Applications of Category Theory in Modern JavaScript

John Bender

john.m.bender@gmail.com

April, 2012

Abstract

The jQuery JavaScript library, used on more than 55% of Alexa's top 10,000 websites [1] makes the manipulation of HTML documents easy and intuitive through fluent method chaining and an intuitive API design. An unfortunate side effect of these user friendly features is that they often incur an otherwise unnecessary performance overhead. While JavaScript execution in desktop browsers has become fast enough to hide much of the problem, the growing complexity of HTML documents and the ubiquity of web enabled mobile devices continue to make performance an important concern when developing JavaScript applications. We address this issue by proposing a category theoretic view of the relationship between jQuery and the Document Object Model. From that view we derive a set of alterations to the jQuery library and demonstrate the performance benefits that result. Additionally we show how the second functor law suggests a set of JavaScript functions and jQuery methods that can be optimized using loop fusion.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Enhancement; D.2.2 [*Design Tools and Techniques*]: Software libraries

General Terms Performance, Design

Keywords JavaScript, Category Theory, Loop Fusion, Optimization

1. Introduction

JavaScript that leverages the jQuery library can often be identified by its fluency. That is, users are encouraged to make alterations to jQuery objects by "chaining" methods and jQuery method authors are counseled to always return the jQuery object on which the method was called to facilitate this form of serial method invocation [2]. In Listing 1, all **HTMLDivElement**s (DOM elements hereafter) in the document are retrieved using the `"div"` CSS selector and used to instantiate a jQuery "object-set". They are then hidden, altered to remove the `data-foo` attribute, and shown again. Each method invocation, `hide`, `addClass`, and `show` alters *all* the elements in the jQuery object-set and then provides them for the next method to do the same. More concretely, if n methods of this form are invoked in sequence it will require n full iterations over the object-set. This presents an opportunity to exploit loop fusion for a possible peformance gain.

Additionally many of the methods that perform element manipulations include user friendly extra invocation patterns. Listing 2 shows two functionally equivelant examples for the `removeAttr` method. To support the second example, the method must parse the first argument in both cases to check for a whitespace delimitted list. Given that this is a common occurence in the library's element manipulation methods, this too represents an opportunity for performance improvements.

In the interest of exploiting these two opportunities for improvement we introduce the following:

1. We introduce category theory and define two novel categories **Html** and **Jqry** along with a Functor that maps from **Html** to **Jqry**. In doing so we provide a rigorous definition of the methods that can be optimized with loop fusion. We also find a clear deliniation between the **Html** morphisms and the user friendly layer that their **Jqry** counterparts add.
2. We propose a backward compatible set of alterations and additions to the jQuery library based on the separation of functionality suggested by these new categories. The result is reduced load times and faster performance for advanced users willing to sacrifice the additional user friendly features.
3. We construct a simple and unobtrusive utility for jQuery developers that will alert them to opportunities to optimize method chains and leverage the propsed API additions to improve application performance.

2. jQuery Object Methods

A short discussion of two important attributes required in the construction of jQuery methods will aid in understanding the forthcoming categories. The first is the context in which jQuery methods are expected to operate. That is, the value of `this` within jQuery methods is an instance of a jQuery object-set, a set of DOM elements.

```
//each method returns the mutated jQuery object
jQuery("div").hide().removeAttr("data-foo").show();
```

Listing 1. Sample method chain

```
// user "unfriendly"
jQuery("div")
  .removeAttr("data-foo")
  .removeAttr("data-bar");

// user friendly
jQuery("div")
  .removeAttr("data-foo data-bar");
```

Listing 2. User friendly overhead

```
 1 jQuery.fn.sampleWhileForm = function(){
 2   var length = this.length;
 3
 4   while( length-- ){
 5     var domElement = this[length];
 6
 7     // ...
 8     // alteration of domElement
 9     // ...
10   }
11 };
12
13 // invocation
14 $( "div" ).sampleWhileForm();
```
Listing 3. Sample jQuery method

Though that context can be set manually, it's most often defined by assigning the method to a property on the jQuery object prototype `jQuery.fn`. The second is the behavior of jQuery methods, in that each must extract DOM elements from the jQuery object-set to perform any meaningful work. This is generally done on either the first element or the whole set depending on the method. For the purposes of our work we are primarily concerned with those methods that opperate on the wholes set. In Listing 3 on line 1 is an example of a method that exhibits these two attributes, while line 14 shows the invocation pattern that assures the context is properly set.

The pattern is relatively simple but, as we will illustrate, it conflates two distinct types of opperations: those on DOM Elements and those on jQuery object-sets.

3. Categories

To define **Html** and **Jqry** we must define the classes $ob(\mathcal{C})$ of objects and $hom(\mathcal{C})$ of morphisms for each. Then for both we must provide the identity morphism, show that composition is possible, show that composition is associative, and finally that each set of morphisms is closed under composition [3, p. 1]. To start we will address **Html**.

We define the objects of **Html** as the set of JavaScript objects that represent HTML elements outlined in the World Wide Web Constortium's HTML5 Element specification [4]. More intuitively, the objects are the result of querying the DOM in browsers complying with the specification using a method like `querySelector` (Listing 4).

We define the morphisms of **Html** as the set of JavaScript functions taking a single DOM element JavaScript object and returning one of the same (that is any object in $ob(\mathbf{Html})$). Generally these functions manipulate the DOM element through the dot method invocation of JavaScript objects.

Next, we define the identity function for **Html** in the manner you would expect (Listing 5, line 1). The composition operation is nearly as simple, returning a new closure that will process the function execution in the expected order (See Appendix B for an example):

```
document.querySelector( "#sample" );
document.querySelectorAll( "#sample" )[0];
document.getElementById( "sample" );
```
Listing 4. Sources of Html objects

$$\begin{aligned} cmps(f, cmps(g,h))(x) &= cmps(cmps(f,g),h)(x) \\ cmps(f, g(h(x))) &= cmps(f(g), h(x)) \\ f(g(h(x))) &= f(g(h(x))) \end{aligned}$$

Figure 1. Reduction of composition

To show associativity it suffices that the reduction remains the same for different associations (Figure 1) and we know that the morphisms in **Html** are closed under composition because the source and target objects for each morphism both exist in $ob(\mathbf{Html})$

To define **Jqry** we must demonstrate the same properties. The class $ob(\mathbf{Jqry})$ has as its members all jQuery objects. The class $hom(\mathbf{Jqry})$ is all morphisms from jQuery objecst to jQuer y objects. As before we first define the identity function and then the composition operation (Listing 6).

Identity in **Jqry** is different from its **Html** counterpart in that it relys on `this` as an implicit parameter. `jQuery.cmps` leverages JavaScript's `apply` to manually define the context in which its argument functions will run, thereby simulating the context equivelant of argument passing. It is defined in the `jQuery` namespace only for the sake of differentiating it from `cmps` in later listings and it needs no context for operation. Again, we take associativity to be evident by reduction and we know that **Jqry** morphisms are closed under composition because the source and target objects are the same.

Next we define a Functor from **Html** to **Jqry** which consists of two operations. The first is the invocation of the `jQuery` function with a raw DOM element which returns a jQuery object set wrapping the element (Listing 7, line **??**). The second uses jQuery's `map` helper inside a new closure that expects a jQuery object as its execution context. `map` will pull each DOM element out of the jQuery object-set (`this`) and pass it to the callback provided as its second argument.

The Functor must also preserve identity and composition [3, p. 36]. The preservation of identity is clear. Wrapping a DOM element returned by the identity **Html** morphism with the functor's first operation produces the same result as invoking the **Jqry** identity morphism on an already wrapped DOM element (Listing 8, Line 2).

Preserving composition is more subtle. While the final results of either side of the equivelance (Listing 8 line 5) will be identical, the side effects of DOM manipulations along with the iterative nature of jQuery methods mean that the order of invocation can be significant. In the original method chain example (Listing 1) the invocation order will perform each opperation on the complete set of elements one method at a time. If it is important to the developer that all the elements be hidden before performing any operations, composition of the **Html** morphisms violates the equivelance. Al-

```
1 function id( a ) {
2   return a;
3 }
4
5 function cmps( f, g ) {
6   return function( a ) {
7     return f(g(a));
8   };
9 }
```
Listing 5. Identity and Composition in Html

$$F : \mathbf{Html} \rightarrow \mathbf{Jqry} \tag{1}$$

$$ob(\mathbf{Jqry}_h) = ob(\mathbf{Jqry}) \tag{2}$$

$$hom(\mathbf{Jqry}_h) = \{F(f) \mid f \in hom(\mathbf{Html})\} \tag{3}$$

$$hom(\mathbf{Jqry}_h) \subsetneq hom(\mathbf{Jqry}) \tag{4}$$

Figure 2. Jqry Subcategory Dependency

ternately if the developer only cares that the opperations take place on each element in the original order then the equivelance holds. We provide more detail on this in Section 5.

4. Splitting jQuery in Two

A close examination of the functor exposes an interesting relationship between **Html** and **Jqry**. Those **Jqry** morphisms that manipulate DOM elements (there are those that operate on the set itself) can always be described in terms of `jQuery.map` and an **Html** morphism. Said another way, there is a subcategory $\mathbf{Jqry}_h$ where $hom(\mathbf{Jqry}_h)$ can be defined entirely with $hom(\mathbf{Html})$ and our functor (Figure 2).

Though this relationship clearly exists in the abstract, the implementation of DOM manipulation methods in jQuery rarely exemplifies it. The code that would otherwise be defined in an **Html** morphism is most often found mixed into a **Jqry** morphism. As an example, the `jQuery.fn.removeProp` method mixes the deletion of DOM element properties with the extraction of those elements from the jQuery object-set (Listing 9, line 2). Deconstructing the mixed together methods along the lines suggested by the dependency presents an opportunity for improvement on a few fronts.

One improvement comes as a consequence of separating the **Html** morphism from each morphism in $hom(\mathbf{Jqry}_h)$ and providing them to the developer directly as a subset of the jQuery API. That is, if the developer only needs the **Html** morphism because they are willing to use the standard DOM API for element selection along with JavaScript's basic looping constructs it's possible to significantly reduce the JavaScript payload size.

[TODO] find actual payload reduction for the library or some significant subset

[TODO] find and discuss the overhead in additional file size for the existing library

In addition to a file size reduction, there are two possible performance improvements that fall out of this separation. The first is a small reduction in the overhead imposed by "rewrapping" DOM elements with the `jQuery` method which is a common idiom in jQuery applications. In the looping methods (e.g. `jQuery.map` and `jQuery.each`) and event bindings the object passed into the callback is a raw DOM element and developers often create a one element jQuery object-set to gain access to the easier-to-use DOM manipulations. Assuming that these same DOM manipulations are available, the rewrapping can be avoided thereby improving performance.

[TODO] gather performance data for wrapping overhead, no-op is faster but how much matters

The second performance improvement requires some additional clarification of the roles of both **Jqry** and **Html** morphisms. In Listing 2, `jQuery.fn.removeAttr` supports two invocation patterns. We propose that in abstracting the **Html** morphism from within methods like `jQuery.fn.removeAttr` a single, canonical invocation pattern be selected and that support for others be retained in the associated **Jqry** morphism. In the aforementioned example that would translate to support for the removal of a single attribute in the **Html** morphism. The result is a significantly less complex and more performant execution path in many cases.

[TODO] graph sample performance data from removeAttr, eventually include larger subset of data

5. Fusing Method Chains

IN PROGRESS Assuming the second equivelance holds it's useful to view and actual invocation of the functions created on both sides. The second, that leverages the more familiar chained method form (Listing 8 line **??**)

When the second Functor law holds we can say with complete confidence that two morphisms in $hom(\mathbf{Html})$ composed and then promoted into $hom(\mathbf{Jqry})$ are equivalent to the composition or chaining of two morphism in $hom(\mathbf{Jqry})$. As a performance optimization the choice of the former is simple form of loop fusion or deforestation where the intermediate data structure is the mutated jQuery object set [7]. **IN PROGRESS**

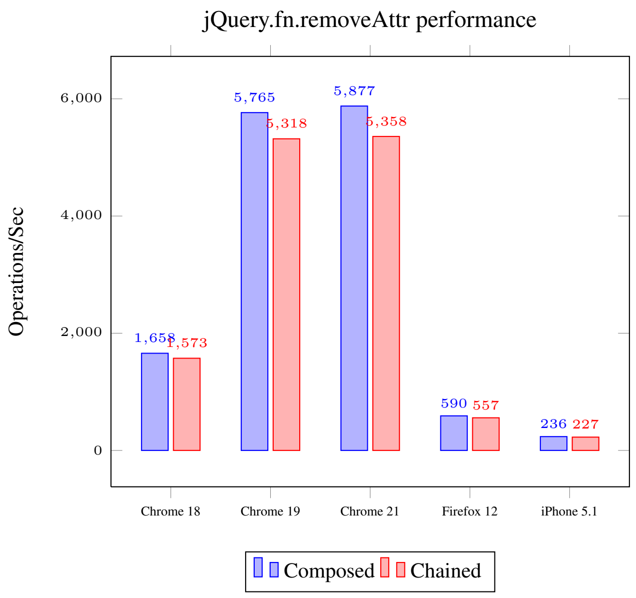
[**TODO**] Discuss the function call overhead reduction, show performance numbers from existing methods with abstracted DOM manipulations.

```
jQuery.fn.id = function() {
  return this;
};

jQuery.cmps = function( f, g ){
  return function() {
    return f.call(g.call(this));
  };
};
```

Listing 6. Identity and Composition in Jqry

```
// ob(Html) -> ob(Jqry)
jQuery( document.querySelector( "#sample" ) );

// hom(Html) -> hom(Jqry)
function functor( morphism ){
  return function(){
    return jQuery.map(this, morphism);
  };
}
```

Listing 7. Functor from Html to Jqry



6. Relaxing The Functor Definition

[**TODO**] Disucss using methods in serial instead of the cmps operation. Discuss the reality of side effects as it relates to preserving identity and composition. Discuss method chains as composition to avoid extra function call overhead which represents the real performance win here.

7. Haskell's List Functor

[**TODO**] Discuss the relationship here with fmap and haskell's list functor. jQuery's objec-set behavior is clearly a list. Cite previous work in that area as an influence.

8. Guidelines to Facilitate Optimization

Given that the loop fusion optimization requires the composition of at least two morphisms from **Html** and that functionality of those morphisms is always provided to the end user as the lifted version that exists in **Jqry**, we propose the following guidelines.

1. All methods defined on the jQuery object prototype `jQuery.fn` that leverage `jQuery.map` to lift an **Html** morphism into **Jqry** must provide the underlying morphism as a property for end users. Since the behavior of a given jQuery method is often too complex for the end user to determine if it meets the criteria for loop fusion, this rightly places the onus on the developers of the jQuery methods to determine and provide the necessary JavaScript function to the end user for optimization in their applications.
2. All properties defined for this purpose should exist on the jQuery method itself to avoid confusion among users wishing to optimize their method chains. We propose `composable`

```
// preservation of identity
jQuery(id(elem)) == jQuery(elem).id();

// preservation of composition
functor(cmps(f, g))
  == jQuery.cmps(functor(f), functor(g));
```

Listing 8. Satisfying the Functor Laws

```
// exists in hom(Jqry)
jQuery.fn.removeProp = function( name ) {
  name = jQuery.propFix[ name ] || name;

  return this.each(function( elem ) {
    elem[ name ] = undefined;
    delete elem[ name ];
  });
};
```

Listing 9. Satisfying the Functor Laws

as the property name. This guarantees that jQuery methods remain the discreet units of functionality extension that they are today, and the proposed property name is semantically useful.

3. Developers should, wherever possible, document and test both the **Html** morphism and its **Jqry** incarnation as discrete pieces of functionality to ensure that each **Html** morphism works independent of its **Jqry** counterpart. This ensures that the individual properties of both the **Jqry** and **Html** morphisms will persist across revisions to both.

To assist jQuery method developers in effecting the above we also propose a small helper function that properly applies `map` to an **Html** morphism, properly forwards arguments in addition to the initial **HTMLElement** argument, and sets the `composable` attribute. See Appendix C.

9. Library Aided Optimization

The reader will note that the guidelines do not attempt to differentiate the optimizable jQuery methods in any appreciable fashion other than the the possible existence of the `composable` property. This is a deliberate omission in preference to an automatic identification of chains with two or more methods that define the `composable` property. In fact it is possible to automatically fuse the underlying JavaScript functions but this incurs a small additional cognitive overhead and an as yet unresolved performance degradation (See Appendix D).

[**TODO**] further discuss automatic fusion and possible benefits to end users.

[**TODO**] explore the reasons for the performance degredation with the automatic approach as it may be relevant to the hand fused approach.

Instead we propose a small library that will log a warning any time two or more methods are invoked in sequence when each provides the `composable` property. Additionally, it will log a warning when two or more of these methods occur in a method chain but are not adjacent. While more detail on one possible implementation is provided in Appendix E, a short explanation here may aid interested parties in creating their own implementation.

Newly instantiated jQuery objects derive the bulk of their functionality from the `jQuery.fn` object defined as their prototype. `jQuery.fn` is also the object onto which new jQuery methods, or **Jqry** morphisms, are defined. Consequently it's possible to create a proxy object that can be inserted between a jQuery instance and `jQuery.fn` in the prototype chain at runtime to record the sequence of method invocations and report opportunities for optimization.

[**TODO**] add diagram to illustrate prototype chain alteration

Taking $\xrightarrow{f}$ to represent the automatic prototype look-up of f on the target, and $\xmapsto{f}$ to represent a invocation of f on the target by

(5) $$jQuery \xrightarrow{f} jQuery.fn$$

(6) $$jQuery \xrightarrow{f} Proxy \overset{f}{\longmapsto} jQuery.fn$$

the source object we have diagram 5 as the default jQuery behavior and diagram 6 as the desired behavior.

The $Proxy$ object must define it's own version of each and every function property of the $jQuery.fn$ prototype object. This allows it to count invocations of those functions and for any count greater than one raise a warning. It also allows it to invoke the method of the same name on the $jQuery.fn$ when no count is recorded, IE $\overset{f}{\mapsto}$. Additionally the size of the jQuery object-set can be taken into account as part of configuration, as small sets of objects won't see the same benefits from composition.

The primary advantage of this approach is that it is entirely unobtrusive and requires nothing more than the inclusion of the library in an HTML document following the inclusion of jQuery itself. In this way it encourages developer adoption through ease of use.

10. jQuery Project Results

[**TODO**] discuss standard with jQuery core team, push for implementation in core. Talk submitted to jQuery Conference in June 2012 with this goal in mind

11. Client Project Results

[**TODO**] discuss implementation with subject application creators using jQuery. Candidates: Originate Labs, The Filament Group, Append To, Bocoup, Adobe

12. Further Work

[**TODO**] discuss further work in applying other category theoretic constructs to the two categories defined here. eg, jQuery is a Monoid, examine cartesian closed categories.

13. Conclusion

Here we have clearly defined a common idiom in JavaScript using the jQuery library that can be targeted for performance optimization with a minimum of effort by developers. In addition we have established a small set of guidelines that jQuery extenders and plug-in authors can use to assist the consumers of their software in performing this optimization and provided the framework for an unobtrusive library that can automatically identify areas of potential performance degradation. In future work we hope to pursue the automatic optimization of jQuery method chains using lazy semantics to further reduce developer involvement while continuing to realize the advantages of loop fusion.

A. Appendix A: Each-form

Another equally popular form of jQuery method construction leverages the jQuery built-in `each` method. Converting the example from Figure **??** yields Figure 3. The key difference being the expectation that a side effect will result from the closure that somehow leverages the information of the index and/or the **HTMLElement**.

B. Appendix B: Composition of Html Morphisms

Examples of what composition of **Html** morphisms will look like can server to reassure the reader that it behaves as necessary. In

```
jQuery.fn.eachForm = function(){
  return this.each(function( index, htmlElement ) {
    // some side effectful computation
  });
};
```

Figure 3. General each-form

Figure 4 an anchor element has its `foo` and `baz` attributes set by the newly composed **Html** morphism.

```
function a( elem ){
  elem.setAttribute( "foo", "bar" );
  return elem;
}

function b( elem ){
  elem.setAttribute( "baz", "bak" );
  return elem;
}

var elem = document.getElementById( "example-anchor" );
elem.getAttribute( "foo" ); // undefined
elem.getAttribute( "baz" ); // undefined

elem = compose( a, b )( elem );
elem.getAttribute( "foo" ); // "bar"
elem.getAttribute( "baz" ); // "bak"
```

Figure 4. Preserving identity and composition

C. Appendix C: Mapable Helper Function

The `jQuery.mapable` helper described in Figure 5 provides a function that can be assigned to a property on the `jQuery.fn` object. It also "tags" the function by setting its `composable` attribute to the original **Html** morphism thereby alerting developers and any libraries wishing to track optimizable **Jqry** morphisms. Additionally it does the work of forwarding any and all arguments as additional parameters to the **Html** morphism.

```
jQuery.mapable = function( htmlMorphism ){
  var jqryMorphism = function(){
    var args = arguments;

    jQuery.map(this, function( elem ){
      var newArgs = Array.prototype.slice( args );
      newArgs.unshift(elem);
      htmlMorphism.apply( elem, newArgs );
    });
  };

  jqryMorphism.composable = htmlMorphism;
  return jqryMorphism;
}j
```

Figure 5. The mapable helper function

D. Appendix D

An initial attempt was made to alter jQuery to support the deferral of method execution until being forced. The idea was to accumulate method calls that defined the `composable` property (`htmlMorphism` in the source) and then do the composition and execution all at once when necessary. While this is possible in JavaScript, the performance overhead of forwarding arguments for method invocations like those in Figure 6 down to the **Html** morphism in conjunction with the capability of modern JavaScript virtual machines to compile simple loops to machine code negated any positive effect of limiting the total iterations. You can view the results of a simple performance test at jsperf.com. You can also view the extension required to effect the lazy optimization at github.com. Further work and testing is required to completely rule out the possibility of this approach

```
jQuery( "div" ).foo( "bar", "baz" ).force();
```

Figure 6. Forwarding string arguments

E. Appendix E

A basic implementation of the object proxy described in Section 9 can be found at GitHub along with a simple counting and logging mechanism for identifying possible optimizations.

References

[1] BuiltWith.com, jQuery Usage Statistics, `http://blog.builtwith.com/2011/10/31/jquery-version-and-usage-report/`

[2] jQuery.com, Plugin Authoring, Maintaining Chainability, `http://docs.jquery.com/Plugins/Authoring#Maintaining_Chainability`

[3] Benjamin C. Pierce, *Basic Category Theory for Computer Scientists*. MIT Press, Massachusets, First Edition, 1991.

[4] www.w3.org, Document Object Model (DOM) Level 3 Core Specification, `http://dev.w3.org/html5/spec/elements.html`

[5] W3.org, HTMLElement interface specification, `http://dev.w3.org/html5/spec/elements.html#htmlelement`

[6] W3.org, HTMLElement list, `http://dev.w3.org/html5/markup/elements.html#html-elements`

[7] P Wadler, *Deforestation: Transforming programs to eliminate trees*. Theoretical computer science, Elsevier, 1990.