# UNIVERSITY OF DUBLIN
# TRINITY COLLEGE

Faculty of Engineering, Mathematics, and Science

School of Computer Science and Statistics

Integrated Computer Science
Year 4 Annual Examination

Hilary Term 2015

CS4012: Topics in Functional Programming

Tuesday, 13th January

Luce Upper

09:30–11:30

Mr. Glenn Strong

**Instructions to Candidates:**

Attempt **three** questions.
(all questions are worth equal marks)

There is a Reference to useful functions at the end of the paper (p6).

**Materials permitted for this examination:**

none

**Q1.**

The following data type can be used to represent Boolean values

```
data BoolExp = And [BoolExp]
             | Or [BoolExp]
             | Not BoolExp
             | Var String
```

And the following type represents a mapping from variable names to values:

```
type Env = [ (String,Bool)]
```

(a) Define a function `eval` to evaluate `BoolExp` values given some environment, using the type below. Make this definition as lazy as you can (that is, try to avoid returning `Nothing` unless you have no alternative).

```
eval :: Env -> BoolExp -> Maybe Bool
```

[12 Marks]

(b) Define functions called `tautology` and `contradiction`:

```
tautology, contradiction :: BoolExp -> Bool
```

such that `tautology` will be true iff the given expression *must* be true in any environment (that defines all of its variables), and `contradiction` will be true iff no environment can make it true.

[16 Marks]

(c) Explain the distinction between *type checking* and *type inference*, and discuss how what influence this distinction may have had on the design of the Haskell type system.

[5 marks]

**Q2.**

(a) How do generalised algebraic data types (GADTs) in Haskell allow programmers to express constraints on data types, and why might it be useful to do this (give at least two reasons)?
[10 Marks]

(b) Normal lists in Haskell are *homogeneous* (that is, all the elements of the list must be of the same type). It is possible to define *heterogeneous* lists using GADTs. Give a definition of such a list type, and explain two mechanisms that could be used to ensure that such lists can be useful (that is, how a programmer can perform operations on the values in the list)
[15 Marks]

(c) It is sometimes said that GADTs allow Haskell to simulate some features of dependent types. Explain what is meant by this, and include an explanation of what is meant by the term "dependent type".
[8 Marks]

**Q3.**

(a) Explain what a *monad transformer* is, and indicate why it is necessary to have a special technique for this (i.e. explain the issue with combining arbitrary monads).

[10 Marks]

(b) Give an implementation for the Error Transformer Monad (that is, give the code for a monad transformer that can add the ability to *fail* (or throw exceptions) to a sequence of actions in another monad). Provide primitive operations `throw` and `catch` with these types:

```
throw :: Monad m => String -> ErrorT m a
catch :: Monad m => ErrorT m a -> (String -> ErrorT m a) -> Error m a
```

[14 Marks]

(c) Explain why the Haskell IO monad cannot be a monad transformer.

[4 Marks]

(d) What does the "lift" function do in the monad transformer class?

[5 Marks]

**Q4.**

(a) The standard Haskell libraries offer both Parallel (through 'par' and 'pseq') and Concurrent (through 'forkIO') approaches to programming. Explain how these approaches differ semantically.

[10 Marks]

(b) Software Transactional Memory (STM) represents an attempt to simplify approaches to concurrent programming by providing an alternative to lock based approaches. Explain the STM approach to concurrency, sketch how an implementation might be provided, and discuss briefly the limitations of the STM approach to designing concurrent programs.

[15 Marks]

(c) Is there a safe way to incorporate either IO actions or `MVars` in STM, and if not why not?

[8 Marks]

## Reference

General:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

class MonadTrans t where
    lift :: Monad m => m a -> t m a
```