Comp 311
Functional Programming

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

August 30, 2018

Announcements

- Homework 0 is “due” next Thursday
  (ensure you can submit your homework without issue!)

- Homework 1 will also be assigned next Thursday

What are The Exceptional Events in Core Scala?

- A “division by zero” error on Ints (but not Doubles)
- We run out of some finite resource
  - The computation never stops
  - The computation uses too much memory

Programming With Intention

Programming With Intention

- There is far too much broken software in the world...
- The number of mission critical domains affected by programming is increasing
    - Space exploration and satellites, defense, medical devices, automobiles, finance

Programming With Intention

- Static types help us reduce some errors by restricting the potential results of a computation
- We still need to defend against exceptional events
- And we need to defend against silent errors
  - *Silent errors are actually our most insidious risk*

Defending Against Exceptional Conditions

- With division on `Int`s, we should ensure that the divisor is non-zero
- We will return to guarding against exhaustion of finite resources later
    - For now, assume we have sufficient resources, provided that our time and space requirements have *some* bound

Defending Against Unbounded Resource Consumption and Silent Failures

- We've discussed some of the caveats when programming with `Int`s and `Double`s
- To further defend against such errors, we will make use of a *design recipe*

The Design Recipe

The Design Recipe

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

- **Contract**: What are the names 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

    1. Write some **tests** (start with example inputs/outputs)

    2. Sketch a function **template**

    3. **Define** the function

Example: Calculating Profit for a Movie Theater

(Problem Statement from "How to Design Programs" 2001)

- The owner of a movie theater collected the following data:
  - At $5.00 per ticket, 120 people attend a performance
  - Decreasing by $0.10 increases attendance by 15 people
  - A performance costs $180 plus $0.04 per attendee
  - Define a function to calculate the exact relationship between ticket price and profit

Analysis

- We are working with monetary values and counts of attendees
- Attendees are whole numbers
- To avoid rounding errors, we will use `Int`s for monetary values
- Therefore all monetary values will be represented in cents

Analysis

- We need to compute *profit*
- Profit is calculated as *revenue - cost*
- Cost is dependent on attendance

Contracts

- First, define a **contract** for our function:
  - What is the name of the function?
    - What considerations should go into the names we choose?
  - What are the static types of the arguments that our function consumes?
    - What other constraints must hold on the values it consumes?
  - What is the static type of its result?
    - What else does it ensure about its result?

Contract for Attendance

```
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  …
} ensuring(result => result >= 0)
```

Syntax and Typing of Contracts

```
def fnName(arg₀: Type₀, …, argₖ: Typeₖ): ReturnType = {
  require(expr)
  expr
} ensuring (expr)
```

- The static type of the *expr* in *require(expr)* is Boolean
- The static type of the *expr* in *ensuring(expr)* is

  ReturnType $\Longrightarrow$ Boolean

Unary Lambda Expressions for the Ensuring Clause

```
result => result == 1

result => 0.0 < result & result < 1.0

result => 0 > result | result > 10
```

- *`result`* `=>` … indicates a unary function of *result.*
- The static type of the argument *result* is inferred from the context of the *ensuring* clause; i.e., it's the ReturnType of the corresponding function's body expression.
- The lambda expression body must return a Boolean.

More Complex Contracts

```
def fnName(arg_0: Type_0, …, arg_k: Type_k): ReturnType = {
      require(expr_precondition_0)
      require(expr_precondition_1)
      require(expr_precondition_k)
      expr_function_body
} ensuring(expr_postcondition_0)
. ensuring(expr_postcondition_1)
. ensuring(expr_postcondition_k)
```

Statement of Purpose

- Use a comment to provide a brief statement of the meaning of the function

- Well chosen names for functions and parameters are often some of the best documentation!

Statement of Purpose for Attendance

```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  ...
} ensuring(result => result >= 0)
```

Write Some Test Examples

```
120 == attendance(500)
```

- We can think of tests as constraint equations in algebra
- The program we are constructing is a solution to these constraints

Sketch a Function Template

```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  an algebraic expression
} ensuring(result => result >= 0)
```

Defining Functions

- **Design Principle: "Keep It Simple, Stupid"**

- Given the tests we've written so far and the template we've sketched, write the simplest solution that passes those tests

- Keeping the definition simple will:

  - Force us to include adequate test coverage

  - Help to keep us from over-engineering

Define The Function

```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 */
def attendance(ticketPrice: Int): Int = {
    require(ticketPrice >= 0)
    120
} ensuring(result => result >= 0)
```

We Need More Tests

```
120 == attendance(500)
135 == attendance(490)
```

Redefinition (Attempt 1)

```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  120 + (500 - ticketPrice) * (15 / 10)
} ensuring(result => result >= 0)
```

But Now Some Tests Fail

```
120 == attendance(500)
135 == attendance(490)
```

Division With Ints

```
attendance(490) ↦

120 + (500 - 490) * (15 / 10) ↦

120 + 10 * (15 / 10) ↦

120 + 10 * (15 / 10) ↦

120 + 10 * 1 ↦

120 + 10 ↦

130
```

Redefinition (Attempt 2)

```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  120 + ((500 - ticketPrice) * 3) / 2
} ensuring(result => result >= 0)
```

Now Our Two Tests Succeed

```
120 == attendance(500)
135 == attendance(490)
```

Let's Add Harder Tests

```
120 == attendance(500)
135 == attendance(490)
0 == attendance(1000)
```

Now our `ensuring` clause fails!

Redefinition (Attempt 3)

```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance
 */
def attendance(ticketPrice: Int): Int = {
  require (ticketPrice >= 0)
  max(0, 120 + ((500 - ticketPrice) * 3) / 2)
} ensuring(result => result >= 0)
```

(To Do: Apply Our Design Recipe to max)

```
def max(m: Int, n: Int) = if (m >= n) m else n
```

Now All Tests Pass

```
120 == attendance(500)
135 == attendance(490)
0 == attendance(1000)
```

Let's Add More Tests

```
120 == attendance(500)
135 == attendance(490)
0 == attendance(1000)
0 == attendance(Int.MaxValue)
```

Overflow Does Not Appear To Be a Problem...

```
120 == attendance(500)
135 == attendance(490)
0 == attendance(1000)
0 == attendance(Int.MaxValue)
```

Or Does It...

```
attendance(2147483647) ↦

max(0, 120 + ((500 - 2147483647) * 3) / 2) ↦

max(0, 120 + (-2147483147 * 3) / 2) ↦

max(0, 120 + -2147482145 / 2) ↦

max(0, 120 + -1073741072) ↦

max(0, -1073740952) ↦

if (0 >= -1073740952) 0 else -1073740952 ↦

0
```

Bounding Cost of Attendance

- We can determine an exact bound for the maximum allowable parameter to `attendance`:
  - For each subexpression, solve for the parameter values that would result in overflow:

```
(500 - ticketPrice) > Int.MaxValue

(500 - ticketPrice) < Int.MinValue

etc.
```

Bounding Values Based on Domain Knowledge

- We can also find appropriate bounds by considering the range of values required by our problem domain
    - Often, these bounds will be much tighter
- In our example, we can see from our formula that attendance is zero whenever the cost of a ticket is $5.80 or above
- We can also see that even free tickets achieve attendance of only 870 people
    - And it is likely that our theater cannot seat 870 people!

Bounding Cost of Attendance

```
def attendance(ticketPrice: Int): Int = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  max(0, 120 + ((500 - ticketPrice) * 3) / 2)
} ensuring(result => result >= 0)
```

Now We Should Remove Our Test on Int.MaxValue

```
120 == attendance(500)
135 == attendance(490)
0 == attendance(1000)
0 == attendance(Int.MaxValue)
```

Add Let's Add Some More Tests While We're At It

```
120 == attendance(500)
135 == attendance(490)
0 == attendance(1000)
0 == attendance(580)
2 == attendance(579)
870 == attendance(0)
```

Now We Can Apply the Design Recipe to Our Remaining Functions

```
/**
 * Returns cost to the theater of showing a film,
 * as a function of ticketPrice.
 */
def cost(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  18000 + 4 * attendance(ticketPrice)
} ensuring(result => result > 0)
```

Now We Can Apply the Design Recipe to Our Remaining Functions

```
/**
 * Returns revenue received by the theater when
 * showing a film, as a function of ticket price.
 */
def revenue(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  ticketPrice * attendance(ticketPrice)
} ensuring(result => result >= 0)
```

What Should Be The Ensuring Clause on Profit?

```
/**
 * Returns profit enjoyed by the theater after showing
 * a film, defined as the difference between revenue
 * costs.
 */
def profit(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  revenue(ticketPrice) - cost(ticketPrice)
}
```

Following The Design Recipe includes writing tests on all of our newly defined functions

```
35130 = profit(510)
 -21480 = profit(0)
-18000 = profit(1000)
…
0 = revenue(0)
0 = revenue(1000)
53550 = revenue(510)
…
18420 = cost(510)
21480 = cost(0)
18000 = cost(1000)
…
```

Can’t Forget About Max!

```
Int.MaxValue == max(0, Int.MaxValue)
        0 == max(-1, 0)
        1 == max(-1, 1)
  0 == max(0, Int.MinValue)
  0 == max(Int.MinValue, 0)
                ...
```

How Many Helper Functions Should We Include?

As a guideline:

- Include a helper function for each of the dependencies mentioned in your problem statement

- Include a helper function for new dependencies discovered during testing

Inlining Into One Large Function Makes Code Far Less Readable

```
def profit(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)

  ticketPrice * max(0, 120 + ((500 - ticketPrice) * 3) / 2) -
    18000 + 4 * max(0, 120 + ((500 - ticketPrice) * 3) / 2)
}
```