# Lab 15

Functional Programming (ITI0212)

2022-05-06

## Logical "and" and "or"

During the lecture, we defined inductive types `And p q` and `Or p q`. A term of `And p q` consists precisely of a proof of `p` and of a proof of `q`. A term of `Or p q` is either a proof of `p` or a proof of `q`. We also noted that these types are *isomorphic* to the types `Pair` and `Either`, respectively.

This indicates a shift in perspective: in the *propositions-as-types* interpretation, a term `(x : Pair a b)` can be seen as either a pair of terms, or as a proof of a conjuction "`a` and `b`". Similarly, whether `(x : Either a b)` is a term that is either an `a` or a `b`, or a proof of a disjunction "`a` or `b`" depends on your perspective.

**Task 1**
Write functions `andToPair`, `pairToAnd`, `orToEither` and `eitherToOr` that convert between those types.

**Task 2** (Commutativity of `And`)
Prove that the logical "and" is commutative: If `p` and `q` holds, then `q` and `p` holds. Prove this by giving a definition for

```
commAnd : And p q -> And q p
```

and compare it to the function `swap`, defined in Lab 3, Task 2.

If you feel comfortable giving proofs of equality, you can prove that these functions are inverses of each other. If not, feel free to skip the next task.

**Task 3** (optional)
Giving definitions

```
invEitherOr :
  (x : Either a b) -> orToEither (eitherToOr x) = x

invOrEither :
  (x : Or p q) -> eitherToOr (orToEither x) = x
```

and similarly for `andToPair` and `pairToAnd`.

## Using negation as an assumption

During the lecture, we proved that the successor of an even number is odd. Let us attempt a (guided proof) of a similar proposition: *The successor of an odd number is even.* This proof will be different, since we are given a negation ("`n` is odd" = "`n` is not even") as an assumption.

**Task 4**
Copy the following skeleton of the proof and fill in the necessary holes:

```
isEvenOddSucc : (n : Nat) -> IsOdd n -> IsEven (S n)
isEvenOddSucc 0 is_odd_0 = ?hole_0
isEvenOddSucc 1 is_odd_1 = ?hole_1
isEvenOddSucc (S (S k)) is_odd_ssk = goal where
  ind_hyp : IsEven (S k)
  ind_hyp = isEvenOddSucc k $ ?hole_is_odd_k

  goal : IsEven (S (S (S k)))
  goal = IsEvenSS ind_hyp
```

Notice that this proof has two base cases, `n = 0` and `n = 1`. Proceed as follows:

1. In `?hole_0`, inspect the assumption `is_odd_0`. Use `contradiction` to prove this case.
2. In `?hole_1`, inspect the goal. It says that 2 is even, which we can easily prove directly.
3. The inductive step is different than usual, since we cannot case-split the assumption `is_odd_ssk : IsOdd (S (S k))`: its type is that of a function, and function types are *not* inductively defined!

   Nonetheless, can obtain our induction hypothesis by recursively applying `isEvenOddSucc` to the subterm `k` of `S (S k)`. Inspect the hole `?hole_is_odd_k`; the goal is to prove that `k` is odd. We assume that `k + 2` is odd, where `is_odd_ssk` is of type `IsEven (S (S k))-> Void`. Can you *compose* this assumption with another function to fill the hole?

# Playing with negation

**Task 5** (Double-negation introduction)
Prove the *principle of double-negation introduction*: Given any proposition $p$, if $p$ holds (=is provable), then its *double-negation* $\neg(\neg p)$ also holds. The following is the type signature of this statement:

```
dni : {p : Type} -> p -> Not (Not p)
```

Proceed as follows:

1. Add a single clause that contains as many arguments (to the left of `=`) as possible.
2. Inspect the types of those arguments.
3. Inspect the type of the goal.
4. Combine the arguments to produce a value of the goal type.

Using your editor integration, Idris can do step 1 for you. The integration might even be able to write the entire proof for you![1]

*Hint:* If you are confused about the nested `Not`s, remember that `Not a` is defined to be the type of functions `a -> Void`. Unfolding this definition, the problems becomes to define a function

```
dni' : p -> (p -> Void) -> Void
```

[1]Using either the helper *expression search* or *generate definition*.

**Task 6** (An impossible task)
Task 5 suggest that the logical converse is also provable: If $\neg(\neg p)$ holds, then $p$ holds, too. This is called the *principle of double-negation elimination*. Surprisingly, this statement is *unprovable* in Idris! You should try it yourself; play around with the following definition:

```
dne : {p : Type} -> Not (Not p) -> p
dne = ?dne_prf
```

You *will* get stuck. This is because Idris is a model of so-called Intuitionistic logic, where double-negation elimination is not derivable for arbitrary propositions.

**Task 7** (A (surprisingly) doable task)
Although double-negation elimination is not provable for arbitrary types `p : Type`, there are some specific types for which it is. An important example of such a case is when `p` is itself a negation `Not q`. This is called the *principle of triple-negation elimination*:

```
tne : {q : Type} -> Not (Not (Not q)) -> Not q
```

To prove this proposition, try again to introduce as many arguments as possible, then inspect the context.

*Hint:* This proof is a bit trickier, but you should obtain two arguments

```
x : q
f : ((q -> Void) -> Void) -> Void
```

Your proof should look like `tne f x = f ?prf`. Inspect the hole and figure out how to use `x` in there.