Formlets

The Essence of Form Abstraction

Motivation

- We’ve got web sites with forms.
- Layout
- Input fields
- Validate
- Consume
- Complicated
- Server side factor

Formlets *POW!*

Formlets *POW!*

Formlets

- Invented by Wadler (yep, that one) The Essence of Form Abstraction in 2008
  - OCaml for Links, with “JSX”-style syntax
  - An Idiom’s Guide to Formlets - 2007
- Chris Eidhof made the formlets package in 2008
- I got excited and wrote a little blog post about it in 2008
- Jasper Van der Jeugt made the digestive-functors package in 2010
- 🌬️ … (tumbleweed, a few other libs, MFlow, ..)
- See also Client-side web programming in Haskell: A retrospective

The essence of a form (classic web programming)

- Parser
- View
- Run it
- Generate (unique form field IDs)
- Hydration
- Errors

A formlet in Haskell

Note: Couples view and parse into one type.

data Form a

instance Applicative Form

1. runForm :: [(Text,Text)] -> Form a -> (Html, Either [Error] a)
2. generateForm :: Form a -> Html

*optional vs required

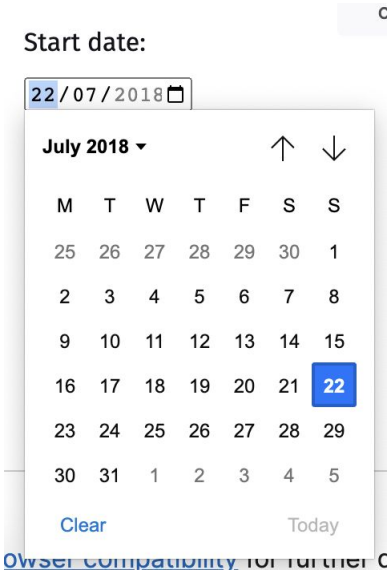
```
*Register> :load "/var/www/chrisdone/blog/db/static/Register.hs"
[1 of 1] Compiling Register ( Register.hs, interpreted )
Ok, modules loaded: Register.
*Register> let env = map (("input"++) . show) [0..] `zip`
                     map Left ["chris","mypassword","mypassword"]
*Register> let (result,xml,_) = runFormState env "" register
*Register> putStrLn . X.prettyHtmlFragment =<< xml
<p>
   <label>
      Username:
   </label>
   <input type="text" name="input0" id="input0" value="chris" />
</p>
<input type="password" name="input1" id="input1" value="mypassword" />
<input type="password" name="input2" id="input2" value="mypassword" />

*Register>
```

Combinators

textInput :: Maybe Text -> Form Text

intInput :: Maybe Int -> Form Int

dateInput :: Maybe Day -> Form Day



Parsing

Re-use: ints and time

```
intInput = bidirectional parseInt printInt textInput
  where parseInt :: Text -> Either Error Int
        printInt :: Int -> Text
timeForm time = Time
    <$> intInput (fmap hour time)
    <*> intInput (fmap min time)
    <*> intInput (fmap secs time)
```

Re-use: ints and time

```
timeForm .. = Time <$> hourForm … <*> minForm … <*> secForm …
  where hourForm = bidirectional parseHour unHour intInput
        …
```

Re-use: Passwords

```
passwordInput =
  parsing (minLength 3 >=> mustHave (not.isAlphaNum)) . textInput

where
minLength :: Text -> Either Error Text
mustHave :: Text -> Either Error Text
```

Re-use: Passwords

```
passwordsInput mdef =
  parsing stringsEqual ((,) <$> passwordInput mdef <*> passwordInput mdef)
where stringsEqual :: (Text,Text) -> Either Error Text
```

Monadic parsing is also fine

parsingM :: Monad m => (a -> **m** (Either Error b)) -> Form **m** a -> Form **m** b

Layout

HTML

html :: Html -> Form ()

Example: left <* html (p_ “hello”) <* right

Wrapping with markup

wrap :: (Html -> Html) -> Form a -> Form a

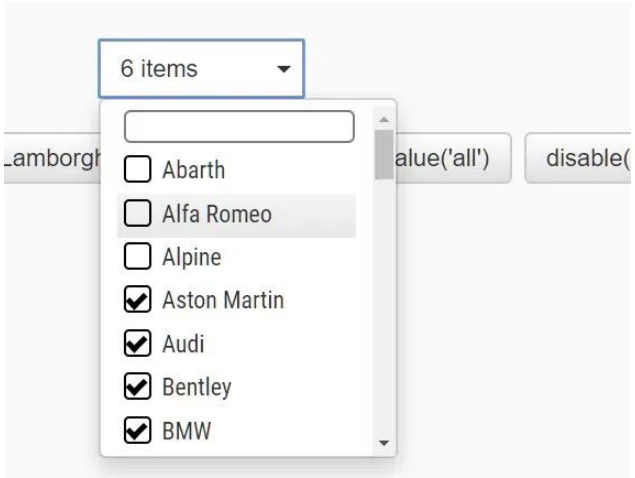
So

label :: Text -> Form a -> Form a

label name = wrap (\inner -> label_ (do toHtml name; inner))

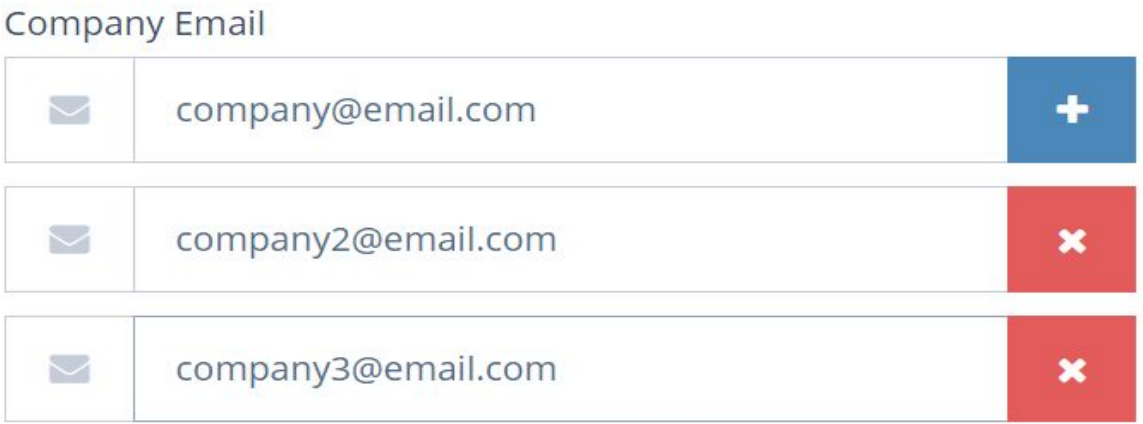
Plural inputs

selectManyInput :: [(Text,a,Bool)] -> Form [a]



Dynamism

- Small hints of JavaScript enhance more principled combinators, e.g.
  - manyForm :: Form a -> Form [a]



What we gained

- **Simplicity**
- Components
- Composition
- Nesting
- Coupling
- Flexibility of layout
- Decomposition of problems
- “Parse Don’t Validate”
- Applicative, so, traverse, alternative, etc.

Similar things you know and love

- Optparse-applicative:
- --help
- Run this and –that and -those and -a etc.
- To some extent, Reflex.
- React hooks–kinda?
- Halogen–kinda?

Challenges and design space considerations

- Uniqueness of field names
- Auto-generated fixes most things, but some things need completion from the browser
- Required vs optional
- Custom error types
- Capture error messages that float up: 🎈
  - ceiling :: ([Error] -> Html -> Html) -> Form a -> Form a
- Drop error messages down: 💣
  - floor :: ([Error] -> Html -> Html) -> Form a -> Form a
- More advanced “Many” forms
- “Bind”/Selective – using results of previous fields

The original one

```
let date_formlet : date formlet = formlet
  <div>
    Month: {input_int ⇒ month}
    Day: {input_int ⇒ day}
  </div>
yields  make_date month day

let travel_formlet : (string × date × date) formlet =
  formlet
    <#>
      Name: {input ⇒ name}
      <div>
        Arrive: {date_formlet ⇒ arrive}
        Depart: {date_formlet ⇒ depart}
      </div>
      {submit "Submit"}
    </#>
  yields  (name, arrive, depart)

let display_itinerary : (string × date × date) → xml =
  fun (name, arrive, depart) →
    <html>
      <head><title>Itinerary</title></head>
      <body>
        Itinerary for: {xml_text name}
        Arriving: {xml_of_date arrive}
        Departing: {xml_of_date depart}
      </body>
    </html>

handle travel_formlet display_itinerary
```

Fig. 1. Date example

“Newer” attempts I played around with

- https://github.com/chrisdone/forge
- https://github.com/chrisdone/named-formlet
- https://hackage.haskell.org/package/yesod-form - didn’t learn much from formlets

But, on the whole, the world has turned to SPAs (and forgotten about formlets).

Interesting points

- Htmx - a resurgence of SSR
- How does that interact with formlets?

FIN

Any questions?