CSE 382: Week 05

The Chaining, Currying, and Partial Completion Patterns

::

> Currying is an incredibly useful technique...It allows you to generate a library of small, easily configured functions that behave consistently, are quick to use, and that can be understood when reading your code..
>
> - M. David Green

The Anti-Pattern of Ugliness

Why is some code ugly? There are a lot reasons...to many to describe. It's easier to describe what is meant by beautiful code.

Fig. 1 - Principles of Beautiful Code

Beautiful code is;

- Distilled. There is nothing extraneous or verbose.
- Balanced. Calculations are distributed between deeply thought out and clearly articulated calculation units.
- Harmonious. The calculation units unite in the reader's mind without cognitive conflict to describe how to accomplish the overall task.
- Audacious and Original. Beautiful code surprises the reader by showing new ways to think about a problem.
- Full of Imagery. Words and names are selected to clearly, and with impact, inform the reader what they represent.
- Structured. The location of the code and its design is as important as what the code says.
- Compassionate. Reading the code is a fun and easy undertaking.

Given this definition, beautiful code is easier to read and comprehend, therefore it is easier to debug. For this same reason, it is also easier to extend. Anything that doesn't match this definition might not be outright ugly, but should be improved if for no other reason that to make supporting the code easier.

Now consider a situation similar to, but not the same as, the map-filter-reduce situation described in the CSE 121E or CSE 121A course reading. Now think about a new situation. You have been asked to map, then filter, and then reduce some list. Maybe you've been asked to work with a list of numbers, double them all, keep only those that are greater than 50, and then add up all that remain. If you use the map, filter, and reduce BIF's in a raw way, the naive version of the code to do what you were asked to write looks like this.

$$reduce :: (filter :: (map :: [a]\ f_1 \rightarrow [b])\ f_2 \rightarrow [c])\ f_3\ \rightarrow d$$

This code is not harmonious. To understand it, you eventually have to realize that you need to read it from the inside to the outside using order of operations as a guide. That time before you have that realization is cognitive conflict and is, therefore, not harmonious. For this same inside-to-outside reason is also why it is not structured well, violating another principle of beautiful code.

While those two reasons are enough to make this code ugly, the problems don't end there. The code is also verbose. Being verbose, it is not distilled. Look at all of those duplicated list notations. Each is just a variation of the last. Why, then, do they have to be there? There is also the name of each function that needs to be used. Again, why?

Is the code compassionate? Not even close. Is it audacious? Nope. It's very pedestrian. If you think about this code snippet some more, you'll find it violates most, if not all, of the principles of beautiful code.

The type of code writing exhibited by the above code is evidence that the programmer doesn't understand the possibilities offered by functional programming languages and those that are functional-like. Is there, then, another way?

Achieving Beauty

Chaining

When considering how to reduce some of the ugliness found in code around the world, it is vital, with any language type, to know and understand all the options that the type of language offers. It is true that functional programming languages, higher-order functions can have a parameter that is a lambda. It is also true that higher-order functions can have a lambda as their value. This may seem strange. Why would such a thing as having a lambda as the value of a function be useful?

Take a look at the ugly `map`, `filter` , `reduce` code snippet above. It would be much improved if the order of the calls to `map`, `filter`, and `reduce` were listed from left to right instead of inside to outside. You can do this if you put the calls to the functions into a chain, a series of calls functions to be done.

If we take another look at how the `map-filter-reduce` example above could be written using chaining, beauty emerges. To make this happen, we need to create a list that we choose to call `links` where each lambda to be applied to is a link and each link is in the desired order of execution. But before we express `chain` using pseudocode let's express it in mathematical notation. Doing so, it would look like this $[f_1, f_2, f_3]$.

Chaining - the process of executing a series of functions where the output of one is used as the input for the next.

Fig. 2

Consider a situation where you have a series of numbers and need to double them, keep only those results which are greater than 200, and then sum them. You can put together a list of functions or lambdas that apply a mapping that does the doubling, a filtering that keeps only those greater than 200, and then a reduction that sums those that are left. Then what is needed is a function to execute each of those functions. Let's call that function `chain` and pass it the links and the data to act on. The code below shows `chain` as a facade function with `chain_worker` being the function that does the actual computation.

$$-spec\ chain\ ::\ [\lambda]\ a \rightarrow b$$

$$\begin{aligned} chain\ &::\ []\ a \rightarrow\ a; \\ chain\ &::\ [h \mid t]\ a \rightarrow \\ &\quad h\ (chain\ t\ a). \end{aligned}$$

Notice that the execution of each lambda link is delayed until the lambda function after it in the chain list completes. Also note that a and b in the pseudocode can be of the same type or their types can differ. All that matters is that they, a and b, are any valid type for what ever language you are using.

While the use of this pattern does lend itself to producing code that is more beautiful than the original inside-to-outside `map-filter-reduce` example, it is still not as beautiful as it could be. It still uses more lines of code than necessary and is not very audacious. Therefore, it could be viewed as an anti-pattern in functional design. There is a much better option, using your language's

`foldl` BIF equivalent. Using this BIF, all of the code for the `chain` function goes away. It isn't needed. The single line of code below replaces it.

$$foldl\ \ (\lambda\ link\ value \rightarrow \\ link\ value))\ initial\_value\ links$$

Starting with the right-most parameter, there is a list of lambdas to apply in the order they should be executed, the $links$. Left of the $links$ list is a parameter that is the initial value that is gradually modified by the links to produce a final result. The value can be any valid type. It could be a list of valid types. The value could also be any tuple or other data structure containing the initial data.

The first parameter, $foldl$ only has three, is a lambda that has two of its own parameters. The first parameter is a link from the $links$ chain and the second is the result of applying previous links. The value of the lambda is the result of applying the lambda to the link. This lambda is not a member of the chain. It is one you write.

Chaining is a very useful pattern when a series of functions or lambdas can be considered as a single computational unit. Think of all the functions you've written that call a series of other functions. How many of those could be replaced with a call to $chain$? Probably a lot of them.

Sometimes the opposite of combining is needed. Then you use something called currying.

Currying

Currying, in Computer Science and Mathematics, is named after the mathematician Haskell Curry and he based this part of his work on what was previously accomplished by Moses Schönfinkel and Gottlob Frege.

Currying - the process of breaking up a function that has n parameters into n functions that have 1 parameter.

Fig. 3

The basic idea behind currying is to map a function that has multiple parameters to a series of functions that each have only one. Then, by executing each of the functions, the same result can be achieved. For example, you could think of the mathematical function $f(a, b)$ as being the same as $f(a)(b)$ if $f(a)$ returned a function that was then passed the parameter b. What you are really doing here, is storing parameters that you do have while you wait for those you don't have yet.

It may seem strange that a function could or should return a function, but that's one of the principles that allows functional programming languages to deeply leverage lambdas and produce beautiful code. Functions, in a functional programming language, are first-class citizens. Or said differently, you can treat a function like anything other type in the language. It can be stored in a variable. It be passed as a parameter. It can be returned from another function. When a function returns a function, this is an example of the Factory Pattern.

One of the very beautiful things about currying in general, is how it can more closely map a function requiring multiple parameters to a real world situation. Consider how people interact with an ATM (cash machine). First they enter a card, then a pin, and finally they indicate what transaction type they want. These three pieces of data could be acted on by a single function after the data has been collected. This is traditionally what you would see in code and the function could be something like $perform\_transaction\ card\_number\ pin\ transaction\_id \rightarrow boolean$. But now you have to write more code that does all the waiting. You also have to do much more thinking about how to keep the state of the transction being done correctly. Why? Because you have to do all of that while you are gathering the data.

If you used currying instead of this monolithic function, each lambda returned from a function could have its own 'wait until' the right thing happens before returning. Now the waiting and state management is done where the result of the waiting and state management have an effect. There is no need to search through the code to find where the waiting and state management are happening. The currying done in this example makes the code much easier to debug.

As in this example, when you are creating a new function and know that there will be times when only some of the parameters are known and others will be known later, you can create your function curried. You can also do this when there are more than a few parameters and currying will make the use of your function more beautiful. You are not limited to currying when creating functions. You can curry an existing function from any library, custom or language standard, using a function that generates the intermediate lambda functions for you. The pseudocode for such a function is below.

In the pseudocode you will find a mystery function called *count* with one parameter, a function. This represents the tools your language makes available to you so you can find out the arity of a function. Another mystery function is $list\_to\_params$. This represents the tools your language makes available, if any, to convert a list to a bunch of parameters.

```
-spec curry :: f -> λ

curry :: f ->
        curry_helper f(count f) [].


-spec curry_helper :: f a ::: Integer accum ::: [b] -> λ

curry_helper :: f 0 accum ->
                f (reverse(list_to_params accum));
curry_helper :: f count accum ->
                λ :: parameter ->
                    curry_helper f count - 1 [parameter] ++ accum.
```

Now be careful and be wise. Remember that your code must be beautiful. If you start using currying when other choices would be better, your code will quickly digress into ugliness. Use currying when it adds beauty.

Partial Completion

There are times when we need to front-load some computation so that the data that is the result of that computation can be used repeatedly. One way to do this is to apply the principle of partial completion.

Partial Completion - the principle of breaking up a function that has n parameters into m functions that have fewer parameters, occasionally this is n functions with 1 parameter.

Fig. 4

When you have a major process that your code needs to complete, it is common to need to reuse the results of a portion of the process repeatedly. For example, your code may need to load a large data set and do one or more mappings. Then later, dependent on other calculations or input, do some filtering and sorting.

As an example, if the data set represented user information for a large social networking company, after doing a bunch of data mapping, there often is a need to accumulate the data in multiple ways. This means you would need to apply different filtering and reducing lambdas for each different accumulation. It would be a waste to reload and re-map the data. It would be much better to reuse the mapped data.

There are multiple ways to deal with this situation. One would be to store the mapped data in a database of some sort. That would then require the data to be reloaded. That's going to be slow. The same would happen if you were writing and reading to disk.

Another option could be to store the data in a variable and then create a series of functions, each of which executed a different set of filters and reductions, and has the variable passed to them as a parameter. That's a lot of code to maintain.

There is a more beautiful way. You can create one function that does the mappings and has as its value a lambda, a, which waits for another lambda to use in a filter. When a is passed some lambda it performs the filter operation and has as its value another lambda waiting for yet another lambda, maybe some more filter, mapping, or reduction lambdas.

Please don't misunderstand me. Partial completion is not limited to situations that use map, filter, or reduce. You are limited only by your imagination on how you can apply this pattern when you are designing your code. It is situationally dependent. What you choose to include at each step of the way is up to you. Just make sure you are increasing the beauty of your code, not making it uglier.

Consider this situation. There is a csv file containing data in a table. You need to get statistical information from various columns of this table. There are times when you need data from one column, and times when you need data from other columns. All of this within a single execution of your application.

You could write a function that has as parameters the file to load, which column you want the data from, and a lambda that performs all of the calculations to generate the statistical information needed from that column. Unfortunately, this approach will mean reloading and re-preparing the data each time you access a column.

As an alternative, you could load and prepare the data and then assign this to a variable. That means you need to pass the variable around to all of the places it is needed, and write a bunch of code, in each place, to indicate the column containing the data and code to calculate the statistics needed. To make this somewhat nicer, you could write a function for each of the columns and statistics needed from that column. Unfortunately, you will either write just a few functions to cover just a few situations or you will need to write a function for all possible combinations of columns and statistics needed. Ouch! That's a lot of functions.

If you were to use partial completion, you could write a function, let's call this one `load_and_prep`, that loads and prepares the data and has as its value a lambda waiting for which column to work with. Let's call that the `column lambda`. The `column lambda` could have as its value a lambda, we'll call this one the `stats lambda`, that performs the statistical calculations you want done. Now you've covered all possible combinations, even those you haven't thought of yet. Using this setup would look like this.

$$data\_\lambda \leftarrow load\_and\_prep\ 'nation\_wide\_purchases.csv'$$

Later, when conditional choices had been made due to further data gathering and calculation, you could decide which column or columns you need to manipulate. Such a use could look like this.

$$columns\_\lambda \leftarrow data\_lambda\ [zip\ region\ state\ amount]$$

Even later, you could apply some lambda to get the statistics you want. You could even apply different lambdas depending on the specific situation.

$$stats\_by\_zip \leftarrow\ columns\_\lambda\ stats\_\lambda$$

It is also possible, if you know all of these elements at the same time to apply them to the loaded and prepared data. For example,

$$stats\_by\_zip \leftarrow load\_and\_prep\ 'nation\_wide\_purchases.csv'\ [zip, region, state, amount]\ \lambda$$

.

Partial completion gives you the most flexibility with the least complicatedness in these types of situations. Remember, use it when it will increase the beauty of your code.