Types for programs and proofs
Take home exam 2015

- Deadline: Friday 23 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.
- Note that there is a total of 75 p in the whole exam, and that you are only expected to solve a fraction of 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. **Distributivity laws.** Prove the following distributivity laws in Agda:

   ```
   (b c d : Bool) -> b && (c || d) == (b && c) || (b && d)
   (B C D : Set) -> B & (C \/ D) <==> (B & C) \/ (B & D)
   (m n p : Nat) -> m * (n + p) == (m * n) + (m * p)
   ```

   Recall that proving a proposition is the same as constructing a proof object of the corresponding type! You can use any definitions you like (including ones from files given in the lecture) of equality (`==`), conjunction `&&`, and disjunction `||`, of Booleans; equivalence (`<==>`), conjunction `/\`, and disjunction `\/`, of propositions, and of equality (`==`), addition +, and multiplication * of natural numbers. (You may also change symbols and use unicode if you like.) (5 p)

2. **Integers.**

   (a) Define the set of (positive and negative) integers `Int` in Agda and the relation

   ```
   _==_ : Int -> Int -> Set
   ```

   of equality of integers. There are several ways to define the integers. Some have a unique representation for each integer and others may have several representations. It is up to you to choose one. (Hint: you may of course base your implementation on existing implementations of natural numbers.)

   (b) Define addition of integers as a function

   ```
   _+_ : Int -> Int -> Int
   ```

   Then prove that it respects equality:

   ```
   (m n : Int) -> m == m' -> n == n' -> m + n == m' + n'
   ```

   (c) Prove that integer addition is associative:

   ```
   (m n p: Int) -> (m + n) + p == m + (n + p)
   ```

   (d) What (reasonable) alternative implementations of integers (in Agda) can you think of? Discuss pros and cons! For example, how do they influence the difficulty of the above proofs?

   (10 p)

3. **Intuitionistic propositional logic.**

   (a) Implement the set of formulas in intuitionistic propositional logic as an inductive data type `Formula` in Agda. Informally we have the following ways to construct formulas:

   Atomic propositions X_n is a formula for each natural number n.

   Connectives If P and Q are formulas, then $P \vee Q, P \wedge Q, P \supset Q, \bot$, and $\top$ are formulas.

   (b) Write a function

   ```
   [[_]]_ : Formula -> Env -> Set
   ```

   which computes the meaning of a propositional formula relative to an environment `Env = Nat -> Set` which assigns a proposition (set) to each atomic proposition X_n. For example, if $\rho :$ `Env` and $\rho\, n = A$, then $[[X_n]]\rho = A$.

   (10 p)

4. **Monoids as records.** In Agda you can implement algebraic structures (and their corresponding notions in programming called "interfaces", "abstract data types", etc) as records. In the lectures we showed how to implement an algebraic structure (interface) for counters. We also showed a trivial implementation of counters as natural numbers.

   A monoid is an algebraic structure with an associative binary operation * and a unit element. That is, it is a set M and two operations

   ```
   _*_ : M -> M -> M
   id : M
   ```

   where * is associative and `id` is an identity wrt *. (See the wikipedia article for more information, but note that we here use a curried version of * and wikipedia uses an uncurried version.)

   (a) Your task is to implement a record for monoids in Agda. To do this you need to have fields both for the underlying set, for the operations, and for the three laws of associativity and identity.

   (b) Also implement three instances:

   - the monoid of Booleans with * as conjunction;
   - the monoid of natural numbers with * as addition;
   - the monoid of functions with * as composition.

   Hint: you first need to figure out the identities.

   You will get partial credit if you don't include the laws, only the operations! (10 p)

5. This is a problem related to Ulf Norell's lecture. Your task is to solve the problems specified in the file `Problem5.agda`. Note that this file imports the files `Bool.agda` and `Nat.agda` from Peter's lectures. (15 p)

6. **Untyped lambda calculus**

   We define $\delta = \lambda x.x\ x$ and $\Omega = \delta\ \delta$ and $t = \lambda x.\Omega$. Is t β-convertible to Ω? Why? (5p)

7. **Krivine Abstract Machine**

   We consider terms in de Bruijn notations

$$t \ ::= \ \mathsf{true} \mid \mathsf{false} \mid n \mid \lambda t \mid t\ t \qquad n \ ::= \ 0 \mid n+1$$

   and *environments* and *closures*

$$\rho, \nu \ ::= \ () \mid (\rho, v) \qquad u \ ::= \ t\rho$$

   The *values* are

$$v \ ::= \ \mathsf{true} \mid \mathsf{false} \mid (\lambda t)\rho$$

(a) **Big-step semantics**

We define then the following operational semantics

$$\frac{}{(\lambda t)\rho \Downarrow (\lambda t)\rho} \qquad \frac{}{\mathsf{true}\rho \Downarrow \mathsf{true}} \qquad \frac{}{\mathsf{false}\rho \Downarrow \mathsf{false}}$$

$$\frac{u \Downarrow v}{0(\rho, u)) \Downarrow v} \qquad \frac{n\rho \Downarrow v}{(n+1)(\rho, u) \Downarrow v}$$

$$\frac{t_0\rho \Downarrow (\lambda t)\nu \qquad t(\nu, t_1\rho) \Downarrow v}{(t_0\ t_1)\rho \Downarrow v}$$

We define also types and contexts

$$T \ ::= \ \mathsf{Bool} \mid T \to T \qquad\qquad \Gamma \ ::= \ () \mid \Gamma.T$$

with the usual typing rules

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \qquad \frac{\Gamma \vdash n : T}{\Gamma.S \vdash n+1 : T} \qquad \frac{}{\Gamma.T \vdash 0 : T}$$

$$\frac{\Gamma.S \vdash t : T}{\Gamma \vdash \lambda t : S \to T} \qquad \frac{\Gamma \vdash t_0 : S \to T \qquad \Gamma \vdash t_1 : S}{\Gamma \vdash t_0\ t_1 : T}$$

and finally

$$\frac{}{() : ()} \qquad \frac{\rho : \Gamma \qquad u : T}{(\rho, u) : \Gamma.T} \qquad \frac{\rho : \Gamma \qquad \Gamma \vdash t : T}{t\rho : T} \qquad \frac{}{\mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\mathsf{false} : \mathsf{Bool}}$$

Show that if $u \Downarrow v$ and $u \Downarrow v_1$ then $v = v_1$. (2p)

Show that if $u : T$ and $u \Downarrow v$ then $v : T$. (3p)

We define $C_T(u)$ by induction on T, if $u : T$ is derivable:

i. $C_{\mathsf{Bool}}(u)$ means that $u \Downarrow \mathsf{true}$ or $u \Downarrow \mathsf{false}$

ii. $C_{S\to T}(u)$ means that $u \Downarrow (\lambda t)\rho$ and $C_T(t(\rho, u_1))$ whenever $C_S(u_1)$ holds

Show that if $u_0 \Downarrow v$ implies $u_1 \Downarrow v$ then $C_T(u_0)$ implies $C_T(u_1)$. (2p)

Show that $u : T$ implies $C_T(u)$ (8p). (Hint: you will have to use $C_\Gamma(\rho)$ defined by $C_{()}()$ and $C_{\Gamma.T}(\rho, u)$ if $C_\Gamma(\rho)$ and $C_T(u)$.)

(b) **Small-step semantics**

We introduce stacks

$$S \ ::= \ () \mid u \cdot S$$

Krivine Abstract Machine is defined by the following small step semantics

$$(0, (\rho, (t, \nu)), S) \to (t, \nu, S) \qquad (n+1, (\rho, u), S) \to (n, \rho, S)$$

$$(t_0\ t_1, \rho, S) \to (t_0, \rho, (t_1\rho) \cdot S) \qquad (\lambda t, \rho, u \cdot S) \to (t, (\rho, u), S)$$

Show that if $t\rho \Downarrow (\lambda t_1)\rho_1$ then we have $(t, \rho, S) \to^* (\lambda t_1, \rho_1, S)$ for all S and if we have $t\rho \Downarrow \mathsf{true}$ (resp. $t\rho \Downarrow \mathsf{false}$) then $(t, \rho, S) \to^*$ $(\mathsf{true}, \rho_1, S)$ (resp. $(t, \rho, S) \to^* (\mathsf{false}, \rho_1, S)$) for some ρ_1. (5p)