Final Exam Review

Scala in general

- What platforms does Scala run on?
- What are the three “pillars” of Scala? And the fourth major feature
  - O-O, functional and statically typed
  - Implicits
- What are the Scalastic principles?
  - Types, Objects, Functions, with Implicits
- What are the Scalastic pragmatics?
  - Usability, Power, Simplicity
- What is Scala good for?
  - Microservices, reactive programming, parallel programming, parsing, general programming.

Structure of a program

- A Scala program/method looks like what?
  - Definitions
  - **Expression**

def sqr(x: Double): Double = x * x

println(*math.sqrt(sqr(3) + sqr(4))*) *[side effect of mutating the program environment but still an expression with no result].*

Functional Programming

- What is a *pure* function?
  - No side effects (no mutation) and
  - output is always same for given input
- What is a side effect?
  - Change of state within the program?
  - Change of state outside the program? (I/O)
  - Abnormal jump or termination (e.g. an exception)
- And why do we care about pure functions?
  - They allow us to “prove” programs to be true.
    - We can prove entire applications to be good.
    - In practice, we don’t
    - Combine testing with proof: that’s good enough.
  - We need pure functions when we use parallelism:
    - because side-effects on remote worker nodes will not help us on the driver node.

Functional Programming (2)

- “Symbolic programming”
  - The compiler is allowed to take short cuts (i.e. evaluate expressions at compile time, rather than leave them until runtime).
  - For example (the most obvious example), de-sugaring is a short cut.
- Lazy (deferred) evaluation
  - Can significantly reduce the amount of work we have to do;
  - Is more mathematical:
    - We can define infinite series
      - Val fibonacci: LazyList = …. fibonacci.scanLeft “terminating” at least for finite number of elements.
      - Val x = x. “non-terminating"

Functional Programming (3)

- Functional programs are expressions
- Higher-order functions and functional composition
  - Output of which is another function or:
  - Input of which is another function.
- Lambda (mathematical concept based on free and bound variables). aka as a “function literal” or “anonymous function”
- Recursion as the primary re-use mechanism (vs. iteration)
  - def factorial(x: Int): Int = if (x<=1) 1 else x * factorial(x-1)

Functional Programming (4)

- Pattern matching
- What did I say about pattern matching?
- What is it the opposite of?
- Expression
- Pattern matching is based on the concept of *un-*expression.

Functional Programming (5)

- Ability to construct types
- I don’t mean the ability to define classes.
- I mean *Option[Int]*.
- In Java Optional<Int>.

- Can you think of any way to construct a new type based on existing types that doesn’t use some name???
  - Val x = 1; val y = math.Pi; val z = “Hello World!”
  - Val q = (x, y, z)
  - Newton’s Approximation: pass in max tries; initial guess; function; first derivative. What we get back is: *either the solution*; or a *reason why it failed*.
  - (x, “reason”). Either[Double, String]. Try[Double]

Language types

- All the normal stuff that you have in, say, Java.
- plus what other kind of type is *built-in* to Scala but not Java?
  - Tuples
- Feature of functional programming:
  - Curried functions
    - f(x, y, z) “tupled” form
    - g(x)(y)(z) “curried” form (same result provided that *g = f.curried*)
    - val x = 1, y = 2
    - val h1 = *f(x, y, _)* and val h2 = *g(x)(y)* are the same thing!
    - h1(3) == h2(3)

Monads and all that jazz

- Do you think monads are a bit scary?
  - They’re really a pretty simple concept—just a wrapper around something.
  - That wrapper can have various cardinalities:
    - 0 or 1: *Option, Future** or *Try**
    - 1: *Either* (actually, *Either* isn’t strictly a monad)
    - 0 to N: *List*, *LazyList*, etc.
- So, if these things are wrappers, how do we unwrap them?
  - Well, the idea is that we can do things to the content of the monad without actually losing the wrapper aspect.
  - The most flexible and important construct for doing this is the for-comprehension

* *actually,* Future *kind of has three possibilities but always 0 or 1*
* *With Try, we always have 1 value (but it might be an exception)*

Monads: for-comprehensions

- Suppose you have two instances of *Option[T]*, and you have a function *f: (T1, T2) => R*;
  - Then you can do something like:
    - *val t1o: Option[T1] = Some(???), t2o: Option[T2] = Some(???)*
    - *val r: Option[R] = for (t1 <- t1o; t2 <- t2o) yield f(t1, t2)*
  - Recall that this is the same as “map2,” a method we talked about but don’t actually need. It’s also our *map3*, *map4*, etc.
  - Realize that, as far as most of your code is concerned, the *t1o*, *t20* and the result of the for-comprehension are all still “wrapped.”
  - We only actually unwrapped these monads in the context of the for-comprehension.

Monads: for-comprehensions (2)

- Monads can also be thought of as a means to sequencing, something which generally isn't important in functional programming.
  - *for (t1 <- t1o; t2 <- t2o) yield f(t1, t2)*
- In this example, we forced t1o to be unwrapped before t2o. Indeed, we might have a for-comprehension that looks like this:
  - *for (t1 <- t1o; t2 <- g(t1)) yield f(t1, t2)*
- Here, it's clear that we can't get a value for *t2* until we have a value for *t1*.

Variance

- Liskov Substitution Principle (assignment)
  - *val x: X = ???*
  - *val y: Y = ???*
  - Assign the value of *y* to variable *x* (e.g. *val x = y*)
    - *val z = x.q*
    - Provided that *X* is a super-class of *Y*
    - So, *Y* is a sub-class of *X*, which implies that:
      - *Y* has all of the properties of *X*, and maybe others.

Variance (2)

- Liskov Substitution Principle (yield or return)

def f(z: Z): Y = z

val x: X = f(z)

val z: Z = ???

  - This all works OK provided that *X* is a super-class of *Y*
  - And *Y* is a super-class of *Z*.

Covariance

- X is a super-class of Y

```
def method(xs: List[X], f: X=>Z): List[Z] {
   for (x <- xs) yield f(x)
}
```

```
val q: List[Y] = List(….)
method(q)
```

```
class List[+X] { …. }
```

We need List[X] to be a sub-class of List[Y] which will be true if X is a sub-class of Y.

```
def get(i: Int): X = ???
```

Contravariance

- If a type is defined thus: *X[-T]*, i.e. where *T* is *contravariant*, then a value of *X[U]* can substitute for a *X[T]*, provided *U* is a *super*-type of *T*.
- The typical usage of contra variance is in a function type, such as:

```
trait Function1[-T, +R] {
  def apply(t: T): R
}
```

Invariance

- X can’t be both covariant and contravariant

```
class List[+X] {
    def ::[Y >: X](y: Y): List[Y] = ??? (but we include Y = X)
    def apply(i: Int): X = ???
}
```

Arrays and ArrayBuffer are invariant in their underlying type.

But the :: method of List gets around the covariance of underlying type by the super-type trick so the following works:

val dogs = List[Dog]

val cat = Cat(“squeaky”)

val animals: List[Animal] = cat :: dogs

Dealing with this super type thing

```
scala> trait Animal
defined trait Animal
scala> case class Dog(name: String) extends Animal
defined class Dog
scala> val dogs = List(Dog("Bentley"))
dogs: List[Dog] = List(Dog(Bentley))
scala> case class Cat(name: String) extends Animal
defined class Cat
scala> val cat = Cat("squeaky")
cat: Cat = Cat(squeaky)
scala> val cat: Cat = Cat("squeaky")
cat: Cat = Cat(squeaky)
scala> cat :: dogs
res2: List[Product with Animal with java.io.Serializable] = List(Cat(squeaky),
Dog(Bentley))
scala> val animals: List[Animal] = cat :: dogs
animals: List[Animal] = List(Cat(squeaky), Dog(Bentley))
scala> val dog = Dog("Xena")
dog: Dog = Dog(Xena)
scala> val dogs2 = dog :: dogs
dogs2: List[Dog] = List(Dog(Xena), Dog(Bentley))
scala> animals(0) match { case dog: Dog => dog.name; case cat: Cat => cat.name; case _ =>
"no name" }
res11: String = squeaky
```

Note: when we did *cat::dogs*, the compiler found all the common super-types