Servant

A Type-Level DSL for Writing and Interacting with web APIs

Julian K. Arni

May 5, 2015

Outline

Section 1

Prelude

The identity function:

```
a -> a
```

The identity function:

```
a -> a
```

The identity (echo) server:

```
Request -> IO Response
```

The reverse function:

```
[a] -> [a]
```

The reverse function:

```
[a] -> [a]
```

The reverse server:

```
Request -> IO Response
```

An informal specification of the identity function:

> *For any input, the function must return the input unchanged.*

An informal specification of the identity function:

> *For any input, the function must return the input unchanged.*

An informal specification of the echo function:

> *For any request, if the request has URL `"echo"`, and the request method is `POST`, and the `Content-Type` header is acceptable according to the `Accept` header of the request, then the function must return the request body unchanged, with the Content-Type header of the response set to that of the request.*

Section 2

Leaving Values Aside

What would types look like if we didn't have to worry about their inhabitants?

What would types look like if we didn't have to worry about their inhabitants?
Like a DSL!

What would types look like if we didn't have to worry about their inhabitants?

```
Exp := Method CTypes Type
     |'ReqBody' CTypes Type ':>' Exp
     | StringLit ':>' Exp
     | Exp ':<|>' Exp
     ...

CTypes := ''[' CType, ... ']'
CType := 'JSON'
       | 'XML'
       | 'HTML'
       ...

Method := 'Get'
        | 'Post'
        ...
```

Terms are (usually) uninhabited types:

```
data a :> b
data Get (ctypes :: [*]) a
data a :<|> b
data ReqBody (ctypes :: [*]) a
...
```

An example:

```
type EchoAPI = "echo" :> ReqBody '[JSON, XML, ...] a
                      :> Post '[JSON, XML, ...] a
```

An example:

```
type EchoAPI = "echo" :> ReqBody '[JSON, XML, ...] a
                      :> Post '[JSON, XML, ...] a
```

Most of our informal specifications are now expressed in the type level.

```
type EchoAPI = "echo" :> ReqBody '[JSON, XML, ...] a
                      :> Post '[JSON, XML, ...] a
```

Most of our informal specifications are now expressed in the type level.

This doesn't *do* anything yet. No values inhabit this type. And we can't get any functionality from it.

```
type EchoAPI = "echo" :> ReqBody '[JSON, XML, ...] a
                      :> Post '[JSON, XML, ...] a
```

Most of our informal specifications are now expressed in the type level.
This doesn't *do* anything yet. No values inhabit this type. And we can't get any functionality from it.
But it is a *unified* language for describing web APIs.

Section 3

Interpreting Types

Subsection 1

Intepreting Types as Values

A simple example.

A simple example.

```
type Five = Add 1 (Add 2 2)
```

A simple example.

```
type Five = Add 1 (Add 2 2)
```

This is just a type.

A simple example.

```
type Five = Add 1 (Add 2 2)
```

This is just a type. But we can *interpret* it!

A simple example.

```
type Five = Add 1 (Add 2 2)

class AsInt a where
    asInt :: Proxy a -> Int
```

A simple example.

```
type Five = Add 1 (Add 2 2)

class AsInt a where
    asInt :: Proxy a -> Int

instance (AsInt a, AsInt b) => AsInt (Add a b) where
    asInt _ = asInt (Proxy :: Proxy a)
            + asInt (Proxy :: Proxy b)
```

A simple example.

```
type Five = Add 1 (Add 2 2)

class AsInt a where
    asInt :: Proxy a -> Integer

instance (AsInt a, AsInt b) => AsInt (Add a b) where
    asInt _ = asInt (Proxy :: Proxy a)
            + asInt (Proxy :: Proxy b)

instance (KnownNat a) => AsInt a where
    asInt _ = natVal (Proxy :: Proxy a)
```

This is the essence of *servant*

This is the essence of *servant*
It is *extensible*.

This is the essence of *servant*
It is *extensible*.
Exercise: add support for `Mult` without changing any of the code we wrote so far.

This is the essence of *servant*
It is *extensible*.
Exercise: add support for `Mult` without changing any of the code we wrote so far.

```
type Application = Request -> IO Response
class HasServer a where
  type Server a
  route :: Proxy a -> Server a -> Application
```

```
class HasServer a where
  type Server a
  route :: Proxy a -> Server a -> Application

instance ToJSON a => HasServer (Get a) where
  type Server (Get a) = IO a
  route Proxy val req = encode <$> val

encode :: ToJSON a => a -> ByteString
```

```
class HasServer a where
  type Server a
  route :: Proxy a -> Server a -> Application

instance (HasServer a, KnownSymbol path)
  => HasServer (path :> a) where
  type Server (path :> a) = Server a
  route Proxy val req = if req `startsWith` p
      then route pr val (req `stripPath` p)
      else return notFound
    where pt = symbolVal (Proxy :: Proxy path)
          pr = Proxy :: Proxy a

encode :: ToJSON a => a -> ByteString
```

Subsection 2

Intepreting Types as Types

So far we've been focusing on types as data.

```
class HasServer a where
  type Server a
  route :: Proxy a -> Server a -> Application
```

```
class HasServer a where
  type Server a  -- Here!
  route :: Proxy a -> Server a -> Application
```

```
class HasServer a where
  type Server a
  route :: Proxy a -> Server a -> Application

instance MimeRender ct a => HasServer (Get ct a) where
  type Server (Get a) = IO a
  route Proxy val req = mimeRender <$> val

class MimeRender ct val where
  mimeRender :: val -> ByteString

data JSON

instance ToJSON val => MimeRender JSON val where
  mimeRender = encode

...
```

```
type family IsElem link api :: Constraint where
  IsElem e (sa :<|> sb) = IsElem e sa `Or` IsElem e sb
  IsElem (a1 :> b1) (a1 :> b2) = IsElem b1 b2
  ...
  IsElem (Get ct typ) (Get ct' typ) = IsSubList ct ct'
```

```
type family IsElem link api :: Constraint where
  IsElem e (sa :<|> sb) = IsElem e sa `Or` IsElem e sb
  IsElem (a1 :> b1) (a1 :> b2) = IsElem b1 b2
  ...
  IsElem (Get ct typ) (Get ct' typ) = IsSubList ct ct'

class HasLink endpoint where
  type MkLink endpoint
  toLink :: Proxy endpoint -> Link -> MkLink endpoint

safeLink :: (IsElem endpoint api, HasLink endpoint)
  => Proxy api -> Proxy endpoint -> MkLink endpoint
```

Section 4

Putting it all together

What does *using* this actually look like?

```
data Person ...
data About ...

type PersonApi = Capture "name" String :> Get '[JSON] (Maybe Person)
            :<|> Get '[JSON] [Person]
            :<|> ReqBody '[JSON] Person :> Post '[] ()

type AboutApi = Get '[XML] About

type TheApi = "person" :> UserApi
         :<|> "about" :> AboutApi

personServer :: Server PersonApi
personServer = \name -> lookupInDb name
          :<|> return lookupAll
          :<|> \person -> saveInDb person

aboutServer :: Server AboutApi
aboutServer = ...

theServer :: Server TheApi
theServer = personServer :<|> aboutServer
```

```
apiProxy :: Proxy TheApi
apiProxy = Proxy

main :: IO ()
main = run 8000 $ serve apiProxy theServer
```

```
type PersonApi = Capture "name" String :> Get '[JSON] (Maybe Person)
            :<|> Get '[JSON] [Person]
            :<|> ReqBody '[JSON] Person :> Post '[] ()

type AboutApi = Get '[XML] About

type TheApi = "person" :> UserApi
         :<|> "about" :> AboutApi

getPerson :: String -> EitherT ServantError IO (Maybe Person)
getPeople :: EitherT ServantError IO [Person]
postPerson :: Person -> EitherT ServantError IO ()
getAbout :: EitherT ServantError IO About
(getPerson :<|> getPeople :<|> postPerson :<|> getAbout)
    = client url apiProxy
  where url = BaseUrl "localhost" 8080
```