Memòria pràctica 2B

DAT QT2019

Alba Mendez

8 de novembre de 2019

Part I.
Introducció

En aquesta segona part de la pràctica es preten consolidar coneixements més avançats de Haskell vists a teoria. Per fer-ho, es dona una aplicació web CGI feta en Haskell, que implementa una calculadora de números complexos. Es demana que l'estudiant completi alguns mètodes d'un dels mòduls.

També es dona un entorn de desenvolupament amb un script que compila i desplega l'aplicació, però m'he construit un en el meu portàtil per poder treballar en local. A banda d'alguns errors (en les noves versions de Haskell, `Monoid` és una subclasse de `Semigroup`) no hi ha hagut problemes significatius.

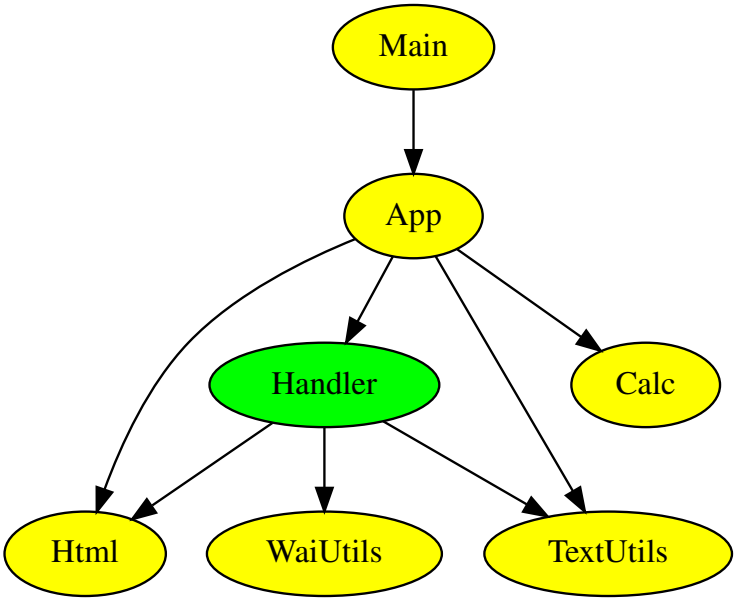

Figura 1: Els mòduls de l'aplicació i les dependències entre ells.

Part II.
Estructura de l'aplicació

Començarem comentant breument cadascun dels mòduls que conformen l'aplicació. El graf de dependències es pot veure a la figura 1, donada pel professor.

1. Mòduls d'utilitat

Primer presentarem aquells mòduls que no tenen rellevància específica a aquesta aplicació, i només serveixen per fer el codi més elegant o implementen accions molt genèriques.

TextUtils Aquest mòdul simplement conté implementacions de funcions típiques de conversió de/a text (`show`, `read`, `readEither` i `showsPrec`), però que accepten (o retornen) `Text` en comptes de `String`. S'han anomenat respectivament `showt`, `readt`, `readtEither` i `showtPrec`.

Html Aquest mòdul és molt simple i preten simplificar la generació d'HTML. Es defineix un writer monad, `HtmlM a`, i unes funcions que generen nodes HTML típics fent servir aquest monad, com ara el *doctype*, un element (amb els seus atributs), o text (escapat apropiadament).

WaiUtils Aquí s'implementen operacions típiques amb WAI:

- Obtenir els paràmetres de la petició.
- Descodificar les dades de la sessió (una llista de parells clau-valor textuals) a partir del contingut de la *cookie* corresponent, i fer la operació inversa.
- Generar una resposta HTTP sencera a partir del contingut HTML donat (text).
- Generar una resposta HTTP que redirigeixi a la URL donada.
- Generar una resposta HTTP d'error, amb el codi i missatges donats.

2. Mòduls d'aplicació

Main En primer lloc tenim aquest mòdul, que és el punt d'entrada de l'executable CGI. El seu codi és molt senzill, només defineix `main`:

```
main :: IO ()
main = do
    -- try :: Exception e => IO a -> IO (Either e a)
    r <- try $
        -- CGI adapter: run :: Application -> IO ()
        run calcApp
    case r of
        Right _ -> pure ()
        Left exc -> do
            -- Exception on initialization
            putStrLn "Status: 500 Internal Server Error"
            putStrLn "Content-Type: text/plain"
            putStrLn ""
            putStrLn "Exception on initialization (while excution of
              ↪ 'calcApp'): "
            putStrLn $ "    " ++ show (exc :: SomeException)
```

Veiem que s'executa `calcApp` mitjançant el mètode `run` de `Network.Wai.Handler.CGI`. En cas que hi hagi qualsevol error en l'execució, s'imprimeix una resposta 500 manual-

ment, que conté la descripció de l'excepció.

App La funció `calcApp` es defineix al mòdul **App**, que dona pas a la lògica principal de l'aplicació. Es defineixen i implementen les possibles operacions que es poden fer amb els nombres, es renderitza la pàgina HTML i es gestionen les accions que faci l'usuari.

Per aconseguir-ho, aquest codi fa servir principalment **Html** i els dos mòduls que s'expliquen a continuació, **Calc** i **Handler**.

Calc Aquest mòdul implementa la gestió de l'estat de la calculadora:

- Es defineix `CalcStack`, que representa l'estat de la calculadora (una pila de valors).
- Es defineix `CalcInstr`, que representa una operació amb aquest estat: ja sigui una manipulació simple de la pila (com afegir, treure o intercanviar valors) o una operació unària o binària genèrica amb els valors de dalt de la pila.
- Es defineix `solveCalc`, que aplica una operació de dalt (o serie d'operacions) a un estat concret, retornant el nou estat.

Cal indicar que aquest mòdul és genèric i no tracta directament amb nombres complexos. La pila pot contenir valors de qualsevol tipus, i no s'implementa cap operació unària o binaria concreta.

Handler Aquest és el mòdul que l'estudiant ha de completar. Es defineix un monad `Handler a`, que té les següents propietats:

- State monad: Manté un estat que es va passant entre acció i acció. Aquest estat emmagatzema principalment les dades de sessió (una llista de parells clau-valor, com s'explica a dalt).
- Reader monad: Permet operar amb la petició HTTP quan encara no la tenim.
- A més, encapsula una acció I/O que retorna el resultat de tipus `a`, i el nou estat.

Es tracta doncs d'un mòdul genèric que construeix una capa d'abstracció sobre la funcionalitat de `WaiUtils`, per fer encara més senzilla la gestió de les dades de la sessió i dels paràmetres de la petició.

En la secció següent s'entra en detall sobre aquest mòdul i la implementació de les funcions demanades.

Part III.
Solució de la pràctica

A continuació s'explicarà l'estructura del mòdul en més detall, i s'exposaran les solucions a les funcions que es demana implementar.

3. Implementació del monad

En primer lloc es defineix el monad com hem explicat en la secció anterior. Se'ns demana implementar `pure` (àlias de `return`) i la funció *bind* del monad.

Anem a la funció `pure`. Per definició, agafa un valor `x` i l'encapsula dins el monad. La implementació és fàcil, només cal cridar el constructor (`HandlerC`) passant una lambda que, donada una petició i un estat, retorna una acció I/O pura de `x` i aquest estat:

```
pure x = HandlerC $ \ _ st0 -> pure ( x, st0 )
```

La funció *bind* és una mica més complicada. Es crida el constructor amb una lambda com abans, però ara cal passar la petició i l'estat al primer monad, per obtenir una acció I/O. El resultat d'*aquesta* acció I/O es passa a `f` per obtenir un nou monad. Llavors cal cridar `runHandler` i passar-li la petició i el nou estat, per obtenir l'acció I/O final:

```
HandlerC hx >>= f = HandlerC $ \ req st0 -> do
    ( x, st1 ) <- hx req st0
    (runHandler $ f x) req st1
```

A continuació s'implementen funcions que fan servir el monad anterior per manipular les dades de la sessió, i també una per obtenir el mètode de la petició, que és la que implementarem a continuació.

4. Funció `getMethod`

La funció es defineix com `getMethod :: Handler Method`. Com el seu nom indica, simplement retorna el mètode de la petició. Per implementar-la primer busquem com extreure el mètode de la petició, i veiem que es fa amb la funció de WAI `requestMethod`.

Llavors només cal construir un monad amb aquest resultat, però no podem fer servir `pure` perquè la petició està encapsulada en el monad. De tota forma, la implementació

és molt similar a `pure`:

```
getMethod = HandlerC $ \ req st0 ->
    pure ( requestMethod req, st0 )
```

5. Funcions `getSession_` i `setSession_`

Finalment es demana implementar `getSession` i `setSession`, que donada una clau, emmagatzemen o retornen el valor que té associat en l'estat de la sessió:

```
getSession :: Read a => Text -> Handler (Maybe a)
setSession :: Show a => Text -> a -> Handler ()
```

Abans però, implementarem `getSession_` i `setSession_`. Aquestes son les versions «base» que accepten i retornen el valor en text, mentre que les versions sense `_` fan servir `showt` o `readt` per convertir el valor a text primer:

```
getSession_ :: Text -> Handler (Maybe Text)
setSession_ :: Text -> Text -> Handler ()
```

Comencem amb `getSession_`. La seva implementació és molt similar a la de `getMethod`, però aquí farem servir `hsSession` per extreure les dades de sessió de l'estat, i després la funció estàndard `lookup` per extreure el valor correponent a la clau donada:

```
getSession_ name = HandlerC $ \ req st0 ->
    pure ( lookup name $ hsSession st0, st0 )
```

Anem amb `setSession_`. Aquesta funció ens ve ja mig implementada; s'extreu la sessió de l'estat com hem fet abans i es reemplaça el valor en la clau donada:

```
setSession_ name value = HandlerC $ \ req st0 -> do
    let newsession = (name, value) : filter ((name /=) . fst)
      ↪  (hsSession st0)
    error "Handler.setSession_: A completar per l'estudiant"
```

L'únic que queda, doncs, és reemplaçar la sessió per `newsession` i retornar el nou estat. Això ho podem fer amb la funció `hsSetSession`. Per tant, el resultat és:

```
setSession_ name value = HandlerC $ \ req st0 -> do
    let newsession = (name, value) : filter ((name /=) . fst)
      ↪  (hsSession st0)
    pure ( (), hsSetSession newsession st0 )
```

6. Funcions `getSession` i `setSession`

Ara és fàcil implementar `getSession` i `setSession` fent servir aquestes funcions. La segona és fàcil, només cal cridar `setSession_` amb el valor convertit:

```
setSession name value = setSession_ name (showt value)
```

`getSession` és una mica més complex ja que `getSession_ name` retorna el resultat dins el monad; cal mapejar-lo. Però el monad no conté directament `Text` sinó `Maybe Text`; per tant no podem mapejar el valor amb `readt`, ho hem de fer amb `(>>= readt)`:

```
getSession name = (>>= readt) <$> getSession_ name
```