FUNCTIONAL PROGRAMMING / JAVASCRIPT

Functional Programming & DSL’s Part 2

FEBRUARY 10, 2014 MICHAEL CROMWELL

Previously…

We managed to get a select function that enabled us to work towards our ideal querying DSL, we also found that although in this example we are using user objects the select function can be used against any object type and demonstrates that in FP a lot more emphasis is put on functions working against specific data structures and that the objects are not associated with particular types but become simple datasets.

Moving onto where

The next step is to see how we can restrict results, if we re-acquaint ourselves with our ideal DSL for querying it looked like this:

```
var results = query users
              where all male, overEighteen
              select id, dob, gender
```

To start with if we simplify the where function so that it deals with a single predicate[1] we end up with:

```
//+ where :: (fun a -> boolean) -> [a] -> [a]
var where = function (pred) {
  return function (items) {
    return _.filter(items, pred);
  };
};
```

We utilise the _.filter function which as you can guess takes an array of items and then a predicate and will return a new array of the items that pass the predicate supplied, as with the other underscore functions we are having to write more code because of the argument ordering, I will use the reverserArgs function I created in part 1 to solve this:

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

var where = function (pred) {
  return _.partial(filter, pred);
};
```

Our new where can partially apply over the predicate supplied and then we just need the items to be provided. This can now be called like so:

```
var male = where(function (u) { return u.gender === 'm'; });
_.first(male(users));
//=> {id: 1, username: "jbloggs", dob: Wed Feb 01 1984 00:00:00 GMT+0000 (GMT Standard Time), displayName: "Joe Bloggs", gender: "m"…}
```

This works well for a single predicate however we want to be able to specify multiple predicates to filter on any number of criteria and this is were all comes into play:

```
//+ all :: [(fun a -> boolean)] -> a -> boolean
var all = function (/* preds */) {
  var preds = _.toArray(arguments);
  return function (item) {
    return _.reduce(preds, function (result, next) {
      return result && next(item);
    }, true);
  };
};
```

It accepts a number of predicates that will be used to test an individual item against
A closure is returned that accepts a single item
We use the _.reduce function to enumerate our predicates the first thing we check is that we are still returning true for the current item and if so we check the item with the next predicate[2]
The end result will either be true or false depending if the item met all the predicate conditions or not
I have a feeling there is a better way this can be expressed in a more succinct way and would be happy if anyone can demonstrate this in the comments ☺ Let’s take it for a test drive:

```
var male = function (u) { return u.gender === 'm'; };
var firstnameIsJuan = function (u) { return (/^juan/i).test(u.displayName); };
var criteria = all (male, firstnameIsJuan);
```

```
_.map(users, criteria);
//=> [false, false, false, true, false]
```

Although a bit convoluted it does demonstrate that the 3rd user in our array is correctly being identified as being male and with a first name of ‘Juan’, because this conforms to a predicate we can now plug it into our where function:

```
var results = where(all (male, firstnameIsJuan));
_.first(results(users));
//=> {id: 4, username: "jfranco", dob: Mon Apr 02 1979 00:00:00 GMT+0100 (GMT Daylight Time), displayName: "Juan Franco", gender: "m"…}
```

The results have 1 item that is as expected our 3rd entry in the users array.

Composing Expressions

One of the nice aspects of this approach is how you can compose together expressions to make them fit the problems we are trying to solve in a nice readable way and also enable re-use, say for starters we have some functions already written to check values:

```
//+ isMale :: string -> boolean
var isMale = function (gender) { return gender === 'm'; };

//+ checkAgeAgainstDate :: (number, date) -> boolean
var checkAgeAgainstDate = function (age, date) {
  return new Date(
          new Date().setFullYear(
              new Date().getFullYear() - age)) > date;
};
```

Now at the moment these functions take in primitive values we are using user objects that have these primitive values inside them, our first try at re-using the functions above may look like this:

```
//+ male :: a -> boolean
var male = function (u) {
  return isMale(u.gender);
};

//+ overEighteen :: a -> boolean
var overEighteen = function (u) {
  return checkAgeAgainstDate(18, u.dob);
};
```

This works but were having to create brand new functions to wrap up the call to the helper functions, notice that in both cases all I’m doing is extracting the correct property from the user object, if I can wrap this action up in a function I can then compose this new function against the corresponding helper function:

```
//+ prop :: (string, a) -> b
var prop = function (prop, obj) { return obj[prop]; };
```

It actually ends up being a really simple function thanks to the dynamic nature of JS, we can now create some functions that partially apply over the correct property:

```
//+ getDob :: a -> date
var getDob = _.partial(prop, "dob");

//+ getGender :: a -> string
var getGender = _.partial(prop, "gender");
```

I have used a get* prefix so as not to confuse with the existing functions we use for selecting the properties we want in the select function we saw in part 1[3], Now we use these to compose against:

```
//+ male :: a -> boolean
var male = _.compose(isMale, getGender);

//+ overEighteen :: a -> boolean
var overEighteen = _.compose(_.partial(checkAgeAgainstDate, 18), getDob);
```

The _.compose function reads from right to left and makes a new function as you’d expect that will call pass getGender into the isMale function (i.e. equiv. to male(isMale(getGender(user)))[4] the overEighteen function is a little different as I’m actually using partial application again to bind the checkAgeAgainstDate function to use 18 for the age argument I have inlined it here but if you were going to use an over 18 check in other places you could assign it to a new function.

Another example of were composition comes in handy can be demonstrated if we want to return all the female users, typically this would be the approach:

```
//+ isFemale :: string -> boolean
var isFemale = function (gender) { return gender !== 'm'; };
```
However we can utilise the existing isMale function an isFemale function is basically the opposite of the isMale function:

```
//+ not :: boolean -> boolean
var not = function (val) { return !val; };

//+ isFemale :: string -> boolean
var isFemale = _.compose(not, isMale);
```
First we define a very handy function not that will return the opposite of any value passed to it then using this we can compose the isMale function against not, we can now define and use our female function for querying against:

```
//+ female :: a -> boolean
var female = _.compose(isFemale, getGender);
var results = where(all (female));
results(users).length;
//=> 2
```
We now have our where and select functions available but to use them at the moment is quite cumbersome having to use _.compose:

```
var query = _.compose(select(id, dob), where(all (male)));
```
In the next part we will look at how to stitch them together and look at how we can get a nice looking DSL.

A predicate in this context is a function that takes a value and returns true or false depending if it meets a certain criteria
This allows us to short circuit the predicate check using the && operator
I could have also scoped them into an object:

```
var userprops = {
  dob: _.partial(prop, "dob"),
  gender: _.partial(prop, "gender")
}
var male = _.compose(isMale, userprops.gender);
```

If you want to find out more about compose I did a post on it and also it’s cousin sequence