UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

ADVANCES IN PROGRAMMING LANGUAGES

Wednesday 18 th May 2011

09:30 to 11:30

Year 4 Courses

Convener: D. K. Arvind
External Examiners: K. Eder, A. Frisch

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

CALCULATORS MAY NOT BE USED IN THIS EXAMINATION

1. This question is about programming for concurrency.

   The code below describes a Java class Point which represents points on a plane. The two methods reset and move change the position of the point, which is stored in the coordinate fields x and y.

```
public class Point {
  int x, y;

  public synchronized void reset () {
    System.out.println( "Resetting to origin" );
    x = 0;
    y = 0;
    System.out.println( "Completed reset" );
  }

  public synchronized void move (int dx, int dy) {
    System.out.println( "Moving by (" + dx + "," + dy + ")" );
    x = x+dx;
    y = y+dy;
    System.out.println( "Completed move" );
  }
}
```

   These methods are *thread safe*.

   (a) Describe what it means for methods to be *thread safe*. [*1 mark*]

   (b) The **synchronized** keyword of Java used here is important for concurrent programming. Explain what happens when a synchronized Java method like move is invoked, compared to an unsynchronized method. [*4 marks*]

   (c) In an unfortunate attempt to speed up code, a misguided programmer removes the **synchronized** keyword to give a similar class, which we shall call BadPoint. Here is the code.

```
public class BadPoint {
  int x, y;

  public void reset () {
    System.out.println( "Resetting to origin" );
    x = 0;
    y = 0;
    System.out.println( "Completed reset" );
  }

  public void move (int dx, int dy) {
```

```
    System.out.println( "Moving by (" + dx + "," + dy + ")" );
    x = x+dx;
    y = y+dy;
    System.out.println( "Completed move" );
  }
}
```

Suppose we have a BadPoint bp with x=1 and y=2. Give an example of how calls to the methods of bp could lead to incorrect results. Include information about which methods start and finish when, what is printed out when, how the values of x and y change, and what are their final values. [*12 marks*]

(d) There is a large design space for concurrent languages. Two requirements are for:

- Co-operation, to allow tasks to work together;
- Separation, to prevent inconsistent use of shared resources.

Two approaches to concurrent programming, different from that built in to Java, are:

(i) The Actor Model;

(ii) Software Transactional Memory.

For each one, explain briefly what distinguishes these paradigms, and in particular how they provide *co-operation* and *separation*. [*8 marks*]

2. This question is about bidirectional programming, which is motivated by the *view update* problem. Recall that a data view $v \in V$ is generated by a function $get(s)$ from a source value $s \in S$, where S and V are respectively the sets of possible source and view values. Given an updated view v', a corresponding change is provided by a function $put(v', s)$, that gives a new source value s'.

   (a) Besides database view updates, give two other application examples for bidirectional programming and explain why they fit a similar setting. [*2 marks*]

   (b) The get and put functions are required to satisfy two fundamental laws: the **PutGet** law and **GetPut** law. State these laws. [*2 marks*]

   (c) A restricted special case of bidirectional programming is when the *get* function is *bijective*, so it has a unique inverse.

      (i) Explain technically and informally why this is a big simplification. [*4 marks*]

      (ii) Give an example application of bijective programming [*2 marks*]

   (d) Boomerang is a general bidirectional programming language for string processing. It uses an abstraction called a *lens* which combines *get*, *put* and a third function, *create*. With reference to the Boomerang program below, answer the questions which follow it.

```
let NAME  = [a-zA-Z.]+ . " " . [a-zA-Z. ]+
let UID   = [a-z]+
let EMAIL = UID . "@" . [a-z.]+

let csv : lens =
    (copy NAME) . del ": " . (ins ", ") . (copy EMAIL)

let csvs : lens = csv . (newline . csv)*

let master : string =
<<
John Backus: backus@ibm.com
Stephen C. Kleene: skleene@maths.wisc.edu
>>
```

      (i) What is the result of `csvs.get master`? [*1 mark*]

      (ii) Give a regular expression type for the *source* of `csvs`. [*2 marks*]

      (iii) Give the semantics of `del ": "` by defining its lens functions. [*3 marks*]

      (iv) Define a lens which maps from the `csvs` *destination* to a view showing on each line a first name, a colon and space, and then a user name (the part of the email address before "`@`"). For example, the first line would be shown as `John: backus`. Be sure that your definition is precise. [*5 marks*]

      (v) Using this lens, show how to update its target view by changing Stephen Kleene's username to `sck` and put the change back into the original `master` list. Recall Boomerang's syntax for invoking the *put* function of a lens is *`lens`*`.put` *`newview`* `into` *`source`*. [*4 marks*]

3. This question is about uses of types in programming languages.

(a) The following are contrasting features that may appear in typed programming languages.

(i) Ad-hoc polymorphism vs. Parametric polymorphism

(ii) Homogeneous lists vs. Heterogeneous lists

(iii) Types vs. Type constructors

For each contrasting pair, do the following:

- Explain the difference between them, in general terms.
- Give a example of each feature.

Each example can be in any programming language — Haskell, Java, Scala, or whatever you think appropriate — but you must say which language it is. Different examples can use different languages. [*12 marks*]

(b) As well as values and types, the Haskell language uses a *kind* system. Give the kind of each of the following:

(i)	`Int`	fixed-precision integers
(ii)	`Maybe`	optional value
(iii)	`[]`	list formation
(iv)	`(,)`	pairing

[*4 marks*]

(c) The following Haskell code declares a representation for an infinite grid of data, with values at every integer coordinate point `(x,y)`. For example, a `Grid Double` could represent thermal simulation across a surface, giving a real-valued temperature at every coordinate point.

```
data Grid a = Grid ((Int,Int) -> a)  -- Grid of data values, each of type a

allZero :: Num a => Grid a           -- The everywhere-zero data grid
allZero = Grid (\(x,y) -> 0)         -- Valid for any numeric type a

coord :: Grid (Int,Int)              -- The coordinate grid
coord = Grid (\(x,y) -> (x,y))       -- Value is always that point's coordinates
```

Two example grids are `allZero` and `coord` given here.

(i) Write a function `fillGrid` which gives a grid that has the same value everywhere.

```
fillGrid :: a -> Grid a
```

[*2 marks*]

(ii) Write a function `sumGrid` which adds a list of grids pointwise, summing the values at each coordinate.

```
sumGrids :: Num a => [Grid a] -> Grid a
```

Note that because of the type qualification Num a =>, addition (+) is sure to be defined at type a. [*4 marks*]

(iii) Write an instance declaration to show that Grid is a Functor.

```
class Functor f where
fmap :: (a -> b) -> f a -> f b
```

[*3 marks*]