Types for programs and proofs
Take home exam 2014

- Deadline: Friday 24 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 the relevant sections in Pierce contain useful information for solving the problems.
- Some of the problems ask you 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. Define a type family Bool^n in Agda such that

$$\mathsf{Bool}^0 = \mathsf{Unit} \qquad \mathsf{Bool}^{n+1} = \mathsf{Bool} \times \mathsf{Bool}^n$$

where Unit is a data type with one constructor $\langle\rangle : \mathsf{Unit}$. The type Bool^n has then 2^n element. Define then a function

$$\mathsf{taut} : (n : \mathsf{N}) \to (\mathsf{Bool}^n \to \mathsf{Bool}) \to \mathsf{Bool}$$

such that $\mathsf{taut}\ n\ F$ is True if, and only if, $F\ u$ is True for all possible 2^n values of the argument $u : \mathsf{Bool}^n$.

(5p)

2. Neither the law of excluded middle nor the law of double negation hold in *intuitionistic* logic.

   (a) However, excluded middle implies double negation:

   $$(A \vee \neg A) \to (\neg\neg A \to A)$$

   for all propositions A! Prove this in Agda (or on paper)!

   (b) What happens if you try to prove the converse in intuitionistic logic (or Agda):

   $$(\neg\neg A \to A) \to (A \vee \neg A)?$$

   Discuss!

   (c) Prove in Agda (or on paper) that if the law of double negation $\neg\neg X \to X$ holds for all propositions X, then the law of excluded middle $X \vee \neg X$ holds for all propositions X. (Note the difference to the formulation in (a)!)

   (d) Also prove in Agda that if the law of excluded middle holds for all propositions X, then the law of double negation $\neg\neg X \to X$ holds for all propositions X. (Hint: this follows easily from (a).)

(6p)

3. The Boolean operation *nand* (the negation of *and*) is functionally complete, since any other Boolean operation (not, and, or, if-then-else) can be implemented using it. Your task here is to show this in Agda.

   (a) First define `nand : Bool -> Bool -> Bool` in Agda.

   (b) In the lecture we defined a data type `Term` of Boolean expressions (terms) with constructors for true, false, and if-then-else. Modify this to get a data type `TermVar` of Boolean expressions with variables, which has an additional constructor `Var : Nat -> TermVar` (we encode variables as natural numbers: think of `Var i` as the variable x_i).

   (c) In the lecture we defined the "denotational semantics" of Boolean expression as a function

   ```
   [[_]] : Term -> Bool
   ```

   which maps a Boolean expression to its value. When we define the denotational semantics of Boolean expressions with variables, we instead do it relative to an *environment*, that is, a function which associates a value (a Boolean) to each variable:

   ```
   Env : Set
   Env = Nat -> Bool
   ```

   Hence the type of the interpretation function (the denotational semantics) is instead

   ```
   [[_]] : TermVar -> Env -> Bool
   ```

   Define this function in Agda!

   (d) Define a type `NandTermVar : Set` in Agda of Boolean expressions built up only by nand and variables!

   (e) Define the interpretation function (the denotational semantics) for Boolean expressions built up from nand and variables.

   ```
   [[_]]' : NandTermVar -> Env -> Bool
   ```

   (f) Define a translation

   ```
   tonand : TermVar -> NandTermVar
   ```

   which preserves the denotational semantics

   (g) Prove in Agda that the the denotational semantics is preserved, that is, that

   ```
   [[ tonand t ]]' env = [[ t ]] env
   ```

   for all `t : TermVar` and all `env : Env`.

   (h) Write the reverse translation

   ```
   toif : NandTermVar -> TermVar
   ```

   and prove that this too preserves the denotational semantics.

(14p)

4. On p 44-45 in Burstall's paper on "Proving properties of programs by structural induction" (presenented in one of the student talks, see link on the wiki) there is a section called **Two lemmas**. Your task is to formalize the content of this section in Agda.

   (a) On p 45 there is a definition of the polymorphic list functions *concat* and *lit*. Implement them in Agda! Hint: *concat* is the same as *append* and you may previously have defined it. Also note that the language ISWIM used by Burstall is untyped, but you should of course write a typed version of *lit* and *concat*.

   (b) The **Example** right after the definition of *lit* states that

   $$lit(+, (2, 3, 4), 1) = 10.$$

   Write Agda code to check this!

   (c) Prove the first of the two **Lemmas** in Agda.

   $$lit(f, concat(xs1, xs2), y) = lit(f, xs1, lit(f, xs2, y))$$

   Hint: you can follow Burstall's proof step by step, but note that when Burstall refers to the Induction Hypothesis, you also need to use a congruence rule which lets you substitute an equal expression for another equal expression. Such a congruence rule can be found in the file `Identity.agda` which was presented in a lecture, but you can also prove your own.

   (d) Prove the second **Lemma** ("Suppose xs is an $\alpha - list$, etc") in Agda. This is an induction principle for *lit*.

   (9p)

5. In one of the lectures we showed how to use Agda's records for specifying and implementing *abstract data types*, that is, collections of operations with their types (cf Java's *interfaces*). As an example we specified an abstract type `Counter` as a record, and then showed how it could be implemented by a number. See the file `Counter.agda`.

   Your task is to do the same for the abstract data type of sets (cf the `Set` interface in `java.util`). This is an abstract data type which can be instantiated with various implementations of finite sets (lists, sorterd lists, binary search trees, red-black trees, etc). More specifically

   (a) Define an Agda record `SetInterface` of sets with the operations `add, contains, remove, size` and `equal`. For simplicity you can restrict yourself to finite sets of natural numbers.

   (b) Instantiate this record with an implementation of sets in terms of lists. You can choose to either allow repetitions of elements or not.

   (c) Write down some properties (typically equations involving the operations) which any implementation of sets must satisfy! Extend `SetInterface` with these laws to a new record `SetInterfaceWithLaws`!

   (d) Prove that your implementation of sets by lists satisfies these laws!

   (e) Can you make a generic record for sets of elements of arbitrary type? Discuss possible problems!

   (8p)

6. (a) We recall the syntax of typed λ-calculus

   $$t \ ::= \ x \mid t \ t \mid \lambda x : T.t$$

   Show that a closed term in β-normal form can be written $\lambda x_1 : T_1 . \ldots \lambda x_k : T_k . x \ t_1 \ \ldots \ t_l$ where we can have $l = 0$ and where $t_1, \ldots, t_l$ are terms in the context $x_1 : T_1, \ldots, x_k : T_k$ and x is one of the variable $x_1, \ldots, x_k$.

   (b) Use this to enunerate the simply typed λ-terms that can be written without using any constant of type

   i. $\mathsf{Bool} \to \mathsf{Bool}$
   ii. $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$
   iii. $\mathsf{Bool} \to (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}$
   iv. $(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}$
   v. $((\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Bool}$

   (7p)

7. Consider simply typed λ-calculus with one base type A one constant $a : A$ and one constant $g : A \to A \to A$. Find all possible closed terms f of type $A \to A$ such that $f \ a = g \ a \ a$.

   (Hint: use the previous exercise.)

   (5p)

8. Let $\rightarrow$ be a binary relation on a set A. We define $\rightarrow^*$ to be the reflexive transitive closure of $\rightarrow$ (least reflexive transitive relation containing $\rightarrow$) and $\simeq$ to be the reflexive symmetric transitive closure of $\rightarrow$. We say that a relation R on a set is *confluent* if $R(a, b)$ and $R(a, c)$ imply that there exists d such that $R(b, d)$ and $R(c, d)$. Show that if $\rightarrow$ is confluent then so is $\rightarrow^*$. Show that if $\rightarrow^*$ is confluent then $a_0 \simeq a_1$ if, and only if, there exists b such that $a_0 \rightarrow^* b$ and $a_1 \rightarrow^* b$.

   You should either give a rigorous handwritten proof or prove it in Agda!

(6p)