THE UNIVERSITY OF AUCKLAND

FIRST SEMESTER, 2018
Campus: City

COMPUTER SCIENCE

Web, Mobile and Enterprise Computing

(Time Allowed: TWO hours)

NOTES: Attempt all questions.

Marks for all questions total 70.

This exam counts for 70% of your final grade.

Write as clearly as possible.

The appendix recalls a few definitions and samples based on the standard F# core library – you can use them in your answers, as appropriate.

1. [9 marks] **Monad Laws**

a) State the three fundamental monad laws in terms of **`unit`** and **`bind`** functions.

b) State the three fundamental monad laws in terms of **Kleisli** operators.

c) Prove that laws in version (a) imply laws in version (b) –justify *all* intermediate steps.

a)

```
1) bind unit = id
2) unit >> bind g = g
3) bind f >> bind g = bind (f >> bind g)
```

b)

```
 I.   f >=> unit = f
II.   unit >=> g = g
III.  (f >=> g) >=> h = f >=> (g >=> h)
```

c) (1), (2), (3) are the unit/bind monad rules of (a)
(A) is the associativity of (classical) function composition
(K) is the Kleisli fish definition: let (**>=>**) f g = f >> (bind g)

```
 I.   f >=> unit
      =K f >> bind unit
      =1 f >> id
      =id f
II.   unit >=> g
      =K unit >> bind g
      =2 g
III.  (f >=> g) >=> h
      =K (f >> bind g) >> bind h
      =A f >> (bind g >> bind h)
      =3 f >> bind (g >> bind h)
      =K f >> bind (g >=> h)
      =K f >=> (g >=> h)
```

2. [8 marks] **Monad Builders and Sugared Monads**

For this question, refer to the skeletal monad builder given in the Appendix.

a) Discuss how monad builder methods map to **`unit`** and **`bind`** functions (cf. Appendix).

b) Discuss how the following sugared monad expressions map to monad builder methods: **`let!`**, **`do!`**, **`return!`**, **`return`**; e.g. **`return`** `t` maps to `m.Return t`

c) Translate the following computations into equivalent sugared monad expressions:

`let a = unit t`	e.g. `let a = m { return t }`
`let b = bind f m`	?
`let c = map h m`	?
`let d = flat mm`	?

a) Also, a short discussion
- `MBuilder.Bind (m, f) = bind f m`
- `MBuilder.ReturnFrom m = m`
- `MBuilder.Return t = unit t`

b) Also, a short discussion
- **`let! do!`** map to `m.Bind`
- **`return!`** maps to `m.ReturnFrom`
- **`return`** maps to `m.Return`

c)

```
let a = m { return t }
let b = m { let! t = m; return! f t }
let c = m { let! t = m; return h t }
let d = m { let! m = mm; return! m }
```

3. [9 marks] **Unbounded Non-Determinism**

Discuss how the Actor model supports unbounded non-determinism. Show a specific example, in F#-like code or pseudo-code.

Unbounded nondeterminism is a property of concurrency by which the amount of delay in servicing a request can become unbounded.

In Actor systems, unbounded non-determinism is supported by not bounding the transit time of messages, but guaranteeing their eventual delivery.

A textbook example is an actor system which is can non-deterministically return any natural number (however large) and is also guaranteed to terminate.

```
let rec und = MailboxProcessor.Start (fun inbox ->
    let rec loop count =
        async {
            und.Post 1
            let! msg = inbox.Receive ()
            match msg with
            | -1 ->
                printfn "%d" count
                return ()
            | _ ->
                return! loop (count+1)
        }
    loop 0)

...
und.Post -1
...
```

4. [7 marks] **Actor Supervision**

Discuss the concepts of actor supervision and monitoring, with reference to Akka.NET.

Supervision describes a dependency relationship between actors: the supervisor delegates tasks to subordinates and therefore must respond to their **failures**. When a subordinate detects a failure (i.e. throws an exception), it suspends itself and all its subordinates and sends a message to its supervisor, signalling failure. Depending on the nature of the work to be supervised and the nature of the failure, the supervisor has a choice of the following four options:

- **Resume** the subordinate, keeping its accumulated internal state
- **Restart** the subordinate, clearing out its accumulated internal state
- **Stop** the subordinate permanently
- **Escalate** the failure to the next parent in the hierarchy, thereby failing itself

Top-level supervision: Root Guardian, User Guardian (user actors), System Guardian (system actors).

Monitoring. In contrast to the special relationship between parent and child described above, each actor may monitor any other actor. Since actors emerge from creation fully alive and restarts are not visible outside of the affected supervisors, the only state change available for monitoring is the transition from alive to dead. Monitoring is thus used to tie one actor to another so that it may react to the other actor's **termination**, in contrast to supervision which reacts to **failure**.

Short discussion possible: different ways to terminate an actor.

Provided as a built-in pattern the Akka.Pattern.BackoffSupervisor implements the so-called **exponential backoff supervision strategy**, starting a child actor again when it fails, each time with a growing time delay between restarts.

There are two classes of supervision strategies which come with Akka: **OneForOneStrategy** and **AllForOneStrategy**. Both are configured with a mapping from exception type to supervision directive and **limits on how often a child is allowed to fail before terminating it**. The difference between them is that the former applies the obtained directive only to the failed child, whereas the latter applies it to all siblings as well.

5. [7 marks] **Actor/Mailbox vs. CMC/Hopac**

Discuss the conceptual similarities and differences between the Actor model, implemented by F# Mailbox, and the Concurrent ML (CML) model, implemented by F# Hopac.

Both Actors/Mailbox and CML/Hopac are models of message passing concurrency, between asynchronous processes. Major differences:

- Different thread weights:
  - Actors use usual (i.e. rather heavyweight and costly) system or managed **threads**.
  - Hopac uses a huge numbers of lightweight "threads", called **jobs**, served by their own thread pool. Unlike usual threads, **lightweight jobs can be blocked** without affecting the system.
- Different basic message passing primitives:
  - Actors are based on **asynchronous** post/receive primitives for actors, supported by **mailbox**-type queues.
  - Hopac is based on **synchronous** channels, essentially **rendezvous** primitives which do not need queues.
  - However, one can build synchronous communications on top of actors and, vice versa, communications channels on top of Hopac jobs.

Where applicable, queue-less rendezvous operations are faster and less demanding on system resources. Hopac is designed and optimized to scale as the number of such relatively independent lightweight elements is increased, e.g. on **parallel** systems.

However, there is no direct support for **distributed** computing, e.g. clusters or clouds.

Bonus if example(s) - but not required.

6. [8 marks] **Software Extensibility**

Discuss the pros and cons of using plugins as an architectural means to allow software extensibility.

7. [8 marks] **Caching**

a) What are the strengths and weaknesses of using caching within the context of software development. Discuss how the strengths can be used to improve the software and how the weaknesses could be overcome.

b) Discuss the caching support HTTP provides and how it could be used in software development.

8. [7 marks] **Compression**

a) Discuss pros and cons of using compression in software development.

b) What is the support that HTTP offers for compression and discuss how such support might be useful in software development.

9. [7 marks] **Software Security**

Discuss how data at rest and in transit could be protected, and how such protection affects software development. Your discussion should include the strengths and weaknesses of the methodologies.

APPENDIX

The F# maybe type (an alias for the option type):

```
type option<'T> =
   | Some of 'T
   | None
type Maybe<'T> = option<'T>          // 'T option
```

Sample F# functions from the Seq module (signatures):

Seq.singleton : 'T -> seq<'T>	unit
Seq.map : ('T -> 'U) -> seq<'T> -> seq<'U>	map
Seq.collect : ('T -> seq<'U>) -> seq<'T> -> seq<'U>	bind
Seq.concat : seq<seq<'T>> -> seq<'T>	flat

Kleisli operators on options

```
// ( >>= ) : m:'a option -> g:('a -> 'b option) -> 'b option
let (>>=) m g = bind g m
// ( >=> ) : f:('a -> 'b option) -> g:('b -> 'c option) -> ('a -> 'c option)
let (>=>) f g = f >> (bind g)
```

Skeletal monad builder

```
type MBuilder () =
   member this.Bind (m, g) = … ?
   member this.ReturnFrom m = … ?
   member this.Return t = … ?              e.g. = unit t
let m = MBuilder ()
```


APPENDIX CONTINUES

Skeletal actor implementation

```
let rec agent = MailboxProcessor.Start (fun inbox ->
        let rec loop … =
                async {
                        let! msg = inbox.Receive ()
                        match msg with …
                        return! … or return …
                }
                loop …)
```

Diagram chasing