Monoids and their uses

Mateusz Kowalczyk
University of Bath
mk440@bath.ac.uk
fuuzetsu@fuuzetsu.co.uk

ABSTRACT
In light of the recent increase in interest towards functional elements of programming, this report aims to provide an overview of monoids. A mathematical definition of a monoid is given followed by some examples of common monoids and examples of usage of monoids in every day programming environment.

Keywords
monoid, functional programming

1. INTRODUCTION
There has recently been a trend where more and more programmers are becoming interested with functional programming languages. This is not a global change where we'll see Haskell or Scheme as the most popular language in the world by the end of 2013. In fact, no purely-functional languages even make the top 20 in popularity rankings[6]. Many of the languages that are gaining popularity nowadays at least include functional programming elements. Some examples include C# (popular LINQ library is implemented with monads), JavaScript (first-class functions and closures) and Python (λ, first-class functions, built-in functional programming libraries[2]). With increased interest in the functional part of the languages that many programmers already use, it's natural that interest in type theory and category theory is increased as that's often the basis for functional programming languages.

In light of this, I'm going to talk about a simple category of monoids and some sample examples of practical uses of monoids in software. I'll also briefly touch upon where monoids lie in terms of category theory.

This report uses Haskell as the language used for demonstration of any code where code is present, unless stated otherwise.

2. MONOIDS
There are a couple of definitions that can be given for what a monoid actually is. Monoids play a part in abstract algebra and category theory. From a point of view of abstract algebra, a monoid is a simple algebraic structure with three axioms. A monoid is a set S and a binary operation $\cdot$ such that:

- Closure
  - $\forall x, y \in S : x \cdot y \in S$
- Associativity
  - $\forall x, y, z \in S : (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Identity
  - $\exists e \in S : \forall x \in S : e \cdot x = x \cdot e = x$

A definition of a monoid in category can also be given and is probably the more interesting entity to study when put in terms of category theory. In this report however, the definition of the monoid as an algebraic structure will suffice. From here on, *mappend* will refer to the binary operation for any monoid and *mempty* to its identity element.

2.1 Examples of structures forming a monoid
There is a great amount of structures in use in mathematics that form a monoid under a certain operation. In fact, many of these are already indirectly studied as monoid naturally forms a semigroup with an identity. It's also worth noting that a group forms a monoid with an inverse. This means that monoids are not a fresh or a totally alien concept to mathematicians.

2.1.1 Natural numbers under addition
A fairly trivial example of a monoid is a set of all natural numbers $\mathbb{N}$ and a binary operation $+$:

- Closure
  - $\forall x, y \in \mathbb{N} : x + y \in \mathbb{N}$
- Associativity
  - $\forall x, y, z \in \mathbb{N} : (x + y) + z = x + (y + z)$
- Identity
  - $\forall x \in \mathbb{N} : 0 + x = x + 0 = x$

In this instance, by property of commutativity of + on integers, we can also see that $x + y = y + x$. Monoids with this property are called commutative monoids. Here $mappend = (+)$ and $mempty = 0$

2.1.2 Lists under concatenation

A bit less trivial example is an example of lists under concatenation. We use `x :: a` to mean `x` is a member of (or a proof of) the type (or a proposition) `a`, for any `x` and any `a`. Denoting lists as with square braces, we use `xs :: [a]` to mean `xs` is a member of type `[a]` where `[a]` is a list of elements of type `a`. Note that this holds even if `xs` is an empty list: `[]`. We now define `++` to be a binary operation that is used to append two lists:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

where `:` is cons operation and pattern matching is used.

To form a monoid under any `x :: [a]` with `++`, it's possible to show that for any set L with all elements of type `[a]`:

- Closure
  - $\forall x, y \in\in L : x + +y \in L$
- Associativity
  - $\forall x, y, z \in L : (x + +y) + +z = x + +(y + +z)$
- Identity
  - $\forall x \in L : [] + +x = x + +[] = x$

Proof by structural induction on `++` is possible and fairly easy but is omitted for brevity. Here `mappend = (++)` and `mempty = []`

This monoid is more interesting than the natural numbers over addition example because it works for any arbitrary list of polymorphic type `a`. As a consequence to this, we are now able to append/concatenate anything that we can directly represent as a list. An example is type `String` which can simply treat as an alias of `[Character]` where `Character` is a concrete type that we can use to represent letters, numbers etc. Given `x = "Hello " :: String` and `y = "World!" :: String`, we can exploit use our binary operation as `x ++ y = "Hello World!"`. For illustration purposes, any `c :: [Character]` will be rendered as a human-readable string, so we will write "Hello" instead of "['H', 'e', 'l', 'l', 'o']".

Last thing to note that unlike the natural numbers example, this monoid isn't commutative. `[2, 3] ++ [4, 5]` $\neq$ `[4, 5] ++ [2,3]`. This is interesting because we can form a whole new monoid, dual to the existing one simply by making our binary operation treat the arguments in reverse order. That is, we apply

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

to the binary operator of a non-commutative monoid to get a whole new operator out and a new monoid as a consequence. It can be shown that the monoid laws still hold.

3. USES

Having talked about what monoids are and giving two examples, we can now demonstrate some uses for practical situations.

3.1 Logging

It is often desirable that during an execution of a function, we gather some data along the results. In a case of a simple iterative function, this task would probably be achieved by appending to a separately defined list after each operation. This solution does not work if we have a recursive function however. In both cases, we have some function `f :: a -> c` where `a` is the type of argument, `c` is the type of the result. With logging, we also want to get some data out of the way. For convenience, I'm going to define a

```
Pair a b
```

type whose constructor takes any value of type `a` and any value of type `b` and stores them. So given `x :: String` and `y :: Integer`, we can form a pair with

```
pair x y :: Pair String Integer
```

. We then define

```
fst (Pair x y) = x
snd (Pair x y) = y
```

With that in place, we want to change our `f :: a -> c` to `g :: a -> Pair b c` where `b` is the type of the log we want to produce. As an example, consider a factorial function `fact`:

```
fact :: Natural -> Natural
fact 0 = 1
fact x = x * (fact (x - 1))
```

This works just fine (albeit very inefficiently) as is but what if we want to also get a string at each step saying what `x` was? Assume that we have a function `show :: Natural -> String`.

```
fact :: Natural -> Pair [String] Natural
fact 0 = pair [``x is 0''] 1
fact x = pair [``x is '' ++ (show x)] (x *
    (fact (x - 1))
```

Very quickly we'd find out that this doesn't work: `x * (fact (x - 1))` can not work as `fact` now returns `Pair [String] Natural` and not `Natural`! We can solve this with the list monoid in a fairly elegant matter.

```
fact :: Natural -> Pair [String] Natural
fact 0 = pair [``x is 0''] 1
fact x = pair ([``x is '' ++ x] ++ log) (x
    * val)
  where resultPair = fact (x - 1)
        log = fst resultPair
        val = snd resultPair
```

This way we simply fetch the result of `fact (x - 1)` and then concatenate the log together. Thanks to the associative property of monoids, we're guaranteed that the log will come

back in the correct order. We can even have the log come back in the reverse order by using the dual of the list monoid, by using `flip (++)` instead of `++`. Even better, the log can be anything that forms a monoid! Here we use a list monoid (inside another list) but we can use this pattern to keep a list of anything that forms a monoid, simply by substituting `++` for the binary operation of that specific monoid.[1] In fact, this is precisely what Haskells `Writer` monad does.

3.2 Folding

Another use of monoids is folding. Folding a `[a]` means applying a certain function `f :: a -> b -> a` to an accumulator and each element of the list. We can define a left fold[2] as follows:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f acc [] = acc
foldl f acc (x:xs) = foldl (f acc x) xs
```

It's actually possible to define this fold operation for all monoids. Even better, we get it for free from the definition of the monoid!

```
mconcat (Monoid a) => [a] -> a
mconcat = foldl mappend mempty
```

Here `(Monoid a) => [a] -> a` denotes that `a` has to be a monoid. We can now use mconcat on any monoid:

```
mconcat ["Hello ", "everyone ", "!!"] = "
    Hello everyone here!!"
mconcat [1, 2, 3] = 6
```

The uses for folding are far and wide in the programming world: calculating sums, products, means, finding minimums/maximums etc. There is an incredible amount of operations that can be done with a fold, even including sorting and finding primes[1]. It's therefore quite convenient that for any monoid we can form, we always have a free folding operation on it. While a list monoid might not have the most exciting fold, all the programmer is limited by is his ability to form monoids. Of interest to many people might be the fact that trees are themselves monoids in a similar fashion that lists are and trees are and both these constructs are very widely used in programming.

It has recently been suggested that monoids can be used to design more efficient and elegant algorithms for the MapReduce framework, even going as far as automating the process of optimisation by using monoidal combiners.[5]

4. CLOSING WORDS

I don't believe that any have touched upon monoids or any closely related topics. In fact, I think that the closest that someones talk has come to the topic was the "Semantics of Lazy Evaluation" talk on 30^{th} of May, simply through the fact that lazy evaluation is mostly found in functional programming languages. My own talk simply touched upon what a monoid is in algebraic terms and gave a few examples. As show during the talk, it's quite possible to shoehorn monoids (and many other structures) onto imperative languages but it requires large amount of boilerplate code. Oftentimes people find it worth to do anyway and libraries encapsulating various functional programming (and by proxy, category theory and often type theory, depending on the language) concepts are written and published[4].

I don't believe that monoids have had a huge impact on foundations of computation, both on how we compute and how we think about computation. Having said that, I'd like to make clear that I'm not saying monoids aren't useful or that they are aren't worth studying. I simply mean to convey that monoids are just a part of a bigger picture. Should we not study monoids because we have more powerful monads? Should we not study semigroups because we have groups? Clearly this isn't the case and very often we can't form a monoid or a group and we need slightly weaker structures or categories. Of course, any research done to monoids also applies to everything that is isomorphic or homomorphic to the monoid category also benefits. It's therefore important to not abandon the study of monoids because they seem like such a simple structure. As Donald Knuth himself says, there is still low hanging through in Computer Science and many, many things haven't yet been properly studied, even though they might have been around for years[3].

[1] We can take this even further by making our `Pair` a monoid itself and then using that as the log. For any monoid `a` and any monoid `b`, we can make `Pair a b` a monoid by making defining `mempty = Pair (mempty a) (mempty b)` and `mappend (Pair a b) (Pair c d) = Pair (mappend a c) (mappend b d)`.

[2] In addition to the left fold, we could also have picked the right fold which starts to apply the function from the end of the given list. Due to associativity of monoids, left and right folds using a monoid operation have exactly the same semantics.

5. REFERENCES

[1] Public effort. Fold. `http://www.haskell.org/haskellwiki/Fold`, May 2013.

[2] Python Software Foundation. itertools. `http://docs.python.org/2/library/itertools.html`, May 2013.

[3] GoogleTechTalks. "All Questions Answered" by Donald Knuth. `http://www.youtube.com/watch?v=xLBvCB2kr4Q`, March 2011.

[4] Functional Java. Functional Java. `http://code.google.com/p/functionaljava/`, May 2013.

[5] Jimmy Lin. Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms. April 2013.

[6] TIOBE. Tiobe programming community index for april 2013. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, May 2013.