Types for programs and proofs
Take home exam 2016

- Deadline: Friday 21 October at 12.00.
- Answers are submitted in the Fire system.
- Grades: 3 = 24 p, 4 = 36 p, 5 = 48 p. Bonus points from talks and homework will be added.
- The maximum score of the exam is 60 p, and in addition there are two optional problems each worth 8 p. You are not expected to solve all the problems. Choose carefully which ones you spend time on.
- In some of the problems you are asked to write programs and proofs in Agda. Alternatively, you may use Haskell for the programs, but you can of course not use it for the proofs. You can then get partial credit for careful, rigorous, handwritten proofs.
- Note that this is an *individual exam*. You are not allowed to help each other. If we discover that you have collaborated, both the helper and the helped will fail the whole exam. We will also consider disciplinary measures.
- Please contact Peter or Thierry if there is an ambiguity in a question or something else is unclear. We will publish any corrections and additions on the course homepage.

1. (a) Define the set of positive natural numbers $\mathsf{PosNat} = \{1, 2, 3, \ldots\}$ in Agda!
   (b) Define the arithmetic operations of addition, multiplication, and exponentiation on these positive natural numbers!
   (c) Prove the associativity law for addition for these positive natural numbers.
   (d) Prove the commutativity law of addition for these positive natural numbers.

   (6 p)

2. We know that

$$2 + 2 = 2 \times 2 = 2^2 = 4$$

for numbers. Similarly, we have that

$$\mathsf{Bool} + \mathsf{Bool} \cong \mathsf{Bool} \times \mathsf{Bool} \cong \mathsf{Bool} \to \mathsf{Bool}$$

where $A \cong B$ means that there is a bijection between A and B, that is, that there are $f : A \to B$ and $g : B \to A$ which are inverses of each other, that is, $g\,(f\,x) = x$ for all $x : A$ and $f\,(g\,y) = y$ for all $y : B$.

   (a) Implement these bijections in Agda!
   (b) Define what it means for two functions $f, g : \mathsf{Bool} \to \mathsf{Bool}$ to be equal! We here mean that they are "extensionally" equal, that is, that for each input they return the same output.
   (c) Prove that your functions in (a) are indeed bijections, where equality on $\mathsf{Bool} \to \mathsf{Bool}$ is extensional equality. (More explicitly: to prove that there is a bijection between $\mathsf{Bool} \times \mathsf{Bool}$ and $\mathsf{Bool} \to \mathsf{Bool}$ means to prove that there are $f : \mathsf{Bool} \times \mathsf{Bool} \to (\mathsf{Bool} \to \mathsf{Bool})$ and $g : (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \times \mathsf{Bool}$ which are inverses of each other, that is, $g\,(f\,x) = x$ for all $x : \mathsf{Bool} \times \mathsf{Bool}$ and $f\,(g\,y)$ is extensionally equal to y for all $y : \mathsf{Bool} \to \mathsf{Bool}$.)
   (d) Prove the general law that if $A \cong B$ and $B \cong C$, then $A \cong C$ in Agda. Here "=" means the identity type of Agda, see eg Identity.agda from the lectures.

   (12 p)

3. (a) Define the set of positive binary numbers BinNat with a leading 1. That is, the set $\{1, 10, 11, 100, 101, 110, \ldots\}$. Define the bijection $\mathsf{PosNat} \cong \mathsf{BinNat}$, where PosNat are your numbers from one. (In this part you only need to define the two functions $\mathsf{nat2bin} : \mathsf{PosNat} \to \mathsf{BinNat}$ and $\mathsf{bin2nat} : \mathsf{BinNat} \to PosNat$.) (4 p)
   (b) Prove that nat2bin and bin2nat are inverses of each other, that is, that they form a bijection. (Optional. This is quite a hard problem, where you need to do some non-trivial equational reasoning. There is a module in Agda's standard library which facilitates equational reasoning by providing some nice syntax.) (8 p)

4. **System T** is defined in "Dependent types at work" section 2.5 (see the home page of the course). Like PCF it is based on the simply typed lambda calculus and has the following constants in common.

   `True, False, Zero, Succ, if`

   However, System T does not have a fixed point combinator `fix` for defining *general recursive* functions, but only a *primitive recursion* combinator `natrec`.

   Your task is now to write some programs in System T and implement them in Agda, following "Dependent types at work". As these constants are called `true, false, zero, succ` and `if_then_else_` respectively.

   (a) The PCF-constants `pred` and `isZero` are not primitive in System T. Show how they can be defined using `natrec`!

   (b) Define the fibonacci function

   `fib : Nat -> Nat`

   in System T using `natrec`!

   (c) Show how to program in System T the function

   `min : Nat -> Nat -> Nat`

   such that `min m n` returns the minimum of `m` and `n`

   (6 p)

5. (a) Define an interface for queues `Queue`as a record in Agda. We show the first three lines of this record:

   ```
   record Queue (A : Set) : Set1 where
     field
       Q : Set
       ...
   ```

   Here `Q` is the type which implements the queue. Your task is to extend this record with four operations: `emptyQ` which builds an empty queue, `insertLast` which inserts a new element at the end of the queue, `removeFirst` which removes an element from the beginning the queue, and `first` which returns the first element of the queue. Use the `Maybe`-type so that you can return the exceptional element "nothing" if you try to remove or return the first element of an empty queue.

   (b) Instantiate your `Queue`-record in two ways, both based on using lists as the implementing type `Q`. In the first one you insert at the beginning of the list and remove from the end. In the second you insert at the end of the list and remove from the beginning. Implement queues as lists, and instantiate the operations.

   (c) Add some properties to the `Queue`-record! For example, that the first element of the queue is the same after having added a new element at the end of the queue. Can you think of more properties?

   (d) Now define a record for "indexed queues", that is queues of a given length. The first three lines of this record are:

   ```
   record IndexedQueue (A : Set) : Set1 where
     field
       Q : Nat -> Set
       ...
   ```

   Note that the implementing type is now an "indexed" type. Your indexed queues should have indexed versions of the same operations as in (a)

   (e) Instantiate your `IndexedQueue`-record in one way using the type of vectors instead of the type of lists. In this way you avoid using the `Maybe`-type.

   (12 p)

6. (a) In Ulf Norell's lecture on the 6 October he implemented a type checker for the simply typed lambda calculus. Your task is to extend this type checker so that it can deal with pairs. You will need to extend Type with a pair type, add a pair constructor and projections to RawTerm and Term and update the type checker and evaluator to handle the new constructs.

   (10 p)

   (b) Optionally you may also extend the parser and pretty printer.

   (8 p)

7. In Chapter 12 in Pierce there is a proof of normalization for the simply typed lambda calculus (theorem 12.1.6). Exercise 12.1.7 Pierce is about extending this proof to include booleans and products.

   Your task here is to extend this proof to a proof of normalization for System T. That is, you extend the simply typed lambda calculus with natural numbers and the constants described in problem 4 above.

   (10 p)