2 Logic and calculational proofs

The learning outcomes of this chapter is “develop adequate notation for mathematical concepts” and “perform calculational proofs” (still in the context of “organize areas of mathematics in DSL terms”).

There will be a fair bit of theory: introducing propositional and first order logic, but also “applications” to mathematics: prime numbers, (ir)rationals, limit points, limits, etc.

```
module DSLsofMath.W02 where
import qualified DSLsofMath.AbstractFOL as FOL
import DSLsofMath.AbstractFOL (andIntro, andElimR, andElimL, notIntro, notElim)
```

2.1 Propositional Calculus

(Swedish: Satslogik[2])

Now we turn to the main topic of this chapter: logic and proofs. Our first DSL for this chapter is the language of *propositional calculus* (or logic), modelling simple propositions with the usual combinators for and, or, implies, etc. The syntactic constructs are collected in Table 2.

$a,\ b,\ c,\ \ldots$	names of propositions	
False, *True*	Constants	
And	$\wedge$	&
Or	$\vee$	|
Implies	$\Rightarrow$	
Not	$\neg$	

Table 2: Syntax for propositions

Some example propositions: $p_1 = a \wedge (\neg a)$, $p_2 = a \Rightarrow b$, $p_3 = a \vee (\neg a)$, $p_4 = (a \wedge b) \Rightarrow (b \wedge a)$. If we assign all combinations of truth values for the names, we can compute a truth value of the whole proposition. In our examples, p_1 is always false, p_2 is mixed and p_3 and p_4 are always true.

Just as we did with simple arithmetic, and with complex number expressions in chapter 1, we can model the abstract syntax of propositions as a datatype:

```
data PropCalc = Con     Bool
              | Name    String
              | And     PropCalc PropCalc
              | Or      PropCalc PropCalc
              | Implies PropCalc PropCalc
              | Not     PropCalc
```

The example expressions can then be expressed as

```
p1 = And (Name "a") (Not (Name "a"))
p2 = Implies (Name "a") (Name "b")
p3 = Or (Name "a") (Not (Name "a"))
p4 = Implies (And a b) (And b a) where a = Name "a"; b = Name "b"
```

From this datatype we can write an evaluator to *Bool* which computes the truth value of a term given an environment:

[2]Some Swe-Eng translations are collected here: `https://github.com/DSLsofMath/DSLsofMath/wiki/Translations-for-mathematical-terms`.

```
type Name = String
evalPC :: (Name → Bool) → PropCalc → Bool
evalPC = error "Exercise"   -- see 2.14 for a similar function
```

The function *evalPC* translates from the syntactic to the semantic domain. (The evaluation function for a DSL describing a logic is often called *check* instead of *eval* but here we stick to *eval*.) Here *PropCalc* is the (abstract) *syntax* of the language of propositional calculus and *Bool* is the *semantic domain*.

Alternatively, we can view (*Name* → *Bool*) → *Bool* as the semantic domain. A value of this type is a mapping from a truth table (for the names) to *Bool*. This mapping is often also tabulated as a truth table with one more "output" column.

As a first example of a truth table, consider the proposition t = *Implies* (*Con False*) a. We will use the shorter notation with just T for true and F for false. The truth table semantics of t is usually drawn as follows: one column for the name a listing all combinations of T and F, and one column for the result of evaluating the expression.

a	t
F	T
T	T

This table shows that no matter what value assignment we try for the only variable a, the semantic value is T = *True*. Thus the whole expression could be simplified to just T without changing the semantics.

If we continue with the example p_4 from above we have two names a and b which together can have any of four combinations of true and false. After the name-columns are filled, we fill in the rest of the table one operation (column) at a time. The & columns become $F\ F\ F\ T$ and finally the ⇒ column (the output) becomes true everywhere.

a	&	b	⇒	b	&	a
F	F	F	T	F	F	F
F	F	T	T	T	F	F
T	F	F	T	F	F	T
T	T	T	T	T	T	T

A proposition whose truth table output is constantly true is called a *tautology*. Thus both t and p_4 are tautologies. Truth table verification is only viable for propositions with few names because of the exponential growth in the number of cases to check: we get 2^n cases for n names. (There are very good heuristic algorithms to look for tautologies even for thousands of names — but that is not part of this course.)

What we call "names" are often called "(propositional) variables" but we will soon add another kind of variables (and quantification over them) to the calculus.

2.2 First Order Logic (predicate logic)

(Swedish: Första ordningens logik = predikatlogik)

Our second DSL is that of *First Order Logic (FOL)*. This language has two datatypes: propositions, and *terms* (new). A *term* is either a (term) *variable* (like x, y, z), or the application of a *function symbol* (like f, g) to a suitable number of terms. If we have the function symbols f of arity 2 and g of arity 3 we can form terms like $f\ (x,x)$, $g\ (y,z,z)$, $g\ (x,y,f\ (x,y))$, etc. The actual function symbols are usually domain specific — we can use rational number expressions as an example. In this case we can model the terms as a datatype:

```
data RatT = RV String | FromI Integer | RPlus RatT RatT | RDiv RatT RatT
  deriving Show
```

This introduces variables and three function symbols: *FromI* of arity 1, *RPlus*, *RDiv* of arity 2.

The propositions from *PropCalc* are extended so that they can refer to terms. We will normally refer to a *FOL* proposition as a *formula*. The names from the propositional calculus are generalised to *predicate symbols* of different arity. The predicate symbols can only be applied to terms, not to other predicate symbols or formulas. If we have the predicate symbols N of arity 0, P of arity 1 and Q of arity 2 we can form *formulas* like N, $P\ (x)$, $Q\ (f\ (x,x),y)$, etc. Note that we have two separate layers: formulas normally refer to terms, but terms cannot refer to formulas.

The formulas introduced so far are all *atomic formulas* but we will add two more concepts: first the logical connectives from the propositional calculus: *And*, *Or*, *Implies*, *Not*, and then two quantifiers: "forall" ($\forall$) and "exists" ($\exists$).

An example FOL formula:

$$\forall\ x.\ \ P\ (x) \Rightarrow (\exists\ y.\ \ Q\ (f\ (x,x),y))$$

Note that FOL can only quantify over *term* variables, not over predicates. (Second order logic and higher order logic allow quantification over predicates.)

Another example: a formula stating that function symbol *plus* is commutative:

$$\forall\ x.\ \ \forall\ y.\ \ Eq\ (plus\ (x,y), plus\ (y,x))$$

Here is the same formula with infix operators:

$$\forall\ x.\ \ \forall\ y.\ \ (x+y) == (y+x)$$

Note that == is a binary predicate symbol (written *Eq* above), while + is a binary function symbol (written *plus* above).

As before we can model the expression syntax (for FOL, in this case) as a datatype. We keep the logical connectives *And*, *Or*, *Implies*, *Not* from the type *PropCalc*, add predicates over terms, and quantification. The constructor *Equal* could be eliminated in favour of *P* "Eq" but is often included.

```
data FOL = P String [RatT]
         | Equal RatT RatT
         | And     FOL FOL
         | Or      FOL FOL
         | Implies FOL FOL
         | Not     FOL
         | FORALL String FOL
         | EXISTS  String FOL
  deriving Show
commPlus :: FOL
commPlus = FORALL "x" (FORALL "y" (Equal (RPlus (RV "x") (RV "y"))
                                         (RPlus (RV "y") (RV "x"))))
```

Quantifiers: meaning, proof and syntax. "Forall"-quantification can be seen as a generalisation of *And*. First we can generalise the binary operator to an n-ary version: And_n. To prove $And_n\ (A_1, A_2, ..., A_n)$ we need a proof of each A_i. Thus we could define $And_n\ (A_1, A_2, ..., A_n) = A_1\ \&\ A_2\ \&\ ...\ \&\ A_n$ where & is the infix version of binary *And*. The next step is to note that the formulas A_i can be generalised to $A\ (i)$ where i is a term variable and A is a unary predicate symbol. We can think of i ranging over an infinite collection of constant terms i_0, i_1, ... Then the final step is to introduce the notation $\forall\ i.\ \ A\ (i)$ for $A\ (i_1)\ \&\ A\ (i_2)\ \&\ ....$

Now, a proof of $\forall\ x.\ \ A\ (x)$ should in some way contain a proof of $A\ (x)$ for every possible x. For the binary *And* we simply provide the two proofs, but in the infinite case, we need an infinite collection of proofs. The standard procedure is to introduce a fresh constant term a and prove $A\ (a)$. Intuitively, if we can show $A\ (a)$ without knowing anything about a, we have proved $\forall\ x.\ \ A\ (x)$. Another way to view this is to say that a proof of $\forall\ x.\ \ P\ x$ is a function f from terms to proofs such that $f\ t$ is a proof of $P\ t$ for each term t.

Note that the syntactic rule for $\forall\ x.\ \ b$ is similar to the rule for a function definition, $f\ x = b$, and for anonymous functions, $\lambda x \rightarrow b$. Just as in those cases we say that the variable x is *bound* in b and that the *scope* of the variable binding extends until the end of b (but not further). The scoping of x in $\exists\ x.\ \ b$ is the same as in $\forall\ x.\ \ b$.

One common source of confusion in mathematical (and other semi-formal) texts is that variable binding sometimes is implicit. A typical example is equations: $x\hat{}2 + 2 * x + 1 == 0$ usually means roughly $\exists\ x.\ \ x\hat{}2 + 2 * x + 1 == 0$. We write "roughly" here because the scope of x very often extends to some text after the equation where something more is said about the solution x.

2.3 An aside: Pure set theory

One way to build mathematics from the ground up is to start from pure set theory and define all concepts by translation to sets. We will only work with this as a mathematical domain to study, not as "the right way" of doing mathematics (there are other ways). In this section we keep the predicate part of the version of *FOL* from the previous section, but we replace the term language *RatT* with pure (untyped) set theory.

The core of the language of pure set theory is captured by four function symbols. We have a nullary function symbol $\{\}$ for the empty set (sometimes written $\emptyset$) and a unary function symbol S for the function that builds a singleton set from an "element". All non-variable terms so far are $\{\}$, $S\ \{\}$, $S\ (S\ \{\})$, ... The first set is empty but all the others are (different) one-element sets.

Next we add two binary function symbols for union and intersection of sets (denoted by terms). Using union we can build sets of more than one element, for example $\mathit{Union}\ (S\ \{\})\ (S\ (S\ \{\}))$ which has two "elements": $\{\}$ and $S\ \{\}$.

In pure set theory we don't actually have any distinguished "elements" to start from (other than sets), but it turns out that quite a large part of mathematics can still be expressed. Every term in pure set theory denotes a set, and the elements of each set are again sets. (Yes, this can make your head spin.)

Natural numbers To talk about things like natural numbers in pure set theory they need to be encoded. FOL does not have function definitions or recursion, but in a suitable meta-language (like Haskell) we can write a function that creates a set with n elements (for any natural number n) as a term in FOL. Here is some pseudo-code defining the "von Neumann" encoding:

```
vN 0       = {}
vN (n + 1) = step (vN n)
step x = Union x (S x)
```

If we use conventional set notation we get $\mathit{vN}\ 0 = \{\}$, $\mathit{vN}\ 1 = \{\{\}\}$, $\mathit{vN}\ 2 = \{\{\},\{\{\}\}\}$, $\mathit{vN}\ 3 = \{\{\},\{\{\}\},\{\{\},\{\{\}\}\}\}$, etc. If we use the shorthand $\overline{n}$ for $\mathit{vN}\ n$ we see that $\overline{0} = \{\}$, $\overline{1} = \{\overline{0}\}$, $\overline{2} = \{\overline{0},\overline{1}\}$, $\overline{3} = \{\overline{0},\overline{1},\overline{2}\}$ and, in general, that $\overline{n}$ has cardinality n (meaning it has n elements). The function *vN* is explored in more detail in the first assignment of the DSLsofMath course.

Pairs The constructions presented so far show that, even starting from no elements, we can embed all natural numbers in pure set theory. We can also embed unordered pairs: $\{a, b\} \overset{\text{def}}{=} \mathit{Union}\ (S\ a)\ (S\ b)$ and normal, ordered pairs: $(a, b) \overset{\text{def}}{=} \{S\ a, \{a, b\}\}$. With a bit more machinery it is possible to step by step encode $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$.

A good read in this direction is "The Haskell Road to Logic, Maths and Programming" [Doets and van Eijck, 2004].

2.4 Back to quantifiers

After this detour through untyped set land, let us get back to the most powerful concept of FOL: the quantifiers. We have already seen how the "forall" quantifier can be seen as a generalisation of *And* and in the same way we can see the "exists" quantifier as a generalisation of *Or*.

First we generalise the binary *Or* to an n-ary Or_n. To prove $\mathit{Or}_n\ A_1\ A_2\ ...\ A_n$ is enough (and necessary) to find one i for which we can prove A_i. As before we then take the step from a family of formulas A_i to one unary predicate A expressing the formulas $A\ (i)$ for the term variable i. Then the final step is to "or" all these formulas to obtain $\exists\ i.\ \ A\ i$.

At this point it is good to sum up and compare the two quantifiers and how to prove them:

> (t, b_t) is a proof of $\exists\ x.\ \ P\ (x)$ if b_t is a proof of $P\ (t)$.
>
> f is a proof of $\forall\ x.\ \ P\ (x)$ if $f\ t$ is a proof of $P\ (t)$ for all t.

Curry-Howard If we abbreviate "is a proof" as : and use the Haskell convention for function application we get

$$\begin{array}{lll} (t, b_t) : (\exists\ x.\ \ P\ x) & \textbf{if} & b_t\ \ : P\ t \\ f \qquad : (\forall\ x.\ \ P\ x) & \textbf{if} & f\ t : P\ t\ \ \text{for all}\ t \end{array}$$

This now very much looks like type rules, and that is not a coincidence. The *Curry-Howard correspondence* says that we can think of propositions as types and proofs as "programs". These typing judgements are not part of FOL, but the correspondence is used quite a bit in this course to keep track of proofs.

We can also interpret the simpler binary connectives using the Curry-Howard correspondence. A proof of *And* $P\ Q$ is a pair of a proof of P and a proof of Q. Or, as terms: if $p : P$ and $q : Q$ then $(p, q) : \mathit{And}\ P\ Q$. Similarly, a proof of *Or* $P\ Q$ is either a proof of P or a proof of Q: we can pick the left (P) or the right (Q) using the Haskell datatype *Either*: if $p : P$ then $\mathit{Left}\ p : \mathit{Or}\ P\ Q$ and if $q : Q$ then $\mathit{Right}\ q : \mathit{Or}\ P\ Q$. In this way we can build up what is called "proof terms" for a large fragment of logic. It turns out that each such proof term is basically a program in a functional programming language, and that the formula a certain term proves is the type for the program.

Typed quantification In each instance of FOL, quantification is always over the full set of terms, but it is often convenient to quantify over a subset with a certain property (like all even numbers, or all non-empty sets). We will use a notation we can call "typed quantification" as a short-hand notation for the full quantification in combination with a restriction to the subset. For existential and universal quantification these are the definitions (assuming T is a property of terms, that is a unary predicate on terms):

$$(\exists\ x : T.\ \ P\ x) \overset{\text{def}}{=} (\exists\ x.\ \ T\ x\ \&\ P\ x)$$
$$(\forall\ x : T.\ \ P\ x) \overset{\text{def}}{=} (\forall\ x.\ \ T\ x \Rightarrow P\ x)$$

A good exercise is to work out the rules for "pushing negation through" typed quantification, from the corresponding rules for full quantification.

2.5 Proof by contradiction

Let's try to express and prove the irrationality of the square root of 2. We have two main concepts involved: the predicate "irrational" and the function "square root of". The square root function (for positive real numbers) can be specified by $r = \sqrt{s}$ iff $r\hat{\ }2 == s$ and $r : \mathbb{R}$. The formula "x is irrational" is just $\neg\ (R\ x)$ where R is the predicate "is rational".

$$R\ x = \exists\ a : \mathbb{N}.\ \ \exists\ b : \mathbb{N}_{>0}.\ \ b * x == a\ \&\ GCD\ (a, b) == 1$$

The classical way to prove a negation $\neg\ P$ is to assume P and derive something absurd (some Q and $\neg\ Q$, for example). Lets take $P = R\ r$ and $Q = GCD\ (a, b) == 1$. Assuming P we immediately get Q so what we need is to prove $\neg\ Q$, that is $GCD\ (a, b) \not= 1$. We can use the equations $b * r == a$ and $r\hat{\ }2 == 2$. Squaring the first equation and using the second we get $b\hat{\ }2 * 2 == a\hat{\ }2$. Thus $a\hat{\ }2$ is even, which means that a is even, thus $a == 2 * c$ for some c. But then $b\hat{\ }2 * 2 == a\hat{\ }2 == 4 * c\hat{\ }2$ which means that $b\hat{\ }2 == 2 * c\hat{\ }2$. By the same reasoning again we have that also b is even. But then $GCD\ (a, b) \geqslant 2$ which implies $\neg\ Q$.

To sum up: by assuming P we can prove both Q and $\neg\ Q$. Thus, by contradiction $\neg\ P$ must hold.

2.6 Proof by cases

As another example, let's prove that there are two irrational numbers p and q such that $p\hat{\ }q$ is rational.

$$S = \exists\ p.\ \ \exists\ q.\ \ \neg\ (R\ p)\ \&\ \neg\ (R\ q)\ \&\ R\ (p\hat{\ }q)$$

We know from above that $r = \sqrt{2}$ is irrational, so as a first attempt we could set $p = q = r$. Then we have satisfied two of the three clauses ($\neg\ (R\ p)$ and $\neg\ (R\ q)$). What about the third clause: is $x = p\hat{\ }q == r\hat{\ }r$ rational? We can reason about two possible cases, one of which has to hold: $R\ x$ or $\neg\ (R\ x)$.

Case 1: $R\ x$ holds. Then we have a proof of S with $p = q = r = \sqrt{2}$.

Case 2: $\neg\ (R\ x)$ holds. Then we have another irrational number x to play with. Let's try $p = x$ and $q = r$. Then $p\hat{\ }q == x\hat{\ }r == (r\hat{\ }r)\hat{\ }r == r\hat{\ }(r * r) == r\hat{\ }2 == 2$ which is clearly rational. Thus, also in this case we have a proof of S, but now with $p = r\hat{\ }r$ and $q = r$.

To sum up: yes, there are irrational numbers such that their power is rational. We can prove the existence without knowing what numbers p and q actually are!

2.7 Functions as proofs

To prove a formula $P \Rightarrow Q$ we assume a proof $p : P$ and derive a proof $q : Q$. Such a proof can be expressed as $(\lambda p \to q) : (P \Rightarrow Q)$: a proof of an implication is a function from proofs to proofs.

As we saw earlier, a similar rule holds for the "forall" quantifier: a function f from terms t to proofs of $P\ t$ is a proof of $\forall\ x.\ \ P\ x$.

```
module DSLsofMath.AbstractFOL where
data And  p q;                  data Or   p q
data Impl p q;                  data Not p

andIntro  :: p → q → And p q;   orElim   :: Or p q → (p → r) → (q → r) → r
andElimL  :: And p q → p;       orIntroL :: p → Or p q
andElimR  :: And p q → q;       orIntroR :: q → Or p q

implIntro :: (p → q) → Impl p q;  notIntro :: (p → And q (Not q)) → Not p
implElim  :: Impl p q → p → q;    notElim  :: Not (Not p) → p

andIntro = u; orElim = u; andElimR = u; orIntroL = u; andElimL = u; orIntroR = u;
implIntro = u; notElim = u; notIntro = u; implElim = u; u = undefined;
```

Figure 4: The Haskell module *AbstractFOL*.

As we saw in section 2.4, a very common kind of formula is "typed quantification": if a type (a set) S of terms can be described as those that satisfy the unary predicate T we can introduce the short-hand notation

$$(\forall\ x : T.\ \ P\ x) = (\forall\ x.\ \ T\ x \Rightarrow P\ x)$$

A proof of this is a two-argument function p which takes a term t and a proof of $T\ t$ to a proof of $P\ t$.

In pseudo-Haskell we can express the implication laws as follows:

```
impIntro : (A → B) → (A ⇒ B)
impElim  : (A ⇒ B) → (A → B)
```

It should come as no surprise that this "API" can be implemented by $(\Rightarrow) = (\rightarrow)$, which means that both *impIntro* and *impElim* can be implemented as *id*.

Similarly we can express the universal quantification laws as:

```
∀-Intro : ((a : Term) → P a) → (∀ x.  P x)
∀-Elim  : (∀ x.  P x) → ((a : Term) → P a)
```

To actually implement this we need a *dependent* function type, which Haskell does not provide. But we can still use it as a tool for understanding and working with logic formulas and mathematical proofs.

Haskell supports limited forms of dependent types and more is coming every year but for proper dependently typed programming I recommend the language Agda.

2.8 Proofs for *And* and *Or*

When formally proving properties in FOL we should use the introduction and elimination rules. The propositional fragment of FOL is given by the rules for $\wedge$, $\rightarrow$, $\longleftrightarrow$, $\neg$, $\vee$. We can use the Haskell type checker to check proofs in this fragment, using the functional models for introduction and elimination rules. Examine Fig. 4 (also available in the file `AbstractFOL.lhs`), which introduces an empty datatype for every connective (except $\longleftrightarrow$), and corresponding types for the introduction and elimination rules. The introduction and elimination rules are explicitly left undefined, but we can still combine them and type check the results. For example:

```
example0 :: FOL.And p q → FOL.And q p
example0 evApq = andIntro (andElimR evApq) (andElimL evApq)
```

(The variable name *evApq* is a mnemonic for "evidence of *And p q*".)

Notice that Haskell will not accept

```
example0 evApq = andIntro (andElimL evApq) (andElimR evApq)
```

unless we change the type.

Another example:

```
example1 :: FOL.And q (FOL.Not q) → p
example1 evAqnq = notElim (notIntro (λhyp → evAqnq))
```

To sum up the *And* case we have one introduction and two elimination rules:

```
andIntro  :: p → q → And p q
andElimL :: And p q → p
andElimR :: And p q → q
```

If we see these introduction and elimination rules as an API, what would be a reasonable implementation of the datatype *And p q*? A type of pairs! Then we see that the corresponding Haskell functions would be

```
pair :: p → q → (p, q)   -- andIntro
fst  :: (p, q) → p        -- andElimL
snd  :: (p, q) → q        -- andElimR
```

Revisiting the tupling transform In exercise 1.9, the "tupling transform" was introduced, relating a pair of functions to a function returning a pair. (Please revisit that exercise if you skipped it before.) There is a logic formula corresponding to the type of the tupling transform:

$$(a \Rightarrow (b \,\&\, c)) \Leftrightarrow (a \Rightarrow b) \,\&\, (a \Rightarrow c)$$

The proof of this formula closely follows the implementation of the transform. Therefore we start with the two directions of the transform as functions:

```
test1' :: (a → (b, c)) → (a → b, a → c)
test1' = λa2bc → (λa → fst (a2bc a)
                  ,λa → snd (a2bc a))
test2' :: (a → b, a → c) → (a → (b, c))
test2' = λfg → λa → (fst fg a, snd fg a)
```

Then we move on to the corresponding logic statements with proofs. Note how the functions are "hidden inside" the proof.

```
test1 :: Impl (Impl a (And b c)) (And (Impl a b) (Impl a c))
test1 = implIntro (λa2bc →
           andIntro (implIntro (λa → andElimL (implElim a2bc a)))
                    (implIntro (λa → andElimR (implElim a2bc a))))
test2 :: Impl (And (Impl a b) (Impl a c)) (Impl a (And b c))
test2 = implIntro (λfg →
           implIntro (λa →
             andIntro
               (implElim (andElimL fg) a)
               (implElim (andElimR fg) a)))
```

Or **is the dual of** *And***.** Most of the properties of *And* have corresponding properties for *Or*. Often it is enough to simply swap the direction of the "arrows" (implications) and swap the role between introduction and elimination.

$$\begin{array}{ll} \mathit{orIntroL} & : P \to (P \mid Q) \\ \mathit{orIntroR} & : Q \to (P \mid Q) \\ \mathit{orElim} & : (P \Rightarrow R) \to (Q \Rightarrow R) \to ((P \mid Q) \Rightarrow R) \end{array}$$

Here the implementation type can be a labelled sum type, also called disjoint union and in Haskell: *Either*.

2.9 Case study: there is always another prime

As an example of combining forall, exists and implication let us turn to one statement of the fact that there are infinitely many primes. If we assume we have a unary predicate expressing that a number is prime and a binary (infix) predicate ordering the natural numbers we can define a formula *IP* for "Infinitely many Primes" as follows:

$$\mathit{IP} = \forall\ n.\ \ \mathit{Prime}\ n \Rightarrow \exists\ m.\ \ \mathit{Prime}\ m\ \&\ m > n$$

Combined with the fact that there is at least one prime (like 2) we can repeatedly refer to this statement to produce a never-ending stream of primes.

To prove this formula we first translate from logic to programs as described above. We can translate step by step, starting from the top level. The forall-quantifier translates to a (dependent) function type $(n : \mathit{Term}) \to$ and the implication to a normal function type $\mathit{Prime}\ n \to$. The exists-quantifier translates to a (dependent) pair type $((m : \mathit{Term}), ...)$ and finally the & translates into a pair type. Putting all this together we get a type signature for any *proof* of the theorem:

$$\mathit{proof} : (n : \mathit{Term}) \to \mathit{Prime}\ n \to ((m : \mathit{Term}), (\mathit{Prime}\ m, m > n))$$

Now we can start filling in the definition of *proof* as a two-argument function returning a nested pair:

```
proof n np = (m, (pm, gt))
  where m' = 1 + factorial n
        m  = {- some non-trivial prime factor of m' -}
        pm = {- a proof that m is prime -}
        gt = {- a proof that m > n -}
```

The proof *pm* is the core of the theorem. First, we note that for any $2 \leqslant p \leqslant n$ we have

$$\begin{array}{ll} m'\%p & == \{\text{- Def. of } m' \text{ -}\} \\ (1 + n!)\%p & == \{\text{- modulo distributes of } + \text{ -}\} \\ 1\%p + (n!)\%p & == \{\text{- modulo comp.: } n! \text{ has } p \text{ as a factor -}\} \\ 1 \quad\ + 0 & == \\ 1 & \end{array}$$

where $x\%y$ is the remainder after integer division of x by y. Thus m' is not divisible by any number from 2 to n. But is it a prime? If m' is prime then $m = m'$ and the proof is done (because $1 + n! \geqslant 1 + n > n$). Otherwise, let m be a prime factor of m' (thus $m' = m * q$, $q > 1$). Then $1 == m'\%p == (m\%p) * (q\%p)$ which means that neither m nor q are divisible by p (otherwise the product would be zero). Thus they must both be $>n$. QED.

Note that the proof can be used to define a somewhat useful function which takes any prime number to some larger prime number. We can compute a few example values:

$2 \mapsto \ \ 3 \quad (1{+}2!)$
$3 \mapsto \ \ 7 \quad (1{+}3!)$
$5 \mapsto 11 \quad (1{+}5! = 121 = 11{*}11)$
$7 \mapsto 71 \quad \ldots$

2.10 Existential quantification as a pair type

We mentioned before that existential quantification can be seen as as a "big *Or*" of a family of formulas $P\ a$ for all terms a. This means that to prove the quantification, we only need exhibit one witness and one proof for that member of the family.

$$\exists\text{-}\mathit{Intro} : (a : \mathit{Term}) \to P\ a \to (\exists\ x.\ \ P\ x)$$

For binary *Or* the "family" only had two members, one labelled L for *Left* and one R for *Right*, and we used one introduction rule for each. Here, for the generalisation of *Or*, we have unified the two rules into one with an added parameter a corresponding to the label which indicates the family member.

In the other direction, if we look at the binary elimination rule, we see the need for two arguments to be sure of how to prove the implication for any family member of the binary *Or*.

$$\mathit{orElim} : (P \Rightarrow R) \to (Q \Rightarrow R) \to ((P \mid Q) \Rightarrow R)$$

The generalisation unifies these two to one family of arguments. If we can prove R for each member of the family, we can be sure to prove R when we encounter some family member:

$$\exists\text{-}\mathit{Elim} : ((a : \mathit{Term}) \to P\ a \Rightarrow R) \to (\exists\ x.\ \ P\ x) \Rightarrow R$$

The datatype corresponding to $\exists\ x.\ \ P\ x$ is a pair of a witness a and a proof of $P\ a$. We sometimes write this type $(a : \mathit{Term}, P\ a)$.

2.11 Basic concepts of calculus

Now we have built up quite a bit of machinery to express logic formulas and proofs. It is time time to apply it to some concepts in calculus. We start we the concept of "limit point" which is used in the formulation of different properties of limits of functions.

Limit point *Definition* (adapted from Rudin [1964], page 28): Let X be a subset of $\mathbb{R}$. A point $p \in \mathbb{R}$ is a limit point of X iff for every $\epsilon > 0$, there exists $q \in X$ such that $q \neq p$ and $|q - p| < \epsilon$.

To express "Let X be a subset of $\mathbb{R}$" we write $X : \mathscr{P}\ \mathbb{R}$. In general, the operator $\mathscr{P}$ takes a set (here $\mathbb{R}$) to the set of all its subsets.

$$\begin{aligned}&\mathit{Limp} : \mathbb{R} \to \mathscr{P}\ \mathbb{R} \to \mathit{Prop}\\ &\mathit{Limp}\ p\ X = \forall \epsilon > 0.\ \exists\ q \in X - \{p\}.\ |q - p| < \epsilon\end{aligned}$$

Notice that q depends on ϵ. Thus by introducing a function getq we can move the $\exists$ out.

$$\begin{aligned}&\textbf{type}\ Q = \mathbb{R}_{>0} \to (X - \{p\})\\ &\mathit{Limp}\ p\ X = \exists \mathit{getq} : Q.\ \forall\ \epsilon > 0.\ |\mathit{getq}\ \epsilon - p| < \epsilon\end{aligned}$$

Next: introduce the "open ball" function B.

$$\begin{aligned}&B : \mathbb{R} \to \mathbb{R}_{>0} \to \mathscr{P}\ \mathbb{R}\\ &B\ c\ r = \{x \mid |x - c| < r\}\end{aligned}$$

$B\ c\ r$ is often called an "open ball" around c of radius r. On the real line this "open ball" is just an open interval, but with complex c or in more dimensions the term feels more natural. In every case $B\ c\ r$ is an open set of values (points) of distance less than r from c. The open balls around c are special cases of *neighbourhoods of* c which can have other shapes but must contain some open ball.

Using B we get

$$\mathit{Limp}\ p\ X = \exists \mathit{getq} : Q.\ \forall \epsilon > 0.\ \mathit{getq}\ \epsilon \in B\ p\ \epsilon$$

Example 1: Is $p = 1$ a limit point of $X = \{1\}$? No! $X - \{p\} = \{\}$ (there is no $q \neq p$ in X), thus there cannot exist a function getq because it would have to return elements in the empty set!

Example 2: Is $p = 1$ a limit point of the open interval $X = (0, 1)$? First note that $p \notin X$, but it is "very close" to X. A proof needs a function getq which from any ϵ computes a point $q = \mathit{getq}\ \epsilon$ which is in both X and $B\ 1\ \epsilon$. We need a point q which is in X and *closer* than ϵ from 1 We can try with $q = 1 - \epsilon / 2$ because $|1 - (1 - \epsilon / 2)| = |\epsilon / 2| = \epsilon / 2 < \epsilon$ which means $q \in B\ 1\ \epsilon$. We also see that $q \neq 1$ because $\epsilon > 0$. The only remaining thing to check is that $q \in X$. This is true for sufficiently small ϵ but the function getq must work for all positive reals. We can use any value in X (for example $17 / 38$) for ϵ which are "too big" ($\epsilon \geqslant 2$). Thus our function can be

$$\begin{aligned} \mathit{getq}\ \epsilon\ &|\ \epsilon < 2 &&= 1 - \epsilon / 2 \\ &|\ \mathit{otherwise} &&= 17 / 38 \end{aligned}$$

A slight variation which is often useful would be to use max to define $\mathit{getq}\ \epsilon = \mathit{max}\ (17/38, 1-\epsilon/2)$. Similarly, we can show that any internal point (like $1 / 2$) is a limit point.

Example 3: limit of an infinite discrete set X

$$X = \{1 / n \mid n \in \mathbb{N}_{>0}\}$$

Show that 0 is a limit point of X. Note (as above) that $0 \notin X$.

We want to prove $\mathit{Limp}\ 0\ X$ which is the same as $\exists \mathit{getq} : Q.\ \forall \epsilon > 0.\ \mathit{getq}\ \epsilon \in B\ 0\ \epsilon$. Thus, we need a function getq which takes any $\epsilon > 0$ to an element of $X - \{0\} = X$ which is less than ϵ away from 0. Or, equivalently, we need a function $\mathit{getn} : \mathbb{R}_{>0} \to \mathbb{N}_{>0}$ such that $1 / n < \epsilon$. Thus, we need to find an n such that $1 / \epsilon < n$. If $1 / \epsilon$ would be an integer we could use the next integer $(1 + 1 / \epsilon)$, so the only step remaining is to round up:

$$\begin{aligned} &\mathit{getq}\ \epsilon = 1 / \mathit{getn}\ \epsilon \\ &\mathit{getn}\ \epsilon = 1 + \mathit{ceiling}\ (1 / \epsilon) \end{aligned}$$

Exercise: prove that 0 is the *only* limit point of X.

Proposition: If X is finite, then it has no limit points.

$$\forall p \in \mathbb{R}.\ \neg\ (\mathit{Limp}\ p\ X)$$

This is a good exercises in quantifier negation!

$$\begin{aligned} &\neg\ (\mathit{Limp}\ p\ X) && = \{\text{- Def. of } \mathit{Limp} \text{ -}\} \\ &\neg\ (\exists \mathit{getq} : Q.\ \forall \epsilon > 0.\ \mathit{getq}\ \epsilon \in B\ p\ \epsilon) && = \{\text{- Negation of existential -}\} \\ &\forall\ \mathit{getq} : Q.\ \neg\ (\forall \epsilon > 0.\ \mathit{getq}\ \epsilon \in B\ p\ \epsilon) && = \{\text{- Negation of universal -}\} \\ &\forall\ \mathit{getq} : Q.\ \exists\ \epsilon > 0.\ \neg\ (\mathit{getq}\ \epsilon \in B\ p\ \epsilon) && = \{\text{- Simplification -}\} \\ &\forall\ \mathit{getq} : Q.\ \exists\ \epsilon > 0.\ |\mathit{getq}\ \epsilon - p| \geqslant \epsilon \end{aligned}$$

Thus, using the "functional interpretation" of this type we see that a proof needs a function *noLim*

$$noLim : (getq : Q) \to \mathbb{R}_{>0}$$

such that **let** $\epsilon = noLim\ getq$ **in** $|getq\ \epsilon - p| \geqslant \epsilon$.

Note that $noLim$ is a *higher-order* function: it takes a function $getq$ as an argument. How can we analyse this function to find a suitable ϵ? The key here is that the range of $getq$ is $X - \{p\}$ which is a finite set (not containing p). Thus we can enumerate all the possible results in a list $xs = [x_1, x_2, \ldots x_n]$, and measure their distances to p: $ds = map\ (\lambda x \to |x - p|)\ xs$. Now, if we let $\epsilon = minimum\ ds$ we can be certain that $|getq\ \epsilon - p| \geqslant \epsilon$ just as required (and $\epsilon \neq 0$ because $p \notin xs$).

Exercise: If $Limp\ p\ X$ we now know that X is infinite. Show how to construct an infinite sequence $a : \mathbb{N} \to \mathbb{R}$ of points in $X - \{p\}$ which gets arbitrarily close to p. Note that this construction can be seen as a proof of $Limp\ p\ X \Rightarrow Infinite\ X$.

2.12 The limit of a sequence

Now we can move from limit points to the more familiar limit of a sequence. At the core of DSLsofMath is the ability to analyse definitions from mathematical texts, and here we will use the definition of the limit of a sequence from Adams and Essex [2010, page 498]:

> We say that sequence a_n converges to the limit L, and we write $\lim_{n\to\infty} a_n = L$, if for every positive real number ϵ there exists an integer N (which may depend on ϵ) such that if $n > N$, then $a\_n - L < \varepsilon$.

The first step is to type the variables introduced. A sequence a is a function from $\mathbb{N}$ to $\mathbb{R}$, thus $a : \mathbb{N} \to \mathbb{R}$ where a_n is special syntax for normal function application of a to $n : \mathbb{N}$. Then we have $L : \mathbb{R}$, $\epsilon : \mathbb{R}_{>0}$, and $N : \mathbb{N}$ (or $N : \mathbb{R}_{>0} \to \mathbb{N}$).

In the next step we analyse the new concept introduced: the syntactic form $\lim_{n\to\infty} a_n = L$ which we could express as an infix binary predicate $haslim$ where $a\ haslim\ L$ is well-typed if $a : \mathbb{N} \to \mathbb{R}$ and $L : \mathbb{R}$.

The third step is to formalise the definition using logic: we define $haslim$ using a ternary helper predicate P:

$$\begin{aligned}
&a\ haslim\ L = \forall \epsilon > 0.\ P\ a\ L\ \epsilon \quad \text{-- “for every positive real number } \epsilon \ldots\text{”}\\
&P\ a\ L\ \epsilon = \exists N : \mathbb{N}.\ \forall n \geqslant N.\ |a_n - L| < \epsilon\\
&\qquad\quad = \exists N : \mathbb{N}.\ \forall n \geqslant N.\ a_n \in B\ L\ \epsilon\\
&\qquad\quad = \exists N : \mathbb{N}.\ I\ a\ N \subseteq B\ L\ \epsilon
\end{aligned}$$

where we have introduced an “image function” for sequences “from N onward”:

$$\begin{aligned}
&I : (\mathbb{N} \to X) \to \mathbb{N} \to \mathcal{P}\ X\\
&I\ a\ N = \{a\ n \mid n \geqslant N\}
\end{aligned}$$

The “forall-exists”-pattern is very common and it is often useful to transform such formulas into another form. In general $\forall\ x : X.\ \ \exists\ y : Y.\ \ Q\ x\ y$ is equivalent to $\exists\ gety : X \to Y.\ \ \forall\ x : X.\ \ Q\ x\ (gety\ x)$. In the new form we more clearly see the function $gety$ which shows how the choice of y depends on x. For our case with $haslim$ we can thus write

$$a\ haslim\ L = \ \exists\ getN : \mathbb{R}_{>0} \to \mathbb{N}.\ \forall \epsilon > 0.\ I\ a\ (getN\ \epsilon) \subseteq B\ L\ \epsilon$$

where we have made the function $getN$ more visible. The core evidence of $a\ haslim\ L$ is the existence of such a function (with suitable properties).

Exercise: Prove that the limit of a sequence is unique.

Exercise: prove that $(a_1\ \mathit{haslim}\ L_1)\ \&\ (a_2\ \mathit{haslim}\ L_2)$ implies $(a_1 + a_2)\ \mathit{haslim}\ (L_1 + L_2)$.

When we are not interested in the exact limit, just that it exists, we say that a sequence a is *convergent* when $\exists L.\ a\ \mathit{haslim}\ L$.

2.13 Case study: The limit of a function

As our next mathematical text book quote we take the definition of the limit of a function of type $\mathbb{R} \to \mathbb{R}$ from Adams and Essex [2010]:

> **A formal definition of limit**
>
> We say that $f(x)$ **approaches the limit** L as x **approaches** a, and we write
>
> $$\lim_{x \to a} f(x) = L,$$
>
> if the following condition is satisfied:
> for every number $\epsilon > 0$ there exists a number $\delta > 0$, possibly depending on ϵ, such that if $0 < |x - a| < \delta$, then x belongs to the domain of f and
>
> $$|f\ (x) - L| < \epsilon.$$

The *lim* notation has four components: a variable name x, a point a an expression $f(x)$ and the limit L. The variable name + the expression can be combined into just the function f and this leaves us with three essential components: f, a, and L. Thus, *lim* can be seen as a ternary (3-argument) predicate which is satisfied if the limit of f exists at a and equals L. If we apply our logic toolbox we can define *lim* starting something like this:

$$\mathit{lim}\ f\ a\ L = \forall\ \epsilon > 0.\ \exists\ \delta > 0.\ P\ \epsilon\ \delta$$

It is often useful to introduce a local name (like P here) to help break the definition down into more manageable parts. If we now naively translate the last part we get this "definition" for P:

$$\textbf{where}\ P\ \epsilon\ \delta = (0 < |x - a| < \delta) \Rightarrow (x \in \mathit{Dom}\ f \wedge |f\ x - L| < \epsilon))$$

Note that there is a scoping problem: we have f, a, and L from the "call" to *lim* and we have ϵ and δ from the two quantifiers, but where did x come from? It turns out that the formulation "if ... then ..." hides a quantifier that binds x. Thus we get this definition:

$$\begin{aligned}&\mathit{lim}\ a\ f\ L = \forall\ \epsilon > 0.\ \exists\ \delta > 0.\ \forall\ x.\ \ P\ \epsilon\ \delta\ x\\ &\quad \textbf{where}\ P\ \epsilon\ \delta\ x = (0 < |x - a| < \delta) \Rightarrow (x \in \mathit{Dom}\ f \wedge |f\ x - L| < \epsilon))\end{aligned}$$

The predicate *lim* can be shown to be a partial function of two arguments, f and a. This means that each function f can have *at most* one limit L at a point a. (This is not evident from the definition and proving it is a good exercise.)

2.14 Recap of syntax tres with variables, *Env* and *lookup*

This was frequently a source of confusion already the first week so there is already a question + answers earlier in this text. But here is an additional example to help clarify the matter.

```
data Rat v = RV v | FromI Integer | RPlus (Rat v) (Rat v) | RDiv (Rat v) (Rat v)
  deriving (Eq, Show)
```

```
newtype RatSem = RSem (Integer, Integer)
```

We have a type *Rat v* for the syntax trees of rational number expressions (with variables of type *v*) and a type *RatSem* for the semantics of those rational number expressions as pairs of integers. The constructor $RV :: v \to Rat\ v$ is used to embed variables with names of type *v* in *Rat v*. We could use *String* instead of *v* but with a type parameter *v* we get more flexibility at the same time as we get better feedback from the type checker. Note that this type parameter serves a different purpose from the type parameter in 1.6.

To evaluate some $e :: Rat\ v$ we need to know how to evaluate the variables we encounter. What does "evaluate" mean for a variable? Well, it just means that we must be able to translate a variable name (of type *v*) to a semantic value (a rational number in this case). To "translate a name to a value" we can use a function (of type $v \to RatSem$) so we can give the following implementation of the evaluator:

```
evalRat1 :: (v → RatSem) → (Rat v → RatSem)
evalRat1 ev (RV v)       = ev v
evalRat1 ev (FromI i)    = fromISem i
evalRat1 ev (RPlus l r)  = plusSem (evalRat1 ev l) (evalRat1 ev r)
evalRat1 ev (RDiv l r)   = divSem  (evalRat1 ev l) (evalRat1 ev r)
```

Notice that we simply added a parameter *ev* for "evaluate variable" to the evaluator. The rest of the definition follows a common pattern: recursively translate each subexpression and apply the corresponding semantic operation to combine the results: *RPlus* is replaced by *plusSem*, etc.

```
fromISem :: Integer → RatSem
fromISem i = RSem (i, 1)

plusSem :: RatSem → RatSem → RatSem
plusSem = undefined  -- TODO: exercise

  -- Division of rational numbers
divSem :: RatSem → RatSem → RatSem
divSem (RSem (a, b)) (RSem (c, d)) = RSem (a * d, b * c)
```

Often the first argument *ev* to the eval function is constructed from a list of pairs:

```
type Env v s = [(v, s)]
envToFun :: (Show v, Eq v) ⇒ Env v s → (v → s)
envToFun [] v = error ("envToFun: variable " ++ show v ++ " not found")
envToFun ((w, s) : env) v
  | w == v      = s
  | otherwise = envToFun env v
```

Thus, *Env v s* can be seen as an implementation of a "lookup table". It could also be implemented using hash tables or binary search trees, but efficiency is not the point here. Finally, with *envToFun* in our hands we can implement a second version of the evaluator:

```
evalRat2 :: (Show v, Eq v) ⇒ (Env v RatSem) → (Rat v → RatSem)
evalRat2 env e = evalRat1 (envToFun env) e
```

SET and PRED Several groups have had trouble grasping the difference between *SET* and *PRED*. This is understandable, because we have so far in the lectures mostly talked about term syntax + semantics, and not so much about predicate syntax and semantics. The one example of terms + predicates covered in the lectures is Predicate Logic and I never actually showed how eval (for the expressions) and check (for the predicates) is implemented.

As an example we can we take our terms to be the rational number expressions defined above and define a type of predicates over those terms:

```
type Term v = Rat v
data RPred v = Equal    (Term v) (Term v)
             | LessThan (Term v) (Term v)
             | Positive (Term v)
             | AND (RPred v) (RPred v)
             | NOT (RPred v)
  deriving (Eq, Show)
```

Note that the first three constructors, *Eq*, *LessThan*, and *Positive*, describe predicates or relations between terms (which can contain term variables) while the two last constructors, *AND* and *NOT*, just combine such relations together. (Terminology: I often mix the words "predicate" and "relation".)

We have already defined the evaluator for the *Term v* type but we need to add a corresponding "evaluator" (called *check*) for the *RPred v* type. Given values for all term variables the predicate checker should just determine if the predicate is true or false.

```
checkRP :: (Eq v, Show v) ⇒ Env v RatSem → RPred v → Bool
checkRP env (Equal    t1 t2) = eqSem       (evalRat2 env t1) (evalRat2 env t2)
checkRP env (LessThan t1 t2) = lessThanSem (evalRat2 env t1) (evalRat2 env t2)
checkRP env (Positive t1)    = positiveSem (evalRat2 env t1)
checkRP env (AND p q) = (checkRP env p) ∧ (checkRP env q)
checkRP env (NOT p)   = ¬ (checkRP env p)
```

Given this recursive definition of *checkRP*, the semantic functions *eqSem*, *lessThanSem*, and *positiveSem* can be defined by just working with the rational number representation:

```
eqSem       :: RatSem → RatSem → Bool
lessThanSem :: RatSem → RatSem → Bool
positiveSem :: RatSem → Bool
eqSem       = error "TODO"
lessThanSem = error "TODO"
positiveSem = error "TODO"
```

2.15 More general code for first order languages

This subsection contains some extra material which is not a required part of the course.

It is possible to make one generic implementation of *FOL* which can be specialised to any first order language.

- *Term* = Syntactic terms
- n = names (of atomic terms)
- f = function names
- v = variable names
- *WFF* = Well Formed Formulas
- p = predicate names

```
data Term n f v = N n | F f [Term n f v] | V v
  deriving Show
data WFF n f v p =
    P p    [Term n f v]
  | Equal (Term n f v)   (Term n f v)
  | And   (WFF n f v p) (WFF n f v p)
  | Or    (WFF n f v p) (WFF n f v p)
  | Equiv (WFF n f v p) (WFF n f v p)
  | Impl  (WFF n f v p) (WFF n f v p)
  | Not   (WFF n f v p)
  | FORALL v (WFF n f v p)
  | EXISTS  v (WFF n f v p)
  deriving Show
```

2.16 Exercises: abstract FOL

```
{-# LANGUAGE EmptyCase #-}
import AbstractFOL
```

2.16.1 Exercises

Short technical note For these first exercises on the propositional fragment of FOL (introduced in section 2.8), you might find it useful to take a look at typed holes, a feature which is enabled by default in GHC and available (the same way as the language extension `EmptyCase` above) from version 7.8.1 onwards: `https://wiki.haskell.org/GHC/Typed_holes`.

If you are familiar with Agda, these will be familiar to use. In summary, when trying to code up the definition of some expression (which you have already typed) you can get GHC's type checker to help you out a little in seeing how far you might be from forming the expression you want. That is, how far you are from constructing something of the appropriate type.

Take *example0* below, and say you are writing:

```
example0 e = andIntro (_ e) _
```

When loading the module, GHC will tell you which types your holes (marked by "_") should have for the expression to be type correct.

On to the exercises.

Exercise 2.1. Prove these three theorems (for arbitrary p and q):

```
Impl (And p q) q
Or p q → Or q p
Or p (Not p)
```

Exercise 2.2. Translate to Haskell and prove the De Morgan laws:

$$\neg (p \vee q) \longleftrightarrow \neg p \wedge \neg q$$
$$\neg (p \wedge q) \longleftrightarrow \neg p \vee \neg q$$

(translate equivalence to conjunction of two implications).

Exercise 2.3. So far, the implementation of the datatypes has played no role. To make this clearer: define the types for connectives in *AbstractFol* in any way you wish, e.g.:

$$\begin{aligned} &\mathit{And}\ p\ q = A\ () \\ &\mathit{Not}\ p \quad\ = B\ p \end{aligned}$$

etc. as long as you still export only the data types, and not the constructors. Convince yourself that the proofs given above still work and that the type checker can indeed be used as a poor man's proof checker.

Exercise 2.4. The introduction and elimination rules suggest that some implementations of the datatypes for connectives might be more reasonable than others. We have seen that the type of evidence for $p \rightarrow q$ is very similar to the type of functions $p \rightarrow q$, so it would make sense to define

> **type** *Impl* $p\ q = (p \rightarrow q)$

Similarly, $\wedge{-}\mathit{ElimL}$ and $\wedge{-}\mathit{ElimR}$ behave like the functions *fst* and *snd* on pairs, so we can take

> **type** *And* $p\ q = (p, q)$

while the notion of proof by cases is very similar to that of writing functions by pattern-matching on the various clauses, making $p \vee q$ similar to *Either*:

> **type** *Or* $p\ q$ = *Either* $p\ q$

1. Define and implement the corresponding introduction and implementation rules as functions.
2. Compare proving the distributivity laws

$$\begin{aligned} &(p \wedge q) \vee r \longleftrightarrow (p \vee r) \wedge (q \vee r) \\ &(p \vee q) \wedge r \longleftrightarrow (p \wedge r) \vee (q \wedge r) \end{aligned}$$

using the "undefined" introduction and elimination rules, with writing the corresponding functions with the given implementations of the datatypes. The first law, for example, requires a pair of functions:

$$\begin{aligned} &(\mathit{Either}\ (p, q)\ r \rightarrow (\mathit{Either}\ p\ r, \mathit{Either}\ q\ r) \\ &, (\mathit{Either}\ p\ r, \mathit{Either}\ q\ r) \rightarrow \mathit{Either}\ (p, q)\ r \\ &) \end{aligned}$$

Moral: The natural question is: is it true that every time we find an implementation using the "pairs, $\rightarrow$, *Either*" translation of sentences, we can also find one using the "undefined" introduction and elimination rules? The answer, perhaps surprisingly, is *yes*, as long as the functions we write are total. This result is known as *the Curry-Howard isomorphism.*

Exercise 2.5. Can we extend the Curry-Howard isomorphism to formulas with $\neg$? In other words, is there a type that we could use to define *Not* p, which would work together with pairs, $\rightarrow$, and *Either* to give a full translation of sentential logic?

Unfortunately, we cannot. The best that can be done is to define an empty type

> **data** *Empty*

and define *Not* as

```
type Not p = p → Empty
```

The reason for this definition is: when p is *Empty*, the type *Not p* is not empty: it contains the identity

```
idEmpty :: Empty → Empty
isEmpty evE = evE
```

When p is not *Empty* (and therefore is true), there is no (total, defined) function of type $p \to$ *Empty*, and therefore *Not p* is false.

Moreover, mathematically, an empty set acts as a contradiction: there is exactly one function from the empty set to any other set, namely the empty function. Thus, if we had an element of the empty set, we could obtain an element of any other set.

Now to the exercise:

Implement *notIntro* using the definition of *Not* above, i.e., find a function

```
notIntro :: (p → (q, q → Empty)) → (p → Empty)
```

Using

```
contraHey :: Empty → p
contraHey evE = case evE of { }
```

prove

$$q \wedge \neg q \to p$$

You will, however, not be able to prove $p \vee \neg p$ (try it!).

Prove

$$\neg\, p \vee \neg q \to \neg(p \wedge q)$$

but you will not be able to prove the converse.

Exercise 2.6. The implementation *Not* $p = p \to$ *Empty* is not adequate for representing all the closed formulas in FOL, but it is adequate for *constructive logic* (also known as *intuitionistic*). In constructive logic, the $\neg p$ is *defined* as $p \to \bot$, and the following elimination rule is given for $\bot$: $\bot{-}Elim : \bot \to a$, corresponding to the principle that everything follows from a contradiction ("if you believe $\bot$, you believe everything").

Every sentence provable in constructive logic is provable in classical logic, but the converse, as we have seen in the previous exercise, does not hold. On the other hand, there is no sentence in classical logic which would be contradicted in constructive logic. In particular, while we cannot prove $p \vee \neg p$, we *can* prove (constructively!) that there is no p for which $\neg(p \vee \neg p)$, i.e., that the sentence $\neg\,\neg\,(p \vee \neg p)$ is always true.

Show this by implementing the following function:

```
noContra :: (Either p (p → Empty) → Empty) → Empty
```

Hint: The key is to use the function argument to *noContra* twice.

Exercise 2.7. *From exam 2016-08-23*

Consider the classical definition of continuity:

> *Definition:* Let $X \subseteq \mathbb{R}$, and $c \in X$. A function $f : X \to \mathbb{R}$ is *continuous at c* if for every $\epsilon > 0$, there exists $\delta > 0$ such that, for every x in the domain of f, if $|x - c| < \delta$, then $|fx - fc| < \epsilon$.

1. Write the definition formally, using logical connectives and quantifiers.
2. Introduce functions and types to simplify the definition.
3. Prove the following proposition: If `f` and `g` are continuous at `c`, `f + g` is continuous at `c`.

Exercise 2.8. *From exam 2017-08-22*

Adequate notation for mathematical concepts and proofs (or "50 shades of continuity").

A formal definition of "$f : X \to \mathbb{R}$ is continuous" and "f is continuous at c" can be written as follows (using the helper predicate Q):

$$\begin{aligned}
C\ (f) &= \forall\, c : X.\ \mathit{Cat}\ (f, c) \\
\mathit{Cat}\ (f, c) &= \forall\, \epsilon > 0.\ \exists\, \delta > 0.\ Q\ (f, c, \epsilon, \delta) \\
Q\ (f, c, \epsilon, \delta) &= \forall\, x : X.\ |x - c| < \delta \Rightarrow |f\ x - f\ c| < \epsilon
\end{aligned}$$

By moving the existential quantifier outwards we can introduce the function $\mathit{get}\delta$ which computes the required δ from c and ϵ:

$$C'\ (f) = \exists\, \mathit{get}\delta : X \to \mathbb{R}_{>0} \to \mathbb{R}_{>0}.\ \forall\, c : X.\ \forall\, \epsilon > 0.\ Q\ (f, c, \epsilon, \mathit{get}\delta\ c\ \epsilon)$$

Now, consider this definition of *uniform continuity*:

> **Definition:** Let $X \subseteq \mathbb{R}$. A function $f : X \to \mathbb{R}$ is *uniformly continuous* if for every $\epsilon > 0$, there exists $\delta > 0$ such that, for every x and y in the domain of f, if $|x - y| < \delta$, then $|f\ x - f\ y| < \epsilon$.

1. Write the definition of $\mathit{UC}\ (f) =$ "f is uniformly continuous" formally, using logical connectives and quantifiers. Try to use Q.
2. Transform $\mathit{UC}\ (f)$ into a new definition $\mathit{UC}'\ (f)$ by a transformation similar to the one from $C\ (f)$ to $C'\ (f)$. Explain the new function $\mathit{new}\delta$ introduced.
3. Prove that $\forall f : X \to \mathbb{R}.\ \mathit{UC}'\ (f) \Rightarrow C'\ (f)$. Explain your reasoning in terms of $\mathit{get}\delta$ and $\mathit{new}\delta$.

2.16.2 More exercises

Preliminary remarks

- when asked to "sketch an implementation" of a function, you must explain how the various results might be obtained from the arguments, in particular, why the evidence required as output may result from the evidence given as input. You may use all the facts you know (for instance, that addition is monotonic) without formalisation.

- to keep things short, let us abbreviate a significant chunk of the definition of *a haslim L* (see section 2.12) by

$$\begin{array}{l} P : \mathit{Seq}\ X \to X \to \mathbb{R}_{>0} \to \mathit{Prop} \\ P\ a\ L\ \epsilon = \exists\ N : \mathbb{N}.\ \ \forall\ n : \mathbb{N}.\ \ (n \geqslant N) \to (|a_n - L| < \epsilon) \end{array}$$

Exercise 2.9. Consider the statement:

The sequence $\{a_n\} = (0, 1, 0, 1, ...)$ does not converge.

a. Define the sequence $\{a_n\}$ as a function $a : \mathbb{N} \to \mathbb{R}$.

b. The statement "the sequence $\{a_n\}$ is convergent" is formalised as

$$\exists\ L : \mathbb{R}.\ \ \forall\ \epsilon > 0.\ \ P\ a\ L\ \epsilon$$

The formalisation of "the sequence $\{a_n\}$ is not convergent" is therefore

$$\neg \exists\ L : \mathbb{R}.\ \ \forall\ \epsilon > 0.\ \ P\ a\ L\ \epsilon$$

Simplify this expression using the rules

$$\begin{array}{l} \neg\ (\exists\ x.\ \ P\ x) \longleftrightarrow (\forall\ x.\ \ \neg\ (P\ x)) \\ \neg\ (\forall\ x.\ \ P\ x) \longleftrightarrow (\exists\ x.\ \ \neg\ (P\ x)) \\ \neg\ (P \to Q) \longleftrightarrow P \wedge \neg Q \end{array}$$

The resulting formula should have no $\neg$ in it (that's possible because the negation of $<$ is $\geqslant$).

c. Give a functional interpretation of the resulting formula.

d. Sketch an implementation of the function, considering two cases: $L \neq 0$ and $L = 0$.

Exercise 2.10. Consider the statement:

The limit of a convergent sequence is unique.

a. There are many ways of formalising this in FOL. For example:

$$\begin{array}{l} \textbf{let}\ Q\ a\ L = \forall\ \epsilon > 0.\ \ P\ a\ L\ \epsilon \\ \textbf{in}\ \forall\ L_1 : \mathbb{R}.\ \ \forall\ L_2 : \mathbb{R}.\ \ (Q\ a\ L_1 \wedge Q\ a\ L_2) \to L_1 = L_2 \end{array}$$

i.e., if the sequence converges to two limits, then they must be equal, or

$$\forall\ L_1 : \mathbb{R}.\ \ \forall\ L_2 : \mathbb{R}.\ \ Q\ a\ L_1 \wedge L_1 \neq L_2 \to \neg\ Q\ a\ L_2$$

i.e., if a sequence converges to a limit, then it doesn't converge to anything that isn't the limit.

Simplify the latter alternative to eliminate the negation and give functional representations of both.

b. Choose one of the functions and sketch an implementation of it.