Servis: A dependently typed DSL for web APIs

Arian van Putten
a.vanputten@uu.nl

July 31, 2016

Abstract

In this paper, we describe a DSL for web APIs named Servis. Servis is heavily inspired by Servant, a haskell type-level DSL for web APIs, but tries to solve identified shortcomings by using a dependently typed language called Idris. Servis uses the universe pattern to deduce types from an API description such that we can verify at compile time that the implementation of the API matches the description.

Furthermore, we take a look at if dependent types can bring more to the table in terms of expressiveness of API descriptions than the original Servant DSL. Specifically, we introduce π-types to describe interdependencies within an API description.

TODO: rewrite abstract *after* I finish the paper :-)

1 Introduction

Servant is an extensible type-level DSL for Haskell, that tries to make web APIs *first-class* [?]. A web API should be a thing that can be named, passed to, or returned by functions. Just like any other term in a language.

It does this by describing the DSL at the type-level. And then uses *type families* (type-level functions) to be able to perform computations on the API specifications. Using these type families, we are able to write different *interpretations* of an API. Example of interpretations are: generating documentation from a specification, generating types of a server implementation, or even generating a client that can interfact with the API.

In this paper, I will describe a port of Servant in Idris, and explore what kind of benefits this brings us.

1.1 An introduction to Servant

I will now give a very short overview of the internals of Servant. We will start by describing the type-level DSL of servant.

```
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
```

Listing 1: The Servant DSL

TODO give a short overview of servant

Servant has a few limitations which we will describe in Section ??.

1.2 Why idris

Idris is a language that in syntax is rather similar to Haskell, but has a more powerful type system. Idris is a dependently-typed language. This means that types are fist-class citizens. We can pass types to functions, let functions create new types, and let types depend on other value-level terms.

Though haskell has type-level functions (called type-families), it does not allow these functions to dpend on term-level values. There is a distinct boundary between type-level and term-level, whilst the two are merged in Idris. This makes doing computations with types a lot more natural (no special constructs are needed, just write functions), but also lets us do some more powerful stuff like letting types actually depend on runtime user input.

Dependent Haskell is currently in the making, and probably planned for around GHC version 8.6 [?]. In the future, it would probably be very feasible to port this library back to Haskell and bring all the benefits along with it.

1.3 Contributions

In this paper, I make the following contributions:

TODO reword

- I point out several problems an limitations of the original Servant DSL
- I show a port of servant to Idris, a dependently-typed language
- I show how new techniques from Idris can help mitigate some of the problems that Servant has.

2 Limitations of Servant

2.1 Term and type-level are not unified in Haskell

As explained in the introduction, the Servant DSL lives in the type-level. In haskell, types and terms are separated. One can lift terms into the type-level using GHC extensions like

XDataKinds, but that only gets us so far. On the term level, Haskell has many great libraries and data structures that can be used. For example, all kinds of string manipulations. It would be useful to be able to use these functions to manipulate APIs, making them truly first-class. Now they are only first-class in the type level.

For example, a useful function would be `path "/users/friends/etc" = "users" :> "friends" :>` to reduce noise in the DSL. But this is currently impossible to write because Strings in the type level are of kind `Symbol` and not `String` so all our favorite string manipulation functions like `splitOn "/"` will not work. This would mean that we would have to re-implement all kinds of utility functions on the type-level.

2.2 A kind of APIs

As mentioned in Section **??**, a delibirate choice has been made to let the Servant DSL live in the open kind $*$. Though this makes the DSL extensible, it also makes the DSL fragile and prone to error.

To give a real-world example, here is a snippet from a user on the Haskell Subreddit that could not get his API to work:

```
type SoundcloudTrackAPI = "tracks" :>
  (    QueryParam "client_id" T.Text
    :> QueryParams "genres" T.Text
    -> Get [JSON] [ST.Track]
  :<|> QueryParam "client_id" T.Text
    :> Capture "id" Int
    :> Get [JSON] ST.Track
  )
```

Listing 2: A malformed API

The problem is hard to spot, but the user typed wrote `->` instead of `:>`. This code compiles fine though. Because the Servant DSL lives in kind $*$, `:>` is of kind `* -> * -> *`. Which is the same kind as the kind of `->`.

Only when one would try to interpret the DSL, like in Listing 3, one would get a type error because no type family pattern matches on `->`.

```
searchTracksByGenre :: Maybe T.Text
                    -> [T.Text]
                    ->  Manager
                    -> BaseUrl
                    -> ClientM [ST.Track]
getTrack            :: Maybe T.Text
                    -> Int
                    -> Manager
                    -> BaseUrl
                    -> ClientM ST.Track
(searchTracksByGenre :<|> getTrack) = client soundcloudAPI
```

.

Listing 3: An interpretation of the API (Listing 2)

Preferably, one would want to define a new open kind `API` such that `(:>) :: API -> API -> API`, but this is not possible in Haskell (or Idris). Instead, one could define a closed kind (using XDataKinds), in which we define the entire Servant DSL. This would mean we would lose extensibility of the DSL, but we gain kind-safety.

Also, as described in Section **??**, the Servant DSL is split up in several semantic portions. For example, `:>` should take an `item` on the left-hand side, which describes either a constraint like `ReqBody` or a path-piece, whilst the right-hand side should be an `api`. These semantic portions are implict and not checked though. So we can create totally non-sensical API types like `type Nonsense = (Int :>Int) :<|> ("hey ":> Int)` that have no interpretation at all.

Instead of having these implicit semantic rules about what kind of arguments an operator can take in the Servant DSL, we could define a distinct closed kind for each semantic portion. Such that the type of `:>` changes to the following: `(:>) :: Item -> API -> API`.

Another implict rule is, that a chain of `:>` should always end in a `Get` or a `Post`. So this should be invalid: `type API = "a" :> ReqBody [JSON] User`. This can also not be checked if we do not add more fine-grained kinds to the DSL.

2.3 Explicit status codes

An API that describes how stuff goes right is good, and API that describes how stuff goes wrong is great. But currently, Servant has no way to indicate in the API description what kind of status codes an endpoint can return. Thus this also does not show up in the generated documentation.

It would be nice to add a list of possible status codes that an endpoint can returns, and then also be able to check whether a server implementation actually only returns one of the specified status codes if something goes wrong (or right).

2.4 Overlapping routes

Another problem with Servant is that there is no protection for overlapping routes. If two routes could resolve to the same path, Servant makes a left-biased choice [?]. It would be nice if we could verify at compile time that no routes in an API specification can overlap.

```
type API   = "users" :> Capture "args" String :> Get [JSON] User
       :<|> "users" :> "favorites" :> GET [JSON] User
```

Listing 4: An example of overlapping routes. The capture will be chosen if args equals "favorites"

3 Servis

In this section, I present a DSL in Idris named Servis. It tries to address each issue listed in Section 2. Firstly, we will define the DSL in which we can write API specifications. Then, we will look at how to write an interpreter of this DSL that generates documentation. Then, we will look at how we would implement HTTP webserver handlers that adhere to specification defined by the API DSL. Next, we'll look at solving the overlapping routes problem by writing proofs in Idris. Finally, we will extend the DSL to add *pi-types*, allowing us to describe dependencies in our API. For example, one might want to describe the size of a list of users in the API, but let the size depend on a query parameter *limit* that limits the response size. It would then force the implementor of the API to only return values smaller or equal to limit in size.

3.1 The DSL

Here we will, step by step, describe the DSL in which we describe APIs. Furthermore, we will address the problem of the open-kindedness of Servant.

3.2 A simple interpreter: Documentation generation

Here we will show a simple interpretation of the DSL, by generating some basic documentation. It's basically a function `docs : API -> String`.

3.3 An interpreter for an HTTP Server

A more advanced example where we define two interpreters. One 'interpreter' that generates types from our DSL such that we can type-check our server handler implementations. (The universe pattern), and another interpreter that routes requests to these handler implementations.

3.3.1 Checking explicit status codes

Shows that we can enforce explicit status codes

3.4 A proof of non-overlapping routes

We will describe types for helping prove that routes in an API definition don't overlap. We do this by providing Decidable Equality proofs for all parts of our DSL.

```
||| A Proof that the GET dtor is injective over its requestType
injGET : (GET x = GET y) -> x = y
injGET Refl = Refl

||| A proof that the POST dtor is injective over its requestType
injPOSTReq : (POST requestType1 responseType1 = POST requestType2 responseType2) -> requ
injPOSTReq Refl = Refl

||| A Proof that the POST dtor is injective over its responseType
injPOSTResp : (POST requestType1 responseType1 = POST requestType2 responseType2) -> res
injPOSTResp Refl = Refl

||| A Proof that a GET is not a POST
getNotPost : (GET responseType = POST requestType responseType1) -> Void
getNotPost Refl impossible

||| A proof that GET is congruent
cong_GET : (prf : responseType1 = responseType2) -> GET responseType1 = GET responseType
cong_GET = cong

||| A proof POST is congruent
cong_POST : (prf2 : responseType1 = responseType2) -> (prf1 : requestType1 = requestType
cong_POST Refl Refl = Refl

(DecEq req, DecEq res) => DecEq (Handler req res) where
  decEq (GET responseType1) (GET responseType2) =
    (case decEq responseType1 responseType2 of
          (Yes prf) => Yes (cong prf)
          (No contra) => No ((\h => contra (injGET h))))
  decEq (POST requestType1 responseType1) (POST requestType2 responseType2) =
    (case decEq requestType1 requestType2 of
          (Yes prf1) => (case decEq responseType1 responseType2 of
                              (Yes prf2) => Yes (cong_POST prf2 prf1)
                              (No contra) => No (\h => contra (injPOSTResp h)))
          (No contra) => No (\h => contra (injPOSTReq h)))
  decEq (GET _) (POST _ _) = No getNotPost
  decEq (POST _ _) (GET _) = No (negEqSym getNotPost)
data DisjointPP : PathPart capture query -> PathPart capture query -> Type where
  ConstD : Not (str = str') -> DisjointPP (Const str) (Const str')

data DisjointPath : Path capture query req res -> Path capture query req res -> Type whe
  OutputsD : Not (handler1 = handler2) -> DisjointPath (Outputs handler1) (Outputs handl
  PathBase : DisjointPP pp1 pp2 -> DisjointPath (pp1 :> p1) (pp2 :> p2)
  PathStep : DisjointPath p1 p2 -> DisjointPath (p :> p1) (p :> p2)


data DisjointAPI : List (Path capture query req res) -> Type where
  Base : DisjointAPI []
  Step : (x : Path capture query req res) -> All (DisjointPath x) xs -> DisjointAPI (x::
```

3.5 Pi-types

Adds dependent pairs to our DSL such that we can describe dependencies between query params, captures and return types. Example:

```
exampleOfDependentPath : Path EmptyU QueryU EmptyU RespU
exampleOfDependentPath =
  QueryParam "limit" NAT :*> \limit => Outputs (GET (BLIST limit USER))

getUsersLimit : (limit : Nat) -> IO (BoundedList limit User)
getUsersLimit limit = pure . take limit $ users

exampleOfDependentPathHandler : el Example.exampleOfDependentPath
exampleOfDependentPathHandler = getUsersLimit
```

3.6 New problems that pi-type create

Here we will describe how pi-types adds problems. For example we cannot easily generate documentation anymore, prove that there are no overlapping routes, or prove that statuscodes are adhered to.

4 Summary

5 Related work

Talk about tel/serv. An inspiration for this project.

6 Conclusions and future work