Servis: A dependently typed DSL for web APIs

Arian van Putten
a.vanputten@uu.nl

August 25, 2016

Abstract

In this paper we describe a DSL for web APIs named Servis. Servis is inspired by Servant, a haskell type-level DSL for web APIs. We identify several issues with the current implementation of Servantm and show how Servis solves these by using features of the dependently typed language called Idris.

Specifically, we show how dependent types can help use describe dependencies within an API, how we can check at compile time that routes in an API dont overlap, and how dependent pairs can let us verify that status codes described in an API are adhered to.

1 Introduction

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. By 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, We will describe Servis, a DSL similar to Servant written in Idris. We will point out various issues with the current Servant DSL, and then show how we can solve this in Idris.

1.1 An introduction to Servant

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

Servant lives in the type-level, and defines its language as part of the kind of types named `*`, which is an open kind. This has been done on purpose so that the language becomes extensible. We can always add new constructs to the DSL by creating new data definitions.

```haskell
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
data PlainText

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

Listing 1.1: The Servant DSL and an example of an API definition

It then uses type classes and type families to describe operations on this type-level DSL. For example, a documentation generation type class exists:

```haskell
docs :: HasDocs api => Proxy api -> String
```

A more exotic example is the server type class, that transforms an API into a web server:

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

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

echo :: Server Echo
echo txt = return txt

main :: IO ()
main = run 8080 (serve (Proxy Echo) echo)
```

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 first-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 depend 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 like special syntax for type-families; you can just write functions like you would normally do with term-level values. It also allows us to do more powerful things 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

- We point out several problems and limitations of the original Servant DSL
- We show a port of servant to Idris, a dependently-typed language
- We 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`. 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. But there is no way to lift existing term-level functionality to the type-level, which makes an API description only first-class on the type level.

For example, a useful function would be
`path "/users/friends/etc" = "users" :> "friends" :> "etc"` 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 useful 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 1.1, a deliberate 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 2.1: A malformed API

The problem is hard to spot, but the user wrote `->` instead of `:>`. This code compiles fine though. Because the Servant DSL lives in kind $*$, `:>` is of kind $* \to * \to *$. The type constructor of functions, `->` is also of kind $* \to * \to *$, so there is no way for the compiler to tell that we used the wrong operator here.

Only when one would try to interpret the DSL, like in Listing 2.2, 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 2.2: An interpretation of the API (Listing 2.1)

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 seen in Listing 1.1, 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, however, are implict and unchecked, 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 would like to describe even more type-safety than is currently possible in Servant. Maybe we want to let the output-type of a request depend on a query parameter. A usecase 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 does become possible.

2.4 Overlapping routes

Another problem with Servant is that there is no protection for overlapping routes. If two routes resolve to the same path, Servant makes a left-biased choice [2]. It would be useful that instead of making a design choice where some routes are shadowed without warning, the compiler could inform the user about a potential mistake.

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

Listing 2.3: An example of overlapping routes. The capture will be chosen if args equals "favorites"

2.5 Explicit status codes

It is great that Servant lets us give a presice description about the structure and types of our web APIs. However, it does now allow us to describe any form of errors within the API types. It is important for a developer to know how an API call could fail, such that he can handle it sufficently. Also, when generating documentation from an API type, it one would like to have this information at hand to inform the users of your API on what kind of errors they can expect.

This would not be hard to add to Servant. A type-level list of error types paired with the return type of an API endpoint would suffice. However, it would also be interesting to explore if we can enforce that the code that implements a server for an API, actually only returns the error codes that are documented.

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 can define this as the following interface:

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
```

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

Listing 3.1: Universe interface in Idris

```
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, we must decide 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 implementing an actual server in Section 3.5, we will discover that this definition is not ideal. What if we would want to write a function that needs to decide its behaviour based on the `responseType` of a `GET`? For example, we have a piece of functionality

in our code base, where we need to decide how to encode a response body to JSON, based on the value of `responseType`. Lets try to implement such a function `selectEncoder` and see what road blocks we hit:

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

userEncoder : Encoder
userEncoder = selectEncoder User
```

Sadly enough, this code 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 as a data constructor `USER : ResponseUniverse`. 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
```

We now change the definition of `Handler` to be parameterised over a request universe (`req : Type`) and a response universe (`res : Type`):

```
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 3.2: 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 a 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`.

```
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 `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 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 `/users/{user_id}?search={query}`, can be described as follows with these new data types:

```
   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 want 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 that can be chosen from.

```
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`.

```
( Universe capture
, Universe query
, Universe req
, Universe res
) => Universe (API capture query req res) where
  el (OneOf xs) = HVect (map el xs)
```

3.3 An example API

Using this DSL, we will now describe an API for interacting with users in a database. We will start off with defining our domain objects, and interactions we can perform on them. These definitions can be found in Listing 3.3.

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

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

Listing 3.3: 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 query params, request body, and response. We will use this example in the upcoming sections to show the behaviour of different interpreters.

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

-- 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" :>
    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 =
  [ findUserByEmail
  , addUser
  , getUsers
  ]
```

Listing 3.4: API definition for users

3.4 A simple interpreter: Documentation generation

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

We start by introducing an interface for docs, which just tells us how to turn something into a string:

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

Now, we just need to define an implementation for `API` for this interface and we are done. To

implement this, we will of course also needs to implement the interface for `Handler`, `PathPart` and `Path`.

To get an idea on how this would work, we will show the implementation of `Handler` and `API`:

```
(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"

( 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/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 have shown a small example on how to do computations on API descriptions, it is time for a less contrived example. We would like to implement handlers for our API and then serve these on the web.

3.5.1 A type for web applications

Before we write such an interpreter, lets first ask the question what a web server should look like in Idris. In the Servant paper, the authors describe the type of http handlers in the `wai` library, which looks like [2]:

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

This is a natural way to think of an HTTP application. A request comes in, and we apply a function of type `Application` to get a response, which can then be sent back.

We will more-or-less use this representation to model our HTTP handlers.An 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`. This leads us to the following definition for `Application`:

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

The router we are going to build uses the same mechanism as described in the Servant paper. In order to be able to write a router, we will have to modify the definition of `Application` slightly [2]. We need to be able to express that the router can not match the API path with the incoming request. In case of a mismatch, the router should then try if the next available path does match the request. Instead of letting the application return a value of `IO String`, let it return a value of `Maybe (IO String)`, where `Just a` 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. We will use this technique here for implementing decoding of requests, and encoding of responses.

As an example, we will show how we would decode a query parameter. 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 `QueryUniverse` in Listing 3.3, would look like this:

```
FromQueryParam QueryUniverse 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

To route a request, we should be able to pattern match on our API, (which is a universe), and apply data from the request to a handler function whose type is dependent on the API universe. This leads us to the following definition for a router:

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

Or if we expand the definition of `RoutingApplication`:

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

The first part of the API for which we will implement the routing mechanics is the `Handler`. Defining this router consists of two cases: A idrisGET or a `POST`. During a `GET`, we just have to execute the handler which will then give us a response. For a `POST`, we need to read the request body from the request, decode it to the right type, and apply it to the handler to get a response:

```
(FromRequest req, ToResponse res) => HasServer (Handler req res) where
  -- route : Handler req res ->
  --         (handler : IO (el res)) ->
  --         URL -> Maybe String -> Maybe (IO String)
  ||| simply turn an (IO (el 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))
```

Next up is routing a `Path`. 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)
```

We will now give a step-by-step description of the path router. We can check if this URL matches the `Path` description, and extract the appropriate query parameters and captures.

In case of a `Const`, we will have to check if our URL starts with the same part as the name of the `Const`:

```
  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
```

In case of a capture, we almost do the same thing, but instead of trying to match the path parts name, we try to parse it as the appropriate type using `fromCapture`:

```
  route ((Capture name type) :> right)
        handler
        (MkURL pathParts params)
        requestBody =
    case pathParts of
      [] => 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
```

In case of a query parameter, we do a lookup in the URL if the expected query parameter was present, and then try to parse it as the appropriate type:

```
  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 does nott 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 Explicit status codes

To better document the behaviour of our API to users of the API, we can add a list of status codes that an endpoint can return by modifing `Handler`:

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

However, it would be convenient if we could automatically deduce a proof that our handlers never return other status codes than are in that list. However, this gets impossible quickly because not all programs are decidable. Instead, we can let our `Universe` implementation generate a type that forces the implementor of the API to provide a proof for us. We can accomplish this by letting the universe implementation return a dependent pair. A dependent

pair in Idris is a tuple where the second element proves some property about the first element. In our case, the first element is the status code, and the second element is the proof that it is one of the pre-defined status codes in our API.

We will now show how we would implement for GET:

```
(Universe req, Universe res) => Universe (Handler req res) where
  el (GET req res statuses) =
    (DPair Int (\n => Elem n statuses), IO (el responseType))
```

So now instead of just returning the IO action, we also return the status code, paired with its proof:

```
api : Handler EmptyU EmptyU
api = GET VOID [200,401,404]

-- A proof that 401 is an acceptable status code
the401IsInStatusCodes : Elem 401 [200, 401, 404]
the401IsInStatusCodes = There Here

handler : el api
handler = (MkDPair 401 (the401IsInStatusCodes), putStrLn "hey")
```

We can now modify the router such that it can use the status code to write it to the response:

```
route (GET responseType statuses)
      (MkDPair status proof, handler)
      url requestBody =
  Just $ do
    putStrLn (show status)
    map (toResponse responseType) handler
```

3.7 A proof of non-overlapping routes

One of the weaknesess of Servant is that it is possible to define routes that overlap. It would be great if we could give a compiler error in Servis if two routes overlap.

In Idris, we can describe propositions using types. And then allows us to make proofs of these propositions by implementing a program of that type. Also, Idris comes with a proof searcher that can automatically search for a proof for a certain proposition. It will do a best effort to find a proof, and if it can not find a proof in time the compiler halts with an error. If it did not find a proof, it does not mean there is no possible proof though. The proof searcher is naive. It will only search 100 levels of recursion deep, and it will not do contradictory proofs. So in some cases, the user will have to provide a proof himself.

We would like to automatically prove the proposition that for every path that we add to our API datatype, it does not overlap with any other path in the API datatype. This way, it is a compiler error if paths in the API overlap. We call this proposition *disjointness*.

Query parameters make comparing URLs clumsy because there can be duplicate query parameters and query parameters can appear in any order. Hence, for simplicity sake, we will omit query parameters when testing this proposition. We do this by defining the following filter function:

```
wqp : Path a b c d -> Path a b c d
wqp (Outputs handler) = Outputs handler
wqp ((QueryParam name type) :> y) = wqp y
wqp (a :> y) = a :> wqp y
```

Now we only have to be concerned with `Const` and `Capture`. Because `Capture` signifies a variable substitution, it will always conflict with a `Const`. E.g., `/users/hey` conflicts with `/users/:var` because `:var` could be substituted by `hey`.

This leads us to defining the following datatype, which allows us to prove that a `PathPart` is disjoint if two `Const` values have different names:

```
data DisjointPP : PathPart capture query -> PathPart capture query -> Type where
  ConstD : IsNo (decEq str str') -> DisjointPP (Const str) (Const str')
```

To make a decision whether or not the string are equal at compile time, we use the `DecEq` interface.

```
interface DecEq t where
  decEq : DecEq t => (a:t) -> (b:t) -> Dec (a = b)

||| Decidability. A decidable property either holds or is a contradiction.
data Dec : Type -> Type where
  ||| The case where the property holds
  ||| @ prf the proof
  Yes : (prf : prop) -> Dec prop
  ||| The case where the property holding would be a contradiction
  ||| @ contra a demonstration that prop would be a contradiction
  No  : (contra : prop -> Void) -> Dec prop
```

We then create a proposition that can ony be proven if a decision is made that a property does not hold:

```
data IsNo : Dec a -> Type where
  ItIsNo : IsNo (No x)
```

The fact that we can not construct a value of `DisjointPP (Const x) (Capture x y)` indicates that a `Const`, and a `Capture` always overlap, which is the behaviour that we wanted to express.

Showing that a `Path` is disjoint has more cases.

```
data DisjointPath : Path capture query req res ->
                    Path capture query req res ->
                    Type where
```

First of all, if two path parts are disjoint, a path that starts with these path parts is disjoint:

```
  PBasePP : DisjointPP pp1 pp2 -> DisjointPath (pp1 : p1) (pp2 : p2)
```

Next, if a path that is not just an `Outputs` is compared to a path that i just an `Outputs`, then these paths are disjoint:

```
  PBaseOutputs1 : DisjointPath (pp :> p) (Outputs h)
  PBaseOutputs2 : DisjointPath (Outputs h) (pp :> p)
```

Now we can prove an entire path disjoint by induction:

```
  PStep : (inducHypth : DisjointPath p1 p2) -> DisjointPath (pp1 :> p1) (pp2 : p2)
```

Now every time we add a path to an API, we want to prove that each path in API is disjoint from the path we want to add. Luckily, Idris comes with a proposition named `All` that checks if a proposition holds for all elements in a list:

```
data All : (P : a -> Type) -> List a -> Type where
  Nil : {P : a -> Type} -> All P Nil
  (::) : {P : a -> Type} -> {xs : List a} -> P x -> All P xs -> All P (x :: xs)
```

We can now define an operator that adds a path to an existing lists of paths, but only if we are able to provide a proof that the path is not in that list:

```
  (+>) : (path : Path c q rq rs) ->
         (xs   : List (Path c q rq rs)) ->
         {auto all : All (\x => DisjointPath (wqp path) (wqp x)) xs} ->
         List (Path c q rq rs)
  (+>) x xs = x::xs
```

The `auto` keyword signifies that Idris will try to automatically find a value fo the implicit argument `all`. If it fails, it will give a compiler error. But if it succeeds, we can be sure that we did not add an overlapping path to our list of paths.

We can now make sure that our API definitions are sane. This will compile:

```
  api' : List (Path CapU EmptyU EmptyU RespU)
  api' =  Const "users" :> Capture "name" STRING :> Outputs (GET USER)
       +> Const "users" :> Capture "name" STRING
                        :> Const "friends" :> Outputs (GET (LIST USER))
       +> Nil
```

However, if we introduce an overlap, the compiler will tell us that this api definition is invalid and that we need to fix our code:

```
apiOverlap : List (Path CapU EmptyU EmptyU RespU)
apiOverlap
  =  Const "users" :> Capture "name" STRING :> Outputs (GET USER)
  +> Const "users" :> Const "all"           :> Outputs (GET (LIST USER))
  +> Nil

When checking right hand side of apiOverlap' with expected type
        List (Path CapU EmptyU EmptyU RespU)

When checking argument ok to function Servis.API.+>:
  Can't find a value of type
    All (\x =>
          DisjointPath (Const "users" :> Capture "name" STRING :> Outputs (GET USER))
                       (wqp x))
        [Const "users" :> Const "all" :> Outputs (GET USER)]
```

3.8 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 has the following signature:

```
data BoundedList : Nat -> Type -> Type where
```

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 a bounded list that is *dependent* on the `limit` parameter.

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

We can now describe an API in which we can express 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 almost identical and left as an exercise to the reader.

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 param) -- <- note the extra argument passed to RHS
          (handler param)
          (MKURL pathParts newParams)
          requestBody
```

3.8.1 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 proof search isn't perfect will be exhausted after a while. Take ourexample with the `limit : Nat` parameter. For Idris to find a proof, it would try every number in `Nat` to make sure that the part of our path after the dependency does not overlap. However, there are a countable infinite number of `Nat`s so Idris would never be able to complete the proof. Hence the auto-searcher will stop searching for a proof after a while and quit.

A solution is that library authors can provide the proof themselves, similarly to how this was solved for explicit status code checks. However, writing these proofs for paths might defeat the point. We wanted them to be automatic to have a sanity check to make sure people are not caught off guard if certain paths are not routed. Having to write a proof for each dependent path you define makes the library cumbersome to use and thus might be undesired.

4 Conclusions

Idris allows us to elegently describe APIs as simple terms. We can then use the Universe pattern to calculate types for handler. These handler functions can then be made routable such that we have a function web server.

Idris makes it easier to write utility functions, like documentation generation because our API definition does not live in the type-level as it did in Servant.

We were able to solve several shortcomings of servant: By not using an open kind, we eliminated the possibility of writing APIs that are not interpretable. We also also the possibility of defining overlapping routes. W made status codes explicit in the API and gave library authors the tools to prove that their servers adhere to these status code definitions. Finally, we were able to implement an dependent API, that allows interdependencies within paths to be able to describe even more powerful APIs.

5 Related work

This project started with me looking at`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 write something similar 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.

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.