Informe Trabajo Práctico Final
Análisis de Lenguajes de Programación

Lucio Bassani

November 1, 2019

Trabajo subido a lbassani12/devteam-eq @ github

1 Descripción de Módulos

Nos situamos en el directorio principal donde se encuentra el código fuente: **devteam-eq/src/**

1.1 TeamEquity.hs

Módulo encargado de encapsular los submódulos que componen la librería. Sólo expone las funciones que fueron expuestas por los submódulos principales:

- Github.hs
- TeamAffinity.hs
- Types.hs

1.2 TeamEquity/

1.2.1 Github.hs

Funciones expuestas:

```
setCredentials:: User -> Password -> Option Https
```

Función que oculta la implementación subyacente de la libreria **Network.Req**. Toma un usuario y una contraseña (ambos tienen tipos sinónimos a *String*) para devolver el tipo necesario para ser usado luego en la query a la API de GitHub. Esta signatura no está presente en el código fuente ya que el compilador nunca me dejó definir el sinónimo de tipos *type Credentials = Option Https*, entonces tuve que dejar que lo infiera. Esto me llevó a no poder dar las signaturas para varias funciones más.

```
setDate :: Day -> Day -> DateRange
```

Función que dado dos días devuelve un *DateRange* que representa el rango entre dos fechas. El tipo de dato *Day* es brindado por la librería *Data.Time.Calendar*.

```
setRepo :: RepoOwner -> RepoId -> Repository
```

Función que dado los datos del repositorio (Owner e ID) devuelve el record *Repository* que lo representa.

```
getRepoInteractions :: Repository -> DateRange -> Credentials -> IO (RepoInteractions)
```

Función que engloba todas las computaciones necesarias para conseguir la lista de *PullRequest* (a.k.a *RepoInteractions*) que son la base de nuestro análisis. Toma los datos proporcionados por las funciones de seteo y consigue todos los issues del repositorio en cuestión. Luego, por cada uno de ellos, se busca los pull requests asociados y los retorna encapsulados en la mónada *IO*.

Funciones complementarias:

```
getIssues :: Repository -> DateRange -> Credentials -> IO (Issues)
```

Función que toma los datos necesarios para realizar las queries a la API de GitHub. Son todos los proporcionados por las funciones anteriormente descriptas. Devuelve la lista de *Issue* (a.k.a *Issues*) encapsulada en la mónada *IO*. Es la encargada de llamar a la función auxiliar *getIssues'* para poder realizar los sucesivos llamados recursivos para manejar la paginación de los resultados que nos devuelve la API de GitHub. Cada página tiene como máximo treinta items y es necesario inspeccionar el header de la query para saber cual es la próxima página y la última. Ésta signatura no esta presente por la misma razón que *setCredentials*.

```
getIssues' :: Repository -> DateRange -> Credentials -> (String, String) -> IO (Issues)
```

Función auxiliar que, además de tomar los argumentos que toma *getIssues*, toma una tupla de *String*s que representa la página a consultar y la última página. La función consiste en configurar la query que se le va a hacer a la API de GitHub y luego usar la función *runReq* para desencapsular el resultado de ella. Luego la encapsulo en la mónada *IO* para no tener que llevar la mónada *Req a* a todos lados. En última instancia, se llama recursivamente con la nueva página a consultar. La lista de *Issue* son todos los issues que son pull requests. De ellos nos interesa su ID y el autor para futuras computaciones. Ésta signatura no esta presente por la misma razón que *setCredentials*.

```
getPullRequest Repository -> Credentials -> Issue -> IO(PullRequest)
```

Función que toma los datos del repositorio, las credenciales y un *Issue* para retornar el record *PullRequest*, que es nuestra representación interna de un pull request, envuelto en la mónada *IO*. La función, de la misma manera que *getIssues'*, desencapsula el resultado de la computación de la query y lo encapsula en la mónada *IO*. Ésta signatura no esta presente por la misma razón que *setCredentials*.

1.2.2 TeamAffinity.hs

Funciones expuestas:

```
generateTeamAffMap :: RepoInteractions -> TeamAff
```

Función que toma la lista de pull requests y genera el mapa de mapas con el cual vamos a representar las afinidades entre developers.

```
getPairAff :: TeamAff -> Developer -> Developer -> Int
```

Función que dado el mapa de mapas de afinidades y dos developers, retorna la afinidad entre ellos.

```
getTeamEquity :: TeamAff -> Float
```

Función que dado el mapa de mapas de afinidades calcula la equidad entre ellas, calculando el desvio estandar entre todas.

```
teamAffMapPP :: TeamAff -> IO ()
```

Función que imprime en pantalla el mapa de mapas de afinidades.

Funciones complementarias:

```
innerMap :: [Developer] -> DevAff
```

Función auxiliar que dada una lista de developers devuelve un mapa cuya clave son los usernames de esos developers y el valor es la afinidad 1.

```
stdDev :: [Float] -> Float
```

Función auxiliar que calcula la desviación estandar entre una lista de Floats.

1.2.3 Parser.hs

Funciones exportadas:

```
parseReviewers :: Value -> Either Aeson.Parser [String] String
```

Parser que toma un *Value* (el tipo base de la representación de JSON en la libreria Aeson) y devuelve una lista de usernames de developers o un mensaje de error en caso de fallar.

```
parseIssues :: Value -> Either Aeson.Parser Issues String
```

Parser que toma un *Value* y devuelve una lista de *Issue* o un mensaje de error en caso de fallar.

```
parseHeader :: String -> Either Parser (String, String) String
```

Parser que toma una string que representa el header Link de la respuesta de la API de GitHub y devuelve una tupla de strings que representan la siguiente página y la última. En caso de fallar devuelve un mensaje de error.

Funciones complementarias:

```
parseSingleItem :: Value -> Aeson.Parser Issue
```

Función que dado un *Value* parsea los campos que nos importan del issue en cuestión, estos son:

- number: es el ID del issue que nos va a servir para identificarlo como pull request luego al hacer una query al endopoint de pull requests de nuestro repositorio.
- user: es el campo que contiene otro JSON con la información del usuario que creó el issue.
- login: es el username del developer que creó el issue. Este es el autor del pull request que queremos buscar.

```
parseItems :: Value -> Aeson.Parser Issues
```

Parser que mapea *parseSingleItem* a todos los items del JSON que se le pasa como argumento para parsear una lista de issues.

```
parseSingleReviewer :: Value -> Aeson.Parser String
```

Función que dado un *Value* parsea el campo *login* y lo devuelve.

```
parseReqReviewers :: Value -> Aeson.Parser [String]
```

Parser que mapea *parseSingleReviewer* a todos los requested_reviewers del JSON que se le pasa como argumento para parsear una lista de strings que son los usernames de ellos.

```
parsePage :: String -> Parser (String, String)
```

Parser que dado una string que representa el tipo de página que se quiere parsear, parsea el número de página correspondiente y la string que se le pasó como arguemento, parseada.

```
parseNextPage, parseLastPage, parsePrevPage :: String -> Parser (String, String)
```

Funciones que parsean la siguiente página, la última página y la página previa.

```
parseNextAndLastPage :: Parser (String, String)
```

Parser que dado un header Link de la respuesta de la API de GitHub devuelve una tupla con las strings que representan el número de página siguiente y el de la última página. Usa el operador de opción porque, dependiendo de que página corresponda el header, puede responder de 3 formas distintas:

- sólo con la página siguiente y última;
- página siguiente, previa, última y primera;
- página primera y previa.

y es necesario estar preparados para parsear cualquiera sea la respuesta.

```
myParse :: Parser a -> String -> Either ParseError a
```

Función similar a *Parser.parse* pero que fija la string de comienzo de error a: "Error in"

1.2.4 Types.hs

Módulo encargado de dar las definiciones de tipos usado a lo largo del proyecto. Mayoritariamente son sinónimos de tipos ya que la mayoría de los tipos son *String* y no queda muy claro a la hora de dar signaturas que tomen n strings y devuelvan otras tantas. También estan definidas las excepciones utilizadas a la hora de controlar fallos en el módulo *GitHub*.

2 Decisiones de Diseño

Las decisiones más importantes a la hora de diagramar el proyecto son las de cómo representar los datos que queremos modelar. Por un lado, qué información tomar de la extensa lista de campos que tiene cada uno de los JSON que responde la API de Github.

Primero, al leer la documentación acerca de los issues como pull requests (1) decidí que la única información que necesitaba de los issues era su autor y el ID con el cual buscarlos como pull request en el endpoint de pull requests. Entonces definí issues de la siguiente forma:

```
type Developer = String
data Issue = Issue { i_author :: Developer
                   , number :: Int
} deriving(Show, Generic)
```

Una vez determinado todos los issues de un repositorio, debía decidir como manejar la información relevante de un pull request. Para mi análisis, solo necesitaba los usernames del autor de un pull request y la lista de usernames de los reviwers del mismo. Siguiendo esta idea, definí pull request de la siguiente forma:

```
data PullRequest = PR { author :: Developer
                      , reviewers :: [Developer]
} deriving (Eq, Show)
```

Con esto decidido, restaba sólo definir cómo vincular esta información.

Ya que el conocimiento de lo que se modifica en el código es mayor por parte del autor de los pull request que por los reviewers, que a veces no se detienen a leer el código en su totalidad ni lo han testeado exhaustivamente, imaginé que la relación de afinidad debería ser unidireccional, es decir, sólo se contabiliza la afinidad de un autor hacia un reviewer, y no desde el reviewer hacia el autor. Este pensamiento me llevó a diagramar la relación de afinidad como un mapa de mapas (2), donde las claves de ambos mapas sean los usernames de los developers y el valor del último, su afinidad:

```
type Affinity = Int
type DevAff = Map.Map Developer Affinity
type TeamAff = Map.Map Developer DevAff
```

ya que este módelo me iba a permitir buscar la afinidad entre dos developers de una manera muy sencilla, usando el operador *Map.!*. De la misma manera, crear el mapa de mapas de era una cuestión sencilla usando la funciones *Map.fromListWith y Map.unionWith*. Dado la simpleza y elegancia de ésta solución, me permitió avanzar rápidamente con el desarrollo del módulo TeamAffinity.hs.

Ahora necesitaba los datos para llenar mis estructuras. Debía hacer requests a la API de GitHub de manera sencilla y encontré la libreria *Network.Req* (3). Con ella podía hacer queries a la API y obtener JSONs como respuesta, los cuales podían ser facilmente parseados usando la libreria *Data.Aeson* (4).

Finalmente, una vez obtenido los resultados de algunas queries, necesitaba usar paginación para poder moverme por ellos (5) (6). Allí use recursión para resolver este problema: la función se iba a llamar a si misma retornando la concatenación de la lista resultante de ese llamado, con el resto.

References

[1] Searching issues and pull requests,

[2] containers: Maps, Sets, and more,

[3] req: Easy-to-use, type-safe, expandable, high-level HTTP client library

[4] Aeson: the tutorial

[5] Pagination in GitHub API v3

[6] Trasversing with Pagination

[7] Search issues and pull requests

[8] List pull requests

[9] Intro to Parsing with Parsec in Haskell

[10] The Haskell Tool Stack