Stellar - A Library for API Programming

André Videla

Glaive, Glasgow, UK
andre@glaive-research.org

Dependently typed programming language manipulate types, but types aren't enough to represent the general *interface* of programs. An Application Programming Interface (API) isn't defined using a type, but a *container* [2] instead. Defining APIs this way allows a declarative style of programming where manipulating APIs is the primary tool for building software. There already exist tools that represent APIs in software, for example the OpenAPI standard [1], or command-line interface DSLs. But we have no way of relating those two APIs. Containers and their morphisms give us the tools to talk about APIs in a principled ways and design complex software in a modular way.

APIs as Containers, What?

Containers defined as a pair of shape and positions can be reinterpreted as a pair of *query* and *response* ($query : Set \triangleright response : query \to Set$). With this interpretation, monoidal operators like coproducts ($+$), tensors ($\times$), composition ($\circ$), perfectly represent valid operations on APIs.

A coproduct of APIs $(query_1 \triangleright response_1) + (query_2 \triangleright response_2)$ constructs an API that gives the choice of what query to send $query_1 + query_2$ and return a response that matches the query that was send $[response_1, response_2]$.

A tensor of APIs $(query_1 \triangleright response_1) \times (query_2 \triangleright response_2)$ produces an API that expects both queries at once, and returns both response at the same time $(query_1 \times query_2 \triangleright response_1 \times response_2)$.

The composition of APIs $(query_1 \triangleright response_1) \circ (query_2 \triangleright response_2)$ defines a specific sequence of operations starting with $query_1$ and depending on its response, sending another query $query_2$, from which we derive the response for both $((q, k) : \Sigma(q : query_1).reponse_1(q) \to query_2 \triangleright \Sigma(r : response_1(q)).reponse_2(k(r))$

An example could help here

For sure! Let's say we are working with the movie database. It advertises a `GET` endpoint with path `/movie/{movie_id}` and this endpoint return responses `200 OK` with a JSON content body representing the movie's data. This API can be equally represented as the container $Info = (\texttt{movie\_id} \triangleright \texttt{Movie})$. Another endpoint `GET /movie/{movie_id}/alternative_titles` returns a list of titles given a movie id, so as a container it can be represented as $Titles = (\texttt{movie\_id} \triangleright \texttt{List Title})$. A server exposing this API gives any client the choice of what endpoint to call, and therefore, can be represented as the container $Info + Titles \equiv (\texttt{movie\_id} \triangleright \texttt{Movie}) + (\texttt{movie\_id} \triangleright \texttt{List Title})$.

What About Morphisms?

Morphisms of APIs tell us how to delegate from a higher-level API to a more specialised one. For example, a command line interface is an API represented with a container $CLI =$

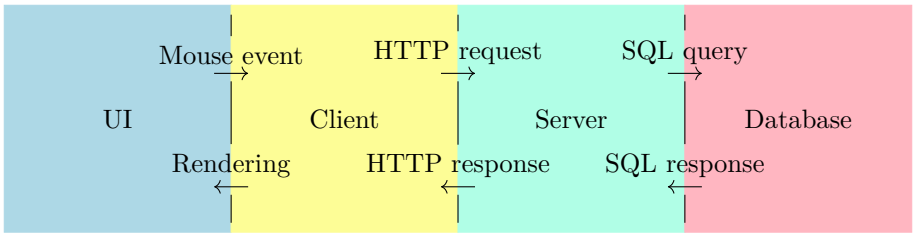

Figure 1: An illustration of a compositional pipeline of API morphisms. Containers are the vertical interstices between colors, the color blocks are container morphisms. The lifecycle of a single interaction can be read clockwise starting from the UI.

($List\ String \triangleright String$) indicating that it expects a list of string as input and returns a string to print as output. A morphism $CLI \twoheadrightarrow DB$ converts messages from $List\ String$ into database queries and, converts back responses from `Table` to `String` to print in the standard output.

Reusing the example of the movie database, if we want to create a program that exposes a command-line interface and forwards requests to the movie database, we could conceptualise this program as the morphism `CLI =>> Info + Titles`. Such morphisms in inhabited by two maps, one to convert command-line arguments to requests:

```
parseArgs : List String -> Movie_id + Movie_id
```

And one to convert http responses into strings we can display in the terminal:

```
convertResponses : Movie + List Title -> String
```

Those two functions can be combined in a single container morphism:

```
app : CLI =>> Info + Titles
app = !! \x => parseArgs x ## convertResponses
```

Surely This Can't Deal With Effects, Can It?

Yes it can! In fact there are many ways to express effectful computation with containers, the most common effects, non-determinism, and failure, can be represented as a List and Maybe monad [6] on containers.

The Maybe monad on container is defined by $Maybe(q \triangleright r) = (Maybe\ q \triangleright Any\ r)$ and the list monad on container is defined by $List(q \triangleright r) = (List\ q \triangleright All\ r)$.

Something like the previous mapping parsing command-line argument can now be written using the `Maybe` monad on containers to accurately represent the fact that parsing might fail: `app : CLI =>> Maybe (Info + Title)`.

Additionally, we can perform effect like IO by mapping the responses of a container with the $(\bullet)$ operation which applies a monad on types to turn it into a comonad on containers: $f \bullet (q \triangleright r) = (q \triangleright f \circ r)$. For example a program that prints its output can be seen as the morphism $IO \bullet (String \triangleright ()) \twoheadrightarrow (String \triangleright String)$ where the forward map is an identity and the backward map is given by `putStrLn : String -> IO ()`, this is an example of a monadic lens [3].

I Get the Idea, But How Do You Even Run Those Things?

A container morphisms which codomain is the monoidal unit can be converted into a function from its query to its response. The function $costate : (q \triangleright r) \twoheadrightarrow I \to (x : q) \to r(x)$ does exactly that. Those leaf morphisms represent computation that can be done immediately and are often found the end of a long sequence of API transformations converting APIs into increasingly simpler ones. Typically, sending an HTTP request amounts to implementing the morphism $HTTP \twoheadrightarrow I$, likewise for database queries and their responses $DB \twoheadrightarrow I$

Conclusion

Implementing containers in a dependently-typed programming language we can create a library which sole purpose is to describe API-level architecture. There are many use-cases for such a library and architecture, including server development, command-line tools, compilers, microservices, and more. We chose Idris [4, 5] for its package manager, ecosystem of low-level libraries, and flexible FFI.

References

[1] OpenAPI Specification v3.1.0 | Introduction, Definitions, & More.

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Martin Hofmann, editors, *Typed Lambda Calculi and Applications*, volume 2701, pages 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[3] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on Monadic Lenses. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 1–31. Springer International Publishing, Cham, 2016.

[4] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, September 2013.

[5] Edwin Brady. Idris 2: Quantitative Type Theory in Practice. *arXiv:2104.00480 [cs]*, April 2021. arXiv: 2104.00480.

[6] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.