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

We will start by looking at 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.

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.

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
                               }
```

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 `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 b) =>
           t Header
        -> Middleware
           m
           (Conn req { writer :: rw HeadersOpen | res } c)
           (Conn req { writer :: rw BodyOpen | res } c)
```

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

The `Response` type class describes types that can be written as responses. It takes three type parameters, where `r` is the original response type, `m` is usually an Applicative or a Monad in which the transformation can be performed, and `b` is the target type.

```
class Response m r b | r -> b where
  toResponse :: r -> m 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.

Request Body Parsing

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
  :: forall 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 = ...
```

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

Type-Level Routing

`Hyper.Routing.TypeLevelRouter` provides an API for expressing the web application routes and their characterics as types, much like Servant does. From this type you get static guarantees about having handled all cases, linking only

to existing routes. You also get a lot of stuff for free, such as type-safe parameters for handlers and links.

Let's say we want to render a `User` value as HTML on the `/` path. We start out by declaring the structure of our site:

```
newtype User = User { firstName :: String
                    , lastName :: String
                    }

type MySite = Get HTML User
```

`Get HTML User` describes a structure with only one endpoint, rendering a User as HTML.

So where does the User value come from? We provide it using a *handler*. A handler for `MySite` would be some value of the following type:

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

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

```
root = pure (User { firstName: "John", lastName: "Bonham" })
```

Nice! But what comes out on the other end? We need something that renders the `User` value as HTML. The `MimeRender` type class encapsulates this concept. We provide an instance for `User` and the `HTML` content type.

```
instance mimeRenderUserHTML :: MimeRender User HTML String where
  mimeRender _ (User { firstName, lastName }) =
    asString $
    p [] [ text firstName
         , text " "
         , text lastName
         ]
```

We are getting ready to create the server. First, we need a value-level representation of the `MySite` type, to be able to pass it to the `router` function. For that we use Proxy. 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 MySite` with the value constructor `Proxy`.

```
mySite :: Proxy MySite
mySite = Proxy
```

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

```
onRoutingError status msg =
```

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

siteRouter = router mySite root onRoutingError
```

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

```
main =
  runServer defaultOptions onListening onRequestError {} siteRouter
  where
   onListening (Port port) = log ("Listening on http://localhost:" <> show port)
     onRequestError err = log ("Request failed: " <> show err)
```

Resource Routing

*This module is **deprecated** in favor of the one described in Type-Level Routing.*

`Hyper.Routing.ResourceRouter` aims to provide type-safe routing, based on REST resources. 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

The central concept is *resources*, as in RESTful resources. Each resource is a record describing its *path*, along with a set of HTTP methods and handlers. Each method implemented 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 function `handler` is used to construct `ResourceMethod` values with the `Supported` type parameter.

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

Resource Routers

The `router` function creates a `ResourceRouter` out of a resource record. The router 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 = runRouter defaultRouterFallbacks (router 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 =
  runRouter
  defaultRouterFallbacks
  (router index <|> router about <|> router contact)
```

Router Fallbacks

The router has a *fallback* concept - functions that provide a response in case no resource matched the request. For instance, if a route path matches the request URL, but not the method, the chain short-circuits and the `onMethodNotAllowed` function provides a response for the *405 Method Not Allowed* case. The same goes for `onNotFound`, in the case of a *404 Not Found*.

The `defaultRoutesFallbacks` can be used to get a set of basic fallbacks. If custom responses are desired, simply provide your own fallbacks, or override some of the defaults.

```
fallbacks =
  { onNotFound:
    writeStatus statusNotFound
    >=> headers []
    >=> respond "What are you doing here?"
  , onMethodNotAllowed:
    \method ->
    writeStatus statusMethodNotAllowed
    >=> headers []
   >=> respond ("No way I'm routing a " <> show method <> " request.")
  }

app = runRouter fallbacks (router index)
```

Type-Safe Links and Forms

The resource router module also provides 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 =
  resource
  { path = ["about"]
  , "GET" = handler (\conn -> respond
                              (linkTo contact (text "Contact Me!"))
                               conn)
  }

contact =
  resource
  { path = ["contact"]
  , "GET" = handler (respond (text "Good luck finding my email address."))
  }
```

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 Example

  Could not match type

    Unsupported

  with type

    Supported
```

Servers

Although Hyper middleware are regular functions, which can applied to Conn values, you often 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.

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
  onListening (Port port) =
    log ("Listening on http://localhost:" <> show port)
  onRequestError err =
    log ("Request failed: " <> show err)
  app =
    writeStatus (Tuple 200 "OK")
    >=> closeHeaders
    >=> respond "Hello there!"
in runServer defaultOptions onListening onRequestError {} app
```

As seen above, `runServer` takes a record of options, two callbacks, an initial *components* record, and your application middleware.

Testing

When running tests you might not want to start a full HTTP server and sends 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: {}
          }
          # app
          # testServer
  testStatus conn `shouldEqual` Just statusOK
  testStringBody conn `shouldEqual` "Hello there!"
```

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.