Servis: A dependently typed DSL for web APIs

Arian van Putten
a.vanputten@uu.nl

August 22, 2016

Abstract

TODO

1 Introduction

TODO expand and reword

Servant is an extensible type-level DSL for Haskell, that tries to make web APIs *first-class* [2]. 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:

```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 API Interdependencies

Sometimes, one one like to describe even more type-safety than currently possible in Servant. Maybe we want to let the output-type of a request depend on a query parameter. A use case could be the following: We have a route "/users/list?limit=100", and we want to make sure that a handler never returns more than 100 users. Thus the return type of the handler should be a `BoundedList 100 User`. Or more generally, the return type should be `BoundedList limit User`, where `limit : Nat` comes from parsing the URL. The full type of the API will thus depend on some runtime value. This is a textbook example of dependent types, and is currently not possible to implement in Haskell. However, with idris this should be a breeze.

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 [2]. It would be nice if we could verify at compile time that no routes in an API specification can overlap and give the user an error if he violates the overlapping constraint.

2.5 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 possibly also be able to check whether a server implementation actually only returns one of the specified status codes if something goes wrong (or right).

```
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 using Idris's automatic proof search. Finally, we will extend the DSL to add dependent 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` [1].

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? An API DSL should allow us to define routes, to which we can attach handler types.

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 3.5, 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 Paths and PathParts 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

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

A Path is simply a list if PathParts, 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 res
infixr 5 :>

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

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 `Path`s.

```
data API : (capture : Type) ->
           (query : Type) ->
           (req : Type) ->
           (res : Type) -> Type where
  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 An example API

Using this DSL, we will now describe an API for interacting with users in a database. We will start of with defining our domain objects, and intercations we could do with them.

```
record User where
  constructor MkUser
  id : Int
  name : String
  email : String
  password : String


findUserByEmail : String -> IO (List User)
getUsers : (limit : Nat) -> IO (List User)
getUserById : (id : Int) -> IO User
addUser : User -> IO User
```

Listing 7: The User object and its interactions

Next, we will describe the HTTP API for interacting with these users. This includes the need of coming up with universes for captures, query params, request body, and response.

3.4 A simple interpreter: Documentation generation

Writing an interpreter for generating documentation is very easy. Our API description isn't some type-level definition, it's just a plain old datatype that can be passed to functions and can be manipulated.

We start by introducing an interface for docs:

```
interface HasDocs a where
  docs :: a -> String
```

Now, we just define an implementation for `API` for this interface and we're done. Which is just calling `docs` recursively on all its children.

```
(HasDocs req, HasDocs res) => HasDocs (Handler req res) where
  docs (GET responseType) =
    "## GET\nResponse type: " ++ docs responseType

  docs (POST requestType responseType) =
    "## POST\nRequest type: "  ++ docs requestType ++ "\n" ++
    "Response type: " ++ docs responseType ++ "\n"

renderParams : HasDocs query => List (String, query) -> String
renderParams = renderParams' . reverse
  where
    renderParams' : List (String, query) -> String
    renderParams' [] = ""
    renderParams' ((a, b) :: []) =
      "?" ++ a ++ "=<"++ docs b ++ ">"
    renderParams' ((a, b) :: xs) =
      renderParams' xs ++ "&" ++ a ++ "=<"++ docs b ++ ">"

renderPath :
  ( HasDocs capture
  , HasDocs query
  , HasDocs req
  , HasDocs resp) => Path capture query req resp -> String -> String
renderPath (Outputs x) params =
    params ++ "\n" ++ docs x
renderPath ((Const x) :> y) params =
    "/" ++ x ++ renderPath y params
renderPath ((Capture name type) :> y) params =
  "/<" ++ name ++ ":" ++ docs type ++ ">" ++ renderPath y params
renderPath _ _ = ""

( HasDocs capture
, HasDocs query
, HasDocs req
, HasDocs resp
) => HasDocs (Path capture query req resp) where
  docs path =
    "# " ++ renderPath path (renderParams . getParams $ path) ++ "\n"
    where

      getParams : Path capture query req resp -> List (String, query)
      getParams (Outputs x) = []
      getParams ((Const x) :> y) = getParams y
      getParams ((Capture name type) :> y) = getParams y
      getParams ((QueryParam name type) :> y) = (name, type) :: getParams y


( HasDocs capture
, HasDocs query
, HasDocs req
, HasDocs resp
) => HasDocs (API capture query req resp) where
  docs (OneOf (x::[])) = docs x
  docs (OneOf (x::y::xs)) = docs x ++ "\n" ++ oneOf (y::xs)
```

Now `docs userAPI`, assuming that the universes implement the `HasDocs` interface, will evaluate to a very basic form of documentation:

```
# /users/<user_id:CINT>
## GET
 Response type: USER
# /users/find?email=<QSTRING>
## GET
 Response type: LIST USER

# /users?limit=<QNAT>
## GET
```

```
 Response type: LIST USER

# /users
## POST
Request type: USER
Response type: USER
```

3.5 An interpreter for an HTTP Server

Now that we've shown that it is easy to manipulate APIs because they're just datatypes, it's time for a more complicated example. Though it is cool that we can generate documentation from an API description, the useful part that we want to implement is an HTTP Server.

3.5.1 A type for web applications

Lets first ask the question how a web server should look like in Idris. In the Servant paper, they describe the type of http handlers in the `wai` library, which looks like [2]:

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

We will more-or-less use this representation to model our HTTP handlers. they are simply a function from request to response. The actual implementation of a web server based on this model is out of scope for now, but it would be interesting to port something like Haskells' `warp` (which impelements this concept) to Idris.

Instead of having two abstract types `Request` and `Response`, we will assume for simplicity that a `Response` is just a `String`, and that a `Request` is an `URL`, and a requestBody of type `Maybe String`.

```
Application : Type
Application = (url : URL) -> (requestBody : Maybe String) -> IO String
```

Just like in the Servant paper, we will need to modify this type a little bit more to suit our needs. We need to be able to express that an `Application` does not match in case of a routing mismatch, such that we can try another route encompassed in `OneOf`.

Instead of returning a value of `IO String`, we can return a value of `Maybe (IO String)`, where `Just` signals a route match, whilst `Nothing` signals a route mistmatch such that we can try another route.

```
RoutingApplication : Type
RoutingApplication = (url : URL) ->
                     (requestBody : Maybe String)
                     -> Maybe (IO String)
```

3.5.2 Decoding and encoding data

In section 3.2.1, we showed that if we want to implement some decoder, we would have to resort to the universe pattern. So that is what we will do exactly here.

As an example, we will show how we would decode a queryparam. All other decoders follow the same pattern. What we want to do is, given a `queryParam : String`, and a value `v : u`, where `u` is the universe of query param types, and `v` a name of a type in that universe, perform the correct decoding to get a value of type `el v`.

We can summarize this behaviour in an interface:

```
interface Universe u => FromQueryParam u where
  fromQueryParam : (v : u) -> String -> Maybe (el v)
```

similarly, we have interfaces for captures and request bodies:

```
interface Universe u => FromCapture u where
  fromCapture : (v : u) -> String -> Maybe (el v)

interface Universe u => FromRequest u where
  fromRequest : (v : u) -> String -> Maybe (el v)
```

An implementation of `FromQueryParam` for the universe Listing 3.3, would look like this:

```
FromQueryParam QueryU where
  fromQueryParam QINT str = maybeParseInt str
```

The fact that we can pattern match on the universe lets us inspect what type we need to return, and select the right decoder accordingly.

We can use the same technique for encoding responses:

```
interface Universe u => ToResponse u where
  toResponse : (v : u) -> el v -> String
```

For example, we could use this to encode values in the `ReqResUniverse` universe from Listing 3.3:

```
  ToResponse ReqResUniverse where
    toResponse USER (MkUser id name email password) =
      "MkUser " ++ show id ++ " " ++ show name
    toResponse (LIST elem) xs =
      foldr (++) ""  . map (toResponse elem) $ xs
```

3.5.3 Routing requests

We will define the following interface to implement routing requests:

```
interface Universe u => HasServer u where
  route : (v : u) -> (handler : el v) -> RoutingApplication
```

Or if we expaned the definition `RoutingApplication`:

```
interface Universe u => HasServer u where
  route : (v : u) ->
          (handler : el v) ->
          (url : URL) ->
          (requestBody : Maybe String)
          -> Maybe (IO String)
```

The idea of the router is, that given some part of the API `v : u`, and a handler of type `el v`, build an application that can either route succesfully or fail. Preferably, the application uses the handler parameter in its implementation of course.

Lets start by defining an `HasServer` instance for `Handler`. Its task is to simply encode responses to Strings, and to handle request bodies to calculate responses by calling the handler and encoding its result.

```
(FromRequest req, ToResponse res) => HasServer (Handler req res) where
  -- route : Handler req res ->
  --         (handler : IO res) ->
  --         URL -> Maybe String -> Maybe (IO String)
  ||| simply turn an (IO res) into an (IO String)
  ||| using toResponse
  route (GET responseType) handler url requestBody =
    Just (map (toResponse responseType) handler)
  route (POST requestType responseType) handler url requestBody = do
    -- see if there is a requestBody. otherwise fail
    body <- requestBody
    -- try to decode the request. this might fail and return Nothing
    request <- fromRequest requestType body
    -- apply the decoded request to the handler, and encode the result
    Just (map (toResponse responseType) (handler request))
```

Now we will look at how we will route a `Path`. As we know, a path is a description of how we extract information from the URL. Because parsing URLs is out of scope, we will assume that URLs are already parsed when they are fed into our routing machinery as the following datatype:

```
record URL where
  constructor MkURL
  pathParts : List String
  params : List (String, String)
```

```
data QueryUniverse
  = QSTRING
  | QINT
  | QNAT
Universe QueryUniverse where
  el QSTRING = String
  el QNAT = Nat

data CaptureUniverse
  = CINT
Universe CaptureUniverse where
  el CINT = Int

-- Note: We use the same universe for requests and responses.
data ReqResUniverse
  = USER
  | LIST ReqResUniverse
Universe ReqResUniverse where
  el USER = User
  el (LIST a) = List (el a)

userAPI : API QueryUniverse CaptureUniverse ReqResUniverse ReqResUniverse
userAPI = OneOf
  [ Const "users" :> Capture "user_id" CINT :> Outputs (GET USER)
  , Const "users" :>
    Const "find" :>
    QueryParam "email" QSTRING :>
    Outputs (GET (LIST USER))
  , Const "users" :> Outputs (POST USER)
  , Const "users" :>
    QueryParam "limit" QNAT :>
    Outputs (GET (LIST USER))
  ]


-- set the handlers
handlers : el userAPI
handlers =
  [ getUserById
  , findUserByEmail
  , addUser
  , getUsers
  ]
```

Listing 8: API definition for users

We will now give a step-by-step description of the path router.

```
( FromCapture capture
, FromQueryParam query
, FromRequest req
, ToResponse resp
) => HasServer (Path capture query req resp) where
  -- routing a Const is simple. Just make sure the const matches
  -- the next path part
  route ((Const path) :> right)
        handler
        (MkURL pathParts params)
        requestBody =
    case pathParts of
      -- if there are no pathParts left, abort. There is no match!
      [] => Nothing
      (x::xs) => do
        -- make sure the pathPart we consume matches the path part we expect
        guard (x == path)
        --- recurse on the remaining path parts
        route right handler (MkURL xs params) requestBody

  -- To route a capture, we pass the captured variable along to our handler
  route ((Capture name type) :> right)
        handler
        (MkURL pathParts params)
        requestBody =
    case pathParts of
      -- if there are no pathParts left, abort. There is no match!
      [] => Nothing
      (x::xs) => do
        -- try to parse the pathPart as a capture parameter of the right type
        -- abort if this fials
        capture <- fromCapture type x
        -- pass the parsed capture to handler, and recurse on
        -- the other pathparts
        route right (handler capture) (MkURL xs params) requestBody
  -- To route a query param, we look up the query param in the URL,
  -- and pass it to the handler
  route ((QueryParam name type) :> right)
        handler
        (MkURL pathParts params)
        requestBody = do
    -- make sure the query param that was expected exists
    val <- lookup name params
    -- try parse it to the appropriate type
    param <- fromQueryParam type val
    -- remove the parsed query param from the URL
    let newParams = dropWhile ((== name) . fst) params
    -- apply the param to the handler,
    -- and recurse on the url without the query param
    route right (handler param) (MkURL pathParts newParams) requestBody
```

Routing multiple paths is easy. We just keep on trying consecutive paths until one doesn't return `Nothing`:

```
( FromCapture capture
, FromQueryParam query
, FromRequest req
, ToResponse resp) =>
  HasServer (API capture query req resp) where

  route (OneOf (path :: []))
        (handler :: [])
        url
        requestBody =
    -- if no handlers are left but this one.
    -- We should allow it to fail
    route path handler url requestBody
  route (OneOf (path :: x :: xs))
        (handler :: h :: handlers)
        url
        requestBody =
    -- if other handlers are present. Try the handler
    -- and otherwise try others
    case route path handler url requestBody of
      Nothing =>
        route (OneOf (x :: xs))
              -- NOTE: this is to help the compiler select correct type
              (the (HVect _ ) (h :: handlers)
              url
              requestBody
      a => a
```

3.6 A proof of non-overlapping routes

TODO

3.7 Adding API interdependencies

As discussed in Section 2.3, it would be nice to describe more complexly typed APIs, where we allow interdependencies within an API path.

We can easily add this functionality to our DSL by introducing a *dependent* version of `:>`, named `:*>` in the `Path` datatype. What we want is that that the value of the lefthand-side of `:*>` influences the type of the right-hand side.

```
data Path : capture -> query -> req -> res -> Type where
  Outputs : (handler : Handler req res) -> Path capture query req res
  (:>) :  (pathPart : PathPart capture query) ->
          (path : Path capture query req res) -> Path capture query req res
  -- The RHS of :*> now takes a function instead of a path. that calculates a path
  (:*>) : (pathPart : PathPart capture query) ->
          (path : (el pathPart -> Path capture query req res)) ->
          Path capture query req res
```

This is enough to express basic dependent APIs.

A good usecase for these kind of dependent types is for endpoints that return a `BoundedList`. A bounded list in idris is a list that can only contain upto a certain amount of elements. It is defined as follows:

```
data BoundedList : Nat -> Type -> Type where
  Nil : BoundedList n a
  (::) : a -> BoundedList n a -> BoundedList (S n) a
```

Lets extend our Universe of Listing 3.3 to include bounded lists:

```
data ReqResUniverse
  = USER
  | LIST ReqResUniverse
  | BLIST Nat ReqResUniverse
Universe ReqResUniverse where
  el USER = User
  el (LIST a) = List (el a)
  el (BLIST n a) = BoundedList n (el a)
```

And lets modify the domain function that queries the database for users in Listing 3.3 to return bounded lists:

```
getUsers : (limit : Nat) -> IO (BoundedList limit User)
```

We can now describe an API in which we can describe the dependency between the query parameter and the size of the returned list of users.

```
getUsersPath : Path CaptureUniverse QueryUniverse ReqResUniverse ReqResUniverse
getUsersPath = Const "users" :>
               QueryParam "limit" QNAT :*>
               \limit => Outputs (GET (BLIST limit USER))

handler : el getUsersPath
handler = getUsers
```

Next, we will need to add extra cases to the `HasServer` implementation of `Path` to handle the dependency. We will only show the implementation for `QueryParam`, but the changes to make `Capture` work are identical.

The idea is, that instead of only passing the parsed query parameter value to the handler, we also pass it along to the right hand side of `:*>`:

```
  route ((QueryParam name type) :*> right)
        handler
        (MkURL pathParts params)
        requestBody = do
    val <- lookup name params
    param <- fromQueryParam type val
    let newParams = dropWhile ((== name) . fst) params
    -- BEFORE :
--  route right
--        (handler param)
--        (MkURL pathParts newParams)
--         requestBody
    -- WITH ADDED DEPENDENCY:
    route (right param) -- <- note the extra argument passed to RHS
          (handler param)
          (MKURL pathParts newParams)
          requestBody
```

And that's it. we now have added dependent types to our API!

3.8 New problems that dependent types create

The dependent types are nice, but our datatype is now a Higher-order abstract syntax instead of a first-order abstract syntax. Which means the datatype contains a function. This makes it difficult to generate documentation as we need to provide a value to apply to the function to traverse the datatype. It isn't impossible though. We could introduce the following interface:

```
interface Universe u => HasSample u where
  sample : (v : u) -> el v

data ExampleU = INT | STRING

Universe ExampleU where
  el INT = Int
  el STRING = String

HasSample ExampleU where
  sample INT = 0
  sample STRING = "A string"
```

Now in our implementation of `HasDocs` we can just call `sample` every time we encounter a function:

```
HasDocs (Path capture query req res) where
  docs ((Capture str type) :*> f) =
    "super nice docs"  ++ docs (f (sample type))
```

The real problem though is that this representation breaks the overlapping path detector. The overlapping path detector uses the feature of idris to search proofs automatically. But that proof search isn't perfect, and won't be able to prove that paths overlap or not if we have arbitrary functions in our datatypes.

A solution this problem would be that library authors would have to provide their own proof of whether or not their api definitions overlap, but this can be very cumbersome and thus is undesired.

4 Related work

This project was heavily inspired by `https://github.com/tel/serv`. Which is a closed-kinded Servant alternative. When I stumbled upon it, I got very curious what would happen if we would port it to Idris. The serv source code is kind of dense because Haskell isn't a very nice language to do type-level programming in and it got me curious whether or not idris would improve readability. Which it did!

Also, once I had a working example of Serv in idris, I wanted to see if idris could add any new tricks to it. This made me implement dependent types and route conflict resolution.

5 Conclusions and future work

Idris allows us to elegantly describe APIs with plain ol' datatypes. We can then use the Universe pattern to map these datatypes to handler function types. These handler function types can then be made routable by writing an interpreter of the API datatype that decodes the incoming requests and passes the correct parameters to the handlers.

Idris makes it easier to write utility functions, like documentation generation or code generation, because our API definition does not live in the type-level anymore, because type-level and term-level are united in idris.

We were able to solve the common mistake of overlapping routes in Servant as well, by using Idris's proof search capabilities.

Finally, we were able to implement an dependent API, that allows interdependencies within paths to be able to describe even more powerful APIs.

In the future, it would be interesting to look at if it's possible to enforce in the type-system that handlers only return HTTP Status codes that are defined in the API description. I explored some ideas on how to implement this, but did not have the time to do an implementation of this.

References

[1] Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming with dependent types. In *Proceedings of the 2006 International Conference on Datatype-generic Programming*, SSDGP'06, pages 209–257, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. Type-level web apis with servant: An exercise in domain-specific generic programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 1–12, New York, NY, USA, 2015. ACM.