Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
The Nature of Doubles
Scientific Notation

- Numeric values in scientific computations can span enormous ranges, from the very large to the very small
- At the same time, scientific measurements are of limited precision
- “Scientific notation” was devised in order to efficiently represent approximate values that span a large range
Scientific Notation

$6.022 \times 10^{23}$

mantissa \hspace{1cm} exponent
Scientific Notation and Efficient Computation

• We normalize the mantissa so that its value is at least 1 but less than 10

• If we

  • Set the number of digits in the mantissa to a fixed precision, and
  • Set the number of digits in the exponent to a fixed precision

• Then all numbers in our notation are of a fixed size
Doubles

• Values of type Double are stored as with fixed sized numbers in scientific notation, but with a few differences:

• Finite, nonzero numeric values can be expressed in the form:

$$\pm m \ 2^e$$
Doubles

\[ \pm m \ 2^e \]

- \[ 1 \leq m \leq 2^{53} - 1 \]
- \[ -2^{10} - 53 + 3 \leq e \leq 2^{10} - 53 \]
Doubles

\[ \pm m \ 2^e \]

- \[ 1 \leq m \leq 2^{53}-1 \]
- \[ -2^{10}-53+3 \leq e \leq 2^{10}-53 \]
- \[ -1074 \leq e \leq 971 \]
Representations of Doubles

- Many quantities have more than one representation in this format:

  \[ 1024 \times 2^{500} \]

  \[ 512 \times 2^{501} \]
Distances Between Doubles

- The distance between adjacent values of type Double is not constant
- The values are most dense near zero
- They grow sparser exponentially as one moves away from zero
Operations and Rounding

• Arithmetic operations round to the closest representable value

• Ties are broken by choosing the value with the smaller absolute value

• We can think of each value of type Double as denoting the range of real numbers that are closest to it
Overflow with Doubles

- Computations on Doubles that result in values larger than the largest finite Double are represented with special values:

  Double.PositiveInfinity

  Double.NegativeInfinity
Underflow with Doubles

- Computations on Doubles that result in values with magnitudes smaller than the smallest non-zero Double are represented with special values:
  
  0.0  -0.0
Division By Zero

- Division of a non-zero finite value by a zero value results in an infinite value:

  \[ 1.0 / 0.0 \mapsto \text{Double.PositiveInfinity} \]
  \[ 1.0 / -0.0 \mapsto \text{Double.NegativeInfinity} \]
Division By Zero

- As does division of an infinite value by a zero value:

  \[
  \text{Double.PositiveInfinity} / 0.0 \mapsto \text{Double.PositiveInfinity}
  \]
Division By Zero

- Division of a zero value by a zero value results in another special value NaN (for “Not a Number”):
  - $0.0 / 0.0 \mapsto \text{Double.NaN}$
  - $-0.0 / 0.0 \mapsto \text{Double.NaN}$
Doubles Break Common Algebraic Properties

• Addition is not associative:

\[(0.1 + 0.2) + 0.3 \mapsto 0.6000000000000001\]

\[0.1 + (0.2 + 0.3) \mapsto 0.6\]
Doubles Break Common Algebraic Properties

• Equality is not reflexive:
  `Double.NaN != Double.NaN`

• Multiplication does not distribute over addition:
  
  \[
  100.0 \times (0.1 + 0.2) \mapsto 30.000000000000004 \\
  100.0 \times 0.1 + 100.0 \times 0.2 \mapsto 30.0
  \]
Morals of Floating Point Computation

• Avoid floating point computation whenever you need to compute precise numeric values (such as monetary values)

• Use floating point values only when calculating with inexact measurements over a range larger than can be represented with precise arithmetic
Morals of Floating Point Computation

• Try to bound the margin of error in your calculation

• Don’t test for equality directly
  • Instead of writing:
    \[ x \ == \ y \]
  • Write:
    \[ \text{abs}(x - y) \ <= \ \text{tolerance} \]
Defining Absolute Value

def abs(x: Double) = if (x >= 0) x else -x
Computing Conditional Expressions

• We used a slight of hand when presenting if expressions

    if (e1) e2 else e3

• According to the substitution model of computation, how do we compute the value of this expression?
Computing Conditional Expressions

if (e1) e2 else e3

• First we compute $e_1 \mapsto v_1$, then $e_2 \mapsto v_2$, then $e_3 \mapsto v_3$

• If $v_1$ is true then reduce to $v_2$

• Otherwise reduce to $v_3$
But Consider the Following Expression

if (false) 1/0 else 3

This expression should reduce to 3
New Rule for Conditional Expressions

• To reduce an if expression:
  • Reduce the **test** clause
    • If the test clause reduces to **true**, reduce the **then** clause
    • Otherwise, reduce the **else** clause
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 keeps getting larger
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 \texttt{Ints}, 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 \texttt{Ints} and \texttt{Doubles}

- To further defend against such errors, we will make use of a \textit{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 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
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 \textit{Ints} 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
• 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 (_ >= 0)
Syntax and Typing of Contracts

def fnName(arg0: type0, ..., argk: typek):returnType = {
    require(expr)

    expr

} ensuring (expr)

The static types of the require and ensuring clauses must be of type Boolean
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 (_ >= 0)
Write Some Tests

\[ 120 == \text{attendance}(500) \]

- We can think of tests as constraint equations in algebra
- The program we are constructing is a solution to these constraints
**
* 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 (_ >= 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 (_ >= 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 (_ >= 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 (_ >= 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 (_ >= 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…

\[
\text{attendance}(2147483647) \mapsto \\
\max(0, 120 + ((500 - 2147483647) \times 3) / 2) \mapsto \\
\max(0, 120 + (-2147483147 \times 3) / 2) \mapsto \\
\max(0, 120 + -2147482145 / 2) \mapsto \\
\max(0, 120 + -1073741072) \mapsto \\
\max(0, -1073740952) \mapsto \\
\text{if } (0 \geq -1073740952) 0 \text{ else } -1073740952 \mapsto \\
0
\]
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:

\[(500 - \text{ticketPrice}) > \text{Int.MaxValue}\]
\[(500 - \text{ticketPrice}) < \text{Int.MinValue}\]

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

```scala
/**
 * Returns cost to the theater of showing a film, 
 * as a function of ticketPrice.
 */

def cost(ticketPrice: Int) = {
  require (ticketPrice >= 0 & ticketPrice <= 1000)
  18000 + 4 * attendance(ticketPrice)
}

ensuring (_ >= 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 (ticketPrice >= 0 & ticketPrice <= 1000)
  ticketPrice * attendance(ticketPrice)
} ensuring (_ >= 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 (ticketPrice >= 0 & ticketPrice <= 1000)
  revenue(ticketPrice) - cost(ticketPrice)
}
Following The Design Recipe includes writing tests on all of our newly defined functions

\[
35130 = \text{profit}(510) \\
-21480 = \text{profit}(0) \\
-18000 = \text{profit}(1000) \\
\ldots \\
0 = \text{revenue}(0) \\
0 = \text{revenue}(1000) \\
53550 = \text{revenue}(510) \\
\ldots \\
18420 = \text{cost}(510) \\
21480 = \text{cost}(0) \\
18000 = \text{cost}(1000) \\
\ldots
\]
And We Haven’t Forgot About Max!

\[
\begin{align*}
\text{Int.MaxValue} &= \max(0, \text{Int.MaxValue}) \\
0 &= \max(-1, 0) \\
1 &= \max(-1, 1) \\
0 &= \max(0, \text{Int.MinValue}) \\
0 &= \max(\text{Int.MinValue}, 0) \\
&\vdots
\end{align*}
\]
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 (ticketPrice >= 0 & ticketPrice <= 1000)

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

Including Constant Definitions

• We can include constant definitions in functions using `val`

• We refer to expressions prefixed with a sequence of constant definitions as compound expressions
def cost(ticketPrice: Int) = {
    require (ticketPrice >= 0 & ticketPrice <= 1000)

    val fixedCost = 18000
    val perAttendeeCost = 4

    fixedCost + perAttendeeCost * attendance(ticketPrice)
} ensuring (_ >= 0)
To Reduce A Compound Expression

- First compute the value of each constant definition, top to bottom
- Then reduce the result expression, replacing each occurrence of a constant name with its computed value