UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

ADVANCES IN PROGRAMMING LANGUAGES

Thursday 20th May 2010

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. (a) There are two major variants of operational semantics: *small-step* semantics $S, C \to S', C'$ and *big-step* semantics $S, C \Downarrow S'$.

   (i) Explain the difference between the two forms, using an example program with two instructions. [*4 marks*]

   (ii) Name two sorts of programs for which the small-step semantics may be more appropriate than the big-step, explaining why. [*4 marks*]

   (b) Operational semantics can be used to justify Hoare Logic for reasoning about programs. Recall the rule for while loops:

$$\frac{\{P \wedge (\mathsf{B} = \mathsf{true})\}\ \ \mathsf{C}\ \ \{P\}}{\{P\}\ \ \mathsf{while\ B\ do\ C}\ \ \{P \wedge (\mathsf{B} \neq \mathsf{true})\}}$$

   Consider the alternative, simplified for loop statement:

$$\mathsf{for\ x\ in\ m...n\ do\ C}$$

   where m and n are constant integers. When m > n it has the same meaning as skip. When m <= n, this has the same meaning as the sequence:

$$\mathsf{x{:=}m\ ;}\ \underbrace{\mathsf{C\ ;\ x{:=}x{+}1}}_{\mathsf{n{-}m\ times}}\ \mathsf{;\ C\ ;\ x{:=}x{+}1}$$

   i.e., the variable x takes on values from m up to n+1 in turn; C is executed exactly n−m+1 times.

   (i) Devise and explain two rules that can be used to prove Hoare triples in each case of the for loop. [*6 marks*]

   (ii) Use your rules to prove the triple:

$$\{\mathsf{a} = 0 \wedge \mathsf{n} > 0\}\ \ \mathsf{for\ x\ in\ \ 1...n\ do\ a{:=}a{+}b}\ \ \{\mathsf{a} = \mathsf{n} * \mathsf{b}\}$$

   mentioning places where you use the consequence rule. [*5 marks*]

   (c) In the *proof-carrying code* approach to security of third-party code, the code is packaged together with a checkable electronic *guarantee* that the code is well behaved. The guarantee might be a representation of a proof in Hoare logic, for example. When the code is delivered to the client, three checks must be made.

   (i) What are these checks? [*3 marks*]

   (ii) Suppose that an attacker is able to read and modify the delivered code and the electronic guarantee. Explain how each of the checks above might fail and hence why each is required. [*3 marks*]

2. This question is about some programming language features, as they appear in the *Haskell* language.

   (a) Haskell includes all of the following features.

   (i) Anonymous functions.
   (ii) Higher-order functions.
   (iii) Parametric polymorphism.
   (iv) Type classes and instances.

   For each one of these, do the following:

   - Explain what it is, in general terms.
   - Give an example, in Haskell, of how it can be used.
   - State what it is that your example does.

   Include types and kinds in your examples. [*20 marks*]

   (b) The following Haskell code declares StackOf as a *multiparameter type class*.

```
class StackOf s e where
  single  :: e -> s              -- Stack with one element
  push    :: e -> s -> s         -- Add element on top of stack
  pop     :: s -> Maybe (e,s)    -- Take element off top of stack, if possible
  isEmpty :: s -> Bool           -- Test to see whether stack is empty
```

   Here StackOf s e declares that type 's' can be used as a stack of elements of type 'e', with the operations listed.

   Write a complete Haskell **instance** declaration to show how the list datatype can be used as a stack of this class. [*5 marks*]

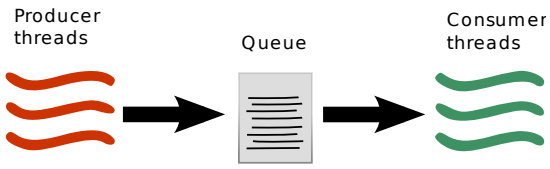
3. (a) Recall that Java provides a cooperative threading mechanism in which threads can start other threads and signal one another, and threads have access to shared memory. Consider the program below:

```
public class NumberWatcher {
    private static int number = 0;
    private static class NumberWatcherThread extends Thread {
        public void run() {
            while (number == 0)
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) { }
            System.out.println(number);
        }
    }
    public static void main(String[] args) {
        Thread nw = new NumberWatcherThread();
        nw.start();
        number = 2009;
        number = 2010;
    }
}
```

(i) What are the possible termination and console outputs behaviours of running this program? Explain carefully how they arise. [*8 marks*]

(ii) Explain and give code for two different additions to the program that could be made to ensure that the program only has a single outcome. [*6 marks*]

(b) Recall the *producer-consumer* pattern illustrated by the picture below.



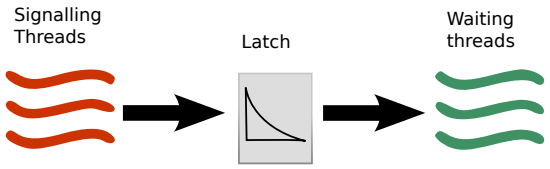
A *blocking queue* in the middle allows tasks to proceed independently on each side, coordinating the control flow among the threads.

(i) Name a thread-safe class in Java that implements a blocking queue. [*1 mark*]

(ii) Besides blocking calls, explain two other access policies available in Java's blocking queues. [*2 marks*]

QUESTION CONTINUES ON NEXT PAGE

(c) Another object that helps coordinate groups of threads is a *count-down latch*, illustrated below:



A count-down latch is initialised with a fixed count value. It can be decremented with the countDown function and is waited on with a blocking await function that blocks until the count down is complete. It also supplies a boolean ready function.

Give a Scala implementation of a class CountDownLatch. You should use an actor to encapsulate the counter state with the minimum amount of message passing needed to write each of the API functions.

You may find it helpful to recall the implementation of bounded buffers in Scala, given below. [*8 marks*]

```
class BoundedBuffer[T](N : int) {
  private case class Put(x: T)
  private case object Get

  def put(x: T) {
     buffer !? Put(x)
  }

  def get: T =
   (buffer !? Get).asInstanceOf[T]

 private val buffer = actor {
    val buf = new Array[T](N)
    var in = 0; var out = 0; var n = 0
    loop {
      react {
        case Put(x) if n < N =>
          buf(in) = x
          in = (in + 1) % N
          n = n + 1; reply()
        case Get if n > 0 =>
          val r = buf(out)
          out = (out + 1) % N
          n = n − 1; reply(r)
} }} }
```