FUNCTIONAL CONCEPTS

Introduction

What's that

We are going to look at the following and try to explain them.

- Setoid
- Semigroup
- Monoid
- Functor
- Applicative
- Monad

We'll then look at why that's a good idea.

What it's not

It's always nice to try and explain why something is a good idea and what the point of it is, it helps with the learning. With these principles it's a little like trying to explain to someone starting to learn to code why it's a good idea to have a kinda little box with a value on that we give a name and call it a variable. Sometimes you just have to go with it, and things become clearer later, so you aren't going to get the "why" up front. We'll be sure to move on to that later though.

This is NOT going to be a mathematically, algebraically, category theoryally strict definition of what we discuss – strict correctness too often means many corollaries, qualifications and generally illegibility so as to make the subject impenetrable to the beginner. We may note as a detail some of the strict things we are glossing over, but not all of them!

- For instance we'll take later about associativity. This can be explained by saying that for some operation ".", then

  ```
  (x.y).z == x.(y.z)
  ```

  ... Or ... the whole 5 screenfulls of various exotic symbols that is https://en.wikipedia.org/wiki/Associative_property

Not being strict means we'll possibly bust a learning principle called *Primacy* (https://en.wikipedia.org/wiki/Principles_of_learning#Primacy). This holds that if you don't teach it right first time, it's much harder to shake the wrong first understanding later. It's especially important in safety-related teaching where stress and confusion can result in subconsciously reverting to those first-taught principles, despite later teaching and experience.

- Say a sub-aqua diver is learning in a shallow training pool and is taught that if their oxygen supply cuts they should stand up and remove the mask. Years later with lots of more advanced training and experience, they may get into difficulty. It's possible they may react instinctively by kicking their legs to stand up – and possibly ripping off the mask. That would be primacy at work.

If theoretical strictness and primacy are your kinda thing, probably best to find another guide!

Show me in C#

We'll show the ideas in C# code whenever possible. The idea is to give a familiar context for an "average" C# programmer to what is being discussed. It's not the intention to give the best code or perfect code, but it will be valid C# code, which is available for download.

There's always a way to write it better - where everyone has their own definition of "better". In fact from experience if there are *n* programmers in a room, they will probably come up with at least 2^n+1 ways to improve any given bit of code.

So the C# given is illustrative of the ideas, not a strict definition or a template for all to follow.

Setoid

What's that

A setoid is a type together with an understanding of which items are "equivalent". A definition for "equivalent" may be "equals" – all the member values of two objects of the type match – but it doesn't have to be this.

The understanding of equivalence may vary depending on the usage even for the same type of object. So there may be more than one Setoid relating to a given type. See the product examples below.

Explain the name

Setoid – some rules about the things that can be in a "set" such as a container or collection (let's not get too strict about what a "set" is).

What it's not

Despite its name, it's not a kind of set, container or collection, or a subtype of one of those - It's not some sort of fancy alternative to a list.

It's not a thing that's in a collection (but in a collection where we care about setoids, the items will be of the type belonging to a setoid).

It's not a type, it's not a way of understand the equivalence between two objects of a type (it's the two together).

Functions/delegates/methods don't form setoids. There can be a collection or container of them, but there's no useful understanding of equivalence - apart from exhaustive trial and error, how could it be determined that f(x) => x^2 is equivalent to f(x) => x * x ?

C# mashes this a bit because it treats functions/delegates/methods as an object which have an "Equals()" method, but this is a reference equals only. It also uses a kludge to enable Equals() to be called on Value types, which is handy for us to do examples using Ints and Floats, but muddies the waters a bit when trying to relate the concepts in C#.

Examples

- When looking at products, if we are doing a stock-take we may wish to consider each item unique based on serial number.
- Or if compiling a catalogue, we may consider all the same product type and package quantity as equivalent.
- Customers on a website may be uniquely identified by the credit card number they use irrespective of what name, spelling or address they use at the time of order.
- More generally if an object has any form of unique id, the equivalence test can be shortcutted so it doesn't look at every member value, just the unique id.

Show me in C#

There's very little to this code-wise. Strictly you could write this.

```
public interface ISetoid<T>
{
    bool IsEquivalentTo(T other);
}
```

But in C# the object base class kinda implements the setoid interface definition, and it's probably better all round in C# not to bother and to just override the Equals() method of the base object, though you'll need to cast to your type and add null protection, such as

```
public class MyClass
{
    public override bool Equals(object obj)
    {
        var other = obj as MyClass;
        if (other == null) return false;
        return this.IsEquivalentTo(other);
    }

    public bool IsEquivalentTo(MyClass other)
    {
        // ...
    }
}
```

This is the simple case just using the default Equals(). If there's a need for different understandings of equivalence for the same type in different use cases, it may be necessary to define and call different methods to test equivalence when using the class as a setoid in different situations.

Deep dive

- Some Haskell examples are given in https://hackage.haskell.org/package/setoid-0.1.0.0/docs/Data-Setoid.html

Get the picture

So here's what a setoid isn't, and is, in pictures.

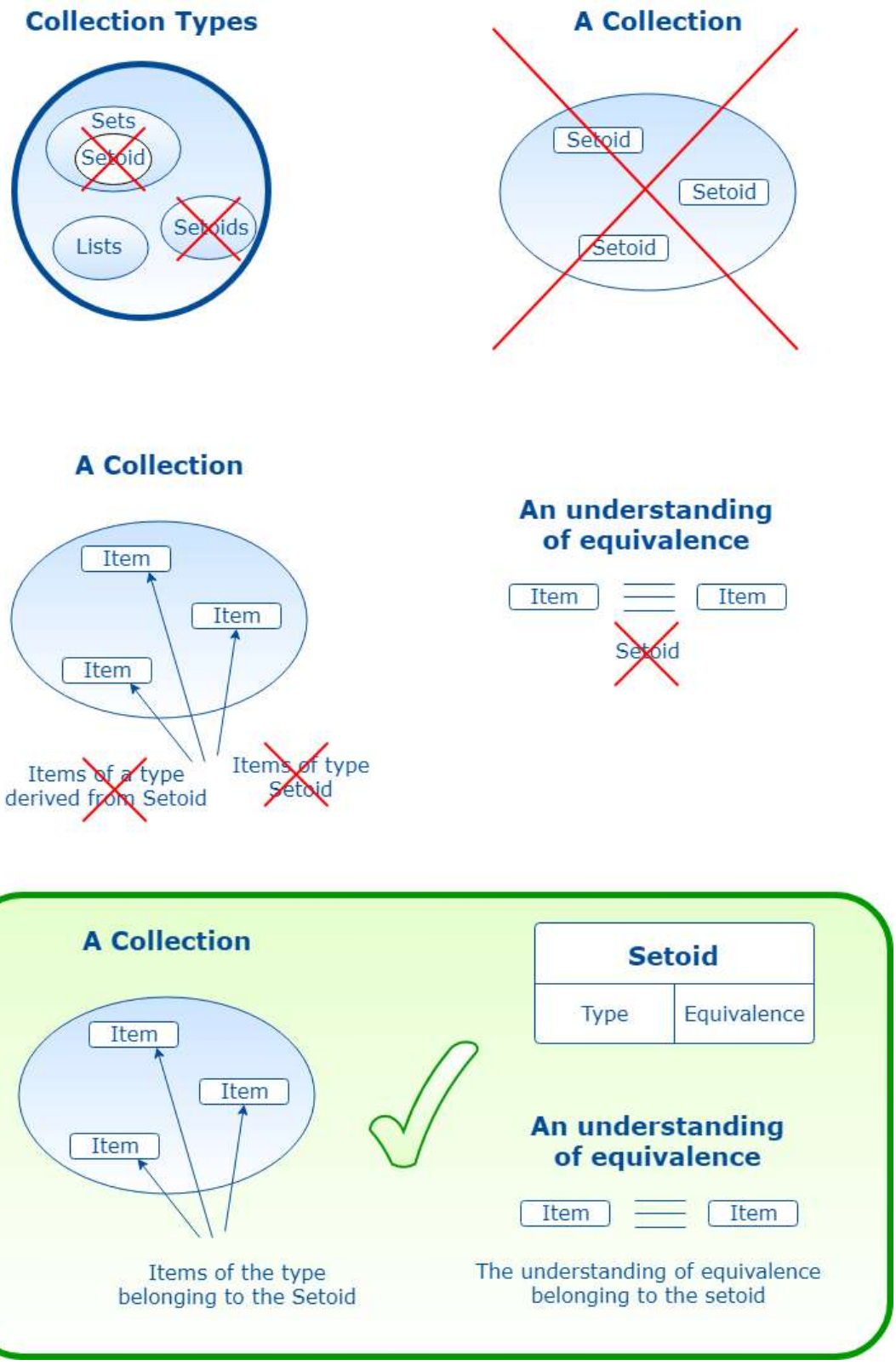

Figure 1 - Setoid

Semigroup

What's that

It's a collection of items of a type belonging to a Setoid together with an *associative* operation.

Associative means that for an operation denoted by ".", that

```
(x.y).z == x.(y.z)
```

And when we say "==", what we mean is the understanding of equivalence that the setoid has.

But it's the two things together

- a collection of items of the a type which belongs to a setoid
- an associative operation

which make a semigroup.

Strictly, the collection and the operation should also be "closed" meaning that for every

```
z = x.y
```

with x and y from the collection, then z will also be from the collection.

This is a more theoretical issue than a practical one for a developer. If the type of the setoid is integer, then this together with the addition operation seems like it would be a semigroup – it's our first example below.

But in C# you get into the issue of maxint so it isn't strictly a semigroup – what's MaxInt+10?. It's either a value outside of the bounds of Int32, or if we try to be clever it's -9, but then we find that the operation is no longer associative.

Mostly the closure bit can be ignored from the functional concepts perspective we are looking at, though as always it will need to be considered when choosing your types based on the range of expected values.

Explain the name

This is probably the clearest-named thing we will look at in functional concepts.

You won't be surprised to hear it's got less rules to abide by than a "group" (which we aren't going to look at).

But it's got one more rule than a ... *Magma* ... mathematicians eh? Go figure!

What it's not

It's not the nature of the collection – dictionary, list etc.

It's not the type of what's in the collection (that's the type which belongs to the semigroup's Setoid).

It's not the collection. It's not the operation. (it's the two together).

Examples

- Positive Integers (1,2,3….n) together with addition form a semigroup.

```
(1 + 2) + 3 == 1 + (2 + 3)
      3 + 3 == 1 + 5
          6 == 6
            ✓
```

- Strings together with concatenation form a semigroup.

```
("hello" + "there") + "world" == "hello" + ("there" + "world")
       "hellothere" + "world" == "hello" + "thereworld"
            "hellothereworld" == "hellothereworld"
                              ✓
```

- Postive integers together with subtraction are, perhaps surprisingly, NOT a semigroup since the operation is not associative

```
(3 - 2) - 1 == 3 - (2 - 1)
      1 - 1 == 3 - 1
          0 == 2
            ✖
```

  This is also why negative integers together with addition are not a semigroup.

- OK, so integers together with subtraction did not surprise you? Well we've already seen that integers and addition do form a semigroup. Then would it surprise you that on a computer, floating point numbers and addition do not form a semigroup, since floating point addition is not associative?

```
(0.001 + 0.001) + 0.003 == 0.001 + (0.001 + 0.003)
          0.002 + 0.003 == 0.001 + 0.004
                        ✖
```

  Don't believe it? Here's the output from Visual Studio 2017's C# interactive window (View -> Other Windows -> C# interactive):

```
> 0.001f + 0.001f
0.002
> 0.002f + 0.003f
0.005
> 0.001f + 0.003f
0.004
> 0.001f + 0.004f
0.00500000035
> (0.001f + 0.001f) + 0.003f == 0.001f + (0.001f + 0.003f)
False
```

There's more detail on this at https://www.quora.com/Is-floating-point-addition-commutative-and-associative and

Show me in C#

Let's make the assumption that the collection of the type which belongs to the setoid needed to form the semigroup can be any and all objects of the type which belongs to the setoid.

If we didn't make this assumption, we'd have to also pass in the collection of objects of the type which belongs to the setoid. That wouldn't be hard to code – another property on the interface of type `IEnumerable<T>` is all. But often we will want the collection to be all possible values of the type which belongs to the Setoid – e.g. all integers. C# doesn't give us a sensible usable way of specifying that, so let's pretend shall we?

We can then make use of a semigroup in C# with the following interface definition:

```
public interface ISemigroup<T>
{
    Func<T, T, T> AssociativeOperation { get; set; }
}
```

OK, let's actually define a semigroup class so we can test some of the examples above.

```
public class Semigroup<T> : ISemigroup<T>
{
    public Func<T, T, T> AssociativeOperation { get; set; }


    public Semigroup(Func<T, T, T> associativeOperation)
    {
        AssociativeOperation = associativeOperation;
    }

    // we can't really test over all possible values, so pass in some values to test with
    // needs to have all (x.y).z == x.(y.z)
    public bool IsValidSemigroup(ICollection<T> itemsToTestWith)
    {
        foreach (var x in itemsToTestWith)
        {
            foreach (var y in itemsToTestWith)
            {
                foreach (var z in itemsToTestWith)
                {
                    // x,y,z are values from itemsToTestWith
                    if (!AssociativeOperation(AssociativeOperation(x, y), z)
                        .Equals(AssociativeOperation(x, AssociativeOperation(y, z))))
                    {
                        // the operation is not associative for this x,y,z
                        return false;
                    }
                }
            }
        }
        return true;
    }
}
```

and let's define some operations to test with

```
public static class Concepts
{
    public static int IntAdd(int x, int y)
    {
        return x + y;
    }

    public static float FloatAdd(float x, float y)
    {
        return x + y;
    }
}
```

Now let's make an actual semigroup and use the method to test if it's valid. We'll use a little trick to generate a collection of integers from 1 to 100 to test with.

```
var semigroupValid = new Semigroup<int>(IntAdd)
    .IsValidSemigroup(Enumerable.Range(1, 100).ToArray());
// true
```

Nice. So let's try the float addition semigroup. We'll beef up the enumerable trick to give as a collection of float values.

```
var semigroupValid = new Semigroup<float>(FloatAdd)
    .IsValidSemigroup(Enumerable.Range(1, 100)
                        .Select(i => (float)i / 1000f)  //0.001, 0.002, ...0.100
                        .ToArray());
//false
```

The reason it's false is the lack of associativity of the float add operation on a computer.

Well we've come this far, so time for a confession – something was deliberately skipped over above when the float addition was pronounced non-associative. Our judgement on that was based on

```
0.002 + 0.003 == 0.001 + 0.004
```

✖

Because it actually evaluated to

```
0.005 == 0.00500000035
```

Except ... what is "=="? Let's go right back to the beginning in the Semigroup first section "What's that":

> *And when we say "==", what we mean is the understanding of equivalence that the setoid has.*

Above we read "==" as "equals" so *clearly* the evaluation above wasn't true. But we can define a semigroup with a setoid that has a different definition of "equivalent". Let's use the usual approach for float of checking if two values are within a tolerance. Now we aren't using a value type, the Add operation will have to be a wee bit more complicated and dig the value out of the object. This is something you may recognise in later sections. ;-)

```
public class Floatish
{
    public float Value;

    public Floatish(float value)
    {
        Value = value;
    }

    public override bool Equals(object obj)
    {
        // c# : null checks
        var other = obj as Floatish;
        return Math.Abs(Value - other.Value) < 0.00001;
    }

    public static Floatish FloatishAdd(Floatish x, Floatish y)
    {
        return new Floatish(x.Value + y.Value);
    }
}
```

Now we can use this to create and test a new semigroup. We need to play with the enumerable trick a bit more to create our collection of our setoid Floatish, but it's straightforward linq stuff.

```
var semigroupValid = new Semigroup<Floatish>(Floatish.FloatishAdd)
    .IsValidSemigroup(Enumerable.Range(1, 100)
        .Select(i => (float)i / 1000f) //0.001, 0.002, ...0.100
        .Select(f => new Floatish(f))
        .ToArray());
//true
```

Cool. Is starting to feel like an actual usable concept now? Where to use it still isn't obvious, but at least it's looking usable!

Deep dive

https://en.wikipedia.org/wiki/Semigroup

http://blog.ploeh.dk/2017/11/27/semigroups/

https://en.wikipedia.org/wiki/Group_(mathematics)

https://en.wikipedia.org/wiki/Magma_(algebra)

Get the picture