Types for Programs and Proofs 2017/2018
Exercise Session 1

September 4, 2017

Homework 1 is now available. It is due on *Monday 18 September at 13.15*. Please submit it in Fire. You should *be prepared* to present your solutions in the exercise session the same day at 15.15 - 17.00.

1 Exercises

1. Define the operation not by gradual refinement and pattern matching.

```
not : Bool → Bool
not p = {!!}
```

2. Define logical or _||_, logical implication _=>_, and logical equivalence _<=>_ by gradual refinement and pattern matching.

```
- Homework 1, Problem 1 a)
_||_ : Bool → Bool → Bool
p || q = {!!}

_=>_ : Bool → Bool → Bool
p => q = {!!}

- Homework 1, Problem 1 b)
_<=>_ : Bool → Bool → Bool
p <=> q = {!!}
```

3. Define cut-off subtraction _-_.

```
_-_ : Nat → Nat → Nat
n - m = {!!}
```

4. Define the less than function _<_.

```
_<_ : Nat → Nat → Bool
n < m = {!!}
```

5. Define a version of the conditional ifn_then_else_ which returns natural numbers.

```
ifn_then_else_ : Bool → Nat → Nat → Nat
ifn b then n else m = {!!}
```

6. Define the power function power.

```
-- Homework 1, Problem 2 b)
power : Nat → Nat → Nat
power n m = {!!}
```

7. Define the factorial function factorial. Compute the result of factorial for some arguments by normalizing. How large factorials can Agda compute?

```
-- Homework 1, Problem 2 a)
factorial : Nat → Nat
factorial n = {!!}
```

8. Define the Ackermann function[3, 4].

```
ack : Nat → Nat → Nat
ack n = {!!}
```

9. Define binary numbers in Agda as a data type Bin with four constuctors bO, bI, _O, _I: For 0 and 1 standing alone and for adding 0 and 1 at the end of a binary number. Define translation functions bin2nat and nat2bin between the binary and the unary numbers. Define addition of binary numbers _+Bin_.

```
data Bin : Set where
  -- ...

bin2nat : Bin → Nat
bin2nat x = {!!}

nat2bin : Nat → Bin
nat2bin n = {!!}

_+Bin_ : Bin → Bin → Bin
x +Bin y = {!!}
```

10. How would you define integers as a data type Int in Agda? Then define addition of integers _+Int_.

```
data Int : Set where
  -- ...

_+Int_ : Int → Int → Int
n +Int m = {!!}
```

11. Define a modified type of natural numbers Nat∗ with a special error value error. Then try to redefine division _÷∗_ by repeated subtraction so that

    $m \div* \text{zero} = \text{error}$

    and note that the termination checker will not accept this definition.

```
data Nat∗ : Set where
  -- ...

_÷∗_ : Nat → Nat → Nat∗
x ÷∗ y = {!!}
```

12. Define equality of natural numbers _==_ using only official primitive recursion natrec. Hint: Use cut-off subtraction _-_.

```
natrec : Nat → (Nat → Nat) → Nat → Nat
natrec c d zero = c
natrec c d (succ n) = d (natrec c d n)

_==_ : Nat → Nat → Nat
n == m = {!!}
```

13. Pick a theorem from [2, Chapter 2] and prove it.

```
notnot-elim : ∀ (b : Bool) → not (not b) ≡ b
notnot-elim b = {!!}

&&-idem : ∀ (p : Bool) → p && p ≡ p
&&-idem p = {!!}

||-idem : ∀ (p : Bool) → p || p ≡ p
||-idem p = {!!}

||≡ff₁ : ∀ (p q : Bool) → p || q ≡ false → false ≡ p
||≡ff₁ = {!!}

||≡ff₂ : ∀ (p q : Bool) → p || q ≡ false → q ≡ false
||≡ff₂ = {!!}

||-tt : ∀ (p : Bool) → p || true ≡ true
||-tt = {!!}

||-cong₁ : ∀ (p p' q : Bool) → p ≡ p' → p || q ≡ p' || q
||-cong₁ = {!!}

||-cong₂ : ∀ (p q q' : Bool) → q ≡ q' → p || q ≡ p || q'
||-cong₂ = {!!}

ite-same : ∀ {A : Set} (b : Bool) (x : A) → if b then x else x ≡ x
ite-same = {!!}

ite-arg : ∀ {A B : Set} (f : A → B) (b : Bool) (x y : A) →
  f (if b then x else y) ≡ if b then f x else f y
ite-arg = {!!}

contra : false ≡ true → ∀ {P : Set} → P
contra = {!!}

||-split : ∀ (p q : Bool) → p || q ≡ true → p ≡ true ⊎ q ≡ true
||-split = {!!}

canon : ∀ (p : Bool) → p ≡ true ⊎ p ≡ false
canon = {!!}

&&-snd : ∀ (p q : Bool) → p && q ≡ true → q ≡ true
```

```
&&-snd = {!!}

&&-fst : ∀ (p q : Bool) → p && q ≡ true → p ≡ true
&&-fst = {!!}

&&-combo : ∀ (p q : Bool) → p ≡ true → q ≡ true → p && q ≡ true
&&-combo = {!!}

&&-false : ∀ (p : Bool) → p && false ≡ false
&&-false = {!!}

false-imp : ∀ (b : Bool) → false => b ≡ true
false-imp = {!!}

=>-true : ∀ (b : Bool) → b => true ≡ true
=>-true = {!!}

=>-false : ∀ (b : Bool) → b => false ≡ not b
=>-false = {!!}

true-=> : ∀ (b : Bool) → true => b ≡ b
true-=> = {!!}

&&-true : ∀ (b : Bool) → b && true ≡ b
&&-true = {!!}

||-false : ∀ (b : Bool) → b || false ≡ b
||-false = {!!}

&&-contra : ∀ (b : Bool) → b && not b ≡ false
&&-contra = {!!}

&&-comm : ∀ (p q : Bool) → p && q ≡ q && p
&&-comm = {!!}

||-comm : ∀ (p q : Bool) → p || q ≡ q || p
||-comm = {!!}

&&-assoc : ∀ (p q b3 : Bool) → p && (q && b3) ≡ (p && q) && b3
&&-assoc = {!!}

||-assoc : ∀ (p q b3 : Bool) → p || (q || b3) ≡ (p || q) || b3
||-assoc = {!!}

not-over-&& : ∀ (p q : Bool) → not ( p && q ) ≡ (not p || not q)
not-over-&& = {!!}

not-over-|| : ∀ (p q : Bool) → not ( p || q ) ≡ (not p && not q)
not-over-|| = {!!}

&&-over-||-l : ∀ (p q r : Bool) → p && (q || r) ≡ (p && q) || (p && r)
&&-over-||-l = {!!}
```

```
&&-over-||-r : ∀ (p q r : Bool) → (p || q) && r ≡ (p && r) || (q && r)
&&-over-||-r = {!!}

||-over-&&-l : ∀ (p q r : Bool) → p || (q && r) ≡ (p || q) && (p || r)
||-over-&&-l = {!!}

||-over-&&-r : ∀ (p q r : Bool) → (p && q) || r ≡ (p || r) && (q || r)
||-over-&&-r = {!!}

&&-cong₁ : ∀ (p p' q : Bool) → p ≡ p' → p && q ≡ p' && q
&&-cong₁ = {!!}

&&-cong₂ : ∀ (p q q' : Bool) → q ≡ q' → p && q ≡ p && q'
&&-cong₂ = {!!}

&&-intro : ∀ (p q : Bool) → p ≡ true → q ≡ true → p && q ≡ true
&&-intro = {!!}

||-intro₁ : ∀ (p q : Bool) → p ≡ true → p || q ≡ true
||-intro₁ = {!!}

&&-elim : ∀ (p q : Bool) → p && q ≡ true → p ≡ true ∧ q ≡ true
&&-elim = {!!}

&&-elim₁ : ∀ (p q : Bool) → p && q ≡ true → p ≡ true
&&-elim₁ = {!!}

&&-elim₂ : ∀ (p q : Bool) → p && q ≡ true → q ≡ true
&&-elim₂ = {!!}

||-elim : ∀ (p q : Bool) → p || q ≡ true → p ≡ true ∨ q ≡ true
||-elim = {!!}

not-cong : ∀ (p q : Bool) → p ≡ q → not p ≡ not q
not-cong = {!!}

ite-cong₁ : ∀ {A : Set} (b b' : Bool) {x y : A} → b ≡ b' → (if b then x else y) ≡ (if b' then x else y)
ite-cong₁ = {!!}

ite-cong₂ : ∀ {A : Set} (b : Bool) {x x' y : A} → x ≡ x' → (if b then x else y) ≡ (if b then x' else y)
ite-cong₂ = {!!}

ite-cong₃ : ∀ {A : Set} (b : Bool) (x y y' : A) → y ≡ y' → if b then x else y ≡ if b then x else y'
ite-cong₃ = {!!}

&&-split : ∀ (p q : Bool) → p && q ≡ false → p ≡ false ⊎ q ≡ false
&&-split = {!!}

=>-same : ∀ (p : Bool) → p => p ≡ true
=>-same = {!!}

=>-to-|| : ∀ (p q : Bool) → (p => q) ≡ (not p || q)
=>-to-|| = {!!}
```

```
=>-mp : ∀ (p q : Bool) → p => q ≡ true → p ≡ true → q ≡ true
=>-mp = {!!}

=>-antisymm : ∀ {p q : Bool} → p => q ≡ true → q => p ≡ true → p ≡ q
=>-antisymm = {!!}
```

References

[1] Peter Dybjer, *An Introduction to Programming and Proving in Agda*, 2017.

[2] Aaron Stump, *Verified Functional Programming in Agda*, Association for Computing Machinery and Morgan & Claypool Publishers, 2016.

[3] Wikipedia contributors, *Ackermann function*, `https://en.wikipedia.org/wiki/Ackermann_function`, 2017.

[4] Bengt Nordström, *The primitive recursive functions*, `http://www.cse.chalmers.se/edu/course/DIT310_Models_of_Computation/reading/The_primitive_recursive_functions.pdf`, 2014.

[5] Agda contributors, *The Agda User Manual*, `https://agda.readthedocs.io/en/latest/`, 2017.