Servant

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-03-05

Goals

- A case study for type-level programming.
- An API description language.
- Using `servant` to define clients and servers.

Describing an API

A simple web API

```
GET    /entry         get all entries
POST   /entry         post a new entry
GET    /entry/<n>     get entry of given number
DELETE /entry/<n>     remove entry of given number
```

Describing the API as a Haskell type

```haskell
type KeyVal =
       "entry" :> Get '[JSON] [Entry]
   :<|> "entry"
          :> ReqBody '[JSON] Text
          :> PostCreated '[JSON] Entry
   :<|> "entry" :> Capture "n" Int
          :> Get '[JSON] Entry
   :<|> "entry" :> Capture "n" Int
          :> DeleteNoContent '[JSON] NoContent
```

Describing the API as a Haskell type

```haskell
type KeyVal =
         "entry" :> Get '[JSON] [Entry]
    :<|> "entry"
               :> ReqBody '[JSON] Text
               :> PostCreated '[JSON] Entry
    :<|> "entry" :> Capture "n" Int
               :> Get '[JSON] Entry
    :<|> "entry" :> Capture "n" Int
               :> DeleteNoContent '[JSON] NoContent
```

Let's first look at how this is even kind-correct.

API combinators

```haskell
data a :<|> b
infixr 8 :<|>

data path :> a
infixr 9 :>

data Capture (sym :: Symbol) a
data ReqBody (contentTypes :: [*]) a

type Get = Verb GET 200
type PostCreated = Verb POST 201
type DeleteNoContent = Verb DELETE 204

data Verb
  method (status :: Nat) (contentTypes :: [*]) a
```

Implementations are irrelevant.

Meaning

- `a :<|> b` is the union of the APIs `a` and `b`,

Meaning

- `a :<|> b` is the union of the APIs `a` and `b`,
- `"foo" :> a` describes an API reachable under path `"foo"`,

Meaning

- `a :<|> b` is the union of the APIs `a` and `b`,
- `"foo" :> a` describes an API reachable under path `"foo"`,
- `Get '[JSON] Entry` describes an endpoint returning an `Entry` with content type `JSON`,

Meaning

- `a :<|> b` is the union of the APIs `a` and `b`,
- `"foo" :> a` describes an API reachable under path `"foo"`,
- `Get '[JSON] Entry` describes an endpoint returning an `Entry` with content type `JSON`,
- `Capture "n" Int :> a` describes an API reachable under a captured integer path,

Meaning

- `a :<|> b` is the union of the APIs `a` and `b`,
- `"foo" :> a` describes an API reachable under path `"foo"`,
- `Get '[JSON] Entry` describes an endpoint returning an `Entry` with content type `JSON`,
- `Capture "n" Int :> a` describes an API reachable under a captured integer path,
- `ReqBody '[JSON] Text :> a` describes an API reachable only if an appropriate request body is present.

More API combinators

```haskell
data JSON
data PlainText
```

```haskell
data StdMethod =
      GET | POST | HEAD | PUT | DELETE
    | TRACE | CONNECT | OPTIONS | PATCH
```

```haskell
data NoContent = NoContent
```

Overloaded strings and numbers

```
GHCi> import GHC.TypeLits
GHCi> :k "foo"
"foo" :: Symbol
GHCi> :k 42
42 :: Nat
```

Entries

Unlike all the others, `Entry` is a domain-specific type for our API and to be defined in our program:

```
data Entry =
  Entry
    { entryId   :: Int
    , entryText :: Text
    }
```

How to model a web server

Request-response

At a high-level, a web server receives requests and sends responses.

Request-response

At a high-level, a web server receives requests and sends responses.

The `wai` package (short for web application interface) is built on top of this simple model:

```
type Application =
      Request
   -> (Response -> IO ResponseReceived)
   -> IO ResponseReceived
```

(This is in *continuation passing style*.)

Running an application

From `Network.Wai.Handler.Warp`:

```haskell
run :: Port -> Application -> IO ()
```

Starts a webserver and runs the given application on the given port.

A trivial webserver

We can build an application that ignores the request and always answers with a `200` response and the text `"Hello"`:

```
alwaysHello :: Application
alwaysHello req respond =
  respond (responseLBS status200 [] "Hello\n")
```

```
test :: IO ()
test =
  run 8888 alwaysHello
```

Testing

```
$ curl -X GET "http://localhost:8888/foo/bar"
Hello
$ curl -X DELETE "http://localhost:8888/foo/baz"
Hello
```

Routing an dispatching

What a “real” application has to do:

- parse the incoming requests,
- depending on the path, the request headers, and the request body, make decisions on how to handle the request,
- dispatch the request to an appropriate handler,
- take the response from the handler and forward it to the request sender, or produce an error response instead.

Routing an dispatching

What a “real” application has to do:

- parse the incoming requests,
- depending on the path, the request headers, and the request body, make decisions on how to handle the request,
- dispatch the request to an appropriate handler,
- take the response from the handler and forward it to the request sender, or produce an error response instead.

These tasks can be automated using `servant` once the API is known.

Computing an application

The interface

```haskell
serve ::
  HasServer api '[]
   => Proxy api -> Server api -> Application
```

The interface

```
serve ::
  HasServer api '[]
   => Proxy api -> Server api -> Application
```

- `api` is an API type such as we have seen,

The interface

```haskell
serve ::
  HasServer api '[]
   => Proxy api -> Server api -> Application
```

- `api` is an API type such as we have seen,
- `HasServer` is a type class defined by `servant`,

The interface

```
serve ::
  HasServer api '[]
   => Proxy api -> Server api -> Application
```

- `api` is an API type such as we have seen,
- `HasServer` is a type class defined by `servant`,
- `Server` is a type family defined by `servant` (non-injective, therefore the `Proxy`).

`Server` and `HasServer`

Actually, `Server` is a wrapper around an associated type:

```
class HasServer api (context :: [*]) where
  type SeverT api (m :: * -> *) :: *

  route ::
    Proxy api
     -> Context context
     -> Delayed env (Server api)
     -> Router env
```

```
type Server api = SeverT api Handler

newtype Handler a = Handler
  {runHandler' :: ExceptT ServantErr IO a}
```

Simplify

We simplify a bit in order to better keep track of the actual ideas:

- We move `SeverT` out of the `HasServer` class.
- We ignore “contexts” for now.
- We ignore `Delayed` and `Router` and adopt a simpler model instead.

Simplified scenario

```
type family ServerT api (m :: * -> *) :: *

class HasServer api where
  route ::
    Proxy api
     -> Server api
     -> Request
     -> Handler Response
```

Simplified scenario

```haskell
type family ServerT api (m :: * -> *) :: *

class HasServer api where
  route ::
    Proxy api
     -> Server api
     -> Request
     -> Handler Response
```

Intuitively, `ServerT` tells us the types of the handlers we need for a given API, and `route` turns the given handlers into a request-response system.

Implementing `ServerT` – verbs

```
type instance ServerT
  (Verb method status ctypes a) m =
  m a
```

Recall that `m` is instantiated to `Handler`, and `Handler a` is `ExceptT ServantErr IO a`.

So at an endpoint, we simply provide an `IO` action with the capability to throw errors and the obligation to return a value of result type `a`.

Implementing `ServerT` – paths

```haskell
type instance ServerT
  ((path :: Symbol) :> api) m =
  ServerT api m
```

A path does not affect the type of handler at all.

Implementing `ServerT` – captures

```
type instance ServerT
  (Capture capture a :> api) m =
  a -> ServerT api m
```

A capture of type `a` is provided to the handler as an *additional input* of type `a`!

Implementing `ServerT` – request body

```
type instance ServerT
  (ReqBody ctypes a :> api) m =
  a -> ServerT api m
```

The same happens for the request body which can also be accessed in the handler.

Implementing `ServerT` – choice

```
type instance ServerT
  (api1 :<|> api2) m =
  ServerT api1 m :<|> ServerT api2 m
```

```
data a :<|> b = a :<|> b
infixr 8 :<|>
```

A handler for two combined APIs is the combination (i.e., a pair) of the handlers for the two individual APIs.

We use a Servant-specific pair constructor (just for optical reasons).

Applying the type family

We can now (e.g. in GHCi) compute `Server KeyVal` and obtain:

```
  Server KeyVal
~ (     Handler [Entry]                  -- get all
  :<|> (Text -> Handler Entry)           -- post
  :<|> (Int -> Handler Entry)            -- get single
  :<|> (Int -> Handler NoContent)        -- delete
  )
```

Implementing the handlers

Maintaining server state

We keep the server state in an STM `TVar`:

```haskell
type Store = TVar (Map Int Text)
```

Maintaining server state

We keep the server state in an STM `TVar`:

```haskell
type Store = TVar (Map Int Text)
```

This gives us thread safety – because when a `wai` application is run, every request will be handled in its own Haskell thread, and be concurrent with possibly many others.

Maintaining server state

We keep the server state in an STM `TVar`:

```
type Store = TVar (Map Int Text)
```

This gives us thread safety – because when a `wai` application is run, every request will be handled in its own Haskell thread, and be concurrent with possibly many others.

In a proper system we would probably use a database to make the state persistent.

Getting all entries

```haskell
getAllEntries :: Store -> Handler [Entry]
getAllEntries store = do
  entries <- liftIO $ readTVarIO store
  return $ map (uncurry Entry) (toList entries)
```

Posting a new entry

```haskell
postEntry :: Store -> Text -> Handler Entry
postEntry store txt =
  liftIO $ atomically $ do
    entries <- readTVar store
    let key = if M.null entries then 1
                 else fst (findMax entries) + 1
    writeTVar store (insert key txt entries)
    return (Entry key txt)
```

Getting a single entry

```haskell
getEntry :: Store -> Int -> Handler Entry
getEntry store n = do
  entries <- liftIO $ readTVarIO store
  case lookup n entries of
    Just txt -> return (Entry n txt)
    Nothing  -> throwError err404
```

Deleting an entry

```haskell
deleteEntry :: Store -> Int -> Handler NoContent
deleteEntry store n = do
  liftIO $ atomically $
    modifyTVar store (delete n)
  return NoContent
```

Everything together

```
keyVal :: Store -> Server KeyVal
keyVal store =
      getAllEntries store
  :<|> postEntry store
  :<|> getEntry store
  :<|> deleteEntry store
```

This is (nearly) all we have to do in order to be able to run the server.

Completing the implementation

The `HasServer` class

Let's discuss how we can implement our (simplified) `HasServer` class:

```
class HasServer api where
  route ::
    Proxy api
     -> Server api
     -> Request
     -> Handler Response
```

Implementing `HasServer` – verbs

```
data Verb
  method (status :: Nat) (contentTypes :: [*]) a
```

Implementing `HasServer` – verbs

```
data Verb
  method (status :: Nat) (contentTypes :: [*]) a
```

The `method` indicates which requests match. We need a way to turn the `method` type into a value.

Implementing `HasServer` – verbs

```
data Verb
  method (status :: Nat) (contentTypes :: [*]) a
```

The `method` indicates which requests match. We need a way to turn the `method` type into a value.

The `status` indicates which response code to return. We need a way to turn the `status` type into a value.

Implementing `HasServer` – verbs

```haskell
data Verb
  method (status :: Nat) (contentTypes :: [*]) a
```

The `method` indicates which requests match. We need a way to turn the `method` type into a value.

The `status` indicates which response code to return. We need a way to turn the `status` type into a value.

The `contentTypes` indicate which formats we can convert our output to. We need some way to translate `a` into these formats.

Reflecting methods

```
class ReflectMethod a where
  reflectMethod :: Proxy a -> Method
```

Reflecting methods

```haskell
class ReflectMethod a where
  reflectMethod :: Proxy a -> Method
```

```haskell
instance ReflectMethod GET where
  reflectMethod _ = methodGet
instance ReflectMethod POST where
  reflectMethod _ = methodPost
instance ReflectMethod DELETE where
  reflectMethod _ = methodDelete
```

Reflecting numbers (and symbols)

This is provided by GHC itself:

```
natVal    :: KnownNat n => Proxy n -> Integer
symbolVal :: KnownSymbol n => Proxy n -> String
```

The constraints `KnownNat` and `KnownSymbol` are satisfied for all literal numbers and strings.

Reflecting numbers (and symbols)

This is provided by GHC itself:

```
natVal    :: KnownNat n => Proxy n -> Integer
symbolVal :: KnownSymbol n => Proxy n -> String
```

The constraints `KnownNat` and `KnownSymbol` are satisfied for all literal numbers and strings.

```
GHCi> natVal (Proxy @ 42)
42
GHCi> symbolVal (Proxy @ "foo")
"foo"
```

Content types

We again present a slightly simplified view:

```
class AllCTRender (ctypes :: [*]) a where
  handleAcceptH ::
    Proxy ctypes -> AcceptHeader
     -> Maybe (ContentType, a -> ByteString)
```

Content types

We again present a slightly simplified view:

```haskell
class AllCTRender (ctypes :: [*]) a where
  handleAcceptH ::
    Proxy ctypes -> AcceptHeader
     -> Maybe (ContentType, a -> ByteString)
```

The `handleAcceptH` method takes an accept header (from a request) and a value and decides which content type and representation to use (or fails).

Base case

```
instance AllCTRender '[] a where
  handleAcceptH _ _ = Nothing
```

If there are no accepted content types, then we can never succeed.

Case for `JSON`

```haskell
instance (ToJSON a, AllCTRender cs a)
   => AllCTRender (JSON ': cs) a where
  handleAcceptH _ (AcceptHeader accept) =
    let
      ct = "application/json"
    in
      (ct, encode) <$> matchAccept [ct, "*/*"] accept
       <|> handleAcceptH (Proxy @cs)
             (AcceptHeader accept)
```

We check if `"application/json"` is among the accepted content types, and either perform the rendering (requiring a `ToJSON a` instance) or fall back on other content types.

Implementing `HasServer` – verbs

- Check that the request method matches.
- Check that the request path is empty.
- Check that the accept header is compatible.
- If all this applies, run the handler and encode the result according to the selected content type.
- In any other case, fail.

Implementing HasServer – verbs

```haskell
instance
  ( AllCTRender ctypes a
  , ReflectMethod method, KnownNat status)
   => HasServer (Verb method status ctypes a) where
  route _ h req
     | requestMethod req == reflectMethod (Proxy @ method)
       && null (pathInfo req) =
        case handleAcceptH (Proxy @ ctypes)
          (acceptHeader req) of
          Just (ct, enc) -> do
            a <- h
            return $ responseLBS
              (Status
                (fromIntegral (natVal (Proxy @ status))) "")
              [("Content-Type", ct)]
              (enc a)
          Nothing -> throwError err404  -- drastic simplification
     | otherwise = throwError err404  -- drastic simplification
```

Implementing `HasServer` – paths

```haskell
instance
  (KnownSymbol path, HasServer api)
   => HasServer (path :> api) where
  route _ h req =
    case pathInfo req of
      (x : xs)
         | x == fromString (symbolVal (Proxy  @ path)) ->
          route (Proxy  @ api) h (req {pathInfo = xs})
      _ -> throwError err404
```

Comparatively simple – we just check if the path prefix matches.

Implementing `HasServer` – captures

```haskell
data Capture (sym :: Symbol) a
```

For captures, we have to be able to parse a path component according to the type of the capture `a`.

The `Symbol` is not being used here – it is contained in the API purely for documentation purposes.

Parsing captures

Once again, we take a simplified view:

```
class FromHttpApiData a where
  parseUrlPiece :: Text -> Maybe a
```

```
instance FromHttpApiData Int where
  parseUrlPiece = readMaybe . unpack
```

Implementing `HasServer` – captures

```
instance
  (FromHttpApiData a, HasServer api)
   => HasServer (Capture capture a :> api) where
  route _ h req =
    case pathInfo req of
      (x : xs) ->
        case parseUrlPiece x of
          Just a ->
            route (Proxy @ api) (h a) (req {pathInfo = xs})
          Nothing -> throwError err404
      _ -> throwError err404
```

Note how we pass the parsed path fragment to the handler when continuing.

Implementing `HasServer` – request body

For the request body, we have to do content type negotiation once more – but in the other direction:

The content type of the body must match one of the content types we are able to parse.

Content types

```haskell
class AllCTUnrender (ctypes :: [*]) a where
  canHandleCTypeH ::
    Proxy ctypes
     -> ContentType
     -> Maybe (ByteString -> Maybe a)
```

The outer `Maybe` signals whether we can handle the given content type – the inner `Maybe` signals parse failure.

Base case

```haskell
instance AllCTUnrender '[] a where
  canHandleCTypeH _ _ = Nothing
```

Without any accepted content types we unsurprisingly fail.

Case for JSON

```
instance (FromJSON a, AllCTUnrender cs a)
   => AllCTUnrender (JSON ': cs) a where
  canHandleCTypeH _ ct
     | ct == "application/json" = Just decode
     | otherwise                =
      canHandleCTypeH (Proxy @cs) ct
```

Implementing `HasServer` – request body

```
instance
  (AllCTUnrender ctypes a, HasServer api)
   => HasServer (ReqBody ctypes a :> api) where
  route _ h req =
    case canHandleCTypeH (Proxy @ctypes)
      (contentTypeHeader req) of
      Just dec -> do
        b <- liftIO $ lazyRequestBody req
        case dec b of
          Just a  -> route (Proxy @api) (h a) req
          Nothing -> throwError err404
      Nothing -> throwError err404
```

Implementing `HasServer` – choice

The final case we have to implement is choice:

- In our simple setting, we test if the first alternative fails with 404, and only if it does, try the second one.
- Full `servant` has a much more sophisticated notion of error codes and priorities, to make sure that the "correct" response code is sent in most situations.

Implementing `HasServer` – choice

```haskell
instance
  (HasServer api1, HasServer api2)
   => HasServer (api1 :<|> api2) where
  route _ (h1 :<|> h2) req =
    catchError
      (route (Proxy @ api1) h1 req)
      (\ err ->
        if errHTTPCode err == 404
          then route (Proxy @ api2) h2 req
          else throwError err)
```

What is left to do?

We still need to implement `serve` in terms of `route`:

```
serve ::
  HasServer api
   => Proxy api -> Server api -> Application
serve p h req respond = do
  result <-
    runExceptT (runHandler' (route p h req))
  case result of
    Left err  -> respond (responseServantErr err)
    Right rsp -> respond rsp
```

`Entry` and JSON

We also still need to provide `FromJSON` and `ToJSON` instances for `Entry` – a trivial job thanks to generic programming:

```
deriving instance Generic Entry

instance FromJSON Entry
instance ToJSON Entry
```

Putting everything together

```haskell
main :: IO ()
main = do
  store <- newTVarIO empty
  run 8000
    (serve (Proxy @KeyVal) (keyVal store))
```

This allocates a `TVar` to hold the store, then computes the server from the API and the handlers we defined before (`keyVal`), and finally runs it.

Testing

```
$ curl -X GET "http://localhost:8000/entry"
[]
$ curl -H "Content-Type: application/json" -d"\"foo\""
  -X POST "http://localhost:8000/entry"
{"entryId":1,"entryText":"foo"}
$ curl -H "Content-Type: application/json" -d"\"bar\""
  -X POST "http://localhost:8000/entry"
{"entryId":2,"entryText":"bar"}
$ curl -X GET "http://localhost:8000/entry"
[{"entryId":1,"entryText":"foo"},{"entryId":2,"entryText":"bar"}]
$ curl -X GET "http://localhost:8000/entry/1"
{"entryId":1,"entryText":"foo"}
$ curl -X DELETE "http://localhost:8000/entry/1"
$ curl -X GET "http://localhost:8000/entry"
[{"entryId":2,"entryText":"bar"}]
```

Recap

We've spent a lot of time on basically re-implementing `servant`, so let's briefly look at what we actually had to do.

We had to:

- Define the API.
- Define the domain datatypes and instances (`Entry`, `FromJSON`, `ToJSON`).
- Define the handlers.
- Compose everything (`main`).

API

```
type KeyVal =
       "entry" :> Get '[JSON] [Entry]
   :<|> "entry"
          :> ReqBody '[JSON] Text
          :> PostCreated '[JSON] Entry
   :<|> "entry" :> Capture "n" Int
          :> Get '[JSON] Entry
   :<|> "entry" :> Capture "n" Int
          :> DeleteNoContent '[JSON] NoContent
```

Domain datatypes and instances

```haskell
data Entry =
  Entry
    { entryId   :: Int
    , entryText :: Text
    }
  deriving (Generic, Show)

instance FromJSON Entry
instance ToJSON Entry
```

Handlers

```haskell
getAllEntries :: Store -> Handler [Entry]
getAllEntries store = do
  entries <- liftIO $ readTVarIO store
  return $ map (uncurry Entry) (toList entries)

postEntry :: Store -> Text -> Handler Entry
postEntry store txt =
  liftIO $ atomically $ do
    entries <- readTVar store
    let key = if M.null entries then 1
              else fst (findMax entries) + 1
    writeTVar store (insert key txt entries)
    return (Entry key txt)

getEntry :: Store -> Int -> Handler Entry
getEntry store n = do
  entries <- liftIO $ readTVarIO store
  case lookup n entries of
    Just txt -> return (Entry n txt)
    Nothing  -> throwError err404

deleteEntry :: Store -> Int -> Handler NoContent
deleteEntry store n = do
  liftIO $ atomically $
    modifyTVar store (delete n)
  return NoContent
```

Combined handler

```
keyVal :: Store -> Server KeyVal
keyVal store =
        getAllEntries store
   :<|> postEntry store
   :<|> getEntry store
   :<|> deleteEntry store
```

Composing everything

```haskell
main :: IO ()
main = do
  store <- newTVarIO empty
  run 8000
    (serve (Proxy @ KeyVal) (keyVal store))
```

What do we get?

A lot of code “for free”:

- All the routing, parsing, serialization, dispatching is taken care of automatically, via the API specification.
- The handlers are just ordinary functions that do not need to know much about running in a web setting.

Type safety:

- The handlers work with dedicated Haskell types, all the conversions to less typed settings (JSON, strings) happen behind the scenes.
- If we change the API type, or use an undeclared input, or produce the wrong output, this is all a type error.

Other “interpretations”

Servant is a domain-specific language

What `servant` really provides is a type-level domain-specific language for describing APIs:

- The API is the “program”.
- The server is an “interpretation” of the program.

Servant is a domain-specific language

What `servant` really provides is a type-level domain-specific language for describing APIs:

- The API is the "program".
- The server is an "interpretation" of the program.

But `servant` offers more interpretations besides the server interpretation.

Deriving a client

Computing client functions

```
client ::
  HasClient api
   => Proxy api -> Client api
```

Very similar – `HasClient` is a class, and `Client` is an associted type.

Instantiating `Client`

```
  Client KeyVal
~ (     ClientM [Entry]                    -- get all
   :<|> (Text -> ClientM Entry)            -- post
   :<|> (Int -> ClientM Entry)             -- get single
   :<|> (Int -> ClientM NoContent)         -- delete
  )
```

Nearly the same as `Server`, only that this time, we *get* these functions, rather than having to provide them.

Obtaining client functions

```haskell
getAll    :: ClientM [Entry]
postNew   :: Text -> ClientM Entry
getSingle :: Int -> ClientM Entry
delSingle :: Int -> ClientM NoContent

getAll :<|> postNew :<|> getSingle :<|> delSingle
  = client (Proxy @KeyVal)
```

Running client functions

```haskell
runClientM ::
  ClientM a -> ClientEnv
   -> IO (Either ServantError a)
```

Running client functions

```haskell
runClientM ::
  ClientM a -> ClientEnv
   -> IO (Either ServantError a)
```

```haskell
data ClientEnv =
  ClientEnv
    { manager :: Manager
    , baseUrl :: BaseUrl
    }
```

Testing in GHCi

Of course, this requires we are running the server.

We can create a manager using the `http-client` package.

```
GHCi> m <- newManager defaultManagerSettings
GHCi> base = BaseUrl Http "localhost" 8000 ""
GHCi> env = ClientEnv m base
GHCi> runClientM (postNew "foo") env
Right (Entry {entryId = 1, entryText = "foo"})
GHCi> runClientM getAll env
Right [Entry {entryId = 1, entryText = "foo"}]
```

Servant for clients

If you derive both a server and a client from a common API, you can be certain that they match.

Servant for clients

If you derive both a server and a client from a common API, you can be certain that they match.

On the other hand, it can also be useful to employ `servant` to just derive a client for a web API you want to bind to, in order to quickly get the right functions in a strongly typed way.

Yet more interpretations

There's more:

- Generate clients in various other / external languages (e.g. Javascript).
- Generate documentation in various formats (e.g. Swagger).
- Generate type-safe links within an API.
- Generate mock servers and mock clients.
- …

Extensibility

Servant is carefully designed to be extensible in two dimensions:

- You can add new interpretations.
- You can add new combinators to the API language.

Both kinds of extension usually do not require access or modifications of the core library.