\!+ ys$ by primitive recursion over xs, if xs is empty, i.e. $xs = []$ then $[] +\!\!+ ys = ys$. On the other hand if xs is a *cons*, i.e. $xs = x \in xs'$ then we know that the combined list starts with x and constinues with the result of appending xs' to ys. That is here we have to use $+\!\!+$ recursively. The recursive use of $+\!\!+$ can be justified in the same way as for **Nat**, the list xs' had to be constructed before $x : xs'$, hence we can calculate the result of $xs' +\!\!+ ys$ before we have to come up with $(x : xs') +\!\!+ ys$. In Haskell we write:

$$\begin{array}{l}
(+\!\!+) \in [a] \to [a] \to [a] \\
[] \qquad\quad +\!\!+\ ys = ys \\
(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{array}$$

Note that $+\!\!+$ is a polymorphic function, we can replace the type variable a by any type. E.g. we may replace a by **Char** arriving at the special case of strings. Hence we can use $+\!\!+$ to append strings, e.g. `"Hel"` $+\!\!+$ `"lo"` $=$ `"Hello"`.

We also have an induction principle for lists. To illustrate this let's first introduce a function, which reverses a list. How to reverse a list? The reverse of the empty list is the empty list. The reverse of a list $x\!:\!xs$ is the reverse of xs with x put on the end. We arrive at the following primitive recursive implementation of reverse:

$$\boxed{\begin{array}{l} rev \in [\,a\,] \to [\,a\,] \\ rev\ [\,] = [\,] \\ rev\ (x : xs) = (rev\ xs) +\!\!+ [\,x\,] \end{array}}$$

E.g. rev `"Hello"` $=$ `"olleH"`. What happens if we reverse twice? Right, we get back where we started, indeed rev `"olleH"` $=$ `"Hello"`. Let's prove $\forall\, xs \in [\,a\,] \,.\, rev\ (rev\ xs) = xs$.

Induction over lists means that to show a property for all lists, we show the property for the empty list $[\,]$ and then we show that if it holds for any list xs then it holds for $x : xs$. The justification is again the same as for induction for the natural numbers.

We will show $\forall\, xs \in [\,a\,] \,.\, rev\ (rev\ xs) = xs$ by induction over lists. The $[\,]$ case is straightforward, since $rev\ [\,] = [\,]$, what about $rev\ (rev\ (x\!:\!xs)) = rev\ ((rev\ xs) +\!\!+ [\,x\,])$? We cannot apply the induction hypothesis directly, we first have to move the outer rev inside $+\!\!+$. The key idea is to show that $rev\ ((rev\ xs) +\!\!+ [\,x\,]) = x\!:\!(rev\ (rev\ xs))$ and then we apply the induction hypothesis that $rev\ (rev\ xs) = xs$. How do we show $rev\ ((rev\ xs) +\!\!+ [\,x\,]) = x : (rev\ (rev\ xs))$? The point is that this property doesn't just hold for $rev\ xs$ but for any list ys, i.e. $rev\ ((ys +\!\!+ [\,x\,]) = x\!:\!(rev\ ys))$, e.g. rev (`"Hall"` $+\!\!+$ `"o"`) $=$ `"o"` $:$ (rev `"Hal"`). How to we show that this property holds? By induction over ys!

Ok, that's the sketch. Let's tidy this up. First we prove the auxilliary property as a lemma:

Lemma 5.1 $\forall\, x \in a \,.\, \forall\, ys \in [\,a\,] \,.\, rev\ ((ys +\!\!+ [\,x\,]) = x : (rev\ ys))$

Proof: Given $x \in a$. We show $\forall\, ys \in [\,a\,] \,.\, rev\ ((ys +\!\!+ [\,x\,]) = x : (rev\ ys))$ by induction over $ys \in [\,a\,]$.

Case $ys = [\,]$ We show $rev\ ([\,] +\!\!+ [\,x\,]) = x : (rev\ [\,])$.

$$\begin{array}{l} rev\ ([\,] +\!\!+ [\,x\,]) \\ \;=\quad \{\text{ definition of } +\!\!+ \} \\ rev\ [\,x\,] \\ \;=\quad \{\text{ definition of } rev \} \\ [\,x\,] \\ \;=\quad \{\text{ definition of } rev \} \\ x : (rev\ [\,]) \end{array}$$

Case $ys = y : ys'$ We assume

$$rev\ ((ys +\!\!+ [\,x\,]) = x : (rev\ ys)) \quad (IH)$$

to show

$rev\ (((y : ys) +\!\!+ [x]) = x : (rev\ (y : ys)))$

$$\begin{array}{l}
rev\ (((y : ys) +\!\!+ [x]) \\
\ = \quad \{ \text{ definition of } +\!\!+ \} \\
rev\ (y : (ys +\!\!+ [x])) \\
\ = \quad \{ \text{ definition of } rev \} \\
(rev\ (ys +\!\!+ [x])) +\!\!+ [y] \\
\ = \quad \{ \text{ (IH) } \} \\
(x : (rev\ ys)) +\!\!+ [y] \\
\ = \quad \{ \text{ definition of } +\!\!+ \} \\
x : ((rev\ ys) +\!\!+ [y]) \\
\ = \quad \{ \text{ definition of } rev \} \\
x : (rev\ (y : ys))
\end{array}$$

□

Theorem 5.2 $\forall\, xs \in [a] \,.\, rev\ (rev\ xs) = xs$

Proof: By induction over xs.

Case $xs = [\,]$ We show $rev\ (rev\ [\,]) = [\,]$

$$\begin{array}{l}
rev\ (rev\ [\,]) \\
\ = \quad \{ \text{ definition of } rev \} \\
rev\ [\,] \\
\ = \quad \{ \text{ definition of } rev \} \\
[\,]
\end{array}$$

Case $xs = x : xs'$ We assume

$rev\ (rev\ xs) = xs \quad (IH)$

to show

$rev\ (rev\ (x : xs')) = x : xs'$

$$\begin{array}{l}
rev\ (rev\ (x : xs')) \\
\ = \quad \{ \text{ definition of } rev \} \\
rev\ ((rev\ xs') +\!\!+ [x]) \\
\ = \quad \{ \text{ lemma for } ys = rev\ xs' \} \\
x : (rev\ (rev\ xs)) \\
\ = \quad \{ \text{ (IH) } \} \\
x : xs
\end{array}$$

□

5.2 Insertion sort

There are many situations, where we want to *sort* a list, e.g. given $[56, 1, 10, 3]$ we would like to produce $[1, 3, 10, 56]$. One of the simplest sorting algorithms is *insertion sort*. The idea can be best explained by describing how to sort a deck of card. We sort from an unsorted pile to a sorted pile by adding one card at a time to the sorted pile. When adding a card to a already sorted pile we go through the sorted pile until we find a card with a higher value and then insert the new card just before this card. This can be expressed as a function *insert*, which is defined by primitive recursion over lists:

```
insert ∈ Nat → [Nat] → [Nat]
insert a []       = [a]
insert a (b : bs) = if a ⩽ b then a : (b : bs)
                    else b : (insert a bs)
```

The important feature of *insert* is that it produces a sorted list with an extra element, if its argument was sorted. We can now define *insertionSort* by primitive recursion over the initially unsorted list:

```
insertionSort ∈ [Nat] → [Nat]
insertionSort []       = []
insertionSort (a : as) = insert a (insertionSort as)
```

And indeed *insertionSort* $[56, 1, 10, 3] = [1, 3, 10, 56]$.

The same algorithm can sort any data wrt. to a given comparison function *comp* : $a \to a \to$ **Bool**, which has the appropriate properties (we shall later discuss in detail what properties this are).

Insertion sort shouldn't be used to sort large sequences, because it is quite inefficent: on average it compares every argument with half of all the others. We say it has *quadratic complexity*, because on average the number of comparisons it has to carry out is proportional to n^2, where n is the length of the list.

5.3 Binary numbers

We can use lists to represent binary numbers. A binary number like 110_2, which is the binary way to write 6, is nothing but a list of Booleans, hence we define:

```
type Binary = [Bool]
```

We use True for 1 and False for 0 and we shall read the list backwards, which has the advantage that it is easy to access the least significant digit, e.g. 110_2 is actually represented as [False, True, True]. [14]

We shall define functions, which translate between **Binary** and **Nat**. We start with *bin2nat* ∈ **Binary** → **Nat**: The idea is that to translate a number which has at least one digit, we recursively translate the rest of the numebr without the last digit, double that and add the value of the last digit to it. This is expressed by the following function:

```
bin2nat ∈ Binary → Nat
bin2nat []           = 0
bin2nat (True : bs)  = Succ (2 ∗ (bin2nat bs))
bin2nat (False : bs) = 2 ∗ (bin2nat bs)
```

[14]We could have avoided this notational inconvenience by using snoc lists, i.e. lists where cons gets its arguments the other way around.

To define the translation in the other direction, $nat2bin \in \mathbf{Nat} \to \mathbf{Binary}$, we introduce a successor function on the binary representation. The successor of the empty list, which is interpreted as 0 is 1. Otherwise, if the last digit is 0 we just change this to 1 and keep the rest of the number the same. If the last digit is 1, we change this to 0 and have a carry bit, which means that we have to recursively calculate the successor of the rest of the number. Hence we define:

$$\boxed{\begin{array}{ll} \multicolumn{2}{l}{bsucc \in \mathbf{Binary} \to \mathbf{Binary}} \\ bsucc\ [\,] & = [\text{True}] \\ bsucc\ (\text{False} : bs) & = \text{True} : bs \\ bsucc\ (\text{True} : bs) & = \text{False} : (bsucc\ bs) \end{array}}$$

Now we can define *nat2bin*, it basically translates 0 by the empty binary and then applies *bsucc* for every successor.

$$\boxed{\begin{array}{ll} \multicolumn{2}{l}{nat2bin \in \mathbf{Nat} \to \mathbf{Binary}} \\ nat2bin\ 0 & = [\,] \\ nat2bin\ (\text{Succ}\ n) & = bsucc\ (nat2bin\ n) \end{array}}$$

Once we have established the two translations, we want to show that they are indeed inverse to each other. E.g. if we start with a natural number n we can translate it to a binary $nat2bin\ n$, and then translate it back by $bin2nat\ (nat2bin\ n)$ and we should hope that we get back to the number we started with, i.e. that

$bin2nat\ (nat2bin\ n) = n.$

If we start with a binary number $bs \in \mathbf{Binary}$, the mirror property

$nat2bin\ (bin2nat\ bs) = bs$

doesn't actually hold. The reason is that our encoding is redundant, leading 0s (actually trailing Falses in our notation) do not matter. For example $nat2bin\ (bin2nat\ [\text{True}, \text{False}]) = [\text{True}]$. We could insist the we do not allow leading 0s and show that for those *normal* numbers the property holds. Actually, following common practive we would make the exception for the case that the number is just 0 but would at the same time outlaw the empty sequence as a representation of a number.

Going back to proving $\forall\ n \in \mathbf{Nat}.n = bin2nat\ (nat2bin\ n)$, unsurprisingly we are going to use induction to establish this. The case for 0 is straightforward but in the case of Succ n' we have that $nat2bin\ (\text{Succ}\ n') = bsucc\ (nat2bin\ n')$ and we would like to use that

$bin2nat\ (bsucc\ (nat2bin\ n')) = \text{Succ}\ (bin2nat\ (nat2bin\ n'))$

to be able to apply the induction hypothesis. This leads to the following lemma:

Lemma 5.3 $\forall\ bs \in \mathbf{Binary}.bin2nat\ (bsucc\ bs) = \text{Succ}\ (bin2nat\ bs)$

Proof: We show this by induction over $bs \in \mathbf{Binary}$.

Case $bs = [\,]$ We have to show $bin2nat\ (bsucc\ [\,]) = \text{Succ}\ (bin2nat\ [\,])$:

$$\begin{array}{l} bin2nat\ (bsucc\ [\,]) \\ =\quad \{\ \text{definition of } bsucc\ \} \end{array}$$

$$\begin{aligned}
&\mathit{bin2nat}\ [\mathrm{True}] \\
&= \quad \{ \text{ calculate } \mathit{bin2nat} \ \} \\
&1 \\
&= \quad \{ \text{ definition of } 1 \ \} \\
&\mathrm{Succ}\ 0 \\
&= \quad \{ \text{ definition of } \mathit{bin2nat} \ \} \\
&\mathrm{Succ}\ (\mathit{bin2nat}\ [\,])
\end{aligned}$$

Case $bs = b : bs'$ We assume

$\mathit{bin2nat}\ (\mathit{bsucc}\ bs') = \mathrm{Succ}\ (\mathit{bin2nat}\ bs') \quad (\mathit{IH})$

to show

$\mathit{bin2nat}\ (\mathit{bsucc}\ (b : bs')) = \mathrm{Succ}\ (\mathit{bin2nat}\ (b : bs'))$

for any $b \in$ **Bool**. Let's analyze $b \in$ **Bool**:

Case $b = \mathrm{False}$ In this case we have to show:

$\mathit{bin2nat}\ (\mathit{bsucc}\ (\mathrm{False} : bs')) = \mathrm{Succ}\ (\mathit{bin2nat}\ (\mathrm{False} : bs'))$:

$$\begin{aligned}
&\mathit{bin2nat}\ (\mathit{bsucc}\ (\mathrm{False} : bs')) \\
&= \quad \{ \text{ definition of } \mathit{bsucc} \ \} \\
&\mathit{bin2nat}\ (\mathrm{True} : bs') \\
&= \quad \{ \text{ definition of } \mathit{bin2nat} \ \} \\
&\mathrm{Succ}\ (2 * (\mathit{bin2nat}\ bs')) \\
&= \quad \{ \text{ definition of } \mathit{bin2nat} \ \} \\
&\mathrm{Succ}\ (\mathit{bin2nat}\ (\mathrm{False} : bs'))
\end{aligned}$$

Case $b = \mathrm{True}$ In this case we have to show:

$\mathit{bin2nat}\ (\mathit{bsucc}\ (\mathrm{True} : bs')) = \mathrm{Succ}\ (\mathit{bin2nat}\ (\mathrm{True} : bs'))$

$$\begin{aligned}
&\mathit{bin2nat}\ (\mathit{bsucc}\ (\mathrm{True} : bs')) \\
&= \quad \{ \text{ definition of } \mathit{bsucc} \ \} \\
&\mathit{bin2nat}\ (\mathrm{False} : (\mathit{bsucc}\ bs')) \\
&= \quad \{ \text{ definition of bin2nat } \} \\
&2 * (\mathit{bin2nat}\ (\mathit{bsucc}\ bs')) \\
&= \quad \{ \text{ (IH) } \} \\
&2 * (\mathrm{Succ}\ (\mathit{bin2nat}\ bs')) \\
&= \quad \{ \text{ algebra: } 2 * (\mathrm{Succ}\ n) = \mathrm{Succ}\ (\mathrm{Succ}\ (2 * n)) \ \} \\
&\mathrm{Succ}\ (\mathrm{Succ}\ (2 * (\mathit{bin2nat}\ bs')) \\
&= \quad \{ \text{ definition of } \mathit{bin2nat} \ \} \\
&\mathrm{Succ}\ (\mathit{bin2nat}\ (\mathrm{True} : bs'))
\end{aligned}$$

□

Now we are ready to prove the main theorem:

Theorem 5.4 $\forall\ n \in$ **Nat**.$\mathit{bin2nat}\ (\mathit{nat2bin}\ n) = n$

Proof: We prove this by induction over $n \in$ **Nat**:

Case $n = 0$ We show $\mathit{bin2nat}\ (\mathit{nat2bin}\ 0) = 0$:

$$\begin{aligned}
&\mathit{bin2nat}\ (\mathit{nat2bin}\ 0) \\
&= \quad \{ \text{ definition of } \mathit{nat2bin} \ \} \\
&\mathit{bin2nat}\ [\,]
\end{aligned}$$

$$\begin{aligned}
&= \quad \{ \text{ definition of } \mathit{bin2nat} \ \} \\
&0
\end{aligned}$$

Case $n = \text{Succ}\ n'$ We assume

$\mathit{bin2nat}\ (\mathit{nat2bin}\ n') = n' \quad (\mathit{IH})$

to show

$\mathit{bin2nat}\ (\mathit{nat2bin}\ (\text{Succ}\ n')) = \text{Succ}\ n'$

$$\begin{aligned}
&\mathit{bin2nat}\ (\mathit{nat2bin}\ (\text{Succ}\ n')) \\
&= \quad \{ \text{ definition of } \mathit{nat2bin} \ \} \\
&\mathit{bin2nat}\ (\mathit{bsucc}\ (\mathit{nat2bin}\ n')) \\
&= \quad \{ \text{ lemma } \mathbf{??} \ \} \\
&\text{Succ}\ (\mathit{bin2nat}\ (\mathit{nat2bin}\ n')) \\
&= \quad \{ \text{ (IH) } \} \\
&\text{Succ}\ n'
\end{aligned}$$

□

To define addition on **Binary** we define addition with carry, which always adds three bits caculating the current bit and the new carry bit. As a first step we define two operations *addbit* and *carry* which calculate the new bit and carry.

$$\begin{aligned}
&\mathit{addbit} \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} \\
&\mathit{addbit}\ a\ b\ c = (a \equiv b) \equiv c
\end{aligned}$$

$$\begin{aligned}
&\mathit{carry} \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool} \\
&\mathit{carry}\ a\ b\ c = (a\&\&b)||(b\&\&c)||(a\&\&c)
\end{aligned}$$

We are now ready to define binary addition with carry:

$$\begin{aligned}
&\mathit{addBinC} \in \mathbf{Bool} \to \mathbf{Binary} \to \mathbf{Binary} \to \mathbf{Binary} \\
&\mathit{addBinC}\ c\ \mathit{as}\ \ [\,] \quad = \mathbf{if}\ c\ \mathbf{then}\ \mathit{bsucc}\ \mathit{as}\ \mathbf{else}\ \mathit{as} \\
&\mathit{addBinC}\ c\ [\,]\ \ \mathit{as} \quad = \mathbf{if}\ c\ \mathbf{then}\ \mathit{bsucc}\ \mathit{as}\ \mathbf{else}\ \mathit{as} \\
&\mathit{addBinC}\ c\ (a : \mathit{as})\ (b : \mathit{bs}) = (\mathit{addbit}\ a\ b\ c) : (\mathit{addBinC}\ (\mathit{carry}\ a\ b\ c)\ \mathit{as}\ \mathit{bs})
\end{aligned}$$

from which we can easily derive binary addition by setting the carry to False:

$$\begin{aligned}
&\mathit{addBin} \in \mathbf{Binary} \to \mathbf{Binary} \to \mathbf{Binary} \\
&\mathit{addBin}\ \mathit{bs}\ \mathit{cs} = \mathit{addBinC}\ \text{False}\ \mathit{bs}\ \mathit{cs}
\end{aligned}$$

Binary multiplication is the usual long multiplication we learn at school. We multiply each digit of one of the numbers with the other numbers always multiplying the other number by 10, i.e. shifting it. In the binary case multiplying a digit with a number is very easy, either the bit is 0 (i.e. False) then the result is 0 or the bit is 1 then the result just is the other number.

$$\begin{aligned}
&\mathit{multBin} \in \mathbf{Binary} \to \mathbf{Binary} \to \mathbf{Binary} \\
&\mathit{multBin}\ [\,]\ \mathit{bs} = [\,] \\
&\mathit{multBin}\ (a : \mathit{as})\ \mathit{bs} = \mathbf{if}\ a \\
&\qquad\qquad \mathbf{then}\ \mathit{addBin}\ \mathit{bs}\ (\mathit{multBin}\ \mathit{as}\ (\text{False} : \mathit{bs})) \\
&\qquad\qquad \mathbf{else}\ \mathit{multBin}\ \mathit{as}\ (\text{False} : \mathit{bs})
\end{aligned}$$

The operations we have defined correspond to the more primitive operations we have defined previously. That is we have:

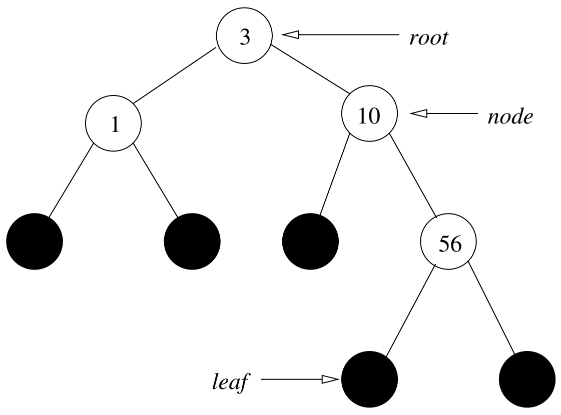
$\forall\ m, n : \mathbf{Nat}\,.\ \mathit{addBin}\ (\mathit{bin2nat}\ m)\ (\mathit{bin2nat}\ n) = \mathit{bin2nat}\ (m + n)$

$\forall\ m, n : \mathbf{Nat}\,.\ \mathit{multBin}\ (\mathit{bin2nat}\ m)\ (\mathit{bin2nat}\ n) = \mathit{bin2nat}\ (m * n)$

These can be established using the principles we have introduced, i.e. induction. However, for reasons of space and potential boredom we shall refrain from carrying this out here.

5.4 Trees

Unlike in ordinary live, trees in computer science have got the roots on top and the leaves on bottom. Here is an example of a binary tree whose nodes are labelled with natural numbers:



Trees are also a very useful datastructure, for example they can be used to efficently store and update an ordered collection of data. In Haskell we would define the type of binary, node-labelled trees as follows:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

The tree pictured in the diagram would be represented as:

```
mytree ∈ Tree Nat
mytree = Node (Node Leaf 1 Leaf)
  3
  (Node Leaf
        10
        (Node Leaf 56 Leaf))
```

You may note that the tree is ordered, i.e. all the leaves in the left subtree of any node has labels less (or equal) than the label on the node and the nodes in the right subtree have greater labels. Indeed, ordered trees give rise to another sorting algorithm, which is called *tree sort*. First we define, by primitive recursion over trees, a function which inserts a node into an ordered tree, leaving the ordering intact.

```
$treeInsert \in \mathbf{Nat} \to (\mathbf{Tree\ Nat}) \to \mathbf{Tree\ Nat}$

$treeInsert\ a$ Leaf           $=$ Node Leaf $a$ Leaf
$treeInsert\ a$ (Node $l\ b\ r$) $=$ **if** $a \leqslant b$
                               **then** Node $(treeInsert\ a\ l)\ b\ r$
                               **else**  Node $l\ b\ (treeInsert\ a\ r)$
```

By inserting all elements of a list into an empty tree, we obtain a function which turns an unsorted list into an ordered tree.

```
$list2tree \in [\mathbf{Nat}] \to (\mathbf{Tree\ Nat})$

$list2tree\ [\,] =$ Leaf
$list2tree\ (a : as) = treeInsert\ a\ (list2tree\ as)$
```

We can also turn an ordered tree into a sorted list.

```
$tree2list \in (\mathbf{Tree}\ a) \to [a]$

$tree2list$ Leaf $= [\,]$
$tree2List$ (Node $l\ a\ r) = (tree2list\ l) +\!\!+ [a] +\!\!+ (tree2list\ r)$
```

Note, that the last function is polymorphic since it doesn't need any operations on the data stored in a tree. Now we obtain *tree sort* by combining the two functions defined above:

```
$treeSort \in [\mathbf{Nat}] \to [\mathbf{Nat}]$

$treeSort\ as = tree2list\ (list2tree\ as)$
```

It may appear wasteful to create a tree just to sort a list but actually *tree sort* is better than *insertion sort*. On the average every element which is inserted into the tree is only compared with a few others on the way down to a leaft, to be precise $\log_2 n$, if the number of nodes is n. Since we have to insert n elements *tree sort*, will need a number proportional to $n \log_2 n$ comparisons to sort a list with n elements, which is considerably better that insertion sort's n^2.

We can also prove propositions about trees by induction over trees. As an example observe that the tree in the example has 4 nodes and 5 leaves. This is no coincidence but a general property of binary trees. Maybe the best way to see that this is the case is to notice that every tree can be generated starting from a leaf by replacing an aribtrary leaf by a node with two leaves. The initial tree has got the proberty and it is preserved by the replacement step, hence any tree will have the property. More formally, we can show this by induction over trees, where we have an induction hypothesis for the left and the right subtree.

To make this precise we first define functions which count the number of nodes and leaves in a tree (ignoring any data):

```
$countNodes \in (\mathbf{Tree}\ a) \to \mathbf{Nat}$

$countNodes$ Leaf           $= 0$
$countNodes$ (Node $l\ x\ r$) $=$ Succ $((countNodes\ l) + (countNodes\ r))$
```

```
$countLeaves \in (\mathbf{Tree}\ a) \to \mathbf{Nat}$

$countLeaves$ Leaf $= 1$
$countLeaves$ (Node $l\ x\ r$) $= (countLeaves\ l) + (countLeaves\ r)$
```

We are now ready to prove our theorem:

Theorem 5.5 $\forall\ t : \textbf{Tree}\ a\ .\ \mathit{countLeaves}\ t = 1 + (\mathit{countNodes}\ t)$

Proof: We proceed by induction over $t \in \textbf{Tree}\ a$:

Case $t = \text{Leaf}$ We show $\mathit{countLeaves}\ \text{Leaf} = 1 + (\mathit{countNodes}\ \text{Leaf})$

$$\begin{aligned}
&\mathit{countLeaves}\ \text{Leaf}\\
&= \quad \{ \text{ definition of } \mathit{countLeaves} \ \}\\
&1\\
&= \quad \{ \text{ definition of } \mathit{countNodes} \ \}\\
&1 + (\mathit{countNodes}\ \text{Leaf})
\end{aligned}$$

Case $t = \text{Node}\ l\ x\ r$ We assume

$\mathit{countLeaves}\ l = 1 + (\mathit{countNodes}\ l) \quad (\mathit{IH} - L)$

$\mathit{countLeaves}\ r = 1 + (\mathit{countNodes}\ r) \quad (\mathit{IH} - R)$

to show $\mathit{countLeaves}\ (\text{Node}\ l\ x\ r) = 1 + (\mathit{countNodes}\ (\text{Node}\ l\ x\ r))$

$$\begin{aligned}
&\mathit{countLeaves}\ (\text{Node}\ l\ x\ r)\\
&= \quad \{ \text{ definition of } \mathit{countLeaves} \ \}\\
&(\mathit{countLeaves}\ l) + (\mathit{countLeaves}\ r)\\
&= \quad \{ \text{ (IH-L),(IH-R) } \}\\
&1 + (\mathit{countNodes}\ l) + 1 + (\mathit{countNodes}\ r)\\
&= \quad \{ \text{ algebra } \}\\
&1 + (1 + (\mathit{countNodes}\ l) + (\mathit{countNodes}\ r))\\
&= \quad \{ \text{ definition of } \mathit{countNodes} \ \}\\
&1 + (\mathit{countNodes}\ (\text{Node}\ l\ x\ r))
\end{aligned}$$

□

Finally note that there are different types of binary trees, depending where we put the data, i.e. there are leaf labelled trees:

$$\boxed{\textbf{data Tree}\ a = \text{Leaf}\ a \mid \text{Node}\ (\textbf{Tree}\ a)\ (\textbf{Tree}\ a)}$$

and trees which have labels on leaves and nodes, potentially of different types.

$$\boxed{\textbf{data Tree}'\ a\ b = \text{Leaf}'\ a \mid \text{Node}'\ (\textbf{Tree}'\ a\ b)\ b\ (\textbf{Tree}'\ a\ b)}$$

6 Predicates and relations

A predicate over a type is a property of elements of that type. An example is $\mathbf{Even} \in \mathbf{Nat} \to \mathbf{Prop}$ which expresses the property of a natural number being even. We can define **Even**

$$\mathbf{Even} \in \mathbf{Nat} \to \mathbf{Prop}$$
$$\mathbf{Even}\ n = \exists\ m \in \mathbf{Nat} \,.\, 2 * m = n$$

That is we say that a number is even, if it is the double of another number. I write $a \to \mathbf{Prop}$ for the type of predicates over a. Note that unlike programs predicates cannot be executed but we can use them for reasoning.

A relation between two types (which may be the same) relates elements of the first type with elements of the 2nd. An example is $\leqslant\ \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$ which expresses the relation of a number being smaller than another. Again we can define $\leqslant$ using $+$:

$$(\leqslant) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$$
$$m \leqslant n = \exists\ k \in \mathbf{Nat} \,.\, m + k = n$$

That is we say that $m \leqslant n$, if there is a number k that if added to m gives n.

In general we consider n-ary relations, that is $R \in a_1 \to a_2 \to \cdots \to a_n \to \mathbf{Prop}$ and predicates are just another word for 1-ary relations. 2-ary (or binary) relations such as $\leqslant$ are usually written infix.

A fundamental binary relation on any type is equality $(=) \in a \to a \to \mathbf{Prop}$. We consider equality as a primitive notion. We know that everything is equal to itself — we say that equality is reflexive, that is $\forall\ x \in a \,.\, x = x$. Two booleans or natural number or lists are equal if they are made only from constructors and look the same. Two functions are equal if they return the same answers for all elements of their domain, that is for $f, g \in a \to b$, we have that $f = g$, if $\forall\ x : a \,.\, f\ x = g\ x$, note that this involves equality on b. This is the principle of extensionality and it is the consequence of our view of functions as black boxes, i.e. we cannot look inside a function, all we can do it is to apply it to arguments.

We have already introduced the functions $\leqslant \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Bool}$, $\equiv \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$ and $\equiv \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Bool}$. They are related to $\leqslant\ \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Prop}$, $= \in \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Prop}$ and $= \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$, indeed they return True, if the relation holds and False otherwise. That is, we can show that

$$\forall\ x, y \in \mathbf{Nat} \,.\, x \leqslant y \iff x \leqslant y = \mathrm{True}$$
$$\forall\ x, y \in \mathbf{Bool} \,.\, x = y \iff x \equiv y = \mathrm{True}$$
$$\forall\ x, y \in \mathbf{Nat} \,.\, x = y \iff x \equiv y = \mathrm{True}$$

We say that a boolean function $f \in a \to b \to \mathbf{Bool}$ **decides** a relation $R \in a \to b \to \mathbf{Prop}$, if $\forall\ x \in a, y \in b \,.\, R\ x\ y \iff f\ x\ y = \mathrm{True}$. A relation which can be decided is called **decidable**.This concept generalizes to relations with any arity, including predicates. I have silently assumed that by a boolean function we mean one which can actually run on a computer, this is different in classical Mathematics, where the notion of a function is more liberal but the definition of a decidable function is more complicated.

Not every relation is decidable, an example of an **undecidable** relation is equality on functions with an infinite domain, e.g. $(=) \in (\mathbf{Nat} \to \mathbf{Bool}) \to (\mathbf{Nat} \to \mathbf{Bool}) \to \mathbf{Prop}$. To decide whether two such functions are equal, that is whether $f = g$ we have to check whether they agree on all natural numbers. By the principle of extensionality there is no other way we can find out anything about a function and checking them for all natural numbers is hardly possible.

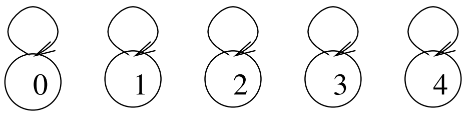
6.1 Examples of relations

In the following we will discuss some examples of relations. It is helpful to visualize relations by drawing their graphs. We can only draw graphs for a decidable relation $R \in a \to a \to \mathbf{Prop}$ on finite type a by a directed graph whose nodes (called vertices) are the elements of a and we draw an arrow (called edge) from x to y, if $R\ x\ y$ holds.

We cannot draw relations on the natural numbers, however, we will instead use the type $\text{Four} = \{0, 1, 2, 3, 4\}$ and drwaw the graph of the relation restricted to the numbers upto 4.

6.1.1 Equality

The graph of $(=) \in \text{Four} \to \text{Four} \to \mathbf{Prop}$ is extremely simple:

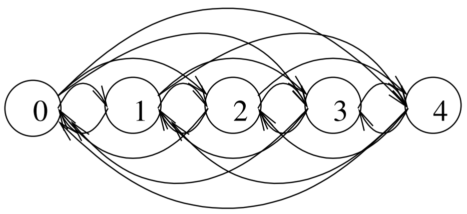


6.1.2 Inequality

We say $x \neq y$ if x and y are not equal. This relation is polymorphic, i.e. it is uniformly defined for any type.

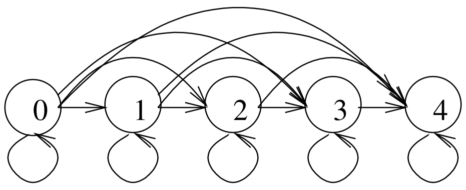
$$(\neq) \in a \to a \to \mathbf{Prop}$$
$$x \neq y = \neg\,(x = y)$$

The graph of $(\neq) \in \text{Four} \to \text{Four} \to \mathbf{Prop}$ contains an edge everywhere the previous graph had none, hence it looks pretty wild:



6.1.3 Less or equal

We have already defined the relation $(\leqslant)$ on the natural numbers. Here is the graph of the relation restricted to Four:

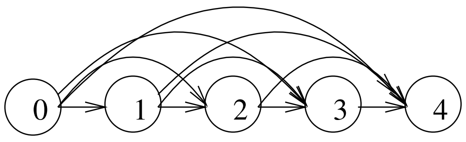


6.1.4 Less than

We say $m < n$ if m is strictly less than n.

$$(<) \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$$
$$m < n = \mathrm{Succ}\ m \leqslant n$$

The graph of $(<) : \mathrm{Four} \to \mathrm{Four} \to \mathbf{Prop}$ looks pretty similar as the one of $(\leqslant)$, only the little circles pointing from every element to itself are missing.



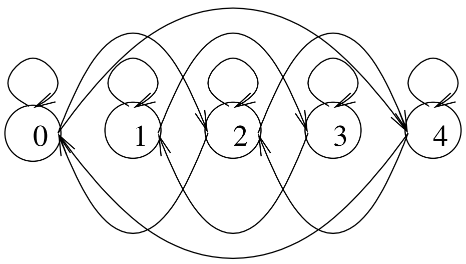
6.1.5 Congruence modulo k

This is an important 3-ary (ternary) relation used in number theory. We say that two numbers $m, n : \mathbf{Nat}$ are congruent modulo k, if they have the same remainder when divided by k. We write $cmod : \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$, that is we write $cmod\ k\ m\ n$ or $m`cmod\ k`n$ [15] , for m is congruent to n modulo k. The standard mathematical notation is $m\lambda cong\ n\ mod\ k$. We can define this relation formally as:

$$cmod \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$$
$$cmod\ k\ m\ n = \exists\ p, q, r \in \mathbf{Nat}\,.\,(r < n)\ \wedge\ p * k + r = m\ \wedge\ q * k + r = n$$

Usually $cmod$ is only considered for $k > 1$. For $k = 1$ our definition of $cmod\ k$ is just $(=)$ and for $k = 0$ it is the empty relation. In number theory $cmod\ k$ is defined a relation on the integers, not just the natural numbers.

While we cannot draw ternary relations so easily, we can do so for $mod\ 2 \in \mathrm{Four} \to \mathrm{Four} \to \mathbf{Prop}$:



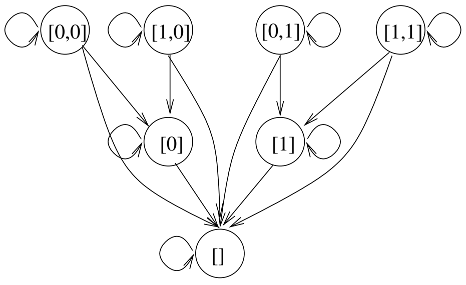
[15] Here we adopt also the Haskell convention that any operator f can be made infix by writing `$f$`.

6.1.6 Suffix of a list

We say that a list xs is a suffix of ys, we write $xs \preceq ys$, if it is occurs at the end of the list. To be precise we define

$$(\preceq) \in [\,a\,] \to [\,a\,] \to \textbf{Prop}$$
$$xs \preceq ys = \exists\, zs \,.\, ys = zs +\!\!+\, xs$$

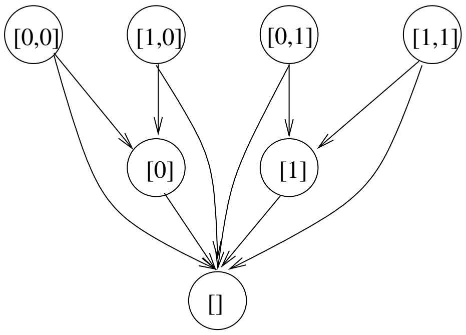
This time we consider lists built from $\{0, 1\}$ with no more than two elements for a finitisation if the relation:



There is also a strict version of the suffix relation:

$$(\prec) \in [\,a\,] \to [\,a\,] \to \textbf{Prop}$$
$$xs \prec ys = \exists\, x \in a \,.\, x : xs \preceq ys$$

and here is its graph



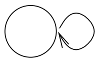
6.2 Equivalence relations

An important class of relations are *equivalence relations*, which group similar things together. Formally, an equivalence relation $R \in a \to a \to \textbf{Prop}$ has the following properties:

Reflexivity Every element is similar to itself.

$$\forall\, x \in a \,.\, R\; x\; x$$

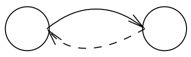
We can recognize the graph of a reflexive relation by the little circles on **each** node.



Symmetry Similarity is symmetric:

$$\forall\, x, y \in a \,.\, R\ x\ y \implies R\ y\ x$$

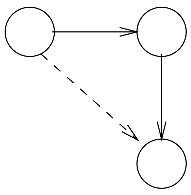
We can see that for every arrow in the graph there is one going into the other direction.



Transitivity If one thing is similar to a second, and the second is similar to a third then the first thing is similar to the third.

$$\forall\, x, y, z \in a \,.\, R\ x\ y \implies R\ y\ z \implies R\ x\ z$$

We recognize transitivity in the graph by noticing that all *shortcuts* exist:



Of the relations we have discussed above equality = and *cmod k* for any k are equivalence relations. $\neq, <, \preceq$ are not equivalence relations, because they are not reflexive. It is a common mistake to think that $\neq$ is transitive but it isn't: $1 \neq 2$ and $2 \neq 1$ but $1 \neq 1$ doesn't hold. $\leqslant, \preceq$ are reflexive and transitive but not symmetric, e.g. $2 \leqslant 3$ but $3 \leqslant 2$ does not hold.

6.3 Order relations

Another important class of relations are orders, when we order objects we can either include equality as in $\leqslant$ or not as in $<$. We will concentrate on the former case here but have a closer look on relations auch as $<$ in the next section. We say a relation is a *partial order*, if it is reflexive, transitive and antisymmetric. The first two we have already seen in the previous section.

Reflexivity

$$\forall\, x \in a \,.\, R\ x\ x$$

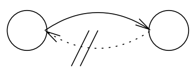
Transitivity

$$\forall\, x, y, z \in a \,.\, R\ x\ y \implies R\ y\ z \implies R\ x\ z$$

Antisymmetry Antisymmetry is what makes the relation an order relation, if you can go one way you cannot come back. However, since we have included equality, we have to allow for the possibility that we went from one element to itself.

$$\forall\, x, y \in a \,.\, R\ x\ y \implies R\ y\ x \implies x = y$$

Looking at the graph we can easily recognize an antisymmetric relation by the fact that if there is an arrow one way there is never one going back:



Of the relations we have introduced so far, $\leqslant$ and $\preceq$ are partial orders. $\leqslant$ is even a *total order*, because it has the property

Totality

$$\forall\, x, y : a\, .\, R\ x\ y\ \vee\ R\ y\ x$$

> We can recognize the graph of a total order by noticing that between any two nodes there is an arrow going one way or another.

Indeed, for two numbers it is the case that one of them os less or equal the the other. However $\preceq$ is not total, e.g. [1] and [0] are not suffixes of each other.

7 Subset and Quotients

7.1 Subsets and comprehension

With every predicate, like **Even**:**Nat** $\rightarrow$ **Prop** we associate a new type, the type of elements which satisfy the predicate. We define a new type by *comprehension* and write $\{ n \in \mathbf{Nat} \mid \mathbf{Even}\ n \}$ for the type of even numbers: E.g. we have

$$\{ n \in \mathbf{Nat} \mid \mathbf{Even}\ n \} = \{ 0, 2, 4, ... \}$$

We call this a subset, because this construction comes from set theory and isn't usually associated with types. E.g. Haskell doesn't support subsets or subtypes.

In general, given a type A and a predicate $P \in A \rightarrow \mathbf{Prop}$ we can construct a subset by *comprehension*, we write $\{ a \in A \mid P\ a \}$. We write $S \subseteq A$ to express that S is a subset of A, in particular we have that $\{ a \in A \mid P\ a \} \subseteq A$. Given $S \subseteq A$ we define a relation

$$\begin{aligned}
&(\in) \in A \rightarrow S \rightarrow \mathbf{Prop} \\
&x \in \{ a \in A \mid P\ a \} = P\ x
\end{aligned}$$

You may notice that we have used $\in$ before to indicate membership of a type. We overload this symbol now to also mean the subset membership relation. Our understanding of subsets is extensional: all we can see about a subset is which elements it contains. Hence two subsets with the same elements are the equal: $(\forall\ a \in A. a \in P \iff a \in Q) \implies P = Q$.

We define a finite subset by enumerating the elements, e.g.

$$\begin{aligned}
&\mathit{onetwothree} \subseteq \mathbf{Nat} \\
&\mathit{onetwothree} = \{ 1, 2, 3 \}
\end{aligned}$$

Actually this can be understood as a special case of comprehension — i.e. we can read the definition above as a shorthand for

$$\mathit{onetwothree} = \{ n \in \mathbf{Nat} \mid n = 1\ \vee\ n = 2\ \vee\ n = 3 \}$$

We write $\mathcal{P}\ A$ for the the powerset, i.e. type of all subsets of A. That is every subset $P \subseteq A$ gives rise to an element of the powerset $P \in A$. In the case of finite types we can enumerate the powerset, e.g.

$$\mathcal{P}\ \mathbf{Bool} = \{ \{ \}, \{ \text{False} \}, \{ \text{True} \}, \{ \text{False}, \text{True} \} \}$$

We can iterate powersets, e.g.

$$\begin{aligned}
\mathcal{P}\ (\mathcal{P}\ \mathbf{Bool}) = \{ & \{ \}, \{ \{ \} \}, \{ \{ \text{False} \} \}, \{ \{ \}, \{ \text{False} \} \}, \{ \{ \text{True} \} \}, \{ \{ \}, \{ \text{True} \} \}, \\
& \{ \{ \text{False} \}, \{ \text{True} \} \}, \{ \{ \}, \{ \text{False} \}, \{ \text{True} \} \}, \{ \{ \text{False}, \text{True} \} \}, \\
& \{ \{ \}, \{ \text{False}, \text{True} \} \}, \{ \{ \text{False} \}, \{ \text{False}, \text{True} \} \}, \{ \{ \}, \{ \text{False} \}, \{ \text{False}, \text{True} \} \}, \\
& \{ \{ \text{True} \}, \{ \text{False}, \text{True} \} \}, \{ \{ \}, \{ \text{True} \}, \{ \text{False}, \text{True} \} \}, \\
& \{ \{ \text{False} \}, \{ \text{True} \}, \{ \text{False}, \text{True} \} \}, \{ \{ \}, \{ \text{False} \}, \{ \text{True} \}, \{ \text{False}, \text{True} \} \} \}
\end{aligned}$$

In the case of a finite type with n elements, the powerset has 2^n elements.

On subsets we define a relation, the subset relation:

$$\begin{aligned}
&(\subseteq) \in \mathcal{P}\ A \rightarrow \mathcal{P}\ A \rightarrow \mathbf{Prop} \\
&P \subseteq Q = \forall\ a \in A. a \in P \implies a \in Q
\end{aligned}$$

As before $\in$ we use $\subseteq$ in two situations; first to say that P is a subset of a given type $P \subseteq A$ and second as a relation between two subsets of the same type.

$\subseteq$ is a partial order on subsets, it is

reflexive $P \subseteq P$ holds because $\forall\ a \in A.a \in P \implies a \in P$ is a tautology

transitive Given $P \subseteq Q$ and $Q \subseteq R$ we know that $\forall\ a \in A.a \in P \implies a \in Q$ and $\forall\ a \in A.a \in Q \implies a \in R$. From this we can prove $\forall\ a \in A.a \in P \implies a \in R$, which means that $P \subseteq R$

antisymmetric Given $P \subseteq Q$ and $Q \subseteq P$ we have that $\forall\ a \in A.a \in P \implies a \in Q$ and $\forall\ a \in A.a \in Q \implies a \in P$. This implies $\forall\ a \in A.a \in Q \iff a \in P$ and by extensionality we have $P = Q$.

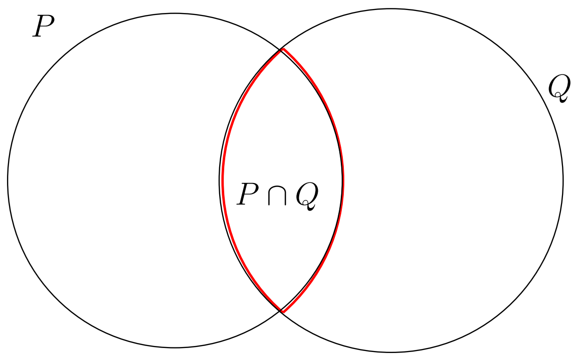
It is not a total order, e.g. $\{\text{True}\}, \{\text{False}\} \subseteq$ **Bool** but neither $\{\text{True}\} \subseteq \{\text{False}\}$ nor $\{\text{False}\} \subseteq \{\text{True}\}$ holds.

7.2 Operations on subsets

We commonly use the following operations on subsets which can be illustrated by Venn diagrams, where each region represents a subset.

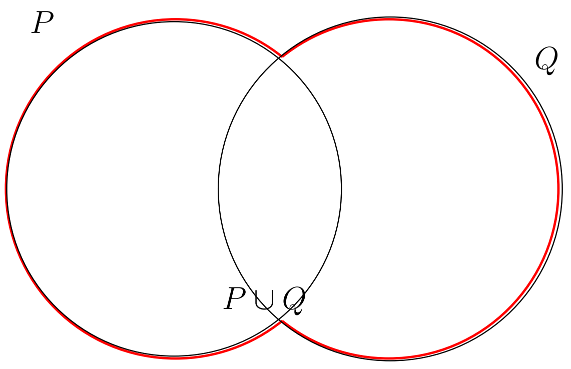
intersection The elements in the intersection of P and Q, $P \cap Q$, are the ones which are in both subsets.

$$(\cap) \in \mathcal{P}\ A \to \mathcal{P}\ A \to \mathcal{P}\ A$$
$$P \cap Q = \{a \in A \mid a \in P \land a \in Q\}$$



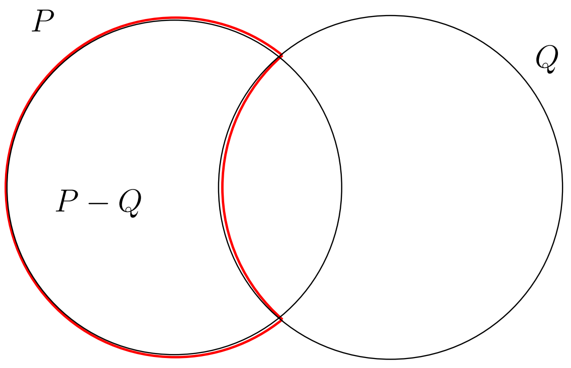
union The elements in the union of P and Q, $P \cup Q$, are the ones which are in either subset.

$$(\cup) \in \mathcal{P}\ A \to \mathcal{P}\ A \to \mathcal{P}\ A$$
$$P \cup Q = \{ a \in A \mid a \in P \lor a \in Q \}$$

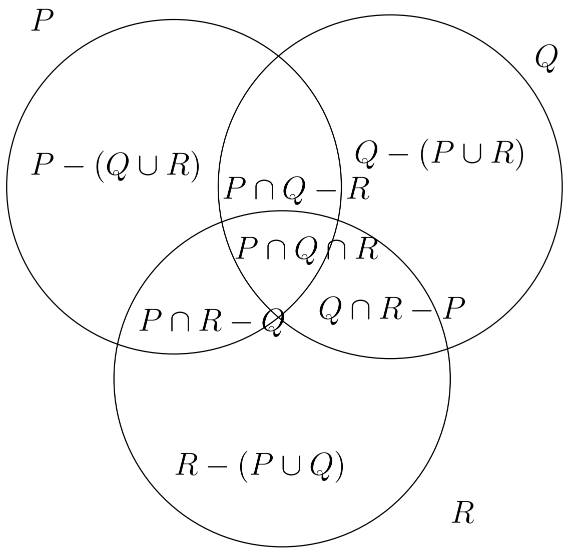


difference The elements in the difference of P and Q, $P - Q$, are the ones which are in P but not in Q.

$$(-) \in \mathcal{P}\ A \to \mathcal{P}\ A \to \mathcal{P}\ A$$
$$P - Q = \{ a \in A \mid a \in P \land \neg\, a \in Q \}$$



Venn diagrams also work fine for three sets, e.g. we can mark all the areas created by three circles with the appropriate subsets.



7.3 Finite subsets as lists

We can represent the enumeration of a finite subset as a list, e.g. we can represent $\{1, 2, 3\}$ by the list $[1, 2, 3]$. The basic predicates on sets can be decided on this representation and the operations from the previous section can be implemented as operations on lists, if the equality of elements is decidable. For simplicity we shall carry out this construction only for finite sets of natural numbers, where we have already seen that equality can be decided.

We first implement a function deciding set membership (that is $(\in) \in \mathbf{Nat} \to \mathcal{P}\ \mathbf{Nat} \to \mathbf{Prop}$) on the list representation:

$$\begin{aligned}
&\mathit{mem} \in \mathbf{Nat} \to [\mathbf{Nat}] \to \mathbf{Bool}\\
&\mathit{mem}\ n\ [\,] = \text{False}\\
&\mathit{mem}\ n\ (m : \mathit{ms}) = \textbf{if}\ n \equiv m\ \textbf{then}\ \text{True}\\
&\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{mem}\ n\ \mathit{ms}
\end{aligned}$$

Using *mem* we can decide $(\subseteq) \in \mathcal{P}\ \mathbf{Nat} \to \mathcal{P}\ \mathbf{Nat} \to \mathbf{Prop}$ by checking whether every element of the first list belongs to the second.

$$\begin{aligned}
&\mathit{subseteq} \in [\mathbf{Nat}] \to [\mathbf{Nat}] \to \mathbf{Bool}\\
&\mathit{subseteq}\ [\,]\ \mathit{ns} = \text{True}\\
&\mathit{subseteq}\ (m : \mathit{ms})\ \mathit{ns} = \mathit{mem}\ m\ \mathit{ns} \&\& \mathit{subseteq}\ \mathit{ms}\ \mathit{ns}
\end{aligned}$$

Does subseteq implement a partial order? We can show that the relation decided by *subseteq* is reflexive and transitive but it is not antisymmetric, e.g. we have that *subseteq* $[1, 2, 3]\ [3, 2, 1] = \text{True}$ and *subseteq* $[3, 2, 1]\ [1, 2, 3] = \text{True}$ but clearly $[1, 2, 3] \neq [3, 2, 1]$.

We shall use precisely this observation to decine set equality, i.e. a function whether two lists are equal as representations of sets, i.e. whether they contain the same elements:

$$\mathit{seteq} \in [\mathbf{Nat}] \to [\mathbf{Nat}] \to \mathbf{Bool}$$

$\quad$ *seteq ms ns* = *subseteq ms ns*&&*subseteq ns ms*

The relation decided by seteq is an equivalence relation. It is reflexive and transitive because *subseteq* is and it is symmetric by definition.

We implement $\cap$ on lists by checking for every element of the first lists whether it is a member of the second and only returning those which path the test:

$$\begin{array}{l}
\mathit{intersect} \in [\mathbf{Nat}] \to [\mathbf{Nat}] \to [\mathbf{Nat}] \\
\mathit{intersect}\ [\,]\ \mathit{ns} \qquad\quad = [\,] \\
\mathit{intersect}\ (m : \mathit{ms})\ \mathit{ns} = \mathbf{if}\ \mathit{mem}\ m\ \mathit{ns}\ \mathbf{then}\ m : (\mathit{intersect}\ \mathit{ms}\ \mathit{ns}) \\
\qquad\qquad\qquad\qquad\quad\ \ \mathbf{else}\ \mathit{intersect}\ \mathit{ms}\ \mathit{ns}
\end{array}$$

We could implement $\cup$ on the list representation simply as $+\!\!+$ but this may lead to duplicates, e.g. $[1,2,3] +\!\!+ [2,3,4] = [1,2,3,3,4,4]$ containing duplicates. An better implementation only uses those elements of the first subset that are not elements of the second.

$$\begin{array}{l}
\mathit{union} \in [\mathbf{Nat}] \to [\mathbf{Nat}] \to [\mathbf{Nat}] \\
\mathit{union}\ [\,] \qquad\quad \mathit{ns} = \mathit{ns} \\
\mathit{union}\ (m : \mathit{ms})\ \mathit{ns} = \mathbf{if}\ \mathit{mem}\ m\ \mathit{ns}\ \mathbf{then}\ \mathit{union}\ \mathit{ms}\ \mathit{ns} \\
\qquad\qquad\qquad\qquad\ \mathbf{else}\ m : (\mathit{union}\ \mathit{ms}\ \mathit{ns})
\end{array}$$

Indeed, we now have *union* $[1,2,3]\ [2,3,4] = [1,2,3,4]$.

The implementation of set difference proceeds similarily, we only return those elements of the first list that are not elements of the second:

$$\begin{array}{l}
\mathit{setminus} \in [\mathbf{Nat}] \to [\mathbf{Nat}] \to [\mathbf{Nat}] \\
\mathit{setminus}\ [\,] \qquad\quad \mathit{ns} = [\,] \\
\mathit{setminus}\ (m : \mathit{ms})\ \mathit{ns} = \mathbf{if}\ \mathit{mem}\ m\ \mathit{ns}\ \mathbf{then}\ \mathit{setminus}\ \mathit{ms}\ \mathit{ns} \\
\qquad\qquad\qquad\qquad\quad\ \mathbf{else}\ m : (\mathit{setminus}\ \mathit{ms}\ \mathit{ns})
\end{array}$$

7.4 Quotients

In the previous section we used the type of lists to implement finite sets. However, we would want to identify two lists which are equal as sets. This construction is known as a quotient in set theory while in Computer Science the term *abstract data type* is used. In this case we shall use the mathematical terminology. Lets write **SetEq** $\in$ [**Nat**] $\to$ [**Nat**] $\to$ **Prop** for the relation decided by seteq, i.e.

$$\begin{array}{l}
\mathbf{SetEq} \in [\mathbf{Bool}] \to [\mathbf{Bool}] \to \mathbf{Prop} \\
\mathbf{SetEq}\ \mathit{bs}\ \mathit{cs} = \forall\ b \in \mathbf{Bool}\,.\ \mathit{mem}\ b\ \mathit{bs} = \mathit{mem}\ b\ \mathit{cs}
\end{array}$$

We introduce FinSet as the quotient of [**Nat**] by **SetEq**, that is we look at [**Nat**] but we identify lists which represent equal sets

$$\mathbf{type}\ \mathbf{FinSet} = [\mathbf{Nat}]\ /\ \mathbf{SetEq}$$

A list of natural numbers $\mathit{ms} : [\mathbf{Nat}]$ gives rise to an element of the quotient $<\!\mathit{ms}\!> \in \mathbf{FinSet}$, two lists which are identified by **SetEq** are identified as finite sets, that is if **SetEq** *ms ns* then $<\!\mathit{ms}\!> = <\!\mathit{ns}\!>$ holds.

To define a predicate or a function on **FinSet** we first define it on [**Nat**] and then show that it is invariant under **SetEq**. E.g. we can use *mem* on **FinSet** after showing that if **SetEq** *ms ns* then for all $n \in \mathbf{Nat}$ it is the case that *mem n ns* = *mem n ns*. Hence *mem* gives rise to $mem' \in \mathbf{Nat} \to \mathbf{FinSet} \to \mathbf{Bool}$. We can show that all the operations defined in the previous section, i.e. *subseteq*,*intersect*,*union* and *setminus* can be lifted to **FinSet**. However, we shall not verify this in detail here.

In general given a type a and an equivalence relation $R \in a \to a \to \mathbf{Prop}$ we introduce the quotient type $a \,/\, R$, whose elements $<x> \in a \,/\, R$ can be constructed from $x \in a$. Two elements of the quotients are equal $<x> = <y>$ if $R\ x\ y$ holds. We can lift predicates or functions from a to $a\,/\,R$ after verifying that they are invariant under R.

Quotients are useful to define the number types which we have ignored so far. E.g. we represent integers as pairs of natural numbers, which represent their difference, e.g. $(2, 1)$ represents $2 - 1 = 1$ while $(1, 2)$ represents $1 - 2 = -1$. However, different pairs represent the same integers, e.g $(1, 0)$ also represents $1 - 0 = 1$ and $(0, 1)$ also represents $0 - 1 = -1$. We would like to say that (a, b) and (c, d) represent the same integer, if $a - b = c - d$, however subtraction already requires integers which we are just about to define. However, we can save the situation by using the equivalent condition that $a + d = c + b$. That leads to the following definition

> **type IntRep** = (**Nat**, **Nat**)
>
> **IntEq** $\in$ **IntRep** $\to$ **IntRep** $\to$ **Prop**
>
> **IntEq** $(a, b)\ (c, d) = a + d = c + d$
>
> **type Int** = **IntRep** / **IntEq**

We can implement the standard arithmetical operation, on the integers, by first defining it in **IntRep**. We start with addition, certainly $a - b + c - d = a + c - (b + d)$ hence

> *intAdd* $\in$ **IntRep** $\to$ **IntRep** $\to$ **IntRep**
>
> *intAdd* $(a, b)\ (c, d) = (a + c, b + d)$

To define multiplication we observe that $(a-b)*(c-d) = a*c+b*d-(b*c+a*d)$ hence

> *intMult* $\in$ **IntRep** $\to$ **IntRep** $\to$ **IntRep**
>
> *intMult* $(a, b)\ (c, d) = (a * c + b * d, b * c + a * d)$

For both function we can show that they are invariant under **IntEq**, e.g. if **IntEq** $i\ i'$ and **IntEq** $j\ j'$ then **IntEq** (*intAdd* $i\ j$) (*intAdd* $i'\ j'$) and also **IntEq** (*intMult* $i\ j$) (*intMult* $i'\ j'$). Hence we can use them to define $(+) \in \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}$ and $(*) \in \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}$. We also can define additive inverses by

> *minus* $\in$ **IntRep** $\to$ **IntRep**
>
> *minus* $(a, b) = (b, a)$

which gives rise to $(\sim) \in \mathbf{Int} \to \mathbf{Int}$. Now we can define

> $(-) \in \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}$
>
> $\quad i - j = i + {\sim}j$

A similar idea works for the rational numbers, e.g. a rational number can be represented as a pair of an integer and a natural number (i, n) standing for $i \,/\, (n + 1)$. Here we avoid the complication of dvision by zero by adding one to the denominator. Two representations (i, n) and (j, m) represent the same number if $i \,/\, (n + 1) = j \,/\, (m + 1)$ and as before with subtraction we can avoid division by instead requiring $i * (m + 1) = j * (n + 1)$. Hence we define

$$\begin{aligned}
&\textbf{type RatRep} = (\textbf{Int}, \textbf{Nat})\\
&\textbf{RatEq} \in \textbf{RatRep} \to \textbf{RatRep} \to \textbf{Prop}\\
&\textbf{RatEq}\ (i, n)\ (j, m) = i * (m + 1) = j * (n + 1)\\
&\textbf{type Rat} = \textbf{RatRep} \,/\, \textbf{RatEq}
\end{aligned}$$

I leave it to the reader to work out how to define arithmetical operations on rational numbers. We will also use the function $\mathit{abs} : \textbf{Rat} \to \textbf{Rat}$, which calcuates the absolute value of a rational number.

Let's just have a quick look at the Reals, following Cauchy a real number can be represented as an infinite sequence of rational numbers such that the absolute difference between the subsequent elements of the sequence gets arbitrarily small. We use functions $\textbf{Nat} \to \textbf{Rat}$ to represent infinite sequences. Two real numbers are equal if the sequence of differences gets arbitrarily small. Hence we define:

$$\begin{aligned}
&\textbf{type RealRep} = \{\mathit{seq} \in \textbf{Nat} \to \textbf{Rat}\\
&\quad \mid \forall\, r : \textbf{Rat}.r \geqslant 0 \implies \exists\, i \in \textbf{Nat}.\mathit{abs}\ (\mathit{seq}\ (i + 1) - \mathit{seq}\ i) \leqslant r\}\\
&\textbf{RealEq}\ \mathit{seq}\ \mathit{seq}' = \forall\, r : \textbf{Rat}.r \geqslant 0 \implies \exists\, i \in \textbf{Nat}.\mathit{abs}\ (\mathit{seq}\ i - \mathit{seq}'\ i) \leqslant r\\
&\textbf{Real} = \textbf{RealRep} \,/\, \textbf{RealEq}
\end{aligned}$$

It is interesting to note that in the case of **Int** and **Rat** we can actually avoid the use of quotients, i.e. we can represent integers as a pair $(\textbf{Bool}, \textbf{Nat})$ where the boolean indicates whether the number is negative — we an easily avoind redundant representations of 0 by decreeing that -1 is represented by $(\mathrm{True}, 0)$ and 0 as $(\mathrm{False}, 0)$. In the case of **Rat** we use pairs that have no common divisors, i.e. we replace quotients by subset. However, in the case of **Real** we cannot avoid the use of quotients.

8 Functions

8.1 Identity and composition

The identity function id is the function which just which just echos its input. It can be defined polymorphically at any type a:

$$id \in a \to a$$
$$id\ x = x$$

If we have two functions $f \in a \to b$ and $g \in b \to c$ we can compose them by first running f and then g giving rise to a new function $g \circ f \in a \to c$. That is for $x \in a$ we have $g \circ f\ x = g\ (f\ x)$.

As an example consider the previously defined functions:

$$length \in [a] \to \mathbf{Nat}$$
$$\begin{aligned} &length\ [\,] = 0 \\ &length\ (a : as) = 1 + length\ as \end{aligned}$$
$$even \in \mathbf{Nat} \to \mathbf{Bool}$$
$$\begin{aligned} &even\ 0 = \text{True} \\ &even\ (n+1) = not\ (f\ n) \end{aligned}$$

What is $even \circ length \in [a] \to \mathbf{Bool}$? Basically, this function calculates the length of a list an then determines whether it is even. We can *fuse* the two functions into one:

$$evenLength \in [a] \to \mathbf{Bool}$$
$$\begin{aligned} &evenLength\ [\,] = \text{True} \\ &evenLength\ (a : as) = not\ (evenLength\ as) \end{aligned}$$

To show that $even \circ length = evenLength$ it is sufficent to show

$$\forall\, as \in [a] \,.\, even \circ length\ as = evenLength\ as$$

using the principle of extensionality. Unfolding the definition of $\circ$ this boils down to

$$\forall\, as \in [a] \,.\, even \circ length\ as = evenLength\ as$$

which can be shown by induction over as.

Maybe you have noticed that composition is written backwards. When we execute $even \circ length$ we first calculate the length and then determine whether it was even. Wouldn't it be more natural to write $length$ first and then $even$ that is since we read from left to right, we would write $length$ left from $even$ as in $length; even$. However, consequently we should then also change the way we write function application, because when we actually calculate $even\ (length\ [1,2,3])$ we first have to calculate $length\ [1,2,3] = 3$ and then $even\ 3 = \text{False}$. Indeed, the way we write composition merely reflects the way we write application. Actually, it has been suggested to write both application and composition the other, more intuitive, way around. However, since most people write it backwards, we follow this convention for ease of communication.

8.2 Higher order functions

$\circ$ is itself a function, that is

$$(\circ) \in (b \to c) \to (a \to b) \to a \to c$$
$$(f \circ g)\ a = f\ (g\ a)$$

$(\circ)$ is a higher order function, that is it is a function on functions: it takes two functions as inputs and returns a function. In general we could define the order of a type as follows (here $*$ is the type of ordinary types):

$$\begin{array}{ll}
\mathit{order} \in * \to \mathbf{Nat} & \\
\mathit{order}\ \mathbf{Nat} & = 0 \\
\mathit{order}\ \mathbf{Bool} & = 0 \\
\mathit{order}\ \emptyset & = 0 \\
\mathit{order}\ [a] & = \mathit{order}\ a \\
\mathit{order}\ (a, b) & = \mathit{max}\ (\mathit{order}\ a)\ (\mathit{order}\ b) \\
\mathit{order}\ (a + b) & = \mathit{max}\ (\mathit{order}\ a)\ (\mathit{order}\ b) \\
\mathit{order}\ (a \to b) & = \mathit{max}\ (1 + (\mathit{order}\ a))\ (\mathit{order}\ b)
\end{array}$$

I said we could because this sort of definition of a function by recursion over the type is not valid in Haskell. It has the intrinsic problem that each time we want to define a new type (e.g. like **Tree**), we have to extend the function. It also raises the question whether $* \in *$ — remember the trouble the barber was having?

Leaving this aside, as a consequence any instance of $(\circ)$ has an order of at least 2. I say it least, because I haven't assign orders to type variables, because they could be instantiated at any type and hence could have any order. However, most people are a bit sloppy and say that $(\circ)$ is a 2nd order function.

E.g. a function like $\mathit{even} \in \mathbf{Nat} \to \mathbf{Bool}$ is a first order function, because its type has order 1:

$$\begin{array}{l}
\mathit{order}\ (\mathbf{Nat} \to \mathbf{Bool}) \\
= \quad \{\ \text{definition of } \mathit{order}\ \} \\
\mathit{max}\ (1 + (\mathit{order}\ \mathbf{Nat}))\ (\mathit{order}\ \mathbf{Bool}) \\
= \quad \{\ \text{definition of } \mathit{order} \text{ and } +\ \} \\
\mathit{max}\ 1\ 0 \\
= \quad \{\ \text{definition of } \mathit{max}\ \} \\
1
\end{array}$$

Note that $\mathit{add} \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat}$ is also first order — currying does not increase the order of the type.

If we instantiate the type variables of $\circ$ by data, i.e. by types of order 0, e.g. $(\circ) \in (\mathbf{Nat} \to \mathbf{Bool}) \to ([\mathbf{Nat}] \to \mathbf{Nat}) \to [\mathbf{Nat}] \to \mathbf{Bool}$ then $(\circ)$ is 2nd order. Can we find functions of even higher order? Yes, just consider composing compositions. To be precise lets fix the first argument in the example above $(\circ)\ \mathit{even} \in ([\mathbf{Nat}] \to \mathbf{Nat}) \to [\mathbf{Nat}] \to \mathbf{Bool}$ and use it again as the first argument to $(\circ)$ as in

$$(\circ)\ ((\circ)\ \mathit{even}) \in (a \to ([\mathbf{Nat}] \to \mathbf{Nat})) \to a \to ([\mathbf{Nat}] \to \mathbf{Bool})$$

which has an order of at least 3 depending on the order of the type variable a. What is this function actually doing? It expects as argument a function which returns a function of type $[\mathbf{Nat}] \to \mathbf{Nat}$ and it transforms this into a function which returns $[\mathbf{Nat}] \to \mathbf{Bool}$ instead by composing with even. It should be clear that we can repeat this process any number of times reaching types of any order. However, intuition will be lost quickly.

8.3 The category of types

Composition and identity have some simple properties:

$$\begin{aligned}
&\forall\, f \in a \to b \,.\, f \circ id = f\\
&\forall\, f \in a \to b \,.\, id \circ f = f\\
&\forall\, f \in a \to b, g \in b \to c, h \in c \to d \,.\, (h \circ g) \circ f = h \circ (g \circ f)
\end{aligned}$$

All of these are easy to verify, e.g. to show the first $\forall\, f \in a \to b \,.\, f \circ id = f$ we assume as given $f \in a \to b$ and to show $f \circ id = f$ using extensionality reduces to showing $\forall\, x \in a \,.\, f \circ id\ x = f\ x$. That is given $x \in a$ we have

$$\begin{aligned}
&f \circ id\ x\\
&= \quad \{ \text{ definition of } \circ \ \}\\
&f\ (id\ x)\\
&= \quad \{ \text{ definition of } id \ \}\\
&f\ x
\end{aligned}$$

The verification of the other properties proceed in a similar manner.

We say that types and functions form a *category*. This is the central notion of *category theory* which was introduced by Saunders McLane and others to be able to concisely state generic properties of mathematical constructions. Category theory is nowadays used extensively in Computer Science because it offers a convenient way to express and reason about concepts which are central in computing.

To catch but a glimpse of this power lets discuss some basic concepts of category theory using the catgory of types as an example. Lists can be constructed at any type, hence we may say that $\mathbf{List}\ a = [a]$ is actually a function on types, i.e. $\mathbf{List} \in * \to *$. Even better **List** can be extended to act on functions, that is we can define

$$\begin{aligned}
&map \in (a \to b) \to [a] \to [b]\\
&map\ f\ [\,] = [\,]\\
&map\ f\ (a : as) = (f\ a) : (map\ f\ as)
\end{aligned}$$

What is map doing? E.g. $map\ even \in [\mathbf{Nat}] \to [\mathbf{Bool}]$ applies *even* to every element of a list, that is $map\ even\ [1, 2, 3] = [\text{False}, \text{True}, \text{False}]$. Note that *map* is a higher order function, it has at least order 2.

map preserves identity and composition, that is we have

$$\begin{aligned}
map\ id &= id\\
map\ (f \circ g) &= (map\ f) \circ (map\ g)
\end{aligned}$$

We can verify these properties by using extensionality and induction on lists.

We say that **List** is a functor. In category theory one uses the same name for the operation on types and functions, that is we would write **List** f for $map\ f$.[16]

[16]Alas, this is inconsistent with the Haskell convention to use uppercase names for construc-

The function *rev* $\in [a] \to [a]$, which reverses a list, is defined uniformly on all types. We note that *rev* commutes with *map* that is

$$\forall f \in a \to b \,.\, rev \circ (map\ f) = (map\ f) \circ rev$$

This property can be expressed in a more readable form by a categorical diagram:

$$\begin{tikzcd}
\textbf{List}\ A \arrow[r, "rev"'] \arrow[d, "map\ f"] & \textbf{List}\ A \arrow[d, "map\ f"] \\
\textbf{List}\ A \arrow[r, "rev"'] & \textbf{List}\ A
\end{tikzcd}$$

We say that the diagram commutes, that is if we follow different paths we will end up with the same function. Intuitively this reflects the fact that it doesn't matter whether we first map a function and then reverse the result or first reverse and then map the same function. E.g. if we apply *map even* to $[1, 2, 4]$ we obtain $[\mathrm{False}, \mathrm{True}, \mathrm{True}]$ and after reversing we get $[\mathrm{True}, \mathrm{True}, \mathrm{False}]$. If on the other hand we first reverse $[1, 2, 4]$ we obtain $[4, 2, 1]$ and after applying *map even* we also end up with $[\mathrm{True}, \mathrm{True}, \mathrm{False}]$. In general we can show this porperty by induction over lists.

We say that *rev* is a *natural transformation* from **List** to **List** because the diagram commutes. This reflects the fact that *rev* acts uniformly on all types. E.g. if we had a function $rev' \in [a] \to [a]$ which reverses lists on all types but is just the identity on **Bool**, it is easy to see that the above property would fail. In fact it is impossible to define such a function in Haskell and hence that naturality holds is called *a free theorem*.

8.4 Properties of functions

A function is **injective** (or one-to-one) if it doesn't identify any elements, e.g. the function *double* $\in$ **Nat** $\to$ **Nat** with $double\ n = 2 * n$ is injective while the function *half* $\in$ **Nat** $\to$ **Nat** with $half\ n = n$ `'div'` 2 isn't because it $half\ 2 = half\ 3 = 1$. A function is **surjective** (or onto), if it covers the whole range, e.g. *half* is surjective but *double* isn't because odd numbers like 3 are never reached by *double*. A function which is both injective and surjective is called **bijective**, neither *half* nor *double* are bijective. However, the function

$$\begin{aligned}
&switch \in \mathbf{Nat} \to \mathbf{Nat}\\
&switch\ n = \mathbf{if}\ even\ n\\
&\qquad\qquad \mathbf{then}\ n + 1\\
&\qquad\qquad \mathbf{else}\ n - 1
\end{aligned}$$

is a bijection: it sends every even number to the next odd number and every odd number to the previous even number. Here is the precise definition of the three predicates on functions:

$$\begin{aligned}
&\mathbf{Injective} \in (a \to b) \to \mathbf{Prop}\\
&\mathbf{Injective}\, f = \forall\, x, y \in a \,.\, f\ x = f\ y \implies x = y\\
&\mathbf{Surjective} \in (a \to b) \to \mathbf{Prop}\\
&\mathbf{Surjective}\, f = \forall\, y \in b \,.\, \exists\, x \in a \,.\, f\ x = y
\end{aligned}$$

tors.

$$\begin{aligned}&\mathbf{Bijective} \in (a \to b) \to \mathbf{Prop}\\&\mathbf{Bijective}\, f = \mathbf{Injective}\, f \;\wedge\; \mathbf{Surjective}\, f\end{aligned}$$

We use the nouns *injection*, *surjection* and *bijection* for functions with the corresponding properties. A bijection is just a reordering of elements, there are only bijections between finite types with the same number of elements. E.g. there is a bijection $f \in \{1,2,3\} \to \{4,5,6\}$ but there isn't any between $\{1,2\}$ and $\{4,5,6\}$. E.g. there is no surjection from $\{1,2\}$ to $\{4,5,6\}$ because we are always one element short and there is no injection from $\{4,5,6\}$ to $\{1,2\}$ because any such function will inevitably identify some elements.

We say that two types are *in bijection*, if there is a bijection between them. This is an equivalence relation on types, it is reflexive because the identity is a bijection, it is symmetric because we can turn around any bijection and it is transitive because the composition of bijections is a bijection. Two bijective types are basically the same upto renaming of elements. Using this relation we can compare infinite types and see whether they have the *same number of elements.*

Unlike for finite types, for infinite types it is not the case that adding an element makes the type different ass for as bijection is concerned. We can generically add an element to a type a by $a + ()$ where $()$ is the type with only one element $() \in ()$. However, we can construct the following bijection:

$$\begin{aligned}&\mathit{succnat} \in \mathbf{Nat} + () \to \mathbf{Nat}\\&\mathit{succnat}\ (\text{Left}\ n) \quad = n + 1\\&\mathit{succnat}\ (\text{Right}\ ()) = 0\end{aligned}$$

Even if we *double* the type, i.e. by $a + a$, we still get a bijection:

$$\begin{aligned}&\mathit{doublenat} \in \mathbf{Nat} + \mathbf{Nat} \to \mathbf{Nat}\\&\mathit{doublenat}\ (\text{Left}\ n) \quad = 2 * n\\&\mathit{doublenat}\ (\text{Right}\ n) = 2 * n + 1\end{aligned}$$

What about the square of a type, i.e. (a, a)? We can still obtain a bijection

$$\begin{aligned}&\mathit{squarenat} \in (\mathbf{Nat}, \mathbf{Nat}) \to \mathbf{Nat}\\&\mathit{sqarenat}\ (m, n) = (m + 1) * (m + n + 1)\ \texttt{'div'}\ 2 + n\end{aligned}$$

The idea of this function is to use the enumeration of pairs starting like this:

	0	1	2	3	...
0	0	1	3	6	...
1	2	4	7	...	
2	5	8	...		
3	9	...			
...	...				

As a concequence of these constructions we notice that **Nat**, **Int** and **Rat** are in bijection, i.e. have *the same number of elements*. However, we get bigger types as soon as we use function types. Indeed, we can show that there is no surjection from **Nat** to $\mathbf{Nat} \to \mathbf{Bool}$: Assume as given a function $f \in \mathbf{Nat} \to (\mathbf{Nat} \to \mathbf{Bool})$, then we can construct $g \in \mathbf{Nat} \to \mathbf{Bool}$ by $g\ n = \mathit{not}\ (f\ n\ n)$. Now for every $i \in \mathbf{Nat}$ the function $f\ i \in \mathbf{Nat} \to \mathbf{Bool}$ is different from

$g \in \mathbf{Nat} \to \mathbf{Bool}$ at i because $g\ i = \mathit{not}\ (f\ i\ i)$. Hence f cannot be surjective, because it will never return g.

This proof method is called *diagonalisation.* We can use the same method to show that there is no bijection between **Nat** and **Real**.

9 Formal proof

What is a proof? Here is an attempt at a definition:

> A **proof** is an argument which will convince a critical but not obnoxious person that a proposition is a tautology.

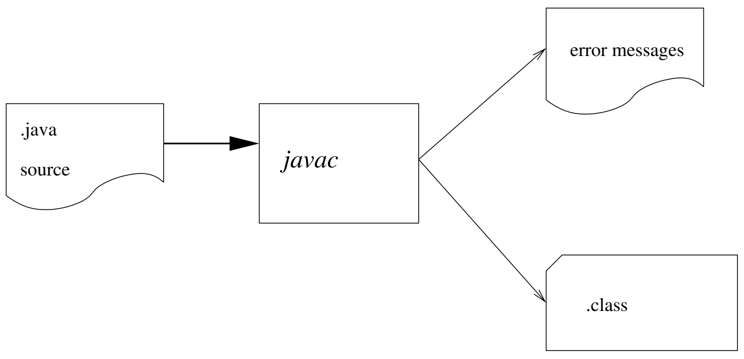
A proof in science is usually read by other scientists (who can be assumed to be critical but not obnoxious) and is accepted as correct, if all the experts agree that it is correct. However, this process may need some iterations — see Imre Lakatos' nice book *Proofs and Refutations.*

Here we are concerned with *formal proofs*, which are especially interesting for Computer Science. Many of the facts which we want to verify in Computer Science aren't interesting from a scientific point of view (*program X works correctly*) but we often want to avoid to many iterations (*program X may control a nuclear power station*). Some typical properties of formal proofs are:

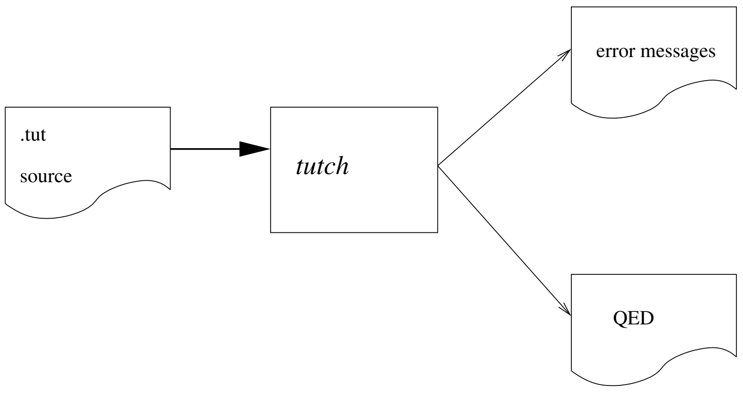
- written in a precisely defined syntax (like a program).
- uses only a fixed set of deduction rules.
- can be checked by a computer program (*a proof checker*)

In this course we are going to work with `tutch` ***tut**orial proof **ch**ecker* a proof checker for educational purposes implemented by Andreas Abel (Munich) when he was working for Frank Pfenning at Carnegie Mellon University, USA in 2001.

`tutch` works similar to a java compiler. A java compiler processes java source files (`.java`) and produces either error messages or a `.class` file.



In contrast `tutch` reads proof sources (`.tut`) and produces as output either error messages or it reports that it accepts the proof: QED (latin *quod erat demonstrandum* — what there was to be proven).



In general tutch knows two sorts of error messages:

syntax errors Like in java, tutch doesn't like it if you forget a semicolon etc.

incomplete proofs tutch complains if a step in the proof doesn't seem to be justified.

With respect to the first definition for a proof given above `tutch` behaves like a very stupid (but reasonable) person who is hard to convince that something is true. There are better tools available (e.g. COQ, Isabelle, NuPRL, ...) but its *stupidity* makes tutch ideal for educational purposes. The best way to learn the art of proofs is if you have to convince somebody (or in this case something) very stupid.

9.1 tutch syntax

A tutch proof is a sequence of propositions or frames separated by semicolons (`;`), each of them is justified by some of the previous propositions or frames. A *frame* is a sequence of propositions or *frames* which is enclosed in `[` and `]`. Frames are used for hypothetical reasoning: the first proposition in a frame is an assumption form which the following propositions can be derived.

Here is an example of a simple propositional proof of the fact $A \wedge B \implies B \wedge A$ in tutch syntax:

```
proof comAnd : A & B => B & A =
begin
[A & B;
  A;
  B;
  B & A];
A & B => B & A
end;
```

The first thing we notice is that tutch isn't actually using the symbols we have introduced previously. A simple reason for this is that they aren't available on the computer keyboard. In general `tutch` uses the following translation:

Logic	tutch
$\wedge$	`&`
$\vee$	`|`
$\neg$	`~`
$\Longrightarrow$	`=>`
True	`T`
False	`F`
$\Longleftrightarrow$	`<=>`
$\forall$	`!`
$\exists$	`?`

However, in these notes we shall use the proper symbols. The tutch proof scripts are available via links in the online version. hence the previous proof looks like that (comAnd.tut)

```
proof comAnd : A ∧ B  ⟹  B ∧ A =
begin
[A ∧ B;
  A;
  B;
  B ∧ A];
A ∧ B  ⟹  B ∧ A
end;
```

This example shows also the basic structure of a tutch proof: It starts with the word `proof` which is followed by the name of the proof (here `comAnd`) and then after : we write the proposition we are going to prove, followed by `=` and the proof. The proof starts with `begin` and ends with `end;` The last line of the proof is identical to the proposition which we were going to prove. A tutch proof script may contain several proofs and in each proof we can use the ones which were proven previously.

To run `tutch` we save the proof script in a text file using any text editor like `emacs` or `notepad` (but not something like `word`) and then run `tutch` by typing

```
  tutch proof-file.tut
```

on the UNIX command prompt. E.g. in our example we would save the proof in `comAnd.tut` and type

```
[mytutch]$ tutch comAnd.tut
```

tutch replies with [17]

```
TUTCH 0.51 beta,
[Opening file comAnd.tut]
Proving comAnd: A ∧ B  ⟹  B ∧ A ...
QED
[Closing file comAnd.tut]
```

[17]Ok, I am cheating — it will use the ASCII symbols...

We can also ask `tutch` to be a bit more explicit and tell us why it accepts our proofs by typing

```
  tutch -v proof-file.tut
```

Here `-v` stands for *verbose*. Tutch replies by

```
TUTCH 0.51 beta, $Date: 2005/10/09 20:17:05 $
[Opening file comAnd.tut]

Proving comAnd: A ∧ B  ⟹  B ∧ A ...
  1  [ A ∧ B;
  2    A;                      by AndEL 1
  3    B;                      by AndER 1
  4    B ∧ A ];                by AndI 3 2
  5  A ∧ B  ⟹  B ∧ A             by ImpI 4
QED
```

I will explain the meaning of words like `AndEL` etc in a moment. Note that the numbers after this are line numbers which refer to previous propositions (or frames) which justify the current line.

9.2 Propositional logic in tutch

`tutch` accepts a proof if each proposition can be justified from the previous propositions or frames by some basic rules. These rules were introduced by Gerhard Gentzen in the 1930ies and are referred to as Gentzen's natural deduction calculus.

9.2.1 Rules for $\wedge$

And introduction *To show $P \wedge Q$ show P and show Q.*

$$\frac{P \qquad Q}{P \wedge Q}\,\texttt{AndI}$$

An example is the line 4 in the proof above, where $B \wedge A$ is justified by having proven A and B previously in line 3 and 2 — the order doesn't matter. Note, that P and Q in the rule are placeholders for arbitrary propositions.

And elimination left *If we know $P \wedge Q$ we also know P.*

$$\frac{P \wedge Q}{P}\,\texttt{AndEL}$$

An example is line 2 where A is justified $A \wedge B$ (in line 1).

And elimination right *If we know $P \wedge Q$ we also know Q.*

$$\frac{P \wedge Q}{Q}\,\texttt{AndER}$$

An example is line 3 where B is justified again by $A \wedge B$ (in line 1).

Rules in natural deduction come in two varieties:

introduction rules These are rules which tell us how to prove something, e.g. *and introduction* (`AndI`) tells us how to prove a conjunction : *To show . . .*

elimination rules This rules tell us how to use something which we have established previously: *If we know . . .*

9.2.2 Rules for $\implies$

Frames are essential for the rule *implication introduction.* Frames are used to represent hypothetical reasoning, i.e. in line 1 of the proof `comAnd` an assumption $A \wedge B$ is made and the subsequent lines of this frame may use this assumption. The assumption cannot be used any longer after the frame is closed in line 4. However assumptions can be used in nested frames.

```
[ A; an assumption A is made.
  ⋮ A can be used here.
  [ ...
    ⋮ A can be used here as well.
  ]
  ⋮ A can be used here!
]
⋮ A cannot be used here!
```

This resembles the *scoping rules* used in programming language, e.g. in java

```
{ int i; a variable i is declared.
  ⋮  i can be used here.
  {
     ⋮ i can be used here as well.
  }
  ⋮ i can be used here!
}
⋮ i cannot be used here!
```

Implication introduction *To show $P \implies Q$ assume P and show Q.*

$$\frac{\begin{matrix}[& P & \\ & \vdots & \\ & Q &]\end{matrix}}{P \implies Q}\,\texttt{ImpI}$$

In the example `comAnd` we use this rule to justify line 5 it refers back to line 4 which is the last line of a frame — to refer to a frame tutch uses the last line of this frame.

Implication elimination *If we know $P \implies Q$ and we can show P we can also show Q.*

$$\frac{P \implies Q \qquad P}{Q}\,\texttt{ImpE}$$

This rule is not used in `comAnd`, here is an example ((monAnd.tut)):

```
proof monAnd : (A ⟹ B) ⟹ (A ∧ C ⟹ B ∧ C) =
begin
[ A ⟹ B;
  [ A ∧ C;
    A;
    C;
    B; % here we use ImpE
    B ∧ C];
  A ∧ C ⟹ B ∧ C];
(A ⟹ B) ⟹ (A ∧ C ⟹ B ∧ C)
end;
```

and here is tutch's output

```
 Proving monAnd: (A ⟹ B) ⟹ A ∧ C ⟹ B ∧ C ...
  1  [ A ⟹ B;
  2    [ A ∧ C;
  3      A;                                  by AndEL 2
  4      C;                                  by AndER 2
  5      B;                                  by ImpE 1 3
  6      B ∧ C ];                            by AndI 5 4
  7    A ∧ C ⟹ B ∧ C ];                        by ImpI 6
  8  (A ⟹ B) ⟹ A ∧ C ⟹ B ∧ C                       by ImpI 7
QED
```

There is also a rule `Hyp` which says that you can prove something which you have assumed previously.

Hypothesis *If we know P then we know P.*

$$\frac{P}{P}\,\texttt{Hyp}$$

An example for the use of `Hyp` is the following proof (I.tut) of the tautology $A \implies A$.

```
proof I: A ⟹ A =
begin
[ A;
  A]; % use Hyp here
A ⟹ A
end;
```

and here is tutch's output:

```
  Proving I: A  ⟹  A ...
  1  [ A;
  2    A ];            by Hyp 1
  3  A  ⟹  A            by ImpI 2
QED
```

9.2.3 Rules for $\vee$

Here is an example of a proof using $\vee$ — we verify $A \vee B \implies B \vee A$ (comOr.tut):

```
proof comOr : A ∨ B  ⟹  B ∨ A =
begin
[ A ∨ B;
  [ A;
    B ∨ A];
  [ B;
    B ∨ A];
  B ∨ A];
A ∨ B  ⟹  B ∨ A
end;
```

and here is tutch's ouput:

```
Proving comOr: A ∨ B  ⟹  B ∨ A ...
  1  [ A ∨ B;
  2    [ A;
  3      B ∨ A ];              by OrIR 2
  4    [ B;
  5      B ∨ A ];              by OrIL 4
  6    B ∨ A ];              by OrE 1 3 5
  7  A ∨ B  ⟹  B ∨ A            by ImpI 6
QED
```

Or introduction left *To show $P \vee Q$, show P.*

$$\frac{P}{P \vee Q}\,\texttt{OrIL}$$

This is used in line 5 in the proof above.

Or Introduction Right *To show $P \vee Q$, show Q.*

$$\frac{P}{P \vee Q}\,\texttt{OrIR}$$

This is used in line 3 in the proof above.

Or elimination *If we know $P \vee Q$ and we can show R assuming P and we can show R assuming Q then we can show R.*

$$\frac{P \vee Q \qquad \begin{matrix}[\;P \\ \vdots \\ R\;]\end{matrix} \qquad \begin{matrix}[\;Q \\ \vdots \\ R\;]\end{matrix}}{R}\,\texttt{OrE}$$

This is used in line 6 above.

9.2.4 True **and** False

There is only an introduction rule for True (which is rather obvious) and only an elimination rule for False — indeed there shouldn't be a way to prove something *false*...

True introduction *To show* True *you don't need to do anything.*

$$\frac{}{\text{True}}\,\texttt{TrueI}$$

False elimination *If we know* False *we can show anything.*

$$\frac{\text{False}}{P}\,\texttt{FalseE}$$

9.2.5 ¬ and ⟺

There are no special rules for ¬ and ⟺ there are simply defined in terms of the other connectives:

$$\begin{aligned} \neg A &= A \implies \text{False} \\ A \iff B &= (A \implies B) \wedge (B \implies A) \end{aligned}$$

As an example for ¬ let's show $\neg(A \wedge \neg A)$ (incons.tut):

```
proof incons : ¬(A ∧ ¬ A) =
begin
[ A ∧ ¬ A;
  A;
  ¬ A;
  F];
¬ (A ∧ ¬ A)
end;
```

and here is tutch's output:

```
  1  [ A ∧ ¬ A;
  2    A;                by AndEL 1
  3    ¬ A;               by AndER 1
  4    F ];              by ImpE 3 2
  5  ¬ (A ∧ ¬ A)            by ImpI 4
```

As an example for ⟺ let's show $A \implies B \implies C \iff A \wedge B \implies C$ — *If A then if B then C* is the same as *If A and B then C*. We call this lemma `curry` (curry.tut) after Haskell Curry after whom also the programming language Haskell is named.

```
proof curry : ( A ∧ B  ⟹  C )  ⟺  (A  ⟹  (B  ⟹  C)) =
begin
[ (A  ⟹  (B  ⟹  C));
  [ A ∧ B;
    A;
    B;
    B  ⟹  C;
    C];
  ( A ∧ B  ⟹  C )];
(A  ⟹  (B  ⟹  C))  ⟹  ( A ∧ B  ⟹  C );
[ A ∧ B  ⟹  C;
  [ A;
    [ B;
      A ∧ B;
      C ];
    B  ⟹  C];
  A  ⟹  (B  ⟹  C)];
( A ∧ B  ⟹  C )  ⟹  (A  ⟹  (B  ⟹  C));
( A ∧ B  ⟹  C )  ⟺  (A  ⟹  (B  ⟹  C))
end;
```

and here is tutch's output

```
 Proving curry: A ∧ B  ⟹  C  ⟺  A  ⟹  B  ⟹  C ...
  1  [ A  ⟹  B  ⟹  C;
  2    [ A ∧ B;
  3      A;                                  by AndEL 2
  4      B;                                  by AndER 2
  5      B  ⟹  C;                              by ImpE 1 3
  6      C ];                                by ImpE 5 4
  7    A ∧ B  ⟹  C ];                           by ImpI 6
  8  (A  ⟹  B  ⟹  C)  ⟹  A ∧ B  ⟹  C;                 by ImpI 7
  9  [ A ∧ B  ⟹  C;
 10    [ A;
 11      [ B;
 12        A ∧ B;                            by AndI 10 11
 13        C ];                              by ImpE 9 12
 14      B  ⟹  C ];                            by ImpI 13
 15    A  ⟹  B  ⟹  C ];                           by ImpI 14
 16  (A ∧ B  ⟹  C)  ⟹  A  ⟹  B  ⟹  C;                 by ImpI 15
 17  A ∧ B  ⟹  C  ⟺  A  ⟹  B  ⟹  C                    by AndI 16 8
QED
```

9.2.6 Using lemmas

A lemma is an auxiliary theorem which is only needed to prove some bigger theorem. E.g. if we want to prove an equivalence like $(A \lor B \implies C) \iff (A \implies C) \land (B \implies C)$ it may be convenient first to establish $(A \lor B \implies C) \implies (A \implies C) \land (B \implies C)$ and $(A \implies C) \land (B \implies C) \implies (A \lor B \implies C)$ as lemmas which can be used to establish the equivalence.

In tutch we can just use a previously established proposition in a proof (andAdj.tut):

```
proof lem1 : (A ∨ B  ⟹  C)  ⟹  (A  ⟹  C) ∧ (B  ⟹  C) =
begin
 ⋮
end;

proof lem2 : (A  ⟹  C) ∧ (B  ⟹  C)  ⟹  (A ∨ B  ⟹  C) =
begin
 ⋮
end;

proof andAdj : (A ∨ B  ⟹  C)  ⟺  (A  ⟹  C) ∧ (B  ⟹  C) =
begin
(A ∨ B  ⟹  C)  ⟹  (A  ⟹  C) ∧ (B  ⟹  C);
(A  ⟹  C) ∧ (B  ⟹  C)  ⟹  (A ∨ B  ⟹  C);
(A ∨ B  ⟹  C)  ⟺  (A  ⟹  C) ∧ (B  ⟹  C)
end;
```

and here is tutch's output for `andAdj`

```
Proving andAdj: A ∨ B  ⟹  C  ⟺  (A  ⟹  C) ∧ (B  ⟹  C) ...
  1  (A ∨ B  ⟹  C)  ⟹  (A  ⟹  C) ∧ (B  ⟹  C);           by Lemma lem1
  2  (A  ⟹  C) ∧ (B  ⟹  C)  ⟹  A ∨ B  ⟹  C;             by Lemma lem2
  3  A ∨ B  ⟹  C  ⟺  (A  ⟹  C) ∧ (B  ⟹  C)              by AndI 1 2
QED
```

9.3 Classical logic

We have seen to views on propositional logic: truth tables and formal proofs ala tutch. It is interesting to compare the two. We say a proof system is:

sound All provable propositions are tautologies.

complete All tautologies are provable.

It turns out that tutch is sound, because all the rules are sound but it is not complete. The simplest counterexample is *to be or not to be*:

$$A \lor \neg A$$

is a tautology (simple exercise in checking its truth table) but it is not provable in tutch: Since we have no assumption to play with we have to prove either A or $\neg A$ but which one should we choose if we don't know anything about A?

However, this incompleteness can be easily cured by adopting the principle of *indirect proof* also called *proof by contradiction.* In tutch this corresponds to the following rule:

indirect proof *To prove P assume $\neg P$ and show that this leads to a contradiction (*False*).*

$$\dfrac{\begin{matrix}[\quad \neg P \\ \vdots \\ \text{False} \quad]\end{matrix}}{P}\,\texttt{Class}$$

This rule is called `Class` because it distinguishes *classical logic* from *intuitionistic logic*. Accordingly any tutch proof using class has to be marked `classical proof`. Here is a classical proof of $A \vee \neg A$ (TND.tut):

```
classical proof TND : A ∨ ¬ A =
begin
[¬ (A ∨ ¬ A);
 [A;
  A ∨ ¬ A;
  F];
 ¬ A;
 A ∨ ¬ A;
 F];
A ∨ ¬ A
end;
```

and here is tutch's output:

```
Proving TND: A ∨ ¬ A ... (classically)
  1  [ ¬ (A ∨ ¬ A);
  2    [ A;
  3      A ∨ ¬ A;        by OrIL 2
  4      F ];          by ImpE 1 3
  5    ¬ A;             by ImpI 4
  6    A ∨ ¬ A;         by OrIR 5
  7    F ];           by ImpE 1 6
  8  A ∨ ¬ A            by Class 7
QED
```

9.4 Predicate logic in tutch

Additional to propositions we also have typings of the form $u : T$ where u is a term and T is a type. A proof may depend on the fact that a term has a type - however typings are not propositions.

To understand the rules for *Forall elimination* and *Exists introduction* we need to introduce *substitution*, if we have a proposition P which may contain a variable x and a term u we write $P[x := u]$ for P where x is replaced by u. Here are some examples:

$$\begin{aligned}
Q(x)[x := u] &= Q(u) \\
(\forall x.Q(x))[x := u] &= \forall x.Q(x) \\
(\forall y.R(x,y))[x := u] &= \forall y.R(u,y)
\end{aligned}$$

In the 2nd line x is not replaced because it is a bound variable.

9.5 Rules for ∀

Forall introduction *To show $\forall x : t.P$ assume as given $x : t$ and show P.*

$$\dfrac{\begin{matrix}[\quad x : t \\ \vdots \\ P \quad]\end{matrix}}{\forall x : t.P}\,\texttt{ForallI}$$

Forall elimination *If we know $\forall x : t.P(x)$ and $u : t$ then $P(u)$*

$$\dfrac{\forall x : t.P \qquad u : t}{P[x := u]}$$

As an example here is a proof that we can commute universal quantifiers: $\forall x : t.\forall y : t.P(x, y) \implies \forall y : t\forall x : t.P(x, y)$ (allCom.tut).

```
proof allCom : (∀  x:t . ∀  y:t. P(x,y)) ⟹  (∀ y:t.∀ x:t.P(x,y)) =
begin
[∀  x:t . ∀  y:t. P(x,y);
 [ y:t;
   [ x:t;
     ∀  y:t. P(x,y);
     P(x,y)];
   ∀ x:t.P(x,y)];
 ∀ y:t.∀ x:t.P(x,y)];
(∀  x:t . ∀  y:t. P(x,y))  ⟹  (∀ y:t.∀ x:t.P(x,y));
end;
```

The tutch output (`tutch -v`) shows where the rules are used:

```
  Proving allCom: (∀ x:t. ∀ y:t. P (x, y))  ⟹  ∀ y:t. ∀ x:t. P (x, y) ...
  1  [ ∀ x:t. ∀ y:t. P (x, y);
  2    [ y: t;
  3      [ x: t;
  4        ∀ y:t. P (x, y);                                    by ForallE 1 3
  5        P (x, y) ];                                        by ForallE 4 2
  6      ∀ x:t. P (x, y) ];                                    by ForallI 5
  7    ∀ y:t. ∀ x:t. P (x, y) ];                                by ForallI 6
  8  (∀ x:t. ∀ y:t. P (x, y))  ⟹  ∀ y:t. ∀ x:t. P (x, y)             by ImpI 7
QED
```

9.6 Rules for ∃

Exists introduction *To show $\exists x : t.P(x)$ construct $u : t$ and show $P(u)$.*

$$\dfrac{u : t \qquad P[x := t]}{\exists x : t.P}\,\texttt{ExistsI}$$

Exists elimination *If we know $\exists x : t.P$ and we can show Q assuming $x : t$ and P then we can show Q.*

$$\frac{\exists x : t.P \qquad \begin{array}{c}[\quad x : t, P\\ \vdots\\ Q \quad]\end{array}}{Q}\ \texttt{existsE}$$

Note the **,** in the first line of the frame. This is needed because we need to assume both: the typing and the proposition at once.

As an example for a proof involving existential quantifiers we show an interesting interaction between universal and existential quantifiers. $(\exists x : t.P(x)) \implies (\forall y : t.P(y) \implies Q) \implies Q$ (exAll.tut):

```
 proof exAll : (∃ x:t.P(x)) ⟹ (∀ y:t.P(y) ⟹ Q) ⟹ Q =
begin
[∃ x:t.P(x);
 [ ∀ y:t.P(y) ⟹ Q;
   [x:t,P(x);
    P(x) ⟹ Q;
    P(x);
    Q];
   Q];
 (∀ y:t.P(y) ⟹ Q) ⟹ Q];
(∃ x:t.P(x)) ⟹ (∀ x:t.P(x) ⟹ Q) ⟹ Q
end;
```

and here are tutch's comments on the proof:

```
Proving exAll: (∃ x:t. P x) ⟹ (∀ y:t. P y ⟹ Q) ⟹ Q ...
  1  [ ∃ x:t. P x;
  2    [ ∀ y:t. P y ⟹ Q;
  3      [ x: t, P x;
  4        P x ⟹ Q;                                    by ForallE 2 3
  5        P x;                                       by Hyp 3
  6        Q ];                                       by ImpE 4 5
  7      Q ];                                         by ExistsE 1 6
  8    (∀ y:t. P y ⟹ Q) ⟹ Q ];                            by ImpI 7
  9  (∃ x:t. P x) ⟹ (∀ x:t. P x ⟹ Q) ⟹ Q                      by ImpI 8
QED
```