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 `writer` field in the `response` record of a Conn is a value provided by the server backend. Functions usually constrain the `writer` field to be a value implementing the `Hyper.Core.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 signatures of functions 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*, writes all specified headers, writes the separating CRLF before the HTTP body, and *marks the state of the response writer 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 HeadersOpen | res } c)
          (Conn req { writer :: rw 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 writer as ended*.

```
respond :: forall r m req res rw c.
           (Monad m, Response r, ResponseWriter rw m) =>
           r
        -> Middleware
           m
           (Conn req { writer :: rw HeadersClosed | res } c)
           (Conn req { writer :: rw 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, when using the Node server, initially a `RequestBody` in the connection. 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. The following type signature resides in `Hyper.Node.Server`, and shows how a request body can be read into a `String`. The `Aff` monad, and the `AVAR` effect, is used to accomplish this asynchronously in the case of the Node server.

```
readBodyAsString    e req res c.
                    Middleware
               (Aff (http  HTTP, err :: EXCEPTION, avar :: AVAR | e))
                    (Conn { body   RequestBody
                          , contentLength :: Maybe Int
                          | req
                          } res c)
```

```
              (Conn {body  String, contentLength :: Maybe Int | req} res c)
```

A simple form parser can use `readBodyAsString` to convert the body a more useful format for the application. The following function checks the `Content-Type` header in the request, splits the request body, builds up a `Form` value, and finally using that value for the `body` field in the resulting `Conn`. The form body has type `Either Error Form` to represent invalid forms.

```
parseForm  forall m req res c.
           Applicative m =>
           Middleware
           m
           (Conn { body   String
                 , headers :: StrMap String
                 | req
                 } res c)
           (Conn { body   Either Error Form
                 , headers :: StrMap String
                 | req
                 }
                 res
                 c)
parseForm conn = do
  let form =
      case lookup "content-type" conn.request.headers >>= parseContentMediaType of
       Nothing -> Left (error "Could not parse content-type header.")
         Just mediaType | mediaType == applicationFormURLEncoded ->
             splitPairs conn.request.body
       Just mediaType -> Left (error $ "Invalid content media type: " <> show mediaType)
  pure (conn { request = (conn.request { body = form }) })
  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 = split (Pattern "&")
                 >>> map splitPair
                 >>> map toTuple
                 >>> sequence
                 >>> map Form
```

More efficient parsers, directly operating on the `RequestBody`, instead of `String`, can of course be built as well.

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.