Functional Programming & DSL’s Part 3

FEBRUARY 17, 2014 MICHAEL CROMWELL

Tidying up our DSL

In the last part we added filtering of results via the where function and showed how we can compose functions to enable expressive and re-usable criteria but we were left with an somewhat awkward way of calling our functions together, what we need is a function to stitch them together.

Creating query

The 2 functions we have created so far (select & where) when executed return another function that then accepts an array of objects (in our example this is user objects) essentially this allows us to configure how we want the functions to behave before sending them the actual users, this is good because it means that we can treat each function as abiding by the same contract that it will accept an array of objects and return an array of objects so our query function simply needs to call each function in turn and pass the results from one to the next[1].

```
//+ query :: ([a], [(fun [a] -> [b])]) -> [b]
var query = function (items /*, funs */) {
  var funs = _.rest(arguments);
  return _.reduce(funs, function (results, fun) {
    return fun(results);
  }, items);
};
```

The first argument is our array of objects, after that is the functions that will be called with the array of objects, we are using the _.reduce function like we did in the all function in the previous part to call each function in turn and capture the results for every call to feed into the next, this can then be called:

```
query(users,
      where (all (female, overEighteen)),
      select (id, dob, gender));
//=> [Object]
```

This is a lot easier to consume and is very close to our ideal DSL, there is still some noise which would be great to get rid of namely parentheses and the semi-colon at the end, lets see how we can improve this.

Enter CoffeeScript

CoffeeScript is a language that transcompiles to JS it has a ton of various features and also cuts down massively on the amount of noise needed compared to standard JS (at present[2]), we can write our expression above as:

```
query users,
      where(all female, overEighteen)
      select id, dob, gender
//=> [Object]
```
I could almost get rid of all the parentheses but found that if I removed it from the where it short circuited and did not execute the remaining functions, I think this is because it struggles to parse on the all function, so you could rewrite it to this if were really keen:

```
criteria = all female, overEighteen
query users,
      where criteria
      select id, dob, gender
//=> [Object]
```
This is probably as close as we can get to our ideal DSL for querying and I think it’s pretty damn close!

```
var results = query users
              where all female, overEighteen
              select id, dob, gender
```
Extending our DSL
Now we have our building blocks we can start to extend the language of our DSL in this example I’m going to demonstrate how we can add ordering into our query, so we should then be able to write the following:

```
query users,
      where(all male, overEighteen)
      orderBy dob
      select id, dob, gender
//=> [Fri Aug 23 1957 00:00:00 GMT+0100 (GMT Daylight Time), Mon Apr 02 1979 00:00:00 GMT+0100 (GMT Daylight Time), Wed Feb 01 1984 00:00:00 GMT+0000 (GMT Standard Time)]
```

To implement orderBy we can use the _.sortBy function which takes in an array of objects and can either sort on a provided function or on a string name for a property we can use the second approach for this and reuse our functions we use for the select function that return us hardcoded strings:

```
var sortBy = reverseArgs(_.sortBy);

//+ orderBy :: (fun -> string) -> [a] -> [a]
var orderBy = function (prop) {
  return _.partial(sortBy, prop());
};
```
We partially apply over the sortBy function and pass it the returned string from the call to the prop function supplied.

Closing Thoughts
I hope this has been a helpful demonstration of how we can use FP with JS (and some CS to get a cleaner syntax) to enable creation of DSL's I know some may be thinking that this seems like quite a bit of work to get working however I think the following need to be taken into account:

A lot of the functions we ended it up writing were general purpose and will either be re-used in other areas or already be provided for in other FP libraries functions like reverseArgs would not even be needed if underscore had it's arguments geared up for partial application and prop are going to be useful in lots of other places.
Like I demonstrated in part 1 this query DSL is not specific to any particular object type and can be applied to any objects this is in contrast to OOP which tends to be built with specific types to work against
The actual code needed when lumped together is 84 lines which is not a great amount[3] considering the nice maintainable DSL that your left with
The final code can be found on jsfiddle

You could also call this function pipeline but query fits closer to the querying domain
If you look at what's in the pipeline for JS a lot of the features CS offers will be added directly to JS, also after spending time looking at FP with JS some of the features offered by CS become less in demand (array comprehensions for example if you have map)
I would imagine using some other FP libraries and having someone more experience with FP than I'am you could probably halve this ☺