Optics for generic declarative server APIs

Andre Videla
University of Strathclyde

1 Introduction

A considerable amount of engineering effort is continuously deployed to implement and manage servers, and not enough research has been done to improve the experience of creating and maintaining servers. Traditionally, they are implemented by parsing a request and performing the action that corresponds to the functionality advertised by the documentation. We make progress in this area by providing a new *purely declarative* library for servers that avoids errors associated with the tight coupling that emerges from imperative solutions. We use Idris2[2], dependent types, and lenses to achieve this goal and demonstrate how to implement and extend server APIs with as little code as possible.

2 Declarative frameworks

A common tool for the modern software engineer working on a server is a framework to define and document its API, we call those *delcarative frameworks*. Ther differ from the traditional approach which consists in implementing the server as a program that parses requests and sends responses. A declarative framework only *describes* incoming and outgoing requests, with no bearing on the implementation. Contemporary declarative frameworks for servers include *Swagger*[1], a very popular choice in the industry, and Haskell's *Servant*[4], the state of the art in declarative server libraries. Swagger allows the programmer to define APIs in an external program, and then generates documentation and source code for the API. Servant takes a more embedded approach by defining the server's APIs as a *type* in Haskell and then requesting the programmer to provide functions that implement the given API. We take a similar approach to Servant but make use of dependent types to unlock new abstractions through the relationship between endpoints and lenses.

3 Server endpoints as resources

A web server can be characterised by its *endpoints*, each endpoint describes what kind of input an HTTP[3] request provides and what kind of response is sent from the server. For our purposes, we are going to identify two operations: *querying* the state of the server, and *updating* the state of the server. We use HTTP *verbs* to declare what kind of operation the request is performing, typically a `GET` request would query the state, while a `POST` request would update the state - while carrying data through a *request body*.

Now, imagine the endpoint returns a list of todo items for a given user: `GET todo/:user` (`:user` is syntax for a url capture). Using our library we can define this endpoint like so : `"todo" / Cap User / Returns (List Todo) Get`. In this one line we've defined the url path (`"todo"`), the url capture (`Cap User` for `:user`), which kind of request we're expecting (`GET` or `POST`), and what kind of response the client expects (a `List Todo`). We require the user to provide an implementation for this endpoint. The client will see this endpoint as a function `User -> List Todo`. But attempting to implement this function as the server will result in unavoidable failure, as there is no way to *lookup* which `List Todo` to return. The intent of the server is to provide a view of its internal state, in our particular case, the state is a dictionary storing `User`s as keys and `List Todo` as values. In Idris this type is called `Map User (List Todo)`. Implementing our server while using its state amounts to implementing the function with the following type signature : `User -> Map User (List Todo) -> List Todo`.

In addition to queries, servers also perform state update operations. We represent these with a `POST` request which carries a request *body*. For example, an endpoint that adds a new todo item for a given user looks like this `"todo" / Cap User / Returns (List Todo) (Post Todo)`. Again we can deduce the corresponding function that implements this endpoint using our state: `User -> Todo -> Map User (List Todo) -> (List Todo, Map User (List Todo))`

The new state is paired up with the response of the server in order for the library to update its internal state. It is extremely common for servers to provide an API that allows both *querying* and *updating* the state. In our library we call those *resources* and we can refactor the previous two endpoints into a single one: `"todo" / Cap User / Resource Todo (List Todo)`.

To understand what how a `Resource` is implemented, it is informative to look at the type that describes APIs:

```
data Path : Type where
  Ends : (returnType : Type) ->
         Show returnType =>
         (v : Verb) -> Path
  Plain : String -> (ps : Path) -> Path
  Capture : (name : String) ->
            (t : Type) -> HasParser t =>
            (ps : Path) -> Path
  Split : List Path -> Path
```

A `Path` is a tree of path components which are either `Type`s or `String`s, they can also embed a list of `Path`s which describes endpoints with a common prefix.

Given this definition, a `Resource` is nothing but an alias for `Split [Ends ret Get, Ends ret (Post val)]` it reprents two endpoints that terminate with a POST and GET request.

4 Resources as lenses

In our previous example, we defined our server as a pair of two endpoints that query and update the same resource. If we place the two generated signatures side by side we recognise a well-known pattern:

```
GET  : User -> Map User (List Todo) -> List Todo
POST : User -> Todo -> Map User (List Todo) ->
            (List Todo, Map User (List Todo))
```

If we abstract over the state and pack up our arguments in pairs we obtain the less verbose signature:

```
GET  : (args, state) -> resource
POST : (args, state) -> newValue ->
            (newResource, state)
```

Which resembles the traditional definition of lenses[5]! As a reminder, a lens is a pair of functions parameterised over the types a, b, s, t:

```
get : s -> a
set : s -> b -> t
```

That is, a resource r with state st, arguments args, updated value nv, and a feedback value fv, is a lens:

```
set : (args, st) -> r
get : (args, st) -> nv -> (fv, st)
```

The signature of this lens is our API and its implementation is the implementation of our server. This guides us toward our first key observation:

> Every resource exposed by a server gives rise to a lens.

5 API definitions using lenses

This result provides us with a new way to define web servers and their API: as the composition of lenses. A server is but a way to *view* and *update* a resource. In the following example, we define an API for a home server that manages lights in a home.

```
lights : Path
lights = "lights" / Resource (Double,Double)
```

This API features an endpoint `lights` which control the intensity of two lights in the house using a `Double` However the current API forces us to update *both lights* in one go. We would like to create endpoints like `lights/kitchen` and `lights/bedroom` to change the intensity of each light. For this we are going to compose our `lights` resource with two lenses, one for each element of the pair:

```
kitchen = lights ~/ "kitchen" / Lens lensFst
bedroom = lights ~/ "bedroom" / Lens lensSnd
```

Where `lensFst` and `lensSnd` are the lenses accessing the first and second element of a pair. The `~/` operator replaces the `Ends` constructor by a new path using the given path component as string (here `kitchen` and `bedroom`) and matching lens for the previous return type.

We observe that those endpoints do not require any additional implementation: Their implementation is given by their lenses. This leads to our second key statement:

> A server consists of a resource type, paired with paths and lenses focusing on different parts of that resource

This statement is a bit less powerful than the previous one because each lens also needs to be augmented with a url path that uniquely identifies the resource as an endpoint. However, we now have a strategy to extend, refactor, and implement any endpoint without writing any more code than necessary. It is enough to describe our main *resource* and to provide lenses that focus on specific parts of it.

6 Conclusion

Compared to Swagger our approach remains entirely *inside* the language, ensuring that the code and its API do not get out of sync. Compared to Servant, having access to first-class types enables various forms of meta-programming. For example one can:

- Manipulate endpoints as values, for example operating on each path component, or generating an endpoint from a string.
- Manipulate APIs at runtime and serve them using the implementation of their lenses.
- Enforce API properties, for example statically checking that the server has no overlapping endpoints.

Some of those advancements have been implemented in the *Servis*[6] project. Servis' primary goal was to fix some of the limitations of Servant due to Haskell not featuring dependent types. Our library improves on Servis by exposing the relationship between endpoints and lenses.

Our library does not say anything about concurrency, two unrelated endpoints should be accessible concurrently, but because of our implementation of state, we cannot serve multiple clients at the same time. What's more, servers do not rely on their state being stored in memory, but use databases to store and fetch data. An obvious improvement would be to explain the relationship between the server's API and the database(s) it communicates with. Finally, error handling is an important aspect of API design and our lens abstraction does not say anything about it. It would be useful to talk about those future developments with the community and explore the place of dependent types in backend software engineering.

References

[1] [n.d.]. . SmartBear. https://swagger.io

[2] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. *arXiv:2104.00480 [cs]* (Apr 2021). http://arxiv.org/abs/2104.00480 arXiv: 2104.00480.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1.* Number RFC2616. RFC2616 pages. https://doi.org/10.17487/rfc2616

[4] Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. 2015. Type-level web APIs with Servant: an exercise in domain-specific generic programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. ACM, 1–12. https://doi.org/10.1145/2808098.2808099

[5] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1, 2 (Apr 2017), 7. https://doi.org/10.22152/programming-journal.org/2017/1/7 arXiv: 1703.10857.

[6] Arian van Putten. 2016. Servis: A dependently typed DSL for web APIs. https://github.com/arianvp/servis