Mecanismos de Sincronización en Programación Funcional Concurrente

Javier Blanco Pablo E. Martínez López Pablo A. Mocciola*

Facultad de Matemática, Astronomía y Física (FAMAF)

Ciudad Universitaria, 5000, Córdoba, República Argentina.

E-mail: blanco@mate.uncor.edu

LIFIA, Departamento de Informática, Universidad Nacional de La Plata.

C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.

E-mail: {fidel,pablom}@info.unlp.edu.ar

URL: http://www-lifia.info.unlp.edu.ar/

Resumen

Muchas aplicaciones y/o algoritmos se expresan de manera más sencilla utilizando lenguajes con estructuras o primitivas de concurrencia. En este tipo de problemas es necesaria la presencia y manipulación de variables o estructuras de estados, las cuales se utilizan como mecanismos de sincronización y comunicación entre los procesos. Los semáforos son herramientas utilizadas para solucionar problemas de secciones críticas e implementar protocolos de sincronización en programación concurrente. Los lenguajes funcionales son herramientas propicias para expresar paralelismo. *Concurrent Haskell* es una extensión concurrente del lenguaje funcional puro y lazy *Haskell*.

En este trabajo se presentan varias implementaciones de semáforos en *Concurrent Haskell* y se propone una generalización de las técnicas y mecanismos de sincronización y comunicación de procesos utilizando funciones de alto orden, como así también la posibilidad de expresar nuevas estructuras de manipulación de concurrencia como tipo de datos abstractos. Un punto importante es estudiar la posibilidad de utilizar lógicas aptas para la concurrencia en la verificación de programas funcionales concurrentes. De esta manera, se muestra que la programación funcional presenta una alternativa para el desarrollo de aplicaciones concurrentes.

Palabras clave: Programación Funcional, Programación Concurrente, *Concurrent Haskell*, Semáforos.

* El presente trabajo fue parcialmente financiado con una beca de la Fundación Antorchas.

Mecanismos de Sincronización

en Programación Funcional Concurrente

1 Introducción

Un programa concurrente es un programa en el cual el orden de ejecución de sus componentes no esta completamente especificado en el texto del mismo. Esta definición incluye, como un caso particular, a los programas en los cuales los diferentes componentes pueden ser ejecutados simultaneamente. Muchas aplicaciones y/o algoritmos pueden expresarse de una manera más sencilla en lenguajes que soportan concurrencia. En este tipo de problemas es necesaria la presencia de mecanismos de sincronización y comunicación entre los procesos. En este artículo, utilizamos la memoria compartida como forma de comunicar datos y a los semáforos como mecanismo de sincronización.

Los lenguajes funcionales [BW88] y en particular los puros y perezosos (*lazy*) son propicios para explotar el paralelismo. La ausencia de efectos laterales (también llamada transparencia referencial), característica fundamental de los lenguajes funcionales puros, permite que para evaluar una expresión `e1 + e2`, se pueda evaluar en paralelo `e1` y `e2`, posiblemente en diferentes procesadores, y luego obtener la suma de ambos resultados. La idea anterior se corresponde a la noción de paralelismo implícito [Hud91, JH93]. Este tipo de paralelismo, ocasionalmente llamado "ideal", es de por sí muy general; por ejemplo un compilador de un lenguaje que lo soporte debería ser también muy general, ya que no tendría suficiente información ni para elegir que subexpresión evaluar en paralelo ni en qué procesador hacerlo. Por otro lado, la idea de expresar problemas concurrentes lleva a pensar en una forma de paralelismo más explícito [Hud91, JH93, JGF96]; es decir, el lenguaje posee mecanismos o "anotaciones" cuya semántica establece la manera y el lugar donde se aplica la evaluación paralela o concurrente.

Concurrent Haskell [JGF96] es una extensión concurrente del lenguaje funcional puro y lazy *Haskell* [PH+96]; el mismo permite expresar explícitamente, mediante una serie de primitivas nuevas, mecanismos de sincronización y comunicación para el desarrollo de aplicaciones concurrentes.

Una de las primeras y más importantes herramientas utilizadas para solucionar problemas de secciones críticas e implementar protocolos de sincronización en programación concurrente [Dij79, Dij80, And91] son los semáforos.

Este trabajo forma parte de un proyecto más amplio, él cual tiene como objetivos analizar las facilidades y

características de un lenguaje funcional puro y lazy con primitivas de concurrencia, *Concurrent Haskell*, para expresar problemas de alto nivel que utilizan concurrencia y encontrar una lógica adecuada para verificar dichos programas.

Este trabajo en particular se centra en la especificación e implementación de los distintos tipos de semáforos (*binary*, *split-binary* and *general*) [Dij79, Dij80, And91] y aplicaciones de los mismos en *Concurrent Haskell*.

Además se presenta una implementación de barreras, las cuales son útiles para resolver los problemas de sicnronización de fase, presentes por ejemplo en lo algoritmos paralelos sobre matrices.

El artículo comienza con una breve explicación de algunos conceptos que se utilizan en las secciones posteriores: mónadas [Wad95], entrada/salida monádica (*monadic I/O*) [JW93] y las características básicas de *Concurrent Haskell* [JGF96]. A continuación se presentan las nociones básicas de semáforos, su representación en *Concurrent Haskell* y diferentes maneras de construir semáforos generales junto con algunos ejemplos característicos de sincronización y comunicación. Para los ejemplos se utiliza la notación de *Haskell* y las extensiones de *Concurrent Haskell*.

2 Ideas básicas

Los lenguajes funcionales puros proveen una excelente base para los sistemas de programación paralela. Esto se debe a la transparencia referencial de los mismos, la cual permite evaluar subexpresiones de un término dado sin ningún tipo de interferencia. Por otro lado, la ausencia de efectos laterales también ha sido vista como una debilidad de los lenguajes funcionales puros ya que de esta manera se excluyen características tradicionales de los lenguajes imperativos tales como manipulación de estados (variables), entrada/salida y excepciones. Las mónadas [Wad95] son una técnica que se utiliza para incorporar alguna de éstas características en los lenguajes funcionales de manera natural y eficiente.

2.1 Mónadas

Se puede definir a las mónadas como una familia de tipos de datos abstractos que encapsulan estados, y efectos sobre esos estados, para controlar la manera en que se usan los aspectos o efectos imperativos antes mencionados. Las únicas operaciones que se pueden aplicar sobre estos tipos de datos abstractos son operaciones de manipulación del estado (leer, escribir, etc.) y de composición de operaciones como las anteriores. Básicamente, una mónada

tiene que tener dos operaciones, `return :: a -> m a`, `(>>=) :: m a ->(a -> m b)-> m b`, y cumplir con una serie de leyes algebraicas; para más detalle consultar [Wad95].

Una expresión de la forma `return e` representa la computación trivial que produce un resultado `e` sin realizar ninguna acción ni efecto sobre el estado que encapsula la mónada. Por otro lado el operador `(>>=)` (llamado *bind*) puede ser pensado como una manera de secuenciar efectos o computaciones. Es decir, `m >>= f` computa la mónada `m`, que devuelve un resultado que es utilizado por `f` para realizar su computación. Si llamamos `a` al resultado de `m`, entonces `(f a)` se llama "continuación" de `m` en `(m >>= f)`.

A fin de simplificar la lectura de expresiones monádicas se provee una notación especial, conocida como *do-notation*. La misma consiste en una lista de expresiones monádicas, con guardas de patrón opcionales, y precedidas por la palabra clave `do`. Una guarda de patrón especifica la variable a la que ligar el resultado de la expresión monádica, y provee una abreviatura para bind. La expresión `e1>>=\var->e2` se escribe `do {var<-e1; e2}` y la expresión `e1>>e2` se escribe `do {e1;e2}`. Gracias a la *layout rule*, que especifica que la indentación es significativa, se pueden eliminar las llaves y puntos y coma de las expresiones `do`. Entonces, `do {e1;...;en}` es equivalente a

```
do e1
   ...
   en
```

2.2 Entrada/salida monádica

Wadler y Peyton-Jones [JW93] utilizaron el estilo monádico para desarrollar lo que se conoce como entrada/salida monádica (*monadic I/O*). Esta técnica ha creado un nuevo estilo de programación funcional, conocido como "programación funcional imperativa".

La entrada/salida monádica se basa en la transformación paulatina de estados (*state transformers*) [Wad95] que produce cada computación, esto es, una función (computación) que transforma el estado actual en uno nuevo. Además se requiere que las computaciones retornen valores o resultados; luego la definición del tipo que representa lo dicho anteriormente queda como: `type IO a = World -> (a, World)`

Llamaremos acciones o computaciones a las expresiones de tipo `IO a`. Además de las acciones monádicas vistas, usaremos las siguientes acciones y operadores.

```
--Combinators
(>>)   :: IO a -> IO b -> IO b            --evalua m, descarta su resultado y evalua n
m >> n = m >>= \_ -> n
sequence   :: [IO a] -> IO ()             --secuencia efectos
sequence   = foldr (>>) return ()
--Actions
hGetChar :: Handle -> IO Char             --lee un caracter desde un disp. determinado
hPutChar :: Handle -> Char -> IO ()       --escribe un caracter en el dispositivo
--other important actions:
skip  :: IO ()                            --no action
skip  = return ()
print :: String -> IO ()                  --imprime st en la salida standard
print st = sequence (map (hPutChar stdout) st)
```

Al ser `IO` una mónada se puede utilizar la *do-notation*, dandole a los programas un aspecto imperativo.

2.3 Concurrent Haskell

Concurrent Haskell [JGF96] es una extensión concurrente ya implementada y disponible gratuitamente del lenguaje funcional *Haskell* [PH+96]. Esta extensión consiste, básicamente, en una serie de primitivas con una semántica determinada. Los ingredientes principales que agrega *Concurrent Haskell* son los procesos, junto con un mecanismo para la inicialización de los mismos, y el concepto de estado mutable (*atomically-mutable state*) o variables mutables, para poder realizar la comunicación y cooperación entre procesos.

La primitiva que provee *Concurrent Haskell* para crear un proceso concurrente es `forkIO :: IO() -> IO()`. La función `forkIO` es un combinador que toma una acción (programa) como argumento y de alguna manera genera o "dispara" un proceso concurrente que realiza dicha acción; es decir, `forkIO` crea un proceso que se ejecuta concurrentemente con el proceso continuación. El siguiente ejemplo ilustra la utilización de `forkIO`.

```
let loop st = print st >> loop st   -- loop st imprime una secuencia infinita de st's
 in forkIO (loop "Concurrent") >> loop "Continuation"
```

La primitiva `forkIO` no es suficiente para proveer programación concurrente en *Haskell*; se necesitan mecanismos que permitan la sincronización y comunicación entre procesos. Las bases para conformar estos mecanismos radican en lo que llamamos anteriormente estados u objetos mutables [JGF96, LJ96, LJ94].

¿Qué es un estado en funcional? Un estado es una referencia a una posición o variable modificable que

puede almacenar un valor. Es difícil imaginarse la idea de estado o variable en un lenguaje funcional puro. El sistema [LJ94] que soporta la manipulación de estados mutables encapsula las computaciones de estados (*stateful computations*) de forma segura, utilizando mónadas, y garantiza una completa transparencia referencial completo control del programador sobre los objetos mutables y propiedades de *lazyness* sobre los mismos, sin perder seguridad.

El tipo y las operaciones primitivas que definen a las variables mutables son las siguientes:

```
type MutVar
newVar   :: a -> IO (MutVar a)          -- Retorna una MutVar's con valor a
readVar  :: MutVar a -> IO a            -- Lee el valor de una MutVar
writeVar :: MutVar a -> a -> IO ()      -- Escribe un valor a en una MutVar
```

La función `newVar` toma un estado inicial, de un tipo dado, y crea una nueva referencia a una variable que contiene dicho valor inicial, mientras que las operaciones `readVar` y `writeVar` leen y escriben, respectivamente, valores de las referencias o variables.

Existen otro tipo de variables mutables, que toman sus bases de las anteriores y que son las que sirven para realizar los mecanismos de sincronización y comunicación de *Concurrent Haskell*. Primero se define el tipo primitivo que las caracteriza: `type Mvar a`. Un valor de tipo `Mvar t`, para algún tipo `t`, es el nombre o referencia a una posición mutable que, o bien está vacía, o bien contiene un valor de tipo `t`. Las operaciones primitivas que actúan sobre las `Mvar` son `newMVar :: IO Mvar` que crea una nueva `Mvar`, `takeMVar :: Mvar a -> IO a` que se bloquea mientras la posición este vacía y luego lee y retorna el valor, dejando la posición nuevamente vacía, y `putMVar :: Mvar a -> a -> IO()` que escribe un valor en la posición especificada. Si hay uno o mas procesos bloqueados en un `takeMVar` de dicha posición, entonces uno es "despertado" y continua su ejecución. Es un error realizar una operación `putMVar` sobre una posición que ya contiene un valor. Como comentario final sobre las `Mvar`, cabe mencionar las diferentes maneras en que éstas pueden ser vistas: una versión sincronizada de las `MutVar`, un canal de comunicación (productor/consumidor), donde sus operaciones (`takeMVar`, `putMVar`) hacen las veces de las operaciones *receive* y *send* de los canales, o un semáforo binario; esta última se utiliza en el resto del artículo.

Es interesante destacar que dos procesos concurrentes creados con la función `fork` comparten las variables mutables creadas anteriormente, pudiendo interferir uno con el otro, por lo cual, para garantizar el acceso seguro a las mismas, deben utilizarse mecanismos de sincronización y comunicación.

3 Semáforos

El concepto de semáforo [And91] está motivado por una de las maneras de sincronizar o controlar el tráfico de trenes para evitar colisiones. Los semáforos de trenes son una especie de bandera que indican si la vía esta libre u ocupada. Los semáforos pueden activarse y luego liberarse, de tal manera que si un tren ocupa la sección crítica de la vía, otro pueda detenerse o disminuir su velocidad hasta que el primero la libere. Los semáforos en programación concurrente son muy similares: proveen un mecanismo de señalización y son usados para implementar exclusión mutua y condiciones de sincronización.

Entonces, un semáforo es un tipo de dato abstracto que encapsula un valor, y que puede ser manipulado solamente por dos operaciones atómicas especiales, `P` y `V` (en algunas bibliografías suelen llamarse `wait` y `signal` respectivamente). La operación `V` señaliza la ocurrencia de un evento; la operación `P` es usada para "dormir" a un proceso hasta que ocurra un evento. Los semáforos generales encapsulan un número natural, inicializado en 0. La implementación imperativa de la operación `P` es `P(s): < await s>0 r s:=s-1 >` y la de `V` es `V(s): < s:=s+1 >`; el invariante del semáforo es `s >= 0`, donde `< S >` indica que la sentencia `S` es ejecutada de manera atómica.

En un primer momento, fijaremos nuestra atención en los semáforos binarios (*binary semaphore*), en los cuales su valor interno puede ser únicamente 0 ó 1. Veremos que con este tipo de semáforos se podrán expresar y solucionar una gran variedad de problemas, entre los cuales podemos mencionar exclusión mutua, grafo de precedencias, escritores/lectores; también se menciona la técnica de semáforo binario partido (*split-binary semaphore*) con la cual se implementan los semáforos generales.

La manera de expresar a los semáforos binarios en *Concurrent Haskell* se origina directamente de la definición o semántica de las `Mvar`s antes mencionadas. La definición de los mismos es:

```
type Sem = Mvar()
newSem :: IO Sem            signal :: Sem -> IO()         wait   :: Sem -> IO()
newSem = newEmptyMVar       signal s = putMVar s ()       wait s = takeMVar s
```

4 Semáforos generales

En esta sección veremos como implementar semáforos generales a partir de semáforos binarios y variables enteras usando la técnica de los *split-binary semaphores* (*SBS*) [Dij79, Dij80, And91]. Los *SBS* están formados por n semáforos binarios que, para cualquier programa, siempre cumplen el invariante global: `SPLIT: 0` $\leq$ `b1 +`

`...+ bn` $\leq$ `1`, donde `bi` es el valor del semáforo i-ésimo. Ellos ofrecen una técnica sistemática para el diseño de programas concurrentes en los cuales la interacción de los componentes secuenciales de los programas están restringidos a secciones críticas.

Volviendo al punto de los semáforos generales, la idea es representar con *SBS* las operaciones e invariante de los semáforos generales vistas anteriormente. Para esto utilizaremos una variable `k`, que representa al semáforo en sí mismo y dos semáforos binarios `m` y `s` (inicializados en 1 y 0 respectivamente). A continuación se muestra la implementación final en funcional de las operaciones `wait` y `signal` para semáforos generales. Para ver el proceso de construcción completo, se recomienda ver [Dij79, Dij80].

Nótese en ete programa, que se mantiene invariante la relación

$$0 \leq m + s \leq 1$$

Por otro lado, si k es positiva, entonces su valor es el valor del semáforo, mientras que si es negativa, entonces es el número de procesos esperando en el correspondiente semáforo.

```
type GSem = (Sem, Sem, MutVar Int)
newGSem :: Int -> IO GSem
newGSem n = do m <- newSem
               signal m
               s <- newSem
               k <- newVar n
               return (m, s, k)

signalG :: GSem -> IO()
signalG (m, s, k) = do wait m
                       valk <- readVar k
                       writeVar k (valk+1
                       signal (if (valk < 0) then s else m)
waitG   :: GSem -> IO()
waitG (m, s, k) = do wait m
                     valk <- readVar k
                     writeVar k (valk-1)
                     if (valk > 0) then skip else signal m >> wait s
                     signal m
```

En [JGF96] se muestra otra implementación de semáforos generales que utiliza una `Mvar` que contiene el valor

del semáforo y una lista de semáforos binarios que indica los procesos que se encuentran dormidos esperando una operación `signal`. Cabe destacar que la implementación presentada en este artículo se construye exclusivamente a partir de semáforos binarios utilizando la técnica *SBS* y sólo se usan dos de ellos, independientemente de la cantidad de procesos dormidos.

5 Readers and Writers

Una buena manera de entender las estructuras concurrentes es mediante ejemplos. En esta sección se describe el ejemplo de lectores/escritores (*readers/writers*).

El problema de los *readers/writers* es un clásico problema de sincronización. Existen dos clases de procesos, *readers* y *writers*, que comparten una base de datos. Los *readers* ejecutan transacciones que sólo examinan los registros de la base de datos; los *writers* también pueden actualizar la base de datos. Las transacciones de los *writers* deben realizarse en forma exclusiva para garantizar la integridad de la base de datos. Si ningún *writer* está accediendo a la base de datos, entonces los *readers* pueden ejecutar concurrentemente sus transacciones. Esta propiedad se especifica más sucintamente como el siguiente invariante, donde nr y nw representan respectivamente el número de *readers* y *writers* respectivamente.

$$nw = 0 \vee (nw = 1 \wedge nr = 0)$$

En la implementación descrita a continuación se utiliza una variante de la técnica *SBS* (ver Sect. 4) llamada *passing the baton* [And91]. Una de las propuestas futuras es especificar esta técnica de transformación de programas como una función de alto orden que dada una especificación *coarsed-grained* de un proceso concurrente retorne una implementación de la misma con *SBS*. Sólo se muestra la implementación del proceso *read*; el proceso *write* se construye de la misma manera, cambiando los contadores y la guarda de condición en el acceso a la base de datos. En esta implementación se le dá prioridad a los lectores, aunque ésto puede cambiarse trivialmente modificando la función `signalSBS`.

```
--Notes: dr,dw counters of rs/ws are delayed; nr,nw counters of rs/ws
--accessing the db; m,r,w are mutex, read and write binary sem.
type SBS = (Sem, Sem, Sem)
type IntVar = MutVar Int
type Vars = (IntVar, IntVar)
```

```
reader :: SBS -> Vars -> Vars -> IO ()
reader sbs@(m,r,w) d@(dr,dw) v@(nr,nw) =
           doTrue (do wait m
                      br <- guardR nw
                      if not br
                         then do inc dr
                                 signal m
                                 wait r
                         else skip
                      inc nr
                      signalSBS sbs d v
                      readDataBase               -- seccion critica
                      wait m
                      dec nr
                      signalSBS sbs d v )
guardR :: IntVar -> IO Bool                      -- evalua si hay algun escritor activo
guardR nw = do valnw <- readVar nw
               return (valnw==0)
guardW :: IntVar -> IntVar -> IO Bool            -- hay lectores o escritores activos
guardW nw nr = do valnw <- readVar nw
                  valnr <- readVar nr
                  return (valnw==0 && valnr==0)
signalSBS :: SBS -> Vars -> Vars -> IO ()
signalSBS sbs@(m,r,w) d@(dr,dw) v@(nr,nw) =
                   do valdr <- readVar dr
                      valdw <- readVar dw
                      br    <- guardR nw         -- br <=> no hay escritores
                      bw    <- guardW nw nr      -- bw <=> no hay escritores ni lectores
                      if (br && dr>0)            -- hay algun lector esperando y puede
                      then do dec dr             --     ser despertado
                              signal r
                      else
                      if (bw && dw>0)            -- hay algun escritor esperando y puede
                      then do dec dw             --     ser despertado
                              signal w
                      else signal m
doTrue :: IO() -> IO()
dotrue io = io >> doTrue
```

6 Barrier Synchronization

En esta sección se presenta otro mecanismo de sincronización para procesos concurrentes. Esta técnica de sincronización se utiliza para modelar y resolver algoritmos interactivos paralelos de la forma:

```
Process[i:1..n]:: do true ->
                     code to implement task i
                     wait for all n tasks to complete
                  od
```

donde, cada proceso necesita esperar a que todos los otros hayan terminado su iteración, antes de continuar con la siguiente; este tipo de sincronización se llama *barrier synchronization*. Los *barriers* [And91] se encargan de controlar este tipo de sincronización, fijando un punto de espera (una barrera) en donde todos los procesos deben esperar a que todos los restantes procesos lleguen para poder continuar su ejecución.

En [And91] se muestran diferentes implementaciones de *barriers* utilizando distintas técnicas de interacción de procesos, algunas de ellas son *shared counter*, *flags and cordinators*, *combining tree*, *butterfly and dissemination*, etc. Por otro lado, la implementación de *barriers* presentada en este trabajo no sigue ninguna de las técnicas antes mencionadas; sólo utiliza las primitivas de `MVar`s provistas por *Concurrent Haskell* y los semáforos binarios de la Sect. 3.

Los *barriers* se definen como un tipo de dato abstracto con dos operaciones principales: una de inicialización y otra de sincronización. La representación interna del *barrier* consta de la cantidad de procesos que utilizan la barrera, una *Mvar* que guarda la cantidad de pasos o veces que se cruzó la barrera, la cantidad de procesos que faltan arribar a la misma y una lista de los semáforos binarios correspondientes a los procesos dormidos; además se tienen los semáforos binarios que se utilizan para dormir a cada proceso en la barrera. A continuación se muestra la implementación en *Haskell*.

```
data Barrier = B Int (Mvar (Int, Int, [Sem])) [Sem]
newBarrier :: Int -> IO Barrier
newBarrier n = do m    <- newMVar (0, n-1, [])       --inicializa la barrera
                  sems <- newEmptySemaphores n       --n semaforos vacios
                  return (B n m sems)
barrier :: Int -> Barrier -> IO ()
barrier i (B n m sems) =
               do (step, toArrive, xs) <- takeMvar m
                  if toArrive == 0
```

```
        then do sequence (map signal xs)    --despierta todos los semaforos
                putMVar m (step+1, n-1, []) --un paso nuevo de la barrera
        else do putMVar m (step, toArrive-1, s:xs)
                wait s
    where s = sems !! (i-1)
```

En este punto, cabe destacar la importancia de representar a los mecanismos de sincronizción como tipos de datos abstractos, ya que de esta manera se puede realizar la especificación de los mismos independientemente de la implementación y la técnica de interacción de procesos utilizada.

7 Sumas Parciales

Un algoritmo paralelo de datos es un algortimo iterativo que manipula constantemente y en paralelo un arreglo compartido. En esta sección se presenta un ejemplo donde se utilizan los *barriers* para desarrollar una solución a un algoritmo paralelo de datos. El problema a resolver se conoce como el problema de las *sumas parciales*, pero puede ser generalizado al cálculo de computaciones prefijas en paralelo (*CPP*) [And91]. Las *CPP* son útiles en muchas aplicaciones, incluyendo procesamiento de imágenes, computaciones de matrices y parsing de lenguajes regulares. En este ejemplo se utiliza la suma matemática, pero el algoritmo básico puede ser usado para cualquier operador binario.

El problema se puede instanciar de la siguiente manera: dado un arreglo $a[1:n]$ calcular $sum[1:n]$, donde $sum[i]$ es la suma de los primeros i elementos de a.

A continuación se muestra una implementación de la generalización (*CPP*)del algoritmo que resuelve este problema. Existe un proceso por cada elemento del arreglo, que va calculando las computaciones prefijas que le corresponda, en paralelo con los otros procesos. Los *barriers* son necesarios para evitar interferencia entre los procesos; por ejemplo, el arreglo $ppc[1:n]$ necesita ser inicializado antes de cualquier proceso pueda leerlo. La implementación utilizada es la de la Sect. 6 El algoritmo comienza con la inicializacion del arreglo de computaciones prefijas (cada proceso colabora en esta tarea), y luego, progresivamente, se lo procesa en forma sincronizada utilizando los *barriers* (función `bodyppc`).

```
type IdProcess  = Int
type BinaryOp a = a->a->a
sumas = doppc (+)
-- ppc : parallel prefix computation
```

```
doppc:: BinaryOp a -> IdProcess -> Barrier -> Array a -> Array a -> Array a -> IO ()
doppc bOp i bar old ppc a =
    do ai <- readArray a i
       writeArray ppc i ai                                  -- ppc[i]:= a[i]
       barrier i bar                                        -- barrier sincronization
       bodyppc bOp i 1 old ppc bar
bodyppc :: BinaryOp a -> IdProcess -> Int -> Array a -> Array a -> Barrier -> IO()
bodyppc opB i d old ppc bar =
   do if (d<n)                                              -- n=num. de elem. en sum
      then do ppci <- readArray ppc i
              writeArray old i ppci                         -- old[i]:=ppc[i]
              barrier i bar                                 -- barrier sincronization
              if (i-d)>=1
              then do oldi_d <- readArray old (i-d)
                      writeArray ppc i (oldi_d 'opB' ppci)  -- ppc[i]:=old[i-d]+ppc[i]
              else skip
              barrier i bar                                 -- barrier sincronization
              bodyppc opB i (d*2) old ppc bar
       else skip
```

También se pudo analizar las diferencias acerca de las distintas implementaciones para la solución del problema. La implementación en paralelo requiere la utilización de mecanismos de sincronización de procesos, como son los *barriers*, pero es mucho más eficiente que la secuencial (del orden de $\lceil log_2 n \rceil$). Las implementaciones imperativas se pueden traducir facilmente a código funcional, pero el algoritmo, en sí mismo, se expresa de manera más natural en un lenguaje funcional. El paso siguiente es probar la equivalencia entre las dos alternativas, secuencial y paralela, utilizando posiblemente lógicas aptas para la concurrencia y razonamiento ecuacional.

8 Conclusiones y Trabajos Futuros

La especificación de mecanismos de sincronización y comunicación de procesos concurrentes, como los semáforos, en un lenguaje funcional como *Concurrent Haskell* se logra de manera muy sencilla.

Durante este trabajo se buscó especificar aspectos de la concurrencia desarrollados en forma imperativa, con un lenguaje funcional. Esto último no invalida la idea de encontrar mecanismos puramente funcionales. Por otro lado, como una conclusión final sobre los semáforos funcionales, podemos decir que estos pueden expresarse

de manera concisa y agregarse fácilmente al lenguaje. Además pueden utilizarse naturalmente para solucionar problemas concurrentes.

Este trabajo constituye uno de los pasos iniciales de un plan de proyecto más grande, el cual tiene como objetivo caracterizar a la programación funcional como una herramienta para expresar programas concurrentes.

Los trabajos próximos radicarán en especificar otros mecanismos de comunicación y sincronización como pueden ser los canales, regiones críticas y monitores. De esta manera, intentamos aprovechar las ventajas del alto orden en el desarrollo de programas concurrentes.

Otro punto sobre el que se centrará la atención, es en la verificación de programas funcionales concurrentes. En programas funcionales que utilizan las mónadas de manera intensiva, el razonamiento ecuacional no parece suficiente para demostrar la corrección de éstos. Por lo tanto, uno de los puntos más importantes a estudiar es la posibilidad de utilizar lógicas aptas para la concurrencia (e.g. la lógica de Owicki-Gries) en un marco con alto orden y si puede mantenerse, por supuesto con restricciones, alguna forma del razonamiento ecuacional que es tan cómodo cuando se trabaja con programas funcionales puros.

Referencias

[And91] G.R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc, 1991.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[Dij79] E. W. Dijkstra. A tutorial on the split binary semaphore. Technical Report EDW703, Nuenen, The Netherlands, March 1979.

[Dij80] E. W. Dijkstra. The superfluity of the general semaphore. Technical Report EDW734, Nuenen, The Netherlands, April 1980.

[Hud91] Paul Hudak. Para-functional programming in Haskell. In *In Parallel functional languages and compilers,* pages 159–196. ACM Press (Ney York) and Addison-Wesley, 1991. Chapter 5.

[JGF96] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages (PoPL'96)*, St.Petersburg Beach, Florida, January 1996.
URL `ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/glasgow`.

[JH93] M. P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Technical report, YALEU/DCS/RR-982, August 1993.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (PoPL)*, pages 71–84, January 1993.

[LJ94] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, Orlando, June 1994.

[LJ96] John Launchbury and Simon Peyton Jones. State in Haskell. Technical report, Kluwer Academic, 1996.

[PH+96] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale University, May 1996.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*. Springer-Verlag, May 1995.