Types for programs and proofs
Take home exam 2011

- Deadline 1 November 2011 at 16.00.
- Electronic answers should be sent by email to Peter (`peterd@chalmers.se`). Handwritten answers can be left in Peter's office (EDIT, 6467) 15.00 - 16.00 on 1 November. Agda code must be sent by email.
- 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.
- 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 if something is unclear.

1. **PCF** is a small typed functional programming language which is often used in research papers. It is based on the simply typed lambda calculus with two base types `Bool` and `Nat` for truth values and natural numbers respectively. There are the following constants (”combinators”):

   ```
   True, False, Zero, Succ, if, pred, isZero, fix
   ```

   where `fix` is the *fixed point* combinator which is used for encoding general recursive definitions (cf the lectures and Pierce section 11.11). A polymorphic version of PCF is presented in the lecture notes:

   ```
   http://www.cse.chalmers.se/edu/course/DAT140_Types/PCF.pdf
   ```

   (The link is on the course home page.) Note that this version of PCF is a sublanguage of Haskell, so you can implement your PCF functions in Haskell by restricting yourself to using variables, application, abstraction, and the above constants.

   See also the wikipedia page for ”Programming language for Computable Functions” where a monomorphic version of PCF with only one base type (nat) is presented.

   (a) Define exponentiation in PCF! That is, define a function

   ```
   power : Nat -> Nat -> Nat
   ```

   such that `power` $m\,n$ is m^n.

   (b) Define equality of natural numbers

   ```
   (==) : Nat -> Nat -> Bool
   ```

   in PCF.

   (c) Define the minimum function

   ```
   min : (Nat -> Bool) -> Nat
   ```

   It takes a function f and returns the least value m such that $f\,m = True$. Again, use `fix`!

   The key point is to use PCF’s fixed point combinator to express recursion! (7 p)

2. **System T** is defined in "Dependent types at work" section 2.5. Like PCF it is based on the simply typed lambda calculus, and has the following constants in common.

   ```
   True, False, Zero, Succ, if
   ```

   (In "Dependent types at work System T is programmed in Agda, and these constants are called `true, false, zero, succ` and `if_then_else_` respectively.) However System T does not have a fixed point combinator `fix` for defining general recursive functions, but only a *primitive recursion* combinator `natrec`. The key point in the following is to use `natrec` to define functions.

   (a) Define exponentiation in System T! That is, define a function

   ```
   power : Nat -> Nat -> Nat
   ```

   such that `power` $m\,n$ is m^n.

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

   (c) Define equality of natural numbers

   ```
   (==) : Nat -> Nat -> Bool
   ```

   in System T.

   (d) Which programming language is more powerful, PCF or system T (we only consider well-typed programs)? A system is less powerful than another if each partial function programmable in it is also programmable in the other. Are both systems equally powerful, or is one of them strictly more powerful than the other, or is none of them more powerful than the other? Motivate! Hint: Can you define the minimum function in problem 1 (c) in System T?

   (10 p)

3. **System F** is presented in Sections 23.3 and 23.4 in Pierce.

   (a) The swap combinator in untyped lambda calculus is defined as $\lambda fxy.f\,y\,x$.

   - Write the corresponding combinator in System F!
   - What is its type in System F? This type should be as "polymorphic" as possible.

   (b) Red-black trees are binary search trees where nodes are colored red or black. (Note that red-black trees can be empty.)

   - Define a Haskell data type `RedBlackTree` for red-black trees! (Remark: this data type does not need to encode the special red-black tree properties, like the absence of "double reds", etc.)
   - Define the corresponding (closed) type expression `RedBlackTree` for red-black trees in System F. Hint: Look at the encoding of lists in 23.5 in Pierce!
   - Your Haskell type probably has two constructors. Define the corresponding constructors for `RedBlackTree` in System F.
   - Define a function which checks whether a red-black tree is empty!

   (10 p)

4. **Existential types** are presented in chapter 24 in Pierce.

   Use existential types for defining an abstract data type of bags of priority queues, with operations `minElement, addElement` and `removeElement`. Cf section 24.2, especially the solution of EXERCISE 24.2.1.

   (5 p)

5. **Agda.** See

   `http://www.cse.chalmers.se/edu/course/DAT140_Types/ExamProblem5.agda`

   There is a preamble in

   `http://www.cse.chalmers.se/edu/course/DAT140_Types/Preamble.agda`

   (15 p)

6. 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 2 above.

   (13 p)