

Comp 311

Functional Programming

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

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

So, what are types?

Values Have **Value Types**

Definition: *A value type is a name for a collection of values with common properties.*

Values Have **Value Types**

- Examples of value types:
 - Natural numbers
 - Integers
 - Floating point numbers
 - *And many more*

Expressions Have **Static Types**

Definition (Attempt 1): A *static type* is an assertion that an expression reduces to a value with a particular *value type*.

Expressions Have **Static Types**

$4 + 5 : \mathbf{N} \mapsto 9 : \mathbf{N}$

Static Type



Value Type



Rules for Static Types

- If an expression is a value, its static type is its value type

5: N

- With each operator, there are “if-then” rules stating the required static types of the operands, and the static type of the application:

Integer Addition: If the operands to + are of type N then the application is of type N

Expressions Have **Static Types**

Definition (Attempt 1): *A static type is an assertion that an expression reduces to a value with a particular value type.*

Not quite.

Expressions Have **Static Types**

16 / 20: **Q** \mapsto 0.8: **Q**

So far, so good...

Expressions Have **Static Types**

16 / 0: **Q** \mapsto ?

Expressions Have **Static Types**

Definition (Attempt 2): A *static type* is an *assertion* that either an expression reduces to a value with a particular *value type*, or one of a well-defined set of exceptional events occurs.

Why Static Types?

- Using our rules, we can determine whether an expression has a static type
- If it does, we say the expression is *well-typed*, and we know that proceeding with our computation is *type safe*:
 - Either our computation will finish with a value of the determined value type, or one of a well-defined exceptional events will occur

What Constitutes the Set of Well-Defined Exceptional Events in Arithmetic?

- A “division by zero” error
- What else?

What are the Well-Defined Exceptional Events in Arithmetic?

- A “division by zero” error
- What if we run out of paper?
 - Or pencil lead? Or erasers?
- What if we run out of time?

What Constitutes the Set of Well-Defined Exceptional Events in Arithmetic?

- A “division by zero” error
- We run out of some finite resource

Our Second Exposure to
Computation:

Algebra

Now, We Learn How to Define Our Own Operators (a.k.a. functions)

$$f(x) = 2x + 1$$

$$f(x, y) = x^2 + y^2$$

And We Learn How to Compute With Them

$$f(x) = 2x + 1$$

$$f(3 + 2) \mapsto$$

$$f(5) \mapsto$$

$$(2 \times 5) + 1 \mapsto$$

$$10 + 1 \mapsto$$

$$11$$

The Substitution Rule of Computation

- To reduce an application of a function to a set of arguments:
 - Reduce the arguments, left to right
 - Reduce the body of the function, with each parameter replaced by the corresponding argument

Using the Substitution Rule

$$f(x, y) = x^2 + y^2$$

$$f(4 - 5, 3 + 1) \mapsto$$

$$f(-1, 3 + 1) \mapsto$$

$$f(-1, 4) \mapsto$$

$$-1^2 + 4^2 \mapsto$$

$$1 + 16 \mapsto$$

What About Types?

- Eventually, we learn that our functions need to include rules indicating the required types of their arguments, and the types of applications
- You might have seen notation like this in a math class:

$$f: \mathbf{Z} \rightarrow \mathbf{Z}$$

Typing Rules for Functions

$$f: \mathbf{Z} \rightarrow \mathbf{Z}$$

What does this rule mean?

Typing Rules for Functions

$$f: \mathbf{Z} \rightarrow \mathbf{Z}$$

- We can interpret the arrow as denoting data flow:

The function f consumes arguments with value type \mathbf{Z} and produces values with value type \mathbf{Z}

(or one of a well-defined set of exceptional events occurs).

Typing Rules for Functions

$$f: \mathbf{Z} \rightarrow \mathbf{Z}$$

- We can also interpret the arrow as logical implication:

If f is applied to an argument expression with static type \mathbf{Z} then the application expression has static type \mathbf{Z} .

What are The Exceptional Events in Algebra?

- A “division by zero” error
- We run out of some finite resource
- What else?

The Substitution Rule Allows for Computations that Never Finish

$$f: \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$$

$$f(x, y) = f(x, y)$$

$$f(4 - 5, 3 + 1) \mapsto$$

$$f(-1, 3 + 1) \mapsto$$

$$f(-1, 4) \mapsto$$

$$f(-1, 4) \mapsto$$

...

The Substitution Rule Allows for Computations that Keep Getting Larger

$$f: \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$$

$$f(x, y) = f(f(x, y), f(x, y))$$

$$f(4 - 5, 3 + 1) \mapsto$$

$$f(-1, 3 + 1) \mapsto$$

$$f(-1, 4) \mapsto$$

$$f(f(-1, 4), f(-1, 4)) \mapsto$$

$$f(f(f(-1, 4), f(-1, 4)), f(f(-1, 4), f(-1, 4))) \mapsto$$

...

But We Need at Least Limited Recursion
to Define Common Algebraic Constructs

$$!: \mathbf{N} \rightarrow \mathbf{N}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

What are The Exceptional Events in Algebra?

- A “division by zero” error
- We run out of some finite resource
- The computation never stops (unbounded time)
- The computation keeps getting larger (unbounded space)

Our Third Exposure to
Computation:

Core Scala

Core Scala

- We will continue to use algebra as our model of computation
- We will switch to Scala syntax
- We will introduce new value types

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!"

Primitive Operators on Ints and Doubles in Core Scala

Algebraic operators:

$$e + e' \quad e - e' \quad e * e' \quad e / e'$$

- For each operator:
 - If both arguments to an application of an operator are of type Int then the application is of type Int
 - If both arguments to an application of an operator are of type Double then the application is of type Double

Primitive Operators on Ints and Doubles in Core Scala

Comparison operators:

$$\begin{array}{ccc} e == e' & e \leq e' & e \geq e' \\ e > e' & e < e' & \end{array}$$

- For each operator:
 - If both arguments to an application of an operator are of type `Int` then the application is of type `Boolean`
 - If both arguments to an application of an operator are of type `Double` then the application is of type `Boolean`

Some Primitive Operators on Booleans in Core Scala

Conjunction, Disjunction:

$e \ \& \ e'$ $e \ | \ e'$

- In both cases:
 - If both arguments to an application are of type Boolean then the application is of type Boolean

More Primitive Operators on Booleans in Core Scala

Negation:

`!e`

- If the argument to an application is of type `Boolean` then the application is of type `Boolean`

Yet More Primitive Operators on Booleans in Core Scala

Conditional Expressions:

`if (e) e' else e''`

- If the first argument is of type `Boolean` and the second and third argument are of the same type T then the application is of type T

Primitive Operators on Strings in Core Scala

String Concatenation:

$$e + e'$$

- If both arguments are of type `String` then the application is of type `String`

An Example Function Definition in Core Scala

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


Syntax for Defining Functions

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

- If there is no recursion, we do not need to declare the return type:

```
def fnName(arg0: type0, ..., argk: typek) =  
    expr
```

The Substitution Rule Works as Before

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

```
square(2.0 * 3.0) ↦
```

```
square(6.0) ↦
```

```
6.0 * 6.0 ↦
```

```
36.0
```

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 $n: \text{Int}$,

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

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”

$$2147483647 + 1 \mapsto -2147483648$$

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

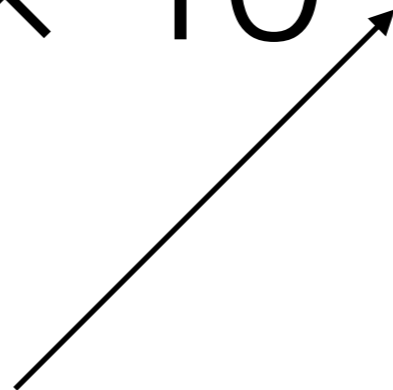
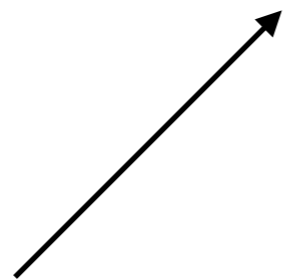
- Numeric values in scientific computations can span enormous ranges, from the very large to the very small
- At the same time, scientific measurements are of limited precision
- “Scientific notation” was devised in order to efficiently represent approximate values that span a large range

Scientific Notation

$$6.022 \times 10^{23}$$

mantissa

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

$$\pm m 2^e$$

- $1 \leq m \leq 2^{53}-1$
- $-2^{10}-53+3 \leq e \leq 2^{10}-53$
- $-1074 \leq e \leq 971$

The Nature of Doubles

Scientific Notation

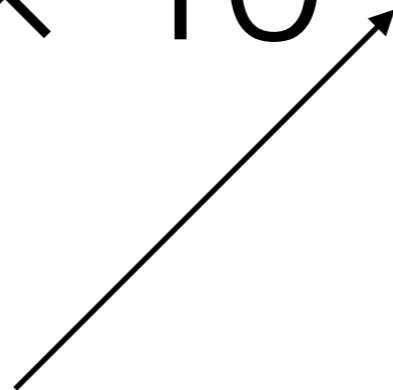
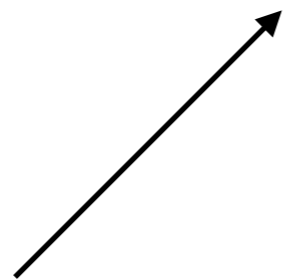
- Numeric values in scientific computations can span enormous ranges, from the very large to the very small
- At the same time, scientific measurements are of limited precision
- “Scientific notation” was devised in order to efficiently represent approximate values that span a large range

Scientific Notation

$$6.022 \times 10^{23}$$

mantissa

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

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 `Double.PositiveInfinity`

`1.0 / -0.0` \mapsto `Double.NegativeInfinity`

Division By Zero

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

```
Double.PositiveInfinity / 0.0 ↪  
    Double.PositiveInfinity
```

Division By Zero

- Division of a zero value by a zero value results in another special value NaN (for “Not a Number”):

$0.0 / 0.0 \mapsto \text{Double.NaN}$

$-0.0 / 0.0 \mapsto \text{Double.NaN}$