Web applications

What does a typical web application look like?

- Database
  - WhateverSQL ElasticSearch, Mongo, Redis, ...
- API client
- Back-end
  - webserver, request routing, API endpoints
- API
  - HTML
  - JSON
  - ...
- Front-end
  - Javascript
- ~~Browser~~

Haskell can be deployed at all levels.

What does a typical web application look like?

- Database
  - WhateverSQL ElasticSearch, Mongo, Redis, ...
- API client
- Back-end
  - webserver, request routing, API endpoints
- API
  - HTML
  - JSON
  - ...
- Front-end
  - Javascript
- ~~Browser~~

Haskell can be deployed at all levels.

HTTP client

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "http://website.lu/page"

  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

HTTP client

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "http://website.lu/page"

  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

HTTP client

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "http://website.lu/page"

  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

HTTP client

```
import Network.HTTP.Simple
import qualified Data.ByteString.Lazy.Char8 as L8

main = do
  response ← httpLBS "http://website.lu/page"

  print (getResponseStatusCode response)
  L8.putStrLn $ getResponseBody response
```

API client

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://website.lu/thing"
  let request
      = setRequestMethod "PUT"
      $ setRequestPath "/drop"
      $ setRequestQueryString [("parameter", Just "value")]
      $ setRequestBodyLBS "DataData"
      $ setRequestSecure True
      $ setRequestPort 443
      $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

API client

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://website.lu/thing"
  let request
      = setRequestMethod "PUT"
      $ setRequestPath "/drop"
      $ setRequestQueryString [("parameter", Just "value")]
      $ setRequestBodyLBS "DataData"
      $ setRequestSecure True
      $ setRequestPort 443
      $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

API client

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://website.lu/thing"
  let request
      = setRequestMethod "PUT"
      $ setRequestPath "/drop"
      $ setRequestQueryString [("parameter", Just "value")]
      $ setRequestBodyLBS "DataData"
      $ setRequestSecure True
      $ setRequestPort 443
      $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

API client

```
import Data.Aeson (Value)
import qualified Data.ByteString.Char8 as S8
import qualified Data.Yaml as Yaml
import Network.HTTP.Simple

main :: IO ()
main = do
  request' ← parseRequest "POST http://website.lu/thing"
  let request
      = setRequestMethod "PUT"
      $ setRequestPath "/drop"
      $ setRequestQueryString [("parameter", Just "value")]
      $ setRequestBodyLBS "DataData"
      $ setRequestSecure True
      $ setRequestPort 443
      $ request'
  response ← httpJSON request

  print $ getResponseHeader "Content-Type" response
  S8.putStrLn $ Yaml.encode (getResponseBody response :: Value)
```

Parsing and generating of JSON is done via Data.Aeson:

```
data Value = Object (Map Text Value)
           | Array (Vector Value)
           | String Text | Number Number | Bool Bool
           | Null

class FromJSON a where
  parseJSON :: Value → Parser a

class ToJSON a where
  toJSON    :: a → Value
```

```
data Amount = Amount {count :: Double,
                      unit :: String}

instance ToJSON Amount where
  toJSON (Amount c u) = Object $
    fromList [("count", Number (D c)),
              ("unit", String u)]

instance FromJSON Amount where
  parseJSON (Object a) =
    Amount <$> a .: "count"
           <*> a .: "unit"
  parseJSON _ = mzero
```

```
data Amount = Amount {count :: Double,
                      unit :: String}

instance ToJSON Amount where
  toJSON (Amount c u) = Object $
    fromList [("count", Number (D c)),
              ("unit", String u)]

instance FromJSON Amount where
  parseJSON (Object a) =
    Amount <$> a .: "count"
           <*> a .: "unit"
  parseJSON _ = mzero
```

```
data Amount = Amount {count :: Double,
                      unit :: String}

instance ToJSON Amount where
  toJSON (Amount c u) = Object $
    fromList [("count", Number (D c)),
              ("unit", String u)]

instance FromJSON Amount where
  parseJSON (Object a) =
    Amount <$> a .: "count"
           <*> a .: "unit"
  parseJSON _ = mzero
```

```
data Amount = Amount {count :: Double,
                      unit :: String}

instance ToJSON Amount where
  toJSON (Amount c u) = Object $
    fromList [("count", Number (D c)),
              ("unit", String u)]

instance FromJSON Amount where
  parseJSON (Object a) =
    Amount <$> a .: "count"
           <*> a .: "unit"
  parseJSON _ = mzero
```

Advantage: practically all JSON-ish packages can transparently use the Aeson conversion instances; many types have the instances already implemented.

Manual use:

$$\mathit{encode} :: \mathrm{ToJSON}\ a \Rightarrow a \to \mathrm{ByteString}$$
$$\mathit{decode} :: \mathrm{FromJSON}\ a \Rightarrow \mathrm{ByteString} \to \mathrm{Maybe}\ a$$

$$\mathit{decode}\ \texttt{"[123,234]"} :: \mathrm{Maybe}\ [\mathrm{Int}]$$
$$\rightsquigarrow \mathrm{Just}\ [123, 234]$$
$$\mathit{encode}\ [123, 234]$$
$$\rightsquigarrow \texttt{"[123,234]"}$$

Aeson automatically

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics

data Amount = Amount {count :: Double,
                      unit :: String}
  deriving (Show, Generic)

instance ToJSON Amount where
  toJSON = genericToJSON

instance FromJSON Amount where
  parseJSON = genericParseJSON

decode "{ \"count\": 123, \"unit\": \"cm\" }" :: Maybe Amount
    ⇝ Just (Amount {count = 123.0, unit = "cm"})
encode $ Amount 42 "kg"
    ⇝ "{\"unit\":\"kg\",\"count\":42}"
```

Aeson automatically

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics

data Amount = Amount {count :: Double,
                      unit :: String}
  deriving (Show, Generic)

instance ToJSON Amount where
  toJSON = genericToJSON

instance FromJSON Amount where
  parseJSON = genericParseJSON

decode "{ \"count\": 123, \"unit\": \"cm\" }" :: Maybe Amount
    ⇝ Just (Amount {count = 123.0, unit = "cm"})
encode $ Amount 42 "kg"
    ⇝ "{\"unit\":\"kg\",\"count\":42}"
```

Aeson automatically with some customization

Use of *defaultOptions* :: Options:

```
instance ToJSON Amount where
  toJSON = genericToJSON $ defaultOptions
    {fieldLabelModifier = λs → "amount_" ◇ s}

encode $ Amount 42 "kg"
  ⇝ "{\"amount_unit\":\"kg\",\"amount_count\":42}"
```

Options may be used similarly with FromJSON.

Aeson automatically with some customization

Use of *defaultOptions* :: Options:

```
instance ToJSON Amount where
  toJSON = genericToJSON $ defaultOptions
    {fieldLabelModifier = λs → "amount_" ◇ s}

encode $ Amount 42 "kg"
    ⇝ "{\"amount_unit\":\"kg\",\"amount_count\":42}"
```

Options may be used similarly with FromJSON.

Aeson automatically with some customization

Use of *defaultOptions* :: Options:

```
instance ToJSON Amount where
  toJSON = genericToJSON $ defaultOptions
    {fieldLabelModifier = λs → "amount_" ◇ s}

encode $ Amount 42 "kg"
    ⇝ "{\"amount_unit\":\"kg\",\"amount_count\":42}"
```

Options may be used similarly with FromJSON.

Generating the HTML manually is quite straightforward. Libraries like Blaze and Lucid additionally minimize the overhead of gluing various types of things together.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Generating the HTML manually is quite straightforward. Libraries like Blaze and Lucid additionally minimize the overhead of gluing various types of things together.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Most of the combinators accepts 'contents' — the monad interface is used as a 'builder'

```
numbers n = do
  head_ $ title_ "Natural numbers"
  body_ $ do
    p_ "A list of natural numbers:"
    ul_ $ forM_ [1..n] (li_ . toHtml . show)
```

Generating the HTML manually is quite straightforward. Libraries like Blaze and Lucid additionally minimize the overhead of gluing various types of things together.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Most of the combinators accepts 'contents' — the monad interface is used as a 'builder'

```
numbers n = do
  head_ $ title_ "Natural numbers"
  body_ $ do
    p_ "A list of natural numbers:"
    ul_ $ forM_ [1..n] (li_ . toHtml . show)
```

Generating the HTML manually is quite straightforward. Libraries like Blaze and Lucid additionally minimize the overhead of gluing various types of things together.

```
{-# LANGUAGE OverloadedStrings #-}
import Lucid
```

Most of the combinators accepts 'contents' — the monad interface is used as a 'builder'

```
numbers n = do
  head_ $ title_ "Natural numbers"
  body_ $ do
    p_ "A list of natural numbers:"
    ul_ $ forM_ [1..n] (li_ . toHtml . show)
```

Attributes:

```
image :: Html ()
image = img_
  [class_ "styled"
  ,src_ "lambda.png"
  ,alt_ "An image of lambda."
  ,id_ "theimage"
  ]
```

Attributes:

```
image :: Html ()
image = img_
  [class_ "styled"
  ,src_ "lambda.png"
  ,alt_ "An image of lambda."
  ,id_ "theimage"
  ]
```

Tag contents

```
paragraph :: Html ()
paragraph = p_ [class_ "nicely-aligned"] $ em_ "Some sentence!"
```

```
import Database.SQLite.Simple

dbBracket :: (Connection → IO a) → IO a
dbBracket = withConnection "database.sqlite"

createUser user password =
  dbBracket $ λdb → do
    pwhash ← newHash password
    execute
      db
      "INSERT INTO users (user, pwhash) VALUES (?,?)"
      (user :: String, pwhash :: String)
```

SQLite overview

```
import Database.SQLite.Simple

dbBracket :: (Connection → IO a) → IO a
dbBracket = withConnection "database.sqlite"

createUser user password =
  dbBracket $ λdb → do
    pwhash ← newHash password
    execute
      db
      "INSERT INTO users (user, pwhash) VALUES (?,?)"
      (user :: String, pwhash :: String)
```

SQLite overview

```
import Database.SQLite.Simple

dbBracket :: (Connection → IO a) → IO a
dbBracket = withConnection "database.sqlite"

createUser user password =
  dbBracket $ λdb → do
    pwhash ← newHash password
    execute
      db
      "INSERT INTO users (user, pwhash) VALUES (?,?)"
      (user :: String, pwhash :: String)
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
verifyPassword user password err f =
  dbBracket $ λdb →
    withTransaction db $ do
      res ←
        query
          db
          "SELECT pwhash FROM users WHERE user = ?"
          (Only (user :: String)) :: IO [[String]]
      case res of
        [[hash]] →
          if checkHash password hash
            then f db
            else err
        _ → err
```

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, OverloadedLabels #-}
import Database.Selda
import Database.Selda.SQLite

data Pet = Dog | Horse | Dragon
  deriving (Show, Read, Bounded, Enum)

instance SqlType Pet

data Person = Person
  { name :: Text
  , age  :: Int
  , pet  :: Maybe Pet
  } deriving Generic

instance SqlRow Person

people :: Table Person
people = table "people" [#name :- primary]
```

SQL DSL (Selda)

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, OverloadedLabels #-}
import Database.Selda
import Database.Selda.SQLite

data Pet = Dog | Horse | Dragon
  deriving (Show, Read, Bounded, Enum)

instance SqlType Pet

data Person = Person
  { name :: Text
  , age  :: Int
  , pet  :: Maybe Pet
  } deriving Generic

instance SqlRow Person

people :: Table Person
people = table "people" [#name :- primary]
```

SQL DSL (Selda)

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings, OverloadedLabels #-}
import Database.Selda
import Database.Selda.SQLite

data Pet = Dog | Horse | Dragon
  deriving (Show, Read, Bounded, Enum)

instance SqlType Pet

data Person = Person
  { name :: Text
  , age  :: Int
  , pet  :: Maybe Pet
  } deriving Generic

instance SqlRow Person

people :: Table Person
people = table "people" [#name :- primary]
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"     19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"       10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name :* person ! #pet
  liftIO $ print adultsAndTheirPets
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"     19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"       10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name :* person ! #pet
  liftIO $ print adultsAndTheirPets
```

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"     19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"       10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name :* person ! #pet
  liftIO $ print adultsAndTheirPets
```

SQL DSL (Selda)

```
main = withSQLite "people.sqlite" $ do
  createTable people
  insert_
    people
    [Person "Velvet"     19 (Just Dog)
    ,Person "Kobayashi" 23 (Just Dragon)
    ,Person "Miyu"       10 Nothing
    ]
  adultsAndTheirPets ← query $ do
    person ← select people
    restrict (person ! #age . >= 18)
    return $ person ! #name :* person ! #pet
  liftIO $ print adultsAndTheirPets
```

HTTP Back-end

HTTP backend package overview:

- `wai` — Web Application Interface, a low-level base for HTTP servers
- `warp` — A fast HTTP server
- Yesod — `warp` wrapped with a RESTful data model and template haskell
- Scotty — A noticeably thinner and more practical wrapping of `warp`
- Servant — Type-defined APIs, automatic generation of server, client and javascript code

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types
import Network.Wai.Handler.Warp (run)

app :: Application
app _ respond = do
  putStrLn "Someone wants a page!"
  respond $ responseLBS
    status200
    [("Content-Type","text/html")]
    "<html><body><h1>Hello there!</h1></body></html>"

main = run 8080 app
```

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types
import Network.Wai.Handler.Warp (run)

app :: Application
app _ respond = do
  putStrLn "Someone wants a page!"
  respond $ responseLBS
    status200
    [("Content-Type","text/html")]
    "<html><body><h1>Hello there!</h1></body></html>"

main = run 8080 app
```

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/greetings/:name" $ do
    nm ← param "name"
    html $ "<h1>Hello, " ◇ nm ◇ "!</h1>"
  post "/log/:message" $ do
    param "message" ≫= liftIO T.putStrLn ∘ ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/greetings/:name" $ do
    nm ← param "name"
    html $ "<h1>Hello, " ◇ nm ◇ "!</h1>"
  post "/log/:message" $ do
    param "message" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/greetings/:name" $ do
    nm ← param "name"
    html $ "<h1>Hello, " ◇ nm ◇ "!</h1>"
  post "/log/:message" $ do
    param "message" ≫= liftIO T.putStrLn ∘ ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/greetings/:name" $ do
    nm ← param "name"
    html $ "<h1>Hello, " ◇ nm ◇ "!</h1>"
  post "/log/:message" $ do
    param "message" >>= liftIO T.putStrLn . ("Log: "◇)
    json $ M.fromList [("status" : "ok")]
```

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Map as M

main = scotty 8080 $
  get "/greetings/:name" $ do
    nm ← param "name"
    html $ "<h1>Hello, " ◇ nm ◇ "!</h1>"
  post "/log/:message" $ do
    param "message" >>= liftIO T.putStrLn . ("Log: " ◇)
    json $ M.fromList [("status" : "ok")]
```

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeOperators #-}

import Prelude ()
import Prelude.Compat

import Control.Monad.Except
import Control.Monad.Reader
import Data.Aeson
import Data.Aeson.Types
import Data.Attoparsec.ByteString
import Data.ByteString (ByteString)
import Data.List
import Data.Maybe
import Data.String.Conversions
import Data.Time.Calendar
import GHC.Generics
import Lucid
import Network.HTTP.Media ((//), (/:))
import Network.Wai
import Network.Wai.Handler.Warp
import Servant
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

```
type UserAPI =
       "users" :> Get '[JSON] [User]
  :<|>
       "posts" :> Get '[JSON] [Post]

server :: Server UserAPI
server = return users :<|> return posts

userAPI :: Proxy UserAPI
userAPI = Proxy

app :: Application
app = serve userAPI server

main = run 8080 app
```

How do we query an API?

```
users :<|> posts = client userAPI
```

Usage:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "user-database.lu" 8080 "")
  )
  :: IO [User]
```

How do we query an API?

```
users :<|> posts = client userAPI
```

Usage:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "user-database.lu" 8080 "")
  )
  :: IO [User]
```

How do we query an API?

```
users :<|> posts = client userAPI
```

Usage:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "user-database.lu" 8080 "")
  )
   :: IO [User]
```

How do we query an API?

```
users :<|> posts = client userAPI
```

Usage:

```
runClientM users
  (mkClientEnv
    (newManager defaultManagerSettings)
    (BaseURL Http "user-database.lu" 8080 "")
  )
  :: IO [User]
```

Javascript API client for jQuery:

```
apiJS :: Text
apiJS = jsForAPI userAPI jquery
```

API that documents a given API:

```
apiDocs :: API
apiDocs = docs userAPI
```

Front-end

The traditional web front-end language is Javascript, and we're probably not going to change that anytime soon.

Front-end

The traditional web front-end language is Javascript, and we're probably not going to change that anytime soon.

Haskell can be translated to Javascript!

- GHCJS (`https://github.com/ghcjs/ghcjs`, discontinued)
- Recent GHCs support compilation to WASM
- There are several useful libraries, such as Reflex-DOM.

Less-brutal solutions

- (Coffee/Type)Script
- PureScript
- Elm

The languages are sometimes kinda depthless, but often do the job much better than Javascript.

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view}

type Msg = Increment | Decrement
```

```
update msg model =
  case msg of
    Increment →
      model + 1

    Decrement →
      model – 1

view model =
  div [ ]
    [button [onClick · (Decrement)] [text "-"]
    ,div [ ] [text (String.fromInt model)]
    ,button [onClick · (Increment)] [text "+"]
    ]
```

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view}

type Msg = Increment | Decrement
```

```
update msg model =
  case msg of
    Increment →
      model + 1

    Decrement →
      model – 1

view model =
  div [ ]
    [button [onClick · (Decrement)] [text "-"]
    ,div [ ] [text (String.fromInt model)]
    ,button [onClick · (Increment)] [text "+"]
    ]
```

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view}

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment →
      model + 1

    Decrement →
      model – 1

view model =
  div [ ]
    [button [onClick · (Decrement)] [text "-"]
    ,div [ ] [text (String.fromInt model)]
    ,button [onClick · (Increment)] [text "+"]
    ]
```

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox {
    init = 0,
    update = update,
    view = view}

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment →
      model + 1

    Decrement →
      model – 1

view model =
  div [ ]
    [button [onClick · (Decrement)] [text "-"]
    ,div [ ] [text (String.fromInt model)]
    ,button [onClick · (Increment)] [text "+"]
    ]
```

Elm playground

https://ellie-app.com/

A short sneaky peek at FRP and Reflex-DOM

Functional Reactive Programming is an interesting way to describe dynamically changing objects. User interfaces are a natural field of use for FRP.

A short sneaky peek at FRP and Reflex-DOM

Functional Reactive Programming is an interesting way to describe dynamically changing objects. User interfaces are a natural field of use for FRP.

- Traditional:
  - *getMousePos*
  - *setWidgetPosition* (*mousePos*)
- Event-based: *onMousePosChange* (*setWidgetPosition*)
- Reactive: *widget* {*position = mousePos*}

FRP models the program data as Behavior, which is an abstraction of the development of the data value through the whole runtime of the program.

FRP models the program data as Behavior, which is an abstraction of the development of the data value through the whole runtime of the program.

That obviously can't be done. We'll say that this problem is an 'implementation detail'.

FRP demo

```
{-# LANGUAGE OverloadedStrings #-}
import Reflex.Dom

main = mainWidget $ el "div" $ do
  t ← inputElement "default1"
  t2 ← inputElement "default2"
  dynText $ _inputElement_value t
              ◇ ", " ◇
            _inputElement_value t2
```

(the code is simplified)

```
data InputElement er d t
   = InputElement
   {_inputElement_value :: Dynamic t Text
   , _inputElement_checked :: Dynamic t Bool
   , _inputElement_checkedChange :: Event t Bool
   , _inputElement_input :: Event t Text
   , _inputElement_hasFocus :: Dynamic t Bool
   , _inputElement_element :: Element er d t
   , _inputElement_raw :: RawInputElement d
   , _inputElement_files :: Dynamic t [RawFile d]
   }
```

```
data InputElement er d t
   = InputElement
   {_inputElement_value :: Dynamic t Text
   , _inputElement_checked :: Dynamic t Bool
   , _inputElement_checkedChange :: Event t Bool
   , _inputElement_input :: Event t Text
   , _inputElement_hasFocus :: Dynamic t Bool
   , _inputElement_element :: Element er d t
   , _inputElement_raw :: RawInputElement d
   , _inputElement_files :: Dynamic t [RawFile d]
   }
```

There's certainly a bright future waiting for some more refined variant of FRP.

GHC can now compile to WASM

WASM GHC Back-end:

+ The whole STG-style evaluation atop WASM
+ Fast
+ Works in most browsers
– Still no direct DOM access
  (that's a general WASM issue)
– Still no good frontend toolkits
  (it's too new)

Example from MiniHM (2025 version)

```
import { WASI } from "https://cdn.jsdelivr.net/npm/@runno/wasi@0.7.0/dist/wasi.js";
import ghc_wasm_jsffi from "./ghc_wasm_jsffi.js";

const wasi = new WASI({ stdout: (out) => console.log("[wasm stdout]", out) });

const jsffiExports = {};
const wasm = await WebAssembly.instantiateStreaming(
  fetch("./minihm.wasm"),
  Object.assign(
    { ghc_wasm_jsffi: ghc_wasm_jsffi(jsffiExports) },
    wasi.getImportObject()
  )
);
Object.assign(jsffiExports, wasm.instance.exports);

wasi.initialize(wasm, { ghc_wasm_jsffi: ghc_wasm_jsffi(jsffiExports) });

export const minihm = wasi.instance.exports.minihm;
```

Example from MiniHM (Haskell glue)

```
import MiniHM
import GHC.Wasm.Prim

foreign export javascript "minihm" minihm :: JSString → JSString

minihm = toJSString . go . fromJSString
  where
    go :: String → String
    go s =
      unlines . execWriter $ do
        x ← runExceptT $ inferType s putLn throwE
        case x of
          Right _ → pure ()
          Left err → tell ["*** Caught exception!", err]

main :: IO ()
main = ⊥
```

The rest of the WASM glue

- How to do stuff in the browser?

```
foreign import javascript unsafe "console.log($1)"
  js_print :: JSString → IO ()

foreign import javascript unsafe "typeof $1 === 'object'"
  js_is_obj :: JSVal → Bool
```

  ...same for `getElementById` and pals.

- What data can I get through to WASM?

```
newtype JSString = JSString JSVal
newtype JSException = JSException JSVal
data JSVal = JSVal JSVal #
newtype JSVal# = JSVal # (Any :: UnliftedType)
```

  ...just a pointer to JavaScript heap. All data access is IO.

JSVal (documentation exerpt)

A `JSVal` is a first-class Haskell value on the Haskell heap that represents a JavaScript value. You can use `JSVal` or its **newtype** as a supported argument or result type in JSFFI import & export declarations, in addition to those lifted FFI types like `Int` or `Ptr` that's already supported by C FFI. It is garbage collected by the GHC RTS:

- There can be different `JSVal`s that point to the same JavaScript value. As long as there's at least one `JSVal` still alive on the Haskell heap, that JavaScript value will still be alive on the JavaScript heap.
- If there's no longer any live `JSVal` that points to the JavaScript value, after Haskell garbage collection, the JavaScript runtime will be able to eventually garbage collect that JavaScript value as well.

Note that even the `FinalizationRegistry` logic can't break cyclic references between the Haskell/JavaScript heap: when an exported Haskell function closure retains a `JSVal` that represents a JavaScript callback. Though this can be solved by explicit *`freeJSVal`* calls.