Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert "Corky" Cartwright, Rice University

September 3, 2019

Homework 0

- Please follow these instructions for checking out your **turnin** repository as soon as possible:
  - Follow the instructions under Homework Submission Guide at the Course Website
  - Submit a **hw_0** folder with a single file **HelloWorld.txt** and a single line of text, **Hello, world!**
  - This submission is not for credit
  - We will let you know if we have not received your submission
  - You will be responsible for successfully submitting your **hw_1** assignment using **turnin**
  - Please bring problems to our attention as soon as possible

Value Types in Core Scala

Int: -3, -2, -1, 0, 1, 2, 3

Double: 1.414, 2.718, 3.14, ∞

Boolean: false, true

String: “Hello, world!”

The Nature of Ints

Fixed Size Ints

- Unlike the integers we might write on a sheet of paper, the values of type Int are of a fixed size.
- For every n: Int,

$$-2^{31} \leq n \leq 2^{31} - 1$$

Fixing the Size of Numbers Has Many Benefits

- The time needed to compute the application of an operation on two numbers is bounded.
- The space needed to store a number is bounded.
- We can easily reuse the space used for one number to store another.

But We Need to Concern Ourselves with Overflow

- If we compute a value larger than $2^{31} - 1$, our representation will "wrap around" (i.e., overflow):

$$2147483647 + 1 \mapsto -2147483648$$

The Moral of Computing with Ints

- If possible, determine the range of potential results of a computation
    - Ensure that this range is no larger than the range of representable values of type Int
- Otherwise, include in your computation a check for overflow

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

$$\underbrace{6.022}_{\text{mantissa}} \times \underbrace{10}_{\text{base}}{}^{\overbrace{23}^{}}$$

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:

$$\pm m \times 2^{e}$$

Doubles

$$\pm m \times 2^{e}$$

- $1 \leq m \leq 2^{53} - 1$
- $-1022 \leq e \leq 971$

For more details, you can read about double-precision binary representation: https://en.wikipedia.org/wiki/Double-precision_floating-point_format

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

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:

Double.PositiveInfinity / 0.0 $\mapsto$ 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 ↦
0.6000000000000001

0.1 + (0.2 + 0.3) ↦
0.6
```

Doubles Break Common Algebraic Properties

- Equality is not reflexive:

```
Double.NaN != Double.NaN
```

- Multiplication does not distribute over addition:

```
100.0 * (0.1 + 0.2) ↦
30.000000000000004

100.0 * 0.1 + 100.0 * 0.2 ↦
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:

```
abs(x - y) <= tolerance
```

Defining Absolute Value

```
def abs(x: Double) = if (x >= 0) x else -x
```

What's wrong here?

```
abs(-0.0) ↦

if (-0.0 >= 0) -0.0 else -(-0.0) ↦

if (true) -0.0 else -(-0.0) ↦

-0.0
```

Defining Absolute Value

```
def abs(x: Double) = if (x > 0) x else 0.0 - x
```

Does it work now?

```
abs(-0.0) ↦

if (-0.0 > 0) -0.0 else 0.0 - -0.0 ↦

if (false) -0.0 else 0.0 - -0.0 ↦

0.0 - -0.0

0.0
```

Review:
Computation by Reduction

Arithmetic Operations

Operation	**Static Type**	**Examples**
v_1 + v_2 v_1 - v_2 v_1 * v_2 v_1 / v_2	Int × Int ⟶ Int Double × Double ⟶ Double	5 - 1 ⟼ 4 9 / 0 ⟼ ⊥ 9.0 / 0.0 ⟼ Double.PositiveInfinity
+ v - v	Int ⟶ Int Double ⟶ Double	-(0) ⟼ 0 +(-7) ⟼ -7 -(-7) ⟼ 7 - (0.0) ⟼ -0.0
v.toDouble	Int ⟶ Double	3.toDouble ⟼ 3.0

Comparison Operations

Operation	Static Type	Examples
$v_1 == v_2$ $v_1 != v_2$	$\tau \times \tau \longrightarrow$ Boolean $v_1: \tau \quad v_2: \tau$	"x" == "x" ↦ true false != false ↦ false (-0.0) == 0.0 ↦ true
$v_1 < v_2$ $v_1 <= v_2$ $v_1 > v_2$ $v_1 >= v_2$	Int × Int ⟶ Boolean Double × Double ⟶ Boolean	1 < 1 ↦ false 5 > 4 ↦ true Double.NegativeInfinity <= Double.NaN ↦ false

Logical Operations

Operation	Static Type	Examples
v_1 & v_2 $v_1 \mid v_2$	Boolean × Boolean ⟶ Boolean	true & true ↦ true true & false ↦ false false | true ↦ true
! v	Boolean ⟶ Boolean	! true ↦ false ! false ↦ true

Function Applications

Given a function definition

```
def fn(x₀: T₀, x₁: T₁, …, xN: TN): TR = {
  expr_body
}
```

we get a corresponding reduction rule:

$$\texttt{fn}(v_0, v_1, \ldots, v_N) \mapsto \{\ \texttt{expr}_{body}[x_0 \mapsto v_0, x_1 \mapsto v_1, \ldots, x_N \mapsto v_N]\ \}$$

i.e., the function application reduces to the function body expression, but with a new rule for each formal parameter's symbol, reducing the symbol to the corresponding argument value from the application.

Function Application Example

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

square(6.0) ↦
6.0 * 6.0 ↦
36.0
```

Conditional Expressions

Computing Conditional Expressions

- We used a bit of hand-waiving 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 `e1 ↦ v1`, then `e2 ↦ v2`, then `e3 ↦ v3`
- If `v1` is true then reduce to `v2`
- Otherwise reduce to `v3`

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

Short-Circuiting Logical Operations as If-Expressions

Short-circuiting operations can be rewritten as equivalent if-expressions:

- `x && y  ↦  if (x) y else false`
- `x || y  ↦  if (x) true else y`

Therefore, we use the same deferred-evaluation rule for the right-hand argument of short-circuiting operations as we use for an *if*-expression's *then/else* subexpressions.

Conditional and Short-Circuiting Operations

Rule	**Static Type**
if (true) $expr_1$ else $expr_2$ $\mapsto$ $expr_1$ if (false) $expr_1$ else $expr_2$ $\mapsto$ $expr_2$	Boolean $\times\ \tau \times \tau \longrightarrow \tau$ $expr_1$: τ $expr_2$: τ
true && $expr_2$ $\mapsto$ $expr_2$ false && $expr_2$ $\mapsto$ false true || $expr_2$ $\mapsto$ true false || $expr_2$ $\mapsto$ $expr_2$	Boolean $\times$ Boolean $\longrightarrow$ Boolean

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*

Scala Comments

Scala supports Java-style comments:

```
/* Multi
Line
Comment */

/** Scala-doc-style
  * multiline comment
  */

// single line comment
```

Contract for Factorial

```
def factorial(n: Int): Int = {
  require(0 <= n & n <= 12)
  // ... implementation ...
} 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 $\Rightarrow$ 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(arg₀: Type₀, …, argₖ: Typeₖ): ReturnType = {
      require(expr_precondition₀)
      require(expr_precondition₁)
      require(expr_preconditionₖ)
      expr_function_body
} ensuring(expr_postcondition₀)
. ensuring(expr_postcondition₁)
. ensuring(expr_postconditionₖ)
```