CI4251 - Programación Funcional Avanzada
Entrega Tarea 3

Brian-Marcial Mendoza
07-41206
<07-41206@usb.ve>

Junio 07, 2015

Eat all the empanadas!

Rafita prepara empanadas en una recóndita playa del oriente del país. La calidad de sus empanadas de cazón es legendaria, tanto que la contratan para servir los pasapalos de las fiestas playeras organizadas por la banda usual de desadaptados que allí pulula en cualquier puente o feriado largo.

Para esos eventos, Rafita se presenta con su gran paila, que tiene capacidad para freír m empanadas. Una vez fritas, echa *todas* las empanadas, simultáneamente, en un gran colador plástico de color indeterminado, raído por el tiempo, los elementos y el manoseo. Hecho esto, y consecuencia del calor por la paila y el inclemente sol, Rafita se sienta a tomar cerveza.

Cuando un parroquiano está hambriento, se acerca a buscar empanadas. Los parroquianos de la fiesta no hacen cola, sino que meten mano y se sirven del colador, tomando y comiendo **una** empanada a la vez, siempre y cuando haya empanadas disponibles. Cuando el parroquiano come, vuelve a tomar cerveza por un rato, hasta que tenga hambre nuevamente.

Si un parroquiano tiene hambre, pero no hay empanadas, le avisa a Rafita para que prepare más.

Se desea que Ud. construya una simulación de este comportamiento usando Haskell. En este sentido, tome en cuenta lo siguiente:

- Rafita es la única que prepara empanadas y siempre prepara *exactamente* m empanadas en lote. Naturalmente, Rafita será modelada con un hilo de ejecución.
- Rafita tarda un tiempo al azar en preparar las empanadas. A efectos de esta simulación, considere un tiempo al azar entre 3 y 5 segundos. La simulación debe indicar claramente `Rafita está cocinando` y pasado el intervalo `Rafita sirvió las empanadas`.
- En la fiesta puede haber un número arbitrario de parroquianos. Su simulación debe estar parametrizada para $n > 0$ parroquianos, y cada parroquiano debe estar representado por un hilo independiente.
- Los hábitos de bebida y comida de los parroquianos son muy variables, así que debe considerar que transcurre un tiempo al azar entre 1 y 7 segundos desde el momento en que empuña su empanada, la consume,

vuelve a su bebida, y tiene hambre de nuevo. La simulación debe indicar claramente `Parroquiano N come empanada` cuando comienza a comer y `Parroquiano N tiene hambre` cuando vuelve a buscar una empanada.

- Rafita tiene ingredientes infinitos para preparar las empanadas, y los parroquianos no tienen nada más productivo que hacer, de manera que una vez que comienza la fiesta, sólo termina cuando se interrumpe la simulación con `Ctrl-C`.

Presente **dos** soluciones para este problema: una empleando técnicas clásicas de concurrencia (sincronización con `MVar`) y otra empleando Memoria Transaccional (`STM`).
En ambos casos, cuando se interrumpa la simulación, presente un resultado sumario indicando:

```
Rafita preparó R empanadas.

Parroquiano 1:    P1
Parroquiano 2:    P2
...
Parroquiano N:    PN
Total:            T
```

Donde `R` es el total de empanadas, necesariamente múltiplo de m; `P1` hasta `PN` corresponden a la cantidad de empanadas que comió cada parroquiano, respectivamente, y `T` resulta de la suma de esos consumos.

Finalmente, para poder comprobar la fidelidad de su simulación es necesario que use números pseudo-aleatorios, como los que se proveen en `System.Random` *fuera* del monad `IO`, usando `randomSeed` como semilla.

Librerías Utilizadas

Para la resolución del problema se utilizaron las siguientes librerías.

```
{-# LANGUAGE ScopedTypeVariables #-}

import Control.Applicative (pure, (<$>))
import Control.Concurrent
import Control.Concurrent.STM
import Control.Exception
import Control.Monad (forever, replicateM)
import Data.Foldable (for_, traverse_)
import Data.Traversable (for, traverse)
import Data.Tuple (swap)
import System.Random
```

1. Solución Clásica

1.1. Funciones Auxiliares

Las funciones auxiliares utilizadas para la solución clásica del problema se presentan a continuación y serán explicados brevemente.

```
randomSeed :: Int
randomSeed = 42
```

Función que proveerá la semilla para el generador de números aleatorios.

```
printer :: Chan String -> IO ()
printer spool
  = do
    xs <- getChanContents spool
    mapM_ putStrLn xs
```

Función que será ejecutada en la solución clásica por el hilo encargado de imprimir en pantalla los mensajes de los hilos de la simulación. Recibe como entrada un canal y va leyendo perezosamente de dicho canal e imprimiendo cada mensaje encontrado en pantalla.

```
randomDelayC :: RandomGen a => Int -> Int -> MVar a -> IO ()
randomDelayC lo hi gen
  = do
    time <- modifyMVar gen
            $ pure . swap . randomR (lo, hi)
    threadDelay time
```

Función que será ejecutada en la solución clásica por los hilos encargados de simular el comportamiento de Rafita y los parroquianos. El invocarlo obtiene un tiempo aleatorio entre los valores lo y hi que son pasados como parámetros. El MVar que contiene el generador del próximo número aleatorio es actualizado y posteriormente se invoca la función threadDelay para simular los tiempos de espera.

```
rafaC :: RandomGen a =>
          MVar a -> MVar Int -> MVar Bool ->
            MVar Bool -> MVar Int -> Chan String ->
              Int -> IO b
rafaC gen empanadas needMore served total spool m
  = forever
  $ do
    _ <- takeMVar needMore
    writeChan spool "Rafita esta cocinando"
    randomDelayC (3 * 10^6)  (5 * 10^6) gen
    writeChan spool "Rafita sirvió las empanadas"
    putMVar served True
    putMVar empanadas (m-1)
    modifyMVar_ total (pure . (+m))
```

Función que será ejecutada en la solución clásica por el hilo encargado de simular el comportamiento de Rafita. Primero se queda esperando hasta que el MVar que actúa como un flag de que se necesitan más empanadas tenga un valor. Luego, escribe al canal para que el hilo de impresión escriba en pantalla que Rafita está cocinando. Después de esto se invoca la función `randomDelay` para simular el tiempo que se tarda en cocinar. Por último, manda un mensaje al hilo de impresión para que escriba en pantalla que Rafita sirvió las empanadas, se llena el MVar que actúa como flag de que están servidas con el valor True, se llena el MVar que simula la cantidad de empanadas en el colador con `m-1` empanadas y se incrementa su contador de empanadas servidas en `m`.

La razón por la cual el MVar del colador se llena con `m-1` en vez de `m` se debe a que el hilo que notó que hacían falta y avisó a Rafita que cocinara tiene garantizado una empanada, por ende solo hay `m-1` empanadas disponibles para los demás que vengan después.

```
parroquianoC :: RandomGen a =>
                  MVar Int -> MVar a -> MVar Int ->
                    MVar Bool -> MVar Bool -> Chan String ->
                      Int -> IO b
parroquianoC counter gen empanadas needMore served spool id
  = forever
  $ do
    randomDelayC (1 * 10^6) (7 * 10^6) gen
    writeChan spool $ "Parroquiano "
                        ++ show id
                        ++ " tiene hambre"
    takeEmpanada
    writeChan spool $ "Parroquiano "
                        ++ show id
                        ++ " come empanada"
  where
    takeEmpanada :: IO ()
    takeEmpanada
      = do
        empanada <- takeMVar empanadas
        if empanada > 0
          then putMVar empanadas (empanada - 1)
          else (do
                putMVar needMore True
                _ <- takeMVar served
                return ()
               )
        modifyMVar_ counter (pure . (+1))
```

Función que será ejecutada en la solución clásica por los hilos encargados de simular el comportamiento de los parroquianos. En primer lugar se ejecuta un `randomDelay` para simular el tiempo que transcurre para que al parroquiano le dé hambre. Luego, se manda un mensaje al hilo encargado de impresión mediante un canal para que se imprima en pantalla el mensaje de que el parroquiano P_i tiene hambre. Posteriormente, trata de agarrar una empanada.

Esto consiste en tomar la cantidad de empanadas del MVar que simula la cantidad de empanadas en el colador. Si esta cantidad es mayor que cero,

entonces se llena dicho MVar con el valor disminuido en 1. Si la cantidad no es mayor que cero, entonces se acabaron las empanadas y el parroquiano le avisa a Rafita que se neccesitan más. Esto se logra primero llenando el MVar que actúa como un flag de que se necesitan más empanadas con el valor True, luego se queda esperando hasta que el MVar que actúa como un flag de que se sirvieron más empanadas contenga un valor. Por la forma que se estructura el programa, se garantiza que el que pidió que se hicieran más tenga una empanada garantizada. Por último, se incrementa el contador de cuantas empanadas ha comido el parroquiano P_i en 1.

El último paso para cada ciclo de simulación es mandar un mensaje al hilo encargado de impresión para que imprima por pantalla un aviso que diga que el parroquiano P_i comió una empanada.

1.2. Solución

A continuación se presenta el cuerpo principal de ejecución de la solución transaccional junto con una explicación de su ejecución.

```
classic :: Int -> Int -> IO ()
classic m n
  = do
    gen <- newMVar $ mkStdGen randomSeed
    empanadas <- newMVar m
    needMore <- newEmptyMVar
    served <- newEmptyMVar
    totalRafa <- newMVar m
    spool <- newChan
    counters <- replicateM n $ newMVar 0
    printerID <- forkIO $ printer spool
    rafaID <- forkIO
                $ rafaC
                    gen
                    empanadas
                    needMore
                    served
                    totalRafa
                    spool
                    m
    threads <- for [1..n] $ \id ->
                  forkIO
                    $ parroquianoC
```

```
                        (counters !! (id-1))
                        gen
                        empanadas
                        needMore
                        served
                        spool
                        id
    forever (threadDelay (10^6))
     `catch`
     (\ (e :: SomeException) -> cleanup
                                (printerID : rafaID : threads)
                                counters
                                totalRafa)
  where
    cleanup :: [ThreadId] -> [MVar Int] -> MVar Int -> IO ()
    cleanup threads counters totalRafa
      = do
        for_ threads killThread
        results <- traverse takeMVar counters
        totalPreparados <- takeMVar totalRafa
        putStrLn "\n"
        putStrLn $ "Rafita preparó "
                     ++ show totalPreparados
                     ++ " empanadas."
        traverse_ imprimir $ zip [1..n] results
        putStrLn $ "Total:\t\t" ++ show (sum results)
      where
        imprimir :: (Int,Int) -> IO ()
        imprimir (id,num)
          = putStrLn $ "Parroquiano "
                         ++ show id
                         ++ ":\t"
                         ++ show num
```

El orden de ejecución de la solución clásica es el siguiente:

- Se generan los MVars y el Chan que serán usados durante la ejecución:

  gen: El MVar que contiene el generador del próximo número aleatorio. Se inicializa con la semilla provista por `randomSeed`.

  empanadas: El MVar que simula la cantidad de empanadas dentro del colador.

  needMore: El MVar que actúa como flag para indicar si se necesitan cocinar más empanadas.

  served: El MVar que actúa como flag para indicar que se sirvieron más empanadas.

totalRafa: El MVar que actúa como contador para llevar la cuenta de cuantas empanadas ha cocinado Rafita.

spool: El Chan que será utilizado como canal de los hilos de simulación al hilo encargado de la impresión de mensajes.

counters: Una lista de MVars que actúan como contadores para cada hilo que simula un parroquiano para llevar la cuenta de cuantas empanadas ha comido cada parroquiano.

- Se ejecuta un `forkIO` para crear el hilo encargado de la impresión por pantalla, recibe como argumento el Chan *spool*.

- Se ejecuta un `forkIO` para crear el hilo encargado de simular el comportamiento de Rafita. Recibe como argumentos los MVars *gen, empanadas, needMore, served, totalRafa*, el Chan *spool* y el número de empanadas a crear cada vez que cocina m.

- Se ejecutan n `forkIO`'s para crear los hilos encargados de simular el comportamiento de un parroquiano P_i. Cada `forkIO` recibe como argumentos los MVars *counter, gen, empanadas, needMore, served, spool* y el número i que identifica a cada parroquiano. El MVar *counter* es uno de los MVars de la lista de MVars *counters* que le corresponde a ese hilo en particular.

- El hilo principal ejecuta `threadDelay` de 1 segundo indefinidamente hasta que atrape una señal de interrupción de la ejecución. Cuando esto suceda se invoca la función `cleanup` pasándole como argumentos la lista de los id's de los hilos creados, la lista *counters* y el MVar *totalRafa*.

- Cuando se ejecuta `cleanup`, primero se matan a todos los hilos creados con la función `killThread`. Luego, se ejecuta `takeMVar` sobre todos los elementos de la lista *counters* y sobre el MVar *totalRafa*. Posteriormente se imprimen en pantalla los resultados de cuantas empanadas preparó Rafita, cuantas empanadas comieron cada uno de los parroquianos y la suma total de las empanadas consumidas.

2. Solución Transaccional

2.1. Funciones Auxiliares

Las funciones auxiliares utilizadas para la solución transaccional del problema se presentan a continuación y serán explicados brevemente. La función auxiliar `randomSeed` de la solución clásica es utilizada de nuevo en esta solución.

```
type Counter = TVar Int

newCount :: Int -> STM Counter
newCount n = newTVar n
```

Nuevo tipo que será utilizado en la solución transaccional para almacenar las empanadas consumidas por cada parroquiano y el total de empanadas hechas por Rafita.

```
output :: TChan String -> IO a
output spool
  = forever
  $ atomically (readTChan spool)
    >>=
    putStrLn
```

Función que será invocada infinitamente en la solución transaccional para leer del TChan e imprimir cada mensaje obtenido por pantalla.

```
randomDelayT :: RandomGen a => Int -> Int -> TVar a -> IO ()
randomDelayT lo hi gen
  = do
    time
      <- atomically
      $ do
        (time, newGen) <- randomR (lo, hi) <$> readTVar gen
        writeTVar gen newGen
        pure time
    threadDelay time
```

Función que será invocada durante la solución transaccional para poner a un hilo a dormir por una cantidad de tiempo aleatoria entre los parámetros

lo y *hi*. El nuevo generador aleatorio es almacenado en su TVar y se invoca la función `threadDelay` para dormir al hilo por esa cantidad de tiempo.

```
rafaT :: RandomGen a =>
          Counter -> Counter -> Int ->
            TChan String -> TVar a -> IO b
rafaT totalRafa empanadas m out gen
  = forever
  $ do
    atomically
      $ do
        checkEmpanadas
        writeTChan out "Rafita esta cocinando"
    randomDelayT (3 * 10^6)  (5 * 10^6) gen
    atomically
      $ do
        modifyTVar' totalRafa (+m)
        writeTChan out "Rafita sirvió las empanadas"
        writeTVar empanadas m
  where
    checkEmpanadas :: STM ()
    checkEmpanadas
      = do
        numEmpanadas <- readTVar empanadas
        if (numEmpanadas > 0)
          then retry
          else return ()
```

Función que será invocada por un hilo que se dedica a simular las acciones de Rafita en la solución transaccional. Primero revisa si todavía hay empanadas. Si hay al menos uno, el hilo se duerme hasta que ocurra un cambio en el número de empanadas y vuelve a revisar. Si ya no hay empanadas (es decir, cero empanadas) entonces manda un mensaje por el TChan para que sea impreso por pantalla, avisando que Rafita empieza a cocinar. Luego, el hilo se duerme por un tiempo aleatorio para simular el tiempo de cocinar las empanadas. Por último, incrementa el total de empanadas que cocinó en m de forma estricta (para evitar tener un thunk muy grande en el TVar), manda un mensaje por el TChan para avisar que ya sirvió las empanadas y actualiza el número de empanadas disponibles a m.

```
parroquianoT :: RandomGen a =>
                Counter -> Counter -> TChan String->
                  TVar a -> Int -> IO b
parroquianoT counter empanadas out gen id
  = forever
  $ do
    randomDelayT (1 * 10^6)  (7 * 10^6) gen
    atomically $ writeTChan out $ "Parroquiano "
                                    ++ show id
                                    ++ " tiene hambre"
    atomically
      $ do
        checkEmpanadas
        modifyTVar' counter (+1)
        writeTChan out $ "Parroquiano "
                           ++ show id
                           ++ " come empanada"
  where
    checkEmpanadas :: STM ()
    checkEmpanadas
      = do
        numEmpanadas <- readTVar empanadas
        if (numEmpanadas <= 0)
          then retry
          else writeTVar empanadas (numEmpanadas-1)
```

Función que será invocada por cada hilo que simula las acciones de un parroquiano P_i en la solución transaccional. Transcurre su tiempo de espera aleatorio para simular el tiempo que se tarda en tener hambre y manda un mensaje por el TChan para que sea impreso en pantalla, avisando que tiene hambre. Posteriormente revisa si hay empanadas, si no hay se duerme el hilo hasta que ocurra algún cambio en la cantidad de empanadas. Si habían empanadas, decrementa el número de empanadas en 1, incrementan estrictamente su contador de empanadas consumidas en 1 (para evitar tener un thunk muy grande en el TVar) y mandan otro mensaje por el TChan para que se imprima en pantalla, avisando que están comiendo una empanada.

2.2. Solución

A continuación se presenta el cuerpo principal de ejecución de la solución transaccional junto con una explicación de su ejecución.

```
transactional :: Int -> Int -> IO ()
transactional m n
  = do
    counters <- atomically $ replicateM n (newCount 0)
    totalRafa <- atomically $ newCount m
    empanadas <- atomically $ newCount m
    gen <- (atomically . newTVar) $ mkStdGen randomSeed
    spool <- atomically newTChan
    rafaID <- forkIO
                $ rafaT totalRafa empanadas m spool gen
    threads <- for [1..n] $ \id ->
                  forkIO
                    $ parroquianoT
                        (counters !! (id-1))
                        empanadas
                        spool
                        gen
                        id
    output spool
      `catch`
      (\ (e :: SomeException) -> cleanup
                                   (rafaID : threads)
                                   counters
                                   totalRafa)
  where
    cleanup :: [ThreadId] -> [Counter] -> Counter -> IO ()
    cleanup threads counters totalRafa
      = do
        for_ threads $ killThread
        results <- atomically $ traverse readTVar counters
        totalPreparados <- atomically $ readTVar totalRafa
        putStrLn "\n"
        putStrLn $ "Rafita preparó "
                     ++ show totalPreparados
                     ++ " empanadas."
        traverse_ imprimir $ zip [1..n] results
        putStrLn $ "Total:\t\t" ++ show (sum results)
      where
        imprimir :: (Int,Int) -> IO ()
        imprimir (id,num)
          = putStrLn $ "Parroquiano "
                         ++ show id
```

```
                                ++ ":\t"
                                ++ show num
```

El orden de ejecución de la solución transaccional es el siguiente:

- Se generan los TVars y el TChan que serán usados durante la ejecución:

  gen: El TVar que contiene el generador del próximo número aleatorio. Se inicializa con la semilla provista por `randomSeed`.

  empanadas: El TVar que simula la cantidad de empanadas dentro del colador.

  totalRafa: El TVar que actúa como contador para llevar la cuenta de cuantas empanadas ha cocinado Rafita.

  spool: El TChan que será utilizado como canal de los hilos de simulación al hilo encargado de la impresión de mensajes.

  counters: Una lista de TVars que actúan como contadores para cada hilo que simula un parroquiano para llevar la cuenta de cuantas empanadas ha comido cada parroquiano.

- Se ejecuta un `forkIO` para crear el hilo encargado de simular el comportamiento de Rafita. Recibe como argumentos los TVars *gen, empanadas, totalRafa*, el TChan *spool* y el número de empanadas a crear cada vez que cocina m.

- Se ejecutan n `forkIO`'s para crear los hilos encargados de simular el comportamiento de un parroquiano P_i. Cada `forkIO` recibe como argumentos los TVars *counter, gen, empanadas*, el TChan *spool* y el número i que identifica a cada parroquiano. El TVar *counter* es uno de los TVars de la lista de TVars *counters* que le corresponde a ese hilo en particular.

- El hilo principal ejecuta `output` con *spool* como parámetro indefinidamente hasta que atrape una señal de interrupción de la ejecución. Cuando esto suceda se invoca la función `cleanup` pasándole como argumentos la lista de los id's de los hilos creados, la lista *counters* y el TVar *totalRafa*.

- Cuando se ejecuta `cleanup`, primero se matan a todos los hilo creados con la función `killThread`. Luego, se ejecuta `readTVar` sobre todos los elementos de la lista *counters* y sobre el TVar *totalRafa*. Posteriormente se imprimen en pantalla los resultados de cuantas empanadas preparó Rafita, cuantas empanadas comieron cada uno de los parroquianos y la suma total de las empanadas consumidas.