Advances in Programming Languages

Lecture 20: Exam Preparation

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 29 November 2018
Semester 1 Week 11

THE UNIVERSITY *of* EDINBURGH

https://wp.inf.ed.ac.uk/apl18
https://course.inf.ed.ac.uk/apl

Outline

Outline

Formative Assessment — "Assessment for Learning"

Aims to help learning by providing feedback to both student and teacher about what areas need more work. Happens while learning a topic; ideally provides room for students to take risks, explore, attempt challenges, and even fail, without risking their course grade.

Summative Assessment — "Assessment of Learning"

Aims to provide information on what a student now knows or can do; measuring performance at the end of a topic. This is the test which determines course marks and grades.

These contrasting aims mean that formative and summative assessment often involve different kinds of task, and certainly different working environments.

Criterion-Referenced Assessment

Evaluating student performance against pre-set outcomes and targets: can you do this, or how many of these things can you do?

Norm-Referenced Assessment

Evaluating student performance against other students in the class, year, or some wider group: which students are better than average, and which worse?

This may involve "grading to the curve".

Approaches to Assessment

Outcome

Performance on a task, demonstration of a skill, display of reasoning.

Process

Continuous evaluation of coursework, participation, attendance, grade-point averages over an extended period.

Outline

Examinable Material

- Material in lectures (green-banded slides only; not the guest lecture)
- Exercises set as homework (only that explicitly set as homework; not the other miscellaneous references)
- That's it.

Exam Format

The format of the exam is the same from year to year, with a fixed rubric.

> Even so, you should read the rubric carefully on the day.

The paper has three questions, and you should choose and answer exactly two.

> I strongly recommend that you read all three questions before choosing which two to answer.

This is a "closed-book" exam: you may not take in any textbooks, printouts, notes, or other supporting material. Calculators are not permitted, and there will not be any questions that require them.

Grades

University of Edinburgh — Extended Common Marking Scheme

Mark (%)	Grade	Description	Honours Class	MSc Level
70-100	A	Excellent	I	Distinction
60-69	B	Very Good	II.1	Merit
50-59	C	Good	II.2	Pass
40-49	D	Undergraduate pass	III	Diploma, no MSc
30-39	E	Marginal Fail	Fail	Fail
20-29	F	Clear Fail		
10-19	G	Bad Fail		
0-9	H	Very Bad Fail		

Exam Preparation

(a) Read through lecture slides, your own notes, the homework exercises.

(b) Write your own notes on each topic. Summarize, organize, make lists of relevant points.

(For anything you really don't understand, consider watching the video. That's a fallback, though — where you possibly can, try to maintain understanding week by week in a course.)

(c) Practice past questions. Write out your answers, in full.

(d) Repeat items (c) and (d).

Exam Technique

- Read the question.
- Answer the question.
- Read the question again and make sure your answer provides precisely the details requested.
- Note that marking in APL is *positive* — marks are added for good things done, not taken away for things omitted. Marks start at zero :-(but then go only upwards :-)
- Look at the mark counts for an indication of how much or how little is required.
- Look at the mark counts to check how you are using time.
- READ THE QUESTION.

Exam Timing

This course has an exam during the April/May diet, at the end of the academic year, in common with other undergraduate honours courses in Informatics.

For such courses there is a period of between four weeks and four months between the end of lectures and the exam itself. This has the following impact:

- You need to recall material from the course a significant period after originally studying it, not just immediately after lectures end.

  *Being able to recall and apply knowledge well after the course has completed is an important outcome; it also engages the **spacing effect** where revisiting material helps learning.*

- You need to demonstrate ability across several different areas at the same time, with different exams covering material from different semesters.

  None of these courses exist in isolation, and working across different areas, combining knowledge, is a key skill for the effective application of Informatics.

What Kinds of Things are Assessed?

Subject area content obviously varies dramatically between different courses and their examinations, but some factors remain steady and most exams try to assess a range of skills.

Here is one listing of different elements that questions aim to stimulate and assess:

• Knowledge	Do you know the thing?
• Understanding	Do you know you know the thing?
• Ability to explain	Can you tell me the thing?
• Application of knowledge	Can you use the thing?
• Judgement	Can you tell which thing to use when?

In most cases questions and parts of questions will call on more than one of these; and there are also many other ways to classify learning and assessment goals.

Outline

2016/17 Question 2(a)

(a) System F extends the simply-typed lambda-calculus with explicit polymorphism: terms that take a type as a parameter. This language is expressive enough to define conventional algebraic datatypes from scratch. For example, if we assume predeclared types Int of integers and Bool of booleans, then we can define a type Prod of pairs of these.

$$\mathsf{Prod} \stackrel{\mathrm{def}}{=} \forall \mathrm{X}.(\mathsf{Int} \to \mathsf{Bool} \to \mathrm{X}) \to \mathrm{X}$$

Consider another type, OptInt, for an "optional integer" with the following operations:

$$\begin{aligned} \mathsf{none} &: \mathsf{OptInt} \\ \mathsf{some} &: \mathsf{Int} \to \mathsf{OptInt} \\ \mathsf{isNone} &: \mathsf{OptInt} \to \mathsf{Bool}\,. \end{aligned}$$

The idea is that an OptInt value can be either none or some(n) for any Int value n, with isNone as a test to see which of these it is.

Write a definition in System F for the OptInt type, similar to that given for Prod, and definitions for each of the operations listed. You may assume the types Int, Bool and constants true, false : Bool.

2016/17 Question 2(a)

The following definitions give one answer.

$$\mathsf{OptInt} \stackrel{\mathrm{def}}{=} \forall X\,.\,(\mathsf{Int} \to X) \to X \to X$$

$$\mathsf{none} \stackrel{\mathrm{def}}{=} \Lambda X\,.\,\lambda f{:}(\mathsf{Int}\to X)\,.\,\lambda x{:}X\,.\,x$$

$$\mathsf{some} \stackrel{\mathrm{def}}{=} \lambda n{:}\mathsf{Int}\,.\,\Lambda X\,.\,\lambda f{:}(\mathsf{Int}\to X)\,.\,\lambda x{:}X\,.\,f\,n$$

$$\mathsf{isNone} \stackrel{\mathrm{def}}{=} \lambda y{:}\mathsf{OptInt}\,.\,y\,\mathsf{Bool}\,(\lambda z.\mathsf{false})\,\mathsf{true}$$

There might be some small variations possible here, but I think only by making the lambda-terms more complicated.

2016/17 Question 2(b)

(b) Recent versions of Java provide facilities for programming with *lambda expressions*, *higher-order functions*, and *closures*. For each of these three give a one-sentence explanation of what it is, and briefly suggest an example.

A *lambda-expression* is a function expressed as a first-class value, without needing an associated class or method name. For example, (x->x*x) is Java for a function that takes a number and returns its square.

A *higher-order function* is a function that takes another function as an argument. For example, a filter method that takes a predicate — a function returning bool as a result — and uses it to pick out members of a collection that satisfy the predicate.

A *closure* is an enhancement of a lambda-expression, where the value combines a function body with an environment assigning values to its free variables. For example, closure $\{k \mapsto 5\}$:(x->k*x) represents a function that multiplies its input x by the value k, which is 5.

2016/17 Question 2(c)

(c) Here is a Java class to tabulate the results of a numerical function.

```
1  import java.util.function.IntToDoubleFunction;
2
3  public class Tabulator {
4
5      private int lower, upper;
6
7      public Tabulator(int from, int to) { lower=from; upper=to; }
8
9      public void tabulate(IntToDoubleFunction f) {
10         for (int i=lower; i<=upper; i++)
11             System.out.println("f(" + i + ") = " + f.applyAsDouble(i));
12     }
13 }
```

The tabulate method is a higher-order function. Is it first-order, second-order, or third-order? Explain why.

2016/17 Question 2(c)

(c) Here is a Java class to tabulate the results of a numerical function.

```
1  import java.util.function.IntToDoubleFunction;
2
3  public class Tabulator {
4
5      private int lower, upper;
6
7      public Tabulator(int from, int to) { lower=from; upper=to; }
8
9      public void tabulate(IntToDoubleFunction f) {
10         for (int i=lower; i<=upper; i++)
11             System.out.println("f(" + i + ") = " + f.applyAsDouble(i));
12     }
13 }
```

The tabulate method is a *second-order function*: it takes a first-order function — IntToDoubleFunction f — as an argument.

2016/17 Question 2(d)

(d) The following code attempts to use a Tabulator to calculate for all numbers from 1 to 10 first their squares and then all their powers from 1 to 5.

```
1  Tabulator t = new Tabulator(1,10);
2
3  IntToDoubleFunction square = x -> x*x;
4
5  t.tabulate(square);
6
7  int n=1;
8  IntToDoubleFunction powern = x -> java.lang.Math.pow(x,n);
9
10 for (int k=1; k<=5; k++) { n = k; t.tabulate(powern); }
```

The types in this code are all correct, as are the automatic conversions between **int** and **double** values. Nevertheless, a limitation in Java means that the compiler reports an error and is unable to compile the combination of lambda expressions and higher-order functions used here. What has gone wrong? Explain which code is causing the problem, and what limitation in Java means the compiler returns an error.

2016/17 Question 2(d)

(d) The following code attempts to use a Tabulator to calculate for all numbers from 1 to 10 first their squares and then all their powers from 1 to 5.

```
1  Tabulator t = new Tabulator(1,10);
2
3  IntToDoubleFunction square = x -> x*x;
4
5  t.tabulate(square);
6
7  int n=1;
8  IntToDoubleFunction powern = x -> java.lang.Math.pow(x,n);
9
10 for (int k=1; k<=5; k++) { n = k; t.tabulate(powern); }
```

The problem is with the combination of closures and mutable store, and centres on the call t.tabulate(powern). The powern function mentions n which refers to a local variable. Moreover, the contents of n is changed between successive calls by the assignment n=k within the loop.

2016/17 Question 2(d)

(d) The following code attempts to use a Tabulator to calculate for all numbers from 1 to 10 first their squares and then all their powers from 1 to 5.

```
1  Tabulator t = new Tabulator(1,10);
2
3  IntToDoubleFunction square = x -> x*x;
4
5  t.tabulate(square);
6
7  int n=1;
8  IntToDoubleFunction powern = x -> java.lang.Math.pow(x,n);
9
10 for (int k=1; k<=5; k++) { n = k; t.tabulate(powern); }
```

Java does not permit variables in the environment of a closure that can actually vary: they must be constant (declared **final**) or "effectively final" (only assigned to once, in a way that the compiler can detect). Here the variable n most definitely changes — that's the point of the loop, to tabulate powern for different values of n.

2016/17 Question 3(a)

Programmers J. and K. are writing Java to implement a Counter class. Here is J's proposed code, which compiles and executes successfully.

```
1  public class Counter {
2
3      private int n=0;
4
5      public void up()      { n = n+1; }
6      public void reset()   { n = 0; }
7      public int read()     { return n; }
8  }
```

K. complains that this class is not *thread safe* and may cause problems in concurrent code.

(a) Suppose we have a Counter c with current value n=5. Give an example of how calls to the methods of c from two concurrent threads could lead to incorrect results. Include information about when each call starts and finishes, when n changes, and its final value. Explain briefly why the outcome you describe is incorrect.

2016/17 Question 3(a)

The following sequence of unsynchronized method invocations in two different threads leads to an incorrect outcome.

n	Thread 1	Thread 2
5	c.up() [call starts]	
5	read n as 5	
5		c.reset() [call starts]
0		write 0 to n
0		**return** [call finishes]
0	calculate 5+1 is 6	
6	write 6 to n	
6	**return** [call finishes]	

The outcome with final value n=6 is incorrect because the reset() call has been lost: if it had occurred before the up(), then n would end up as 1; if it occurred after, then n would be 0.

An alternative problematic execution would be two concurrent up() invocations, where the final result was only increased by 1.

2016/17 Question 3(b)

(b) K. recommends using **synchronized** methods. Explain what happens when a synchronized method of an object is invoked, compared with an unsynchronized method.

When a **synchronized** Java method is invoked, the calling thread must first acquire the *lock* associated with the invoked method's object. Each instance of a Java object has its own unique lock. Obtaining the lock may require the thread to wait until some other thread releases the lock. Once the lock is held the method executes as normal; and then the thread releases the lock again.

2016/17 Question 3(c)

(c) J. is worried that using **synchronized** methods can cause "bottlenecks", where code runs much more slowly. Is J. right or wrong? Explain your answer.

J. is correct. When several threads running concurrently on multiple cores all simultaneously attempt to invoke methods on the same object, they each have to wait to take turns holding the lock. This reduces potentially parallel code to serial execution.

2016/17 Question 3(d)

(d) J. and K. discover that it is not always necessary to synchronize all methods in a class. To make Counter thread safe, which methods need to be **synchronized** and which can be left as they are? Explain your choices.

Both the up() and reset() methods need to be **synchronized**; there is however no need to synchronize the read() method.

- up() is not atomic, and must have exclusive access to ensure it writes back the correct value of n.
- reset() will update n atomically, but needs to obtain the lock to ensure that up() is not currently running.
- read() will atomically read n, and does not need synchronization as neither of the other methods places n in any incorrect intermediate state.

Course Enhancement Questionnaires

Please complete the online feedback survey for APL. It's anonymised, and I read every submission. Thanks.

Surveys through MyEd

http://edin.ac/CEQ

Select **Advances in Programming Languages**

Thank You

. . . for your attention, interest and participation in this course.

I wish you all the best in the exam, your degree, and your future enjoyment of programming languages.