!!YPER

Type-safe, statically checked composition of HTTP servers

v0.5.0

March 6, 2017

Contents

Type-safe, statically checked composition of HTTP servers

Hyper is an experimental middleware architecture for HTTP servers written in PureScript. Its main focus is correctness and type-safety, using type-level information to enforce correct composition and abstraction for web servers. The Hyper project is also a breeding ground for higher-level web server constructs, which tend to fall under the "framework" category.

Note: Until recently, most work on extensions and higher-level constructs have started their life in the main repository, but have then been separated into their own packages. More things might be separated in the near future, so prepare for unstability. Hyper itself should be considered experimental, for now.

This documentation is divided into sections, each being suited for different types of information you might be looking for.

- *Introduction* (page 3) describes the project itself; its motivations, goals, and relevant information for contributors to the project.
- *Core API* (page 5) is an introduction and reference of the Core API of Hyper, on which the higher-level features build.
- *Topic Guides* (page 8) explain how to solve specifics tasks of writing web servers using features of Hyper.

- *Tutorials* (page 12) are step-by-step guides on how to build complete web applications.
- *Extensions* (page 13) provides an overview of extensions to Hyper.

Chapter 1

Introduction

Composing middleware in NodeJS is a risky business. They mutate the HTTP request and response objects freely, and are often dependent on each others' side-effects. There are no guarantees that you have stacked the middleware functions in a sensible order, and it is often the case, in the author's experience, that misconfigured middleware takes a lot of time and effort to debug.

1.1 Goals

The goal of *Hyper* is to make use of row polymorphism and other tasty type system features in PureScript to enforce correctly stacked middleware in HTTP server applications. All effects of middleware should be reflected in the types to ensure that common mistakes cannot be made. A few examples of such mistakes could be:

- Incorrect ordering header and body writing
- Writing incomplete responses
- Writing multiple responses
- Trying to consume a non-parsed request body
- Consuming a request body parsed as the wrong type
- Incorrect ordering of, or missing, error handling middleware
- Incorrect ordering of middleware for sessions, authentication, authorization
- Missing authentication and/or authorization checks
- Linking, in an HTML anchor, to a resource that is not routed
- Posting, in an HTML form, to a resource that is not routed

Hyper aims to solve these issues, in part through its Core API for middleware, but also through a number of extensions for building safely composable and maintainable web applications.

1.2 Contributing

While Hyper is currently an experiment, and in constant flux, you are welcome to contribute. Please post ideas and start discussions using the issue tracker on GitHub[1]. You can also contact Oskar Wickström[2] directly for design discussions. If this project grows, we can setup a mailing list, or other some other means of communication.

[1] https://github.com/owickstrom/hyper/issues
[2] https://wickstrom.tech/about.html

Please note that sending pull requests without first discussing the design is probably a waste of time, if not only fixing simple things like typos.

Chapter 2

Core API

This chapter explains the central components of Hyper, called the *Core API*. While focusing heavily on safety, Hyper tries to provide an open API that can support multiple PureScript backends, and different styles of web applications.

The design of Hyper is inspired by a number of projects. The middleware chain lends much from *Plug*, an abstract HTTP interface in Elixir, that enables various HTTP libraries to inter-operate. You might also find similarities with *connect* in NodeJS. On the type system side, Hyper tries to bring in ideas from *Haskell* and *Idris*, specifically the use of phantom types and GADTs to lift invariants to the type level and increase safety.

The central components of the Core API are:

2.1 Conn

A *Conn*, short for "connection", models the entirety of a connection between the HTTP server and the user agent, both request and response.

```
type Conn req res components =
  { request :: req
  , response :: res
  , components :: components
  }
```

The `request` and `response` hold the values representing the HTTP request and response, respectively. The purpose of the `components` field, however, is not that obvious. It is used for things not directly related to the HTTP, but nonetheless related to the act of responding to the HTTP request. A middleware can add information the Conn using components, like providing authentication or authorization values. The types of these components then becomes part of the Conn type, and you get compile-time guarantees when using the provided components.

2.2 Middleware

A *middleware* is an *indexed monadic action* transforming one `Conn` to another `Conn`. It operates in some base monad `m`, and is indexed by `i` and `o`, the *input* and *output* Conn types of the middleware action.

```
newtype Middleware m i o a = ...
```

The input and output type parameters are used to ensure that a Conn is transformed, and that side-effects are performed, correctly, throughout the middleware chain.

Being able to parameterize `Middleware` with some type `m`, you can customize the chain depending on the needs of your middleware and handlers. Applications can use monad transformers to track state, provide configuration, gather metrics, and much more, in the chain of middleware.

Middleware are composed using `ibind`, the indexed monadic version of `bind`. The simplest way of composing middleware is by chaining them with `:*>`, from `Control.IxMonad`. See haskell-indexed-monad[3] for more information.

```
writeStatus statusOK
:*> closeHeaders
:*> respond "We're composing middleware!"
```

If you want to feed the return value of one middleware into another, use `:>>=`, the infix operator alias for `mbind`.

```
getUser :>>= renderUser
```

You can also rebind the *do block* syntax to use `ibind` instead of regular `bind`.

```
do
  user <- getUser
  writeStatus statusOK
  closeHeaders
  respond ("User: " <> user.name)
  where bind = ibind
```

2.3 Response State Transitions

The `writer` field in the `response` record of a Conn is a value provided by the server backend. Middleware often constrain the `writer` field to be a value implementing the `ResponseWriter` type class. This makes it possible to provide response writing abstractions without depending on a specific server backend.

The state of a response writer is tracked in its type parameter. This state tracking, and the type-indexed middleware using the response writer, guarantee correctness in response handling, preventing incorrect ordering of headers and body writes, incomplete responses, or other such mistakes. Let us have a look at the type signatures of some of response writing functions in `Hyper.Response`.

We see that `headers` takes a traversable collection of headers, and gives back a middleware that, given a connection where headers are ready to be written (`HeadersOpen`), writes all specified headers, writes the separating CRLF before the HTTP body, and marks the state of the response writer as being ready to write the body (`BodyOpen`).

```
headers :: forall t m req res rw c.
           (Traversable t, Monad m, ResponseWriter rw m b) =>
           t Header
        -> Middleware
           m
           (Conn req { writer :: rw HeadersOpen | res } c)
           (Conn req { writer :: rw BodyOpen | res } c)
           Unit
```

To be used in combination with `headers`, the `respond` function takes some `Response m r b`, and gives back a middleware that, given a connection *where all headers have been written*, writes a response, and *marks the state of the response writer as ended.*

```
respond :: forall m r b req res rw c.
           (Monad m, Response m r b, ResponseWriter rw m b) =>
           r
        -> Middleware
           m
```

[3] https://pursuit.haskell.org/packages/haskell-indexed-monad/0.1.1

```
        (Conn req { writer :: rw BodyOpen | res } c)
        (Conn req { writer :: rw ResponseEnded | res } c)
        Unit
```

The `Response` type class describes types that can be written as responses. It takes three type parameters, where `b` is the target type, `m` is a base monad for the Middleware returned, and `r` is the original response type,

```
class Response b m r where
  toResponse :: forall i. r -> Middleware m i i b
```

This mechanism allows servers to provide specific types for the response body, along with instances for common response types. When using the Node server, which has a response body type wrapping `Buffer`, you can still respond with a `String` or `HTML` value directly.

Aside from convenience in having a single function for most response types and servers, the polymorphism of `respond` lets middleware be decoupled from specific servers. It only requires an instance matching the response type used by the middleware and the type required by the server.

2.4 Servers

Although Hyper middleware can applied directly to Conn values using `runMiddleware`, you likely want a *server* to run your middleware. Hyper tries to be as open as possible when it comes to servers – your application, and the middleware it depends on, should not be tied to a specific server. This allows for greater reuse and the ability to test entire applications without running the “real” server. Currently Hyper bundles a NodeJS server, described in *NodeJS* (page 10), as well as a test server, described in *Testing* (page 10).

Chapter 3

Topic Guides

The topic guides explain how to solve specifics tasks of writing web servers using features of Hyper. They are not full-length tutorials, but try to cover the details, and possible considerations, of a single subject. The following topic guides are available:

3.1 Request Body Reading

The `RequestBodyReader` type class has one operation, `readBody`, which supports different servers to provide different types of request body values.

```
class RequestBodyReader r m b | r -> b where
  readBody
    :: forall req res c.
       Middleware
       m
       (Conn { body :: r | req } res c)
       (Conn { body :: r | req } res c)
       b
```

Given that there is an instance for the body `b`, and the return type `r`, we can use this middleware together with other middleware, like so:

```
onPost =
  readBody :>>=
  case _ of
    "" ->
      writeStatus statusBadRequest
      :*> closeHeaders
      :*> respond "... anyone there?"
    msg ->
      writeStatus statusBadRequest
      :*> closeHeaders
      :*> respond ("You said: " <> msg)
```

3.2 Forms

When working with form data, we often want to serialize and deserialize forms as custom data types, instead of working with the key-value pairs directly. The `ToForm` and `FromForm` type classes abstracts serialization and deserialization to form data, respectively.

We first declare our data types, and some instance which we will need later.

```
data MealType = Vegan | Vegetarian | Omnivore | Carnivore

derive instance genericMealType :: Generic MealType _
instance eqMealType :: Eq MealType where eq = genericEq
instance showMealType :: Show MealType where show = genericShow

newtype Order = Order { beers :: Int, meal :: MealType }
```

In this example we will only deserialize forms, and thus we only need the `FromForm` instance.

```
instance fromFormOrder :: FromForm Order where
  fromForm form = do
    beers <- required "beers" form >>= parseBeers
    meal <- required "meal" form >>= parseMealType
    pure (Order { beers: beers, meal: meal })
    where
      parseBeers s =
        maybe
        (throwError ("Invalid number: " <> s))
        pure
        (Int.fromString s)

      parseMealType =
        case _ of
          "Vegan" -> pure Vegan
          "Vegetarian" -> pure Vegetarian
          "Omnivore" -> pure Omnivore
          "Carnivore" -> pure Carnivore
          s -> throwError ("Invalid meal type: " <> s)
```

Now we are ready to write our handler. We use `parseFromForm` to get a value of type `Either String Order`, where the `String` explains parsing errors. By pattern matching using record field puns, we extract the `beers` and `meal` values, and respond based on those values.

```
onPost =
  parseFromForm :>>=
  case _ of
    Left err ->
      writeStatus statusBadRequest
      :*> closeHeaders
      :*> respond (err <> "\n")
    Right (Order { beers, meal })
      | meal == Omnivore || meal == Carnivore ->
        writeStatus statusBadRequest
        :*> closeHeaders
        :*> respond "Sorry, we do not serve meat here.\n"
      | otherwise ->
        writeStatus statusBadRequest
        :*> closeHeaders
        :*> respond ("One " <> show meal <> " meal and "
                     <> show beers <> " beers coming up!\n")
```

Let's try this server out at the command line.

```
$ curl -X POST -d 'beers=6' http://localhost:3000
Missing field: meal
$ curl -X POST -d 'meal=Vegan&beers=foo' http://localhost:3000
Invalid number: foo
$ curl -X POST -d 'meal=Omnivore&beers=6' http://localhost:3000
Sorry, we do not serve meat here.
$ curl -X POST -d 'meal=Vegetarian&beers=6' http://localhost:3000
One Vegetarian meal and 6 beers coming up!
```

3.3 NodeJS

The server in `Hyper.Node.Server` wraps the `http` module in NodeJS, and serves middleware using the `Aff` monad. Here is how you can start a Node server:

```
let
  app =
    writeStatus (Tuple 200 "OK")
    :*> closeHeaders
    :*> respond "Hello there!"
in runServer defaultOptions {} app
```

As seen above, `runServer` takes a record of options, an initial *components* record, and your application middleware. If you want to do logging on server startup, and on any request handling errors, use `defaultOptionsWithLogging`.

3.3.1 Monad Transformers

You might want to use a monad transformer stack in your application, for instance as a way to pass configuration, or to accumulate some state in the chain of middleware. The underlying monad of `Middleware` is parameterized for this exact purpose. When running the NodeJS server with monad transformers, you need to use `runServer'` instead of the regular `runServer`, and pass a function that runs your monad and returns an `Aff` value.

The following code runs a middleware using the `ReaderT` monad transformer. Note that the `runAppM` function might need to be defined at the top-level to please the type checker.

```
type MyConfig = { thingToSay :: String }

runAppM
  :: forall e a.
    ReaderT MyConfig (Aff e) a
  -> Aff e a
runAppM = flip runReaderT { thingToSay: "Hello, ReaderT!" }

main :: forall e. Eff (console :: CONSOLE, http :: HTTP | e) Unit
main =
  let app = do
        config <- lift' ask
        writeStatus statusOK
        closeHeaders
        respond config.thingToSay
        where bind = ibind
  in runServer' defaultOptionsWithLogging {} runAppM app
```

In a real-world application the configuration type `MyConfig` could hold a database connection pool, or settings read from the environment, for example.

3.4 Testing

When running tests you might not want to start a full HTTP server and send requests using an HTTP client. Instead you can use the server in `Hyper.Test.TestServer`. It runs your middleware directly on `Conn` values, and collects the response using a Writer monad. You get back a `TestResponse` from which you can extract the status code, headers, and the response body.

```
it "responds with a friendly message" do
  conn <- { request: {}
          , response: { writer: testResponseWriter }
          , components: {}
```

```
        }
        # evalMiddleware app
        # testServer
testStatus conn `shouldEqual` Just statusOK
testStringBody conn `shouldEqual` "Hello there!"
```

Chapter 4

Tutorials

No tutorials are available here yet, but you can check out the runnable examples at GitHub[4].

[4] https://github.com/owickstrom/hyper/tree/master/examples

Chapter 5

Extensions

This chapter shows extensions built on top of the Core API, providing higher-level abstractions. They are, for technical reasons, usually hosted as separate Git repositories.

5.1 Type-Level Routing

The purescript-hyper-routing[5] package provides a concept called *routing types*. Its API, inspired heavily by the Haskell library Servant[6], lets us express web application routing at the type-level.

By using routing types we get static guarantees about having handled all cases and having correctly serializing and deserializing data. We also get a lot of stuff for free, such as type-safe parameters for handlers, generated type-safe URIs to endpoints, and generated clients and servers.

Based on the routing types, a couple of packages provide the machinery for creating servers and clients using routing types:

5.1.1 Servers for Routing Types

The primary use of routing types in Hyper is for writing web servers. The purescript-hyper-routing-server[7] package provides a router middleware which, together with our handler and rendering functions, gives us a full-fledged server.

A Single-Endpoint Example

Let's say we want to render a home page as HTML. We start out by declaring the endpoint data type `Home`, and the structure of our site:

```
data Home = Home

type Site1 = Get HTML Home
```

`Get HTML Home` is a routing type with only one endpoint, rendering a `Home` value as HTML. So where does the `Home` value come from? We provide it using a *handler*. A handler for `Site1` would be some value of the following type:

```
forall m. Monad m => ExceptT RoutingError m Home
```

We can construct such a value using `pure` and a `Home` value:

[5] https://github.com/owickstrom/purescript-hyper-routing
[6] https://haskell-servant.github.io
[7] https://github.com/owickstrom/purescript-hyper-routing-server

```
home :: forall m. Applicative m => m Home
home = pure Home
```

Nice! But what comes out on the other end? We need something that renders the `Home` value as HTML. By providing an instance of `EncodeHTML` for `Home`, we instruct the endpoint how to render.

```
instance encodeHTMLHome :: EncodeHTML Home where
  encodeHTML Home =
    p (text "Welcome to my site!")
```

The `HTML` type is a phantom type, only used as a marker type, and the actual markup is written in the `MarkupM` DSL from purescript-smolder[8].

We are getting ready to create the server. First, we need a value-level representation of the `Site1` type, to be able to pass it to the `router` function. For that we use Proxy[9]. Its documentation describes it as follows:

> The Proxy type and values are for situations where type information is required for an input to determine the type of an output, but where it is not possible or convenient to provide a value for the input.

We create a top-level definition of the type `Proxy Site1` with the value constructor `Proxy`.

```
site1 :: Proxy Site1
site1 = Proxy
```

We pass the proxy, our handler, and the `onRoutingError` function for cases where no route matched the request, to the `router` function.

```
onRoutingError status msg =
  writeStatus status
  :*> contentType textHTML
  :*> closeHeaders
  :*> respond (maybe "" id msg)

siteRouter = router site1 home onRoutingError
```

The value returned by `router` is regular middleware, ready to be passed to a server.

```
main :: forall e. Eff (http :: HTTP, console :: CONSOLE, buffer :: BUFFER | e) Unit
main =
  runServer defaultOptions {} siteRouter
  where
```

Routing Multiple Endpoints

Real-world servers often need more than one endpoint. Let's define a router for an application that shows a home page with links, a page listing users, and a page rendering a specific user.

```
data Home = Home

data AllUsers = AllUsers (Array User)

newtype User = User { id :: Int, name :: String }

type Site2 =
  Get HTML Home
  :<|> "users" :/ Get HTML AllUsers
  :<|> "users" :/ Capture "user-id" Int :> Get HTML User
```

8 https://github.com/bodil/purescript-smolder
9 https://pursuit.purescript.org/packages/purescript-proxy/1.0.0/docs/Type.Proxy

```
site2 :: Proxy Site2
site2 = Proxy
```

Let's go through the new constructs used:

- `:<|>` is a type operator that separates *alternatives*. A router for this type will try each route in order until one matches.
- `:/` separates a literal path segment and the rest of the endpoint type.
- `Capture` takes a descriptive string and a type. It takes the next available path segment and tries to convert it to the given type. Each capture in an endpoint type corresponds to an argument in the handler function.
- `:>` separates a an endpoint modifier, like `Capture`, and the rest of the endpoint type.

We define handlers for our routes as regular functions on the specified data types, returning `ExceptT RoutingError m a` values, where `m` is the monad of our middleware, and `a` is the type to render for the endpoint.

```
home :: forall m. Monad m => ExceptT RoutingError m Home
home = pure Home

allUsers :: forall m. Monad m => ExceptT RoutingError m AllUsers
allUsers = AllUsers <$> getUsers

getUser :: forall m. Monad m => Int -> ExceptT RoutingError m User
getUser id' =
  find userWithId <$> getUsers >>=
  case _ of
    Just user -> pure user
    Nothing ->
      throwError (HTTPError { status: statusNotFound
                            , message: Just "User not found."
                            })
  where
    userWithId (User u) = u.id == id'
```

As in the single-endpoint example, we want to render as HTML. Let's create instances for our data types. Notice how we can create links between routes in a type-safe manner.

```
instance encodeHTMLHome :: EncodeHTML Home where
  encodeHTML Home =
    case linksTo site2 of
      _ :<|> allUsers' :<|> _ ->
        p do
          text "Welcome to my site! Go check out my "
          linkTo allUsers' (text "Users")
          text "."

instance encodeHTMLAllUsers :: EncodeHTML AllUsers where
  encodeHTML (AllUsers users) =
    div do
      h1 (text "Users")
      ul (traverse_ linkToUser users)
    where
      linkToUser (User u) =
        case linksTo site2 of
          _ :<|> _ :<|> getUser' ->
            li (linkTo (getUser' u.id) (text u.name))

instance encodeHTMLUser :: EncodeHTML User where
```

```
  encodeHTML (User { name }) =
    h1 (text name)
```

The pattern match on the value returned by `linksTo` must match the structure of the routing type. We use `:<|>` to pattern match on links. Each matched link will have a type based on the corresponding endpoint. `getUser` in the previous code has type `Int -> URI`, while `allUsers` has no captures and thus has type `URI`.

We are still missing `getUsers`, our source of User values. In a real application it would probably be a database query, but for this example we simply hard-code some famous users of proper instruments.

```
getUsers :: forall m. Applicative m => m (Array User)
getUsers =
  pure
  [ User { id: 1, name: "John Paul Jones" }
  , User { id: 2, name: "Tal Wilkenfeld" }
  , User { id: 3, name: "John Patitucci" }
  , User { id: 4, name: "Jaco Pastorious" }
  ]
```

Almost done! We just need to create the router, and start a server.

```
main :: forall e. Eff (http :: HTTP, console :: CONSOLE, buffer :: BUFFER | e) Unit
main =
  let otherSiteRouter =
        router site2 (home :<|> allUsers :<|> getUser) onRoutingError

      onRoutingError status msg =
        writeStatus status
        :*> contentType textHTML
        :*> closeHeaders
        :*> respond (maybe "" id msg)

  in runServer defaultOptions {} otherSiteRouter
```

Notice how the composition of handler functions, using the value-level operator `:<|>`, matches the structure of our routing type. If we fail to match the type we get a compile error.

Content Negotiation

By specifying alternative content types for an endpoint, Hyper can choose a response and content type based on the request `Accept` header. This is called *content negotiation*. Instead of specifying a single type, like `HTML` or `JSON`, we provide alternatives using `:<|>`. All content types must have `MimeRender` instances for the response body type.

```
type Site3 =
  Get HTML Home
  :<|> "users" :/ Get (HTML :<|> JSON) AllUsers
  :<|> "users" :/ Capture "user-id" Int :> Get (HTML :<|> JSON) User
```

By making requests to this site, using `Accept` headers, we can see how the router chooses the matching content type (output formatted and shortened for readability).

```
$ <strong>curl -H 'Accept: application/json' http://localhost:3000/users</strong>
[
  {
    "name": "John Paul Jones",
    "id": "1"
  },
  {
    "name": "Tal Wilkenfeld",
```

```
    "id": "2"
  },
  ...
]
```

There is support for *wildcards* and *qualities* as well.

```
$ curl -H 'Accept: text/*;q=1.0' http://localhost:3000/users
<div>
  <h1>Users</h1>
  <ul>
    <li><a href="/users/1">John Paul Jones</a></li>
    <li><a href="/users/2">Tal Wilkenfeld</a></li>
    ...
  </ul>
</div>
```

5.1.2 Automatically Derived XHR Clients

As we represent routing as types, we can derive XHR clients from those types. In a client-side application, for example one written using Pux[10] or Halogen[11], we use the purescript-hyper-routing-xhr[12] library to derive such client functions. We get functions with type-safe parameters and encoding/decoding, using our routed types, and results in the Aff[13] monad, which are easily integrated into most frameworks.

[10] https://www.purescript-pux.org
[11] https://github.com/slamdata/purescript-halogen
[12] https://github.com/owickstrom/purescript-hyper-routing-xhr
[13] https://github.com/slamdata/purescript-aff