Types for programs and proofs
Take home exam 2017

- Deadline: Friday 20 October at 12.00.
- Answers are submitted in the Fire system.
- Grades: Chalmers: 3 = 24 p, 4 = 36 p, 5 = 48 p. GU: G = 24 p, VG = 48 p. Bonus points from talks and homework will be added.
- The maximum score of the exam is 60 p. (In addition to this there is an optional problem worth 4 p.)
- 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. Programming with lists in Agda. You may start with the code in the file `List.agda` presented in the lectures.

   (a) Write a function `length` that computes the `length` of a list.

   (b) Write a function `snoc` which adds an element at the end of a list.

   (c) Write a proof in Agda which shows that `snoc` increases the length of a list by 1.

   (d) Write a function `reverse` which reverses a list.

   (e) Write a proof in Agda which shows that the `length` of a list is preserved by `reverse`.

   (5 p)

2. Programming with vectors in Agda. You may start with the code in the file `Vector.agda` presented in the lectures.

   (a) Write a function `vsnoc` which adds an element at the end of a vector. The type should record that the length of the vector is increased by 1.

   (b) Write a function `vreverse` which reverses a vector. The type should record that the length of the vector is preserved by `vreverse`.

   (3 p)

3. Reasoning about equivalence and isomorphism in Agda.

   (a) Define logical equivalence in Agda! Two propositions $A, B : \mathsf{Set}$ are logically equivalent ($A \Leftrightarrow B$) if they imply each other.

   (b) Prove reflexivity of logical equivalence, that is, that $A \Leftrightarrow A$.

   (c) Prove symmetry of logical equivalence.

   (d) Prove transitivity of logical equivalence.

   (e) Define isomorphism in Agda! Two propositions $A, B : \mathsf{Set}$ are isomorphic ($A \cong B$) if there are functions $f : A \to B$ and $g : B \to A$ which are mutually inverse.

   (f) Prove in Agda that $A \cong B \to A \Leftrightarrow B$ for all $A, B : \mathsf{Set}$.

   (g) Prove in Agda that it is not the case that $(A \Leftrightarrow B \to A \cong B)$ for all $A, B : \mathsf{Set}$.

   (7 p)

4. (a) In one of his lectures Thierry presented a simplified version of a classic proof of compiler correctness by McCarthy and Painter. The main theorem (0.1) refers to the relation $\mapsto^*$, the reflexive-transitive closure of the relation of "small step semantics" for the machine in question. Thierry defined the reflexive-transitive closure R^* of an arbitrary relation R by an inductive definitions as follows (cf Homework 3): it is the least relation R^* such that

   - R^* is reflexive
   - if aRb and bR^*c then aR^*c

   However, in Peter's formalization of this proof in `McCarthyPainter.agda`, he used a different definition of reflexive-transitive closre. Modify Peter's proof so that it instead uses the above definition.

   (b) Extend (in Agda) the compiler correctness proof to a source language with a successor operation, that is, the grammar of the source language is now

$$e ::= n \mid e + e \mid \mathit{succ}\, e$$

   Hint. You need to extend

   - the data type Expr of source language expression with *succ*
   - the function val;
   - the data type Code of machine language with an instruction `SUC`;
   - the notion of one-step reduction $\Rightarrow$ with a clause sucstep;
   - and finally extend the compiler correctness proof.

   (6 p) An optional question is to extend the proof to the whole language of arithmetic expressions! In order to avoid handling type-errors you can implement false by 0 and true by the numbers greater than 0. (Optional, 4 p)

5. (a) Define an Agda record corresponding to Haskell's Ord class!
   `https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Ord.html`
   It suffices if your class has a few operations (e g `compare`, $<, \leq$).

   (b) Define the two instances Bool and Nat.

   (c) Write down a few axioms characterizing the property that you have a total order (see wikipedia for the definition).

   (d) Finally prove that your instances satisfies those axiom.

   (e) Explain what the subclass declaration `Eq a => Ord a` would mean for Agda records. (Optional, 4 p)

   (7 p)

6. Add a `natrec` construct to the simply typed λ-calculus from the Agda lecture in week 6. `natrec` is a primitive recursion combinator. It takes three arguments: the first is the base case, the second is the step case, and the third is a natural number. It should have the following typing rule and semantics

$$\frac{}{\Gamma \vdash \texttt{natrec}\ A : A \Rightarrow (nat \Rightarrow A \Rightarrow A) \Rightarrow nat \Rightarrow A}$$

$$\frac{}{\texttt{natrec}\ A\ z\ s\ \texttt{zero} \mapsto z}$$

$$\frac{}{\texttt{natrec}\ A\ z\ s\ (\texttt{suc}\ n) \mapsto s\ n\ (\texttt{natrec}\ A\ z\ s\ n)}$$

   (a) Extend the raw and well-typed terms in `Term.agda`.

   (b) Extend the well-typed evaluator in `Term/Eval.agda`.

   (c) Extend the type checker in `TypeCheck.agda`.

   (d) Using `natrec`, define raw terms

   ```
   add : RawTerm
   mul : RawTerm
   fac : RawTerm
   ```

   implementing addition, multiplication, and the factorial function. Addition and multiplication should have type `nat => nat => nat` and factorial should have type `nat => nat`.

   Verify that your raw terms are correct by proving

   ```
   facCheck : evalRaw nat (app fac (lit 6)) ≡ pure 720
   facCheck = refl
   ```

   (`evalRaw` is defined in `Main.agda`)

   (8 p)

7. In this question we follow the note on simply typed lambda-calculus which you can find on the home page of the course. Each question can be solved informally or using Agda (except the last question, which has to be in Agda).

   We define $N(c)$ to mean $\exists v\ (c \mapsto^* v)$ which expresses that c *normalizes*.

   (a) Prove Theorem 2.
   (6 p)

   (b) Prove Lemmas 1 and 2.
   (6 p)

   (c) Define by induction on the type A a value $0_A : A$. Use this to prove that if $c : B$ where B is an *arbitrary* type then we have $N(c)$

   Hint: A Lemma can be that $N(c)$ holds whenever $N(c\ c')$ holds.
   (6 p)

   (d) Define internally in Agda the term of type N

$$t_n = (((\ldots((\mathsf{twice}_n\ \mathsf{twice}_{n-1})\ \mathsf{twice}_{n-2})\ldots)\ \mathsf{twice}_0)\ \mathsf{S})\ \mathsf{z}$$

   so that t_n is a term of type N. So in Agda, you have to define a function of type $\mathsf{Nat} \to \mathsf{Term}$ where Nat is the type of natural numbers and Term the type of terms.

   Hint: Try first to generalize the question.
   (6 p)