Advanced Caterwaul Programming

Spencer Tipping

March 24, 2011

Contents

Chapter 1

Introduction

Client-Side Caterwaul introduces Caterwaul in an applied context. This guide talks about leveraging the more advanced features of Caterwaul to write rich client and server applications. I also introduce some design patterns that help keep code readable despite being very terse.

1.1 Useful libraries

Caterwaul alone, even with a growing standard library, is only so useful. As I find repeated use cases I write extension modules or libraries. At this point Caterwaul supports:

- Finite and infinite sequences and concise comprehensions (`seq`)
- Tail-call optimization and delimited continuations (`continuation`)
- Packrat parser combinators (`parser`)
- HAML-style HTML element generation (`montenegro`, an external library)

The Montenegro project[1] is briefly documented in its readme, but I'll cover its usage here by example. Now the HTML scaffolding should look like this:

```
<!doctype html>
<html>
  <head>
    <title>...</title>
    <script src='jquery.js'></script>
    <script src='caterwaul.all.js'></script>
    <script src='montenegro.client.js'></script>
    <script src='page.js'></script>
```

[1] http://github.com/spencertipping/montenegro

```
    <link rel='stylesheet' href='...' />
  </head>
  <body></body>
</html>
```

`page.js` is the page driver, and that's where all of the logic will end up.

Chapter 2

Creating Markup

Montenegro was written to integrate markup creation into the page logic. Before you vote to have me excommunicated for suggesting that this is a good thing, let's consider the simplicity it offers.

2.1 Functional UI modeling

Caterwaul includes macros to make function definitions very lightweight in order to encourage abstraction where appropriate. One of the places it's most helpful is factoring out the implementation of a particular UI element. So, for example, if we want a box asking for someone's name and e-mail address, we could do it this way:

```
$('body').append(html[table(
  tr(td('Name: '), td(input.name)),
  tr(td('Email: '), td(input.email)))]);
```

But if we're going to be creating many of them, it's better to factor it out:

```
$('body').append(name_and_email()),
where*[
  name_and_email() = html[table[row_for('name'), row_for('email')]],
  row_for(field)   = html[tr(td['#{field}: '], td(input /addClass(field)))]];
```

> **N.B.** This code illustrates a design pattern I've found to be very useful: *one concern per line*. As I'll discuss later on, Caterwaul code can be very terse, and it's often useful to leverage that terseness without creating spaghetti code.

2.2 Side-effects

Normally in functional style you want to use as few side-effects as possible and have each function be pure. However, this breaks down when you have to work with continuations. The classic example is loading some content via AJAX:

```
// side-effect at toplevel
name_and_email(fn[element][$('body').append(element)]),
where*[
  name_and_email(cc) = l/cps[data <- $.getJSON('...', _)] in
                       cc(name_and_email_helper(data)),
  name_and_email_helper(data) = html[table[row_for('name'), row_for('email')]],
  row_for(field) = /* same as before */];
```

The problem here is obvious; we haven't really gotten rid of the side-effect, we've just pushed it up to the toplevel. More useful is to push the side-effect into the abstraction so that we have the usual behavior from `name_and_email`. Here's how to do it:

```
// side-effect encapsulated inside name_and_email()
$('body').append(name_and_email()),
where*[
  name_and_email() = html[div] /se[populate(_)],
  populate(div)    = l/cps[data <- $.getJSON('...', _)] in
                     div.append(html[table[row_for('name'), row_for('email')]]),
  row_for(field)   = /* same as before */];
```

Chapter 3

Sequences

`seq` is probably my favorite Caterwaul library. It can reduce complex map/-fold/filter expressions down to a single line and can represent most if not all of the cases where bulkier constructs such as `while` and `for` would normally be used.

3.1 The `seq[]` macro

As with most macros in Caterwaul, `seq[]` creates a context in which operators are reinterpreted. However, none of the functionality exposed by `seq[]` is created by the macro; all generated code refers to public methods of the sequence classes. For example:

```
seq[xs *[_ + 1]]   ->   xs.map(fn[_, _i][_ + 1])
seq[xs %[_]]       ->   xs.filter(fn[_, _i][_])
seq[xs *~[_ *[_]]] ->   xs.map(fn[_, _i][_.map(fn[_, _i][_])])
```

All of the transformations are documented in the source code.[1] Operators apply with their usual precedence, which generally is intuitive except in the case of flat-map, represented by `-`.[2]

I'll explain the semantics of the operations encoded by various operators as I go. It will probably be handy to have the source code available, and perhaps also the Caterwaul live compiler.[3]

[1] http://github.com/spencertipping/caterwaul/blob/master/modules/caterwaul.seq.js

[2] Sadly, I ran out of multiplication-precedence operators and had to go down one level. It's unfortunate, but still usable.

[3] http://spencertipping.com/caterwaul/compiler

3.2 Use case: A list of resources

Often in AJAX applications there's some list of stuff that needs to be displayed to the user. For simplicity, let's assume that we have an index of things, each of which can be clicked on to view the whole object. In straight jQuery/HTML a common way to code this would be (preserving the functional-DOM pattern):

```
var a_thing = function (name) {
  return $('<a>').text(name).attr('href', 'javascript:void(0)').
                             click(open_page_for(name));
};
var list_of_things = function () {
  var result = $('<div>');
  $.getJSON('/index', function (items) {
    for (var i = 0, l = items.length; i < l; ++i)
      result.append(a_thing(items[i]));
  });
  return result;
};
$('body').append(list_of_things());
```

This Caterwaul code does the same thing:

```
$('body').append(list_of_things()),
where*[
  list_of_things() = html[div] /se[add_things_to(_)],
  add_things_to(x) = l/cps[things <- $.getJSON('/index', _)] in
                     seq[~things *+a_thing *![x.append(_)]],
  a_thing(name)    = html[a[name] *href('javascript:void(0)')
                                  /click(open_page_for(name))]];
```

We saved the most space with `l/cps` and `seq[]`. Here's how to read the `add_things_to` function:

> Pull *things* from a `getJSON` call using a callback. `l/cps` stands for 'let-in-continuation-passing-style', so it binds a variable when the continuation gets invoked.
>
> ```
> l/cps[things <- $.getJSON('/index', _)] in
> ```
>
> Promote the array *things* into a Caterwaul finite sequence. Then map each one through the function `a_thing`. For each element in this new sequence, append it to `x`.
>
> ```
> seq[~things *+a_thing *![x.append(_)]]
> ```