Haskell for Web Developers

The perpetual myth persists that Haskell cannot be used for “real world applications”. Normally real world is usually left undefined in such a discussion, but can often be taken to mean that Haskell is not suited for database and web development work.

Haskell has a rich library ecosystem and is well-suited for these tasks but I concede that there might be a systemic lack of introductory material for many domain specific tasks. Something that many projects and companies are trying to remedy.

Haskell does indeed have several great web frameworks along the lines of RoR, Django, Flask, Pyramid etc.

1. Yesod
2. Snap
3. Happstack

I will not discuss these though because I really couldn’t give a better introduction than their own documentation. Instead I will focus on simple motivating examples for smaller libraries which provide a rich feature base for web development tasks while leveraging the strengths of Haskell language itself, and many of which can integrate with the larger frameworks.

Clay

Clay is a library for programmatic generation of CSS. Is it an embedded DSL (EDSL) that exposes selectors and styles for the CSS3 grammmer. Clay is designed to layer logic on top of the CSS as to encode variables, color mixing, complex selector logic and nested rules more easily than with base CSS. Clay can also be usefull as a lower-level combinator library to describe complex CSS layouts.

```
$ cabal install clay
```

```
{-# LANGUAGE OverloadedStrings #-}

import Clay
import Data.Text
import Prelude hiding (div)

bodyStyle :: Css
bodyStyle = body ? do
  background  aquamarine
  fontFamily  ["Helvetica Neue"] [sansSerif]

codeStyle :: Css
codeStyle = code ?
  do fontFamily  ["Monaco", "Inconsolata"] [monospace]
     fontSize    (px 14)
     lineHeight  (ex 1.8)

emphasis :: Css
emphasis = do
  fontWeight     bold
  color          black
  textTransform  uppercase

container :: Selector
```

```
container = div # ".code"

containerStyle :: Css
containerStyle = container ?
  do width (px 800)
     borderColor gray

main :: IO ()
main = putCss $ do
  bodyStyle
  codeStyle
  containerStyle
```

The above will generate the following stylesheet:

```
body
{
  background  : rgb(127,255,212);
  font-family : "Helvetica Neue", sans-serif;
}

code
{
  font-family : "Monaco","Inconsolata", monospace;
  font-size   : 14px;
  line-height : 1.80000ex;
}

div.code
{
  width        : 800px;
  border-color : rgb(128,128,128);
}
```

Blaze

```
$ cabal install blaze-html
```

Blaze is the bread and butter of markup generation in Haskell. It is described as a “blazingly fast HTML combinator library” which programmtically generates HTML and several other markup languages from an embedded DSL.

In this module, the language extension `OverloadedStrings` is used so that the type inferencer can infer common coercions between String-like types without having to do explicit calls to boilerplate functions (`pack`, `unpack`, `html`) for each string-like literal. This will be pretty common use for all the examples from here out that use `ByteString` or HTML.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Monoid
import Text.Blaze (ToMarkup(..))
import Text.Blaze.Html5 hiding (html, param)
import Text.Blaze.Html.Renderer.Text (renderHtml)
```

```haskell
import qualified Text.Blaze.Html5 as H

gen :: Html -> [Html] -> Html
gen title elts  = H.html $ do
  H.head $
    H.title title
  H.body $
    H.ul $ mapM_ H.li elts

main :: IO ()
main = do
  print $ renderHtml $ gen "My Blog" ["foo", "bar", "fizz"]
```

This would output HTML like the following:

```html
<html>
   <head>
      <title>My Blog</title>
   </head>
   <body>
      <ul>
          <li>foo</li>
          <li>bar</li>
          <li>fizz</li>
      </ul>
   </body>
</html>
```

In addition to generating HTML we can also derive from it’s internal ToMarkup classes to provide HTML representations for any datatype in Haskell. A silly example might be:

```haskell
data Example = A | B deriving (Show)
data List a = Cons a | Nil deriving (Show)

instance ToMarkup Animal where
  toMarkup = toHtml . show

instance (ToMarkup a) => ToMarkup (List a) where
  toMarkup x = case x of
      Cons a -> H.ul $ H.li $ toHtml a
      Nil -> ""
```

```html
<!-- Cons (Cons (Cons A)) -->
<ul>
   <li>
      <ul>
         <li>
            <ul>
               <li>A</li>
            </ul>
         </li>
      </ul>
   </li>
</ul>
```

It is worth noting that the Blaze builder overloads do-notation as some EDSLs do, but the `Html` type is not a monad. It is functionally a monoid.

For non-embedded template languages along the lines of *Jinja* or *erb* refer to the Shakespearean templates or heist.

JMacro

```
$ cabal install jmacro
```

JMacro is quasiquoter for Javascript code generation. The underlying implementation is rather clever and allows Haskell and Javascript to share functions and values across quotation boundaries. The end result is a fusion of Haskell and JavaScript that serves as a foundation to higher abstractions and as a very convienant way to implement code generation for compilers targetting Javascript.

As an example of we'll use JMacro to implement a simple translator for the untyped typed lambda calculus, something one might do if writing a language that transpiles to Javascript.

```
{-# LANGUAGE QuasiQuotes, TypeSynonymInstances, FlexibleInstances, OverloadedStrings #-}

import Data.String
import Language.Javascript.JMacro

jref :: Sym -> JExpr
jref = ValExpr . JVar . StrI

jvar :: Sym -> JStat
jvar sym = DeclStat (StrI sym) Nothing

jprint x = [jmacroE|console.log(x)|]

instance IsString Expr where
  fromString x = Var (Sym x)

data Val = Sym Sym
         | Lit Lit
         deriving (Show)

type Sym = String

data Lit = LStr String
         | LInt Int
         deriving (Show)

data Expr = App Expr Expr
          | Lam Sym Expr
          | Var Val
          deriving (Show)

-- Convert Haskell expressions to Javascript expressions

instance ToJExpr Val where
  toJExpr (Sym s) = toJExpr s
  toJExpr (Lit l) = toJExpr l
```

```haskell
instance ToJExpr Lit where
  toJExpr = toJExpr

instance ToJExpr Sym where
  toJExpr = jref

instance ToJExpr Expr where
  toJExpr (Lam s ex) =
      [jmacroE|
          function(arg) {
              `(jvar s)`;
              `(jref s)` = `(arg)`;
              return `(ex)`;
          }
      |]

  toJExpr (App f x) =
      [jmacroE| `(f)`(`(x)`) |]

  toJExpr (Var v) =
      toJExpr v

compile :: ToJExpr a => a -> String
compile = show . renderJs . toJExpr

s, k, i0, i1 :: Expr
s = Lam "f" $ Lam "g" $ Lam "x" $ (App "f" "x") `App` (App "g" "x")
k = Lam "x" $ Lam "y" "x"

i0 = Lam "x" "x"
i1 = App (App s k) k

main :: IO ()
main = do
  putStrLn $ compile s
  putStrLn $ compile k
  putStrLn $ compile i0
  putStrLn $ compile i1
```

Fay

```
$ cabal install fay
```

Fay is a growing ecosystem of packages that can compile Haskell to Javascript. Fay works with a strict subset of Haskell that preserves Haskell semantics such as currying and laziness. In addition to the core language, ther are interfaces for jquery and DOM manipulation so that Fay-compiled Haskell code can effectively access the browser internals.

```
$ cabal install fay-dom fay-jquery
```

The code generation is rather verbose given that it compiles quite a bit of the Haskell Prelude. The below example is very simple and only the interesting part of the outputted source is shown below. Notably the generated code is very readable.

```
-- demo.hs
import FFI
import Prelude
import JQuery

puts :: String -> Fay ()
puts = ffi "console.log(%1)"

example = take 25 [1..]

main :: Fay ()
main = ready $ do
    el <- select "#mydiv"
    setCss "background-color" "red" el

    puts "Hello World!"
    puts (show [1,2,3])
```

To compile invoke the compiler:

```
$ fay demo.hs --package fay-jquery
```

Some of the generated code:

```
var Prelude$enumFrom = function ($p1) {
    return new Fay$$$(function () {
        var i = $p1;
        return Fay$$_(Fay$$_(
            Fay$$cons)(i))(Fay$$_(
            Prelude$enumFrom)(
            Fay$$_(Fay$$_(
                Fay$$add)(i))(1)
        ));
    });
};

var Main$example = new Fay$$$(
    function () {
        return Fay$$_(Fay$$_(
            Prelude$take)(25))(
            Prelude$enumFrom(1));
    });
```

To call this code from vanilla Javascript:

```
var main = new Main();
main._(main.Main$main);
```

Fay is part of a larger community of compilers that transpile functional languages to Javascript. Another library of note is Roy. Although not Haskell, it has a sophisticated type system and notably an implementation of typeclasses, a feature that Fay currently does not implement.

Aeson

```
$ cabal install aeson
```

Aeson is the de-facto JSON parsing and generation library for Haskell. It's usage couldn't be simpler, we simply declare instance of `toJSON` and `fromJSON` for our types and Aeson takes care of the mappings and exception handling. By using `DeriveGeneric` we can create instances with very little code.

```
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}

import Data.Aeson

import GHC.Generics
import Network.HTTP
import Control.Applicative
import Data.ByteString (ByteString)

data Message = Message {
      text :: ByteString
    , date :: ByteString
    } deriving (Show, Generic)

instance FromJSON Message
instance ToJSON Message

fromStdin :: IO (Either String Message)
fromStdin = eitherDecode <$> readLn
```

postgres-simple

```
$ cabal install postgres-simple
```

Postgres-simple is a library for communicating with Postgres databases and mapping data between Haskell and SQL types. Although not an ORM, `postgres-simple` lets us generate and execute SQL queries and map result sets onto our algebraic datatypes very simply by deriving instances to declare schemas.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text
import Control.Applicative
import Control.Monad
import Database.PostgreSQL.Simple
import Database.PostgreSQL.Simple.FromRow

import qualified Data.ByteString as B
import qualified Database.PostgreSQL.Simple as Pg
```

```haskell
data Client = Client { firstName :: Text
                     , lastName :: Text
                     , clientLocation :: Location
                     }

data Location = Location { address :: Text
                         , location :: Text
                         }

instance Pg.FromRow Location where
  fromRow = Location <$> field <*> field

instance Pg.FromRow Client where
  fromRow = Client <$> field <*> field <*> liftM2 Location field field

queryClients :: Connection -> IO [Client]
queryClients c = query_ c "SELECT firstname, lastname, location FROM clients"

main :: IO [Client]
main = do
  uri <- B.getLine
  conn <- connectPostgreSQL uri
  queryClients conn
```

Acid-State

```
$ cabal install acid-state
```

We can further exploit Haskell's algebraic datatypes to give us a storage engine simply from the specification of our types and a little bit of `TemplateHaskell` usage. For instance, a simple `Map` container from `Data.Map` can be transformed to a disk-backed disk backed key-value store which can be interacted with as if it were a normal Haksell data structure.

```haskell
type Key = String
type Value = String
data Database = Database !(Map.Map Key Value)
```

Like it's name implies `acid-state` provides transactional guarantees for storage. Specifically that writes will be applied completely or not at all and that data will be consistent during reads.

```haskell
{-# LANGUAGE OverloadedStrings, TypeFamilies, DeriveDataTypeable, TemplateHaskell #-}

import Data.Acid
import Data.Typeable
import Data.SafeCopy
import Control.Monad.Reader (ask)

import qualified Data.Map as Map
import qualified Control.Monad.State as S

type Key = String
type Value = String
```

```haskell
data Database = Database !(Map.Map Key Value)
    deriving (Show, Ord, Eq, Typeable)

$(deriveSafeCopy 0 'base ''Database)

insertKey :: Key -> Value -> Update Database ()
insertKey key value
    = do Database m <- S.get
         S.put (Database (Map.insert key value m))

lookupKey :: Key -> Query Database (Maybe Value)
lookupKey key
    = do Database m <- ask
         return (Map.lookup key m)

deleteKey :: Key -> Update Database ()
deleteKey key
    = do Database m <- S.get
         S.put (Database (Map.delete key m))

allKeys :: Int -> Query Database [(Key, Value)]
allKeys limit
    = do Database m <- ask
         return $ take limit (Map.toList m)

$(makeAcidic ''Database ['insertKey, 'lookupKey, 'allKeys, 'deleteKey])

fixtures :: Map.Map String String
fixtures = Map.empty

test ::  Key -> Value -> IO ()
test key val = do
    database <- openLocalStateFrom "db/" (Database fixtures)
    result <- update database (InsertKey key val)
    result <- query database (AllKeys 10)
    print result
```

Digestive Functors

```
$ cabal install digestive-functors digestive-functors-blaze
```

Digestive functors solve the very mundane but mechanical task of validating forms. The library provides a way to specify views and validation logic and handle the control flow of validation between the end-user and the server. There are several backends to render the form and handle request/response cycles depending on your choice of framework For arbitrary reasons we'll choose Happstack for this example.

```
$ cabal install digestive-functors-happstack
```

We'll build a simple signup page with username and email validation logic.


```haskell
{-# LANGUAGE OverloadedStrings #-}

import Data.Maybe
import Text.Printf
import Control.Applicative
import Data.Text (Text, find, splitOn)

import Text.Digestive
import Text.Digestive.Happstack
import Text.Digestive.Blaze.Html5
import qualified Text.Blaze.Html5 as H

import qualified Happstack.Server as HS

data User = User
    { userName :: Text
    , userMail :: Text
    } deriving (Show)

userForm :: Monad m => Form Text m User
userForm = User
  <$> "name" .: check "Name must be two words" checkName (text Nothing)
  <*> "email" .: check "Not a valid email address" checkEmail (text Nothing)

checkEmail :: Text -> Bool
checkEmail = isJust . find (== '@')

checkName :: Text -> Bool
checkName s = length (splitOn " " s) == 2

signupView :: View H.Html -> H.Html
signupView view = form view "/" $ do

    label     "name" view "Full Name:"
    inputText "name" view
    H.br

    label     "email" view "Email:"
    inputText "email" view
    H.br

    childErrorList "" view

    inputSubmit "Signup"

template :: H.Html -> H.Html
template body = H.docTypeHtml $ do
    H.head $ H.title "Example form:"
    H.body body

reply m = HS.ok $ HS.toResponse $ template m
```

```haskell
page :: HS.ServerPart HS.Response
page = do
  HS.decodeBody $ HS.defaultBodyPolicy "/tmp/" 0 40960 40960
  r <- runForm "test" userForm
  case r of

      (view, Nothing) -> do
          let view' = fmap H.toHtml view
          reply $ form view' "/" (signupView view')

      (_, Just response) ->
          reply $ do
            H.h1 "Form is valid."
            H.p $ H.toHtml $ show response

config :: HS.Conf
config = HS.nullConf { HS.port = 5000 }

main :: IO ()
main = do
  printf "Listening on port %d\n" (HS.port config)
  HS.simpleHTTP config page
```

Servers

A great deal of effort has been put into making the Haskell runtime implement efficient event driven programming such that applications can take advantage of the Haskell threading support.

A simple single-threaded Hello World might be written like the following:

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Network
import Data.ByteString.Char8
import System.IO (hClose, hSetBuffering, Handle, BufferMode(LineBuffering))

msg = "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n\r\nHello World!\r\n"

handleClient :: Handle -> IO ()
handleClient handle = do
  hSetBuffering handle LineBuffering
  hGetLine handle
  hPutStrLn handle msg
  hClose handle

listenLoop :: Socket -> IO ()
listenLoop asock = do
  (handle, _, _) <- accept asock
  handleClient handle
  listenLoop asock

main ::  IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 5000
  listenLoop sock
```

To make this concurrent we use the function `forkIO` which utilizes the event-driven IO manager in GHC's runtime system to spawn lightweight user threads which are distributed across multiple system threads. When compiled with `-threaded` the Haskell standard library also will use non-blocking system calls which are scheduled by the IO manager (with `epoll()` under the hood) and can transparently switch to threaded scheduling for other blocking operations.

```
{-# LANGUAGE OverloadedStrings #-}

import Network
import Control.Monad
import Control.Concurrent
import Data.ByteString.Char8
import GHC.Conc (numCapabilities)
import System.IO (hClose, hSetBuffering, Handle, BufferMode(LineBuffering))

numCores = numCapabilities - 1

msg = "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n\r\nHello World!\r\n"

handleClient :: Handle -> IO ()
handleClient handle = do
  hSetBuffering handle LineBuffering
  hGetLine handle
  hPutStrLn handle msg
  hClose handle

listenLoop :: Socket -> IO ()
listenLoop asock = do
  (handle, _, _) <- accept asock
  forkIO (handleClient handle)
  listenLoop asock

main ::  IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 5000
  forM_ [0..numCores] $ \n ->
    forkOn n (listenLoop sock)
  threadDelay maxBound
```

This example is admittedly very simple but does illustrate that we can switch from serial to concurrent code in Haskell while still preserving sequential logic. Notably this server isn't really doing anything terribly clever to get performance, it's simply just spawning threads and all the heavy lifting is handled by the RTS. Yet with only a three line change the server can utilize all available cores.

Compiling with `-O2` and running with `+RTS -N4 -qm -qa` I get the following numbers on my Intel Core i5:

```
Requests per second:    12446.25 [#/sec] (mean)
```

There are other Haskell servers which do much more clever things such as the Warp server.

Websockets

The Warp server can utilize the async event notification system to implement asynchronous applications using `Control.Concurrent` primitives. The prime example is so called "realtime web programming" using websockets. In this example we'll implement a chat room with a mutable MVar which synchronizes messages across all threads in the server and broadcasts messages to the clients.

```
$ cabal install wai-websockets
```

```
{-# LANGUAGE OverloadedStrings #-}

import Control.Monad
import Text.Printf
import Data.Text (Text)
import Control.Concurrent
import Control.Monad.IO.Class (liftIO)

import Data.Aeson
import qualified Data.Text as T
import qualified Data.Text.IO as T

import qualified Network.Wai
import qualified Network.WebSockets as WS
import qualified Network.Wai.Handler.Warp as Warp
import qualified Network.Wai.Handler.WebSockets as WaiWS
import Network.Wai.Application.Static (defaultFileServerSettings, staticApp)

type Msg = Text
type Room = [Client]
type Client = (Text, WS.Sink WS.Hybi00)

broadcast :: Msg -> Room -> IO ()
broadcast message clients = do
  T.putStrLn message
  forM_ clients $ \(_, sock) -> WS.sendSink sock $ WS.textData message

app :: MVar Room -> WS.Request -> WS.WebSockets WS.Hybi00 ()
app state req = do
  WS.acceptRequest req
  sock <- WS.getSink
  msg <- WS.receiveData
  userHandler msg sock

  where
    userHandler msg sock = do
      let client = (msg, sock)
      liftIO $ T.putStrLn msg
      liftIO $ modifyMVar_ state $ \s -> do
          let s’ = client : s
          WS.sendSink sock $ WS.textData $
              encode $ map fst s
          return s’
      userLoop state client

userLoop :: WS.Protocol p => MVar Room -> Client -> WS.WebSockets p ()
userLoop state client = forever $ do
  msg <- WS.receiveData
  liftIO $ readMVar state >>= broadcast (T.concat [fst client, ” : ”, msg])

staticContent :: Network.Wai.Application
```

```haskell
staticContent = staticApp $ defaultFileServerSettings "."

config :: Int -> MVar Room -> Warp.Settings
config port state = Warp.defaultSettings
    { Warp.settingsPort = port
    , Warp.settingsIntercept = WaiWS.intercept (app state)
    }

port :: Int
port = 5000

main :: IO ()
main = do
  state <- newMVar []
  printf "Starting server on port %d\n" port
  Warp.runSettings (config 5000 state) staticContent
```

In the browser we can connet to our server using Javascript:

```javascript
ws = new WebSocket('ws://localhost:5000')

ws.onmessage(function(msg){console.log(msg)});
ws.send('User271828')
ws.send('My message!')
```

Cloud Haskell

```
$ cabal install distributed-process distributed-process-simplelocalnet
```

One of the most exciting projects in Haskell is a collections of projects developed under the Cloud Haskell metaproject. Cloud haskell brings language integrated messaging passing capability to Haskell under a very simple API which provides a foundation to build all sorts of distributed computations on top of simple actor primitives.

The core mechanism of action is a `Process` monad which encapsulates a actor-like computation that can exchange messages across an abstract network backend. On top of this the `distributed-process` library provides the language-integrated ability to send arbitrary Haskell functions back and forth between processes much like one can move code in Erlang, but while still persiving Haskell type-safety across the message layer. The signatures for the messaging functions are:

```haskell
send :: Serializable a => ProcessId -> a -> Process ()
expect :: forall a. Serializable a => Process a
```

The network backend is an abstract protocol that specific libraries (i.e. `distributed-process-simplelocalnet`) can implement to provide the transport layer indepenent of the rest of the stack. Many other protocols like TCP, IPC, and ZeroMQ can be used for the network transport.

The simplest possible example is a simple ping and pong between between several `Process`. Notably we don't encode any mechanism for binary serialization of code or data since Haskell can derive these for us.

```haskell
{-# LANGUAGE TemplateHaskell, DeriveDataTypeable, DeriveGeneric, GeneralizedNewtypeDeriving #-}
```

```
import Text.Printf
import Data.Binary
import Data.Typeable
import Control.Monad
import System.Environment (getArgs)
import Control.Concurrent (threadDelay)

import Control.Distributed.Process
import Control.Distributed.Process.Closure
import Control.Distributed.Process.Backend.SimpleLocalnet
import Control.Distributed.Process.Node (initRemoteTable,)

newtype Message = Ping ProcessId deriving (Eq, Ord, Binary, Typeable)

pingLoop :: Process ()
pingLoop = do
  liftIO $ putStrLn "Connected with master node."
  forever $ do
    (Ping remote_pid) <- expect
    say $ printf "Ping from %s" (show remote_pid)

    local_pid <- getSelfPid
    send remote_pid (Ping local_pid)

    liftIO $ putStrLn "Pong!"

remotable [ 'pingLoop ]

master :: [NodeId] -> Process ()
master peers = do
  pids <- forM peers $ \nid -> do
      say $ printf "Executing remote function on %s" (show nid)
      spawn nid $(mkStaticClosure 'pingLoop)

  local_pid <- getSelfPid

  forever $ do
    forM_ pids $ \pid -> do
      say $ printf "Pinging remote node %s" (show pid)
      send pid (Ping local_pid)

    forM_ pids $ \_ -> do
      (Ping pid) <- expect
      say $ printf "Received pong from %s"  (show pid)

    liftIO $ threadDelay 1000000

main :: IO ()
main = do
  args <- getArgs

  let host = "localhost"
  let rtable = Main.__remoteTable initRemoteTable

  case args of
    ["master", port] -> do
      printf "Starting master on %s:%s\n" host port
```

```haskell
      ctx <- initializeBackend host port rtable
      startMaster ctx master

    ["worker", port] -> do
      printf "Starting client on %s:%s\n" host port

      ctx <- initializeBackend host port rtable
      startSlave ctx

    otherwise -> error "Invalid arguments: master|worker <port>"
```

We can then spawn any number of instances from the shell:

```
$ runhaskell cloud.hs worker 5001
$ runhaskell cloud.hs worker 5002
$ runhaskell cloud.hs master 5003
```

Conclusion

Hopefully you feel for what exists in the ecosystem and feel slightly more empowered to use the amazing tools we have.