Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 14, 2017

Homework 1

- Please submit your homework via the SVN / `turnin` system, in a folder named `hw_1`
- The specific files to submit are defined in the description for each assignments
- For each section, please turn in only your final program resulting from completion of the section
- Think about overflow!

Please Restrict Your Homework Submission to Features Covered in Class

Current Core Scala Features

- `object`
- `case class`
- `val`
- `if` / `else`
- `match` / `case`
- `require`, `ensuring`
- `Int`, `Double`, `String`
- `Array`, Tuples
- Arithmetic operators
- (In)equality operators
- Logical and / or
- `assertEquals` etc.
- λ-expressions (`ensuring`)
- Plus the stuff from today!

Please Restrict Your Homework Submission to Features Covered in Class

These should be the only import statements you need:

```
import junit.framework.TestCase

import junit.framework.Assert._
```

(or equivalent imports auto-generated by your IDE for your JUnit test class)

Type Checking

Type Checking

- So far, we have been rigorous about computation of programs, but we have relied on intuition for static type checking

- We can provide a *static semantics* for Core Scala along with our *dynamic semantics*

The Substitution Model of Type Checking

- To type check a value **v**, replace **v** with its value type

```
1.003 ⇒ Double
```

- To type check a constant **c**, reduce the defining expression of **c** to a static type **T**, then replace all occurrences of **c** with **T**

```
pi = 3.14 ⇒

pi : Double

pi * radius * radius ⇒

Double * radius * radius
```

The Substitution Model of Type Checking

- To type check a function definition:
    - Type check the body of the definition, replacing all occurrences of each parameter with the corresponding parameter type
- To type check the occurrence of a function name:
    - Replace the name with an *arrow type*, where the parameter types of the function are to the left of the arrow and the return type is to the right

```
square(x: Double): Double = x * x

square(3.14) ⇒

(Double → Double)(3.14)
```

The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types match the corresponding parameter types, reduce the application to the return type

The Substitution Model of Type Checking

```
square(3.14) ⇒

(Double → Double)(3.14) ⇒

(Double → Double)(Double) ⇒

Double
```

Methods and Operators

Syntactic Sugar For Binary Methods

- We refer to methods that take one parameter (in addition to the receiver) as *binary methods*

```
case class Coordinate(x: Int, y: Int) {
  def magnitude() = x*x + y*y

  def add(that: Coordinate) =
     Coordinate(x + that.x, y + that.y)
}
```

Syntactic Sugar For Binary Methods

```
Coordinate(1,2).add(Coordinate(3,4))
↦
Coordinate(4,6)
```

Syntactic Sugar For Binary Methods

- We can elide the dot in method calls on binary methods
- We can also elide the enclosing parentheses around the sole argument

Syntactic Sugar For Binary Methods

```
Coordinate(1,2) add Coordinate(3,4)
↦
Coordinate(4,6)
```

Operator Symbols

- Scala allows the use of operator symbols in method names
- In fact, operators are simply methods in Scala

```
1 + 2 → 3

1.+(2) → 3
```

Coordinate Custom +

```scala
case class Coordinate(x: Int, y: Int) {
  def magnitude() = x*x + y*y

  def +(that: Coordinate) =
     Coordinate(x + that.x, y + that.y)
}
```

Coordinate Custom +

```
Coordinate(1,2) + Coordinate(3,4)
↦
Coordinate(4,6)
```

Requires Clauses on Class Constructors

```
case class Name(field1: Type1, …, fieldN: TypeN) {
    require (boolean-expression)
```

- Checked on every constructor call
- Because case class instances are immutable, this ensures the property holds for the lifetime of an instance

Equals on Case Classes

- The equals method on a case class instance checks for structural equality with its argument:

```
Rational(4,6).equals(Rational(4,6)) ↦

                true
```

Equals on Case Classes

- Note that equals is a binary method, and so we can also write this expression as:

```
Rational(4,6) equals Rational(4,6) ↦

true
```

Equals on Case Classes

- Of course, the built in equals method does not check for mathematical equality:

```
Rational(4,6) equals Rational(2,3) ↦

false
```

Equals on Case Classes

- Why is this definition of equality acceptable on case classes?
- What other definition is available to us?

```
Rational(4,6) equals Rational(2,3) ↦

false
```

Short-Circuiting And and Or Operators

- Just as we have defined a short-circuiting if-then-else operator, we can define short-circuiting and/or operators:

$$\&\& \qquad\qquad ||$$

  - How do we define the static and dynamic semantics of these operators?

  - When are they useful?

Calling and Defining Parameterless Methods Without Parentheses

```
def toString() = { … }
```

vs.

```
def toString = { … }
```

Calling and Defining Parameterless Methods Without Parentheses

```
Rational(4,6).toString()
```

vs.

```
Rational(4,6).toString
```

The Uniform Access Principle

- Client code should not be affected by whether an attribute is defined as a field or a method
  - Only applies to *pure* (side-effect free) methods
  - Can be strange even for some pure methods (what are some examples?)

Abstract Datatypes

Abstract Datatypes

- Often, we wish to abstract over a collection of compound datatypes that share common properties

- For example, we might wish to define an abstract datatype for shapes, with separate case classes for each of several shapes

- For this purpose, we define an *abstract class* and use *subclassing*

Abstract Datatypes

```
abstract class Shape
case class Circle(radius: Double) extends Shape
case class Square(side: Double) extends Shape
case class Rectangle(height: Double, width: Double) extends Shape
```

Recall Our Design Recipe

- **Analysis**: What are the objects in the problem domain? What data types we will use to represent them?

- **Contract**: What is name of our functions and their parameters? What are the requirements of the data they consume and produce? What is the meaning of what our program computes?

- **Repeat** until we are confident in our program's correctness

    - Write some **tests**

    - Sketch a function **template**

    - **Define** the function

Recall Our Design Recipe

- **Analysis**: This is the stage where we would discover we wish to model our problem domain with functions over an abstract datatype
- **Contract**: What contract holds for each function? Do additional constraints and assurances hold for specific subclasses?
- **Repeat** until we are confident in our program's correctness
    - Write some **tests:** Same as before
    - Sketch a function **template**: This needs re-examination
    - **Define** the function

The Design Recipe for Abstract Datatypes

- Our Function Template for computing with abstract datatypes depends on answering the following questions:
  - Do I expect to eventually add more subclasses?
  - Do I expect to eventually add more functions?

Case 1
We Expect Few New Functions But Many New Variants

Case 1: We Expect Few New Functions But Many New Variants

- This is a case that object-oriented programming handles well
- Classic example domains: GUI Programming, Productivity Apps, Graphics, Games
- Declare an abstract method in our superclass and provide a concrete definition for each sub-class

a.k.a.,

The Union Pattern (for the datatype definitions)

The Template Method Pattern (for the function definitions)

Abstract Datatypes

```scala
abstract class Shape {
  def area: Double
}
```

Abstract Datatypes

```
case class Circle(radius: Double) extends Shape {
  val pi = 3.14

  def area = pi * radius * radius

}
```

Abstract Datatypes

```
case class Square(side: Double) extends Shape {
  def area = side * side
}
```

Abstract Datatypes

```
case class Rectangle(length: Double, width: Double)
    extends Shape {

  def area = length * width

}
```

How Do Abstract Classes Affect Our Type Checking Rules?

- When type checking a class definition, ensure that all abstract methods declared in the superclass are actually defined, with *compatible* method types

- When type checking a collection of class definitions, ensure that there are no cycles in the class hierarchy!

How Do Abstract Classes Affect Our Type Checking Rules?

- If a method is called on a receiver whose static type is an abstract class, extract an arrow type from the declaration (just as with a definition in a concrete class)

```
expr.area ↦

Shape.area ↦

() → Double
```

Type Checking Arguments to a Method Call

- The static types of an argument might no longer be an exact match:

```
abstract class Shape {
  def area: Double

  def makeLikeMe(that: Shape): Shape
}
```

(Let us set aside the concrete definitions of `makeLikeMe` for awhile)

Now Consider a Call to Matcher With Concrete Types

```
Circle(1).makeLikeMe(Circle(2)) ⇒

Circle.makeLikeMe(Circle) ⇒

(Shape → Shape)(Circle)
```

And now we are stuck…

Recall The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types match the corresponding parameter types, reduce the application to the return type

Subtyping

- We need to widen our definition of *matching* a type to include subtyping

- A class is a subtype of the class it extends

- Subtyping is Reflexive:

`A <: A`

- Subtyping is Transitive:

If `A <: B` and `B <: C` then `A <: C`

Subtyping

- All types are a subtype of type `Any`
- Type `Nothing` is a subtype of all types
    - There is no value with value type `Nothing`

Recall The Substitution Model of Type Checking

- To type check the application of a function to arguments:
  - Reduce the function to an arrow type
  - Reduce the arguments, left to right, to static types
  - If the argument types **are subtypes of** the corresponding parameter types, reduce the application to the return type

Applying a Class Method Revisited

- To reduce the application of a method:

```
C(v1, …, vk).m(arg1, …, argN)
```

  - Reduce the receiver and arguments, left to right
  - Reduce the body of m, replacing constructor parameters with constructor arguments and method parameters with method arguments

Applying a Class Method Revisited

- To reduce the application of a method:

```
C(v1, …, vk).m(arg1, …, argN)
```

  - Reduce the receiver and arguments, left to right
  - **Find the body of m in C and reduce to that,** replacing constructor parameters with constructor arguments and method parameters with method arguments

The Body of *m*

- To find the body of method `m` in type `C`:
    - Find the definition of m in the body of `C`, if it exists
    - Otherwise, find the body of `m` in the immediate superclass of `C`