Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sagnak Tasirlar, Two Sigma Investments
Homework 1

• Please submit your homework via the turnin system, in a folder named hw_1

• The specific files to submit are defined in the 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
Please Restrict Your Homework Submission to Features Covered in Class

• These should be the only import statements in your program:

    import junit.framework.TestCase

    import junit.framework.Assert._
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

\[
1.003 \Rightarrow \text{Double}
\]

• To type check a constant \( c \), reduce the defining expression of \( c \) to a static type \( T \), then replace all occurrences of \( c \) with \( T \)

\[
\text{pi} = 3.14 \Rightarrow
\]

\[
\text{pi} : \text{Double}
\]

\[
\text{pi} \times \text{radius} \times \text{radius} \Rightarrow
\]

\[
\text{Double} \times \text{radius} \times \text{radius}
\]
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

\[
\text{square}(x: \text{Double}): \text{Double} = x \times x
\]

\[
\text{square}(3.14) \Rightarrow
\]

\[
(\text{Double} \rightarrow \text{Double})(3.14)
\]
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

\[
\text{square}(3.14) \Rightarrow
\]

\[\text{(Double} \to \text{Double})(3.14) \Rightarrow\]

\[\text{(Double} \to \text{Double})(\text{Double}) \Rightarrow\]

\[\text{Double}\]
Methods and Operators
Syntactic Sugar For Binary Methods

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

```scala
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

• With binary methods, we can elide the dot in a method call

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

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

\[
\text{Coordinate}(1,2) + \text{Coordinate}(3,4) \rightarrow \text{Coordinate}(4,6)
\]
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:

\[
\text{Rational}(4, 6).\text{equals}(\text{Rational}(4, 6)) \rightarrow \text{true}
\]
Equals on Case Classes

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

\[
\text{Rational}(4,6) \text{ equals Rational}(4,6) \mapsto \text{true}
\]
Equals on Case Classes

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

    Rational(4,6) equals Rational(2,3) \rightarrow 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:

```
&&     ||
```

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

• When are they useful?
Calling and Defining Parameterless Methods Without Parentheses

```scala
def toString() = { ... }
```

vs.

```scala
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 immutable methods
  - Can be strange even for some immutable 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 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

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)

\[
\begin{align*}
\text{expr} & . \text{area}() \mapsto \\
\text{Shape} & . \text{area}() \mapsto \\
() & \rightarrow \text{Double}
\end{align*}
\]
Type Checking Arguments to a Method Call

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

```java
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 \texttt{Any}
- Type \texttt{Nothing} is a subtype of all types
  - There is no value with value type \texttt{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:

\[ C(v_1, \ldots, v_k).m(\text{arg}_1, \ldots, \text{arg}_N) \]

• 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:

\[ C(v_1, \ldots, v_k).m(arg_1, \ldots, arg_N) \]

• 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 $C$