Type-level Web APIs with Servant

An exercise in domain-specific generic programming

Alp Mestanogullari
The Kitty Knows Ltd
alp@thekittyknows.com

Sönke Hahn, Julian K. Arni
Zalora SEA
{soenke.hahn, julian.arni}@zalora.com

Andres Löh
Well-Typed LLP
andres@well-typed.com

Abstract

We describe the design and motivation for Servant, an extensible, type-level DSL for describing Web APIs. Servant APIs are Haskell types. An API type can be interpreted in several different ways: as a server that processes requests, interprets them and dispatches them to appropriate handlers; as a client that can correctly query the endpoints of the API; as systematic documentation for the API; and more. Servant is fully extensible: the API language can be augmented with new constructs, and new interpretations can be defined. The key Haskell features making all this possible are data kinds, (open) type families and (open) type classes. The techniques we use are reminiscent of general-purpose generic programming. However, where most generic programming libraries are interested in automatically deriving programs for a large class of datatypes from many different domains, we are only interested in a small class of datatypes that is used in the DSL for describing APIs.

1. Introduction

The interface of a web application is described by its API. For example, an HTTP API describes what kind of requests are accepted by an application, what constraints are being imposed on the URLs, the request headers, the body of input requests, and what kind of responses can be expected.

In this paper, we introduce Servant, an extensible type-level DSL for Haskell that aims at making web APIs *first-class*. A Servant web API is just a Haskell type. It can be named, passed to or returned by (type) functions, composed, exported by modules, and more.

There are already quite a number of other web DSLs that allow the concise description of a *web server*: the program is structured in such a way that the API can be easily read off the way in which the handler functions are written and structured. Sometimes, this approach is even taken further, and the server is augmented with additional information such that API documentation can be generated automatically from the same code. However, there are limits to this approach: the API here exists *implicitly* and is determined by the server implementation; the documentation and the server implementation are mangled together and cannot easily be separated; it is difficult to extend the language. For example, such an approach makes it difficult to say that two different servers implement the same API, or make precise the differences in the API they implement. We also cannot easily implement a client for this API and express in the type system that the API of the client matches that of the server.

There are also other, more general-purpose, API description languages. However, they suffer from their own set of disadvantages. They often lack the abstraction capabilities that a DSL embedded into a language like Haskell can offer. Also, because such descriptions are completely detached from an implementation, it can be a lot of work to establish that an implemented server (or client) actually conforms to the described API.

With Servant, we make APIs first-class citizens that are independent of, but can be checked against their implementation(s). This allows us to compare different implementations of a single API, to express that a client and a server are API-compatible, and to perform many other interesting operations on APIs.

1.1 An introductory example

Here is the API of a minimalistic "echo" service in Servant:

```
type Echo = "echo"
         :> ReqBody '[PlainText] String
         :> Get     '[PlainText] String
```

The Haskell type `Echo`[1] describes an API that accepts only GET requests directed at the route `/echo`. It expects a request body of content type `text/plain`, and produces a result of the same content type. In the Haskell world, the content will be treated as a value of type `String`.

The challenge we want to address in Servant is not just promoting APIs to first-class entities, but to do so in a way that is both *type-safe* and *extensible*.

By type safety, we mean that we want to be able to have the compiler determine statically whether our code conforms to the API. For example, a server implementing the `Echo` API really gets a `String` as input, and has to return a `String` as output. Defining the code and having it type-checked could not be any simpler:

```
handleEcho :: Server Echo
handleEcho txt = return txt
```

Here, the type `Server Echo` expands to (essentially) `String -> IO String`.

By extensibility, we mean two different aspects: First, the API description language itself is not closed. Servant defines operations such as `:>`, `ReqBody`, `Get` and many more, but if that is not suffi-

[1] In Servant, we make use of a number of Haskell extensions that GHC implements. The string `"echo"` is a type-level string. The syntax `'[PlainText]` denotes a one-element type-level list, not the type of lists. Both are made available by the `DataKinds` GHC extension. And `:>` is an infix operator on the type-level, requiring `TypeOperators`.

cient, then it is easy to define new constructs without touching the core library. Next to the obvious advantages in structuring the code of Servant itself, this also lifts some pressure from the developers: users of Servant who lack a feature can likely implement it on their own, without having to change Servant itself.

Second, the number of interpretations associated with an API is also open. Servant itself comes with quite a number of interpretations already: we can supply suitable handlers to run a web service conforming to the API; we can generate clients in two different ways, either as a number of Haskell functions that can be used as part of a larger client application, or as JavaScript functions that can be used for quickly testing the API interactively from the browser; there is the option to generate Markdown documentation of the API; we can generate type-safe hyperlinks for routes within an API. But there are many other things we can imagine doing with web APIs: generating automatic test suites, generating documentation in the format expected by general-purpose API specification languages, generating servers for other low-level HTTP server implementations and many more. With Servant, adding these is easily possible.

The challenge of achieving extensiblity in these two ways is commonly known as the *expression problem* (Wadler 1998), and the implementation of Servant is a solution of one particular instance of this problem.

1.2 Implementation techniques

The techniques we are using in order to implement Servant are reminiscent of datatype-generic programming (Rodriguez et al. 2008). We use our API types as codes in a universe, and have several interpretations of such codes as various other types as well as associated generic functions on them. However, where datatype-generic programming usually aims at representing as many datatypes as possible via codes, Servant is deliberately *domain-specific*: we are only interested in a very limited (yet extensible) set of codes.

Furthermore, we are using quite a number of advanced features of the Haskell type system: type operators to provide a concise look-and-feel to the API language; strings on the type level (in other words, data kinds and kind polymorphism); most essentially, we use (open) type families and (open) type classes to obtain the exensibility we so desire.

We believe that Servant provides evidence that these techniques are feasible for solving the expression problem and designing type-safe extensible DSLs, for web APIs as well as in other domains.

1.3 Contributions and structure of the paper

In this paper, we make the following contributions:

- we introduce Servant itself, and describe the various benefits one gets by using it (Section 2);
- we show how Haskell's type system can be used to implement Servant, by using a form of domain-specific generic programming (we talk about the syntax of the DSL in Section 3 and about the implementation of interpretations in Sections 4 and 5);
- we show that the implementation is fully extensible, thereby providing a solution to the expression problem (Section 6).

In Section 7, we discuss related work, before we outline future work and conclude in Section 8.

1.4 The status of the Servant implementation

Servant already exists for a while, and has evolved a bit. The original motivation for developing Servant came from implementation needs at Zalora. At the time of writing, Servant is available for use on Github and Hackage as a collection of packages: servant for the core API DSL, and dedicated packages for each of the interpretations. This paper describes version 0.4, and some aspects currently in development. Servant is used in production right now at Zalora and by several other users.

2. Using Servant

In the introduction, we've provided a very simple example of an "echo" server API and its implementation. In this section, we are going to look at a slightly more involved example, and explain in a bit more detail how the existing features of Servant can be put to use. We are not yet concerned with how everything works internally – this will be covered in Sections 3 to 6.

2.1 Stepping a counter

For a start, consider the following informally specified API:

```
GET   /      obtain the current value of a counter
POST  /step  increment the current value of said counter
```

In Servant, one way to specify this API is as follows:

```
newtype CounterVal = CounterVal {getCounterVal :: Int}
  deriving (Show, Num, FromJSON, ToJSON)

type GetCounter    = Get '[JSON] CounterVal
type StepCounter   = "step" :> Post '[] ()

type Counter       = GetCounter :<|> StepCounter
```

We introduce a `newtype` for counter values to provide some extra type safety. (Such semantic typing is also useful for implementing type class instances that should only apply to `CounterVal`, and not to `Int` in general.) We derive several type classes that `Int` is a member of for `CounterVal`.

The `GetCounter` type specifies the GET part of the API, as indicated by the use of `Get`. It is accessible via route `/`. The result is a Haskell `CounterVal`, which will be transformed to a JSON number. The `StepCounter` type is for the POST part, as indicated by the use of `Post`. This part is accessible under the route `/step`, and it returns an empty response. In `Counter`, both routes are combined into a single API by means of `:<|>` (which is provided by Servant).

2.2 Generating documentation

If you are implementing a web service, it is very convenient to have up-to-date documentation of its API. With Servant, we can generate documentation directly from an API, and augment it as needed with additional information.

Here, we are only going to show how to obtain basic documentation without any additional descriptions. For this purpose, Servant provides the function

```
docs :: HasDocs api => Proxy api -> API
```

We have to tell `docs` what API we want to use. Because the result of `docs` is always just `API`, an abstract structured description of the API documentation, it provides no clue to the type checker what instance of `HasDocs` to use. The common approach to solve this problem is to pass in a *proxy*:

```
data Proxy a = Proxy
```

A proxy has a trivial run-time representation, but as a `data`-defined type is injective by construction. It can thus be used by the type checker to infer the desired value of `api`.

Servant also provides a function

```
markdown :: API -> String
```

that can render documentation as a `String` in Markdown format.

However, if we try to say

```
counterAPI :: Proxy Counter
counterAPI = Proxy

counterDocs :: String
counterDocs = markdown $ docs counterAPI
```

```
## GET /

#### Response:

- Status code 200
- Headers: []

- Supported content types are:

- ‘application/json‘

- Response body as below.

‘‘‘javascript
42
‘‘‘
```

Figure 1. The GET part of `counterDocs`

we get a type error. In order to describe the format of input and output parameters, Servant tries to generate examples based on their types. For `CounterVal`, we have not yet specified how to do so. We need to say something like

```
instance ToSample CounterVal CounterVal where
  toSample _ = Just (CounterVal 42)
```

in order to make the definition of `counterDocs` type-check and instruct Servant that `42` is a suitable example value for a counter.

We show the output fragment generated for the GET part of the API in Figure 1.

2.3 Implementing a server

In implementing a server conforming to `Counter`, we follow its structure. We can first define a handler for each of the components. Let us first look at the GET part:

```
handleGetCounter :: TVar CounterVal -> Server GetCounter
handleGetCounter ctr = liftIO $ readTVarIO ctr
```

Obviously, we need to maintain the state of the counter. In this example, we are ignoring any question of persistence – as it is orthogonal to how Servant works – and just use a transactional variable (`TVar`) that we pass as an additional argument to the handler.

The Servant library defines a type family called `Server` that computes the type of the required server code from any given API specification. We have that

```
Server GetCounter ~ EitherT ServantErr IO CounterVal
```

(note that ~ is GHC's way of denoting type equality). This type indicates that we have two options: we can successfully process the request, and produce a `CounterVal`; or we can fail and produce a `ServantErr`, which among other things allows us to choose an appropriate status code and message to indicate the kind of failure. In making this decision, we are allowed to use `IO`.

For `handleGetCounter`, we are only interested in reading the transactional variable, so we always succeed. By using

```
liftIO :: IO a -> EitherT e IO a
```

we can embed an `IO` action into the more complicated type.

The code for the POST part is very similar:

```
handleStepCounter :: TVar CounterVal -> Server StepCounter
handleStepCounter ctr =
  liftIO $ atomically $ modifyTVar ctr (+ 1)
```

Finally, we can compose a server for the whole API:

```
handleCounter :: TVar CounterVal -> Server Counter
handleCounter ctr  =  handleGetCounter  ctr
                 :<|> handleStepCounter ctr
```

The type `Server Counter` expands to a `Server GetCounter` paired with a `Server StepCounter`, but for purely aesthetic reasons, Servant uses a term-level `:<|>` instead of Haskell's standard pair constructor for joining the individual handlers.

2.4 Using the server

Once we have `handleCounter`, we would like to run our server. We can do so by using the function `serve` that is provided by Servant:

```
serve :: HasServer api
      => Proxy api -> Server api -> Application
```

A proxy is needed again. While `api` is mentioned in the call `Server api`, GHC cannot use it to infer `api`, as `Server` is a type function and – unlike a `data`-defined type – not necessarily injective. The `Application` type is provided by the wai package, which is independent from Servant. Several other Haskell packages can deal with wai applications – in particular, the warp webserver.

Putting it all together, we need the following code to actually start up a web server serving our API (on port 8000):

```
start :: IO ()
start = do
  initCtr <- newTVarIO 0
  run 8000 (serve counterAPI (handleCounter initCtr))
```

2.5 Querying the server

Once the server is running, we can use a generic web client such as `curl` in order to observe that it seems to work as expected:

```
$ curl -X GET localhost:8000/ -w '\n'
0
$ curl -X POST localhost:8000/step
$ curl -X GET localhost:8000/ -w '\n'
1
```

We can also send requests that do not conform to the API:

```
$ curl -X POST localhost:8000/ -w '\n'
method not allowed
$ curl -X GET localhost:8000/foo -w '\n'
not found
$ curl -H "Accept: text/plain" -X GET localhost:8000/ -w '\n'
unsupported media type
```

The last request fails because we explicitly state that we expect a plain-text result, but our API specifies that the result can only be returned as `JSON` (which corresponds to `application/json`).

Instead of using `curl`, we can also use Servant itself to give us a client for the API. To do so, we simply say:

```
getCounter :<|> stepCounter =
  client counterAPI (BaseUrl Http "localhost" 8000)
```

The function

```
client :: HasClient api
       => Proxy api -> BaseUrl -> Client api
```

computes client functions for a given API and base URL. The types of the client functions obviously depend on the type of the API, and are computed by the type function `Client`. In this case, we obtain

```
getCounter  :: EitherT ServantError IO CounterVal
stepCounter :: EitherT ServantError IO ()
```

A `ServantError` occurs e.g. if we try to connect to a non-existing server or there is any other kind of problem with processing the request.

We can now also interact with our server from a Haskell program or a GHCi session:

```
GHCi> runEitherT getCounter
Right (CounterVal {getCounterVal = 0})
```

```
GHCi> runEitherT (stepCounter >> getCounter)
Right (CounterVal {getCounterVal = 1})
```

Keep in mind that while it is useful to handle both server and client code for the same API via Servant (because we then get the guarantee that they are really using the same API) it is by no means required that we do so. We can also specify a web service API using the Servant DSL, and use Servant to communicate with clients or servers that are not implemented using Servant.

2.6 Modifying the API

Let us extend the current API with a new route:

```
type SetCounter = ReqBody '[JSON] CounterVal :> Put '[] ()

type Counter = GetCounter :<|> StepCounter :<|> SetCounter
```

While `SetCounter` is new, `Counter` is a modification of the original definition. Once we do this, nearly everything we have done with the old `Counter` API becomes type-incorrect – and this is a good thing! Let us just look at the server: clearly, our old `handleCounter` is no longer a faithful implementation of this new and modified API.

However, adapting the server is easy enough:

```
handleSetCounter :: TVar CounterVal -> Server SetCounter
handleSetCounter ctr newValue =
  liftIO $ atomically $ writeTVar ctr newValue
```

Because `SetCounter` contains the `ReqBody` component, the handler of type `Server SetCounter` can expect an additional `CounterVal` argument that contains the decoded request body. It remains to update the definition of `handleCounter`:

```
handleCounter :: TVar CounterVal -> Server Counter
handleCounter ctr  =  handleGetCounter  ctr
                  :<|> handleStepCounter ctr
                  :<|> handleSetCounter  ctr
```

If we use `client` on the new API, we now obtain three functions:

```
getCounter :<|> stepCounter :<|> setCounter =
  client counterAPI (BaseUrl Http "localhost" 8000)
```

2.7 Type-safe links

Another possible extension of the API is to provide an HTML as well as a JSON representation of counter values. The HTML representation should include a button that can be used to step the counter.

We can do this in principle by changing

```
type GetCounter = Get '[JSON] CounterVal
```

to

```
type GetCounter = Get '[JSON, HTML] CounterVal
```

However, making this change will result in another type error: GHC will complain that it does not know how to convert a `CounterVal` into HTML. For JSON, we did not have a problem because we derived the `ToJSON` instance for `CounterVal`, which falls back on the predefined `ToJSON` implementation for `Int`. For `CounterVal`, we could in principle do a similar thing and fall back on a generic HTML conversion function for integers – but that is not actually what we want here! We want to produce a specific representation for counters, and therefore define our own instance[2]:

```
instance ToHtml CounterVal where
  toHtml (CounterVal val) =
         p_  ( toHtml
             $ "Current value: " ++ show val ++ "."
             )
      <> with form_ [action_ stepUrl, method_ "POST"]
               (input_ [type_ "submit", value_ "Step!"])
    where
      stepUrl = "step"
```

By saying `stepUrl = "step"`, we are providing the URL of the page that we wish to POST to. However, putting a `String` link here is quite unsatisfactory, and we can do better than that. We can simply say

```
        stepUrl = pack . show $ safeLink counterAPI
                                  (Proxy :: Proxy StepCounter)
```

instead to get the same effect in a safer way.

The Servant function

```
safeLink :: (Elem endp api ~ True, HasLink endp)
         => Proxy api -> Proxy endp -> MkLink endp
```

takes two proxies: one for the entire API we are operating on, and one for an element of the API describing the path to a single endpoint. The constraints here ensure that a call to `safeLink` only type-checks if the given API contains endp. The result is a `MkLink endp`, where `MkLink` is yet another type function. Some links may require additional parameters – e.g., to render query parameters into the link – but in our particular case, `MkLink StepCounter` expands just to an `URI` type, which we subsequently convert to `Text` for use in the HTML library.

2.8 Summary

All of Servant is centered around its DSL for defining APIs. The example API we have discussed is very simple. Nevertheless, it suffices to demonstrate the benefits of Servant's approach. By turning the API into a Haskell type and thereby into a first-class citizen of the language, we gain the ability to perform many operations on the API, while statically checking that all our applications are in fact conforming to the desired API and consistent with each other. Applications we have seen so far are documentation generation, implementation of servers, generation of client functions, and type-safe links.

In each case, as much information as possible is encoded in and automatically inferred from the provided API. But we also see that additional information that is compatible with the API in question can be supplied, such as the implementation of the handlers for the server.

In addition, we benefit from the abstraction capabilities of the underlying Haskell language in that we can abstract and reuse APIs. If, for example, we suddenly decide that we would prefer to integrate the counter API into a larger service, we can simply define

```
type Full  =  "counter" :> Counter
         :<|> ...
```

and reuse `handleCounter` in defining `handleFull`. And if we then decide that we would actually like to version our API, we can apply the same approach again by saying

```
type FullVersioned  =  "v1" :> Full
                   :<|> "v2" :> FullV2
                   :<|> "v3" :> ...
```

once again reusing handlers (and other aspects like documentation information) between versions where unchanged.

3. The DSL for APIs

After having seen several examples of how to use Servant, let us now look more closely at how it works. We will start with the API description language provided by Servant. In Section 3.1, we summarize what a web API actually describes. In Section 3.2, we provide a grammar for our language, and in Section 3.3, we describe how we implement the grammar as a type-level DSL in Haskell.

[2] Out of the box, Servant supports two popular Haskell HTML libraries: blaze-html and lucid. We are using lucid here.

3.1 Request, Response

A web API is an API that follows the pattern of request-response and is made available via the network. Web APIs follow the HTTP protocol, which specifies that request messages should contain among other things a method (`GET`, `POST`, `DELETE`, etc.), a path, zero or more headers, a query string and a possibly empty body. HTTP responses must contain a status code (a three-digit number describing the outcome of the request), zero or more headers, and a possibly empty entity-body. *Endpoints* are specific combinations of constraints on HTTP requests (usually consisting at least of a constraint on the path the request is made to, such as "equal to `/users`"). The code that runs and produces a response from a request at a given endpoint is usually called a *request handler*.

A request handler can be specified by listing exactly the constraints it places on the request (e.g., on the path, the method, the parameters or the body) and the guarantees it makes about its response (e.g., headers present or the shape of the body). Such a description provides enough information for external systems to reliably interact with the web service, without having to know anything about the code running inside its request handlers.

3.2 Syntax of APIs

To design a DSL that lets us describe APIs in a programming language, we need various constructs: one for each HTTP method, one for paths, one for request bodies and so on. Since a webservice usually consists of more than one endpoint, our DSL should also support a way to glue several endpoint descriptions together. The (extensible) grammar of our DSL is shown in Figure 2 .

The two core combinators for building APIs are `:<|>`, which expresses a choice between two endpoints, and `:>` which sequences constraints on a request. The various forms of constraints on a request that can be specified are given by item. If a type-level string (such as `"step"` in Section 2) is used, we specify one component of the path of the URL of the request. We can use `Header` to require the presence of a particular request header. Similarly, with `ReqBody`, we indicate that we expect the request body to be decodable to a specific Haskell type from a list of possible formats. With `Capture`, we indicate that we expect a *variable* path component at this position and that it must be decodable as a certain type. By using `QueryFlag`, `QueryParam` or `QueryParams`, we check for the presence of a particular query string parameter as part of the request.

Every route in an api must end in a method. This category corresponds to the HTTP request methods, of which the most common ones are directly supported. Each method is parameterized by a list of content types and a Haskell type. The Haskell type specifies the type of the result in the Haskell world; the list of content types indicates the representations the Haskell type is available in. The `Headers` construct can be used to attach some headers to a response. Finally, there is a `Raw` construct used to integrate external web applications of type `Application` under some path in an API – it does not induce any constraint on the request.

3.3 Types are first-class citizens

Our DSL lives on the type level of Haskell, i.e., each expression built according to the grammar in Figure 2 denotes a Haskell type. As we have hinted at in Section 1, there are a number of reasons for moving to the type level: First, we want to separate the API from any intepretation of the API (such as a web service implementing the API). Second, we want to provide as much type safety as possible and as much automation as possible when associating interpretations with a given API. And third, we want the whole system to be extensible in two dimensions (Wadler 1998): we support the addition of new DSL constructs as well as the addition of new interpretations.

Since the grammar is used to structure Haskell types, it suggests we should map each of the syntactic categories in the grammar to a distinct Haskell *kind*. Unfortunately, while Haskell now supports the definition of new kinds via datatype promotion (Yorgey et al. 2012), such kinds are *closed*, whereas we want all major categories to be *open*, as indicated in the grammar. Haskell (or rather GHC) supports just two open kinds: `*` and `Constraint`. As `Constraint` is rather special-purpose, we have no other choice but to use `*`.[3] Nearly every construct shown in Figure 2 corresponds to one Haskell *datatype* of kind `*`. [4] Our API types are merely codes or descriptions. We will use type functions to interpret them in various ways. We are losing some kind-safety in this way, which is unfortunate. The safety of the overall program is not affected: if we make mistakes, the program will still fail at compile time. But the error messages may be worse than they would be otherwise, and we lack the extra guidance that stronger kinds would provide.

As an example, we show the type declarations for some representative DSL constructs:

```
data api1 :<|> api2
infixr 8 :<|>

data (item :: k) :> api
infixr 9 :>

data ReqBody (ctypes :: [*]) (t :: *)
data Capture (symbol :: Symbol) (t :: *)

data Get (ctypes :: [*]) (t :: *)
data Post (ctypes :: [*]) (t :: *)

data JSON
```

As an exception to the general rule, we keep the first argument of `:>` kind-polymorphic. This allows us to directly use type-level strings (which have kind `Symbol` in Haskell) as well as the other item constructs (which we define to be of kind `*`) in this position.

The DSL is reminiscent of a universe or view used in general-purpose datatype-generic programming (cf. Section 7.1). There, the codes typically describe arbitrary (Haskell) datatypes. An operator similar to `:<|>` is used to denote choice between constructors, and an operator similar to `:>` is used to group multiple arguments of a single constructor. Such general-purpose codes are then interpreted as datatypes, and generic functions are defined over the structure of the codes in order to work for all datatypes covered by the universe.

The Servant situation can thus be seen as *domain-specific generic programming*: Our codes do not aim to describe a large class of datatypes. They describe APIs. Nevertheless, we interpret them as other types, as we shall see in Sections 4 and 5. And we define "generic" functions over them that automatically work for all valid APIs.

Every construct we have now defined is just an individual datatype, and completely independent of the others. In fact, they

[3] As described by Eisenberg and Weirich (2012), there actually is a trick to create more closely matching kinds that are still open. For e.g. content types, we cannot promote

```
data ContentType = JSON | HTML | PlainText
```

because that would be closed. But we can define

```
data ContentType
data JSON (c :: ContentType)
...
```

Now all content types are of kind `ContentType -> *`, distinct from `*`. However, this encoding introduces extra type arguments in many places, reducing readability for relatively little extra safety, so we do not use it. It would be nice if Haskell had direct support for user-defined open kinds.

[4] These types can be uninhabited as far as the description language is concerned. Some of them (such as `:<|>`) actually have constructors, but only in order to reuse the type in some of the interpretations.

```
api    ::= api :<|> api
         | item :> api
         | method

item   ::= symbol
         | header
         | ReqBody      ctypes type
         | Capture      symbol type
         | QueryFlag    symbol
         | QueryParam   symbol type
         | QueryParams  symbol type
         | ...

method ::= Get     ctypes rtype
         | Put     ctypes rtype
         | Post    ctypes rtype
         | Delete  ctypes rtype
         | Patch   ctypes rtype
         | Raw
         | ...

rtype  ::= Headers headers type
         | type

headers ::= '[header, ...]
ctypes  ::= '[ctype, ...]
header  ::= Header symbol type
symbol  ::= a type-level string
type    ::= a Haskell type
ctype   ::= PlainText
          | JSON
          | HTML
          | ...
```

Figure 2. Grammar of the Servant type-level DSL for APIs

are defined in separate modules in the Servant library, and they could very well be defined in separate packages. This is essential for extensibility. If we want to extend the API language, we can do so by defining a new datatype elsewhere, without changing the existing Servant library. No combinator is special and user-written ones can freely be mixed with the ones above.

4. Interpreting an API as a server interface

The DSL described in the previous section lets us write down descriptions of web APIs as types. However, the types alone are not yet very useful. So far, they are merely abstract descriptors.

In this and the following section, we are going to explore how we can interpret APIs in several different ways. We will start by looking at what is probably the most prominent interpretation: that as a web service.

In Section 2.4, we have already seen that Servant provides a function

```
serve :: HasServer api
      => Proxy api -> Server api -> Application
```

that takes a proxy for an API type as well as suitable handler functions to a target type that is called `Application`. It is the goal of this section to explain how to implement this function.

The rest of this section is structured as follows: we start by briefly introducing the wai library that we target and that provides the `Application` type (Section 4.1). We then look at the general approach of how to use both type classes and type families in order to provide semantics to an open type-level language such as our DSL (Section 4.2). And finally (Sections 4.3 to 4.9), we explain how this technique can be applied in the implementation of Servant.

4.1 The wai view on web applications

Because we do not want to implement a web server from scratch, we choose a suitable low-level target library called wai (the "Web Application Interface", short wai). Using wai is not essential for using Servant though – we could target other similar libraries instead.

At the core of wai is the `Application` type that we have already seen. It is a type synonym that is defined as follows:

```
type Application = Request
                -> (Response -> IO ResponseReceived)
                -> IO ResponseReceived
```

This is very nearly

```
type Application = Request -> IO Response
```

only that instead of producing the `Response` directly, we are provided with a continuation that we can call once we have a `Response`. The types `Request` and `Response` are Haskell record types describing an HTTP request and a response, respectively.

Once we have defined a value of type `Application`, we can use a Haskell web server such as warp to run it. Our aim is thus to interpret our API language as an `Application`, as performed by the `serve` function. But we also want the interpretation to be *compositional*: it should follow the structure of the API language, i.e., the grammar given in Figure 2, and the semantics of complex API types should be defined in terms of the semantics of its components.

However, for the components, we need slightly more information than `Application` itself is able to provide: the idea of the choice combinator `:<|>` is that we can combine several API parts, and if the first part does not match a given request, we will try the second. But how do we express "does not match" using the `Application` type? With `Application`, we have to produce a `Response`, but while a `Response` can of course represent an HTTP error, we need a way to distinguish a *hard failure* (i.e., a request that was matched and resulted in a true error) from a *soft failure* indicating that the request has not been matched yet.

In Servant, we extend `Application` to `RoutingApplication` for this reason:

```
type RoutingApplication =
     Request
  -> (RouteResult Response -> IO ResponseReceived)
  -> IO ResponseReceived
```

The only difference is in the continuation, which now takes a `RouteResult Response` rather than a plain `Response`, where

```
data RouteResult a = NotMatched | Matched a
```

allows us to use `NotMatched` in a component if we want to indicate a soft failure.[5]

It is easy enough to go back to an `Application` in the end – all we have to do is to decide how to interpret an unhandled `NotMatched`, which we do by sending a generic empty 404 response:

```
toApplication :: RoutingApplication -> Application
toApplication app request respond =
  app request $ \routeResult -> case routeResult of
    Matched result -> respond result
    NotMatched     -> respond
                        (responseLBS notFound404 [] "")
```

4.2 Interpreting open type-level DSLs

Before we put `RoutingApplication` to good use by interpreting the rather large Servant DSL, let us first take a step back and look at

[5] We have simplified `RouteResult` to keep the presentation concise. In the actual Servant implementation, `RouteResult` contains additional information that tracks the exact reason for the failure. This information is then used to send appropriate HTTP status codes in the responses.

a much simpler type-level DSL, using it as an example to describe the general technique. Consider the following grammar:

```
expr ::= One | expr :+ expr | Hole | ...
```

This is an open expression language that allows us to denote the constant number "one" as well as addition of two expressions. We also allow holes in our expressions. As described in Section 3, we are mapping this grammar to the Haskell type level by introducing a new uninhabited datatype for each construct:

```
data One
data e1 :+ e2
data Hole
```

As with Servant itself, we can use type synonyms to define terms in this language:

```
type Two   = One :+ One
type Holes = Hole :+ One :+ Hole
```

A simple interpretation of such expressions would be as a string, printing holes using a special character, e.g., `'_'`. A more interesting situation arises if we want to interpret expressions as their integer values, and take additional arguments for each `Hole`. To implement this, we define a type class `HasValue` that contains an *associated type* – an open type function defined within the class:

```
class HasValue a where
  type Value a r :: *
  valOf :: Proxy a -> (Int -> r) -> Value a r
```

The function `valOf` gets a continuation to feed the resulting integer to. By setting `Value`, each construct can decide whether it wants to return the final result `r` immediately, or demand additional arguments.

We now give the interpretation simply by defining instances for each construct, following the structure of the grammar. The case for `One` does not demand any additional arguments:

```
instance HasValue One where
  type Value One r = r
  valOf _ ret = ret 1
```

The case for `:+` calls the components in continuation passing style:

```
instance (HasValue e1, HasValue e2) =>
         HasValue (e1 :+ e2) where
  type Value (e1 :+ e2) r = Value e1 (Value e2 r)
  valOf _ ret = valOf (Proxy :: Proxy e1) (\v1 ->
                valOf (Proxy :: Proxy e2) (\v2 ->
                ret (v1 + v2)))
```

For holes, we do expect an additional integer argument:

```
instance HasValue Hole where
  type Value Hole r = Int -> r
  valOf _ ret n = ret n
```

Because of `Value` being set to `Int -> r`, we can expect the integer `n` in `valOf` and then use it.

It remains to define a wrapper initializing the continuation to `id` to just produce the final result:

```
valueOf :: HasValue a => Proxy a -> Value a Int
valueOf p = valOf p id
```

This works as expected:

```
GHCi> let t1 = valueOf (Proxy :: Proxy Two)
GHCi> let t2 = valueOf (Proxy :: Proxy Holes)
GHCi> :t (t1, t2)
(t1, t2) :: (Int, Int -> Int -> Int)
GHCi> (t1, t2 3 4)
(2, 8)
```

Type families thus allow us to compute new types from the type descriptors that make up our DSL. The DSL is fully extensible: We can define new interpretations simply by adding another class. We can also add a new construct to our DSL by adding a new datatype definition and adding a new instance for each of the classes.

We are now going to use the same technique to provide an interpretation for the Servant DSL.

4.3 The `HasServer` class

The interpretation of an API as a web server is given by the `HasServer` class:[6]

```
class HasServer api where
  type Server api :: *
  route :: Proxy api -> Server api -> RoutingApplication
```

The API alone is not enough to describe a server. A server consists in essence of a collection of suitable handlers that can deal with the incoming requests. The type of this collection is computed by the associated type `Server`. Once the server is running, we have to route requests to the appropriate handler. This is the task of `route`.

As in Section 4.2, we proceed by giving instances for all the constructs of our DSL, following the grammar of the DSL as given in Figure 2. But there are some concerns: in our simple example, we had just one syntactic category expr; now we have several. When we want to interpret a *term*-level DSL given by several (mutually recursive) datatypes, we provide a different interpretation function for each datatype. In the *type*-level scenario, this would mean a different type class for each syntactic category, such as

```
class HasServerAPI api
class HasServerItem item
class HasServerMethod method
...
```

However, applying this strategy will in some places lead to overlapping instances that are difficult to resolve for GHC. Instead, we inline the item, header, method and rtype categories shown in Figure 2 into the api category, so that the transformed equivalent grammar looks as follows:

```
api ::= api :<|> api
      | symbol :> api
      | Header symbol type :> api
      | ReqBody ctypes type :> api
      | ...
      | Get ctypes (Headers headers type)
      | Get ctypes type
      | ...
```

From here, we go directly to instance declarations:

```
instance ... => HasServer (api1 :<|> api2)
instance ... => HasServer ((sym :: Symbol) :> api)
instance ... => HasServer (Header sym t :> api)
instance ... => HasServer (ReqBody ctype t)
...
instance ... => HasServer (Get ctypes (Headers hs t))
instance ... => HasServer (Get ctypes t)
```

Note that the instance `(sym :: Symbol) :> api` does not overlap with the other `:>`-instances: We made `:>` kind-polymorphic in its first argument, and the kind here is `Symbol`, whereas all other instances have an argument of kind `*` in this position – and GHC uses kind information during instance resolution.

We will look at a few of these instances more closely and discuss the main ideas that are being used throughout the implementation.

[6] The actual implementation defines a `ServerT` monad transformer rather than a `Server` monad, thereby giving extra flexibility to users in that they can use (parts of) their own monad transformer stack when implementing servers.

4.4 The case of `Get`

Let us look at the instance for `Get` first. Recall that

```
data Get (ctypes :: [*]) (t :: *)
```

is parameterized by a list of content types and the Haskell return type of the handler. Let us at first consider a simplified version of `Get` that instead of the general content-type handling always returns a textual result:

```
data GetText (t :: *)
```

If the API is a plain `GetText` node, then we expect `t` to be convertible into a `String` (i.e., to be in the `Show` class), and the handler corresponding to such a node is a monadic action that takes no arguments and produces a value of type `t` if successful. The monad we use here is `IO` with the additional option to produce error responses of type `ServantErr`:

```
instance Show t => HasServer (GetText t) where
  type Server (GetText t) = EitherT ServantErr IO t

  route :: Proxy (GetText t) -> Server (GetText t)
        -> RoutingApplication
  route _ handler request respond
      | pathIsEmpty request
     && requestMethod request == methodGet = accept
      | otherwise = respond NotMatched
```

The function `route` takes four arguments: it ignores the proxy (which we need only for guiding the type checker), takes the server, which is a single handler. It furthermore gets the request and the continuation for responding from the `RoutingApplication`. We expect the path of the incoming request to be empty, and the request method to be `GET`. We either accept or reject the request. We still have to implement `accept`:

```
    where
    accept = do
      e <- runEitherT handler
      respond $ case e of
        Right t  -> Matched $ responseLBS ok200
                    [("Content-Type", "text/plain")]
                    (fromString (show t))
        Left err -> Matched $ responseServantErr err
```

Accepting the request means that we call the handler and process its result. Since the `Server (GetText t)` type expands to an `EitherT`, the handler can itself succeed (`Right`) or fail (`Left`). On success, we create a successful (status 200) response and convert the result of the handler into the response body. If the handler fails, we pass on the error using `responseServantErr`, which will result in an error response being sent to clients.

4.5 Content types

If we want to move from the simplified `GetText` to the full `Get` combinator, we have to understand how Servant treats content types: An API specification such as

```
Get '[JSON, HTML, PlainText] MyType
```

means that Servant can use three different methods to encode a value of `MyType` as the body of the response. Each content-type descriptor (such as `JSON`, `HTML`, `PlainText`) determines not only the `Content-Type` header of the response, but also the conversion method that is being used. By default, Servant will use the first type mentioned in the list. However, clients are allowed to use `Accept` headers in the requests they send to indicate what content types they expect to see.

Every content-type descriptor in Servant must be a member of the `Accept` class, which specifies the media type to use (Freed and Borenstein 1996):

```
class Accept ctype where
  contentType :: Proxy ctype -> MediaType
```

The next step is to define how Haskell types can be rendered (e.g. in responses) or unrendered (e.g. in request bodies) according to this content-type method. For rendering, there is

```
class Accept ctype => MimeRender ctype t where
  mimeRender :: Proxy ctype -> t -> ByteString
```

For unrendering, there is a similar class `MimeUnrender`. For most content-type descriptors, there is a rather generic `MimeRender` instance that delegates the actual translation work to some other library.

Next, there is a class called `AllMimeRender` (and similarly `AllMimeUnrender`) that lifts the `MimeRender` functionality to type-level lists containing content type descriptors:

```
class AllMimeRender (ctypes :: [*]) t where
  allMimeRender :: Proxy ctypes -> t
                -> [(MediaType, ByteString)]
```

Using `allMimeRender`, we can compute a lookup table of different `ByteString` encodings, indexed by content type. The instances follow the inductive structure of (type-level) lists:

```
instance AllMimeRender '[] t where ...
instance (MimeRender ctype t, AllMimeRender ctypes t) =>
         AllMimeRender (ctype ': ctypes) t where ...
```

Note that these instances are defined once and forall – they do not depend on the individual content types.

Finally, there is a function

```
handleAcceptH :: AllMimeRender ctypes t
              => Proxy ctypes -> AcceptHeader -> t
              -> Maybe (ByteString, ByteString)
```

that takes an `Accept` header from a request and a value of type `t`. It uses `allMimeRender` to compute the lookup table of available encodings and picks a suitable one allowed by the `Accept` header. Both the selected content type and the encoding are returned as `ByteString`s.

This function is then put to use in the interpretation of `Get`:

```
instance AllMimeRender ctypes t =>
         HasServer (Get ctypes t) where ...
```

We omit the instance body, as there are not many differences compared to `GetText`. The definition of `Server` is identical, and the definition of `route` is the same except that it additionally runs the result of the handler through `handleAcceptH`.

The cases for the other HTTP methods (`Post`, `Put`, `Delete`, etc.) are all very similar to the case for `Get`. In particular, they all reuse exactly the same machinery for content types that we have just discussed.

4.6 The case of symbols

Let us now look at one of the many instances that deal with `:>`, depending on its first argument. If we encounter an API specification of the form

```
"foo" :> rest
```

then the presence of the `"foo" :>` prefix plays no role whatsoever for the handler, so we can reuse the type of the handler for `rest`:

```
instance (KnownSymbol sym, HasServer api) =>
         HasServer (sym :> api) where

  type Server (sym :> api) = Server api
```

For routing, we expect the incoming request to be for a URL path that has `foo` as its first component. If this is the case, we just forward the request to `rest`, with the first path component stripped. Otherwise, we can immediately reject the request:

```
route _ handler request respond =
  case processedPathInfo request of
    p : ps | p == symval -> forward ps
    _                    -> respond NotMatched
  where
  symval     = pack (symbolVal (Proxy :: Proxy sym))
  forward ps = route (Proxy :: Proxy api) handler
                 (request {pathInfo = ps}) respond
```

We use

```
symbolVal :: KnownSymbol sym => Proxy sym -> String
```

from the GHC base libraries in order to obtain the term-level `String` corresponding to a type-level symbol.

4.7 The case of `Capture`

Interesting are the cases that add additional arguments to the type of the handler. Examples are all items that extract dynamic information from the request that is then made accessible to the handler – such as `Capture`. For example, a route

```
Capture "arg" Int :> rest
```

means that the first path component of the request has to be decodable as an `Int`. The rest of the request is then forwarded to `rest`. The decoded `Int` is passed to the handler, which therefore has to be a function expecting an `Int`. The label `"arg"` is not actually being used by the server interpretation – it is primarily being used for generating documentation.

The implementation of the general case looks as follows:

```
instance (KnownSymbol sym, FromText t, HasServer api)
      => HasServer (Capture sym t :> api) where

  type Server (Capture sym t :> api) = t -> Server api
```

The definition of `Server` specifies that the handler in this case is a function taking an argument of type `t`.

```
route _ handler request respond =
  case processedPathInfo request of
    p : ps | Just v <- (fromText p :: Maybe t)
       -> forward ps v
    _  -> respond NotMatched
  where
  forward ps v = route (Proxy :: Proxy api) (handler v)
                   (request {pathInfo = ps}) respond
```

The Servant library provides

```
fromText :: (FromText t) => Text -> Maybe t
```

which is used to attempt a conversion of the first path component of the request into the desired type. If successful, we obtain a decoded value `v` which is then passed to the handler in `forward`.

Other DSL constructs such as e.g. `ReqBody`, `Header` or `QueryParam` use a similar approach, only they try to decode the value being exposed to the handler from a different part of the request.

4.8 The case of `:<|>`

As a final example case, let us look at the choice operator `:<|>`. The handler associated with a choice is a choice between handlers – in other words, a pair of handlers. Because there are often chains of choices occurring in APIs, and nested pairs look syntactically awkward in Haskell, and moving to larger tuples is tricky to do generically, Servant introduces an isomorphic copy of the pair type for this purpose, and reuses the `:<|>` descriptor type for this:[7]

```
data a :<|> b = a :<|> b
infixr 8 :<|>

instance  (HasServer api1, HasServer api2) =>
          HasServer (api1 :<|> api2) where

  type  Server (api1 :<|> api2) =
          Server api1 :<|> Server api2

  route _ (handler1 :<|> handler2) request respond =
    route (Proxy :: Proxy api1) handler1 request $ \r ->
      case r of
        Matched result -> respond (Matched result)
        NotMatched     -> route (Proxy :: Proxy api2)
                            handler2 request respond
```

For routing, the strategy is as follows: We obtain a pair of handlers. We try to feed the request to the first handler. In its continuation, we look at the response `r` of that handler. If the handler accepted the request (`Matched`), we respond with its result. But if the handler rejected the request (`NotMatched`), we try the same request with `handler2` instead.

4.9 Summary

This concludes our discussion of the server interpretation of the API DSL. Note that this is by no means the only way in which we can interpret the Servant DSL as a server. Instead of targetting the wai `Application` type, we could have targetted any other low-level (or even high-level) Haskell web server framework.

It is also possible to be more clever about routing. The actual implementation already does a bit more work because it has a more informative `RouteResult` type and makes use of the information contained therein to send better responses in case of failure. But much like general-purpose parsers can benefit from a left-factoring of the underlying grammar and committing quickly to choices based on limited look-ahead, we can in principle apply similar techniques in Servant: the API is specified statically – we can detect if multiple paths joined by `:<|>` have common prefixes and factor them out; we can also look ahead into the "first sets" allowed for the paths in each of the options, and dispatch more quickly to the right choice rather than trying each branch one by one.

The main purpose of explaining the server interpretation was, however, to give a non-trivial example of how to structure an interpretation in general. Other interpretations follow the same structure (and indeed, the variants of the server interpretation mentioned above could just be implemented as *additional* interpretations). In Section 5, we will look more briefly at some of the other interpretations that come predefined with Servant.

5. Other interpretations

Servant provides more than just one interpretation of the API DSL. Fortunately – while each interpretation adds useful functionality to the Servant landscape – they hardly pose any new difficulties on the implementation side.

Nearly all interpretations follow the scheme that we have outlined in Section 4: there is one type class, and the structure of the instances follows the structure of the inlined grammar as shown in Section 4.3. In this section, we will therefore mainly focus on the generation of type-safe links (Section 5.2), because it introduces a few new implementation ideas, and only very briefly look at the generation of client functions (Section 5.1) and documentation (Section 5.3).

5.1 Generating client functions

Servant provides two ways of generating clients: by directly interpreting the API as a collection of Haskell client functions, and a more classic code-generation approach that produces JavaScript code based on JQuery which can then be used to interact with the server from a web page or a JavaScript program.

[7] This is one of the cases we mentioned in Section 3 where the API descriptor type is not actually uninhabited for aesthetic reasons.

The native Haskell approach is particularly appealing because it does not involve any additional generation step, and any change to the API is immediately reflected in the types of the client functions. The structure of this interpretation is exactly the same as that of the server implementation. The class here is called `HasClient` and again comprises an associated type and a function:

```
class HasClient api where
  type Client api :: *
  clientWithRoute :: Proxy api
                  -> Req -> BaseUrl -> Client api
```

The function `clientWithRoute` takes the proxy for the API, an accumulated request of type `Req`, the base URL for the server, and produces client functions that match the API with the help of `Client`. Users will not invoke `clientWithRoute` directly, but rather `client`, which does nothing else but to initialize `Req` with the representation of an empty request.

The type function `Client` is very similar to the type function `Server` – it computes the types of the client functions from the API type. The main difference is that `Server api` is an input to the `route` function, whereas `Client api` is an output of `clientWithRoute`.

5.2 Type-safe links

As briefly discussed in Section 2.7, Servant also provides the possibility to compute type-safe links to individual endpoints within a given API. The core of this interpretation is formed by another type class:

```
class HasLink endp where
  type MkLink endp
  toLink :: Proxy endp -> Link -> MkLink endp
```

Much like `Req` for `HasClient`, `Link` is an accumulated abstract description of the path to the link up to the current point. The `MkLink` type family computes a function type resulting in an `URI`, but depending on the endp, we might need additional parameters.

One aspect of `HasLink` is significantly different from the server and client interpretations that we have seen before: The `HasLink` class does not range over essentially all API types, but only over a subset of the API DSL, namely the subset without choice (`:<|>`).

The idea is that endp is instantiated with an individual endpoint that is *contained* within a given API. The `toLink` function can be used in order to compute the proper URL path to this point. In order to actually make it safe, we have to separately check whether the endp we are interested in is actually contained in the full API we are working with. For this purpose, there is

```
safeLink :: (Elem endp api ~ True, HasLink endp)
         => Proxy api -> Proxy endp -> MkLink endp
safeLink _ endp = toLink endp emptyLink
```

that additionally performs a suitable check via `Elem`, a type family that is another essential part of the Servant machinery for safe links.

Let us look at `Elem` more closely, [8] because it is a nice demonstration of the power resulting from making APIs first class – we can easily define arbitrary computations on or transformations of APIs, as type-level functions.

The general recursive structure of `Elem` is the same as that of the type classes (cf. Section 4.3), the only difference being that `Elem` is an open type family:

```
type family Elem (e :: *) (a :: *) :: Bool
type instance Elem e (a1 :<|> a2)        = ElChoice e a1 a2
type instance Elem e ((s :: Symbol) :> a) = ElSymbol e s a
...
```

[8] The current implementation of the element check deviates quite a bit from what is presented here – although the ideas are the same. We expect a future release of Servant to adapt the approach described here.

Each case dispatches to another type synonym or type family. These individual type families may be closed (closed type families allow overlapping cases, open type families do not). It is important that `Elem` itself is extensible with new cases when new DSL constructs are added.

As an example of an individual case, let us look at `ElChoice`:

```
type ElChoice e a1 a2 = Or (Elem e a1) (Elem e a2)

type family Or (b1 :: Bool) (b2 :: Bool) :: Bool where
  Or True  b2 = True
  Or False b2 = b2
```

An endpoint is contained in a choice of two APIs if it is contained in the one or in the other.

For a symbol to match, the same symbol must be present in both the endpoint and the API at this point:

```
type family ElSymbol e (s :: Symbol) a :: Bool where
  ElSymbol (s :> e) s a = Elem e a
  ElSymbol e        s a = False
```

Note that type families in GHC – unlike normal function definitions – allow non-linear patterns, and the double occurrence of `s` on the left hand side of the first case implies that both symbols must be equal.

Some other items are considered optional: e.g. a `ReqBody` can be included in the endpoint specification at the correct point, but it can also be omitted, and the element check will still succeed. For the list of content types that appear in the HTTP method combinators, we do not expect exact equality, but merely a sublist relationship.

5.3 Documentation generation

The `Elem` type function discussed above is remarkable because it provides us with a way of identifying parts of an API by name rather than by position.

Recall for example `HasClient` from Section 5.1: a call to `client` produces a collection of Haskell client functions. To get at the individual client functions, we have to pattern match on the resulting structure, using `:<|>` to extract the components (cf. 2.5). But by using a function similar to `Elem`, we could also provide a version of `client` that can generate a specific client function on request, by providing a proxy for the path to its endpoint.

A similar technique is being used in the documentation generator provided by Servant. As shown in Section 2.2, the default format just lists the paths to the endpoints, the methods, and the types of inputs and outputs (possibly by providing examples).

Good documentation, however, contains much more information than that. We might want to flexibly add additional information to various places of the API. But at the same time, we do not necessarily need to attach information everywhere. By using a name-based rather a position-based scheme, we can achieve this goal: the default documentation being generated is free of additional information, but can subsequently augmented with extra material in specific places based on providing paths into the API.

6. Extending the API language

The goal of this section is to demonstrate what a user has to do in order to extend the Servant library in various ways, and that this is indeed feasible without touching existing code. We discuss two examples below: the addition of a CSV content type in Section 6.1, and the addition of a new combinator that allow handlers to access the IP address of clients in Section 6.2.

We have already seen in Section 5 that adding completely new interpretations is easily possible, as the ones shipping with Servant are in no way privileged. We have furthermore seen that sometimes, interesting applications may become possible by the use of additional type functions that operate on the API DSL, such as `Elem` in Section 5.2.

6.1 Adding a new content type

In the course of writing a web application, content types not yet provided by Servant may be needed. Creating a new one is straightforward. We choose as an example the CSV format, and assume that we have an independent library that provides us with a general conversion interface such as the following:

```
class ToCSV   a where encodeCSV :: a -> ByteString
class FromCSV a where decodeCSV :: ByteString -> Maybe a
```

We can then define a new content type descriptor `CSV`, which according to the outline given in Section 4.5, must be made an instance of the `Accept`, `MimeRender` and `MimeUnrender` classes:

```
data CSV

instance Accept CSV where
  contentType _ = "text" // "csv"

instance ToCSV a => MimeRender CSV a where
  mimeRender _ = encodeCSV

instance FromCSV a => MimeUnrender CSV a where
  mimeUnrender _ = decodeCSV
```

The `Accept` instance associates our type with the appropriate media type. The other two instances delegate encoding and decoding to the conversion functions supplied by the assumed CSV library. Taken together, these three instances make `CSV` a fully supported content type one can use in API types.

If defined like above, the `MimeRender` and `MimeUnrender` classes cannot be extended with other instances easily (that would lead to overlapping instances). If we for example wanted to support two different CSV libraries, the correct approach would be to define two different Servant content types that would both map to the `text/csv` media type.

6.2 Adding a `Host` combinator

As a second example, let us extend the API description language with a new construct that belongs into the item category of the grammar: a `Host` combinator that indicates that an endpoint needs access to the IP addresses or hosts that the request originates from.

Haskell's main networking library provides a `SockAddr` type whose very purpose is to represent hosts and IP addresses, which makes it a great fit for the interpretation of the host combinator as an argument of that type to request handlers. As with every other combinator, we start by declaring an unhabited type.

```
data Host
```

We now simply have to extend all the interpretations that we are interested in by providing additional instances dealing with `Host` for the appropriate classes.

The `HasServer` instance is quite straight-forward. It states that any handler using `Host` must take an argument of type `SockAddr` and tells Servant that it can find this information in the `remoteHost` field of wai's `Request` type.

```
instance HasServer api => HasServer (Host :> api) where
  type Server (Host :> api) = SockAddr -> Server api

  route _ handler request respond =
    route (Proxy :: Proxy api) (handler h) request respond
    where h = remoteHost request
```

For other interpretations, instances can be added similarly. Most are of similar complexity or even simpler. For example, for `HasClient`, the `Host` item can be just ignored. Any HTTP client automatically inserts the IP address or host, which means that users of an API should not have to fill in the IP address themselves.

7. Related Work

The field of web programming is so vast that we cannot possibly do it justice. We single out two important aspects and try to position Servant with respect to other libraries and approaches: In Section 7.1, we look at the implementation of Servant, and in Section 7.2, we look at what we achieve, i.e., the relation to other web programming libraries and DSLs from a user perspective.

7.1 The internals of Servant

The expression problem Our approach to solving the expression problem is close to that of Lämmel and Ostermann (2006), in that we essentially use type classes and nothing else. The important point to keep in mind is that Haskell's (GHC's) type system has evolved significantly since 2006. It is the combination of type classes with type families and data kinds that makes this approach so fruitful in our case.

A very popular Haskell solution to the expression problem is "datatypes à la carte" (Swierstra 2008; Bahr and Hvitved 2011), which provides a way to "add new constructors" to an already existing datatype. It works on the term level, which is in principle appealing. However, we have moved Servant to the type level for a combination of reasons. Next to the desired extensibility, it allows us to properly separate the concept of an API from its interpretations in a type-safe way.

Also related are finally tagless encodings (Carette et al. 2009), which can achieve an astonishing amount of extensibility by turning contructors into class methods, and interpretations into instances of the class. While it is entirely conceivable that such an encoding could be used for Servant, it is not immediately clear how all our type-level computation would map to such an approach, and we do believe there is some clarity in having API types around as manifest first-order types, rather than as a collection of possible interpretations.

Generic programming Altenkirch and McBride (2003) have observed in the context of general-purpose datatype-generic programming that the choice of the universe (i.e., the set of datatypes) affects what generic programs we can write. In general, the more general the universe, the fewer generic functions are supported. This important fact has been observed again by several people at various times and has resulted in a wide variety of generic programming approaches and universes that describe different (sub)sets of datatypes (Holdermans et al. 2006; Rodriguez et al. 2008).

Servant explores one extreme end of the spectrum. We are using a deliberately small universe. The API types are not a particularly simple domain, but one we know a lot about, and this domain-specific knowledge allows us to define special-purpose "generic" functions that we otherwise could not. Despite the specific nature, our DSL has still similarities with general-purpose universes: there is a strong correspondence between `:<|>` and binary sums, and between `:>` and (list-like) products (Chakravarty et al.; de Vries and Löh 2014). The way we encode the universes and interpret them in Haskell is also similar, with the difference that most generic programming approaches are not concerned with extensibility.

7.2 The externals of Servant

Haskell web frameworks A few Haskell web frameworks provide ideas that overlap significantly with the features or objectives of Servant.

Most notably, both Yesod and Happstack (Snoyman 2012; Happstack Team) use a reified representation of a web service's *sitemap* – i.e., the URL paths that compose its API. This reification facilitates safety and reuse. Safety, because internal links can be referred to by their constructor rather than by an opaque string, which makes uses of non-existent links hard or impossible. And

reuse, because the sitemap need only be described once. Unlike Servant, defining how the sitemap datatype maps onto routing is an extra step, which can however often be dispensed with via the use of Template Haskell.

Also, handler types in Yesod and Happstack are similarly expressive as the ones in Servant – they can reflect the types of parameters and results. However, there is no independent specification of the API that can be checked against the handler types at compile time, or could be used to derive other functionality such as client functions.

API description languages Closer to the spirit of Servant are approaches that strive to give a unified treatment of web API descriptions.

The rest (rest-core) and api-tools (Dornan and Gundry) packages have at their core a datatype representing what in the REST HTTP web programming paradigm is known as a "resource". These datatypes can be composed (possibly along with further routing information) into full-fledged API descriptions.

Having datatypes represent resources and APIs makes it possible to develop functionality that operates on these datatype; both libraries do this to great effect, generating client libraries, documentation, and (in the case of rest) servers much in the same way as Servant. They do in some respects even go beyond the base functionality of Servant, offering functionality such as versioning or (in the case of api-tools) data migrations out of the box.

Because the complete description of the API is only present at the term level, Haskell client libraries are created via code generation, and api-tools in general uses Template Haskell to great extent. Using values of normal datatypes limits extensibility of data, making it hard to add new components to the description of a resource or API besides those already provided by the library authors.

Finally, a number of API modelling languages, such as WSDL (Christensen et al. 2001), API Blueprint (Blueprint) and RAML (RAML Workgroup), also provide a standardized format for describing web APIs, and can be used to automatically derive functionality for those APIs. The formats used (e.g. JSON) are not full-fledged programming languages, which hampers flexibility and extensibility.

8. Future work and conclusions

The most important conclusion is that Servant actually works – and it works not just in theory, but in practice. It is in use in several places, and has an active community of contributors. In that sense, it has fulfilled and exceeded the expectations of its original authors.

However, the industrial use of Servant has also resulted in several new items for the wish list. Among the most prominent features that will hopefully be implemented soon are out-of-the-box support to reflect authentication requirements as part of a Servant API (with the obvious requirement that a user should easily be able to add support for new mechanisms), and improvements to the routing system that improve the complexity of dispatching to the right route to be sublinear in the number of choices (cf. Section 4.9). There have also been requests for automatic generation of descriptions of Servant APIs in the modelling languages mentioned in Section 7.2.

Another unrelated but appealing project is to explore the design space for a Servant-like library in dependently typed languages such as Agda or Idris. There certainly is a significant amount of type-level computation happening in Servant. In a dependently typed language, we could move that back to the term level without losing any safety. However, it is currently unclear whether there is then an easy match for our extensibility requirements in these languages.

Finally, while Servant itself is all about web APIs, the same technique of domain-specific generic programming can be applied to other domains. There already is an experimental library providing a type-level DSL for maintaining configuration data of a given application that can be interpreted as e.g. both a configuration file and a command line parser (Fischmann and Löh 2015). Another obvious application domain is that of database schemas. We believe that there are many more applications that would benefit from a first-class and extensible type-level description language that admits several interpretations.

References

API Blueprint: Format 1A Revision 8. URL `https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md`. Accessed: 2015-05-22.

rest-core. URL `https://hackage.haskell.org/package/rest-core`. Accessed: 2015-05-22.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 WGP*, pages 1–20. Kluwer, 2003.

P. Bahr and T. Hvitved. Compositional data types. In *WGP 2011*, pages 83–94. ACM, 2011.

J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19:509–543, Sept. 2009.

M. M. T. Chakravarty, G. C. Ditu, and R. Leshchinskiy. Instant generics: Fast and easy. Accessed: 2015-05-22. URL `https://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, Mar. 2001. URL `http://www.w3.org/TR/wsdl`. Accessed: 2015-05-22.

E. de Vries and A. Löh. True sums of products. In *WGP 2014*, pages 83–94. ACM, 2014.

C. Dornan and A. Gundry. api-tools. URL `https://hackage.haskell.org/package/api-tools`. Accessed: 2015-05-22.

R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell 2012*, pages 117–130. ACM, 2012.

M. Fischmann and A. Löh. configifier, 2015. URL `https://hackage.haskell.org/package/configifier`. Accessed: 2015-05-22.

N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part two: Media types, Nov. 1996. URL `http://tools.ietf.org/html/rfc2046`.

Happstack Team. happstack-server. URL `https://hackage.haskell.org/package/happstack-server`. Accessed: 2015-05-22.

S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In *MPC 2006*, pages 209–234. Springer-Verlag, 2006.

R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE 2006*, pages 161–170. ACM, 2006.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell 2008*, pages 111–122. ACM, 2008.

M. Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Apr. 2012.

W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

The RAML Workgroup. RAML™ version 0.8: RESTful API modeling. URL `http://raml.org/spec.html`. Accessed: 2015-05-22.

P. Wadler. The expression problem, Nov. 1998. URL `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`. Email to the `java-genericity` mailing list.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI 2012*, pages 53–66. ACM, 2012.