Purely-Functional Web Apps

Using React and PureScript

Phil Freeman

Intro

- Please ask questions at any time
- Useful resources:
    - `purescript-thermite` module documentation
    - pursuit.purescript.org for general library help

About Me

- Haskell + Scala + TypeScript + PureScript
- Original developer of the PureScript compiler
- Wrote `purescript-thermite` and `purescript-halogen`

React

- The UI is a function of application state
- DOM updates become implicit
- An excellent fit for PureScript
- See also virtual-dom

Thermite

- A PureScript library which wraps React
- The UI is described by
  - A state type
  - An action type
  - A rendering function
  - An event handler

Modules

The usual modules:

```
import qualified Thermite                 as T
import qualified Thermite.Html            as T
import qualified Thermite.Html.Elements   as T
import qualified Thermite.Html.Attributes as A
import qualified Thermite.Action          as T
import qualified Thermite.Events          as T
import qualified Thermite.Types           as T
```

Later:

```
import qualified Thermite.SVG            as S
import qualified Thermite.SVG.Attributes as SA
```

Rendering

```
type Render eff state props action
  = Context state action -> -- Event context
    state ->                 -- Component state
    props ->                 -- Component properties
    [Html eff] ->            -- Child elements
    Html eff
```

HTML Documents

```
type State = State { on :: Boolean }

render _   state _ _ =
  H.div (A.className "container")
    [ H.h1' [ T.text "Switch" ]
    , H.p' [ T.text (if state.on
                      then "On"
                      else "Off")
           ]
    ]
```

- HTML attributes form a Monoid (use <> to combine them)
- Child elements are specified in an array

Event Handlers

In `Thermite.Events`:

```
onClick :: forall state props action.
              Context state action ->
              (MouseEvent -> action) ->
              Attr
```

Event handlers receive the React `Context` as an argument

(essentially just ‘`this`’)

so that they can update state etc.

Event Handlers

```
type State = State { on :: Boolean }

data Action = ToggleSwitch

render ctx state _ _ =
  H.div (A.className "container")
    [ H.h1' [ T.text "Switch" ]
    , H.p' [ T.text (if state.on
                     then "On"
                     else "Off")
           ]
    , H.button (E.onClick ctx \_ -> ToggleSwitch)
        [ T.text "Toggle" ]
    ]
```

Interpreting Actions

Once we've generated our actions, we need to *interpret* them:

```
type PerformAction eff state props action
  = props ->                       -- Component properties
    action ->                      -- Action to perform
    Action eff state Unit
```

Interpreting Actions

For example:

```
performAction :: PerformAction eff State props Action
performAction _ ToggleState =
  T.modifyState \st -> { on: not st.on }
```

- The first argument gives the component properties (not used here)
- The second argument is the action to perform
- The result is a computation in the `Action` monad (we’ll come back to this)

Components

- In React, we define *component classes*.
- Component classes can be used to bundle functionality for reuse.
- In Thermite, we build a component class from a `Spec`:

```
newtype Spec eff state props action

simpleSpec :: forall eff state props action.
  state ->
  PerformAction eff state props action ->
  Render eff state props action ->
  Spec eff state props action
```

Putting Things Together

Our final component class can be rendered in `main`:

```
main = do
  let component = T.createClass spec
  T.render component {}
```

Demo and Exercises

Exercise Set 1:

- Compile and run the code in the `warmup/` directory.
- Modify the state to include an integer-valued counter.
- Add a label to display the current state.
- Add a button to increment the counter.
- Add a button to reset the counter.

The Action Monad

In this section:

- The `Action` monad and its operations
- The `aff` and `affjax` libraries for AJAX calls

Setup:

- `git stash && git checkout like`

purescript-aff

- `Aff` is an asynchronous effect monad which supports error handling.
- Effects:

```
liftEff :: forall e a. Eff e a -> Aff e a
```

- Error handling:

```
attempt :: forall e a. Aff e a -> Aff e (Either Error a)
throwError :: forall e a. Error -> Aff e a
```

Concurrency

The `Par` applicative functor supports parallel composition of asynchronous results:

```
runPar (preparePage <$> Par Ajax.loadLanguages
                    <*> Par Ajax.loadTags)
```

`runPar` and `Par` allow parallel (`Par`) and sequential (`Aff`) portions of code to be interleaved.

Parallel-for is simple:

```
parFor f xs = runPar $ traverse (Par <<< f) xs
```

affjax

`affjax` is an AJAX library built on top of `Aff`:

```
do tags <- get “api/tag”
   parFor loadTagDetails tags
   where
   loadTagDetails tag = get (“api/tag/” <> tag.id)
```

Actions

The Action monad supports the following operations:

```
getState :: forall eff state. Action eff state state

setState :: forall eff state. state -> Action eff state Unit

modifyState :: forall eff state. (state -> state) ->
                                 Action eff state Unit
```

Actions

The Action monad supports the following operations:

```
sync :: forall e state a. Eff e a ->
                          Action e state a

async :: forall e state a. ((a -> Eff e Unit) -> Eff e Unit) -
>
                             Action e state a

asyncSetState :: forall e state. ((state -> Eff e Unit)
                                         -> Eff e Unit) ->
                                 Action e state Unit
```

Aff + Action

Just have to make the pieces fit:

```
runAff :: forall e a.
          (Error -> Eff e Unit) -> -- Error handler
          (a -> Eff e Unit) ->     -- Success handler
          Aff e a ->               -- Async computation
          Eff e Unit

async :: forall e state a. ((a -> Eff e Unit) -> Eff e Unit) -
>
                                 Action e state a
```

We want a function `Aff e a -> Action e state a`

Demo and Exercises

Exercise Set 2:

- Add a new `Action` data constructor to `Like` a language
- Attach the Like button to the `Like` action
- Write a function using `affjax` which wraps `POST /api/lang/:id/like`
- Implement `Like` in the `performAction` function

Additional exercise:

- Modify the Home action handler to load tags and languages in parallel.

Coffee Break

Applicative Validation

In this section:

- `Applicative` functors and validation

Setup:

- `git stash && git checkout validation`
- `pulp dep update`

Applicative Functors

Reminder:

- Every `Monad` is an `Applicative`, not every `Applicative` is a `Monad`.
- `Applicative` functors let us lift functions of n `>=` `0` arguments over a type constructor:

```
data PageState = PageState [Lang] [Tag]

loadPage :: Par PageState
loadPage = PageState <$> loadLangs <*> loadTags
```

Monadic Validation

We can use the Either monad to validate data:

```
type FormState = { name :: String, city :: String }

type ValidationError = String

validateForm :: FormState -> Either ValidationError FormState
validateForm fs = do
  name <- validateName fs.name
  city <- validateCity fs.city
  …
  return { name: name, city: city, … }
```

But there is a problem...

Multiple Errors

Monadic validation only gives us *the first error*:

```
> validateForm { name: “Phil”, city: “” }
Left “City is required”

> validateForm { name: “”, city: “” }
Left “Name is required”
```

We want both errors!

Applicative validation solves this problem.

Applicative Validation

The V functor is a validation functor with an Applicative instance but no Monad instance.

V collects errors in parallel, where our errors type is any Semigroup:

```
type FormState = { name :: String, city :: String }

type ValidationError = [String]

validateForm :: FormState -> V ValidationError FormState
validateForm fs = { name: _, city: _ }
  <$> validateName fs.name
  <*> validateCity fs.city
```

V

V provides the following functions:

```
invalid :: forall err result. err -> V err result

isValid :: forall err result. V err result -> Boolean

runV :: forall err result r. (err -> r) ->      -- Failure
                             (result -> r) ->  -- Success
                             V err result ->
                             r
```

Validators

The simplest validator checks for a required field:

```
required :: String -> V ValidationErrors String
required “” = invalid [“Field was required”]
required s  = pure s
```

We can write lots more:

- Regular expression
- Comparison
- Range validator

Multiple Errors

Applicative validation gives us all errors:

```
> validateForm { name: “Phil”, city: “” }
Left [“City is required”]

> validateForm { name: “”, city: “” }
Left [“Name is required”, “City is required”]
```

Enhancing Errors

Suppose we want to give our errors more structure:

- Associate a severity with an error message
- Display each error alongside its form field
- Use `HTML` instead of `String`
- Use a `Set` or `Map` instead of an `Array`

With `V`, we can use any `Semigroup`.

Demo and Exercises

Exercise Set 3:

- Use the Data.String.Regex module to implement a function

  ```
  matchesRegex :: Regex -> String -> V ValidationErrors String
  ```

- Use your function to validate the homepage field.

Additional exercises:

- Write a function to validate the `tags` field (should be non-empty and without duplicates)
- Modify the `ValidationErrors` type to use `Map FieldId String`.

SVG Graphics

In this section:

- Interactive vector graphics with React & SVG

Setup:

- `git stash && git checkout svg`
- `pulp dep update`
- Restart the server

SVG

SVG (Scalable Vector Graphics) is

- a markup language for vector graphics
- embedded in `<svg>` tags in HTML and rendered in browsers
- supported by React and Thermite

Basic Shapes

Rectangles:

```
rect :: forall eff. Attr -> [Html eff] -> Html eff
```

For example:

```
S.rect (A.width “100”
          <> A.height “150”
          <> SA.x “0”
          <> SA.Y “50”) []
```

Basic Shapes

Ellipses:

```
circle, ellipse :: forall eff. Attr -> [Html eff] -> Html eff
```

For example:

```
S.circle (SA.r “50”
            <> SA.cx “100”
            <> SA.cy “120”) []
S.ellipse (SA.rx “50”
            <> SA.ry “80”
            <> SA.cx “100”
            <> SA.cy “120”) []
```

Text

Text:

```
text :: forall eff. Attr -> [Html eff] -> Html eff
```

For example:

```
S.text (SA.x “50”
            <> SA.y “100”
            <> SA.fontSize “16px”
            <> SA.fontFamily “Comic Sans MS”
            <> Unsafe.innerHTML “SVG!”) []
```

We have to use `innerHTML` due to a limitation of React :(

Colors

```
fill, stroke :: String -> Attr
```

For example:

```
S.circle (SA.r “50”
            <> SA.cx “100”
            <> SA.cy “120”
            <> SA.fill “red”
            <> SA.stroke “black”) []
```

Grouping

Grouping elements:

```
g :: forall eff. Attr -> [Html eff] -> Html eff
```

For example:

```
S.g mempty
  [ S.circle ...
  , S.text ...
  ]
```

And more...

- Transformations
- Gradients
- Clipping
- Transparency
- …

`purescript-svg` library, anyone …?

Demo and Exercises

Exercise Set 4:

- Add a function which wraps the `GET /api/popular` call in `UI.AJAX`.
- Use your function to load popular languages in the `LoadList` action.
- Update the render function to render a bar graph of the top 5 most popular languages.

Additional exercises:

- An an `onClick` handler to each bar, linking to the appropriate language
- Try a different type of chart (bubble, pie, etc.)

Routing Combinators

In this section:

- Type-safe routing using `Applicative` and `Alternative`
- History API integration

Setup:

- `git stash && git checkout routes`
- `pulp dep update`

Routing

The problem:

- We want a concise syntax for precisely specifying routes in our application.
- We want a type-safe representation of routes
- We want composable errors
- We want to use the History API

The `purescript-routing` library solves these problems.

Literals and Variables

The simplest routes are constants:

```
about :: Match Unit
about = lit “about”
```

And variables:

```
str  :: Match String
num  :: Match Number
bool :: Match Boolean
```

We want to combine these simple routes.

Applicative Routing

`Applicative` parsing gives an elegant way to represent context-free grammars by combining simpler grammars.

Start with an ADT representing routes:

```
newtype UserID = UserID String
newtype PostID = PostID String

data Route
  = Home
  | User UserID
  | Post UserID PostID
```

Applicative Routing

The `Match a` type represents parsers which attempt to construct matches of type `a`.

`Home` is easy to parse:

```
home :: Match Route
home = pure Home
```

Applicative Routing

To parse User, we need Functor:

```
user :: Match Route
user = User <$> userId

userId :: Match UserID
userId = UserID <$> str
```

Applicative Routing

To parse `Post`, we need `Applicative`:

```
post :: Match Route
post = Post <$> userId <*> postId

postId :: Match PostID
postId = PostID <$> str
```

Why?

- `Functor` lets us lift 1-ary functions
- `Applicative` lets us lift n-ary functions

Alternatives

We want to combine all of our routes into a single routing table.

`Alternative` lets us combine multiple alternatives:

```
(<|>) :: forall f a. (Alternative f) => f a -> f a -> f a
```

We can write:

```
routes :: Match Route
routes = home <|> user <|> post
```

Not quite right...

Ordering Routes

We can write:

```
routes :: Match Route
routes = home <|> user <|> post
```

What happens when we visit `/user/phil/post/lambdaconf`?

The `user` route matches first (remaining input is ignored)

Ordering is important!

```
routes :: Match Route
routes = home <|> post <|> user
```

Factoring Routes

The `Alternative` laws say that we can refactor safely.

```
data Route
  = Home
  | User UserID
  | Post UserID PostID
```

```
routes :: Match Route
routes = home <|> user <|> post
  where
  home :: Match Route
  home = pure Home

  user :: Match Route
  user = User <$> userId

  post :: Match Route
  post = Post <$> userId
              <*> postId
```

Factoring Routes

Combine User and Post:

```
data Route


  = Home


  | User UserID UserRoute



data UserRoute
  = Post PostID
  | UserHome
```

```
routes :: Match Route
routes = home <|> user
  where
  home :: Match Route
  home = pure Home

  user :: Match Route
  user = User <$> userId
              <*> userRoute

  userRoute :: Match UserRoute
  userRoute = Post <$> postId
            <|> pure UserHome
```

Routing Errors

`purescript-routing` defines a rich type of parsing errors:

```
data MatchError -- One error
```

`Match` is implemented in terms of `Free MatchError`, the free `Semiring`.

We can fail in many ways:

- Parsers can fail in series, as part of a single alternative
- Multiple alternatives can fail in parallel

These correspond to multiplication and addition of errors (errors form a `Semiring`!)

When we fail, we get detailed errors.

Routes API

The `purescript-routing` API can be summarized in a single type class:

```
class (Alternative f) <= MatchClass f where
  lit   :: String -> f Unit           -- Literals
  str   ::           f String         -- Variables
  num   ::           f Number
  bool  ::           f Boolean
  param :: String -> f String         -- Query parameters
  fail  :: forall a. String -> f a -- Failure
```

along with certain laws, which tell us when and how it is safe to refactor.

History API

Getting onhashchanged updates is simple:

```
matches :: forall e a.
           Match a ->                        -- Routing table
           (Maybe a -> a -> Eff e Unit) ->   -- Callback
           Eff e Unit
```

Integrating with Actions

To use `matches` in the `Action` monad, we can use `async`:

```
subscribe :: T.Action Unit
subscribe = do
  nextAction <- T.async $ \k ->
                  matches route \_ action ->
                    k action
  performAction props nextAction
```

Demo and Exercises

Exercise Set 5:

- Modify the routing table to add routes for `LoadTag` and `LoadEditLang`
- Verify that your new routes work and have the correct ordering
- Modify the UI to link to your routes using `<a href>`

Additional exercises:

- Refactor the `Action` type to extract a `Route` ADT.
- Factor your `Route` ADT to bring `Key` to the left.

EOF

Join us on #purescript IRC!