COMP3141

Software System Design and Implementation

Lecture 3: Monoids, Property-based Testing

Zoltan A. Kocsis
University of New South Wales
Term 2 2022

Announcements

Exercise 01: Submission should now be working. Extension: a generous one week June 23, 09:00 AM.
Exercise 02: Released tomorrow.
Assignment 01: Released on Monday, due July 3.

Quizzes: No individual extensions.

Revision: Monoids

A monoid consists of three things:

- a type $\mathtt{T}$,
- a *monoid operation* $\bullet :: \mathtt{T} \to \mathtt{T} \to \mathtt{T}$,
- an *identity element* $\mathrm{e} :: \mathtt{T}$

that obey the equations (laws)

1. $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ (associativity),
2. $x \bullet \mathrm{e} = x = \mathrm{e} \bullet x$

for all $x, y :: \mathtt{T}$.

Revision: Declaring Monoids

To declare the monoid consisting of `Int`, addition and the identity element `0`:

```
instance Semigroup Int where
  x <> y = x + y  -- bullet (semigroup operation)
instance Monoid Int where
  mappend = (<>)  -- bullet (same as Semigroup op)
  mempty = 0      -- identity
-- We have to check the laws ourselves!
```

Never declare `Monoid Int`!

Revision: Declaring Monoids

Never declare `Monoid Int`! Why?

- Each type can have at most one `Monoid` instance.
- But `Int` forms a monoid in multiple ways: $(\mathbb{Z}, +, 0)$, $(\mathbb{Z}, \times, 1)$
- Instead: use a `data`/`newtype` for each instance.

```
data Sum = Sum Int deriving (Eq, Show)
instance Semigroup Int where
  Sum x <> Sum y = Sum (x + y)
instance Monoid Sum where
  mappend = (<>)
  mempty = Sum 0
```

Demo: define Prod

Why care about monoids?

- Monoids: a simple type class.
- Type classes are somewhat like interfaces.
- Most important: the **laws** they satisfy.

The laws allows us to write algorithms that work **correctly** with every monoid.

Monoid Algorithms

Advantages of generic algorithms that work with every monoid:

- **Write once**: need to write only one implementation.
- **Test/prove once**: establish correctness using the laws.
- **Reuse**: hundreds of monoids occur in real-world code.

  But you have to learn to recognize them!

Two examples: Fast Monoid Exponentiation; MapReduce.

Fast Monoid Exponentiation I

We can define repeated application of the monoid operation.

$$x^2 = x \bullet x$$

$$x^3 = x \bullet x \bullet x$$

$$x^5 = x \bullet x \bullet x \bullet x \bullet x$$

This looks sort-of like usual exponentiation. NB it is well-defined only because $\bullet$ is associative.

Fast Monoid Exponentiation II

We can define it recursively: $x^0 = \mathrm{e}$, $x^{n+1} = x \bullet x^n$

Or in Haskell:

```
expo :: (Monoid g) => g -> Int -> g
expo x 0 = mempty
expo x n = x <> expo x (n-1)
  where (<>) = mappend
```

Notice: works for every monoid, once and for all

Fast Monoid Exponentiation III

Can we do better? Think about how we evaluate

$$2^8 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2.$$

We could do it naively/iteratively:

$$\begin{aligned}
2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot \color{red}{2 \cdot 2} \\
2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot \color{red}{2 \cdot 4} \\
2 \cdot 2 \cdot 2 \cdot 2 \cdot \color{red}{2 \cdot 8} \\
2 \cdot 2 \cdot 2 \cdot \color{red}{2 \cdot 16} \\
2 \cdot 2 \cdot \color{red}{2 \cdot 32} \\
2 \cdot \color{red}{2 \cdot 64} \\
\color{red}{2 \cdot 128} \\
256
\end{aligned}$$

Took 7 steps (i.e. 7 unique multiplications).

Fast Monoid Exponentiation IV

But we could do better! Notice that

$$2^8 = (2 \cdot 2 \cdot 2 \cdot 2)(2 \cdot 2 \cdot 2 \cdot 2).$$

So we could do $2 \cdot 2 \cdot 2 \cdot 2$ only once:

$$\begin{aligned}(2 \cdot 2 \cdot 2 \cdot 2) \cdot (2 \cdot 2 \cdot 2 \cdot 2)\\ (2 \cdot 2 \cdot 4) \cdot (2 \cdot 2 \cdot 4)\\ (2 \cdot 8) \cdot (2 \cdot 8)\\ (16) \cdot (16)\\ 256\end{aligned}$$

Took 4 steps (i.e. 4 unique multiplications).

Fast Monoid Exponentiation V

But we could do **even** better! Repeat the same trick:

$$2^8 = ((2 \cdot 2) \cdot (2 \cdot 2)) \cdot ((2 \cdot 2) \cdot (2 \cdot 2)).$$

So we could do $2 \cdot 2 \cdot 2 \cdot 2$ only once:

$$\begin{aligned} &((2 \cdot 2) \cdot (2 \cdot 2)) \cdot ((2 \cdot 2) \cdot (2 \cdot 2)) \\ &(4 \cdot 4) \cdot (4 \cdot 4) \\ &(16) \cdot (16) \\ &256 \end{aligned}$$

Took 3 steps (i.e. 3 unique multiplications).

Fast Monoid Exponentiation VI

Everything we've done relies only on associativity.
Therefore: The same trick works correctly for *every* monoid! We can define fast exponentiation:

```
fexpo :: (Monoid g) => g -> Int -> g
fexpo x 0 = mempty
fexpo x n
  | even n = y <> y
  | otherwise = x <> fexpo x (n - 1) where
  y = fexpo x (n `div` 2)
```

It will work *equivalently to* expo for every monoid.
Demo: performance comparison

FME Applications

Applications:

- Public-key cryptography (fast modular exp.)
- Linear algebra - finance, science, ML, etc. (fast matrix exp.)
- In the practical tomorrow.

Benefits:

- Optimize it once,
- Test it once (or prove it correct),
- Reuse many times (just supply the monoid).

MapReduce I

MapReduce

An algorithm for efficient, parallel processing of very large data sets.

Parallel: across multiple machines (on a network); across multiple cores (on a machine)

- Introduced at Google in 2004 (first to face the issue: building the search index).
- Simple processing problems (GOOG: build index, calculate PageRank).
- Massive data (GOOG: entire Internet).
- Lots of computers (GOOG: data centers).
- How do we split the jobs up?
- Requirements: efficient, correct, easy to (re)use.

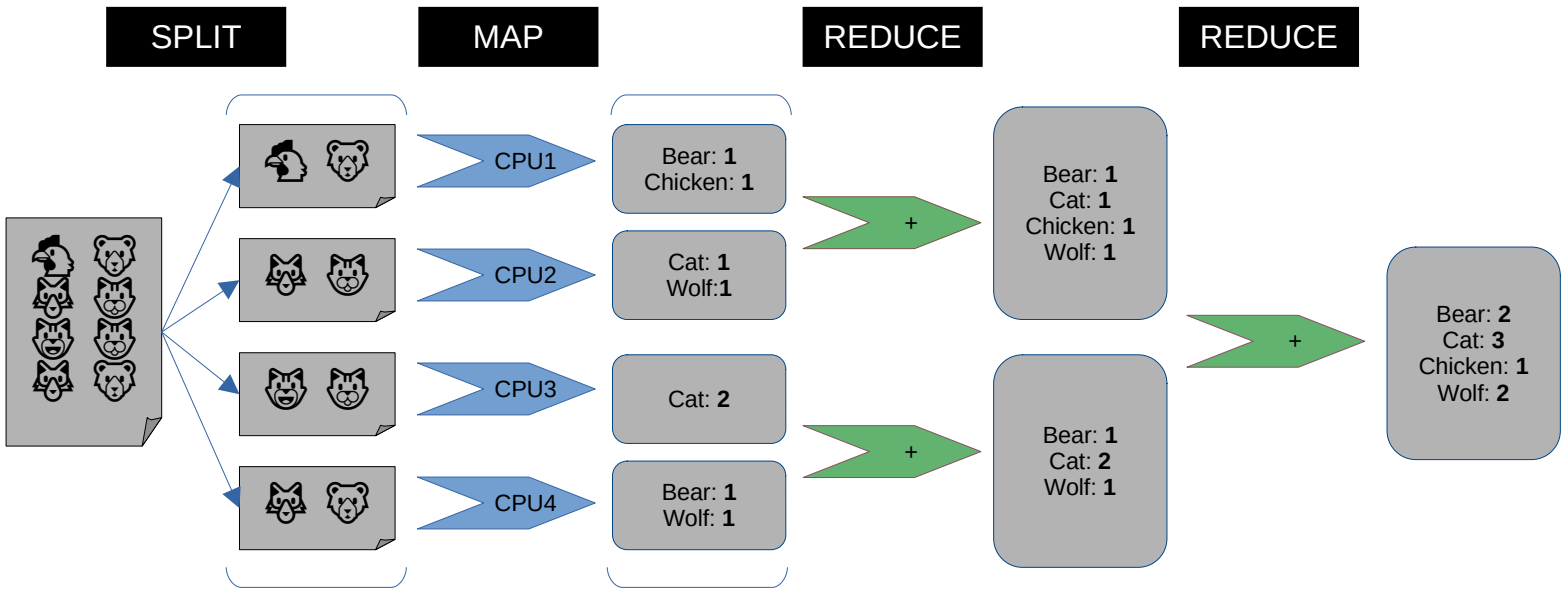
MapReduce II

What emerged:

- A generic algorithm (MapReduce)
- A programming model (confusingly also called MapReduce)
- In wide use today: Hadoop, Spark (in-memory MR), etc.
- Powered by monoids!

What it does: allows computations to take advantage of a large number of machines/cores/etc.

MapReduce III



MapReduce IV

Observation

What makes the reduce phase work is that the reduction operation

```
(<>) ::  Counts -> Counts -> Counts
```

forms a **monoid**!

MapReduce V

We have a bunch of results:

$$r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8$$

We need to combine them using $\bullet$:

$$r_1 \bullet r_2 \bullet r_3 \bullet r_4 \bullet r_5 \bullet r_6 \bullet r_7 \bullet r_8$$

Thanks to associativity, we can parenthesize them:

$$(r_1 \bullet r_2) \bullet (r_3 \bullet r_4) \bullet (r_5 \bullet r_6) \bullet (r_7 \bullet r_8)$$

Run each block in parallel (on 4 different computers):

$$(r_1 \bullet r_2) \bullet (r_3 \bullet r_4) \bullet (r_5 \bullet r_6) \bullet (r_7 \bullet r_8)$$

Repeat on the 4 intermediate results (w/ 2 computers)...

$$(R_1 \bullet R_2) \bullet (R_3 \bullet R_4) \Rightarrow S_1 \bullet S_2 \Rightarrow F$$

MapReduce VI

MapReduce fails if the reduce operation is not associative. Think of the *rock-paper-scissors* operation $\bullet :: \mathtt{RPS} \to \mathtt{RPS} \to \mathtt{RPS}$ which returns the winner. A list of results `[R,R,R,R,S,P]` reduced on 3 computers would yield:

$$\begin{aligned}(\mathtt{R} \bullet \mathtt{R}) \bullet (\mathtt{R} \bullet \mathtt{R}) \bullet (\mathtt{S} \bullet \mathtt{P}) &= \\ \mathtt{R} \bullet \mathtt{R} \bullet \mathtt{S} &= \\ \mathtt{R} \bullet \mathtt{S} &= \\ \mathtt{R}.\end{aligned}$$

but if we have only 2 computers it would yield:

$$\begin{aligned}(\mathtt{R} \bullet \mathtt{R} \bullet \mathtt{R}) \bullet (\mathtt{R} \bullet \mathtt{S} \bullet \mathtt{P}) &= \\ \mathtt{R} \bullet \mathtt{P} &= \\ \mathtt{P}.\end{aligned}$$

NB Ad-hoc parenthesizing non-associative operations does not make sense. MapReduce needs associativity!

MapReduce VII

Non-parallel sketch of MapReduce in Haskell:

```
mapReduce -- non-parallelized
  :: (Monoid g) =>
     (i -> [p]) -- splitter function
  -> (p -> g)   -- map function
  -> i          -- input to split
  -> g          -- final result
mapReduce split f input = mconcat . map f . split $ input
```

MapReduce VIII

Somewhat parallel version: also easy to write.

```
import Control.Parallel (par)
import Control.Parallel.Strategies (using, parMap, rpar)

mapReduce' :: (Monoid g) =>
              (i -> [p]) -> (p -> g) -> i -> g
mapReduce' split f input = mapResult `par` reduceResult
 where
  mapResult  = parMap rpar f (split input)
  reduceResult = mconcat mapResult `using` rpar
```

This (and only this!) slide is not going to be examined: you don't need to understand how parallelism works.

MapReduce IX

- Keep in mind that this is about the MapReduce algorithm, as opposed to the framework (programming model).
- A real MapReduce framework implementation does a lot of other stuff: shuffling data, dealing with machines that crash, saving temporary results, etc.
- You need to understand the algorithm as presented here. You don't need to learn a MapReduce framework.

Free Properties

Moral of the story so fafr: To correctly use generic algorithms (like MapReduce) we need to make sure that certain properties (associativity of $\bullet$) actually hold.
Haskell already ensures many properties automatically with its language design and type system.

1. Memory is accessed where and when it is safe and permitted to be accessed (*memory safety*).
2. Values declared with a certain static type will actually have that type at run time (*type safety*).
3. All functions are *pure*: Programs don't have side effects, equational reasoning works (*purely functional programming*).

Functional Properties

But Haskell can't guarantee most functional properties: properties not of a function implementation (performance, memory safety), but purely of the input-output mapping.
We have already seen a few examples.

Example (Properties)

1. `reverse` is an *involution*: `reverse (reverse xs) == xs`
2. right identity for `(++)`: `xs ++ [] == xs`
3. transitivity of `(>)`: `(a > b)` $\wedge$ `(b > c)` $\Rightarrow$ `(a > c)`

The set of properties that capture all of our requirements for our program is called the *functional correctness specification* of our software.

This defines what it means for software to be correct.

Proofs

Last week we saw some *proof methods* for Haskell programs. We could prove that our implementation preserves invariants (`map/length`), meets its correctness specification, or generates the same output as a simpler, slower implementation.

Such proofs certainly offer a high degree of assurance, but:

- Proofs must make some assumptions about the environment and the semantics of the software.
- Proof complexity grows with implementation complexity, sometimes drastically.
- If software is incorrect, a proof attempt might simply become stuck: we do not always get constructive negative feedback.
- Proofs can be labour and time intensive ($$$), or require highly specialised knowledge ($$$).

Testing

Compared to proofs:

- Tests typically run the actual program, so requires fewer assumptions about the language semantics or operating environment.
- Test complexity does not grow with implementation complexity, so long as the specification is unchanged.
- Incorrect software when tested leads to immediate, debuggable counterexamples.
- Testing is typically cheaper and faster than proving.
- Tests care about efficiency and computability, unlike proofs.

We lose some assurance, but gain some convenience ($$$).

Property Based Testing

Key idea: Generate random input values, and test properties by running them.

Example (QuickCheck Property)

```
prop_reverseApp xs ys =
    reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Property: this should hold for all lists `xs`,`ys`.
Test: try many random lists `xs`,`ys`.

Haskell's *QuickCheck* wass the first library invented for property-based testing. The concept has since been ported to Erlang, Scheme, Common Lisp, Perl, Python, Ruby, Java, Scala, F#, OCaml, Standard ML, C and C++.

PBT vs. Unit Testing

- Properties are more compact than unit tests, and describe more cases.
  $\Rightarrow$ Less testing code
- Property-based testing heavily depends on test data generation:
  - Random inputs may not be as informative as hand-crafted inputs
    $\Rightarrow$ use shrinking
  - Random inputs may not cover all necessary corner cases:
    $\Rightarrow$ use a coverage checker
  - Random inputs must be generated for user-defined types:
    $\Rightarrow$ QuickCheck includes functions to build custom generators
- By increasing the number of random inputs, we improve code coverage in PBT.

Test Data Generation

Data which can be generated randomly is represented by the following type class:

```
class Arbitrary a where
  arbitrary :: Gen a  -- more on this later
  shrink :: a -> [a]
```

Most of the types we have seen so far implement `Arbitrary`.

Shrinking

The `shrink` function is for when test cases fail. If a given input `x` fails, QuickCheck will try all inputs in `shrink x`; repeating the process until the smallest possible input is found.

Testable Types

The type of the `quickCheck` function is:

```
                                    -- more on IO later
quickCheck :: (Testable a) => a -> IO ()
```

The `Testable` type class is the class of things that can be converted into properties. This includes:

- `Bool` values,
- Any function from an `Arbitrary` input to a `Testable` output:

  ```
  instance (Arbitrary i, Testable o)
        => Testable (i -> o) ...
  ```

Thus the type `[Int] -> [Int] -> Bool` (as used earlier) is `Testable`.

Demo: Simple example

Is this function reflexive?

```
divisible :: Integer -> Integer -> Bool
divisible x y = x `mod` y == 0

prop_refl :: Integer -> Bool
prop_refl x = divisible x x
```

- Encode pre-conditions with the (==>) operator:

  ```
  prop_refl :: Integer -> Bool
  prop_refl x = x > 0 ==> divisible x x
  ```

  (but may generate a lot of spurious cases)

- or select different generators with modifier `data` types.

  ```
  prop_refl :: Positive Integer -> Bool
  prop_refl (Positive x) = divisible x x
  ```

  (but may require you to define custom generators)

Arbitrary instances

`Gen t` is a data type that describes *how* to generate a random element of type `t`.

When writing your own `Arbitrary` instances, you'll need to work with `Gen`, and you'll need to know:

```
arbitrary :: Arbitrary t => Gen t
(<$>) :: (a -> b) -> Gen a -> Gen b
(<*>) :: Gen (a -> b) -> Gen a -> Gen b
pure :: a -> Gen a
oneof :: [Gen a] -> Gen a
```

Warning: `:t` will give more general types!

Arbitrary example

```haskell
data Color = Color Int Int Int deriving (Show, Eq)
-- safe constructor
color :: Int -> Int -> Int -> Color
color x y z =
  Color (x `mod` 255) (y `mod` 255) (z `mod` 255)

instance Arbitrary Color where
  arbitrary =
    color <$> arbitrary <*> arbitrary <*> arbitrary
```

Demo: RPS, Color arbitrary instance

FIN

1. Don't forget to submit Quiz 2.
2. Exercise 1: submission instructions will be posted today, you have a week-long extension.
3. Exercise 2 and Quiz 3 will be released tomorrow.