BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Concurrencia Tradicional en Programación Funcional

Pablo Andrés Mocciola

Director: Dr. Javier Blanco
Co-Director: Prof. Gabriel Baum

Licenciatura en Informática
Facultad de Ciencias Exactas
Dpto. de Informática

Universidad Nacional de La Plata

DONACION
$
Fecha 28-9-05
Inv. E. Inv. B. 2060

TESIS
98/16 ej. 1

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

A mis padres, Rafa y Mary.

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Agradecimientos

Querría agradecer a mis directores Javier Blanco y Gabriel Baum por su confianza, recomendaciones y orientación a lo largo de estos últimos años, sin los cuales este trabajo no hubiera sido posible. De la misma manera me gustaría agraceder a Juan Echague, Pedro D'argenio, Pablo Martínez López y Natalio Krasnogor por sus innumerables consejos e interminables charlas de café.

Después de tantos años de estudios, esta tesis marca de alguna manera el final de mi carrera de Licenciado en Informática. Es ahora el momento de recordar y agradecer a todas aquellas personas, corriendo el riesgo de olvidarme de alguien, que contribuyeron para que pudiera llegar a esta instancia final.

A mis viejos, por el esfuerzo impagable que hicieron para que pudiera soñar, volar y alcanzar mis objetivos. Los amo con todo mi corazón, a ellos le dedico este trabajo final con todo mi amor. A mis hermanos Lino y Rosi, a Suyay y a mis tres angelitos Bruno, Lucas e Iliana. Al resto de mi gran familia les agradezco con todo mi corazón la confianza y el apoyo que me dieron, a Fer, Ari, Viqui, Ceci, María, Mica, Marce, Chiqui y Edu. A la abuela, que desde su mundo sepa comprender todo lo que la quiero y extraño. A mis padrinos, Alicia y Cuqui, por ser excelentes personas, por hacerme sentir que siempre están conmigo a pesar de los kilómetros. A mi "nueva" familia de Bragado por el cariño que me brindan.

Al Tavo, un hermano de la vida, compañero de vivencias durante estos años en La Plata. Gracias Tavo por "bancarme" estos años y suerte en esta nueva etapa. A mis amigos de Roca, la misma banda que añoro reencontrar en cada verano, la dueña de tantas aventuras, al Chelo, Mendu, Caño, Teto, Cabeza, Pipí, Ariel, Cachi, Piñon, Negro, Tucho, Colo y fundamentalmente a Santi para que pueda superar este amargo momento de la vida, a todos ellos millones de gracias.

Me parece mentira pensar que ya pasaron seis años, recuerdo las horas de estudios con los entonces compañeros y ahora amigos, a todos ellos también quiero agradecerles. Gracias Negra, Fidel, Ariel, Martín, Tito, Ramiro,

Guara, Fernando, Tucu, Pincha, Pola, Pete y Bocha. A Viviana, mi mejor amiga, por estar siempre en los momentos que la necesité y por presentarme a su amiga Laura, la mejor mujer del mundo.

A los directores del LIFIA, Gustavo Rossi, Gabriel Baum, Silvia Gordillo y Alicia Díaz por iniciarme en el mundo de la investigación. A toda la gente del laboratorio y en especial a los del boliche "λ". No quiero olvidarme de agradecer a compañeros del laboratorio, que tal vez por ser más grandes que yo, supieron transmitirme sus vivencias y consejos. Gracias Román, Vero, Majo, Nachi, David, Natalio, Matías, Eduardo y Gustavo.

A Javier e Ines por su generosa hospitalidad durante mi estadía en la ciudad de Córdoba. Gracias por hacerme sentir como en mi casa. A la gente del FAMAF de la Universidad de Córdoba por su coooperación y calidez.

A Victoria Horwitz, y mediante ella, a toda la Fundación Antorchas por confiar en alumnos como yo y fomentar la investigación en la Argentina. A la Profesora Ana María Ferrari, directora de la Escuela de Lenguas de la Facultad de Humanidades y Ciencias de la Educación de la UNLP, y a quién ella considere pertinente, mis más sinceros agradecimientos por darme la posibilidad de estudiar el idioma Inglés.

A toda la gente que cree en la paz y en un mundo mejor para toda la humanidad.

Por último, pero no por eso menos importante, quiero agradecer a Laura, el amor de mi vida, por su ternura y paciencia. Mi Amor, ¡gracias por existir!

Pablo Andrés Mocciola
La Plata, 16 de diciembre de 1998

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Prefacio

La Programación Funcional pura ha madurado en los últimos quince años. Varios desarrollos, tanto en el lado teórico como en lo práctico, han producido un florecimiento del área. El lenguaje Haskell es un lenguaje avanzado de programación funcional pura y perezosa, pensado para aplicaciones del mundo real, el cual es el resultado de los mencionados desarrollos.

Con el advenimiento del modelo de entrada/salida monádico, a partir de la versión 1.3 del Haskell, la cáscara de más alto nivel de los programas funcionales se caracteriza por una secuencia de comandos u operaciones imperativas que tienen reminiscencia con los programas escritos en lenguajes imperativos como Pascal. Extensiones concurrentes y paralelas de Haskell han sido desarrolladas. Concurrent Haskell, la extensión concurrente del Haskell, trabaja sobre el *framework* monádico. Las propiedades de los lenguajes funcionales son mantenidas adecuadamente y aseguradas por el lenguaje. Sin embargo, ningún enfoque "imperativo" sobre los lenguajes funcionales ha sido aplicado con anterioridad.

Este trabajo es un experimento para aplicar la "Cocurrencia Tradicional en Programación Funcional". Mecanismos, técnicas y sistemas de pruebas son tomados de la teoría convencional de lenguajes y aplicados al estudio de la programación funcional concurrente.

La presentación se divide en dos partes principales. Una de desarrollo, donde se presenta el estudio, comparación e implementación de un conjunto de mecanismos de sincronización y comunicación entre procesos concurrentes. En la segunda parte se estudia la aplicación de métodos formales para la verificación de programas funcionales imperativos y concurrentes.

El material presentado es el resultado de los estudios de la Licenciatura en Informática de la Facultad de Ciencias Exactas, que realicé en la Universidad Nacional de La Plata, entre los años 1993 y 1998. Los mismos fueron parcialmente financiados por una beca de la Fundación Antorchas y una pasantía del FOMEC.

El presente trabajo constituye parte del plan de trabajo a estudiar, investigar y desarrollar dentro del ámbito del grupo de teoría de la computación

del LIFIA[1]. Algunos resultados importantes obtenidos para destacar son

- SEMÁFOROS EN PROGRAMACIÓN FUNCIONAL, Pablo Mocciola. En el marco del capítulo de trabajos estudiantiles de las 26 Jornadas Argentinas de Informática e Investigación Operativa (26 JAIIO).

- MECANISMOS DE SINCRONIZACIÓN EN PROGRAMACIÓN FUNCIONAL CONCURRENTE, Javier Blanco, Pablo E. Martínez López, Pablo Mocciola. Publicado en los anales de la segunda Conferencia Latinoamericana de Programación Funcional.

- VERIFICACIÓN DE PROGRAMAS FUNCIONALES IMPERATIVOS Y CONCURRENTES, Javier Blanco, Pablo Mocciola, Demetrio Vilela. Submitido a la tercera Conferencia Latinoamericana de Programación Funcional.

- PROGRAMACIÓN FUNCIONAL PARALELA: UNA MANERA DE SEPARAR EL QUÉ DEL CÓMO, Pablo Mocciola. En el marco del capítulo de trabajos estudiantiles del IX Congreso Latino-Iberoamericano de Investigación Operativa (IX CLAIO) y las 27 Jornadas Argentinas de Informática e Investigación Operativa (27 JAIIO).

---

[1] http://www-lifia.info.unlp.edu.ar/

Índice General

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Parte I

Introducción, Objetivos y Motivación

Capítulo 1

Introducción

Históricamente la programación funcional ha sido relegada al campo teórico-matemático dentro de la comunidad informática. A partir de los nuevos desarrollos teóricos (*Functional I/O*, *Concurrent/Parallel Functional Programming*, *Strong functional Programming*, etc.) y las herramientas que estos avances han traído aparejados (*Parallel and Concurrent Haskell, Fudgets, Monads*, etc.) se han empezado a desarrollar aplicaciones sobre problemas reales. La comunidad de programación funcional manifiesta explícitamente este cambio hacia la integración de la programación funcional como una herramienta para la resolución de problemas reales. Prueba de ello son las aplicaciones realizadas sobre diferentes campos diciplinarios (Ingeniería de Software, Base de Datos, Sistemas Reactivos, Optimización Combinatoria, Biología Computacional, Multimedia, etc.), los ambientes de desarrollo para lenguajes funcionales (*debbuging*, *profiling*, *foreign-language interface*) y el énfasis puesto en cada uno de los congresos y publicaciones referentes al tema.

Siguiendo esta "nueva" línea de la comunidad, este trabajo se centrará en el estudio de la utilización de la programación funcional como una herramienta viable para la resolución y prototipación de problemas reales. En particular se centrará la atención a la especificación de programas concurrentes en programación funcional.

Un programa concurrente es un programa en el cual el orden de ejecución de sus componentes no está 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 simultáneamente. Muchas aplicaciones y/o algoritmos pueden expresarse de una manera más adecuada 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 trabajo, se utiliza la memoria compartida como for-

ma de comunicar datos y se analizan distintos mecanismos de sincronización tales como semáforos, barreras y monitores.

Los lenguajes funcionales [Bir98], 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, por ejemplo, que para evaluar una expresión de la forma $e_1 + e_2$, se pueda evaluar en paralelo e_1 y e_2, posiblemente en diferentes procesadores, y luego obtener la suma de ambos resultados. La idea anterior 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 necesidad de expresar ciertos patrones de sincronización en problemas concurrentes lleva a pensar en una forma de paralelismo más explícito [Hud91, JH93, PJGF96]; es decir, un lenguaje que posea mecanismos o "anotaciones" cuya semántica establece la manera y el lugar donde se aplica la evaluación paralela o concurrente. Por último, en [Loi98, Moc98b] se describe un enfoque novedoso e intermedio a los dos anteriores que se basa en un paralelismo semi-explícito y un *run-time system* poderoso que se encarga de administrar de manera eficiente aspectos relevantes al funcionamiento dinámico de los programas paralelos.

A partir de las diferencias reveladas por los enfoques anteriores, se realizó un trabajo [Moc98b] (paralelo a esta tésis) donde se presenta una discusión sobre la "escencia" de la concurrencia y el paralelismo con respecto al dominio de la aplicación a resolver. La reflexión descansa en la división que presentan las aplicaciones que denominamos "determinísticas" y aquellas asociadas a los sistemas "reactivos". El primer tipo de aplicaciones se asocia al enfoque de paralelismo de [Hud91, JH93] ("implícito") y [Loi98] ("semi-explícito"), donde se explotan mejor las buenas características de la programación funcional y el objetivo consiste en un mejoramiento de la performance o *speed-up*. Por otro lado, las aplicaciones "reactivas" están determinadas por un conjunto de componentes que cooperan en la resolución de alguna tarea [FPJ96]. El no-determinismo y la manipulación de estados impuestos por la concurrencia puede expresarse, en un lenguaje funcional, de manera segura preservando propiedades semánticas como la trasparencia referecial [LJ94, LJ95, LS97].

Las técnicas, mecanismos y sistemas formales concurrentes en programación funcional son estudiados a lo largo de este trabajo. Como lenguaje base para el desarrollo del trabajo se utiliza al lenguaje funcional concurrente Concurrent Haskell [PJGF96]. Para etapas de prueba y prototipación, un intérprete del lenguaje Haskell con módulos de concurrencia, llamado

HUGS [Jon98], fue utilizado.

Concurrent Haskell es una extensión concurrente del lenguaje funcional puro y lazy Haskell [PH+97]; el mismo permite expresar explícitamente, mediante una serie de primitivas nuevas de bajo nivel, mecanismos de sincronización y comunicación para el desarrollo de aplicaciones concurrentes.

1.1 Motivaciones y Objetivos

"Cocurrencia Tradicional en Programación Funcional" es un trabajo que relaciona dos importantes paradigmas, el Funcional y el Concurrente. Como su nombre lo indica, mecanismos, técnicas y sistemas de pruebas son tomados de la teoría convencional de lenguajes y aplicados al estudio de la programación funcional concurrente.

Con el advenimiento actual del modelo de entrada/salida monádico, la cáscara de más alto nivel de los programas funcionales se caracteriza por una secuencia de comandos u operaciones imperativas similares a los programas escritos en lenguajes como Pascal. Sin embargo, el verdadero poder de los lenguajes funcionales queda al margen de dicho modelo, y puede seguir utilizándose. A partir de esto nos proponemos estudiar la aplicación técnicas "imperativas" para el manejo de esa "cáscara superior", y combinarlas de alguna forma con el razonamiento ecuacional que es tan cómodo cuando se trabaja con programas funcionales puros. Lo anterior resume una de las motivaciones más importantes del trabajo. Otro punto importante es el estudio y desarrollo de herramientas, mecanismos y técnicas de sincroniazción de alto nivel para la especificación de programas funcionales concurrentes.

A forma de resumen, se presentan a continuación los objetivos de éste trabajo final, a saber:

1. estudiar los mecanismos de sincronización tradicionales usados en programación concurrente (semáforos, monitores, etc.), así también como las lógicas para la verificación de los mismos;

2. estudiar las características y particularidades de Concurrent Haskell, como así también los conceptos previos que lo fundamentan (mónadas [Wad95], entrada/salida monádica (*monadic I/O framework*) [JW93] y manipulación de estados de manera segura [LJ94, LJ95, LS97]);

3. analizar y ejemplificar la expresividad de programas concurrentes en Concurrent Haskell [BMLM97];

4. extender Concurrent Haskell para soportar monitores [Moc98a]; y

5. plantear un método de razonamiento adecuado para la verificación de programas concurrentes en programación funcional, en particular, estudiar la posibilidad de adaptar los métodos tradicionales usados para este fin.

La presentación se divide en dos partes principales. Una de desarrollo, donde se presenta el estudio, comparación e implementación de un conjunto de mecanismos de sincronización entre procesos concurrentes utilizando la memoria compartida como modelo de comunicación (capítulos 4 y 5). En la segunda parte se estudia la aplicación de métodos formales para la verificación de programas funcionales imperativos y concurrentes (capítulo 6). Previamente se realiza una breve descripción de los conceptos fundamentales estudiados sobre Programación Funcional y Programación Concurrente (capítulos 2 y 3 respectivamente). Por último, se enuncian las conclusiones finales obtenidas y los posibles trabajos futuros a investigar y profundizar (capítulo 7).

Parte II

Conceptos Fundamentales

Capítulo 2

Programación Funcional

2.1 Introducción

La programación funcional ha ido creciendo en popularidad en los últimos treinta años, desde sus comienzos con dialectos de LISP hasta hoy en día, donde se dispone de una gran variedad de lenguajes tales como Haskell, Hope, Miranda, Clean y Standrad ML (SML) entre otros [PH+97, BMS80, Tur85, Har86]. Aunque existen diferencias entre estos lenguajes, todos ellos coinciden y proveen las siguientes características.

λ-Cálculo El λ-Cálculo es el formalismo matemático/lógico que aporta las bases para el estudio de la programación aplicativa. En su versión más sencilla el λ-Cálculo provee un lenguaje de programación funcional donde todos los objetos son funciones.

Funciones de Primera Clase Las funciones son "ciudadanos" de primera clase, es decir, ellas pueden ser pasadas como argumento, ser retornadas como resultado de otra función, formar parte de estructuras de datos complejas, etc. A esto también se lo conoce como funciones de alto orden. Un ejemplo es la función *map*, que toma una función f como argumento y retorna la función que toma una lista x_s como argumento y retorna la lista resultante de alpicar f a cada elemento de la lista x_s.

Sistemas de tipado fuerte El lenguaje contiene mecanismos de distinción entre diferentes valores, clasificándolos mediante un sistema de tipos. El tipado de las expresiones restringe la aplicación de operadores y constructores de datos, lo que permite detectar errores en tiempo de compilación, como por ejemplo que una expresión booleana sea aplicada al operador de suma. Además, la condición de tipado fuerte (*strong*

typing) determina que todos los errores de tipos queden determinados en tiempo de compilación.

Tipos polimórficos Utilizando un sistema de tipos polimórficos como el sistema de tipos de Hindley-Milner [Jon95] se permite que una función genérica, como la función identidad, sea instanciada con cualquier tipo. Por ejemplo, el tipo de la función identidad se denota como $\alpha \rightarrow \alpha$, indicando que el tipo de la función es un tipo funcional (una función), donde el rango de valores del dominio es igual al del codominio y está determinado por la variable de tipo α. Es decir, que si se utiliza sobre booleanos, retorna booleanos, si se aplica sobre funciones numéricas, retorna funciones numéricas, etc. Esto determina un factor importante con respecto a la reusabilidad de código que tienen los lenguajes funcionales.

Tipos algebraicos Listas, árboles y otros tipos pueden ser definidos directamente mediante definiciones recursivas, en vez de utilizar indeseables definiciones con punteros. El análisis de casos y selección de los componentes del tipo son realizados mediante *pattern matching*, guardas o expresiones **if** y **case**, dependiendo del lenguaje.

Modularidad El lenguaje ofrece un sistema de módulos y bibliotecas para facilitar el desarrollo de sistemas de considerable magnitud.

Dentro de las diferencias entre los lenguajes funcionales mencionados anteriormente, existen dos muy importantes que dividen a los lenguajes funcionales en dos grupos, por un lado se clasifican en puros e impuros, y por el otro, en estrictos y perezosos. Dichas clasificaciones estan determinadas, respectivamente, por la disponibilidad o nó de la asignación y el mecanismo de evaluación subyacente.

En este trabajo se utiliza el lenguaje Haskell [PH+97, Hud92, Tho96, Dav92] como lenguaje funcional de estudio y extensiones particulares del mismo para soportar concurrencia. El lenguaje Haskell, llamado así en honor al matemático Haskell Curry, es un lenguaje funcional que pertenece al grupo de los lenguajes funcionales puros y perezosos. La elección del mismo se debe a las propiedades y características que los lenguajes de esta categoría agregan a las características anteriores, a saber:

Transparencia Referencial La existencia de un mecanismo de asignación en un lenguaje liga al mismo a una arquitectura subyacente que adolece con el problema del cuello de botella de Von Neumann. La actualización de un estado mediante expresiones que realizan asignaciones

(efectos laterales) implica determinar a priori un orden de ejecución de dichas expresiones. Por otro lado, si solo se permiten expresiones en las cuales no pueden ocurrir efectos laterales, el valor de una expresión solo dependerá del valor de sus subexpresiones. La propiedad anterior se conoce como transparencia referencial, y permite que los programas funcionales puedan razonarse como funciones matemáticas (razonamiento ecuacional).

Evaluación perezosa Bajo este tipo de evaluación, los argumentos de las funciones y los componentes de los tipos de datos sólo son evaluados cuando y si son necesarios. Esto permite un estilo de programación basado en objetos infinitos, estructuras de datos parcialmente definidas, procesamiento paralelo mediante *pipeline* de funciones, etc.

Amr Sabry realiza en [Sab98] un estudio y una definición más formal acerca de la clasificación de los lenguajes funcionales en puros e impuros. De allí,

> un lenguaje funcional se considera puro si (i) incluye a todos los términos del λ-Cálculo tipado, y (ii) sus implementaciones *call-by-name*, *call-by-need* y *call-by-value* son equivalentes (módulo divergencia y errores).

Por motivos de espacio y alcance del trabajo, no se proporciona una completa introducción a la programación funcional, ni tampoco al lenguaje Haskell. Para aspectos introductorios a la programación funcional se recomienda ver [Bir98, Mac90], y en particular, información sobre el lenguaje Haskell puede encontrarse en [Dav92, Tho96, PH+97].

Otra importante característica del lenguaje Haskell es el concepto de mónada [Wad95]. Avances teóricos provenientes de la teoría de categorías han presentado a las mónadas como un mecanismo adecuado para razonar de manera segura sobre computaciones con estados [Wad95], variables mutables [LJ94, LJ95, LS97], entrada/salida monádica (*monadic I/O*) [JW93] y procesamiento concurrente [PJGF96]. Una característica importante del estilo monádico es que permite escribir programas funcionales que imitan o se asemejan a los programas escritos en lenguajes imperativos tales como C y Pascal. En las secciones siguientes, se describen con cierto grado de detalle, algunos de los conceptos anteriores que resultan fundamentales a lo largo del desarrollo del trabajo.

2.2 Mónadas

Se puede definir a las mónadas como una familia de tipos de datos abstractos que encapsulan estados, y efectos sobre ellos, 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,

$$\begin{aligned} &\mathit{return} :: \alpha \to m\ \alpha \\ &(\rhd) \quad :: m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta \end{aligned}$$

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 ($\rhd$), llamado *bind* y cuya sintáxis en Haskell es `>>=`, puede ser pensado como una manera de secuenciar efectos o computaciones. Es decir, $m \rhd f$ 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 $(m \rhd f)$. El operador $\gg$ es un caso particular de la función $\rhd$ donde no se tiene en cuenta el resultado de la computación intermedia (ver figura 2.1).

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 $e_1 \rhd \lambda var \to e_2$ se escribe $\mathbf{do}\{var \leftarrow e_1; e_2\}$ y la expresión $e_1 \gg e_2$ se escribe $\mathbf{do}\{e_1; e_2\}$. 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, $\mathbf{do}\{e_1; \ldots; e_n\}$ es equivalente a

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

2.2.1 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".

```
    — Combinators
(≫)        :: IO α → IO β → IO β        — descarta el resultado de m
m ≫ n      = m ▷ λ_ → n
sequence   :: [IO α] → IO ()            — secuencia efectos
sequence   = foldr (≫) skip
    — Actions
hGetChar   :: Handle → IO Char          — lee un caracter desde un disp.
hPutChar   :: Handle → Char → IO ()     — escribe un caracter en el disp.
    — other important actions
skip       :: IO ()                     — no action
skip       = return ()
print      :: String → IO ()            — imprime st en la salida std.
print st   =
  sequence (map (hPutChar stdout) st)
while      :: IO Bool → IO α → IO ()
while cond body =
  do b ← cond
     if b then do _ ← body
                  while cond body
          else skip
```

Figura 2.1: Acciones y operadores monádicos de I/O

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 una posible definición del tipo que representa lo dicho anteriormente quedaría como: **type** $IO\ \alpha = World \rightarrow (\alpha, World)$

Llamaremos acciones o computaciones a las expresiones de tipo $IO\ \alpha$. Además de las acciones monádicas vistas, usaremos algunas de las acciones y operadores definidos en la figura 2.1.

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

2.3 Concurrent Haskell

Concurrent Haskell [PJGF96] es una extensión concurrente ya implementada y disponible gratuitamente del lenguaje funcional Haskell [PH+97]. 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 $forkIO :: IO() \rightarrow IO()$. 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*. El mismo imprime secuencias infinitas de "*Concurrent*"s y "*Continuation*"s intercaladas en un orden no estipulado.

```
let loop st = print st >> loop st      -- 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 [PJGF96, LJ95, 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 α = ...

newVar    :: α → IO(MutVar α)
readVar   :: MutVar α → IO α
writeVar  :: MutVar α → α → IO ()
```

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

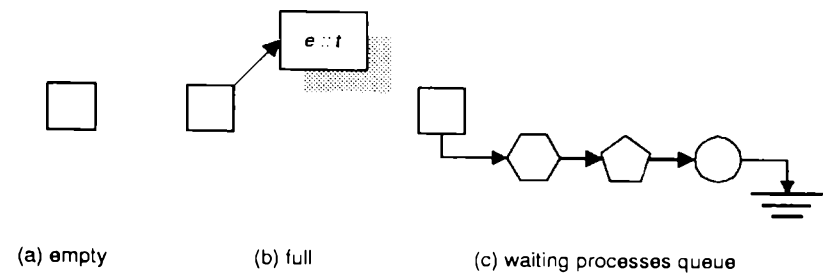

Figura 2.2: Posibles estados de una $MVar\ t$.

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. El tipo primitivo que caracteriza a las variables mutables sincronizadas es **type** $MVar\ \alpha$. 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

- $newMVar\ ::\ IO\ MVar$, crea una nueva $MVar$,
- $takeMVar\ ::\ MVar\ \alpha \rightarrow IO\ \alpha$, se bloquea mientras la posición este vacía y luego lee y retorna el valor, dejando la posición nuevamente vacía, y
- $putMVar\ ::\ MVar\ \alpha \rightarrow \alpha \rightarrow IO()$, 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.

Una interpretación gráfica de esto se muestra en la figura 2.2. Como comentario final sobre las $MVar$s, cabe mencionar las diferentes maneras en que éstas pueden ser interpretadas: 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 más adelante.

Es interesante destacar que dos procesos concurrentes creados con la función *forkIO* 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.

Capítulo 3

Programación Concurrente

3.1 Introducción

Un programa concurrente es un programa en el cual el orden de ejecución de sus instrucciones no está completamente determinado en el texto de éste. Generalmente, un programa concurrente define dos o más procesos que cooperan en la realización de una tarea. Cada proceso es un programa secuencial que ejecuta una secuencia de sentencias. Los procesos cooperan comunicándose usando o bién variables compartidas, o bién pasaje de mensages. El proceso de comunicación motiva la necesidad de sincronización entre los procesos. La ejecución de un programa concurrente genera una secuencia de acciones atómicas de los distintos componentes. Cuando los procesos interactúan, no todas las posibles formas de intercalar dichas acciones atómicas son aceptables o deseadas. El rol de la sincronización entre procesos es evitar dichas configuraciones indeseadas. Existen dos tipos importantes de sincronización: exclusión mutua y sincronización por condición. La primera de ellas implica que determinados objetos compartidos solo pueden ser manipulados por un proceso a la vez. Por otro lado, la sincronización por condición determina que un proceso debe demorarse hasta que se cumpla determinada condición.

Muchas aplicaciones, en las cuales el orden de ejcución de algunas sentencias es irrelevante, son más fáciles de expresar con un lenguaje concurrente. Algunos ejemplos donde se utilizan programas concurrentes son los sistemas reactivos, manejo de recursos en un sistema operativo, protocolos de comunicación sobre redes, computaciones sobre arreglos y matrices, etc. Los programas de la última clase son usualmente llamado programas paralelos.

Los programas concurrentes son inherentemente más complejos que los programas secuenciales. Además, los programas concurrentes son difíciles de diseñar, y los errores son más una regla que una excepción. La justificación

informal, o mediante un razonamiento operacional (análisis de todas las posibles computaciones), de que un programa concurrente satisface determinada propiedad no es suficiente, ni mucho menos práctico. Es por esto que se vuelve aún más necesario que en la programación secuencial el uso de un enfoque sistemático de demostración o verificación de programas concurrentes.

Las herramientas formales se basan en el razonamiento mediante aserciones, también conocido como razonamiento axiomático. El estado de un programa se caracteriza por un predicado lógico llamado asersión lógica. Mediante un conjunto de axiomas y reglas que definen un sistema de demostración se puede razonar sobre un programa concurrente y probar propiedades del mismo.

Se han utilizado un número considerable de enfoques para la verificación de programas tanto secuenciales como concurrentes. Entre los más importantes se encuentran el método de axiomatización de Floyd [Flo67] para verificación de diagramas de flujo, el sistema de demostración de Hoare [Hoa69, Apt81] para lenguajes con estructura **while**, la teoría para programas concurrentes y paralelos de Owicki y Gries [OG76], lógicas temporales [Pnu77], estudio de técnicas automáticas de *model checking*, etc.

En este capítulo introductorio, sólo se presenta una breve descripción de los conceptos fundamentales sobre programación concurrente, utilizados a lo largo del trabajo. En este trabajo se utiliza principalmete el paradigma de memoria compartida como mecanismo de comunicación entre procesos concurrentes. Por ello, un análisis y descripción de algunos de los mecanismos de sincronización entre procesos (semáforos y monitores) que utilizan tal paradigma de comunicación es presentada a continuación.

También se describen nociones elementales sobre verificación de programas secuenciales y concurrentes. Para una descripción más detallada se recomienda ver [AO97, And91, Apt81, Mar98].

3.2 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.

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 y la de V son:

$$\begin{array}{ll} P(s): & < \textbf{await}\ (s > 0) \rightarrow\ s := s - 1 > \\ V(s): & < s := s + 1 > \end{array}$$

donde $< S >$ indica que la sentencia S es ejecutada de manera atómica, y la sentencia **await** $c \rightarrow s$ determina que para ejecutar s se debe esperar hasta que la condición c sea verdadera. El invariante del semáforo general es $s \geq 0$.

Un caso particular y más sencillo son los semáforos binarios (*binary semaphore*), en los cuales su valor interno puede ser únicamente 0 ó 1. Con este tipo de semáforos se pueden expresar y solucionar una gran variedad de problemas, entre los cuales podemos mencionar exclusión mutua, grafo de precedencias, escritores/lectores, etc.

Además, a partir de semáforos binarios y variables enteras, y usando la técnica de los *split-binary semaphores* (SBS) se pueden implementar semáforos generales [Dij79, Dij80, And91]. Los SBS están formados por n semáforos binarios que, para cualquier programa, siempre cumplen el invariante global: $SPLIT : 0 \leq\ b_1 + \ldots + b_n \leq\ 1$, donde b_i 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. Dicha técnica también es conocida como *passing the baton*, y se detalla a continuación.

3.2.1 Passing the baton

La técnica de *passing the baton* (pasando la "posta") [And91] es utilizada para implementar sentencias atómicas de la forma

$$F_1 :< S_i > \qquad \text{o} \qquad F_2 :< \textbf{await} Bj \rightarrow S_j >$$

mediante semáforos binarios partidos y variables enteras. La exclusión mútua y condición de sincronización de este tipo de sentencias es garantizada de la siguiente manera. Definir un semáforo binario e, inicializado en 1, para controlar el acceso exclusivo en las acciones atómicas. Luego, asociar un semáforo binario b_j y un contador d_j a cada guarda B_j de las sentencias atómicas del tipo F_2.

Notar que en la traducción de las sentencias atómicas del tipo F_1 y F_2 en un programa concurrente, el conjunto de semáforos binarios (e y todos los b_j) trabajan como un semáforo binario partido. Las sentencias del tipo F_1 son reemplazadas por el siguiente fragmento de programa

F_1 : *wait* e
S_i
SIGNAL

Mientras que, las sentencias del tipo F_2 son reemplazadas por el siguiente código

F_2 : *wait* e
if *not* $B_j \rightarrow d_j := d_j + 1$; *signal* e; *wait* b_j **fi**
S_j
SIGNAL

donde d_j cuenta la cantidad de procesos esperando por la condición B_j en el semáforo b_j, y SIGNAL se define como el siguiente condicional

SIGNAL : **if** B_1 *and* $d_1 > 0 \rightarrow d_j := d_j - 1$; *signal* b_1
$\square \ldots$
$\square$ B_n *and* $d_n > 0 \rightarrow d_n := d_n - 1$; *signal* b_n
else $\rightarrow$ *signal* e
fi

Las primeras líneas de SIGNAL chequean si hay algún proceso esperando por alguna condición que ahora es verdadera. La guarda **else** libera la exclusión mútua de la ejecución de la sentencia atómica.

La técnica se denomina "pasando la posta" por la manera en que los semáforos se van activando. Cuando un proceso esta ejecutando su región crítica, tiene el permiso de ejecución, posee la "posta". Cuando el proceso termina su ejecución (SIGNAL) pasa la posta a algún proceso demorado que ahora puede continuar o al siguiente proceso que entra a su sección crítica por primera vez.

3.3 Monitores

Los monitores son un mecanismo de sincronización entre procesos que utiliza el paradigma de comunicación de memoria compartida [Hoa74, How76, How, And91]. Los monitores encapsulan la representación abstracta de un recurso compartido. El acceso exclusivo, a dicho recurso, es garantizado implícitamente por la semántica propia del monitor, y solo se puede llevar a

cabo por medio de las operaciones que este brinda. Los monitores son, de alguna manera, similares a las regiones críticas. Sin embargo, logran una mayor estructuración y modularización en la especificación de los programas concurrentes, y además, pueden implementarse eficientemente.

Si asumimos que todos los recursos compartidos son manipulados únicamente utilizando monitores, dos o más procesos pueden interactuar solamente llamando a las operaciones de dicho monitor. La modularización resultante rescata una serie de buenas características que los monitores heredan de los tipos de datos abstractos. Primero, un proceso que llama a una operación del monitor puede ignorar la implementación de la misma; lo único que importa son los efectos visibles de la invocación. Segundo, el programador del monitor puede ignorar como o donde las operaciones del monitor serán usadas. Además, una vez que el monitor es implementado correctamente, su correctitud no varía, idependientemente del número de procesos que accedan al monitor. De la misma manera, el programador puede variar la implementación del monitor, y que esto resulte transparente para los usuarios, siempre y cuando no cambie la especificación del mismo. Todo esto junto facilita el desarrollo y legibilidad de los programas concurrentes.

Por otro lado, lo que distingue a los monitores de un mecanismo de abstracción de datos en un lenguaje de programación secuencial es que el monitor es compartido concurrentemente por varios procesos. Dichos procesos pueden requerir exclusión mutua, para evitar interferencias, y sincronización de acuerdo a alguna condición, esperar que el monitor cumpla cierta condición. La exclusión mutua es garantizada implícitamente por los monitores, mientras que la condición de sincronización es provista por un mecanismo de bajo nivel, similar a los semáforos, conocidos como variables de condición (*condition variables*). Las mismas se detallan en el apartado 3.3.2 de este capítulo.

3.3.1 Notación

Un monitor encapsula un recurso a ser compartido y provee una serie de operaciones que implementan dicho recurso y permiten manipularlo. La declaración de un monitor tiene la siguiente forma:

monitor *Mname* **where**
  var *declarations of resources*
  cond *condition variables*
  init *params* = *initialization code*
  procedure op_1 ($params_1$)
  ...
  procedure op_n ($params_n$)

endmonitor

Las variables del monitor representan el estado actual del monitor, las variables de condición sirven para sincronizar a los procesos que utilizan el monitor, el proceso de inicialización fija el estado inicial del monitor al momento en el cual este se crea, y por último los procedimientos son las operaciones mediante las cuales se puede acceder y manipular el recurso compartido representado en el monitor.

Un punto importante, que no es especificado en la declaración sintáctica del monitor, es la definición de un *invariante* del monitor que refleja, mediante un predicado, el estado que el monitor debe cumplir al momento que ningún proceso lo está accediendo. El código de inicialización debe establecer dicho invariante y las opraciones deben mantenerlo.

3.3.2 Sincronización: Variables de condición

Una variable de condición se utiliza para demorar a un proceso que no puede continuar su ejecución hasta que el estado del monitor satisfaga determinada propiedad o condición. De la misma manera, se utiliza para despertar a los procesos demorados cuando la condición pasa a ser verdadera. Las variables de condición se declaran utilizando la palabra reservada **cond** seguida de las variables de condición usadas en el monitor.

Las variables de condición son un tipo de datos abstracto que encapsulan una cola de procesos esperando por determinada condición. Ellas solo pueden ser declaradas y usadas dentro de los monitores. Las operaciones permitidas sobre una variable de condición c son:

- *wait* c, si un proceso busca demorarse en la condición determinada por c, y
- *signal* c para indicar que la condición c se satisface y eventualmente un proceso previamente demorado puede ser despertado.

La ejecución de *wait* causa que el proceso sea demorado al final de la cola de procesos de la variable de condición c. Además se libera la exclusión mutua del monitor para que otros procesos puedan accederlo. El funcionamiento de *signal* depende de la semántica asociada al mismo. En este caso se considera la semántica *signal-and-continue*. Existen otras disiplinas alternativas equivalentes de *signal*, a saber:

Automatic Signaling Es caracterizada por la operación **await**, se realiza de manera implícita y el proceso despertado no se apropia del recurso del monitor (*no-preemptive*).

Signal and Continue Explícita y *no-preemptive*, es decir, el proceso que realiza la operación de *signal* sigue ejecutándose (ver más adelante).

Signal and Exit Explícita y prioritaria, es decir, el proceso que realiza la operación de *signal* debe abandonar el monitor inmediatamente.

Signal and Wait Explícita y prioritaria, además, el proceso que realiza la operación de *signal* debe esperar para continuar su ejecución.

Signal and Urgent Wait Similar al anterior, pero el proceso que realiza la operación de *signal* y espera, adquiere el acceso al monitor inmediatamente despues de que la ejecución del proceso despertado haya terminado.

Entonces, el funcionamiento de *signal* c utilizando *signal-and-continue* consiste en despertar el proceso ubicado al frente de la cola de procesos demorados de c, en caso de que esta no sea vacía. Si la cola de c está vacía, la ejecución de *signal* no tiene ningún efecto, es decir, es equivalente a *skip*. Independientemente de lo anterior, el proceso que ejecuta *signal* mantiene el control de exclusión del monitor y continua con su ejecución, mientras que el proceso despertado por la invocación de *signal* se ejecutará en el momento en que pueda acceder al monitor, es decir, logre el control de exclusión.

Se puede definir una serie de operaciones adicionales sobre las variables de condición las cuales permiten especificar determinados problemas de una manera más sencilla. A saber, ellas son:

- *empty* c, determina si existe, o no, al menos un proceso demorado en la cola de la variable de condición c.
- *p_wait* c *rank*, es un *wait* con prioridad que de acuerdo a *rank* fija la posición del proceso dentro de la cola de demorados de c.
- *minrank* c, devuelve el valor mínimo de la prioridad de los procesos demorados en c. Si c está vacía devuelve algún valor arbitrario.
- *signal_all* c, despierta a todos los procesos demorados en la cola de c (*broadcast*).

3.4 Verificación de programas

La verificación de programas es un enfoque formal y sistemático para probar la corrección de programas, donde correctitud significa que el programa safisface cierta propiedad deseada. Para programas secuenciales estas propiedades

se refieren a la obtención de resultados correctos y a la terminación. Para programas concurrentes, aquellos con muchos componentes activos, algunas propiedades importantes son la de no-interferencia, ausencia de *deadlock* y funcionamiento justo (*fair*). Tales propiedades suelen dividirse en *safety and liveness properties.*

3.4.1 Sistema de demostración formal

Un sistema de demostración o un cálculo $\mathcal{P}$ sobre un conjunto Φ de fórmulas es un conjunto finito de esquemas de axiomas y reglas de demostración o inferencia. Un esquema de axioma $\mathcal{A}$ es un subconjunto decidible de Φ. Para describir un esquema de axioma se utiliza la siguiente notación.

$$\mathcal{A}: \qquad \varphi \qquad \text{where "..."}$$

Donde φ denota la fórmula considerada como válida en el sistema de demostración.

Con la ayuda de las reglas de inferencia más formulas pueden ser deducidas. Las reglas de demostración se anotan como

$$\mathcal{R}: \quad \frac{\varphi_1, \ldots, \varphi_n}{\varphi} \quad \text{where "..."}$$

Intuitivamente, la regla $\mathcal{R}$ dice que a partir de las fórmulas $\varphi_1, \ldots, \varphi_n$ la fórmula φ puede ser deducida si la condición "..." vale. Las $\varphi_1, \ldots, \varphi_n$ son llamadas las hipótesis o premisas, y φ la conclusión.

Luego, una demostración de una fórmula φ en un sistema de demostración $\mathcal{P}$ es una secuencia finita de fórmulas

$$\begin{array}{c} \varphi_1 \\ \vdots \\ \varphi_k \end{array}$$

que satisfacen las siguientes propiedades

- $\varphi = \varphi_k$
- φ_i con $i \in \{1, \ldots, k\}$ es, o bien un axioma de $\mathcal{P}$, o bien puede ser obtenido por la aplicación de una regla de demostración $\mathcal{R}$ de $\mathcal{P}$. En particular, la primera fórmula φ_1 es un axioma.

Además, la última fórmula φ de una demostración es denominada teorema del sistema de demostración $\mathcal{P}$. Para un sistema de demostración $\mathcal{P}$ dado, y una fórmula φ, se escribe $\vdash_{\mathcal{P}} \varphi$, si φ es un terorema de $\mathcal{P}$.

3.4.2 Modelos de verificación

Informalmente, un programa determinístico es correcto si satisface la relación de pre y poscondición. Las técnicas de verificación de programas de Hoare [AO97, Apt81] permiten demostrar que dado un programa imperativo S, si éste comienza en un estado que satisface una precondición $\{P\}$ termina en un estado que satisface una poscondición $\{Q\}$. Tradicionalmente la notación usada para esto es la de ternas de Hoare $\{P\}S\{Q\}$, donde P y Q son predicados sobre las variables de S que corresponden a la pre y poscondición respectivamente. Sea un sistema de demostración formal $\mathcal{P}$ correcto y completo para el lenguaje que describe a S, una terna de Hoare $\{P\}S\{Q\}$ es válida si y solo si es teorema del sistema de demostración $\mathcal{P}$.

Otra notación o metodología utilizada para la verificación de programas, es la precondición más débil (*weakest preconditions*). La precondición más débil de un programa S y un predicado Q, denotado $wp(S, Q)$, es el predicado que caracteriza al conjunto más grande tal que, si la ejecución de S es comenzada en cualquier estado que satisface $wp(S, Q)$, entonces se garantiza que la ejecución termina en un estado que satisface Q. Nuevamente, la precondición más débil debe definirse sobre la estructura del lenguaje que determina a S.

Se puede ver que demostrar $P \Rightarrow wp(S, Q)$ es equivalente a probar la veracidad de la terna de Hoare $\{P\}S\{Q\}$. Existen distintas formulaciones equivalentes de las reglas para demostrar la corrección de los programas. La ventaja de alguna de éstas es meramente notacional. En particular, las dos notaciones anteriores son muy utilizadas y su aplicación depende de las características del problema a tratar.

En el caso de los programas concurrentes, los componentes se anotan con aserciones de manera similar a los programas secuenciales, aunque hay más obligaciones de prueba. Un programa concurrente tiene además una precondición y, en caso de que todos sus componentes termine, una poscondición. La teoría de Owicki y Gries [OG76] dice que una anotación es correcta si:

- es localmente correcta, es decir, si es establecida por ese componente sin tener en cuenta al resto de los componentes,
- globalmente correcta, es decir, si no es falsificada por ninguna sentencia atómica de otro componente (también conocido como no-interferencia),
- la precondición de cada componente debe ser implicada por la precondición del programa concurrrente, y
- la conjunción de las poscondiciones de los componentes debe implicar la poscondición del programa concurrente (si esta existe).

Un concepto muy importante utilizado en la verificación de programas es el concepto de invariante. La importancia de los invariantes deriva del hecho de que es solo necesario verificar su corrección local. También son de mucha utilidad en la demostración de programas el uso de variables auxiliares. Con las mismas se pueden reflejar estados o situaciones de un programa que facilitan la demostración.

En los capítulos posteriores se plantea un método de razonamiento adecuado para la verificación de programas concurrentes en programación funcional, en particular, se estudia la posibilidad de adaptar los métodos tradicionales mencionados anteriormente aquí y usados para tal fin.

Parte III

Mecanismos de Sincronización en Programación Funcional Concurrente

Capítulo 4

Semáforos y Barreras

4.1 Introducción

Concurrent Haskell es un extensión del lenguaje funcional Haskell para que soporte concurrencia (ver sección 2.3). El mismo provee una serie de estructuras primitivas que permiten la creación de procesos concurrentes (*forkIO*) y la sincronización y comunicación entre ellos (*MVar*s). Sin embargo, estas estructuras primitivas son de muy bajo nivel para expresar problemas de concurrencia de una manera sencilla.

Uno de los objetivos de este trabajo es extender el conjunto de herramientas primitivas de concurrencia, e implementar nuevos mecanismos de comunicación y sincronización para el desarrollo de programas concurrentes funcionales.

Esta etapa comienza con la especificación e implementación de los distintos tipos de semáforos (binarios, binarios partidos, y generales) [Dij79, Dij80, And91]. Además se presenta una implementación de barreras, las cuales son útiles para resolver problemas de sicronización de fase, presentes por ejemplo en lo algoritmos paralelos sobre matrices. La descripción de sendas herramientas de sincronización se acompaña con ejemplos realizados en Concurrent Haskell.

4.2 Semáforos

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.

4.2.1 Semáforos Binarios

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
newSem   = newEmptyMVar

signal   :: Sem → IO()
signal s = putMVar s ()

wait     :: Sem → IO()
wait s   = takeMVar s
```

4.2.2 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]. Como se mencionó en la sección 3.2, los SBS están formados por n semáforos binarios que, para cualquier programa, siempre cumplen el invariante global: $SPLIT : 0 \leq b_1 + \ldots + b_n \leq 1$, donde b_i es el valor del semáforo i-ésimo.

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). La implementación final en funcional de las operaciones *wait* y *signal* para semáforos generales se muestra en la figura 4.1. Para ver el proceso de construcción completo, se recomienda ver [Dij79, Dij80].

Nótese en este 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)

newG :: Int → IO GSem
newG 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 do{signal m; wait s}
                     signal m
```

Figura 4.1: Semáforos generales.

En [PJGF96] 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 trabajo 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.

4.2.3 Ejemplos

Una buena manera de entender las estructuras concurrentes presentadas es mediante ejemplos. A continuación se describen los ejemplos de sincronización de procesos, lectores/escritores (*readers/writers*) y productores/consumidores (producers/consumers).

Sincronización de procesos

El primer ejemplo que veremos es el de sincronización de procesos o tareas. Una planificación de tareas queda definida con un mapa o grafo de precedencia. Un grafo de precedencia es un grafo acíclico y dirigido. Los nodos representan las tareas o procesos, y los arcos indican el orden en el cual las tareas se llevan a cabo. En particular, un proceso puede ejecutarse sí y sólo sí todos sus predecesores lo han hecho. En la figura 4.2 se puede ver el grafo de procesos utilizado como ejemplo y los semáforos asociados a las aristas.

Primero se muestra la implementación del ejemplo utilizando una notación de un lenguaje concurrente imperativo, donde **co** $S_1 \parallel \ldots \parallel S_n$ **oc** ejecuta S_i concurrentemente.

Var $a, b, c, d, e, f, g : sem := 0$
begin co
  begin S_1; $V(a)$; $V(b)$ **end**
  begin $P(a)$; S_2; S_4; $V(c)$; $V(d)$ **end**
  begin $P(b)$; S_3; $V(e)$ **end**
  begin $P(c)$; S_5; $V(f)$ **end**
  begin $P(d)$; $P(e)$; S_6; $V(g)$ **end**
  begin $P(f)$; $P(g)$; S_7 **end**
oc end

A continuación se escribe el ejemplo anterior en Concurrent Haskell. La sentencia **co** que especifica ejecución concurrente, puede ser implementada utilizando la operación *forkIO*.

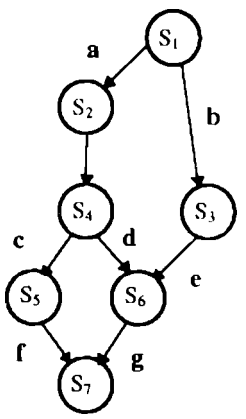

Figura 4.2: Grafo de procesos a modelar

```
main :: IO ()
main = do {a ← newSem; ...; g ← newSem;    — sems' initiation
           co [ do {S1; signal a; signal b},
                do {wait a; S2; S4; signal c; signal d},
                do {wait b; S3; signal e},
                do {wait c; S5; signal f},
                do {wait d; wait e; S6; signal g},
                do {wait f; wait g; S7}]
          }
co = sequence . map forkIO
```

Este ejemplo puede generalizarse fácilmente en programación funcional, ya que teniendo representado el grafo de precedencia, que indica la sincronización de los procesos, se puede especificar una función de sincronización de alto orden cuyo resultado sea el programa que evalúa concurrentemente dicha planificación.

Lectores y Escritores

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 procesos lectores ejecutan transacciones que sólo examinan los registros de la base de datos; los escritores pueden también actualizar la base de datos. Las transacciones de los procesos escritores deben realizarse en forma exclusiva para garantizar la integridad de la base de datos. Si ningún proceso escritor está accediendo a la base de datos, entonces los procesos lectores 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 la sección 3.2.1) 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.

Los SBS se representan como una terna de semáforos binarios (m, r, w). El primer semáforo binario de la terna se utiliza para asegurar la exclusión mútua, mientras que los otros dos controlan el invariante de sincronización entre los lectores y escritores. El número de lectores y escritores accediendo a la base de datos (nr, nw), como así también el número de lectores y escritores demorados en el acceso (nr, nw), se representan mediante un par de variables mutables enteras.

```
type SBS    = (Sem, Sem, Sem)
type IntVar = MutVar Int
type Vars   = (IntVar, IntVar)
```

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. Notar que la función *doTrue* simula un bucle infinito; toma una computación monádica como argumento y la repite indefinidamente.

```
reader :: SBS → Vars → Vars → IO ()
reader (m, r, w) (dr, dw) (nr, nw) =
  doTrue (do wait m
             br ← guardR
             if not br then do inc dr
                               signal m
                               wait r
                    else skip
             inc nr
             signalSBS
             < readDataBase >    — sección crítica
             wait m
             dec nr
             signalSBS )
```

La función *signalSBS* (definida abajo) se encarga de otorgarle el turno (*the baton is passed*) a algún proceso que se encuentre esperando por una condición que ahora es verdadera. Por ejemplo, si no hay un proceso escritor accediendo a la base de datos, y hay por lo menos un proceso lector esperando, este es habilitado para acceder al recurso. Cuando no hay procesos esperando por una condición, se permite entrar a procesos nuevos (último **else**, se libera el semáforo de la exclusión mútua).

```
where
  signalSBS = do
      valdr ← readVar dr
      valdw ← readVar dw
      br ← guardR          — br ⇔ no hay escritores
      bw ← guardW          — bw ⇔ no hay esc. ni lec.
      if (br ∧ valdr > 0)  — hay algún lector esperando y puede
          then do dec dr   — ser despertado
                  signal r
          else
      if (bw ∧ valdw > 0)  — hay algún esc. esperando y puede
          then do dec dw   — ser despertado
                  signal w
      else signal m
```

Por último, las funciones *guardR* y *guardW* determinan si se cumple la condición de acceso para los lectores y escritores, respectivamente. Es decir, si no hay ningún escritor activo y si el recurso no esta siendo accedido por ninguno de los dos tipos de procesos.

```
where
  guardR  = do valnw ← readVar nw
               return (valnw == 0)
  guardW  = do valnw ← readVar nw
               valnr ← readVar nr
               return (valnw == 0 ∧ valnr == 0)
```

En esta implementación en particular se le dá prioridad a los lectores, aunque ésto puede cambiarse trivialmente modificando la función *signalSBS*.

Productores y Consumidores

El problema de los productores y consumidores (*producers and consumers*) presentado en esta sección es uno de los problemas clásicos en programación concurrente [And91]. Dos tipos de procesos (consumidores y productores)

```
deposit :: Buffer α → GSem → GSem → IO α
deposit buffer full empty =
  do waitG empty
     info ← getFrom buffer
     signalG full
     return info

fetch :: α → Buffer α → GSem → GSem → IO ()
fetch info buffer full empty =
  do waitG full
     putInto buffer info
     signalG empty
```

Figura 4.3: Productores/Consumidores utilizando semáforos generales.

se ejecutan concurrentemente compartiendo un *buffer* limitado. Los procesos productores almacenan los datos en el *buffer* (operación *deposit*), y los procesos consumidores los sacan de allí (operación *fetch*). La condición o invariante que debe asegurar un programa que modelice este problema es, por un lado, que los procesos del tipo productor no pongan datos en el *buffer* si éste se encuentra lleno, y por el otro, que procesos consumidores no saquen datos del *buffer* si se encuentra vacio.

Dentro de las distintas variantes del problema, el caso más general es cuando existen múltiples procesos consumidores y productores, y el *buffer* puede almacenar más de un elemento. El problema de los productores/consumidores ha sido implementado utilizando distintas técnicas de sincronización y comunicación [And91]. En [PJGF96] se utilizan estructuras particulares del lenguaje construidas a partir de las *MutVar*s para resolver dicho problema. En esta parte del trabajo se muestra una implementación utilizando los semáforos generales definidos anteriormente (sección 4.2.2). Otra opción, es utilizar la técnica de los SBS, de manera similar que en el problema de los lectores/escritores. En el capítulo 5 se detalla una solución al problema utilizando monitores en $\mathcal{M}$-Haskell.

La solución del problema, descripta en la figura 4.3, utiliza dos semáforos generales *full* y *empty* inicializados en 0 y n respectivamente, con n igual al tamaño del *buffer*. Además, ellos controlan, respectivamente, la cantidad de lugares ocupados y libres del *buffer*. Es importante notar que el buffer es accedido con exclusión mútua. Esto último puede implementarse fácilmente mediante un semáforo binario. De todas maneras, esta característica

es transparente para las operaciones *fecth* y *deposit* quedando oculta en la implementación del *buffer*.

El invariante que expresa la cantidad de procesos escritores y lectores que pueden acceder a la vez de manera correcta al *buffer* está dado por el valor de los semáforos generales *full* y *empty*, a saber

$$empty \geq 0 \wedge full \geq 0$$

4.3 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 sección 4.2.

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 Concurrent 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 semáforos 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 semáforos
                        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)   — i-ésimo semáforo
```

En este punto, cabe destacar la importancia de representar a los mecanismos de sincronización 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.

4.3.1 Sumas Parciales

Un algoritmo paralelo de datos es un algoritmo 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 (*parallel prefix compuations* PPC) [And91]. Las PPC son útiles en muchas aplicaciones, incluyendo procesamiento de imágenes, computaciones de matrices y parsing de lenguajes regulares. La especificación del problema de las sumas parciales se entiende como dado un arreglo $a[1 : n]$ calcular $sum[1 : n]$, donde $sum[i]$ es la suma de los primeros i elementos de a. Abajo se define la función *sumas*, que resuelve el problema en cuestión, utilizando la función general *parallelPrefixC*.

$$sumas \;=\; parallelPrefixC\ 0\ (+)$$

El ejemplo anterior utiliza la suma matemática como operador binario y el 0 como elemento neutro de la misma, sin embargo el algoritmo básico puede ser usado para cualquier operador binario y su correspondiente elemento neutro. A continuación se muestra una implementación para el problema PPC. La función *parallelPrefixC* toma como argumento un operador binario y el arreglo a procesar, y devuelve el arreglo con las computaciones prefijas calculadas en paralelo.

```
parallelPrefixC :: α → BinaryOp α → Array α → IO (Array α)
parallelPPrefixC neutro bOp a =
  do n ← lengthArray a
     ppc ← newArray n neutro
     bar ← newBarrier n
     old ← newArray n neutro
     let process i = ...
     co . map process $ [1..n]
     return ppc
```

Existe un proceso por cada elemento del arreglo, que va calculando las computaciones prefijas que le corresponda, en paralelo con los otros procesos. Cada proceso necesita hacer una copia del valor viejo de $ppc[i]$ antes de actualizarlo (arreglo $old[i]$). 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 de *barriers* utilizada es la de la sección 4.3. Cada proceso 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 *bodyprocess*). La constante n es igual a la cantidad de elementos del arreglo a, y se calcula al principio del algoritmo (*parallelPrefixC*).

```
process i =
  do a_i ← readArray a i
     writeArray ppc i a_i          — ppc[i] := a[i]
     barrier i bar                 — barrier sincronization
     let bodyprocess d = ...
     bodyprocess 1


bodyprocess d =
```

```
if (d < n)
then do ppc_i ← readArray ppc i
        writeArray old i ppc_i          — old[i] := ppc[i]
        barrier i bar                   — barrier sincronization
        if (i − d) ≥ 1
                then do old_x ← readArray old (i − d)
                        writeArray ppc i (old_x 'opB' ppc_i)
                          — ppc[i] := old[i − d] + ppc[i]
                else skip
        barrier i bar                   — barrier sincronization
        bodyprocess (d * 2)
else skip
```

Por último cabe mencionar que aunque la implementación paralela del problema de las computaciones prefijas de un arreglo tiene como desventaja que requiere la utilización de mecanismos de sincronización de procesos, como son los *barriers*, ésta es mucho más eficiente que la versión secuencial (del orden de $\lceil \log_2 n \rceil$).

4.4 Resumen

En este capítulo se vió que 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. Se especificaron aspectos de la concurrencia desarrollados en forma imperativa con un lenguaje funcional. Esto último no invalida la idea de encontrar mecanismos puramente funcionales (*barriers*). Los semáforos pueden expresarse de manera concisa y agregarse fácilmente al lenguaje Concurrent Haskell. Además pueden utilizarse naturalmente para solucionar problemas concurrentes.

En el capítulo siguiente se presenta otro mecanismo de comunicación y sincronización, los monitores. La verificación de programas funcionales concurrentes se desarrolla en el capítulo 6.

Capítulo 5

$\mathcal{M}$-Haskell

5.1 Introducción

En este capítulo se presenta una extensión del lenguaje funcional Haskell, llamadada $\mathcal{M}$-Haskell [Moc98a], la cual permite escribir programas funcionales concurrentes utilizando monitores [Hoa74, And91]. Los monitores fueron presentados en el apartado 3.3.

Concurrent Haskell [PJGF96] permite la ejecución concurrente mediante la instansiasión de múltiples *threads* (*forkIO*). Además, mediante las variables mutables sincronizadas (*MVar*s), se provee un mecanismo de sincronización y comunicación entre dichos *threads* (ver sección 2.3). Indudablemente, las primitivas ofrecidas por Concurrent Haskell son de muy bajo nivel. El propósito de éstas ha sido, realmente, actuar como un conjunto de bloques básicos para la construcción de otras abstracciones de más alto nivel. Este proceso fue la motivación de lo realizado en el capítulo anterior, y de alguna manera continúa a lo largo de este capítulo. Sin embargo, esta situación parece paradójicamente una contradicción a los lenguajes funcionales puros como Haskell, que han abandonado de alguna manera características de bajo nivel como la asignación, estructuras de control, saltos condicionales e incondicionales, etc. con el objetivo de ganar calidad en el desarrollo de programas. Es decir, Concurrent Haskell permite expresar algoritmos que cumplan y mantengan determinadas propiedades tales como exclusión mutua, sin embargo no existe ningún mecanismo en el lenguage que imponga tales estructuras en los programas.

Una solución ideal sería un lenguaje que combine las virtudes de la concurrencia explícita de Concurrent Haskell y la programación funcional con un mecanismo de alto nivel para el desarrollo de programas concurrentes. Además tal mecanismo debería capturar la noción de abstración y ser fle-

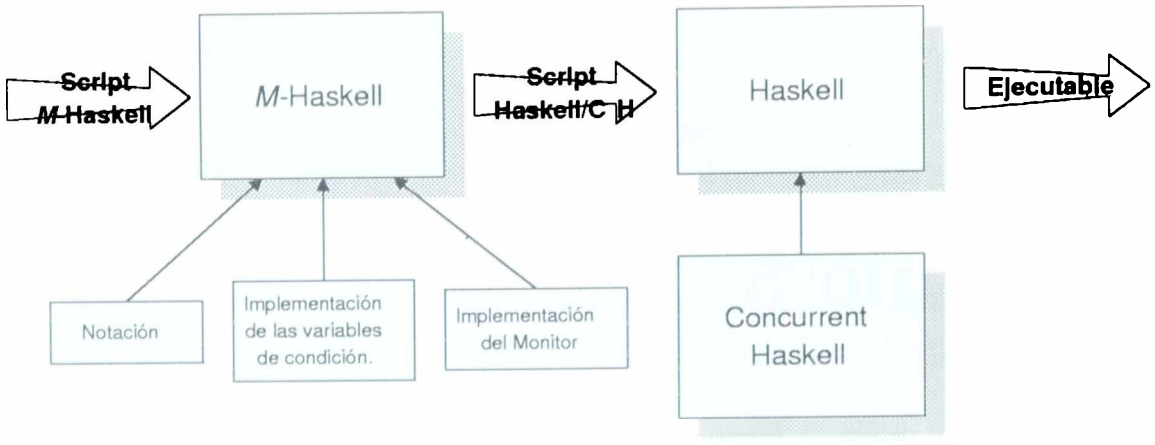

Figura 5.1: Compilación en $\mathcal{M}$-Haskell.

xible, asi como intuitivo. En esta parte del trabajo se realiza un intento experimental en busca de dicha solución ideal. Básicamente, se presenta un preprocesador del lenguaje Haskell, que compila código escrito en $\mathcal{M}$-Haskell y produce código ejecutable de Concurrent Haskell.

En las secciones siguientes se describen los detalles generales de $\mathcal{M}$-Haskell (sección 5.2), se muestran algunos ejemplos de programas concurrentes escritos en tal lenguaje (sección 5.3), y por último, se presenta una generalización de los mecanismos de sincronización estudiados mediante la utilización del sistema de clases del lenguaje Haskell [PH+97, Jon95] (sección 5.4)

5.2 Conceptos Generales

$\mathcal{M}$-Haskell es una extensión del lenguaje funcional Haskell, la cual permite escribir programas concurrentes utilizando el concepto de monitores. Los monitores son un mecanismo de sincronización entre procesos que utiliza el paradigma de comunicación de memoria compartida (ver sección 3.3).

$\mathcal{M}$-Haskell trabaja como una etapa de procesamiento previo del compilador Haskell y Concurrent Haskell. Un *script* con un programa funcional con monitores es procesado por el $\mathcal{M}$-Haskell, y transformado a un programa funcional Haskell con mecanismos de concurrencias implementados al estilo Concurrent Haskell. El proceso de compilación de un programa escrito en $\mathcal{M}$-Haskell puede verse en la figura 5.1.

En la implementación de monitores en $\mathcal{M}$-Haskell se tuvieron en cuenta los siguientes factores:

- como garantizar la exclusión mutua en el acceso al monitor de manera

implícita,

- como representar los recursos que el monitor encapsula,
- como implementar las variables de condición y sus operaciones para proveer la sincronización entre los procesos que utilizan el monitor, y
- utilizar una notación que respete los conceptos de abstracción y modularización de los monitores.

Los recursos se representan mediante variables mutables (*MutVars*), y se declaran mediante la palabra reservada **var** como se describe abajo en la sección que trata sobre la notación utilizada. Los puntos referentes al acceso exclusivo y las variables de sincronización merecen un párrafo aparte y se detallan a continuación.

5.2.1 Notación

La notación utilizada en *M*-Haskell para declarar un monitor es similar a la presentada en la sección 3.3.1. A saber, un monitor encapsula un recurso a ser compartido y provee una serie de operaciones que implementan dicho recurso y permiten manipularlo.

monitor *Mname* **where**
  var *declarations of resources*
  cond *condition variables*
  init *params* = *initialization code*
  procedure op_1 ($params_1$)
  ...
  procedure op_n ($params_n$)
endmonitor

La única manera de alterar el estado del monitor es mediante sus operaciones. Las mismas se invocan usando el nombre del monitor seguido del nombre de la función del monitor, separada por un punto, como indica la siguiente línea de código.

$Mname.op_n$

5.2.2 Acceso exclusivo implícito

Para implementar el acceso exclusivo al monitor se utiliza un semáforo binario por cada monitor. Sea e el semáforo asociado al monitor M, inicializado en 1, el protocolo a seguir se resume en realizar una operación *wait* sobre el semáforo e cuando se accede al monitor y otra de *signal* al abandonarlo. Tanto el semáforo e como el protocolo de acceso exclusivo al monitor, son agregados automáticamnete por $\mathcal{M}$-Haskell. Este proceso es ilustrado en la sección 5.3 mediante el ejemplo de los productores/consumidores.

5.2.3 Variables de condición

Tanto la determinación de la notación del lenguaje para que contemple la utilización de monitores, como la elección del mecanismo para la implementación del acceso son muy importantes. En la primera de ellas se debe definir una grámatica para generar el lenguaje e implementar el *lexer* y *parser* correspondiente. Para el segundo punto, diferentes alternativas tales como semáforos, variables mutables sincronizadas (*MVar*s), etc. fueron consideradas.

Sin embargo, la definición e implementación de las variables de condición es el punto más dificil e interesante de $\mathcal{M}$-Haskell. Intuitivamente, una variable de condición forma una especie de cola de procesos esperando (operación *wait*) a que se satisfaga una condición para ser despertados (operación *signal*). La implementación de las variables de condición mediante semáforos generales es directa, si se considera a dicha cola como una cola FIFO[1] (ver figura 5.2). Notar que también se supone que el *scheduling* de los mecanismos de sincronización primitivos de Concurrent Haskell es FIFO. A las operaciones de los semáforos generales enunciadas en la sección 4.2.2 se les agrega la función $emptyG :: GSem \rightarrow IOBool$ que dado un semáforo general determina si su contenido es igual a 0, es decir, si no hay ningún proceso demorado en el semáforo.

La implementación sencilla anterior tiene problemas a la hora de implementar el *wait* con prioridad. En este caso la cola de procesos subyacente de la variable de condición es una cola con prioridades y no puede implementarse utilizando semáforos generales. La solución a esto radica en utiliar una cola de prioridades de semáforos. De esta manera la operación de *p_wait rank* consiste en crear un nuevo semáforo vacío s, agregarlo a la cola con prioridad *rank*, y luego hacer *wait* s. Por otro lado, la operación *signal* reside en obtener de la cola (y borrar) el semáforo "dormido" de menor prioridad y hacer una operación de *signal* sobre dicho semáforo. Esta misma técnica

[1] *First-In First-Out*

```
data CV          = CV GSem

new              = do gsem ← newG 0
                      return (CV gsem)
signal (CV gsem) = do b ← emptyG gsem
                      if b then skip else signalG gsem
wait   (CV gsem) = waitG gsem
empty  (CV gsem) = emptyG gsem
```

Figura 5.2: Variables de Condición usando Semáforos Generales.

puede también ser usada para implementar el *wait* sin prioridad, en el caso de que el *scheduling* del semáforo no sea FIFO.

Sin embargo existe otro problema aún mayor. Cómo representar las distintas políticas o semánticas del *signal* (ver sección 3.3.2). La difcultad de este problema proviene de que existe una extrecha relación y dependencia con los puntos antes mencionados, es decir, la imlementación del acceso exclusivo, la implementación del *wait* y, obviamente, del *signal*, y en algunos casos aspectos referentes a la notación o restricciones sintácticas de los programas (*Signal and Exit*).

La semántica *signal-and-continue*, en la cual el proceso que ejecuta *signal* mantiene el control de exclusión del monitor y continua su ejecución, es la alternativa más fácil de implementar. La definición de las operaciones *signal* y *wait* de las variables de condición quedan como en la figura 5.2. $\mathcal{M}$-Haskell sólo debe reemplazar dichas operaciones por su correspondiente implementación, y en el caso del *wait*, liberar el acceso exclusivo del monitor antes de agregarlo a la cola de espera, es decir, reemplazar cada *wait cv* de un monitor por el protocolo $do\{signal\ e; wait\ cv; wait\ e\}$, donde e es el semáforo que controla el acceso exclusivo al monitor (ver ejemplo 5.4).

Las posibilidad de que sea el usuario el que determine la semántica de *signal* a utilizar en $\mathcal{M}$-Haskell conlleva a pensar en una implementación flexible del lenguaje. Esta idea motivó la generalizacion que se presenta en la sección 5.4 y la implementación de las variables de condición, y su semántica de *signal* asociada, como un módulo independiente del $\mathcal{M}$-Haskell (ver figura 5.1).

5.3 Ejemplo: Productores & Consumidores

En esta sección se analiza como expresar en $\mathcal{M}$-Haskell el problema de productores/consumidores. Además, se presenta la salida resultante de la transformación sintáctica realizada por el $\mathcal{M}$-Haskell.

El monitor *P&C* encapsula un *buffer* con una cantidad finita de posiciones vacías, recurso que comparten concurrentemente procesos productores y consumidores. El *buffer* en si mismo es un tipo de dato abstracto implementado en el módulo *Buffer*. Los procesos productores acceden al monitor con la operación *deposit*, mientras que los consumidores lo hacen con la función *fetch*. El *script* que representa al módulo donde se define el monitor *P&C* puede verse en la figura 5.3.

```
monitor P&C where
  import Buffer

  var buffer :: Buffer info
  cond empty, full
  init length initInfo = do buffer ← newBuffer length initInfo

  procedure deposit :: info → IO ()
  procedure deposit info =
                    do while (isFull buffer) (wait full)
                       putInto buffer info
                       signal empty

  procedure fetch :: IO info
  procedure fetch =
                    do while (isEmpty buffer) (wait empty)
                       info ← getFrom buffer
                       signal full
                       return info
endmonitor
```

Figura 5.3: Módulo `PC.mhs`

Las variables de condición *full* y *empty* controlan los aspectos referentes a la sincronización entre procesos. Una evita que procesos del tipo productor pongan datos en el buffer si éste se encuentra lleno, y la otra, que procesos consumidores saquen datos del buffer si se encuentra vacío. Por último, un *buffer* vacío es creado por el código de inicialización.

El *script* producido por $\mathcal{M}$-Haskell, es un programa fuente para Concurrent Haskell, que consta básicamente de un tipo de datos que representa al monitor, y de las funciones que lo manipulan (ver figura 5.4). El tipo representa[2] al recurso encapsulado por el monitor (indicado con la palabra reservada **var**), las variables de condición, y el mecanismo utilizado para garantizar el acceso exclusivo al monitor (en este caso un semáforo binario).

La creación del monitor se realiza mediante el llamado de la función *Mname.initparams*, la cual retorna un monitor inicializado. La traducción resultante del preprocesamiento de dicha función es la función $init_{Mname}$ *params* (ver figura 5.4).

Resta mostrar en un ejemplo la invocación al monitor. Las operaciones del estilo $m.op_i$ *params* indican operaciones de invocación del monitor, donde m es el monitor obtenido de ejecutar *Mname.init params* y op_i es una operación del mismo definida en el monitor *Mname*. Las mismas se traducen en op_i m *params*. El código del algoritmo de productores/consumidores, como así también la invocación de sus operaciones por parte de los procesos productor y consumidor es presentada a continuación.

```
main          = do m ← P&C.init 10 φ   — buffer's size=10, φ=empty
                   forkIO (producer m)
                   consumer m
producer m    = doTrue (do info ← < produce info >
                           m.deposit info)
consumer m    = doTrue (do info ← m.fetch
                           < consume info >)
```

La cual se traduce en las siguientes líneas de código. Las líneas de código modificadas son señaladas.

```
main          = do m ← init_P&C 10 φ          — modified by M-Haskell
                   forkIO (producer m)
                   consumer m
producer m    = doTrue ( do info ← < produce info >
                            deposit m info)    — modified by M-Haskell
consumer m    = doTrue ( do info ← fetch m    — modified by M-Haskell
                            < consume info >)
```

[2]se utiliza la notación de registros de Haskell.

```
module MonitorP&C where
  import Buffer

  data MonitorP&C vc info = M{empty :: vc,
                              full  :: vc,
                              buffer :: Buffer info,
                              mutex :: Sem}

  initP&C :: info → Int → IO (MonitorP&C vc info)
  initP&C initInfo length =
                  do empty ← new
                     full  ← new
                     buffer ← newBuffer length initInfo
                     mutex ← newSem
                     return (M{empty = empty,
                               full  = full,
                               buffer = buffer,
                               mutex = mutex})

  deposit :: MonitorP&C vc info → info → IO ()
  deposit (M empty,full, buffer, mutex) info =
     do wait mutex
        while (isFull buffer) (do {signal mutex; wait full; wait mutex})
        putInto buffer info
        signal empty
        signal mutex

  fetch :: MonitorP&C vc info → IO info
  fetch (M empty, full, buffer, mutex) =
   do wait mutex
      while (isEmpty buffer) (do {signal mutex; wait empty; wait mutex})
      info ← getFrom buffer
      signal full
      signal mutex
      return info
```

Figura 5.4: Traducción sintáctica del módulo `PC.mhs`

5.4 Signal and Wait: una generalización

De la definición de semáforos binarios y generales (sección 4.2 y 4.2.2), así también como de la definición de variables de condición (sección 5.2.3) se puede observar que estos mecanismos comparten una interface en común. Utilizando el sistema de clases de Haskell [PH+97, Jon95] la propiedad anterior se especifica mediante la clase *Synchronization* que se muestra a continuación.

```
class Synchronization α where
  new    :: IO α
  signal :: a → IO ()
  wait   :: a → IO ()
  empty  :: a → IO Bool
```

De esta manera tanto los semáforos binarios y generales, como las variables de condición son instancias de la clase *Synchronization*. Claramente, una de las ventajas que provee el sistema de clases es la sobrecarga de operadores (*overloading*), en este caso para los mecanismos de sincronización mencionados. Por ejemplo en la figura 5.5 se muestra la instancia de los semáforos generales a la clase *Synchronization*. Donde las operaciones *opG* son las definidas en la figura 4.1.

```
instance Sincronization GSemaphore where
  new    = newG 0
  signal = signalG
  wait   = waitG
  empty  = emptyG
```

Figura 5.5: Semáforos Generales como instancia de *Synchronization*.

De lo anterior se desprende otra característica importante que presenta la utilización del sistema de clases para especificar la interfase de algunos de los mecanismos de sincronización. Las aplicaciones concurrentes pueden utilizar al mencanismo de sincronización de manera transparente sin contemplar detalles de su implementación. Por ejemplo, el problema de productores/consumidores de la sección 4.2.3 utiliza dos semáforos generales *full* y *empty* para restringir el acceso concurrente al *buffer* (ver figura 4.3). Si dicha solución usase la interfase de la clase *Synchronization*, la implementación de los semáforos quedaría oculta y podría variarse sin necesidad de modificar el algoritmo. Una implementación alternativa de semáforos generales es descripta en el *script* de código *M*-Haskell de la figura 5.6. Dicha

```
monitor SEM where
  var value :: MutVar Int
  cond pos
  init n = do value ← newVar n

  procedure waitM :: IO ()
  procedure waitM = do while (isZero value) (do wait pos)
                       dec value
  procedure signalM :: IO ()
  procedure signalM = do inc value
                         signal pos
  procedure emptyM :: IO Bool
  procedure emptyM = do v ← readVar value
                        return (v == 0)
endmonitor
```

Figura 5.6: Semáforos Generales con monitores (`SEM.mhs`).

implementación de semáforos también puede escribirse como instancia de la clase *Synchronization* de la siguiente manera:

```
instance Sincronization SEM where
  new    = SEM.init 0
  signal = SEM.signalM
  wait   = SEM.waitM
  empty  = SEM.emptyM
```

Amén de las ventajas mencionadas, la motivación de la utilización de este enfoque proviene de la idea de generalizar la implementación de *signal* y *wait* de las variables de condición a fin de que $\mathcal{M}$-Haskell soporte las diferentes semánticas del *signal*. Las distintas implementaciones de las variables de condición se escriben como instancias de la clase *Synchronization*. Los programas escritos en $\mathcal{M}$-Haskell son compilados indicando el modelo de sincronización deseado. Tener en cuenta que, en algunos casos, aspectos de la notación y de la exclusión mutua al monitor entran también en juego.

5.5 Algunas consideraciones

Antes de finalizar con el capítulo de $\mathcal{M}$-Haskell, es importante tener en cuenta algunas aclaraciones y/o consideraciones.

¿Porque los monitores no pueden expresarse como tipos de datos? La idea de expresar a un monitor como un tipo de datos de la forma

data *Monitor* ... = ...

se ve imposibilitada por la restricción del sistema de tipos del lenguaje sobre tipos de primer orden. Los parámetros del monitor (recursos, variables de condición, procedimientos) son muy generales y variables para asociarlos a un tipo genérico monitor. El monitor, más bién, puede verse como un constructor de tipos, que se encuentra al mismo nivel de los constructores **data**, **type** y **newtype** del lenguaje Haskell. Una alternativa futura de continuación de este trabajo se vislumbra en la posibilidad de estudiar una implementación de Haskell que contemple un constructor de tipos para expresar monitores.

Por otro lado, un enfoque diferente es utilizado en la implementación de $\mathcal{M}$-Haskell. $\mathcal{M}$-Haskell es una extensión del lenguaje funcional Haskell, la cual trabaja como una etapa de procesamiento previo del compilador Haskell y Concurrent Haskell (sección 5.2). Las técnicas y detalles de *parsing* y compilación referentes a la implementación del preprocesador escapan al alcance de este trabajo y son omitidas. Para más detalles ver [Moc98a].

Finalmente, un punto importante a tener en cuenta es la idea de agregar a la definición del monitor una operación que represente el invariante asociado al mismo. El invariante, muchas veces representado como una aserción lógica que incluye las variables del monitor, puede expresarse fácilmente mediante una función. De esta manera, se podría pedirle a $\mathcal{M}$-Haskell que chequee, en tiempo de ejecución, el cumplimiento del invariante, al menos en la etapa de desarrollo.

5.6 Resumen

En este capítulo se presentó $\mathcal{M}$-Haskell, una herramienta para expresar programas funcionales concurrentes utilizando monitores. A pesar de que los detalles y formalismos del lenguaje fueron omitidos, se logró especificar una jerarquía de mecanismos de sincronización de procesos concurrentes más abstracta y fácil de utilizar. Se enunciaron los conceptos generales utilizados y se mostraron algunos ejemplos de programas.

Por último, cabe mencionar que se cumplió el objetivo principal de este capítulo y el anterior: ampliar los mecanismos primitivos de Concurrent Haskell. Además, alternativas interesantes de estudios futuros quedan abiertas y pueden continuarse investigando.

Parte IV

Verificación de Programas Funcionales

Capítulo 6

Verificación de Programas Funcionales Imperativos y Concurrentes

6.1 Introducción

La programación funcional es una herramienta poderosa para la especificación de problemas [Hug89, Bir98]. El razonamiento ecuacional que posibilitan los lenguajes funcionales facilita la comprensión, derivación y verificación de programas. Sin embargo, cuando se trabaja con programas funcionales que realizan entrada/salida [JW93] o realizan un procesamiento concurrente [PJGF96, FPJ96], la técnica anterior pierde su simplicidad, aún en los casos en los cuales se preservan propiedades semánticas como la trasparencia referencial [Wad95, LJ94, LJ95, LS97]. Los tipos de programas anteriores se caracterizan por trabajar sobre un estado subyacente, el cual manipulan para producir un resultado. Un ejemplo concreto de estos es la utilización de operaciones monádicas [Wad95].

Utilizando una notación particular, un programa funcional se asemeja a uno escrito en un lenguaje imperativo [JW93], donde sintácticamente se refleja el secuenciamiento de cada operación. Un programa imperativo comienza su ejecución en un estado inicial dado (representado por el conjunto de variables), y cada operación del mismo transforma y produce un nuevo estado. El resultado final del programa (si termina), quedará reflejado en el espacio de variables que determina el estado. El orden de ejecución de cada operación es secuencial y está determinado explícitamente en la codificación del programa. En el caso de programas concurrentes, las operaciones de cada componente se entrecruzan sin ningún orden estipulado (*interleaving*) en el

hilo de ejecución.

Las técnicas tradicionales de verificación de programas secuenciales y concurrentes [AO97, And91, Apt81] se diferencian del razonamiento ecuacional antes mencionado el cual puede aplicarse a los programas funcionales. El diseño y verificación de programas secuenciales y concurrentes [AO97] es usualmente más difícil. El objetivo del trabajo es plantear un método de razonamiento adecuado para la verificación de programas concurrentes en programación funcional, en particular, estudiar la posibilidad de adaptar los métodos tradicionales usados para este fin.

En este trabajo se utiliza el lenguaje funcional lazy y puro Haskell [Tho96, PH+97], en particular la versión GHC de Glasgow [PJHH+93] con sus extensiones con variables mutables y concurrencia.

Una motivación más detallada sobre la importancia del trabajo se describe en la sección 6.2. En la sección 6.3 se realiza un estudio de las técnicas convencionales de verificación de programas sobre programas funcionales sencillos basados principalmente en transformadores de estados (*state transformers*). El concepto anterior, pero adaptado para que soporte variables mutables se presenta en la sección 6.4. En la sección 6.5 se estudia un sistema de demostración para programas funcionales concurrentes. Finalmente la sección 6.6 es un resumen del capítulo.

6.2 Algunas consideraciones

Los lenguajes de programación funcionales puros (en particular la versión GHC [PJHH+93] del Haskell [Tho96, PH+97] que utilizamos) permiten que muchos algoritmos sean expresados de manera concisa. Sin embargo, algunos algoritmos (tablas de hash, grafos, entrada/salida, etc.) hacen un uso intenso de actualización sobre un estado, aún cuando su especificación externa sea puramente funcional. Por otra parte, una gran clase de aplicaciones son naturalmente descriptas como una colección de componentes que cooperan para resolver alguna tarea. Escribir sistemas reactivos o programas concurrentes en un lenguaje funcional puro se torna problemático. Los mecanismos de encapsulación de estados [Wad95, LJ95], entrada/salida [JW93] y concurrencia [PJGF96] conocidos permiten describir dicha clase de algoritmos sin perder las principales virtudes de los lenguajes funcionales: independencia del orden de evaluación (propiedad de Church-Rosser), transparencia referencial, semántica no estricta, etc.

A pesar de que en los trabajos antes mencionados se detallan aspectos referentes a la semántica de las computaciones monádicas, no presentan éstos ningún método que permita razonar o demostrar propiedades de los progra-

mas de este tipo de manera práctica. Si consideramos que un programa funcional que maneja estados es similar a uno imperativo, podemos entonces usar técnicas de verificación de programas conocidas para el paradigma imperativo. De esta manera, uno de los objetivos principales de este trabajo es adaptar algunos métodos conocidos para razonar sobre programas imperativos al caso en consideración, es decir programas funcionales imperativos (lo cual incluye tanto las versiones secuencial como concurrente). Aunque el razonamiento lógico usado en las técnicas de verificación convencionales suele ser más complicado que el razonamiento ecuacional usado en programación funcional y se presenta el problema de tener que hablar de dos clases de variables (las puras de funcional y las mutables) el razonamiento con estados sería necesario sólo en pocas partes de los programas, dado que se espera que la mayor parte del código de un programa esté determinado por funciones que no utilizan acciones sobre estados.

6.3 Transformadores de estados

En un primer momento se analiza la verificación de programas funcionales que manejan el estado de manera implícita (estilo propuesto por Wadler en [Wad95]). El espacio de variables sobre el cual se definen los estados de una computación está determinado desde el principio, y no puede cambiarse dinámicamente. A continuación se presenta el modelo monádico utilizado. Se describen las funciones monádicas a través de su traducción a funciones que manejan explícitamente el estado. Los conceptos utilizados son los mismo que los presentados en la sección 2.2. El espacio de estados se describe con un array, el cual se maneja usando las operaciones standard de *index* y *update*.

$$
\begin{array}{lrl}
\textbf{type} & State & = Arr \\
\textbf{type} & Ix & = Id \\
\textbf{type} & Val & = Int \\
\textbf{type} & M\ a & = State \rightarrow (a, State) \\
return\ a & & = \lambda s \rightarrow (a, s) \\
m \rhd k & & = \lambda s \rightarrow \textbf{let}\ (a, y) = m\ s \quad \textbf{in} \\
 & & \qquad \textbf{let}\ (b, z) = k\ a\ y\ \ \textbf{in}\ (b, z) \\
m \gg k & & = m \rhd \lambda \_ \rightarrow k \\
fetch\ i & & = \lambda s \rightarrow (index\ i\ s,\ s) \\
assign\ i\ v & & = \lambda s \rightarrow ((), update\ i\ v\ s)
\end{array}
$$

Este modelo puede utilizarse como marco para la construcción de programas que transforman estados. Los programas funcionales con estas características se asemejan a los programas convencionales imperativos ya que

toman un estado inicial, realizan una secuencia de operaciones sobre dicho estado y obtienen un estado final y un resultado. Para resaltar sintácticamente esta semejanza se utiliza la *do notation* [Bir98] (ver sección 2.2), que permite escribir a los programas como una secuencia de operaciones. Generalmente, en esta parte del trabajo, no se hace uso de la propiedad de *layout rule*, y se escriben explícitamente las llaves y puntos y comas de *do notation* para facilitar las demostraciones.

Informalmente, un programa determinístico es correcto si satisface la relación de pre y poscondición. Las técnicas de verificación de programas de Hoare [AO97, Apt81] (sección 3.4) permiten demostrar que dado un programa imperativo S, si éste comienza en un estado que satisface una precondición $\{P\}$ termina en un estado que satisface una poscondición $\{Q\}$. Tradicionalmente la notación usada para esto es la de ternas de Hoare $\{P\}S\{Q\}$, donde P y Q son predicados sobre las variables de S que corresponden a la pre y poscondición respectivamente. Para demostrar la validez de las ternas de Hoare presentamos un conjunto de axiomas y reglas que definen un sistema de demostración para los transformadores de estados.

6.3.1 Sistema de demostración

En este apartado se presentan los axiomas y reglas de inferencia que definen al sistema ST (por *state transformers*), el cual permite demostrar la corrección de programas funcionales basados en transformadores de estados. Para esto se utiliza un razonamiento sintáctico sobre la estructura del programa. Este sistema provee axiomas para cada una de las operaciones de la mónada de estado y reglas para las expresiones y construcciones del lenguaje más comunes (ver figura 6.1). Además, al igual que los sistemas de demostración tradicionales de los lenguajes imperativos, se provee la regla de consecuencia.

Como un primer ejemplo del método de verificación y demostración de programas funcionales propuesto en este trabajo, se analiza el programa *swap*, que intercambia el valor de dos variables dadas.

$$\begin{aligned} \mathit{swap}\ x\ y\ =\ \mathbf{do}\{ & a \leftarrow \mathbf{fetch}\ x\ ; \\ & b \leftarrow \mathbf{fetch}\ y\ ; \\ & \mathbf{assign}\ x\ b\ ; \\ & \mathbf{assign}\ y\ a\ \} \end{aligned}$$

Una propiedad interesante a demostrar sobre la función *swap* es

$$\{x = X \wedge y = Y\}\ \mathit{swap}\ \{x = Y \wedge y = X\},$$

$$\{P\}\ \textbf{return}\ a\{P\} \qquad \texttt{(A1)}\quad \textbf{return}$$

$$\{P(a := x)\}\ a \leftarrow \textbf{fetch}\ x\ \{P\} \qquad \texttt{(A2)}\quad \textbf{fetch}$$

$$\{P(x := e)\}\ \textbf{assign}\ x\ e\{P\} \qquad \texttt{(A3)}\quad \textbf{assign}$$

$$\{P(a := e)\}\ \textbf{let}\ a = e\{P\} \qquad \texttt{(A4)}\quad \textbf{let}$$

$$\frac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}} \qquad \texttt{(R1)}\quad \textbf{bind}$$

$$\frac{\{P \wedge b\}S_1\{Q\} \quad \{P \wedge \neg b\}S_2\{Q\}}{\{P\}\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\{Q\}} \qquad \texttt{(R2)}\quad \textbf{if}$$

$$\frac{P\prime \Rightarrow P \quad \{P\}S\{Q\} \quad Q \Rightarrow Q\prime}{\{P\prime\}S\{Q\prime\}} \qquad \texttt{(R3)}\quad \textbf{cons}$$

Figura 6.1: Sistema de demostración ST.

donde X e Y son dos valores cualesquiera. En la demostración que se presenta a continuación, algunas aserciones intermedias son escritas entre la definición de la función para explicitar el proceso de la demostración. Además se agregan comentarios sobre los axiomas y las reglas aplicadas.

$\{x = X \wedge\ y = Y\} \Rightarrow \{y = Y \wedge\ x = X\}$
  $a \leftarrow$ **fetch** x; — por A2 y R1
$\{y = Y \wedge\ a = X\}$
  $b \leftarrow$ **fetch** y; — por A2 y R1
$\{b = Y \wedge\ a = X\}$
  assign $x\ b$; — por A3 y R1
$\{x = Y \wedge\ a = X\}$
  assign $y\ a$ — por A3
$\{x = Y \wedge\ y = X\}$

6.3.2 Regla de recursión

De la misma manera que los lenguajes imperativos utilizan en gran medida estructuras cíclicas o de repetición (**while**), los lenguajes funcionales utilizan las definiciones recursivas. Para razonar con programas funcionales recursivos hace falta una regla que contemple la recursión. De las posibles

presentaciones de estas reglas, usamos una de las más simples la cual resulta suficiente para el ejemplo considerado más adelante.

Si la función F (sin parámetros) se define con la ecuación $F = S_0$, entonces para demostrar $\{P\}F\{Q\}$ es necesario demostrar $\{P\}S_0\{Q\}$ suponiendo que vale $\{P\}F\{Q\}$ para cada aparición de F en S_0.

$$\frac{\{P\}F\{Q\} \vdash \{P\}S_0\{Q\}}{\{P\}F\{Q\}} \quad \texttt{(R4)} \quad \textbf{rec}$$

La regla de recursión (R4) se utiliza en la demostración del siguiente ejemplo. La función gcd encuentra el máximo común divisor entre dos números. Se sabe que para $x > 0$ e $y > 0$, $x \gcd y$ debe cumplir

1. $x \gcd x = x$
2. $x > y \Rightarrow (x - y) \gcd y$
3. $x < y \Rightarrow x \gcd(y - x)$

Una posible implementación de la función *gcd* es

```
gcd x y = do{ a ← fetch x;
              b ← fetch y;
              do{ if a == b then return a;
                  else if (a > b) then assign x (a − b);
                                  else assign y (b − a);
                  gcd x y }}
```

Para demostrar la propiedad $\{x = X \wedge y = Y\}\ gcd\ \{x = X \gcd Y\}$ tenemos que probar:

1. $p \Rightarrow I$
2. $I \wedge C \Rightarrow q$
3. $\{I\}\,\text{gcd}\{I\}$

donde $p \equiv x > 0 \wedge y > 0 \wedge x = X \wedge y = Y$ y $q \equiv x = X \gcd Y$ son respectivamente la pre y poscondición de gcd, $I \equiv x > 0 \wedge y > 0 \wedge x \gcd y = X \gcd Y$ es el invariante del programa y $C \equiv x = y$ el caso base de la recursión. Los pasos 1 y 2 se deducen fácilmente mediante deducciones lógicas. El paso 3 requiere un poco más de análisis. Suponiendo $\{I\}\,\text{gcd}\{I\}$ se tiene la siguiente demostración.

$\{I\}$

$a \leftarrow$ *fetch* x ;
$b \leftarrow$ *fetch* y ;
$\{H \equiv I \wedge x = a \wedge y = b\}$ — Por A2 dos veces y R1
if $a == b$ **then**
$\{H \wedge a = b\}$
return a; — Por A1, R3 y R2
$\{H \wedge a = b\} \Rightarrow \{I\}$
else if$(a > b)$
then
$\{H \wedge a \neq b \wedge a > b\} \Rightarrow \{I[x := a - b]\}$
assign x $(a - b)$; — Por A3, R3 y R2 dos veces
$\{I\}$
else
$\{H \wedge a \neq b \wedge a \leq b\} \Rightarrow \{I[y := b - a]\}$
assign y $(b - a)$; — Por A3, R3 y R2 dos veces
$\{I\}$
$\{I\}$
gcd — Vale por R1 e hipótesis de R4
$\{I\}$

6.4 Variables Mutables

En la sección anterior se presentó un método de razonamiento sobre transformadores de estados, donde el manejo del estado se realiza en forma implícita. Por otro lado, en el estilo propuesto por Launchbury y Peyton Jones en [LJ95], el manejo del estado se realiza de manera explícita y dinámica[1]. Lo anterior se logra con una serie de funciones primitivas que permiten crear variables mutables, leer su contenido y modificarlas. De esta manera el estado subyacente de un programa con variables mutables se compone de las variables mutables que existen en el programa.

Las variables mutables pueden utilizarse de manera segura dentro de la mónada de entrada/salida. Las operaciones que utilizaremos son las vistas en la sección 2.3, y se resumen a continuación.

type $MutVar\ \alpha = \ldots$

newVar :: $\alpha \rightarrow IO(MutVar\ \alpha)$
readVar :: $MutVar\ \alpha \rightarrow IO\ \alpha$
writeVar :: $MutVar\ \alpha \rightarrow \alpha \rightarrow IO()$

[1] Para una comparación más detallada entre los dos estilos ver [Kag97].

$$\{P[a := x\wedge]\}\ a \leftarrow \mathbf{readVar}\ x\ \{P\} \qquad \texttt{(A2*)}\quad \mathbf{readVar}$$

$$\{P[x\wedge := e]\}\ \mathbf{writeVar}\ x\ e\ \{P\} \qquad \texttt{(A3*)}\quad \mathbf{writeVar}$$

$$\{P[x\wedge := e]\}\ x \leftarrow \mathbf{newVar}\ e\ \{P\} \qquad \texttt{(A5*)}\quad \mathbf{newVar}$$

Figura 6.2: Axiomas para el sistema de demostración MV.

En la ejecución de un programa, la función[2] $x \leftarrow$ **newVar** e "agrega al estado una nueva variable denominada mediante la variable local x y que referencia a la expresión e. La función $a \leftarrow$ **readVar** x lee el valor que contiene la variable mutable x, no modifica el estado, y se lo asigna a la variable local a. Finalmente, la función **writeVar** x e asigna la expresión e como el contenido de la variable mutable x.

6.4.1 Un sistema de demostración para las MutVars

El sistema de demostración ST debe ser adaptado para que contemple el manejo del estado de manera explícita. Dado que en este artículo no usaremos variables mutables que hagan referencia a otras variables, es consistente referirse al valor de una variable usando el nombre de ésta. Sin embargo, se utiliza la notación $x\wedge$ para simbolizar al valor que contiene la variable mutable x. Una discusión más amplia y basada, seguramente en estudios futuros, plantea la eliminación de dicha referencia. El nuevo sistema de pruebas MV (por *mutable variables*) para variables mutables resulta de reemplazar los axiomas A2 y A3 por A2* y A3*, y agregar A5* (ver figura 6.2).

Nótese la simetría entre los axiomas para leer y escribir una variable mutable. Esto se debe a que para poder hacer referencia al valor de una variable mutable en una expresión es necesario tomar antes el valor devuelto por el transformador de estados **readVar**. Puede pensarse a la variable ligada (a en el axioma) como una fotografía de la variable mutable que se usa en las expresiones. Obviamente, si la variable mutable cambia el valor la fotografía pierde su actualidad.

Usando este nuevo sistema pueden demostrarse de manera similar los ejemplos desarrollados anteriormente adaptados al estilo explícito, es decir con el uso de variables mutables. En vez de realizar esa tarea, demostraremos una regla que utilizaremos más adelante en la construcción de programas. La función *inc* toma una variable mutable que contiene un número e incrementa su contenido en uno. La propiedad o teorema relacionado a *inc* es

[2]Utilizando notación **do**

$\{P[x\wedge := x \wedge +1]\}inc\ x\{P\}$, la cual se demuestra fácilmente a partir de la definición de *inc* dentro del sistema MV.

$$
\begin{array}{lll}
\{P[x\wedge := x \wedge +1]\} \equiv \{P[x\wedge := a+1][a := x\wedge]\} & & \\
\quad a \leftarrow \textbf{readVar}\ x; & & \text{— por A2* y R1} \\
\{P[x\wedge := a+1]\} & & \\
\quad \textbf{writeVar}\ x\ (a+1) & & \text{— por A3*} \\
\{P\} & &
\end{array}
$$

6.4.2 Problemas con las MutVars

A diferencia de los programas tranformadores de estados donde las variables que determinan el estado quedan determinadas de manera estática, los programas con variables mutables agregan un problema, el del indireccionamiento. Nada prohibe que las variables mutables hagan referencia a otras variables mutables. A partir de esto, una variable mutable se asemeja a los punteros de los lenguajes imperativos, lo cual restringe el razonamiento del sistema de demostración MV a programas que sólo contengan variables mutables sin indireccionamiento. En el ejemplo *bad* se puede ver la falla del sistema de prueba MV a la hora de demostrar la propiedad, claramente válida, $\{true\}\ bad\ \{x\wedge\ = 6\}$.

$$
\begin{array}{lll}
bad\ = & \textbf{do}\{\ x \leftarrow \textbf{newVar}\ 5; & \\
& \quad y \leftarrow \textbf{newVar}\ x; & \\
& \quad \tilde{x} \leftarrow \textbf{readVar}\ y; & \\
& \quad \textbf{writeVar}\ \tilde{x}\ 6\} &
\end{array}
$$

6.5 Concurrencia

En este apartado se estudian los programas funcionales concurrentes utilizando como lenguaje al Concurrent Haskell, y se presenta el sistema de demostración SV (*Synchronous Variables*) para la verificación de los mismos. Concurrent Haskell provee mecanismos de concurrencia de bajo nivel. Aunque en capítulos (4 y 5) y trabajos anteriores [BMLM97, Moc98a] se definen estructuras de concurrencia de más alto nivel, el sistema sólo contempla la visión más primitiva.

Concurrent Haskell [PJGF96] es una extensión concurrente del lenguaje funcional Haskell [PH+97]. Esta extensión consiste, básicamente, en una

serie de primitivas con una semántica determinada. Las características principales que agrega *Concurrent Haskell* son los procesos, junto con un mecanismo para la inicialización de los mismos (*forkIO*), y el concepto de variables mutables sincronizadas (*MVar*), para poder realizar la comunicación y sincronización entre procesos. El mecanismo primitivo que provee sincronización y comunicación entre procesos en *Concurrent Haskell* son las *MVar*. Las *MVar* son una versión sincronizada de las *MutVar* presentadas anteriormente. Para más detalle se recomienda ver la sección 2.3.

Es interesante destacar que dos procesos concurrentes creados con la función *forkIO* comparten las variables mutables creadas anteriormente, pudiendo interferir uno con el otro, por lo cual debe garantizarse que esto no ocurra de manera indebida.

Un método usado exitosamente en la programación imperativa para asegurar esta propiedad, consiste en colocar aserciones entre cada par de operaciones consideradas atómicas y comprobar que éstas son local y globalmente correctas, esto es que son establecidas dentro de la componente a la cual pertenecen y que no son falsificadas por las acciones atómicas de las otras (ver sección 3.4).

6.5.1 Sistema de demostración SV

Un programa concurrente está compuesto por dos o más componentes que cooperan. Cada componente se ejecuta como un programa secuencial. Las componentes interactúan mediante las variables sincronizadas que comparten. La verificación de un programa concurrente consta de la corrección local de cada componente y la corrección global de todo el programa. Cada componente secuencial se verifica por separado. Como el orden de ejecución no está completamente especificado, se debe demostrar que ninguna acción de una componente afecta la correción local de cada uno del resto de las componentes. Esto implica que las aserciones deben elegirse considerando no sólo el comportamiento local sino también las posibles interferencias de otras componentes, lo cual hace que las aserciones suelan ser más débiles de lo esperado.

El sistema de demostración SV está compuesto por los axiomas y reglas de la figura 6.3. Estas reglas son una generalización de las usadas para semáforos binarios, donde puede interpretarse la aserción $b = \square$ como que el semáforo binario b está en cero, o vacío.

Como ejemplo del posible uso del sistema de demostración se usa el problema de sincronización por barreras [And91]. La sincronización por barreras es un mecanismo para conseguir que varios procesos concurrentes se mantengan ejecutando "en fase", es decir, que ninguno se adelante en su ejecución

$$\{P[x\wedge := e]\}\ x \leftarrow \mathbf{newMVar}\ e\ \{P\} \qquad \texttt{(A1+)}\quad \mathbf{newSV}$$

$$\{x \neq \square \Rightarrow\ P[x := \square, a := x\wedge]\}\ a \leftarrow \mathbf{takeMVar}\ x\ \{P\} \qquad \texttt{(A2+)}\quad \mathbf{take}$$

$$\{x = \square \Rightarrow\ P[x\wedge := e]\}\ \mathbf{putMVar}\ x\ e\{P\} \qquad \texttt{(A3+)}\quad \mathbf{put}$$

$$\frac{\{P\}S_1\{Q\} \quad \{R\}S_2\{S\} \quad LI}{\{P \wedge R\}\, forkIO\ S_1; S_2\{Q \wedge S\}} \qquad \texttt{(R1+)}\quad \mathbf{fork}$$

$$\frac{\{P\}S\{Q\}}{\{P\} < S > \{Q\}} \qquad \texttt{(R2+)}\quad \mathbf{atomic}$$

Figura 6.3: Sistema de demostración SV.

mientras haya alguno que aún no ha completado su etapa. Para esto se establece una barrera en la cual cada proceso debe esperar a que todos los otros hayan llegado para poder continuar con su ejecución.

Por una cuestión de simplicidad se consideran dos procesos que comparten una barrera. Una implementación de barreras para más procesos puede verse en la sección 4.3. Abajo se muestra el código del programa funcional con las aserciones intermedias correspondientes.

$barrier = \mathbf{do}\{forkIO\ p_1;\ p_2\}\ \mathbf{where}$

$p_i = \mathbf{do}\{\{\mathsf{Barrier}\} \Rightarrow$
$\{b_i = \square \Rightarrow ((){=}\square \Leftrightarrow a_i{+}1{=}d_j) \wedge R_j \wedge d_j \leq a_i + 1 \leq d_j + 1 \wedge S_j\}$
$< putMVar\ b_i\ ();\ inc\ a_i >;$ — Por A3+,inc ,R2+ y R3
$\{\mathsf{Barrier}\} \Rightarrow$
$\{b_j \neq \square \Rightarrow R_i \wedge (\square{=}\square \Leftrightarrow a_j = d_i{+}1) \wedge S_i \wedge d_i{+}1 \leq a_j \leq d_j{+}2\}$
$< \_ \leftarrow takeMVar\ b_j; inc\ d_i >;$ — Por A2+,inc, R2+,R1 y R3
$\{\mathsf{Barrier}\}$
$p_i;$
$\{\mathsf{Barrier}\}\}$

donde $i = 1, 2$ y $j = (i\, mod\, 2) + 1$ y el invariante de las componentes concurrentes es $\mathsf{Barrier} \equiv (\bigwedge_i R_i) \wedge (\bigwedge_i S_i)$, siendo $R_i \equiv b_i = \square \Leftrightarrow a_i = d_j$ y $S_i \equiv d_j \leq a_i \leq d_j + 1$, que representan respectivamente, el invariante de las *MVar* y el invariante de las variables mutables auxiliares. La computación atómica ($< S >$) se utiliza para que se mantenga el invariante. Sin embargo, las variables auxiliares no son necesarias en la versión final del programa, ya que sólo se utilizan para demostrar propiedades del programa concurrente,

pueden eliminarse del código del programa (ver, por ejemplo, [And91] para una formulación precisa de dicha regla). En el ejemplo, se puede ver cómo las operaciones de las *MVar* funcionan como las operaciones *signal* y *wait* de un semáforo binario, es decir, nos interesa la condición de sincronización de las *MVar* y no su contenido (ver sección 4.2).

Por cuestiones de claridad se demuestra la implicación

$$\mathsf{Barrier} \Rightarrow (b_j \neq \square \Rightarrow R_i \wedge (\square = \square \Leftrightarrow a_j = d_i + 1) \wedge S_i \wedge d_i + 1 \leq a_j \leq d_j + 2)$$

que aparece en el problema de las barreras. Las otras implicaciones lógicas son similares y se omiten. Se utiliza la propiedad de que demostrar un predicado lógico de la forma $A \Rightarrow (B_1 \wedge \ldots \wedge B_n)$ es quivalente a demostrar por separado cada predicado $A \Rightarrow B_i$ para $i = 1 \ldots n$. Trivialmente se ve que por la definición de Barrier los predicados

$$(\mathsf{Barrier} \equiv R_i \wedge R_j \wedge S_i \wedge S_j) \Rightarrow (b_j \neq \square \Rightarrow R_i) \quad \text{y}$$
$$(\mathsf{Barrier} \equiv R_i \wedge R_j \wedge S_i \wedge S_j) \Rightarrow (b_j \neq \square \Rightarrow S_i)$$

son verdaderos. Por último resta demostrar

$$\mathsf{Barrier} \Rightarrow (b_j \neq \square \Rightarrow (\square = \square \Leftrightarrow a_j = d_i + 1)) \quad \text{y}$$
$$\mathsf{Barrier} \Rightarrow (b_j \neq \square \Rightarrow d_i + 1 \leq a_j \leq d_j + 2)$$

La primera de ellas es verdadera ya que por R_j y $b_j \neq \square$ se sabe que $a_j \neq d_i$, luego por S_j vale que $a_j = d_i + 1$. El segundo predicado es también válido y se deduce de la demostración anterior.

6.6 Resumen

En este capítulo se introducen formas de razonar con los programas funcionales imperativos (tanto secuenciales como concurrentes) que son adaptaciones de los mecanismos tradicionales para verificar y derivar programas imperativos. El objetivo del trabajo es mostrar que esto es posible analizando las diferencias con los métodos tradicionales así también como las dificultades a ser resueltas. Una de las novedades interesantes es la necesidad y la posibilidad de usar variables de estado junto a variables ligadas (por cláusulas "let" o abstración lambda) en las aserciones sobre estados. Esto implica reglas relativamente simétricas para el razonamiento con los estados.

La demostración de consistencia de los sistemas lógicos usados está fuera del alcance de este trabajo. Los axiomas y las reglas usadas son tradicionales y deberían ser satisfechos por cualquier implementación de las primitivas de manejo de estado y concurrencia. Adicionalmente, no se considera en ningún

momento el orden de evaluación de las operaciones, lo cual hace que la lógica sea aceptable cualquiera sea éste. Obviamente, esto implica que no va a ser (relativamente) completa, si se considera la "semántica lazy".

Un punto inmediato para continuar trabajando es en la idea de usar versiones más poderosas de la regla de la recursión, lo cual abre posibilidades aún no demasiado exploradas cuando se combina con la ejecución concurrente de los procesos. Una extensión simple es poder referirse no sólo al estado modificado por una función recursiva (como se hace en el caso del máximo común divisor) sino también al valor devuelto por dicha función.

Parte V

Conclusiones

Capítulo 7

Conclusiones y Trabajos Futuros

La aspiración principal de este proyecto de investigación es el estudio de las técnicas, mecanismos y formalismos concurrentes con el objetivo de caracterizar a la programación funcional como una herramienta para expresar programas concurrentes. Para esto, características y particularidades de Concurrent Haskell, como así también los conceptos previos que lo fundamentan, han sido estudiados. De la misma manera, se estudiaron algunos de los mecanismos de sincronización y comunicación entre procesos tradicionales usados en programación concurrente (semáforos, monitores, etc.). Formalismos lógicos para la verificación de programas (módelo de Hoare, lógica de Owicki-Gries) fueron también de interés al proyecto.

Se analizó y ejemplificó la expresividad de programas concurrentes en Concurrent Haskell. Concurrent Haskell presenta mecanismos primitivos de concurrencia de muy bajo nivel. Nuevas herramientas para la especificación de programas concurrentes han sido desarrolladas a partir de los conceptos primitivos de Concurrent Haskell. Las mismas fueron desarrolladas jerarquícamente de acuerdo a su poder de abstracción. En este punto, aspectos de la concurrencia tradicional fueron especificados con un lenguaje funcional. Esto último no invalida la idea de encontrar mecanismos puramente funcionales.

Dentro de las herramientas desarrolladas en programación funcional concurrente, los semáforos funcionales, pueden expresarse de manera concisa y agregarse fácilmente al lenguaje. Además pueden utilizarse naturalmente para solucionar problemas concurrentes y contribuyen a la expresividad del lenguaje. Otra característica importante de los semáforos es que constituyen un bloque para la construcción de mecanismos de mayor poder expresivo en el desarrollo de programas concurrentes. Los mecanismos de sincronización

que se basan en la sincronización por condición (operaciones *signal* y *wait*) han sido generalizados mediante el sistema de clases del lenguaje Haskell.

En este trabajo se presentó $\mathcal{M}$-Haskell, una herramienta para expresar programas funcionales concurrentes utilizando monitores. Debido a sus características de expresividad, modularización y abstracción, los monitores son el concepto de concurrencia que más se adecua a la programación funcional concurrente. Por lo tanto, el estudio de $\mathcal{M}$-Haskell, así como otras alternativas de agregar monitores a Haskell, seguirá siendo tema de estudio e investigación. También se estudiarán la posibilidad de agregar mecanismos de sincronización que utilizan el paradigma de comunicación basado en el pasaje de mensajes.

La otra meta importante de este trabajo era la verificación de programas funcionales. En programas funcionales que utilizan las mónadas de manera intensiva, el razonamiento ecuacional no parece suficiente para demostrar la corrección de éstos. En este trabajo se introducen formas de razonar con los programas funcionales imperativos (tanto secuenciales como concurrentes) que son adaptaciones de los mecanismos tradicionales para verificar y derivar programas imperativos. Esto es posible analizando las diferencias con los métodos tradicionales así también como las dificultades a ser resueltas.

Existen varios caminos por los cuales se puede continuar esta rama del trabajo. Por un lado, usar versiones más poderosas de la regla de la recursión, lo cual abre posibilidades aún no demasiado exploradas cuando se combina con la ejecución concurrente de los procesos. Un tema a ser estudiado próximamente es el análisis de las propiedades de consistencia y completitud relativa de los sistemas considerados, y como influye en las mismas el orden de evaluación del lenguaje.

Una extensión que permite un uso más amigable de las mónadas, es disponer de un operador de elimínción de las mismas. En varios trabajos se han propuesto diversas variantes (generalmente llamados RunXX), siendo siempre el problema principal asegurar que su uso sea seguro, esto es, que el valor no dependa del "estado del mundo". Esta propiedad es asegurada a través del sistema de tipos, mostrando que los transformadores de estado admisibles son "*single threaded*". Para el caso concurrente esto no es, por supuesto, posible. Sin embargo, es posible que bajo determinadas condiciones pueda eliminarse la mónada (por ejemplo en los casos en los cuales el valor final de la mónada es único). Si bien estas condiciones no parecen fácilmente tratables por un sistema de tipos, podrían ser demostradas usando lógica de la programación y aplicando en esos casos la expresión RunXX correspondiente. Si bien esto permitiría expresiones sintácticamente correctas (y tipables) que no posean propiedades interesantes que sean demostrables, esto es de uso común en programación (por ejemplo un ciclo infinito).

Por último, para tratar el caso general de Concurrent Haskell, hace falta poder razonar con punteros, lo cual se sabe que es difícil. A esto se suma la dificultad de la sincronización involucrada (podrían pensarse las *MVar*s como punteros sincronizados).

Bibliografía

Bibliografía

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

[AO97] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, second edition, 1997.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey part I. *ACM Transactions on Programming Languages ans Systems*, 3(4):431 483, October 1981.

[Bir98] Richard S. Bird. *Introduction to Functional Programming.* Prentice Hall, second edition, 1998.

[BMLM97] J.. Blanco, P.E. Martínez López, and P.A. Mocciola. Mecanismos de sincronización en programación funcional. In *Proceedings of 2nd Latin-American Conference on Functional Programming (CLaPF) that was held in III CACiC*, La Plata, Argentina, October 3-4 1997.

[BMS80] R. Burstall, D. MacQueen, and D. Sanella. Hope: An experimental applicative language. Technical report, Department of Computer Science, University of Edinburgh, 1980.

[Dav92] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.

[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.

[Flo67] R. Floyd. Assigning meaning to programs. In *Proceedings of Symposium on Applied Mathematics 19*, 1967.

[FPJ96] Sigbjorn Fine and Simon Peyton Jones. Programming reactive systems in Haskell. Technical report, University of Glasgow, 1996.

[Har86] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, Universuty of Edinburgh, November 1986.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communication of ACM*, 12:576–583, 1969.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. In *Comm. ACM*, volume 17, pages 549–557, October 1974.

[How] John H. Howard. Signaling in monitors. Technical report, University of texas at Austin.

[How76] John H. Howard. Proving monitors. In R. S. Gaines, editor, *Communications of the ACM*, volume 19, pages 273–279, May 1976.

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

[Hud92] Paul Hudak and J.H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), 1992.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

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

[Jon95] M.P. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming,*, LNCS 925(925):97 136, May 1995.

[Jon98] Mark P. Jones. Hugs 1.4 - the Haskell user's Gofer system. User manual. Technical report, Department of Computer Science, University of Nottingham, January 1998.
`http://haskell.systemsz.cs.yale.edu/hugs/`.

[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.

[Kag97] Koji Kagawa. Implicit and reference-passing styles in stateful functional programming. In *Proceeedings of the International Conference on Functional Programming*, Amsterdam, June 9-11 1997. ACM SIGPLAN, ACM Press.

[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.

[LJ95] John Launchbury and Simon Peyton Jones. State in Haskell. In *Lisp and Symbolic Computation*, volume 8, pages 193 341, 1995.

[Loi98] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[LS97] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *Proceeedings of the International Conference on Functional Programming*, pages 227–238, Amsterdam, June 9-11 1997. ACM SIGPLAN, ACM Press.

[Mac90] Bruce J. Mac Lennan. *Functional Programming: Practice and Theory.* Addison-Wesley Publishing Company, 1990.

[Mar98] Ricardo Peña Marí. *Diseño de Programas: Formalismo y abstracción.* Prentice Hall, Madrid, 1998.

[Moc98a] Pablo A. Mocciola. $\mathcal{M}$-Haskell. Technical report, LIFIA. Universidad de La Plata., 1998.

[Moc98b] Pablo A. Mocciola. Programación funcional paralela: Una manera de separar el qué del cómo, Julio 1998. Trabajo Estudiantil. JAIIO 27.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communication of ACM*, 19:279–285, 1976.

[PH+97] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.4. Technical report, Yale University, April 1997.

[PJGF96] 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`.

[PJHH+93] SL. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: A technical overview. In *Join Framework for Information Technology Technical Conference*, Keele, 1993.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[Sab98] Amr Sabry. What is purely functional language? *Journal of Functional Programming*, 8(1), January 1998.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[Tur85] David Turner. Miranda: a non-strict functional language with polymorphic types. *Functional Programming Languages and Computer Architecture*, 1985.

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

BIBLIOTECA
U.N.L.P.

TES
98/16
DIF-02060
SALA

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMATICA
Biblioteca
50 y 120 La Plata
catalogo.info.unlp.edu.ar
biblioteca@info.unlp.edu.ar

DIF-02060