Hyper

Oskar Wickström

Goals

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 my experience, that misconfigured middleware takes a lot of time and effort to debug.

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 could be:

- Incorrect ordering header and body writing
- Writing incomplete responses
- Overwriting headers
- Writing multiple responses
- Trying to consume a non-parsed request body
- Consuming a request body parsed as the wrong type
- Incorrect ordering of 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

Can we use the PureScript type system to eradicate this class of errors? Let's find out!

Design

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

Conn

A *Conn*, short for "connection", models the entirety of a connection between the HTTP server and the user agent - both request and response. This design is adopted from *Plug*, an abstract HTTP interface in Elixir, that enables various HTTP libraries to inter-operate.

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

Middleware

A *middleware* is a function transforming a `Conn` to another `Conn`, in some monadic type `m`. The `Middleware` type synonym encapsulates this concept, but it is still a regular function.

```
type Middleware m c c' = c -> m c'
```

Being able to parameterize `Conn` 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.

Response State Transitions

The fields `state` and `writer` are usually present in the `response` record of a Conn, as they are required by the functions in `Hyper.Response`. The state of a response is tracked in type signatures to guarantee correctness in response handling, preventing incorrect ordering of headers and body writes, incomplete responses, or other such mistakes.

The `writer` is a value provided by the server of the application, constrained by the `Hyper.Core.ResponseWriter` type class, and is used by functions in `Hyper.Response` to provide higher-level ways of responding to request. Let us have a look at the type signatures of some of those functions.

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*, writes all specified headers, writes the separating CRLF before the HTTP body, and *marks the state of the response as headers being closed*.

```
headers :: forall t m req res rw c.
           (Traversable t, Monad m, ResponseWriter rw m) =>
           t Header
        -> Middleware
```

```
          m
         (Conn req { writer :: rw, state :: HeadersOpen | res } c)
        (Conn req { writer :: rw, state :: HeadersClosed | res } c)
```

To be used in combination with `headers`, the `respond` function takes some `Response r`, 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 as ended*.

```
respond :: forall r m req res rw c.
           (Monad m, Response r, ResponseWriter rw m) =>
           r
        -> Middleware
           m
         (Conn req { writer :: rw, state :: HeadersClosed | res } c)
         (Conn req { writer :: rw, state :: ResponseEnded | res } c)
```

The `Response` type class describes types that can be written as responses.

```
class Response r where
  toResponse :: r -> String
```

NOTE: The return type of `toResponse` should probably be something other than `String` (GitHub issue).

Use Cases

Here follows a collection of loosely organized thoughts on how to implement safe middleware in Hyper. Very much work-in-progress.

Parsing the Request Body

Warning! Rough edges here, see the GitHub issue for details.

The request body is initially a `Stream Read Initial` in the connection. It might not always be of interest, thus it is not read, and not parsed, by default. Instead, the user explicitly chooses to read and parse the body with a given parser, which returns a new connection of a type reflecting the action.

```
class BodyParser p t | p -> t where
  parse :: forall req res c h. p
        -> Middleware
           (Conn
            { bodyStream :: Stream Read Initial
            , headers :: { "content-type" :: String
                         , "content-length" :: String
                         | h
```

```
                  }
           | req
           }
           res
           c)
          (Conn
           { bodyStream :: Stream Read Closed
           , headers :: { "content-type" :: String
                        , "content-length" :: String
                        | h
                        }
           , body :: t
           | req
           }
          res
          c)
```

Given this type, the request body can neither be read more than once, nor can the connection's `body` be overwritten. An example parser is the `BodyParser` instance for `FormParser` and `Form`.

```
-- | A form represents a "www-form-urlencoded" form.
newtype Form = Form (Array (Tuple String String))

-- | Placeholder constructor without any options.
data FormParser = FormParser

instance bodyParserFormParser :: BodyParser FormParser Form where
  parse _ = parseBodyFromString splitPairs
    where
     toTuple :: Array String -> Either Error (Tuple String String)
     toTuple kv =
       case kv of
         [key, value] → Right (Tuple (decodeURIComponent key)
                                     (decodeURIComponent value))
        parts       → Left (error ("Invalid form key-value pair: "
                                   <> joinWith " " parts))
     splitPair = split (Pattern "=")
     splitPairs  String → Either Error Form
     splitPairs = (<$>) Form
                  <<< sequence
                  <<< map toTuple
                  <<< map splitPair
                  <<< split (Pattern "&")
```

This instance uses the helper `parseBodyFromString` to first read the body as a string, then parse that string as a `www-form-urlencoded` form. Any invalid

form will throw an error in the Aff monad, which can be caught and handled.

Cohesion of Links, Forms, and Routes

It should not be possible to link, using an HTML anchor, to a resource in the web application that does not exist, or that does not handle the GET method. Neither should it be possible to create a form that posts to a non-existing resource, or a resource not handling POST requests.

Resources

Hyper has a concept of *resources*. Each resource is a record describing its *path*, along with a set of HTTP methods and handlers. Each method implemented in Hyper must be specified explicitly in the record with a `ResourceMethod` value, and those values are parameterized with one of the marker types describing if it is routed - `Supported` or `NotSupported`. The helper functions `handler` and `notSupported` are used to construct `ResourceMethod` values.

```
index =
  { path: []
  , "GET": handler (html (h1 (text "Welcome!")))
  , "POST": notSupported
  }
```

Resource Routers

The `resource` function creates a `ResourceRouter` that tries to route HTTP requests to handlers in its resource. It should also add the application resources as a type in the components of the Conn, giving subsequent middleware access to that information. *The encoding of resource types in the Conn is NOT supported yet.*

```
app = fallbackTo notFound (resource index)
```

The `ResourceRouter` provides an instance for `Alt`, making it possible to chain resources and have them try to match the request in order.

```
app = fallbackTo notFound (resource index <|> resource about <|> resource contact)
```

HTML DSL

A separate DSL for writing HTML, providing functions that take resources as arguments, creates links and forms to resources in the application *only if they are in scope and support the required HTTP methods.* Paths are used from the

resource, so you cannot make a typo in the URL. In other words, mistakes in routing and references between resources give you compile-time errors.

```
about =
  { path: ["about"]
  , "GET": handler (\conn -> html
                             (linkTo contact (text "Contact Me!"))
                             conn)
  , "POST": notSupported
  }

contact =
  { path: ["contact"]
 , "GET": handler (html (text "Good luck finding my email address."))
  , "POST": notSupported
  }
```

As resources have to be in scope to be referred, you cannot refer to a non-existing resource. You can, however, refer to an existing resource *that is not routed*. This is described above in Resource Routers.

Erroneously using the `contact` resource together with `formTo` results in a compile error, as there is no handler for the `POST` method in `contact`.

```
Error found:
in module Hyper.HTML.Example

  Could not match type

    Unsupported

  with type

    Supported
```

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. You can also contact Oskar Wickström directly for design discussions. If this project grows, we can setup a mailing list, or other some other means of communication.

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.