Lecture 14’. A web server in Haskell

Functional Programming 2018/19

Alejandro Serrano

Dear student

Don’t worry, this is not part of the exam

Goals

Look some Haskell in action!

- Scotty as a web server
- Lucid to produce HTML
- STM to keep some state

Side-effect: learn what a **DSL** is

Route-based web frameworks

Your first Scotty app

```haskell
import Web.Scotty

hallo :: ScottyM ()
hallo =
  get "/hallo" $ do
    html $ "<h1>Hallo!</h1>"

main :: IO ()
main = scotty 8000 hallo
```

Your first Scotty app

```
hallo :: ScottyM ()
hallo =
  get "/hallo" $ do
    html $ "<h1>Hallo!</h1>"
```

`hallo` is a Scotty *application*

- Each application is made of several *routes*
- When the user points the browser into a route, the associated code is run
  - Inside a specific monad `ActionM`
  - You can query information from the request
  - The response is set by calling `html`

Your first Scotty app

```
main :: IO ()
main = scotty 8000 hallo
```

`hallo` is just a *description* of how the server behaves

The `scotty` function *executes* the web server

- In this case, in the port 8000
- Point your browser to `http://localhost:8000/hallo` to be greeted

Your first Scotty app, redux

```
hallo = do
  get "/hallo" $ do ...
  get "/hallo/:naam" $ do
    naam <- param "naam"
    html $ "<h1>Hallo, " <> naam <> "!</h1>"
```

A web server may respond to several routes

- Declared together within the `ScottyM` monad

`:naam` specifies a *parameter* to the route

- With `param` you get its value
- `/hallo/Alejandro` shows *Hallo, Alejandro!* as response

Route-based web frameworks

This approach to defining web services is used across in many languages

- Sinatra for Ruby (the original one)
- Flask for Python
- Scalatra for Scala
- Nancy for C#
- ...

RASAAS: Replicate A String As A Service

```
rasaas =
  get "/replicate/:n/:s" $ do
    n <- param "n"
    s <- param "s"
    let elt = "<li>" <> pack s <> "</li>"
        lst = mconcat (replicate n elt)
    html $ "<ul>" <> lst <> "</ul>"
```

1. This code is full of `Monoid` functions
   - Scotty uses `Text` instead of `String` for performance
   - But you use it with the same interface!
2. The `param` function is able to return
   - An integer in the call with `"n"`
   - A `Text` value in the call with `"s"`

Handling different type uniformly

How does `param` work?

Handling different type uniformly

How does `param` work?

A type class!

```
class Parsable a where
  parseParam :: Text -> Either Text a

param :: Parsable a => Text   -- ^ Param name
                    -> ActionM a
```

`Either` is like `Maybe`, but returns more failure information

```
data Maybe    a = Nothing | Just  a
data Either e a = Left e  | Right a
```

HTML as a DSL

Aaaargh!!!

```
rasaas =
  get "/replicate/:n/:s" $ do
    ...
    let elt = "<li>" <> pack s <> "</li>"
        lst = mconcat (replicate n elt)
    html $ "<ul>" <> lst <> "</ul>"
```

Producing HTML code by hand is both:

1. Error prone: what if you miss(type) a `</li>`?
2. Security issue: the user may send HTML fragments as parameter `s`, and you do not validated them
   - This is called **HTML injection**

Use Lucid to build HTML documents

```
import Lucid
import Control.Monad (replicateM_)

rasaas2 =
  get "/replicate/:n/:s" $ do
    n <- param "n"
    (s :: String) <- param "s"
    -- Build the document
    html $ renderText $ do
      html_ $ do
        head_ $ title_ "Replicate a string"
        body_ $
          ul_ $ replicateM_ n $
            li_ (toHtml s)
```

Using code to describe documents

```
html_ $ do
  head_ $ title_ "Replicate a string"
  body_ $
    ul_ $ replicateM_ n $
      li_ (toHtml s)
```

Lucid includes one function per HTML element

- Arguments become nested tags

`Html` is a *monad* for describing HTML documents

- `do` is used to put elements one after the other
- We can use monadic utilities as `replicateM_`

Domain-Specific Languages

HTML is a **Domain-Specific Language**, DSL for short

- As opposed to *general purpose languages*, they are only useful for some programming tasks
  - Less powerful but easier to optimize
- The goal is to allow more people than just trained programmers to use the DSL
  - More intuitive and declarative

Other examples: SQL for databases, Make for dependencies

Embedded DSLs

Lucid **embeds** HTML into Haskell

- The syntax is encoded inside that of a host language

Advantages:

- Escape hatch to the host language
- Reuse existing libraries, compilers, IDEs
- Easy to combine with other DSLs

Embedded DSLs

Lucid **embeds** HTML into Haskell

- The syntax is encoded inside that of a host language

Advantages:

- Escape hatch to the host language
- Reuse existing libraries, compilers, IDEs
- Easy to combine with other DSLs

In Haskell, **type safety**!

An ever-growing to-do list

Problems with mutability

We need to keep a list of to-do elements

- To be accessed by several concurrent users

Problems with mutability

We need to keep a list of to-do elements

- To be accessed by several concurrent users

`State` does not handle this scenario

- This requires a linear sequence of changes
- *Not* the case with concurrent use

Problems with mutability

We need to keep a list of to-do elements

- To be accessed by several concurrent users

`State` does not handle this scenario

- This requires a linear sequence of changes
- *Not* the case with concurrent use

A secret: **mutable** variables exist in Haskell

- They are called `IORef` and work on `IO` monad
- Means going back to locks, race conditions…

Software Transactional Memory

Variables which are guaranteed to work correctly in concurrent environments

- Changes are described together as a *transaction*
- A transaction runs completely and isolated from others
- And does with very low overhead!

```haskell
-- build transactions
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
modifyTVar :: TVar a -> (a -> a) -> STM ()
-- run a transaction
atomically :: STM a -> IO a
```

The to-do list app

```haskell
main = do  -- In IO monad
  lst <- newTVarIO []
  scotty 8000 (todo lst)

todo vr = do  -- In ScottyM monad
  get "/show" htmlLst
  get "/add/:thing" $ do  -- In ActionM monad
    (t :: String) <- param "thing"
    liftIO $ atomically $ modifyTVar vr (t :)
    htmlLst
  where
    htmlLst = do  -- In ActionM monad
      lst <- liftIO $ atomically $ readTVar vr
      html $ renderText $ do  -- In Html monad
        h3_ "Your to-do list"
        ul_ $ forM_ lst (li_ . toHtml)
```

The key line

```
todo vr = ...
  liftIO $ atomically $ modifyTVar vr (t :)
```

1. `modifyTVar` *describes* the modification we want to perform to our variable
   - In this case, add `t` to the front
2. `atomically` *executes* the transaction
   - This prevents collision from concurrent threads
3. `liftIO` is required to run an `IO` action within a route

The missing part: persistence

`STM` variables live in *memory*

- They are gone if the process is shut down or restarted
- This happens often in a real server

The missing part: persistence

`STM` variables live in *memory*

- They are gone if the process is shut down or restarted
- This happens often in a real server

You can *persist* the information in a database

- `acid-state` is a database for Haskell values
- `persistent` + `esqueleto` talk to relational databases
  - `persistent` take care of mapping rows to Haskell values
  - `esqueleto` embeds SQL within Haskell

Summary

Embedded DSLs can describe solutions to problems in a given domain in a concise and elegant way

- Common technique in the functional world
- Other methodologies such as Domain-Driven Design (DDD) stem from bringing DSLs into other paradigms

Haskell is a great language for DSLs

- Types reflect the domain terms and their invariants
- Use common abstractions: monoids, monads, …