Programación Funcional Avanzada

Memoria Transaccional

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia tradicional

. . . la historia hasta ahora

- Concurrencia explícita usando `forkIO`.
  - Hilos explícitos que se comunican con `MVars`.
  - Complejidad semántica para controlar la mutación del estado – *deadlock*, *starvation* como errores de programación.
  - Difícil razonar sobre las relaciones – ¿cómo combinar procesos concurrentes?
  - Control preciso sobre la granularidad del trabajo.
- Ideas con casi 30 años de historia, manifestadas de diferentes maneras pero con los mismos conceptos – variables condicionales y bloqueos.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué hay de malo en los bloqueos?

- Condiciones de *carrera* – ¡olvidé hacer un bloqueo!

¿Qué hay de malo en los bloqueos?

- Condiciones de *carrera* – ¡olvidé hacer un bloqueo!
- *Deadlock* – ¡equivoqué el orden de bloqueo!

¿Qué hay de malo en los bloqueos?

- Condiciones de *carrera* – ¡olvidé hacer un bloqueo!
- *Deadlock* – ¡equivoqué el orden de bloqueo!
- Omisión de *wakeup* – ¡olvidé notificar la condición!

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué hay de malo en los bloqueos?

- Condiciones de *carrera* – ¡olvidé hacer un bloqueo!
- *Deadlock* – ¡equivoqué el orden de bloqueo!
- Omisión de *wakeup* – ¡olvidé notificar la condición!
- *Error recovery from hell*
  - Restaurar invariantes.
  - Liberar bloqueos.
  - Ser consistente en los manejadores de excepciones.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué hay de malo en los bloqueos?

- Condiciones de *carrera* – ¡olvidé hacer un bloqueo!
- *Deadlock* – ¡equivoqué el orden de bloqueo!
- Omisión de *wakeup* – ¡olvidé notificar la condición!
- *Error recovery from hell*
  - Restaurar invariantes.
  - Liberar bloqueos.
  - Ser consistente en los manejadores de excepciones.

Pero esto no es lo **realmente** malo...

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué hay de malo en los bloqueos?

Los bloqueos y variables de condición **no** permiten programar de forma modular.

- Problemas pequeños "fáciles" de resolver con bloqueos no pueden *componerse* para resolver un problema más grande – reprogramar.
- Una vez que el problema se complica, el incremento en la complejidad introducida por los bloqueos es *absurdo*.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué hay de malo en los bloqueos?

Los bloqueos y variables de condición **no** permiten programar de forma modular.

- Problemas pequeños "fáciles" de resolver con bloqueos no pueden *componerse* para resolver un problema más grande – reprogramar.
- Una vez que el problema se complica, el incremento en la complejidad introducida por los bloqueos es *absurdo*.

Programar con bloqueos es
Malo, Dañino, Perverso y Cruel.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

- Imitación parcial de transacciones como en base de datos.

Software Transactional Memory

- Imitación parcial de transacciones como en base de datos.
- Escribir código secuencial optimista y *envolverlo* en un transacción.

Software Transactional Memory

- Imitación parcial de transacciones como en base de datos.
- Escribir código secuencial optimista y *envolverlo* en un transacción.
- Semántica "todo o nada" – *commit* atómico.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

- Imitación parcial de transacciones como en base de datos.
- Escribir código secuencial optimista y *envolverlo* en un transacción.
- Semántica "todo o nada" – *commit* atómico.
- El bloque se ejecuta aislado del resto.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

- Imitación parcial de transacciones como en base de datos.
- Escribir código secuencial optimista y *envolverlo* en un transacción.
- Semántica "todo o nada" – *commit* atómico.
- El bloque se ejecuta aislado del resto.
- *Deadlock* es imposible – ¡no hay bloqueos en los que equivocarse!

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

- Imitación parcial de transacciones como en base de datos.
- Escribir código secuencial optimista y *envolverlo* en un transacción.
- Semántica "todo o nada" – *commit* atómico.
- El bloque se ejecuta aislado del resto.
- *Deadlock* es imposible – ¡no hay bloqueos en los que equivocarse!
- Recuperación de errores simplificada – lanzar y atrapar excepciones.

Concepto aplicable, en principio, a cualquier lenguaje.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

La ventaja de usar Haskell

- Sistema de tipos separa *explícitamente* cómputos puros de impuros.

Software Transactional Memory

La ventaja de usar Haskell

- Sistema de tipos separa *explícitamente* cómputos puros de impuros.
- Cómputos puros no necesitan ser transaccionales – ¡no mutan estado!

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

La ventaja de usar Haskell

- Sistema de tipos separa *explícitamente* cómputos puros de impuros.
- Cómputos puros no necesitan ser transaccionales – ¡no mutan estado!
- Sólo queda implantar STM en Haskell con:
  - Tipo abstracto para modelar transacciones.
  - Tipo abstracto para modelar variables mutables *sólo* en transacciones.
  - Mecanismo para expresar transacciones.
  - Mecanismo para atrapar excepciones.
- Mantener el tipo transaccional separado de `IO`.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

La ventaja de usar Haskell

- Sistema de tipos separa *explícitamente* cómputos puros de impuros.
- Cómputos puros no necesitan ser transaccionales – ¡no mutan estado!
- Sólo queda implantar STM en Haskell con:
  - Tipo abstracto para modelar transacciones.
  - Tipo abstracto para modelar variables mutables *sólo* en transacciones.
  - Mecanismo para expresar transacciones.
  - Mecanismo para atrapar excepciones.
- Mantener el tipo transaccional separado de `IO`.

Un lenguaje funcional puro y con un sistema de tipos estricto y estático es el ambiente ideal para STM.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

`Control.Concurrent.STM`

- Modelar transacciones

```
data STM a                    -- Tipo abstracto
instance Monad STM            -- Secuenciar operaciones
```

  - Abstracto – implantación opaca en el *RTS*.
  - Imposible escapar del Monad transaccional.

UNIVERSIDAD SIMÓN BOLÍVAR

Software Transactional Memory

`Control.Concurrent.STM`

- Modelar transacciones

```
data STM a                    -- Tipo abstracto
instance Monad STM            -- Secuenciar operaciones
```

  - Abstracto – implantación opaca en el *RTS*.
  - Imposible escapar del Monad transaccional.

- Modelar variables mutables transaccionales

```
data TVar a                   -- Tipo abstracto
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

  - Existen y son mutables dentro del Monad STM.
  - ¡Polimórficas!

UNIVERSIDAD SIMÓN BOLÍVAR

Manipulando `TVar`

`Control.Concurrent.STM.TVar`

- Acciones monádicas manipulan `TVar` – sólo dentro del Monad `STM`.
- Cuenta Corriente

```
type Account = TVar Double
```

- Depósito en la cuenta

```
deposit :: Account -> Double -> STM ()
deposit a m = do s <- readTVar a
                 writeTVar a (s+m)
```

Falta expresar las transacciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Expresando transacciones

- Efectuar una operación de manera atómica – todo o nada.

```
atomically :: STM a -> IO a
```

  - Efectúa la transacción en memoria – la convierte en acción de I/O.
  - Ejecutar la acción de I/O toma en cuenta el contexto – otras transacciones concurrentes.
  - . . . y no se puede escapar del Monad IO.

UNIVERSIDAD SIMÓN BOLÍVAR

Expresando transacciones

- Efectuar una operación de manera atómica – todo o nada.

```
atomically :: STM a -> IO a
```

  - Efectúa la transacción en memoria – la convierte en acción de I/O.
  - Ejecutar la acción de I/O toma en cuenta el contexto – otras transacciones concurrentes.
  - ...y no se puede escapar del Monad IO.

- Fácil combinar varias operaciones para hacer un depósito ...

```
main = do ...
          atomically $ deposit account 42.0
          ...
```

UNIVERSIDAD SIMÓN BOLÍVAR

La separación efectiva

El mundo se separa en `IO` y `STM`

- Dentro del Monad `STM` sólo operaciones puras o sobre `TVars`
  - Porque las operaciones de I/O son *irrevocables*.
  - No se puede hacer `nukEmFromTheSky` en una transacción.
  - ¡Y el sistema de tipos lo garantiza!

UNIVERSIDAD SIMÓN BOLÍVAR

La separación efectiva

El mundo se separa en `IO` y `STM`

- Dentro del Monad `STM` sólo operaciones puras o sobre `TVars`
  - Porque las operaciones de I/O son *irrevocables*.
  - No se puede hacer `nukEmFromTheSky` en una transacción.
  - ¡Y el sistema de tipos lo garantiza!
- Fuera del Monad `STM` es imposible operar sobre `TVars`
  - Es imposible "obviar la protección" de `atomically`.
  - `TVars` son objetos de primera clase –
    construirlos y pasarlos de un lado para otro.
  - Modificarlos siempre estará restringido al contexto transaccional.
  - ¡Y el sistema de tipos lo garantiza!

Refinando las transacciones

- A veces hay que esperar por eventos consecuencia de la acción de otras transacciones concurrentes.

```
retry :: STM a
```

- Reintentar la transacción desde el principio – *rollback* explícito.
- Esperar por la modificación de al menos una de las `TVars` involucradas en la transacción.

Refinando las transacciones

- A veces hay que esperar por eventos consecuencia de la acción de otras transacciones concurrentes.

```
retry :: STM a
```

- Reintentar la transacción desde el principio – *rollback* explícito.
- Esperar por la modificación de al menos una de las `TVars` involucradas en la transacción.
- Retiro de la cuenta – más vale que haya fondos

```
withdraw :: Account -> Double -> STM ()
withdraw a m = do s <- readTVar a
                  if (s < m)
                    then retry
                    else writeTVar a (s-m)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Composición de transacciones

La separación paga dividendos

- `atomically` permite combinar operaciones –
  la combinación se efectuará. . . ¡atómicamente!
- La composición secuencial de transacciones es automágica –
  transferencia de una cuenta a otra

```
transfer :: Account -> Account -> Double -> IO ()
transfer src dst m =
  atomically $ do
    withdraw src m
    deposit dst m
```

- Si `withdraw` falla, gracias a `retry` se comienza desde el principio.

Haber definido `deposit` y `withdraw` como acciones en `STM` nos permite *componerlas*.

UNIVERSIDAD SIMÓN BOLÍVAR

Más combinadores ...

- Composicón secuencial automática – `STM` es un Monad.
- Componer con alternativas

```
orElse :: STM a -> STM a -> STM a
```

  - Si la primera operación tiene éxito, la transacción también es exitosa.
  - Si la primera operación falla, se intenta la segunda.
  - Si la segunda también falla, toda la transacción reintenta.
  - "Una o la otra" – excluyentes.
- Suele utilizarse en forma infija por razones obvias.
- Combinador poderoso para flujo condicional – el *invocante* decide si se bloquea o no.

UNIVERSIDAD SIMÓN BOLÍVAR

Aprovechando las alternativas

- Nuestra solución previa al problema de la transferencia se bloquea indefinidamente si no hay fondos – es mejor

```
funds :: Account -> Double -> STM Bool
funds acc m = do ( withdraw acc m
                   return True )
                 `orElse` return False

transfer :: Account -> Account -> Double -> IO ()
transfer src dst m =
  atomically $ do
    enough <- funds src m
    if enough
       then deposit dst m
       else warnAboutFundingOrRetry
```

Invariantes transaccionales

- Una condición que siempre debe cumplirse

```
always :: STM Bool -> STM ()
```

Invariantes transaccionales

- Una condición que siempre debe cumplirse

```
always :: STM Bool -> STM ()
```

- "No está permitido sobregirarse"

```
newAccount :: STM (TVar Int)
newAccount = do v <- newTVar 0
                always (do cts <- readTVar v
                           return (cts >= 0))
                return v
```

UNIVERSIDAD SIMÓN BOLÍVAR

Invariantes transaccionales

- Una condición que siempre debe cumplirse

```
always :: STM Bool -> STM ()
```

- "No está permitido sobregirarse"

```
newAccount :: STM (TVar Int)
newAccount = do v <- newTVar 0
                always (do cts <- readTVar v
                           return (cts >= 0))
                return v
```

- `always` agrega el invariante a un *pool* de invariantes que deben cumplirse antes y después de cada transacción.
- Ambiente de ejecución sólo verifica invariantes sobre `TVar` que cambiaron en la transacción en curso.
- `TVar` sale de alcance, invariantes asociadas también desaparecen.

UNIVERSIDAD SIMÓN BOLÍVAR

Cuando pasen cosas horribles

- Fracaso de transacción ligado a eventos impredecibles – excepciones.
- Capturar una excepción dentro de un transacción

```
throwSTM :: SomeException -> STM a
catchSTM :: STM a
         -> (SomeException -> STM a)
         -> STM a
```

- `SomeException` es un tipo "en serio"
  - Es *cualquier* excepción de la clase `Exception e`
  - Puede usarse `throwSTM` con libertad.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo implantarlo?

- Concurrencia optimista – asumir que "todo se puede" y fallar si el estado mutable no se mantiene consistente.

¿Cómo implantarlo?

- Concurrencia optimista – asumir que "todo se puede" y fallar si el estado mutable no se mantiene consistente.
- Registrar lecturas y escrituras en un *log* local a cada hilo – registra las `TVars` accedidas por la transacción.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo implantarlo?

- Concurrencia optimista – asumir que "todo se puede" y fallar si el estado mutable no se mantiene consistente.
- Registrar lecturas y escrituras en un *log* local a cada hilo – registra las `TVars` accedidas por la transacción.
- Al finalizar el segmento de código, verificar que el *log* siga siendo válido en relación a la memoria compartida – ¿las `TVar` cambiaron?.
  - ¿Válido? – aplicar cambios con *commit* irrevocable al *heap*.
  - ¿Inválido? – descartar el *log* y repetir desde el principio.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo implantarlo?

- Concurrencia optimista – asumir que "todo se puede" y fallar si el estado mutable no se mantiene consistente.
- Registrar lecturas y escrituras en un *log* local a cada hilo – registra las `TVars` accedidas por la transacción.
- Al finalizar el segmento de código, verificar que el *log* siga siendo válido en relación a la memoria compartida – ¿las `TVar` cambiaron?.
  - ¿Válido? – aplicar cambios con *commit* irrevocable al *heap*.
  - ¿Inválido? – descartar el *log* y repetir desde el principio.
- ¿Excepción? – descartar el *log* e invocar al manejador.
- ¿Y si no hay manejador? – propagarla fuera del `atomically`.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo implantarlo?

- Concurrencia optimista – asumir que "todo se puede" y fallar si el estado mutable no se mantiene consistente.
- Registrar lecturas y escrituras en un *log* local a cada hilo – registra las `TVars` accedidas por la transacción.
- Al finalizar el segmento de código, verificar que el *log* siga siendo válido en relación a la memoria compartida – ¿las `TVar` cambiaron?.
  - ¿Válido? – aplicar cambios con *commit* irrevocable al *heap*.
  - ¿Inválido? – descartar el *log* y repetir desde el principio.
- ¿Excepción? – descartar el *log* e invocar al manejador.
- ¿Y si no hay manejador? – propagarla fuera del `atomically`.
- ¿`retry`? – hay que esperar por cambios en el ambiente.
  - Hilo pasa a la lista de espera por modificación de las `TVar`.
  - La lista se revisa cada vez que hay un *commit*.

UNIVERSIDAD SIMÓN BOLÍVAR

MVars implantadas con STM

```
type MVar a = TVar (Maybe a)

newEmptyMVar = newTVar Nothing

takeMVar mv = do
  v <- readTVar mv
  case v of
    Nothing  -> retry
    Just val -> writeTVar mv Nothing >> return val

putMVar mv val = do
  v <- readTVar mv
  case v of
    Nothing -> writeTVar mv $ Just val
    Just _  -> retry
```

UNIVERSIDAD SIMÓN BOLÍVAR

MVars implantadas con STM

```
tryPutMVar mv val = do
  ( putMVar mv val >> return True )
    `orElse` return False

tryTakeMVar mv = do
  ( takeMVar mv >>= return . Just )
    `orElse` return Nothing
```

UNIVERSIDAD SIMÓN BOLÍVAR

Canales transaccionales

`Control.Concurrent.STM.TChan`

```
data TChan a                -- Tipo abstracto
newTChan      :: STM (TChan a)
readTChan     :: TChan a -> STM a
writeTChan    :: TChan a -> a -> STM ()
isEmptyTChan  :: TChan a -> STM Bool
```

- Canal no acotado que sólo puede manipularse en el monad `STM`.
- `TChan` es a `STM` como `Chan` es a `forkIO`.

Productor-consumidor con STM

```
main = do
  tc <- atomically $ newTChan

  -- Diez productores
  sequence_ . replicate 10 . forkIO $ do
    forM_ [1..100] $ \i -> do
      threadDelay 2
      atomically $ writeTChan tc i
    putStrLn "work, work!"

  -- Un consumidor
  forkIO . forever $ do
    threadDelay (1000 * 25)
    x <- atomically $ readTChan tc
    putStr $ (show x) ++ " "
```

Creatividad Composicional Concurrente

- Aplicar funciones a un `TVar` – ¡no son `Functor`!

```
updateTVar :: TVar a -> (a -> a) -> STM ()
updateTVar tv f = readTVar tv >>= writeTVar tv . f
```

Creatividad Composicional Concurrente

- Aplicar funciones a un `TVar` – ¡no son `Functor`!

```
updateTVar :: TVar a -> (a -> a) -> STM ()
updateTVar tv f = readTVar tv >>= writeTVar tv . f
```

- Combinar transacciones en una sola

```
merge :: [STM a] -> STM a
merge = foldr1 orElse
```

UNIVERSIDAD SIMÓN BOLÍVAR

Creatividad Composicional Concurrente

- Aplicar funciones a un `TVar` – ¡no son `Functor`!

```
updateTVar :: TVar a -> (a -> a) -> STM ()
updateTVar tv f = readTVar tv >>= writeTVar tv . f
```

- Combinar transacciones en una sola

```
merge :: [STM a] -> STM a
merge = foldr1 orElse
```

- Escoger una transacción con su acción IO asociada

```
choose :: [(STM a, a -> IO ())] -> IO ()
choose pairs = (atomically $ merge actions) >>= id
  where actions :: [ STM (IO ()) ]
        actions = [ txn >>= return . act |
                          (txn,act) <- pairs ]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Semáforos simples usando STM

```
type Semaphore = TVar Bool
```

- ...pero dentro del Monad IO – esperen un poco.

```
newSem available = newTVarIO available
```

- Las operaciones clásicas.

```
p sem = do b <- readTVar sem
           if b then writeTVar sem False
                else retry

v sem = writeTVar sem True
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un *buffer* transaccional

```
type Buffer = TVar (Seq a)
```

- ...pero dentro del Monad IO – sigan esperando.

```
newBuffer = newTVarIO empty
```

- Las operaciones

```
put b i = do ls <- readTVar b
             writeTVar b (ls |> i)

get b = do ls <- readTVar b
           case viewl ls of
             EmptyL   -> retry
             (i :< r) -> do writeTVar b r
                            return i
```

UNIVERSIDAD SIMÓN BOLÍVAR

La cena de los filósofos

Espero que lo hayan estudiado en Operativos

- Problema *clásico* de sincronización entre procesos (Dijkstra, Hoare, 1965)
- Hay n filósofos alrededor de la mesa – piensan `xor` comen.
- La mesa es circular y hay un gran perol de sushi en medio.
- Hay n palitos (*chopsticks*) distribuidos entre los filósofos – cada filósofo necesita **dos** para poder comer.
- Cada filósofo sólo usará los palitos a su derecha e izquierda.
- Los filósofos no se hablan – posible *deadlock*.
- Algunos piensan más que otros – posible *starvation*.

UNIVERSIDAD SIMÓN BOLÍVAR

La solución con STM

```
philosopher :: Int -> Buffer String
            -> Semaphore -> Semaphore -> IO ()
philosopher n out chst1 chst2 = do
  atomically $
    put out ("Filosofo " ++ show n ++ " pensando.")
  randomDelay
  atomically $ p chst1 >> p chst2
  atomically $
    put out ("Filosofo " ++ show n ++ " comiendo.")
  randomDelay
  atomically $ v chst1 >> v chst2
  philosopher n out chst1 chst2
```

- Un semáforo por palito – un *buffer* para registrar actividades.
- `randomDelay` es la razón del Monad IO.

UNIVERSIDAD SIMÓN BOLÍVAR

Las funciones auxiliares obvias

- Registrar la actividad del filósofo usando el *buffer*.

```
output buffer =
    do str <- atomically $ get buffer
       putStrLn str
       output buffer
```

¡Es un ciclo infinito recursivo de cola!

- Hacer que el hilo haga una pausa según la profundidad del pensamiento o hambre del filósofo.

```
randomDelay = do r <- randomRIO (100000,500000)
                 threadDelay r
```

UNIVERSIDAD SIMÓN BOLÍVAR

La simulación ...

Hilos explícitos con memoria transaccional – FTW!

```
simulation n = do
   -- Un semaforo por palito y registro de actividades
   chopsticks   <- replicateM n (newSem True)
   outputBuffer <- newBuffer

   -- Un hilo por filosofo asociado a sus palitos
   forM_ [0..n-1] $ \i ->
     forkIO (philosopher i outputBuffer
             (chopsticks !! i)
             (chopsticks !! ((i+1) `mod` n)))

   -- Muestra las actividades
   output outputBuffer
```

. . . simplemente funciona

```
$ ghci cena.hs
ghci> simulation 5
Filosofo 0 pensando.
Filosofo 1 pensando.
Filosofo 2 pensando.
Filosofo 3 pensando.
Filosofo 4 pensando.
Filosofo 0 comiendo.
Filosofo 2 comiendo.
...
```

. . . *ad nauseam*

Usar múltiples *cores* es cuestión de recompilar.

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Software Transactional Memory en WikiPedia
- Control.Concurrent.STM
- Control.Monad.STM
- Beautiful Concurrency

UNIVERSIDAD SIMÓN BOLÍVAR