# Servis: A dependently typed DSL for web APIs

Arian van Putten

a.vanputten@uu.nl

August 21, 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 $\pi$-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 interface 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:

```haskell
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 `->`.

```haskell
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 Universe pattern

The Servis DSL makes use of the Universe pattern to map an API description to types of handlers. A universe is a type `U : Type` which contains names for types and a type family `el : U -> Type` that assigns to every name `a : U` the type of its elements `el a : Type` [?].

In idris, we could define this as the following interface:

```
interface Universe (u : Type) where
  el : u -> Type
```

Listing 5: Universe interface in Idris

In our case, we have a datatype `API : Type`, which describes our API, and `el` maps values to handler types. A Short example:

```
-- we define a value of API
api : API
api = Const "users" :> Capture Int "userId" :> Outputs (Get USER)

getUser : Int -> IO User
getUser = ?someDatabaseMagic

--  el api evaluates to  Int -> IO User
handler : el api
handler = getUser
```

## 3.2 The DSL

So what does this type `API` look like?

### 3.2.1 Handler

Before we will look at routes, lets think of what an endpoint should look like. We should be able to handle the HTTP verbs. We will only implement `GET` and `POST`, because the others are very similar. For a `GET`, we should be able to specify a return type of the handler, and for a `POST`, we should be able to specify a request body type, and a return type.

A first try at describing this as a datatype for our universe could look something like this:

```
data Handler : Type where
  GET : (responseType : Type) -> Handler req res
  POST : (requestType : Type) -> (responseType : Type) -> Handler req res

Universe Handler where
  -- a GET handler just returns something of type responseType
  el (GET responseType) = IO responseType
  -- a POST handler takes the requestBody as argument, and returns a response
  el (POST requestType responseType) = requestType -> IO responseType
```

We could then define a handler for `GET`ting users as follows:

```
record User where
  constructor MkUser
  name : String
  email : String

userHandler : Handler
userHandler = GET User
```

However, once we start implementating an actual server in Section **??**, we will discover that this definition is not ideal. What if we want to write some function that does something different based on the `responseType` of a `GET`? For example, based on value of `responseType`, we need to decide how to encode the response body to JSON. Lets imagine a hypothetical function `selectEncoder` that tries to implement this behaviour:

```
selectEncoder : Type -> Encoder
selectEncoder User = ?jsonEncoderForAUser

userEncoder : Encoder
userEncoder = selectEncoder User
```

Sadly enough, this will give us a compiler error. The problem is that in Idirs, we cannot pattern match on values of `Type`. However, this exact problem can be solved with universes! We can define a data type `ResponseUniverse : Type` which names the `User` type. We can then pattern-match on `ResponseUniverse` instead of `Type`! And whenever we need the `User` type, we can just call `el USER : Type`.

```
data ResponseUniverse : Type where
  USER : ResponseUniverse

Universe ResponseUniverse where
  el USER = User

selectEncoder : ResponseUniverse -> Encoder
selectEncoder USER = ?jsonEncoderForAUser

userEncoder : Encoder
userEncoder : selectEncoder USER
```

This leads us to the following definition of `Handler`:

```
data Handler : (req : Type)  -> (res : Type) -> Type where
  GET : (responseType : res) -> Handler req res
  POST : (requestType : req) -> (responseType : res) -> Handler req res

(Universe req, Universe res) => Universe (Handler req res) where
  el (GET responseType) = IO (el responseType)
  el (POST requestType responseType) =
    el requestType -> IO (el responseType)
```

Listing 6: The new Handler definition

### 3.2.2 Path and PathPart

Of course we do not only get request information from the request body, we also get information about a request by parsing it's request URL. This is where `Path`s and `PathPart`s come into play. When we talk about path parts, we mean sections of an path template that convey parsable information. For example, the following path template has three path parts: `/users/{user_id}?search={query}`. A constant piece in the path `users`, a variable capture `user_id` and a query param `search`. This decomposition of a path template can be written down as a datatype. In this datatype we do not only want to describe the names of the parts of these paths, but also their types. Hence, we will parameterize over a universe similarly to how we defined `Handler`.

```
||| PathPart is a part of a path.
||| @ capture the universe of captures
||| @ query   the universe of query params
data PathPart : (capture : Type) -> (query : Type) -> Type where
  ||| A constant piece of text
  ||| @ path the path part
  Const : (path : String) -> PathPart capture query
  ||| A capture of a variable name of a specific type
  ||| @ name  the name of the variable
  ||| @ type  the type of the variable
  Capture : (name : String) -> (type : capture) -> PathPart capture query
  ||| A query param of a variable name of a specific type
  ||| @ name  the name of the variable
  ||| @ type  the type of the variable
  QueryParam : (name : String) -> (type : query) -> PathPart capture query
```

A `Path` is simply a list if `PathPart`s, which ends in a `Handler`:

```
||| Describes a Path. A Path consists of path parts followed by a handler
data Path : (capture : Type) ->
            (query : Type) ->
            (req : Type) ->
            (res : Type) -> Type where
  ||| Ends a path with a handler
  ||| @ handler the handler
  Outputs : (handler : Handler req res) -> Path capture query req res
  ||| Conses a pathpart to a path
  (:>) : PathPart capture query -> Path capture query req res -> Path capture query req
infixr 5 :>
```

The Path we just described can now be written as

```
   Const "users"
:> Capture "user_id" INT
:> QueryParam "search" STRING
:> Outputs (GET USER)
```

The universe implementation is straightforward for paths, except for that we do not wannt to have spurious arguments in case when we encounter a `Const`:

```
( Universe capture
, Universe query
```

```
) => Universe (PathPart capture query) where
  el (Const path) = ()
  el (Capture name type) = el type
  el (QueryParam name type) = el type


( Universe capture
, Universe query
, Universe req
, Universe resp
) => Universe (Path capture query req resp) where
  -- special case because we don't like () in our handler functions
  el (Const path :> right) =  el right
  el (pathPart :> right) = el pathPart -> el right
  el (Outputs handler) = el handler
```

### 3.2.3   The API Datatype

Finally, we define an API as an non-empty list of Paths.

```
data API : (capture : Type) -> (query : Type) -> (req : Type) -> (res : Type) -> Type wh
  OneOf : (paths : Vect (S n) (Path capture query req res)) -> API capture query req res
```

The universe implementation of an API is the type of a list of handlers. Every handler is of a different type though, so instead of using an ordinary list, we use an heterogenous list, where each element x in xs is of type el x.

## 3.3   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
```

## 3.4   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