Universidad Nacional de Rosario

Tesina de grado
para la obtención del grado de
Licenciado en Ciencias de la Computación

Simulación de Programas Paralelos en Haskell

Autor:
Martín A. Ceresa

Director:
Dr. Mauro Jaskelioff
Co-Director:
Lic. Exequiel Rivas

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

27 de marzo de 2015

Resumen

El aumento del poder de cómputo de procesadores con un solo núcleo ha llegado a su fin, y por lo tanto los procesadores multinúcleo ya están presentes en la mayoría de dispositivos electrónicos. Esto hace que la programación paralela sea un requerimiento importante al momento de desarrollar un programa.

Para incorporar paralelismo en los programas se han extendido los lenguajes de programación, o bien se han desarrollado bibliotecas, con diversos grados de éxito. El estudio de formas de incorporar en forma simple y eficiente la programación paralela al desarrollo de software sigue siendo objeto de investigación. En particular, en el lenguaje de programación Haskell (Marlow 2010) se han desarrollado una variedad de extensiones del lenguaje, bibliotecas, y abstracciones, que permiten encarar el problema de la programación paralela desde diversos ángulos.

Para dar soporte a la experimentación con formas nuevas de incorporar paralelismo al desarrollo de software es necesario el desarrollo de herramientas adecuadas que permitan el análisis de las ejecuciones obtenidas. En este trabajo se propone una herramienta para el estudio de programas paralelos dentro del lenguaje de programación Haskell. Esta permite al programador observar directamente qué computaciones se van a paralelizar, dándole un mayor entendimiento sobre la ejecución de su programa

Agradecimientos

Gracias a Mauro y a Exe por darme un lugar más a su par, brindarme grandes cantidades de tiempo, y muchísima paciencia. Agradecer además a todo el plantel docente que forman la gran LCC, particularmente a Ana, Raúl y Dante que siempre están dispuestos a ayudar y están abiertos a opiniones sobre la carrera. Quería además, agradecer a mis compañeros, Mariano, Lucía, Esteban, César, Javier, Julián y Federico, por haberme acompañado a lo largo de la carrera, compartiendo apuntes, tiempo, asados, salidas, Antares, y momentos para distenderse después de horas de estudio. Sin ellos la carrera habría sido más aburrida.

Y por último pero no menos importante, a mi familia que me apoyó durante todos los pasos y me permitió dedicarme a estudiar de la forma más cómoda posible.

Índice general

Capítulo 1

Introducción

La presencia de procesadores multinúcleo en la mayoría de los dispositivos condiciona al desarrollador a escalar sus programas mediante la ejecución paralela. Debido a la necesidad de incorporar paralelismo en los programas se han extendido los lenguajes de programación, o bien se han desarrollado bibliotecas, con diversos grados de éxito. El estudio de las formas de incorporar de manera simple y eficiente la programación paralela al desarrollo de software sigue siendo objeto de investigación. En particular, en el lenguaje de programación Haskell (Marlow 2010) se han desarrollado una variedad de extensiones, bibliotecas, y abstracciones, que permiten encarar el problema de la programación paralela desde diversos ángulos.

El desarrollo de herramientas adecuadas que permitan el análisis de las ejecuciones obtenidas es necesario al momento de dar soporte a la experimentación con nuevas formas de incorporar paralelismo.

En este trabajo se desarrolla un lenguaje de dominio específico embebido en Haskell para permitir la simulación simbólica de programas paralelos. Se espera permitirle al programador poder observar directamente qué computaciones se van a paralelizar, permitiéndole tener un mayor entendimiento sobre la ejecución de su programa. Se presenta el desarrollo de la herramienta *Klytius*, donde se implementa un lenguaje de dominio específico destinado a representar programas paralelos, y se definen representaciones gráficas para la estructura de programas paralelos.

El resto del documento está organizado de la siguiente forma: en lo que resta de la introducción se presenta el contexto histórico para dar fundamentos a la necesidad de incorporar paralelismo a los programas, y las consecuencias de hacerlo. En el capítulo 2, se introduce al lector cómo paralelizar dentro de los lenguajes funcionales, particularmente el paralelismo básico dentro de Haskell. En el capítulo 3, se muestra el uso de la herramienta *Klytius*, y en el capítulo 4 explicamos cómo fue implementada. Las conclusiones y trabajos futuros se discuten en el capítulo 5.

1.1. Historia del CPU multinúcleo

En 1971 Intel presenta el primer microprocesador de la historia (Mazor, Member y Noyce 1995), el **Intel 4004**. Éste era un pequeño chip del tamaño de una uña, el cual vendría a reemplazar las computadoras que en ese momento eran del tamaño de habitaciones. Este chip se incorporó rápidamente al mercado en forma de máquinas expendedoras de comida, controladores de ascensores, control de semáforos e instrumentación médica, entre otros instrumentos electrónicos pequeños.

Actualmente los microprocesadores, presentes en la mayoría de los dispositivos electrónicos, no sólo son estructuras más complejas, con un poder computacional enorme en comparación a los 4-bits del Intel 4004, y con capacidad para manejar varios núcleos de procesamiento, sino que además poseen funciones adicionales, como ser, unidades de punto flotante, memoria caché, controladores de memoria, y preprocesadores multimedia. Sin embargo, siguen siendo en esencia iguales al primer microprocesador, un chip semiconductor que realiza las computaciones principales del sistema (Borkar y Chien 2011).

Dentro de dichos avances, el más notable fue el incremento de la capacidad computacional de los microprocesadores, debido a que al aumentar la velocidad de cómputo el campo de aplicación se aumentaba y les permitía adquirir nuevas funciones. El incremento en la velocidad de cómputo de los microprocesadores se debió principalmente a una técnica denominada **transistor-speed scaling** presentada por R. Dennard (Dennard y col. 1974). Dicha técnica permite reducir la dimensión de los transistores un 30 % en cada generación[1] manteniendo el campo de energía constante, dando espacio a aumentar el poder computacional.

Cada vez que se reducía el tamaño de los transistores, el área total se reducía a la mitad, permitiendo a los fabricantes duplicar la cantidad de transistores en cada generación. Además, para mantener los campos de energía constantes, al reducir el tamaño también se reducía el consumo de energía de los transistores. Es decir, en cada generación se obtenían microprocesadores un 40 % más rápidos consumiendo exactamente la misma cantidad de energía (aunque con el doble de transistores).

Sin embargo, al llegar a tamaños muy pequeños, los transistores no se comportan como un interruptor perfecto, ocasionando que una porción de la energía utilizada se pierda. Dicha pérdida de energía volvió prohibitivo continuar aumentando de esta manera la velocidad de cómputo de los procesadores, por lo que se volvió necesario buscar otra manera de aumentar el poder computacional.

Como forma alternativa de aumentar la capacidad de procesamiento, se optó por retomar la idea de aumentar la cantidad de unidades de procesa-

[1] Aproximadamente 2 años.

miento (núcleos) dentro de un mismo procesador (Noguchi, Ohnishi y Morita 1975).

Problemáticas del Paralelismo

Cuando se aumenta la frecuencia de los microprocesadores se ejecutan **exactamente las mismas** instrucciones más rápido. Es decir, se ejecutan con mayor velocidad **los mismos algoritmos** sin modificación alguna. Dejando al programador simplemente la tarea de conseguir hardware más potente para que los programas funcionen más rápido, librándolo de modificar su código o pensar en cómo administrar las computaciones de su programa.

Al aumentar las unidades de procesamiento se da lugar a la posibilidad de ejecutar varias instrucciones de manera simultánea, es decir, permite *paralelizar* la **ejecución** de los programas. Esto, en teoría, permite multiplicar el poder de cómputo, aunque como veremos, en la práctica no es tan fácil obtener un mayor rendimiento. Esto sucede por dos razones:

- No todo es paralelizable. Hay algoritmos que son fácilmente paralelizables, como por ejemplo, imaginemos que tenemos que pintar una habitación. Si un pintor puede pintarla en 16 hs, 4 pintores cada uno con sus herramientas pueden dividirse el trabajo y pintarla en 4 hs. En cambio, hay tareas que son inherentemente secuenciales, es decir, que no se van a poder paralelizar completamente sino que tienen al menos un proceso el cual se tiene que realizar de manera secuencial e indivisible. Por ejemplo, supongamos que tenemos que cocinar, a un cocinero le llevará 4 hs, y si agregamos otro cocinero más para ayudarlo, podrían cocinar mas rápido pero no exactamente el doble de rápido, debido a que hay acciones que se tienen que llevar a cabo que no pueden ser divididas. Por ejemplo, no pueden batir la preparación los dos juntos.

- Paralelizar no es *gratis*. Aún dados algoritmos que son totalmente paralelizables, deberemos saber cómo dividiremos el algoritmo, cuantos núcleos hay disponibles, cómo distribuiremos las distintas tareas en los diferentes núcleos y cómo se comunican entre ellos en el caso que sea necesario. Por ejemplo, en el caso de los pintores y la habitación, deberemos tener la disponibilidad de los 4 pintores, pensar en cómo dividir el trabajo, contar con 4 sets de herramientas, etc.

Como resultado, el programador debe modificar su código para incorporar la posibilidad de ejecutar su programa utilizando todos los núcleos disponibles.

1.2. Modelos de Paralelismo

Hoy en día prácticamente todos los dispositivos poseen procesadores multinúcleo, por lo que la programación paralela se ha vuelto una necesidad. Para

explotar todo el hardware subyacente el programador debe *pensar* de manera paralela, obteniendo beneficios en velocidad, con la desventaja que el diseño del software se torna más complicado. Se debe tener en cuenta dos aspectos: 1. intrínseco al diseño del algoritmo, identificar qué tareas se pueden realizar en paralelo, analizar la dependencia de datos, etc; 2. hardware disponible, cantidad de núcleos disponibles, cómo distribuir las diferentes tareas y establecer canales de comunicación entre ellos.

La programación paralela introduce nuevas problemáticas, como por ejemplo, dado que en la mayoría de los lenguajes de programación el programador no tiene acceso a las políticas de *scheduling*, es decir, no puede asegurar un orden de ejecución sobre diferentes *hilos* creados, esto produce que el programador deba garantizar la correcta ejecución sin importar el orden en que se ejecuten (Peyton Jones 1989).

Para facilitar la tarea de organizar las computaciones de los programas, se han planteado diferentes modelos de paralelismo, entre los cuales el más destacado es el modelo de Flynn.

Michael J. Flynn (Flynn 1966) propone utilizar una clasificación de sistemas computacionales en base a dos factores, la cantidad de instrucciones que ejecuta el microprocesador, y cómo es accedida la información:

- **Single Instruction Stream, Single Data Stream***(SISD)*: procesadores compuestos por un solo núcleo, el cual ejecuta una sola instrucción sobre un segmento de datos particular. Un ejemplo son los procesadores con un solo núcleo.

- **Single Instruction Stream, Multiple Data Stream***(SIMD)*: procesadores de varios núcleos donde cada uno de ellos ejecuta una sola instrucción (la misma para todos) aunque cada uno accede a un segmento de datos diferente. Las unidades de procesamiento gráfico (GPU) siguen este modelo.

- **Multiple Instruction Stream, Single Data Stream***(MISD)*: procesadores de varios núcleos que permiten realizar distintas tareas sobre un solo segmento de datos en particular. No se encuentran ejemplos concretos de este modelo de procesadores.

- **Multiple Instruction Stream, Multiple Data Stream***(MIMD)*: procesadores de varios núcleos donde cada uno de ellos es totalmente independiente del otro, pudiendo ejecutar diferentes instrucciones cada uno y acceder a diferentes segmentos de datos. Dentro de esta categoría se encuentran los microprocesadores multinúcleos actuales.

Si bien el modelo de Flynn es el más establecido en la literatura, se han propuesto y presentados otros con mayor nivel de detalle, como ser el trabajo de Skillicorn y Talia (Skillicorn y Talia 1998).

En los centros de cómputo de alto rendimiento el paralelismo se utiliza a través de computaciones distribuidas, el cual es modelado mediante sistemas de intercambio de mensajes. Un ejemplo concreto es *Message Passing Interface* (MPI) (Forum 2012). Se basa en proveer un mecanismo de comunicación entre dos procesos para que estos puedan comunicarse libremente. Dicho mecanismo se puede implementar de diferentes maneras, aunque en general se utilizan primitivas de *concurrencia*.

1.3. Paralelismo en Lenguajes Funcionales Puros

Los lenguajes funcionales puros nos permiten representar las computaciones en términos de funciones puras, es decir, nos permiten presentar a un programa como una función que obtiene toda la información necesaria a través de sus argumentos y devuelve algún valor como resultado de su computación, sin utilizar información externa a ella que no este explícitamente dentro de sus argumentos, ni modificar nada fuera del resultado que otorga al finalizar. Por lo tanto todo comportamiento se vuelve explícito. Esto permite que sea más fácil razonar sobre los programas y estudiar su comportamiento.

Particularmente al tener computaciones puras la ejecución se vuelve determinista, el resultado **sólo** depende de sus argumentos. Por lo tanto, todo programa que se ejecute secuencialmente tendrá el mismo resultado que su versión paralela con los mismos datos de entrada, resaltando que el paralelismo es una herramienta para arribar a la solución más rápido. Esto nos permite razonar sobre los programas paralelos de la misma manera que los secuenciales y depurarlos sin la necesidad de efectivamente ejecutarlos en paralelo. Permite al programador independizarse de las políticas de *scheduling*, ya que el orden de las diferentes tareas que componen al programa puede variar, influyendo posiblemente en el tiempo de ejecución, pero no en el valor resultado de la misma. No existe la posibilidad de que haya *deadlock*, excepto en condiciones donde su versión secuencial no termine debido a dependencias cíclicas (Roe 1991).

Los lenguajes funcionales, nos permiten delegar todos los aspectos de coordinación al *runtime*, es decir, libran al programador de tareas como: distribuir las diferentes tareas entre los núcleos, establecer canales de comunicación entre los procesos, la generación de nuevas tareas e incluso es independiente tanto de la arquitectura como de las políticas de *scheduling*. Esto se debe a que los lenguajes funcionales utilizan como mecanismo de ejecución abstracto la *Reducción de Grafos* (Johnsson 1984; Augustsson 1984; Kieburtz 1985) el cual es fácil de modificar para permitir la evaluación paralela de las computaciones (Augustsson y Johnsson 1989). Estos sistemas representan a las computaciones como grafos donde cada hoja es un valor y los nodos, que no son hojas, representan aplicaciones. Se basan en ir reduciendo cada nodo a su valor correspondiente, posiblemente generando que otros subgrafos sean evaluados en

el proceso.

En particular, Haskell es un lenguaje funcional puro con evaluación *perezosa*. Las computaciones son evaluadas en el momento que su valor sea necesario o no evaluadas si no lo son. Al momento de incorporar paralelismo se requiere otorgar la posibilidad de evaluar expresiones sobre los distintos núcleos disponibles, posiblemente sin que esas expresiones se necesiten en ese momento, por lo que será necesario prestar atención cuando se trata de paralelizar una computación. Se deberá forzar explícitamente la evaluación que se requiera para una expresión dada, ya que sino se suspenderá su evaluación y sólo se evaluará (de manera secuencial) cuando se necesite su valor.

1.4. Herramientas para el Análisis de Programas Paralelos

Encontrar problemas en el rendimiento de los programas es una tarea muy complicada, que requiere de una gran experiencia previa y conocimiento de cómo funciona el compilador y el entorno de ejecución. Para aliviar esta tarea, se utilizan herramientas para realizar análisis sobre computaciones paralelas.

Las herramientas de *profiling* son muy utilizadas debido a que el comportamiento de los programas paralelos depende del estado general del sistema. Estas, con ayuda del *runtime*, toman muestras del comportamiento del programa durante su ejecución, para luego analizar los resultados e investigar el origen de problemas de rendimiento, como ser, trabajo de los procesadores desbalanceado o cuellos de botella.

Actualmente Haskell cuenta con *ThreadScope*, es una interfaz gráfica para el sistema de *profiling* de GHC (Jones Jr, Marlow y Singh 2009; Marlow, Peyton Jones y Singh 2009; Harris, Marlow y Jones 2005), que permite dar un análisis de cómo fue el comportamiento del programa, basada en los eventos que fueron detectados en el ejecución (*Analysing Event Logs* 2015). Si bien se muestra el estado del sistema, momento a momento, no hay una relación directa (observable) entre la construcción del programa paralelo diseñado por el programador y la evaluación de las computaciones. Es decir, el programador puede observar el comportamiento de todo su programa, pero también parte de su tarea es determinar, a medida que transcurre la ejecución, a qué parte concreta de su programa se debe el comportamiento observado.

La extensión de Haskell, Eden (Loogen 2012), presenta una herramienta previa a la creación de *ThreadScope*, llamada EvenTV (Berthold y col. s.f.), que de forma muy similar a *ThreadScope* muestra gráficamente, momento a momento, cómo fueron utilizados los diferentes procesadores. La herramienta EdenTV utiliza información provista por el monitor de rendimiento *Eden Tracing*.

Dentro del paradigma de programación imperativa se encuentran una gran variedad de herramientas de análisis de programas paralelos, donde en su ma-

yoría son herramientas que permiten monitorear el comportamiento del sistema al momento de ejecución, como ser, el manejo de memoria, el uso de los diferentes núcleos, o son herramientas de *profiling*, como ser, ParaGraph, XPVM, Vampir, Scalasca y TUA. Por ejemplo, Vampir es una herramienta que permite monitorear el comportamiento de la ejecución de programas paralelos. Puede ser utilizado en diferentes lenguajes como ser, C o Fortan, y con diferentes librerías como MPI, OpenMP o CUDA. Para mayor información sobre las diferentes herramientas de análisis de programas paralelos, Al Saeed, Trinder y Maier muestran una comparación entre las herramientas antes mencionadas (Al Saeed, Trinder y Maier 2013).

A su vez, modelos específicos de programación paralela permiten realizar análisis inherentes al modelo, por lo que se encuentran aun más herramientas de monitoreo y *profiling*, como ser, sobre el modelo *MPI* se pueden utilizar, *DEEP/MPI*, *MPE*, *Pablo Performance Analysis Tools*, *Paradyn*, etc. Estas herramientas se basan en detectar información sobre cuándo se utilizan las llamadas a las librerías de *MPI*, qué *mensajes* se intercambian mediantes los canales de comunicaciones establecidos, etc. Para mayor información sobre comparaciones entre dichas herramientas ver Moore y col. 2001.

Todas las herramientas mencionadas anteriormente realizan monitoreo de la actividad del sistema, o bien, realizan *profiling* sobre la ejecución del programa. En este trabajo se propone analizar la estructura de computaciones paralelas. Dicha estructura se construye de forma dinámica, pero ésta no depende del estado del sistema, sino que depende del código escrito. Ninguna de las herramientas antes mencionadas permiten realizar éste análisis.

1.5. Contribuciones

En este trabajo se desarrolla un lenguaje de dominio específico embebido en Haskell que permite la simulación simbólica de programas paralelos. Esta permite al programador observar directamente qué computaciones se van a paralelizar, dándole un mayor entendimiento sobre la ejecución de su programa. Además, se presenta el desarrollo de la herramienta *Klytius*, donde se implementa el lenguaje de dominio específico destinado a representar programas paralelos. Éste lenguaje provee un conjunto de operadores que permiten construir programas paralelos simplemente utilizando operadores similares a los combinadores de paralelismo puro dentro de Haskell, pero a la vez otorgando la posibilidad de realizar gráficos que representan la evaluación de las computaciones.

Actualmente la herramienta se encuentra alojada en un repositorio *Git* público, `https://bitbucket.org/martinceresa/klytius/`. Se presenta como un paquete a instalar por la herramienta *Cabal* de la plataforma de Haskell.

Capítulo 2

Paralelismo en Lenguajes Funcionales

En este capítulo se realiza una breve introducción a la programación paralela dentro del paradigma de programación funcional, particularmente dentro de Haskell. Se exploran las diferentes posibilidades de particionar un programa, se analizan los combinadores básicos de paralelismo puro dentro de Haskell, y el impacto de la evaluación *perezosa* sobre la programación paralela. Por último, se construye un modelo más abstracto sobre los combinadores básicos, que permite separar los algoritmos del comportamiento dinámico.

2.1. Particionamiento de programas

Dada la posibilidad de ejecutar diferentes computaciones sobre diferentes núcleos, se introduce un nuevo requerimiento dentro del diseño del algoritmo, que es decidir cuáles son las computaciones que se van a distribuir, o cómo *particionar* los programas para arribar más rápido la solución.

Dentro de los lenguajes funcionales existe un gran espectro de opciones para explotar el paralelismo presente en los programas. Podemos destacar dos estrategias básicas para decidir cómo particionar un programa (Hammond 1994):

- **particionamiento implícito:** El compilador luego de un análisis estático del código es quien decide cuáles son las tareas a realizar en paralelo. Este tipo de paralelismo automático puede generar tareas muy pequeñas en comparación con el costo de realizarlas en paralelo e incluso, por dependencia de datos, podría no generar paralelismo alguno.

- **particionamiento explícito:** El programador es quien decide cuáles tareas se realizarán en paralelo, debiendo tener en cuenta el tamaño de las tareas a paralelizar, el tiempo de ejecución de las mismas, la cantidad de núcleos disponibles, etc. Si bien el resultado es bueno, el programador

```
left pid = 2*pid
right pid = 2*pid+1

parsum1 f g x = (f(x) $on 0) + (g(x) $on 1)

parsum2 f g x = (f(x) $on left($self)) + (g(x) $on right($self))
```

Código 2.1: Ejemplo de anotaciones

debe dedicarle más tiempo y esfuerzo al diseño del algoritmo mezclando el código con primitivas de paralelismo del lenguaje.

Los lenguajes que utilizan particionamiento implícito definen un lenguaje de bajo nivel para el control de paralelismo, el cual no está disponible para uso directo por parte del programador, sino que, es el compilador quien detecta estáticamente cómo particionar un programa e inserta automáticamente el lenguaje de control de paralelismo. Un ejemplo de lenguaje de control es el uso de *combinadores seriales* (Hudak y Goldberg 1985). El enfoque implícito tiene como ventaja que el programador no tiene contacto con el paralelismo, por lo que notará que sus programas utilizan el hardware disponible en la computadora de manera automática. El problema con estos métodos automáticos es que el paralelismo obtenido tiende a ser demasiado *granular*, es decir que las computaciones son relativamente pequeñas, por lo que el costo de generar el paralelismo (crear el hilo, establecer canales de comunicación, etc) puede ser mayor que el beneficio obtenido.

Por otro lado, los lenguajes explícitos utilizan técnicas como anotar en el programa el control del paralelismo (Hudak 1986). Estas anotaciones forman parte de un *meta-lenguaje* dedicado a controlar el paralelismo, donde el programador es el encargado de distribuir las computaciones en los distintos procesadores. Por ejemplo, utilizando el operador infijo `$on` para indicar en qué procesador queremos evaluar la expresión, podemos escribir la suma de dos expresiones de forma paralela en dos procesadores, como se muestra en el Código 2.1. En la primer definición `parsum1`, se establece que `f(x)` se evaluará en el procesador 0 y `g(x)` en el procesador 1, mientras que en la segunda definición, `parsum2`, los identificadores de los procesadores se generan dinámicamente en base al procesador actual (`$self`).

Los lenguajes de particionamiento explícito otorgan al programador mayor control sobre el paralelismo, con la desventaja que el código resultante es una mezcla (muy acoplada) entre el código del algoritmo y código de control. Incluso hay que modelar el hardware subyacente, como se observa en el Código 2.1, donde utilizamos una estructura de árbol binario para acceder a los identificadores de los procesadores.

(a) **par** a b (b) **pseq** a b

Figura 2.2: Comparación gráfica entre **par** y **pseq**

2.2. Paralelismo Semi-Explícito

Un enfoque intermedio entre el paralelismo explícito donde toda la coordinación, comunicación y control es de manera explícita (diseñado por el programador) y el paralelismo implícito donde no se tiene control alguno sobre el paralelismo, es el *particionamiento semi-explícito*. En este enfoque, el programador introduce ciertas anotaciones en el programa pero todos los aspectos de coordinación son delegados al entorno de ejecución (Loidl y col. 2008).

El lenguaje de programación Haskell, cuenta con dos combinadores básicos de coordinación de computaciones.

```
par , pseq  :: a → b → b
```

Ambos combinadores son los únicos con acceso al control del paralelismo del programa[1].

Denotacionalmente ambas primitivas son proyecciones en su segundo argumento.

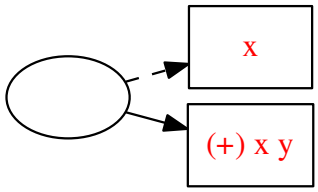
```
par  a b = b
pseq a b = b
```

Operacionalmente **pseq** establece que el primer argumento **debe** ser evaluado antes que el segundo, mientras que **par** indica que el primer argumento **puede** llegar a ser evaluado en paralelo con la evaluación del segundo. Utilizando los combinadores básicos **par** y **pseq** el programador establece el **qué** y **cómo** paralelizar sus programas. Podemos representarlo de manera gráfica como se muestra en la Figura 2.2: **par** marca la computación a para ser evaluada en paralelo (línea punteada), y continúa con la evaluación de b (línea continua), mientras que, **pseq** establece que para evaluar b primero se deberá evaluar a. Ambos combinadores proponen la evaluación de sus argumentos: **par** a b propone que a se evalúe en otro núcleo, de forma independiente de b, mientras que **pseq** a b establece que antes de evaluar b se debe evaluar a.

[1] Si bien existen otros métodos, en general no se recomienda mezclar varias técnicas.

```
parsuma :: Int → Int →
    Int
parsuma x y = par x (x+y)
```

(a) Suma paralela



(b) Representación gráfica

Cuadro 2.3: Suma utilizando par

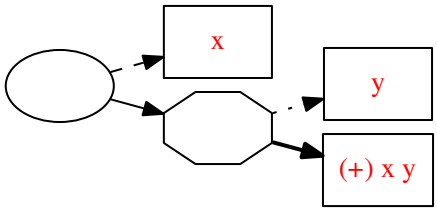
Toda computación, a, marcada por **par** a b, será vista por el entorno de ejecución como una oportunidad de trabajo a realizar en paralelo, la cual se denomina *spark*. La acción de *sparking* no fuerza inmediatamente la creación de un *hilo*, sino que es el *runtime* el encargado de determinar cuáles *sparks* van a ejecutarse, tomando esta decisión en base al estado general del sistema, como ser, si encuentra disponible a algún núcleo en el cual ejecutar la evaluación del *spark*. El *runtime* es también el encargado de manejar los detalles de la ejecución de computaciones en paralelo, como ser, la creación de hilos, la comunicación entre ellos, el *workload balancing*, etc.

Intentar paralelizar la evaluación de una expresión se puede realizar sencillamente, como se muestra en el fragmento de código presentado en el Cuadro 2.3a, donde se busca paralelizar la evaluación de los enteros que vienen como argumentos de parsuma, utilizando el operador **par**, y luego devolver la suma de los mismos. Podemos observar gráficamente la expresión a evaluar en el Cuadro 2.3b, como al comenzar a evaluar la computación se genera un *spark* para evaluar la expresión de x, y continúa con la evaluación de la expresión ((+) x y). Pero el programa mostrado en el Cuadro 2.3a, presenta un error sutil. La función (+) tal como se encuentra definida en el *preludio* de Haskell, es estricta en su primer argumento, por lo que evaluará primero a x. Es decir que el valor del *spark* generado por el combinador **par** (para la evaluación de x) será necesitado inmediatamente al evaluar ((+) x y), **eliminando** toda posibilidad de paralelismo. No puede haber paralelismo debido a que si todavía no fue evaluado, será evaluado por el mismo hilo de la suma, y si está en proceso de evaluación, la computación esperará hasta que éste haya terminado su evaluación, y luego continuará con la evaluación de y.

Una posible solución a este problema es intercambiar el orden de los argumentos de la función (+) (ya que sabemos que es conmutativa) por (y+x), pero esta idea no es aplicable para cualquier función y puede ser que dada una función no sepamos el orden de evaluación que ésta impone sobre sus

```
parsuma :: Int → Int →
    Int
parsuma x y =
   par x (pseq y (x+y))
```

(a) Suma paralela correcta



(b) Representación gráfica

Cuadro 2.4: Suma utilizando **par** y **pseq**

argumentos.

El combinador **pseq** permite establecer el orden de evaluación y de esta manera *secuencializar* la evaluación de (x+y) con el valor de y como se muestra en el Código 2.4a. De esta manera se logra tener paralelamente dos hilos, uno para la evaluación de x y otro para la evaluación de y que además realizará la suma de ambos cuando sus argumentos hayan sido evaluados, independientemente del orden que impone la suma sobre sus argumentos. Gráficamente en la figura presente en el Cuadro 2.4b, observamos como al comenzar la evaluación de la expresión parsuma x y, se genera un *spark* para la evaluación de x, mientras se evalúa la expresión **pseq** y (x + y), donde se evalúa primero a y y por último (x + y).

2.3. Profundidad de la Evaluación

Haskell, por defecto, evalúa los valores a forma normal débil a la cabeza, es decir, evalúa una computación hasta encontrar el primer constructor del tipo correspondiente. Por ejemplo, la forma normal débil a la cabeza de un valor tipo lista es la lista vacía ([]) o una lista con al menos un elemento (_ : _). Es decir, la política de evaluación de Haskell es realizar el mínimo trabajo necesario, las expresiones son calculadas cuando son necesarias para continuar con la ejecución del programa.

Los combinadores **par** y **pseq**, evalúan sus argumentos a forma normal débil a la cabeza, por lo que el programador deberá asegurarse que los *sparks* realmente estén realizando el trabajo que se espera que realicen. Esta problemática es una de las principales causas de que la programación paralela en lenguajes de evaluación perezosa sea difícil, como es el caso de Haskell. Exploremos un ejemplo para identificar la problemática, y presentar posibles soluciones.

La programación funcional incentiva a utilizar ciertos patrones como los

```
parmap (+1) [23,4,79014] = par (23+1) ((23+1) : parmap (+1)
    [4,79014])
parmap (+1) [4,79014]    = par (4+1) ((4+1): parmap (+1) [79014])
parmap (+1) [79014]      = par (79014+1) ((79014+1) : [])
```

Código 2.5: parmap (+1) [23,4,79014]

funtores, los cuáles nos permiten aplicar una función a los valores de un tipo de datos sin modificar la estructura, es decir, nos permite modificar los valores que contiene la estructura, pero no la estructura en sí. Particularmente al trabajar sobre listas de elementos, es muy común utilizar la función **map**, la cual nos permite aplicar una función a cada uno de los elementos de la lista.

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Utilizando la función **map**, podemos manipular los elementos de una lista, sin pasar por estructuras intermedias, ni cambiar de representación. Por ejemplo, para sumarle uno a cada uno de los elementos de una lista de enteros, podemos escribir:

```
sumauno xs = map (+1) xs
```

El mapeo de una función sobre la lista, es una función altamente paralelizable. Dado que se aplica una misma función de forma independiente a cada uno de los elementos de la lista, podemos paralelizar la evaluación de las aplicaciones. De forma muy intuitiva, una primer aproximación es utilizar el combinador **par**, como se muestra a continuación.

```
parmap :: (a → b) → [a] → [b]
parmap f [] = []
parmap f (x:xs) = par (x') (x' : (parmap f xs))
    where
        x' = f x
```

Utilizando parmap, a medida que se va recorriendo la lista, se indica que la evaluación de las expresiones x' (correspondiente a la evaluación de f x) se pueden realizar de forma paralela. Por ejemplo, si expandimos la expresión generada por parmap (+1) [23,4,79014] (ver Código 2.5), vemos que se van generando *sparks* que evalúan cada una de las aplicaciones de la función sobre los elementos de la lista. Podemos observar el mismo comportamiento de manera gráfica en la Figura 2.6.

Sin embargo, esta implementación de parmap **no** funciona como esperamos para todos los tipos ya que en algunos casos los *sparks* generados no realizarán suficiente trabajo. Para el tipo **Int**, el comportamiento es correcto ya que su forma normal débil a la cabeza coincide con su forma normal, y de esta manera los *sparks* realizan el trabajo de evaluar completamente los valores. Para hacer evidente este detalle, supongamos que tenemos ahora una lista de

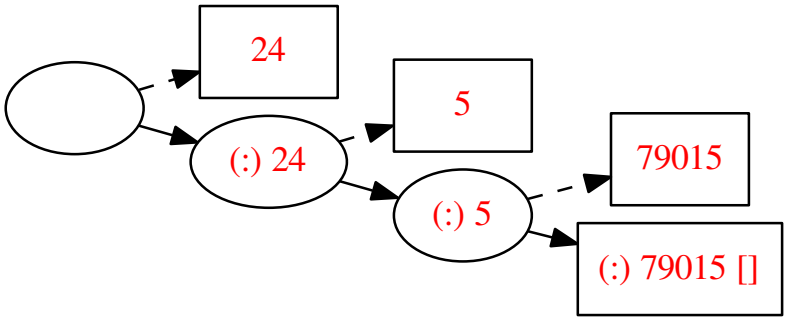


Figura 2.6: parmap (+1) [23,4,79014]

```
maybemap :: (a → b) → Maybe a → Maybe b
maybemap f Nothing  = Nothing
maybemap f (Just x) = Just (f x)
```

Código 2.7: Definición de maybemap

posibles enteros, [**Maybe Int**], a los cuáles queremos sumarle uno. Definiendo una función similar a **map** pero para elementos de tipo **Maybe** (ver Código 2.7), podemos definir la siguiente expresión.

```
parmap (maybemap (+1)) [Just 23, Just 4, Just 79014]
```

Podemos observar gráficamente en la Figura 2.8, que los *sparks* son creados para la evaluación de expresiones de elementos **Maybe** a. El problema se origina dado que la forma normal débil a la cabeza de elementos de tipo **Maybe** a es, **Nothing** o **Just** x, donde x **no** es evaluado. Es decir, si seguimos el gráfico, los *sparks* van a evaluar hasta encontrar el constructor **Just** y luego terminar su evaluación. Si bien en el gráfico los valores se muestran, son simplemente para identificar las diferentes expresiones, y no significa que los valores sean evaluados realmente.

El combinador **pseq** permite no sólo establecer el orden de las computaciones, sino también, establecer la profundidad a la cual los elementos serán evaluados, forzando de alguna manera la evaluación. Podríamos introducir el combinador dentro de la función parmap, como se muestra a continuación.

```
parmap f (x:xs) = par (pseq (f x) ()) ((f x) : (parmap f xs))
```

Pero estamos en exactamente la misma situación que se nos presentó anteriormente, debido a que **pseq** simplemente evaluará a forma normal débil a la

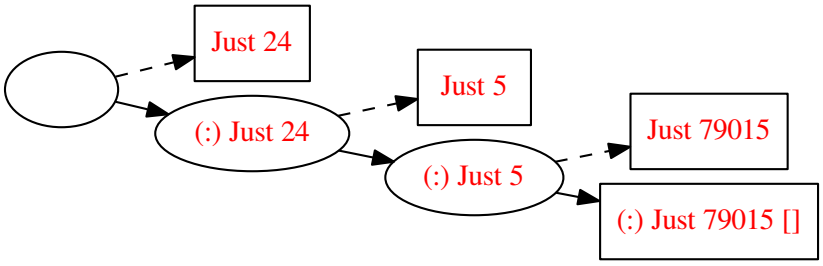


Figura 2.8: parmap (maybemap (+1)) [**Just** 23,**Just** 4,**Just** 79014]

cabeza sus argumentos, del mismo modo que hace **par**.

Debemos aplicar el combinador **pseq** al elemento *dentro* del tipo **Maybe** a. Para esto definimos una función posMaybe que fuerza la evaluación de sus elementos:

```
posMaybe :: Maybe b → ()
posMaybe Nothing = ()
posMaybe (Just x) = pseq x ()
```

Notar que el tipo de retorno de la función posMaybe es (), lo que indica que el valor de retorno **no** nos interesa, la expresión tiene el efecto de forzar la evaluación sin modificar ninguno de los valores que observa. Además, dicha función sólo nos permite evaluar a forma normal débil a la cabeza el valor dentro de **Maybe** b, en el caso de ser un valor de la forma **Just** x, por lo que deberemos redefinir la función parmap especializando su tipo de retorno a **Maybe** b como se muestra a continuación.

```
maybeapp :: (a → Maybe b) → [a] → [Maybe b]
maybeapp f []     = []
maybeapp f (x:xs) = par (posMaybe (f x)) ((f x) : (maybeapp f xs))
```

Podemos observar de manera gráfica (ver Figura 2.9) el resultado de la evaluación de maybeapp (maybemap (+1)) [**Just** 23, **Just** 4, **Just** 79014], donde al momento de evaluar los *spark* dedicados a la evaluación de expresiones de la forma **Just** x, también fuerzan la evaluación de x, gracias a la aplicación de la función posMaybe.

Si bien esta solución realiza correctamente el trabajo que buscamos para elementos de tipo **Maybe Int**, no realiza el trabajo correcto en el caso que el tipo dentro de **Maybe** tenga constructores, es decir, donde su forma normal débil a la cabeza no coincida con su forma normal. Por ejemplo, **Maybe** (**Maybe Int**), donde nos encontramos en un caso similar al comenzar a explorar la profundidad de la evaluación que realizan **par** y **pseq**. Es posible continuar aplicando la función posMaybe, y continuar especializando el tipo de maybeapp, pero no

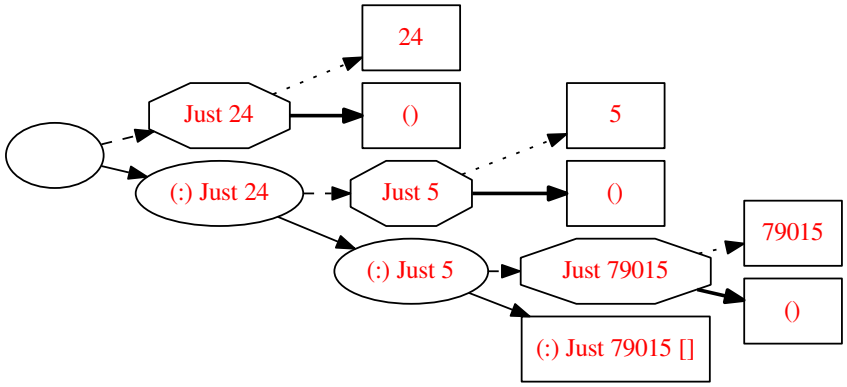


Figura 2.9: maybeapp (maybemap (+1)) [**Just** 23, **Just** 4, **Just** 79014]

es una solución definitiva. Siguiendo este camino deberíamos definir diferentes funciones especializadas en base a parmap, donde a los *sparks* generados se les aplica una función para forzar la evaluación deseada.

La solución a este problema es definir la función genparmap donde tomamos como argumento una función que indica cómo se tiene que realizar la evaluación de los *sparks* generados de forma similar a la función posMaybe, como se muestra a continuación.

```
genparmap :: (b → ()) → (a → b) → [a] → [b]
genparmap ev f []     = []
genparmap ev f (x:xs) =
    par (ev (f x))
        ((f x) : (genparmap ev f xs))
```

Particularmente, utilizando las funciones genparmap y posMaybe, podemos redefinir la función maybeapp como se muestra a continuación.

```
maybeapp :: (a → Maybe c) → [a] → [Maybe c]
maybeapp f xs = genparmap posMaybe f xs
```

Utilizando la función genparmap podemos evaluar la aplicación de una función sobre los elementos de una lista en paralelo, pero la utilización de los combinadores **par** y **pseq** comienzan a obscurecer el código y se necesita entender completamente si los *sparks* generan el trabajo esperado. Obteniendo además un programa donde la definición del algoritmo (**map** en nuestro ejemplo) se encuentra muy ligada a cómo el programador quiere que se evalué.

2.4. Estrategias

En la sección anterior analizamos el comportamiento de los combinadores **par** y **pseq**, y cómo éstos evalúan sus argumentos. Mostramos que el programador debe tener un conocimiento completo sobre los valores que va a utilizar y forzar la completa evaluación de las computaciones dedicadas a la evaluar dichos valores. A su vez, el código se plaga de los combinadores de paralelismo, obscureciendo el algoritmo. En esta sección se plantea la utilización de *Estrategias* como método para separar la definición del algoritmo de la descripción de su evaluación. A su vez, permite esclarecer y definir correctamente el trabajo necesario para la evaluación completa de un tipo de datos concreto.

Definimos como *comportamiento dinámico* a toda función que describa como organizar las computaciones, como ser, control del paralelismo (el uso de los combinadores **par** y **pseq**) o el grado de evaluación de una computación. Es decir, el comportamiento dinámico es toda función originada por el requerimiento de incorporar paralelismo y que no sea parte del algoritmo. La función posMaybe definida anteriormente es de comportamiento dinámico, ya que describe como evaluar los elementos de tipo **Maybe**, y sólo se requiere por su efecto en la evaluación.

A partir de **par** y **pseq** es posible construir un modelo más abstracto, donde se busca poder separar lo más posible el comportamiento dinámico de la definición de la función, simplificando la construcción de programas paralelos. Para el control del paralelismo y el grado de evaluación de expresiones dentro de Haskell, se introduce el concepto de *Estrategias*, funciones no estrictas polimórficas y de alto orden (Trinder, Hammond y col. 1998). Estas funciones nos permiten separar completamente la definición del algoritmo de su comportamiento dinámico. Las estrategias sufrieron pequeñas modificaciones, debido principalmente a detalles de implementación, hasta dar con la implementación actual definida por Simon Marlow (Marlow y col. 2010b). La estrategias se presentan entonces mediante el uso (interno) de la mónada Eval.

Una estrategia de evaluación es una función que especifica el comportamiento dinámico de un valor. No tiene contribución alguna al valor que está siendo computado por el algoritmo, sino que sólo se utiliza por su *efecto* en la evaluación del mismo.

Dentro de Haskell las estrategias son representadas como un tipo Strategy mostrado en el fragmento de Código 2.10, en conjunto con operaciones básicas. Las estrategias son presentadas utilizando la mónada **Eval**, el programador puede elegir utilizar su interfaz monádica o no. Utilizando esta definición, podemos definir:

1. la composición de estrategias, dot s2 s1, como la estrategia que realiza todo el comportamiento dinámico definido por s1 y luego todo el comportamiento dinámico definido por s2,
2. la aplicación de una estrategia sobre un valor, using x s, realiza todo el comportamiento dinámico definido por s sobre x, y luego retorna x,

```haskell
data Eval a = Done a

instance Monad Eval where
    return x = Done x
    Done x >>= k = k x

runEval :: Eval a → a
runEval (Done x) = x

type Strategy a = a → Eval a

dot :: Strategy a → Strategy a → Strategy a
s2 `dot` s1 = s2 . runEval . s1

using :: a → Strategy a → a
using x st = runEval (st x)

rpar :: Strategy a
rpar x = x `par` return x

rpseq :: Strategy a
rpseq x = x `pseq` return x

parList :: Strategy a → Strategy [a]
parList s []     = return []
parList s (x:xs) = do
    x'  ← rpar (s x)
    xs' ← (parList s xs)
    return (x':xs')
```

Código 2.10: Definición de estrategias

3. una estrategia parList s xs, que evalúa en paralelo el comportamiento dinámico definido por s para cada uno de los elementos de xs,
4. ciertas estrategias básicas como, rpar x que evalúa a x de forma paralela, o rpseq x que evalúa a x y luego continua normalmente.

Utilizando la interfaz descripta de estrategias, se puede ocultar el uso de la mónada **Eval**, librando al programador de tener que modificar su programa para el manejo de efectos monádicos.

Definiendo una nueva clase en Haskell NFData (ver Código 2.11) se puede representar el concepto de forma normal de un tipo abstracto de datos, como una función que fuerza la evaluación completa de una expresión de un tipo concreto, el cual no retorna elemento alguno. Por ejemplo, para una lista de elementos, su forma normal es valor resultante de la evaluación secuencial de sus elementos a forma normal.

Utilizando la clase NFData definida en el fragmento de Código 2.11, podemos redefinir la función parmap vista en la sección anterior, como se muestra a continuación.

```
class NFData a where
    rnf :: a → ()

instance NFData a ⇒ NFData [a] where
    rnf [] = ()
    rnf (x:xs) = rnf x `pseq` (rnf xs)

instance (NFData a) ⇒ NFData (Maybe a) where
    rnf Nothing  = ()
    rnf (Just x) = rnf x
```

Código 2.11: Evaluación de valores a formal normal

```
parmap :: (NFData b) ⇒ (a → b) → [a] → [b]
parmap f []     = []
parmap f (x:xs) = par (rnf (f x)) ((f x) : (parmap f xs))
```

La definición de la clase NFData b nos provee de la posibilidad de utilizar la función rnf sobre elementos de tipo b para forzar la completa evaluación de estos, de manera que los sparks generados por la función parmap realicen el trabajo esperado.

La clase NFData, nos permite además definir una nueva estrategia, rdeepseq, que evalúa una expresión a su forma normal.

```
rdeepseq :: NFData a ⇒ Strategy a
rdeepseq x = pseq (rnf x) (return x)
```

Al definir la evaluación a forma normal como una estrategia, nos permite combinarla con las definidas anteriormente, como ser parList (Código 2.10), y definir la evaluación paralela de una lista como se muestra a continuación.

```
parmap :: NFData b ⇒ (a → b) → [a] → [b]
parmap f xs = map f xs `using` parList rdeepseq
```

La función using nos permite separar totalmente el algoritmo (**map**) de su comportamiento dinámico (parList rdeepseq), permitiéndole al programador que pueda desarrollar por separado el algoritmo de la forma en que será evaluado. De esta manera, simplemente se tiene que definir una estrategia que identifique la reducción de una expresión a forma normal de un tipo determinado, y ya es posible asegurar que los *sparks* que son creados realizan el trabajo esperado.

Utilizando el modelo de estrategias se puede especificar el comportamiento dinámico por separado del algoritmo muy fácilmente, permitiendo paralelizar el algoritmo mediante una estrategia acorde a la necesidad.

2.5. Mónadas para el Paralelismo

Además de la mónada Eval, utilizada internamente por las estrategias, se encuentra la mónada Par (Marlow, Newton y Jones 2011). Internamente utiliza primitivas de concurrencia en conjunto con un mecanismo de comunicación

entre procesos basado en *IVars* (Arvind, Nikhil y Pingali 1989), que permite definir variables donde van a ser alojados los resultados de las computaciones, como es el caso de xf en el ejemplo mostrado debajo. En el trabajo presentado por Simon Marlow (Marlow, Newton y Jones 2011) se muestra que es posible conseguir un modelo de paralelismo totalmente determinístico basado en primitivas de concurrencia.

Por ejemplo, escribimos la función Fibonacci en ambas mónadas como se muestra a continuación.

```
parfib :: Int → Par Int
parfib n | n < 2 = return 1
parfib n = do
    xf ← spawn_ $ parfib (n-1)
    y  ← parfib (n-2)
    x  ← get xf
    return (x+y)
```

(a) Fibonacci utilizando la mónada Par

```
strfib :: Int → Int
strfib n | n < 2 = 1
strfib n = runEval $ do
    x ← rpar (strfib (n-1))
    y ← rseq (strfib (n-2))
    return (x+y)
```

(b) Fibonnaci utilizando la mónada Eval

Cuadro 2.12: Comparación entre mónadas Par y Eval

La función Fibonacci implementada como se muestra en el fragmento de código del Cuadro 2.12a, toma un entero, n, y devuelve fib n dentro de la mónada Par. Para los casos bases simplemente retorna 1, mientras que para $n >= 2$, genera un nuevo hilo al cual le asigna el trabajo de evaluar la expresión parfib (n−1), se realiza el calculo de parfib (n−2) y luego se suman. Internamente, como mencionamos en el párrafo anterior, la mónada Par utiliza *IVars*, las cuales son variables estrictas, es decir, sólo comunican valores en forma normal, obligando a que el nuevo hilo evalúe completamente la expresión. Mientras que en la implementación utilizando la mónada Eval (ver Cuadro 2.12b), simplemente utiliza de forma monádica las estrategias rpar y rseq.

En el código se observa que el programador está obligado a utilizar un estilo monádico de programación. Utilizando estrategias es posible reducir el impacto, restringiendo a que sólo se trabaje con mónadas al momento de definir nuevas estrategias. Por ejemplo, se define una estrategia de evaluación para la suma y luego se la aplica utilizando using, como se muestra en el fragmento de código a continuación.

```
parfib n = x + y `using` strat
where
    x = strfib (n-1)
    y = strfib (n-2)
    strat v = do rpar x; rseq y; return v
```

Capítulo 3

Observar Paralelismo con *Klytius*

En este capítulo se muestra cómo utilizar a la herramienta *Klytius* para el análisis de programas paralelos. Gracias a los gráficos generados por la herramienta, es posible analizar programas paralelos que utilizan los combinadores **par** y **pseq**.

Utilizando particionamiento semi-explícito el programador debe dar indicaciones sobre qué es lo que quiere paralelizar (**par**) y cómo ordenar las computaciones (**pseq**). A partir de estos combinadores básicos, se construye un modelo de más alto nivel, que nos permite separar el comportamiento dinámico del algoritmo, las estrategias. Pero el programador **no** tiene forma de observar directamente las construcciones dinámicas de sus programas. Esto lleva a que la construcción de programas paralelos en Haskell sea complicada, se necesite un gran entendimiento a bajo nivel y se base fuertemente en la experiencia previa.

Se presenta una nueva herramienta para complementar la visión que tiene el programador sobre su programa y exponer directamente el comportamiento dinámico del mismo. El objetivo es generar gráficos de forma automática que ilustren el comportamiento de las computaciones. Todos los gráficos del capítulo anterior, los cuales visualizan el comportamiento dinámico de una expresión, fueron generados con *Klytius*.

Repasaremos como utilizar a *Klytius*, dado que el proceso no se encuentra totalmente automatizado, por lo que es necesario que el usuario modifique levemente sus programas.

3.1. Operadores Básicos

Para poder graficar el comportamiento dinámico de un programa paralelo es necesario realizar algunas modificaciones. Los operadores básicos de paralelismo, se reemplazan por los operadores mostrados en el fragmento de

```
par     :: TPar a → TPar b → TPar b
pseq    :: TPar a → TPar b → TPar b
mkVar   :: a → TPar a

(<$>) :: (a → b) → TPar a → TPar b
(<*>) :: TPar (a → b) → TPar a → TPar b
```

Código 3.1: Combinadores básicos de paralelismo y operadores básicos de **TPar**

```
parsuma :: Int → Int →
    Int
parsuma x y =
    par x
    (pseq y
    (x + y))
```

(a) Suma paralela.

```
parsuma :: Int → Int → TPar Int
parsuma x y = par x'
                (pseq y' (x' + y'))
    where
        x' = mkVar x
        y' = mkVar y
```

(b) Suma paralela utilizando a *Klytius*.

Cuadro 3.2: Comparación entre el código original y el código modificado.

Código 3.1. Estos nuevos operadores permiten recopilar información del comportamiento dinámico de la evaluación de las expresiones.

Se introduce, además, el operador **mkVar**, el cual nos permite introducir valores dentro del tipo **TPar**. El tipo de datos **TPar** x modela a x con cierto comportamiento dinámico adicional. Capturamos el comportamiento de los operadores básicos de paralelismo de Haskell, e introducimos los operadores (<$>) y (<*>), para poder operar sobre los elementos que son representados. De esta manera podemos reemplazar los operadores de paralelismo, por los presentados en el fragmento de Código 3.1, de forma casi directa.

Utilizando los combinadores presentados en el Código 3.1, reescribimos el ejemplo de una implementación paralela de la suma (parsuma, vista en el capítulo anterior) como se muestra en el Cuadro 3.2, donde se comparan ambas versiones. A izquierda encontramos la versión utilizando los combinadores de Haskell, y a derecha utilizando los combinadores de *Klytius*.

Utilizando las construcciones generadas por los operadores de *Klytius*, se puede generar un gráfico de la computación, representando el comportamiento dinámico. El gráfico se genera a partir de la ejecución de la computación. Por ejemplo, la Figura 3.3 es el resultado de la ejecución de parsuma x y, donde tanto x como y toman valores enteros.

Las computaciones son representadas en forma de un árbol, los cuales pueden comenzar del lado izquierdo y crecer hacia la derecha, o comenzar desde arriba y crecer hacia abajo. El constructor **par** a b es representado como una *elipse*, de donde salen 2 lineas, la linea segmentada indica la creación del *spark* destinado a evaluar la expresión a, y la linea continua indica la la evaluación de la expresión b. El constructor **pseq** a b es representado como

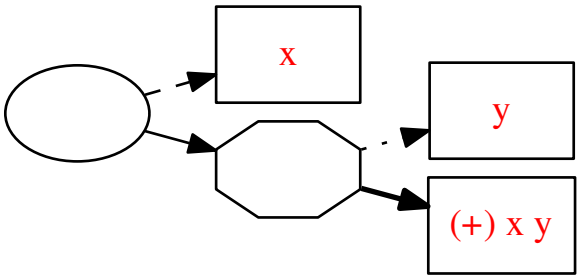


Figura 3.3: Gráfico del comportamiento dinámico de parsuma x y

un *octágono*, donde se indica con una linea de puntos la evaluación de la expresión a, y con una linea más gruesa la evaluación de la expresión b. Los valores son representados como cajas. Como podemos ver en al Figura 3.3, el gráfico comienza con una elipse, mostrando la presencia del operador **par**, el cual crea un *spark* destinado a la evaluación de la expresión x y continua con la evaluación de un octágono, es decir, un **pseq**, que evalúa primero la expresión y y luego evalúa el resultado (+) x y. Podemos observar entonces, que el grafo resultante se encuentra ligado al algoritmo implementado, y de esta manera se puede observar directamente el comportamiento dinámico del programa.

Utilizando esta representación gráfica, el valor de toda la expresión se encuentra descripto por el último valor resultante de seguir siempre las líneas continuas.

3.2. Profundidad de la evaluación

En el Capítulo 2, mostramos que paralelizar computaciones de forma naif produce que los *sparks* generados puedan no llegar a realizar el trabajo que uno espera, terminando con un programa al cual se le agregan nuevas directivas y por ende es más complicado, sin obtener ganancias en velocidad. Pero luego de un cierto análisis de las computaciones que se paralelizan, alcanzamos un punto donde el trabajo se realiza de manera efectiva, optimizando la ejecución del programa. Ello se logra al encontrar la profundidad de evaluación correcta de las expresiones, la cual no resulta ser extremadamente difícil en un pequeño ejemplo, pero a medida que se van haciendo más complejas las construcciones, es cada vez más difícil identificar correctamente la profundidad buscada.

Utilizamos a *Klytius* para observar directamente el comportamiento dinámico de los programas, y esclarecer la profundidad de la evaluación. Para esto,

```
parmap :: (a -> b) -> [a]
    -> [b]
parmap f [] = []
parmap f (x:xs) =
    par x'
    (x' : (parmap f xs))
    where
        x' = f x
```

(a) Clásico parmap.

```
parmap :: (Show a) => (a -> b) -> [a]
    -> TPar [b]
parmap f [] = mkVar []
parmap f (x:xs) =
    par x'
    ((:) <$> x' <*> (parmap f xs))
    where
        x' = mkVars $ f x
```

(b) parmap utilizando a *Klytius*.

Cuadro 3.4: Comparación entre el código original y la incorporación de la herramienta.

mostramos cómo el programa debe ser modificado, utilizando los programas presentados en el capítulo 2.

Modificaremos la función parmap, para poder generar elementos dentro de **TPar**, simplemente reemplazando el operador **par** de la librería Control.Parallel por el operador **par** de *Klytius*, las aplicación por los operadores (<$>) y (<*>) según corresponda, y agregando también **mkVar** cuando sea necesario. Podemos observar la comparación entre ambos códigos en el Cuadro 3.4.

A diferencia de la versión normal, ahora es posible generar el grafo que representa la evaluación de la computación, mediante la función graficar.

```
graficar :: Bool -> TPar a -> String -> IO FilePath
withCanv :: Bool -> TPar a -> IO ()
```

La función graficar, toma como argumento un elemento de tipo **Bool**, que indica si se quiere observar o no los identificadores únicos de los nodos. En el caso que reciba como argumento el valor **True**, los identificadores se mostraran en conjunto con las etiquetas de los nodos. Si un nodo ya posee una etiqueta, se separan mediante el símbolo @, y en el caso que no posea etiqueta simplemente se mostrará el identificador. Estos identificadores sirven como referencia sobre los nodos, y pueden ser útiles cuando se utiliza el intérprete *GHCi*, el cual permite interactuar con el código. La función withCanv, nos permite interactuar dentro del intérprete *GHCi*, genera el gráfico y lo muestra sin tener que cerrar el intérprete.

La función graficar nos permite generar un gráfico, como el presentado en la Figura 3.5. Éste gráfico muestra explícitamente las operaciones que fueron realizadas, así como también muestra una identificación, la cual se observa dentro de los nodos a la derecha del símbolo @ en el caso que tengan una etiqueta con mas información (como ser, Just 24@2) o simplemente el identificador (como ser, el primer nodo con identificador 1). Estos identificadores fueron asignados a los nodos durante la ejecución de la herramienta. En dicha figura se observa directamente cuales son las expresiones que los diferentes *sparks* evalúan.

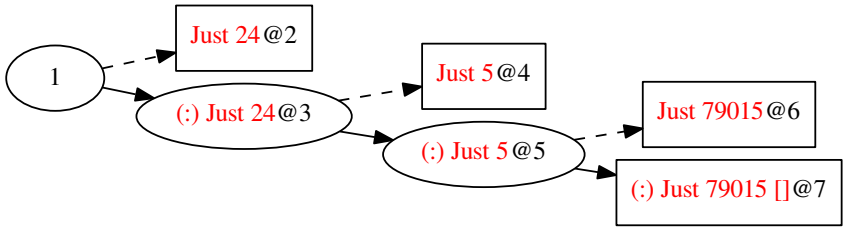


Figura 3.5: graficar **True** (parmap (maybemap (+1)) [**Just** 23, **Just** 4, **Just** 79014])

Repasemos cómo es la evaluación de la expresión

```
parmap (maybemap (+1)) [Just 23, Just 4, Just 79014]},
```

donde parmap aplica una función a una lista de elementos, paralelizando la evaluación de la aplicación de la función sobre cada uno de los elementos. Al comenzar a recorrer la lista, se crea un *spark* dedicado a la evaluación de maybemap (+1) (**Just** 23), y luego continua con el resto de los elementos, generando los *sparks* correspondientes. En la Figura 3.5, podemos observar como se genera un *spark* dedicado a la evaluación de los diferentes elementos, pero los *sparks* no realizan todo el trabajo. Dada la evaluación a forma normal débil a la cabeza impuesta por el operador **par** sobre los *sparks* que son creados, la expresión maybemap (+1) (**Just** 23) será evaluada de la siguiente forma, se aplicará la función maybemap, obteniendo la expresión **Just** ((+1) 23), y terminará la evaluación dado que ya se habrá alcanzado la forma normal débil a la cabeza para elementos de tipo **Maybe Int**. Gráficamente, este comportamiento lo podemos observar gracias a las etiquetas de los nodos. Al comenzar la evaluación de la expresión que representa el nodo 1, se genera un *spark* para la evaluación de la expresión presentada en el nodo con identificación 2, donde éste evalúa la expresión hasta encontrar el constructor Just dejando sin evaluar el resto de la expresión, en este caso (+1) 23.

El mismo fenómeno sucede con los nodos 4 y 6. De este modo, la operación que buscamos que se realice en paralelo, (+1) sobre cada uno de los elementos de la lista, no ha sido realizada como esperábamos, es decir, no se paralelizó la aplicación completa a los elementos de la lista, sino que al momento de observar la lista resultante las sumas faltantes ((+1) 23, (+1) 4, (+1) 79014) se hacen de forma secuencial.

Para garantizar que el trabajo se realice dentro de los *sparks* que son generados, utilizamos la función posMaybe antes presentada modificada para usar el EDSL. La función ahora retorna un **TPar** () e inserta los valores a representar utilizando el operador **mkVar** (ver Código 3.6). Debido a las modificaciones realizadas, el tipo resultante de la función posMaybe es b → **TPar**(), pero los va-

```
posMaybe :: b → TPar ()
posMaybe Nothing  = mkVar ()
posMaybe (Just x) = pseq x (mkVar ())

maybeapp :: (Maybe a → Maybe b) → [Maybe a] → TPar [Maybe b]
maybeapp f [] = mkVars []
maybeapp f (x:xs) =
    par (posMaybe =<< x')
        ((:) <$> x' <*> (maybeapp f xs))
    where
        x' = mkVars (f x)
```

Código 3.6: Versión paralela correcta de maybeapp utilizando a *Klytius*

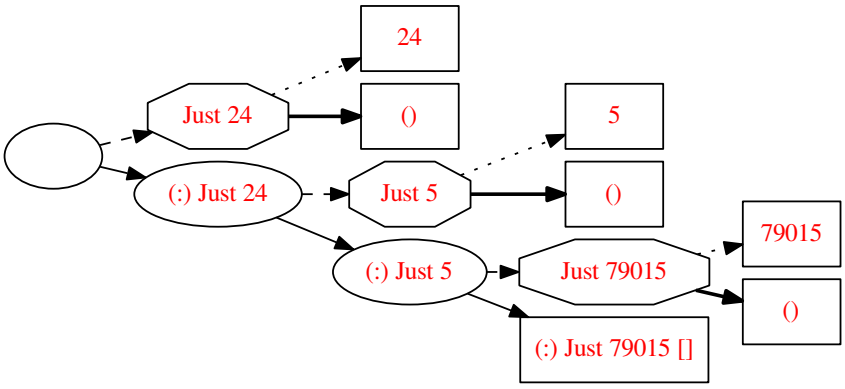


Figura 3.7: Gráfico resultado de: graficar **False** (maybeapp [**Just** 23, **Just** 4, **Just** 79014])

lores a manipular son de tipo **TPar** b, con lo que podemos utilizar la aplicación monádica (=<<) que permite acumular el comportamiento dinámico generado por la función posMaybe. De esta manera generamos el grafo mostrado en la Figura 3.7. Podemos observar la aparición de nuevos rombos, representando la mayor utilización de **pseq**. Particularmente, dichos rombos se encuentran en lugar de los valores en el gráfico anterior (Figura 3.5). Ahora podemos observar como la función posMaybe, ha agregado el comportamiento dinámico necesario, forzando la evaluación a forma normal débil a la cabeza de los diferentes valores dentro del tipo **Maybe**.

Mediante los gráficos generados pudimos observar directamente la necesidad de una función posMaybe para forzar la evaluación completa, y al hacerlo también se pudo analizar el trabajo que realizan los *sparks*.

3.3. Estrategias dentro de *Klytius*

Las estrategias buscan separar el comportamiento dinámico lo más posible de la implementación del algoritmo. Esta separación se logra definiendo el comportamiento dinámico de los diferentes tipos de datos a utilizar, y su nivel de evaluación, permitiendo mezclar el comportamiento dinámico y los niveles de evaluación para alcanzar el paralelismo buscado.

Utilizaremos a *Klytius* para observar el comportamiento de las estrategias tal cual se encuentran definidas en la versión actual de GHC dentro del paquete *Control.Parallel.Strategies* (*Strategies* 2015).

Las estrategias son funciones que describen el comportamiento dinámico sobre un elemento. Éste comportamiento dinámico adicional define cuales expresiones serán evaluadas en paralelo, y cuan profunda es la evaluación de los elementos. Al momento de realizar un análisis sobre las estrategias uno espera poder observar la estructura generada al aplicar una estrategia particular sobre un elemento. Utilizando el EDSL de *Klytius*, las estrategias son simplemente funciones que dado un elemento pueden construir la estructura que describe su comportamiento dinámico, por lo que son redefinidas como se muestra a continuación.

```
type Strategy a = a → TPar a
```

Al redefinir las estrategias, es necesario realizar ciertas modificaciones sobre las funciones básicas de las estrategias, aunque estos cambios son sencillos e intuitivos. A modo de ejemplo, mostramos como modificar la estrategia parList.

```
parList :: Strategy a → Strategy [a]
parList st []     = mkVar []
parList st (x:xs) =
    par (st =<< x')
        ((:) <$> x' <*> (parList st xs))
    where
        x' = mkVar x
```

Donde se insertan los elementos utilizando el operador **mkVar**, se reemplazan las aplicaciones de funciones por los operadores (<$>), (<*>) y la aplicación monádica (=<<) que nos permite acumular el comportamiento dinámico.

Una vez definidas las estrategias en base al tipo **TPar**, podemos modificar la implementación de parmap, de la misma forma que fue definida en el Capítulo 2.

```
parmap :: (NFData b) ⇒ (a → b) → [a] → TPar [b]
parmap f xs = map f xs `using` parList rdeepseq
```

Definiendo las estrategias utilizando el tipo **TPar**, obtenemos la habilidad de graficar los resultados, sin tener que modificar el código ya programado en la mayoría de los casos. Es decir, en código donde el comportamiento dinámico estaba dado por una estrategia, sólo es necesario modificar levemente la estrategia para poder utilizar nuestra herramienta.

3.4. Caso de estudio: Map Reduce

MapReduce es un patrón de diseño para el cómputo de programas sobre grandes cantidades de información. El usuario específica una función denominada *map*, que procesa un par clave-valor para generar un conjunto de pares clave-valor intermedios, los cuales son ordenados y recolectados en base a la clave intermedia para ser utilizados por una función, denominada *reduce*, que sintetiza todos los valores de una misma clave intermedia (Dean y Ghemawat 2008). El modelo, además, tiene la característica que es muy fácil de paralelizar, principalmente debido a la naturaleza de la aplicación de la función *map*. Debido a que la aplicación de la función es aplicada a cada elemento, es decir, la aplicación es independiente del resto de los elementos, permitiendo distribuir todas las expresiones correspondientes a las aplicaciones sobre los procesadores disponibles. Luego de terminar las computaciones, los valores son recolectados para ser sintetizados por la función *reduce*.

Siguiendo el ejemplo de libro *Real World Haskell* (O'Sullivan, Goerzen y Stewart 2008) podemos obtener una función basada en el funcionamiento general de *Map Reduce*, que nos permite operar de manera **similar** a la antes planteada. Son necesarias dos funciones: primero la función *map* de tipo $a \to b$, que será aplicada a una lista de valores de tipo a; en segundo lugar, necesitamos una función *reduce* que sintetice los valores generados por la función *map* de tipo $[b] \to c$. A continuación se muestra la versión secuencial del algoritmo.

```
simpleMapReduce ::
            (a → b)
         → ([b] → c)
         → [a]
         → c
simpleMapReduce mapper reducer xs = reducer $ map mapper xs
```

Dadas las funciones mapper y reducer, y una lista xs, simplemente aplicamos la función mapper a cada uno de los elementos de xs, y luego sintetizamos todos los resultados utilizando reducer.

En el algoritmo presentado, es posible explotar el paralelismo de la evaluación de la lista utilizada, como ser, utilizando la estrategia parList, u otra estrategia de evaluación sobre listas de elementos, por lo que se toma como argumento la estrategia que desea aplicar el usuario. Además, es posible que el elemento generado por la función reducer, pueda ser evaluado de forma paralela, dando la necesidad de agregar un último argumento para la estrategia de evaluación de elementos de tipo c. De esta manera, podemos aplicar estrategias para dar una definición del algoritmo que explote el paralelismo del programa.

```
mapReduce ::
        Strategy [b]     -- ^ Estrategia evaluación de lista de b
     -> (a -> b)         -- ^ Map
     -> Strategy c       -- ^ Estrategia evaluación de c
     -> ([b] -> c)       -- ^ Reduce
     -> [a]
     -> TPar c
mapReduce strb mp strc red xs = pseq mapRes redRes
    where
        mapRes = map mp xs `using` strb
        redRes = (red <$> mapRes) `rusing` strc
```

A medida que se acumula comportamiento dinámico, el programador ya no tiene acceso a elementos x de tipo a, sino que manipula elementos con cierto comportamiento dinámico adicional, x' de tipo **TPar** a. Debido a esto, se introduce una nueva aplicación de estrategias, que nos permite operar de la misma forma, acumulando nuevo comportamiento dinámico. A continuación se muestra la interfaz de la función.

```
rusing :: TPar a -> Strategy a -> TPar a
```

Utilizando esta estructura propuesta por mapReduce, podemos dar un algoritmo, por ejemplo, para contar la aparición de las palabras en un texto dado, como se muestra en el fragmento de Código 3.8. Donde encontramos las funciones:

- oneWord, que dada una palabra le asigna el valor 1,
- summ, que suma todos los valores asignados a una palabra específica,
- normalize, recorre la lista utilizando summ para acumular los resultados,
- rightTup, es una estrategia sobre pares de tipo (**String**, **Int**), donde solamente evalúa a forma normal la componente derecha del par, y asigna las etiquetas correspondientes, para luego en el gráfico saber a que palabra hace referencia un nodo.

El resultado de la computación resultado, es una lista de pares que indican cuantas apariciones en el texto ejemplo tiene la primer componente.

Podemos observar el gráfico resultante de la computación de contar las palabras dentro del pequeño ejemplo, en la Figura 3.9. La aplicación de la función oneWord es aplicada a cada uno de los elementos de la lista ["contar","pegar","contar","saltar"], y luego, la lista resultante es sintetizada por la función normalize, la cual suma la cantidad de apariciones de cada palabra.

Gracias al gráfico mostrado en la Figura 3.9 podemos observar, que se genera un *spark* para la evaluación de oneWord sobre cada una de las palabras, sumando que conocemos que la aplicación de la función no requiere mucho trabajo, resulta en un paralelismo muy granular. Es decir, la evaluación de la expresión resultado, genera *sparks* con poco trabajo a realizar. Esto produce que el costo de evaluar la expresión en paralelo sea mayor que la evaluación de la expresión sí. Una forma sencilla de evitar este problema, es particionar la lista en pequeñas lista, llamadas *chunks*, y evaluar cada una de estas

```
oneWord :: String → (String, Int)
oneWord s = (s,1)

summ :: [(String,Int)] → String → Int → [(String, Int)]
summ [] s i = [(s,i)]
summ ((s,i):xs) ns ni    | (s == ns) = (s,i+ni) : xs
                         | s < ns = (s,i) : (summ xs ns ni)
                         | otherwise = (ns, ni) : ((s,i):xs)

normalize :: [(String,Int)] → [(String,Int)]
normalize xs = foldl (\ a (w,i) → summ a w i) [] xs

rightTup :: Strategy (String,Int)
rightTup (a,b) = mkVar (,) <*> (vlbl a (r0 a)) <*> (vlbl (show b)
    (rwhnf b))

parContPal :: [String] → TPar [(String, Int)]
parContPal xs = mapReduce (parList rightTup) oneWord
    (parList rightTup) normalize xs

parContPalC :: [String] → TPar [(String, Int)]
parContPalC xs = mapReduce (parListChunk 4 rightTup) oneWord
    (parListChunk 4 rightTup) normalize xs

ejemplo :: [String]
ejemplo = "contar saltar contar pegar"

ejemplo2 :: String
ejemplo2 = "Al fin llego el día en que Martín va a dejar de
    molestar a los profesores."

resultado :: [(String, Int)]
resultado = parContPal (splitOn " " ejemplo)

resultado2 :: [(String, Int)]
resultado2 = parContPal (splitOn " " ejemplo2)
```

Código 3.8: Contar palabras utilizando mapReduce.

listas por separado, es decir, acumular trabajo suficiente y luego distribuirlo por los diferentes núcleos. Utilizaremos a resultado2 como ejemplo, debido que parContPalC divide la lista de palabras en listas de 4 palabras. El resultado se puede apreciar en la Figura 3.10, donde podemos observar que se paralelizan las aplicaciones de a 4 elementos.

Resumen

Durante el capítulo desarrollamos como utilizar la herramienta para observar la estructura dinámica de las computaciones basadas en los operadores básicos **par** y **pseq**. Se logró identificar si las expresiones realizaban el trabajo

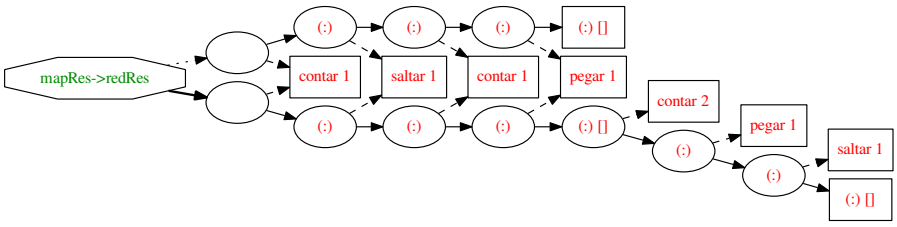


Figura 3.9: Gráfico representando la estructura dinámica de resultado

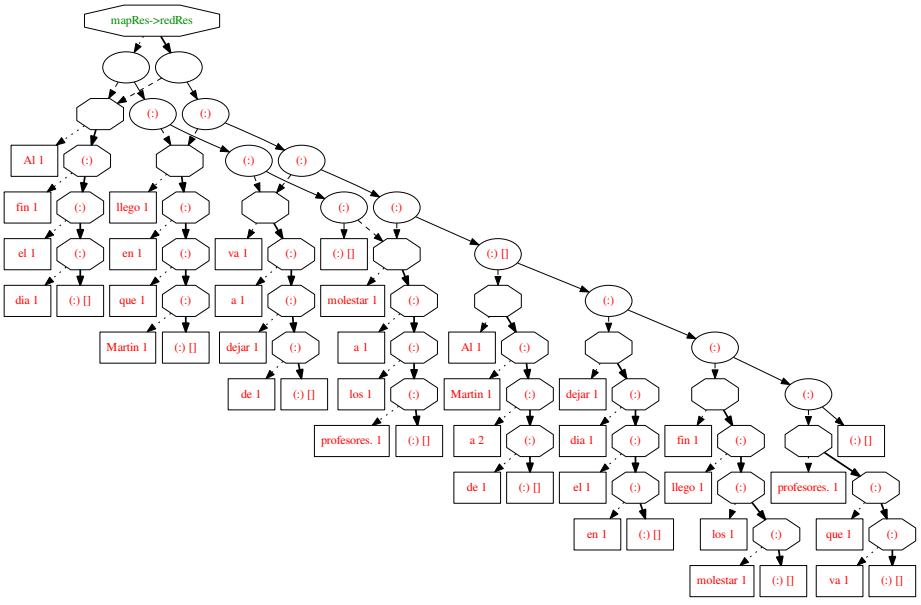


Figura 3.10: Gráfico representando la estructura dinámica de resultado2

esperado o si la evaluación perezosa impedía que se terminara de evaluar toda la expresión, y se detectó la generación de *sparks* demasiado granulares. Si bien los programas sufren leves modificaciones, nos entregan un mapa de la información estática sobre el comportamiento dinámico de la expresión, lo que nos permite entender mejor los programas a paralelizar.

Capítulo 4

Implementación

Este capítulo presenta la implementación de *Klytius* y las técnicas utilizadas. Primero introducimos los fundamentos de los lenguajes de dominio específico, luego plantearemos el problema de observar computaciones compartidas, y por último explicaremos cómo se generan los gráficos de la estructura dinámica de los programas vistos en los capítulos anteriores.

4.1. Lenguajes de Dominios Específicos

Una solución prometedora para independizar un sistema de la forma en que se va a ejecutar el código consiste en la creación de lenguajes diseñados exclusivamente para facilitar la implementación de la lógica de negocios en un dominio en particular. Estos lenguajes son conocidos como lenguajes de dominio específico (DSL). Un DSL permite trabajar a un nivel de abstracción ideal, donde los conceptos con los que se trabaja son los propios del dominio de aplicación (Bentley 1986). Por ejemplo, un ingeniero puede escribir un programa en MATLAB u otras herramientas, utilizando primitivas del lenguaje específicas para el desarrollo de puentes sin preocuparse por el manejo de estructuras internas de un lenguaje de programación. Otros ejemplos muy difundidos de DSLs son LaTeX para la edición de texto científico, y SQL para el manejo de base de datos.

Los DSL tienen la desventaja de que necesitan del desarrollo de herramientas especiales. Mientras que para los lenguajes de propósito general están disponibles, editores, compiladores (lo que incluye parsers, optimizadores y generadores de código), bibliotecas, generadores de documentación, depuradores, etc., los DSL no disponen de herramienta alguna. Es tarea del desarrollador proveer las herramientas necesarias para su DSL. Por esto, han adquirido popularidad los DSLs embebidos (EDSL) en un lenguaje de propósito general (Hudak 1996). Un EDSL es un lenguaje definido dentro de otro lenguaje, lo que permite utilizar toda la maquinaria ya implementada para el lenguaje anfitrión, disminuyendo enormemente la tarea al momento de implementar el

lenguaje.

Al construir un EDSL, diseñamos un lenguaje con ciertas primitivas, un cierto *idioma* que nos permite manipular elementos y poder construir un resultado final en un dominio específico. Dentro de esta construcción podemos optar por dos caminos (Gill 2014):

- **Embebido Superficial**, a medida que el lenguaje va identificando instrucciones, manipula los valores sobre los cuales se aplica, retornando un nuevo valor. Es decir, no se genera estructura intermedia alguna, sino que simplemente los operadores son funciones que van operando sobre los valores directamente. El resultado de los lenguajes con embebido superficial es un valor, el resultado de la ejecución de las instrucciones dadas.

- **Embebido Profundo**, utilizando las instrucciones se genera un nuevo tipo abstracto de datos, permitiendo al usuario construir un *árbol abstracto* de la computación. De esta manera, se puede recorrer la estructura, dando la posibilidad de optimizar o sintetizar la computación, y luego ser consumida por una función que evalúa y otorga la semántica deseada a los constructores. Se logra separar el idioma del EDSL de las operaciones mecánicas que se necesitan para evaluar correctamente el resultado, permitiendo, por ejemplo, exportar las computaciones en el caso que se quieran ejecutar fuera del entorno de Haskell (como ser, en la GPU) o dar varios comportamientos diferentes a las operaciones del lenguaje. Permite exponer toda la estructura y composición del programa.

Para ejemplificar los dos enfoques, supongamos que nuestro idioma está compuesto por operaciones aritméticas sobre enteros, más precisamente, nuestro lenguaje está compuesto por la suma, la resta y la multiplicación. En el caso del embebido superficial, bastaría con definir un tipo `EnteroS` y las siguientes funciones:

```
newtype EnteroS = E Int

suma :: EnteroS → EnteroS → EnteroS
suma (E x) (E y) = E (x + y)

resta :: EnteroS → EnteroS → EnteroS
resta (E x) (E y) = E (x − y)

mult :: EnteroS → EnteroS → EnteroS
mult (E x) (E y) = E (x ∗ y)

runS :: EnteroS → Int
runS (E x) = x
```

Donde, por ejemplo, `suma (E 4) (E 5) = E 9`.

En el caso de embebido profundo, en cambio, deberíamos definir un nuevo tipo abstracto de datos:

```
data EnteroD where
    Lit   :: Int → EnteroD
    Suma  :: EnteroD → EnteroD → EnteroD
    Resta :: EnteroD → EnteroD → EnteroD
    Mult  :: EnteroD → EnteroD → EnteroD

sumaD :: EnteroD → EnteroD → EnteroD
sumaD = Suma
restaD :: EnteroD → EnteroD → EnteroD
restaD = Resta
multD :: EnteroD → EnteroD → EnteroD
multD = Mult
```

En el ejemplo antes mencionado, `sumaD (Lit 4) (Lit 5) = Suma (Lit 4) (Lit 5)`, permitiendo observar directamente la construcción de la computación. En el caso que se quiera obtener el resultado de la evaluación, es necesario consumir el árbol abstracto, como se muestra en la siguiente función:

```
runD :: EnteroD → Int
runD (Lit n)      = n
runD (Suma x y)   = let
    x' = runD x
    y' = runD y
    in (x' + y')
runD (Resta x y) = let
    x' = runD x
    y' = runD y
    in (x' - y')
runD (Mult x y)   = let
    x' = runD x
    y' = runD y
    in (x' * y')
```

Utilizando la función `runD`, se obtiene el resultado de la aplicación de los operadores que indican los diferentes constructores. Utilizando dichas definiciones de `runS` y `runD`, obtenemos la igualdad `runS s = runD d`, para cualquier valor `s` de tipo `EnteroS` y `d` de tipo `EnteroD`.

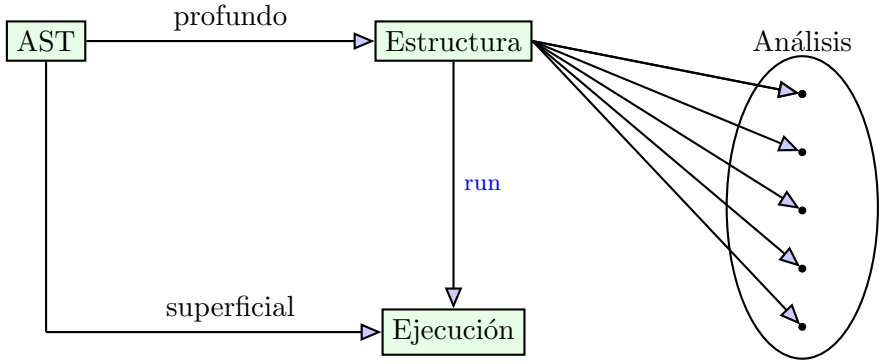
Pero en el caso de tener un EDSL profundo, no solo se puede obtener el valor resultado de evaluar la expresión, sino que, por ejemplo, se puede contar la cantidad de sumas en una expresión fácilmente sin modificar las funciones anteriores.

```
contsuma :: EnteroD → Int
contsuma (Lit _)      = 0
contsuma (Suma l r)   = (contsuma l) + (contsuma r) + 1
contsuma (Resta l r)  = (contsuma l) + (contsuma r)
contsuma (Mult l r)   = (contsuma l) + (contsuma r)
```

De esta manera el embebido profundo combinado con una función específica (como en este caso `runD`), puede ser equivalente a un embebido superficial.

Además, permite utilizar la construcción de alguna otra manera, como ser, para extraer más información (contsuma).

En la figura que se muestra a continuación, se ilustra la diferencia entre los EDSL profundos y superficiales. En el caso de un EDSL superficial, obtenemos un valor, el valor resultante de la aplicación de los diferentes operadores del lenguaje utilizados por el usuario, mientras que en el caso de un EDSL embebido, obtenemos un árbol abstracto de las operaciones que fueron aplicadas, de esta manera es posible realizar análisis, estudiar la estructura obtenida, o ejecutar el programa *run* y obtener un resultado equivalente al resultado de un EDSL superficial.



Los EDSL profundos nos permiten construir el árbol abstracto de las expresiones del lenguaje que describe. El árbol abstracto contiene la información de cómo un valor dentro del lenguaje fue construido, por lo que ya no se manipulan valores, sino árboles abstractos. Ésto trae consigo la imposibilidad de utilizar **directamente** estructuras de control de flujo, como *if-then-else*, sin consumir la estructura. Además al tratar con estructuras complejas, y no con valores manipulables directamente, se reducen las optimizaciones que el compilador del lenguaje anfitrión puede realizar. La construcción de ciertos EDSL puede generar ciclos infinitos. Esto quita la posibilidad de recorrer la estructura generada, y genera la necesidad de un mecanismo adicional para la detección de dichos ciclos.

EDSL profundo para Paralelismo en Haskell

En Haskell hay una gran diversidad de abstracciones y DSLs para modelar paralelismo. Entre los más populares están, la mónada Eval (Marlow y col. 2010a), la mónada Par (Marlow, Newton y Peyton Jones 2011), Data Parallel Haskell (Chakravarty, Leshchinskiy y col. 2007), la librería Repa (Keller y col. 2010), el lenguaje Accelerate (Chakravarty, Keller, Lee y col. 2011), y el lenguaje Obsidian (Svensson, Sheeran y Claessen 2011). Una comparación entre varios enfoques puede verse en (Trinder, Loidl y Pointon 2002).

Dado que nuestro interés está centrado en el estudio del paralelismo esencial dentro de Haskell, desarrollamos un EDSL profundo en base a los combinadores básicos, **par** y **pseq**. Tomando estos combinadores como constructores

```
data TPar s where
    Par :: TPar a → TPar s → TPar s
    Seq :: TPar a → TPar s → TPar s
    Val :: s → TPar s
```

Código 4.1: EDSL profundo para modelar paralelismo.

```
parsuma :: Int → Int → TPar Int
parsuma l r =
    par l' (pseq r' (l' + r'))
    where
      l' = mkVar l
      r' = mkVar r
```

Código 4.2

del lenguaje, definimos el tipo de datos como se muestra en el fragmento de Código 4.1

Este tipo de datos nos permite modelar el comportamiento dinámico de las computaciones generadas al momento de ejecutar el programa. Es, además, un tipo de datos con estructura de árbol, donde las hojas son los valores introducidos por el constructor Val y los nodos (que no son hojas) representan el combinador **par** (Par), y el combinador **pseq** (Seq).

Cuando en Haskell utilizamos los operadores básicos **par** y **pseq**, el resultado es un comportamiento dinámico que no es observado directamente, sino que, como ya ha sido mostrado, **par** x y = y y **pseq** x y = y. Utilizando el EDSL planteado en *Klytius* se puede observar la construcción del comportamiento dinámico de la expresión, ya que se conserva toda la información de los operadores básicos utilizados, es decir, **par** x' y' = Par x' y' y **pseq** x' y' = Seq x' y', donde tanto x', como y', pueden tener a su vez comportamiento dinámico adicional.

Por ejemplo, para la suma paralela (ver Código 4.2), obtenemos el siguiente árbol abstracto:

```
parsuma 4 7 = Par (Val 4) (Seq (Val 7) (Val ((+) 4 7) ))
```

De esta manera es posible construir un AST de las construcciones dinámicas de nuestro programa, que nos permite **observar todo el comportamiento dinámico de la computación**. El EDSL de *Klytius* provee primitivas de alto nivel para construir fácilmente dicho AST, y luego, mostrarlo gráficamente, permitiéndole al programador realizar un análisis más exhaustivo del programa.

4.2. Observando Computaciones Compartidas

Una parte vital del desarrollo de la herramienta es la detección de computaciones compartidas. Éste es un problema muy estudiado dentro de los EDSL y en optimizaciones de compiladores, dado que las computaciones compartidas permiten evaluar una única vez una expresión y luego compartir el resultado, minimizando el trabajo a realizar. Por ejemplo, en una expresión como `doble x = let y = x + 1 in y + y`, la expresión `y` es compartida para la suma (`y+y`), y su evaluación puede realizarse una sola vez.

Al desarrollar un programa paralelo en Haskell, el programador, genera *sparks* para la evaluación de expresiones que se podrían utilizar luego. Es decir, cuando tenemos una expresión como:

```
par x m
```

la computación `x` es luego utilizada en `m`. Esto mismo, podemos verlo directamente en una expresión como `par y y`. Nuestro objetivo es representar las computaciones de manera gráfica, por lo que deberíamos identificar en este caso a `y` como una referencia a la misma computación, así como identificar todas las referencias de `x` que ocurran dentro de `m`.

En un EDSL profundo las computaciones son representadas como un árbol abstracto, por lo que las computaciones recursivas o que poseen referencias mutuas pueden dar lugar a un árbol *infinito*[1].

Andy Gill presenta un mecanismo de detección de computaciones compartidas, utilizando funciones dentro de la mónada IO de Haskell y funciones de tipo (Chakravarty, Keller y Jones 2005). Esta solución nos permite construir el árbol abstracto utilizando primitivas de nuestro EDSL y luego generar el grafo que representa nuestra computación, y por lo tanto nos permite observar las computaciones compartidas (Gill 2009).

En dicha solución introduce la función `reifyGraph` (de tipo como se muestra a continuación) que, dado un elemento `x` de tipo `t` construye un grafo de tipo `Graph`, el cual representa al elemento `x` detectando las computaciones compartidas. La clase `MuRef` provee las operaciones necesarias para poder observar la estructura de un tipo dado, y para construir grafos con las computaciones compartidas. Para definir una instancia es necesario dar una función de tipo `DeRef`, que actúa como un sinónimo, entre el tipo a instanciar y su representación como un grafo, y una función `mapDeRef` que permite recorrer el árbol y transformar sus elementos en el nodo que los representa dentro del grafo final.

```
reifyGraph :: (MuRef t) => t -> IO (Graph (DeRef t))
```

En Haskell se define de la siguiente manera:

```
class MuRef a where
    type DeRef a :: * -> *
    mapDeRef     :: (Applicative f) =>
```

[1] Podemos hablar de infinito gracias a la evaluación lazy.

```
    (a → f u)
→ a
→ f (DeRef a u)
```

El primer argumento de mapDeRef es una función que es aplicada a los hijos del segundo argumento, el cual es el nodo que se está analizando. El resultado es un nodo que contiene valores únicos de tipo u que fueron generados en base al primer argumento. La función mapDeRef utiliza funtores aplicativos (McBride y Paterson 2007) para proveer los efectos necesarios para la correcta asignación de los valores únicos.

A continuación mostramos como definir la representación de un árbol de tipo **TPar**, mediante el tipo Node. Notar que en el caso de la transformación de Val' (**TPar**) a Val(Node), dado en la definición de mapDeRef, el valor es ignorado, ya que no es necesario conservar el elemento, por lo que se representa y se trata como una caja cerrada.

```
data TPar s where
    Par' :: TPar a → TPar s → TPar s
    Seq' :: TPar a → TPar s → TPar s
    Val' :: s → TPar s

data Node s where
    Val :: Node s
    Par :: s → s → Node s
    Seq :: s → s → Node s
    deriving Show

instance MuRef (TPar s) where
    type DeRef (TPar s) = Node
    mapDeRef f (Par' l r)  = Par  <$> f l <*> f r
    mapDeRef f (Seq' l r)  = Seq  <$> f l <*> f r
    mapDeRef f (Val' _)    = pure Val
```

Utilizando la definición de MuRef (**TPar** s), permite detectar las computaciones compartidas una vez obtenido un valor de tipo **TPar** s, permitiendo al usuario construir su programa normalmente (sin obligar el uso de mónadas), y luego aplicar la función reifyGraph, ahora sí dentro de la mónada **IO**.

Por ejemplo en el caso de la suma paralela (Código 4.2), el resultado de aplicar la función reifyGraph es:

```
reifyGraph (parsuma 4 7) =
    let
        [(1,Par 2 3),(3,Seq 4 5),(5,Val ("(+) 4 7")),(4,Val ("7"))
            ,(2,Val ("4"))]
    in 1
```

El resultado es una lista de pares compuestos por el valor único que representa al nodo, y el constructor correspondiente, y como resultado el valor único de la raíz del árbol. A modo descriptivo se introducen etiquetas que no están representadas en el constructor dado anteriormente.

Existen otras soluciones al problema de detectar computaciones compartidas. A continuación detallamos algunas de estas y explicamos por qué no fueron utilizadas en este trabajo.

- Extensión a los lenguajes funcionales que permite observar (y por ende recorrer) este tipos de estructuras (Claessen y Sands 1999). La solución consiste en poder definir un tipo de datos Ref, con tres operaciones. Permitir la creación de referencias a partir de un elemento, poder obtener el elemento a través de su referencia, y poder comparar referencias. Es una solución no conservativa, la cual implica tener que modificar el lenguaje anfitrión, en este caso Haskell.

- Etiquetado explícito, etiquetar a los nodos del árbol con un elemento único. En este caso, el usuario es el que provee las etiquetas manualmente.

- Las mónadas pueden ser utilizadas para generar las etiquetas de manera implícita, o bien, utilizar funtores aplicativos. El problema de esta solución es que impacta directamente en el tipo de las primitivas básicas de la herramienta, obligando al usuario a utilizar un modelo monádico, obscureciendo el código. Es posible utilizar llamadas inseguras a la mónada **IO**, y utilizar, lo que en programación imperativa se conoce como un contador, rompiendo las garantías que Haskell provee.

La solución elegida en el presente trabajo no exhibe los problemas descriptos. Permite al usuario de *Klytius* el desarrollo libre de efectos monádicos generados por la detección de computaciones compartidas, sin tener que modificar el EDSL presentado en este trabajo, ni Haskell.

4.3. Graficando Computaciones

Utilizando la función de reifyGraph, logramos generar un grafo que representa la computación propuesta por el *AST*, detectando las computaciones compartidas como se explicó en la sección anterior. Para generar la figura utilizamos la librería para la herramienta *Dot* dentro de la suite *GraphViz* (Gansner, Koutsofios y North 2005; *Graph Visualization Software* 2015; Gansner y North 2000).

Dot es una herramienta muy versátil que permite dibujar grafos dirigidos, y se encarga de distribuir los nodos y las aristas sobre un mapa cartesiano. La herramienta toma como entrada un archivo de texto descriptivo del grafo y genera el archivo de salida con la imagen del grafo. Para generar dichos archivos de texto que representan los grafos a graficar, utilizamos la librería de Haskell, *GraphViz* (Matthew Sackman 2015).

La librería se basa en un tipo de datos GraphvizParams n nl el cl l, que contiene toda la configuración gráfica del grafo, donde se indica que el grafo tiene

```
graficar :: Bool → TPar a → String → IO FilePath
graficar b g s = do
    g' ← reifyGraph g
    runGraphvizCommand Dot (graphToDot (grafiParams b False 0.25)
        (graph x)) Pdf (s++".pdf")
```

Código 4.3: Función graficar

nodos de tipo n, las etiquetas de los nodos son de tipo nl, la etiqueta de las aristas de tipo el, y el resto corresponden a tipos para realizar *clustering* de nodos.

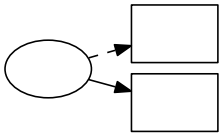
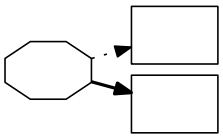
El tipo de datos GraphvizParams n nl el cl l es un *record* compuesto, entre otra información utilizada para realizar *clustering* de nodos, por:

- isDirected :: **Bool**, donde se indica **True** si el grafo es dirigido y **False** sino.
- globalAttributes :: [GlobalAttributes], donde se especifican atributos generales a todo el grafo.
- fmtNode :: (n, l) → Attributes, donde se indican los atributos de los nodos.
- fmtEdge :: (n, n, el) → Attributes, donde se indican los atributos de las aristas.

La librería pone a disposición constructores de parámetros básicos, en conjunto con funciones para generar el grafo Dot, exportar el archivo de texto y utilizando la herramienta externa, generar el archivo con formato *Pdf*.

Definiendo un elemento de tipo GraphvizParams, el cual contiene las propiedades gráficas de la imagen que se obtiene como resultado, se define una función, graficar (ver Código 4.3), que toma como argumento, un valor booleano indicando si se requiere mostrar información adicional, un elemento de tipo **TPar** s, y el nombre del archivo a generar. Una vez obtenidos los argumentos, la función genera un archivo con el gráfico de la computación en el mismo directorio donde se ejecutó el programa.

La función graficar permite obtener las representaciones gráficas de los diferentes programas resultantes de utilizar a *Klytius*, en el Cuadro 4.4 se muestran las representaciones de los constructores utilizados.

Par Val Val	
Seq Val Val	

Cuadro 4.4: Constructores de **TPar** con su representación gráfica asociada.

Etiquetado de Nodos en *Klytius*

El usuario tiene la habilidad de introducir etiquetas mediante la utilización de primitivas dentro de *Klytius*, que se muestran a continuación.

```
pars :: String → TPar a → TPar b → TPar b
seqs :: String → TPar a → TPar b → TPar b
smap :: String → (a → b) → TPar a → TPar b
mkVar' :: String → TPar a
mkVars :: (Show a) ⇒ TPar a
```

Utilizando estos constructores se pueden introducir etiquetas que serán mostradas dentro de los nodos del gráfico resultante. Las etiquetas no manipulan de manera alguna los valores, son simplemente atributos gráficos que permiten al programador identificar a qué expresión corresponde cada nodo del gráfico. Podemos ver como las etiquetas son mostradas en la Figura 4.5

Es recomendable el uso de la etiquetas, ya que permiten interpretar mejor los valores representados. Sin embargo son un arma de doble filo, debido a que se muestra lo que usuario expone, y no cuál es el valor real a representar.

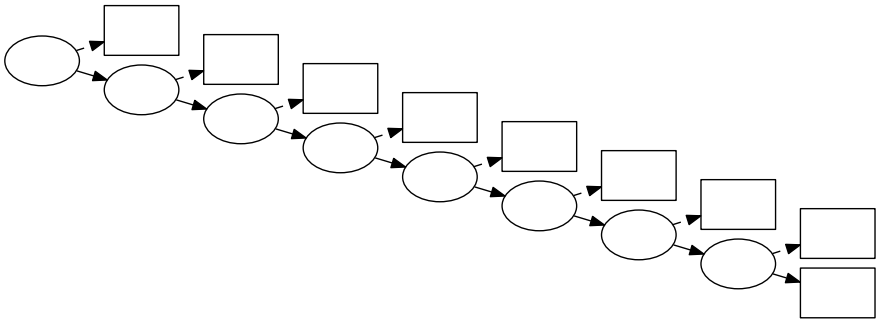
Por ejemplo, tomando un ejemplo ya visto en los capítulos anteriores, podemos paralelizar la aplicación de una función sobre una lista, y vamos a tener una estructura dinámica como resultado. En este caso, las etiquetas pueden ayudar a ver si la forma normal débil a la cabeza es profundidad suficiente para la evaluación de las expresiones paralelizadas. Se puede observar la diferencia en los gráficos obtenidos en la Figura 4.6

Utilizando etiquetas el usuario puede indicar que valores son expresados dentro de las “cajas“. Además el usuario puede agregar etiquetas descriptivas a los diferentes nodos que representan la estructura dinámica de control de

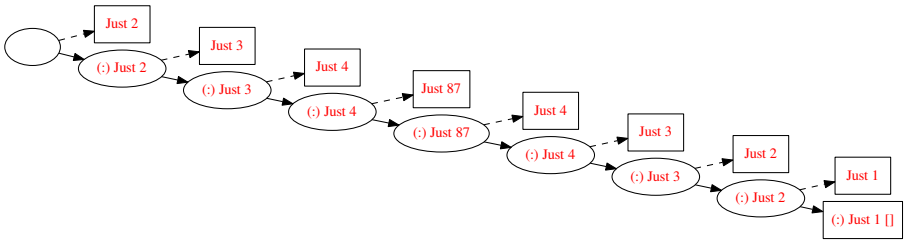
(a) Etiqueta dentro del nodo que representa a **pars** "Etiqueta Par"a b

(b) Etiqueta dentro del nodo que representa a **seqs** "Etiqueta Seq"a b

Figura 4.5: Etiquetas dentro de los nodos



(a) Paralelizar una lista sin etiquetas



(b) Paralelizar una lista mostrando todas las etiquetas

Figura 4.6: Diferencia entre mostrar etiquetas o no mostrarlas

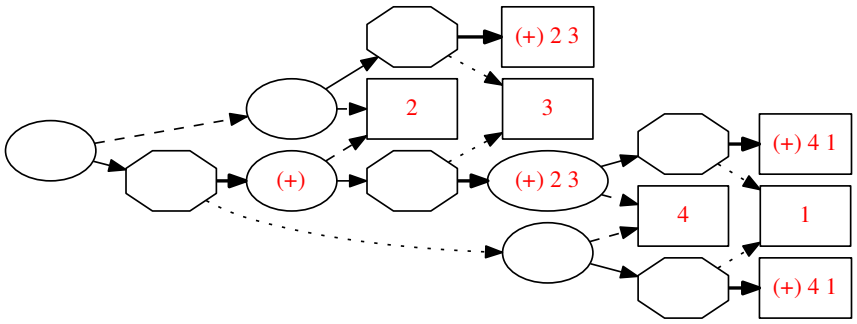
```
suml :: Int → [Int] → TPar Int
suml _ []  = mkVars 0
suml _ [x] = mkVars x
suml n xs  = par lf (pseq rf (lf+rf))
    where
        nn = div n 2
        (lf,rf) =   (suml nn (take nn xs)
                    ,suml nn (drop nn xs))

sl :: Int → [Int] → TPar Int
sl _ []  = mkVars 0
sl _ [x] = mkVars x
sl n xs  = pars "FPart" lf (seqs "SPart" rf (lf+rf))
    where
        nn = div n 2
        (lf,rf) =   (sl nn (take nn xs)
                    ,sl nn (drop nn xs))

ejemplodin :: IO FilePath
ejemplodin = do
    let ej = ([2,3,4,1] :: [Int])
    let ln = length ej
    graficar False (suml ln ej) "suml"
    graficar False (sl ln ej) "sumletiq"
```

Código 4.7: Sumatoria de una lista de enteros de forma paralela.

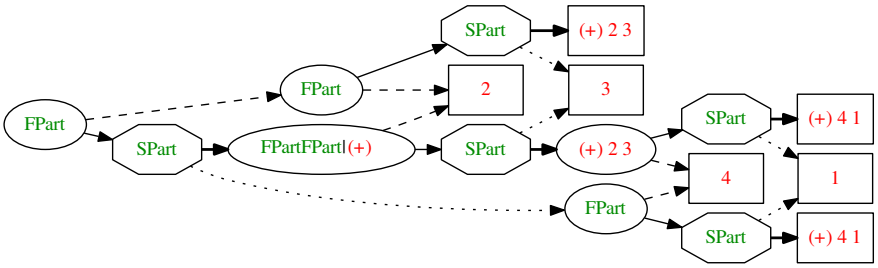
paralelismo.

Por ejemplo, en el caso que quisiéramos sumar todos los enteros de una lista, para explotar el paralelismo es conveniente realizar un poco más de trabajo, como contar los elementos de la lista y luego evaluar de forma paralela la sumatoria de la primer mitad, computar la segunda mitad y luego realizar la suma de los valores resultantes (ver Código 4.7). Podemos observar en la Figura 4.8 el contraste entre los gráficos resultantes de utilizar etiquetas sobre los nodos de comportamiento dinámico, y no utilizar etiquetas sobre estos.

Para obtener las representaciones gráficas resultantes de utilizar a *Klytius*, las cuales fueron mostradas en todo el trabajo, es necesario realizar algunas definiciones de la información que llevaran los nodos y etiquetas. Definimos los tipos Lbl y NLbl, para modelar las aristas y los nodos respectivamente, como se muestra a continuación.



(a) Paralelizar una lista sin etiquetas



(b) Paralelizar una lista mostrando todas las etiquetas

Figura 4.8: Diferencia entre mostrar etiquetas o no mostrarlas

```
data Lbl where -- Aristas
    Spark    :: Lbl
    LSeq     :: Lbl
    PExec    :: Lbl
    Exec     :: Unique → Lbl

data NLbl where -- Nodos
    NPar :: TLabel → NLbl
    NSeq :: TLabel → NLbl
    NVal :: TLabel → NLbl
```

Definiendo los tipos de datos de las aristas y nodos, podemos identificar los diferentes elementos generados del grafo reificado. Esto nos permite asignarles diferentes propiedades gráficas, como ser, los diferentes estilos de aristas vistos (punteada, contigua, o compuesta por guiones). Los nodos llevan información adicional: una etiqueta que luego se podrá mostrar en el gráfico para identificar las computaciones.

```
nodeSTD :: Bool → (Unique, NLbl) → Attributes
nodeSTD b (x,l@(NPar s)) =
    case b of
         True  → [labelStyle (show x) s, shapeNods l]
         False → [labelStyle "" s, shapeNods l]
nodeSTD b (x,l@(NSeq s)) =
    case b of
         True  → [labelStyle (show x) s, shapeNods l]
         False → [labelStyle "" s, shapeNods l]

edgeSTD :: Bool → (a,b,Lbl) → Attributes
edgeSTD (s,d, Spark) = [Style [SItem Dashed []]]
edgeSTD (s,d, GEdge) = [Style [SItem Dotted []]]
edgeSTD (s,d, LSeq)  = [Dir NoDir,Style [SItem Bold []]]
edgeSTD (s,d,Exec x) = [Style [SItem Bold [], SItem Solid []]]
edgeSTD (s,d,l)      = [emptyLbl]
```

Código 4.9: Estilo por defecto para nodos de tipo NPar y aristas que denotan un spark.

Mostraremos cómo se da formato a algunos casos, para que sea más fácil la lectura. Definimos dos funciones, nodeSTD y edgeSTD (Código 4.9), las cuales utilizamos dentro de *Klytius* para dar un formato gráfico a las computaciones, el cual puede ser modificado fácilmente.

Las funciones nodeSTD y edgeSTD, son las que definen el estilo de los nodos y las aristas utilizadas en la definición de grafiParams, como fmtNode y fmtEdge respectivamente.

Gracias al alto orden de las funciones en Haskell, y que el tipo de datos Attributes es simplemente una lista de atributos, es muy sencillo redefinir los atributos de los diferentes elementos, en el caso que el usuario quisiera realizar modificaciones visuales.

Manipulación de elementos dentro de *Klytius*

Para facilitar el uso de la herramienta, se implementan mecanismos de manipulación de elementos embebidos en el EDSL. Esto se logra mediante las instancias de funtor, aplicativo, y mónada del EDSL, como se muestra en el fragmento de Código 4.10. Todas las instancias aplican la transformación sobre el hijo derecho de los constructores. Esto se debe a la semántica que representan; recordemos que **par** a b = b (igual que **pseq** a b = b), por lo que si aplicamos una función, f (**par** a b) = f b (igual que f (**pseq** a b) = f b).

Utilizando la instancia aplicativa en el caso que se obtenga una función con cierto comportamiento dinámico asociado, (f :: **TPar** (a →b)), y un elemento con cierto comportamiento, (x :: **TPar** a), podemos aplicar la función sobre el elemento (f <*> x), acumulando los efectos necesarios para determinar el valor de f, y luego los efectos necesarios para determinar el valor de

```
instance Functor (TPar a) where
    fmap f (Par l r) = Par l (fmap f r)
    fmap f (Seq l r) = Seq l (fmap r)
    fmap f (Val x)   = Val (f x)

instance Applicative TPar where
    pure = Val
    (Par l f) <*> x = Par l (f <*> x)
    (Seq l f) <*> x = Seq l (f <*> x)
    (Val f)   <*> x = fmap f x

instance Monad TPar where
    return = Val
    (Par l r)  >>= f = Par l (r >>= f)
    (Seq l r)  >>= f = Seq l (r >>= f)
    (Val x)    >>= f = f x
```

Código 4.10: EDSL para paralelismo y sus instancias.

```
resum = map (fmap (+1)) [Just 23, Just 4, Just 79014] `using`
    (parList rdeepseq)
const2 :: a → b → b
const2 _ y = y
```

Código 4.11: Efectos dinámicos lazy.

x, y por último, realizar la aplicación de la función, f x. Dicha acumulación de comportamiento dinámico no concuerda con la evaluación perezosa de los efectos dinámicos. Por ejemplo, dentro del Código 4.11, podemos tener un valor con cierto comportamiento dinámico resum, y una función const2 que toma dos argumentos y devuelve el segundo. Nos podemos preguntar si al evaluar la expresión const2 resum 42, todo, nada, o alguna parte del comportamiento dinámico puede ser disparado. Como indica la evaluación perezosa, a menos que la función sea estricta, el comportamiento dinámico no será evaluado.

Dentro de este capítulo repasamos las técnicas utilizadas para la implementación de la herramienta, definimos las bases teóricas para la creación de un EDSL profundo, cómo detectar las computaciones compartidas, cómo son graficadas las estructuras dinámicas y por último cómo son operados los elementos dentro del EDSL.

Capítulo 5

Conclusiones

Dada la (inevitable) presencia de procesadores multinúcleos dentro de la mayoría de los dispositivos electrónicos, surge la necesidad de ejecutar programas de forma paralela. Es decir, se necesita ejecutar varios procesos de forma simultánea para arribar a la solución de forma más rápida. Un enfoque consiste en particionar el programa en subprogramas, probando diferentes soluciones en paralelo y quedándose con la que termine primero, etc. Sin embargo,al momento de definir el uso de los diferentes procesadores nos encontramos con problemáticas de bajo nivel, como ser la distribución de tareas a los diferentes procesadores, y además cuestiones de más alto nivel, por ejemplo modificar el diseño del algoritmo para maximizar la paralelización del mismo. Si bien gran parte de estos problemas se pueden resolver mediante análisis estáticos por parte del compilador dentro de los lenguajes funcionales, nos queda determinar qué tareas realizamos en paralelo, y cómo organizamos estas tareas. Dentro de Haskell los combinadores básicos de paralelismo puro semi-explícito **par** y **pseq**, nos permiten indicar exactamente las tareas antes mencionadas.

Al utilizar los combinadores básicos, los programas son modificados para indicar el comportamiento dinámico buscado, sin embargo, el código se obscurece con los operadores básicos para asegurar la correcta ejecución de programas paralelos. Además, debido a que el comportamiento dinámico impone ciertas condiciones sobre el momento en el cual se evalúan ciertas expresiones, entra en conflicto con la evaluación perezosa, produciendo que el trabajo a realizar de los diferentes *sparks* se vuelve incierto. A fin de solucionar estos problemas, se introdujeron las estrategias, como mecanismo para separar el algoritmo de su comportamiento dinámico lo más posible, y controlar la evaluación de los diferentes elementos definiendo instancias NFData para asegurar su completa evaluación.

A pesar de que las estrategias logran separar efectivamente el algoritmo de su comportamiento dinámico, y permiten establecer la profundidad de la evaluación de los diferentes elementos, el programador continua sin un entendimiento profundo de si el trabajo se logra realizar, de si es necesario forzar aún

más la evaluación de los elementos y de si son efectivamente las computaciones que se quieren paralelizar o secuencializar las que se indicaron mediante el uso de los combinadores.

Incorporar paralelismo a los requerimientos del desarrollo de un programa es difícil, como se muestra a lo largo de este trabajo, se desarrolló un EDSL que permite observar de forma más detenida la estructura del comportamiento dinámico que se desarrolla a lo largo del programa.

Al momento de desarrollar un EDSL, es necesario tener en claro cuáles son las expresiones que se quieren observar, y qué información se quiere conservar para luego poder realizar ciertos análisis sobre la estructura generada. Debido a que los programas paralelos se construyen en base a las primitivas básicas **pseq** y **par**, conservar las estructuras generadas por ellos es de vital importancia. Además, es posible que se necesite mas información para realizar un análisis más detallado sobre los programas y su comportamiento dinámico, por lo que se incluyeron etiquetas para permitirle al programador asignarle un mayor valor semántico a los nodos.

Durante el desarrollo de la herramienta, fueron probadas diferentes representaciones, hasta llegar al EDSL propuesto. Estas fueron descartadas debido a que no representaban correctamente las estructuras dinámicas del programa o introducían más nodos sin información importante.

Al mismo tiempo, se investigaron diferentes técnicas de detección de computaciones compartidas. Decidimos finalmente utilizar la técnica de Gill dado que se priorizó que la herramienta sea de fácil uso, y que simplifique la transición con la librería *Control.Parallel*.

Debido a la evaluación perezosa, el EDSL tal como ha sido presentado en el capítulo anterior representa correctamente sólo aquellas funciones que sean estrictas en sus argumentos, marcando aquí una limitación de *Klytius*.

Además de proponer un EDSL para observar la estructura dinámica generada por los programas, proponemos una representación gráfica de las mismas. Como se observó en los capítulos 2 y 3, el resultado es un grafo donde se presentan como nodos los diferentes constructores del EDSL, conectados con aristas que representan secuenciación o creación de sparks. Los gráficos generados tienden a crecer rápidamente debido a la cantidad de elementos que se pueden observar al momento de realizar ejecuciones. Además la generación de nodos se encuentra fuertemente ligada a la forma que el programador utiliza los combinadores de paralelismo.

La construcción del EDSL permite al programador observar directamente el comportamiento dinámico de los programas. Esto es particularmente útil al momento de experimentar con nuevas abstracciones, ya que es posible contrastar el comportamiento dinámico generado por las nuevas abstracciones con el de abstracciones conocidas, ahora observables gracias a nuestra herramienta.

La posibilidad de observar las estructuras dinámicas generadas por los programas, particularmente la habilidad de generar gráficos que representan

dichas estructuras, facilita el aprendizaje de paralelismo sobre lenguajes funcionales.

El paralelismo dentro de los programas ya no es un efecto invisible que hace que los programas se ejecuten más rápidos, sino que son estructuras visibles que definen un comportamiento sobre los valores de los programas.

Se lograron alcanzar los objetivos específicos propuestos y se definió e implementó un EDSL en Haskell, el cual nos permite capturar los efectos generados por los combinadores básicos **par** y **pseq**. Además, se implementó un conjunto de operadores sobre dicho EDSL para que el usuario final pueda manipular fácilmente los elementos generados, y se propuso una representación gráfica para elementos dentro del EDSL, permitiendo al usuario observar las computaciones generadas.

5.1. Trabajo Futuro

Si bien la herramienta ya puede ser utilizada hay varias formas de mejorarla para facilitar su uso.

Modificar los programas para utilizar *Klytius* es fácil e intuitivo, basta con reemplazar los combinadores básicos **par** y **pseq** de la librería Control.Parallel por los de la herramienta, y las aplicaciones por (<$>), (<∗>), o (=<<) según corresponda. Este procedimiento se podría automatizar utilizando las facilidades de meta-programación provistas por *Template Haskell*, la cuál permite tomar un fragmento de código Haskell y manipularlo.

Al momento de depurar e inspeccionar los programas dentro de Haskell, es muy común utilizar el intérprete GHCi. El intérprete, a su vez, exhibe toda la información sobre las computaciones y el estado general del sistema. Es decir, es posible observar toda la configuración del sistema, información que podría ser muy útil para saber qué expresiones son evaluadas, si se realiza el trabajo esperado, etc. Se propone utilizar esta información en conjunto con la herramienta para permitirle al usuario observar directamente la estructura generada por el comportamiento dinámico en *Klytius* desde GHCi.

Dado que en general los gráficos producidos son muy grandes, se plantea implementar una interfaz que permita explorar los gráficos producidos de manera interactiva. En conjunto con el punto anterior, permitir que el usuario pueda interactuar entre el gráfico y el intérprete GHCi, mostrando de manera elegante información del sistema.

Dado que la estructura dinámica se encuentra muy ligada a la forma en que el programador utilizó los combinadores básicos de paralelismo, se puede buscar ciertas reglas que permitan encontrar una estructura semánticamente equivalente, pero que permita desacoplar o independizar la estructura dinámica generada de la forma en que fue programada. Se plantea definir una forma normal de expresiones del EDSL, y un conjunto de reglas, que permitan presentar una estructura dinámica esencial y no ligada al estilo del programador.

Bibliografía

Analysing Event Logs (2015). http://www.haskell.org/haskellwiki/Ghc-events. (Visitado 27-03-2015).

Arvind, Rishiyur S. Nikhil y Keshav K. Pingali (1989). "I-structures: Data Structures for Parallel Computing". En: *ACM Trans. Program. Lang. Syst.* 11.4, págs. 598-632. ISSN: 0164-0925. DOI: 10.1145/69558.69562. URL: http://doi.acm.org/10.1145/69558.69562.

Augustsson, Lennart (1984). "A Compiler for Lazy ML". En: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: ACM, págs. 218-227. ISBN: 0-89791-142-3. DOI: 10.1145/800055.802038. URL: http://doi.acm.org/10.1145/800055.802038.

Augustsson, Lennart y Thomas Johnsson (1989). "Parallel Graph Reduction with the (V , G)-machine". En: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, págs. 202-213. ISBN: 0-89791-328-0. DOI: 10.1145/99370.99386. URL: http://doi.acm.org/10.1145/99370.99386.

Bentley, Jon (1986). "Programming Pearls: Little Languages". En: *Commun. ACM* 29.8, págs. 711-721. ISSN: 0001-0782. DOI: 10.1145/6424.315691. URL: http://doi.acm.org/10.1145/6424.315691.

Berthold, Jost y col. *Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer*.

Borkar, Shekhar y Andrew A. Chien (2011). "The Future of Microprocessors". En: *Commun. ACM* 54.5, págs. 67-77. ISSN: 0001-0782. DOI: 10.1145/1941487.1941507. URL: http://doi.acm.org/10.1145/1941487.1941507.

Al Saeed, Majed, Phil Trinder y Patrick Maier (2013). *Critical Analysis of Parallel Functional Profilers*. School of Mathematical & Computer Sciences, Heriot-Watt University, Scotland, UK, EH14 4AS. HW-MACS-TR-0099.

Chakravarty, Manuel M. T., Gabriele Keller y Simon Peyton Jones (2005). "Associated Type Synonyms". En: *SIGPLAN Not.* 40.9, págs. 241-253. ISSN: 0362-1340. DOI: 10.1145/1090189.1086397. URL: http://doi.acm.org/10.1145/1090189.1086397.

Chakravarty, Manuel M. T., Roman Leshchinskiy y col. (2007). "Data Parallel Haskell: A Status Report". En: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. DAMP '07. Nice, France: ACM, págs. 10-18. ISBN: 978-1-59593-690-5. DOI: 10.1145/1248648.1248652. URL: http://doi.acm.org/10.1145/1248648.1248652.

Chakravarty, Manuel M.T., Gabriele Keller, Sean Lee y col. (2011). "Accelerating Haskell Array Codes with Multicore GPUs". En: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. Austin, Texas, USA: ACM, págs. 3-14. ISBN: 978-1-4503-0486-3. DOI: 10.1145/1926354.1926358. URL: http://doi.acm.org/10.1145/1926354.1926358.

Claessen, Koen y David Sands (1999). "Observable sharing for functional circuit description". En: *In Asian Computing Science Conference*. Springer Verlag, págs. 62-73.

Dean, Jeffrey y Sanjay Ghemawat (2008). "MapReduce: Simplified Data Processing on Large Clusters". En: *Commun. ACM* 51.1, págs. 107-113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

Dennard, Robert H. y col. (1974). "Design of ion-implanted MOSFETs with very small physical dimensions". En: *IEEE J. Solid-State Circuits*, pág. 256.

Flynn, Michael J. (1966). "Very High-Speed Computing Systems". En: *Proceedings of the IEEE* 54.12, págs. 1901-1909. ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273. URL: http://www.cs.utexas.edu/users/dburger/teaching/cs395t-s08/papers/5_flynn.pdf.

Forum, Message Passing Interface (2012). *MPI: A Message-Passing Interface Standard Version 3.0*. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

Gansner, Emden R., Eleftherios Koutsofios y Stephen North (2005). *Drawing graphs with dot*. http://www.graphviz.org/pdf/dotguide.pdf. (Visitado 27-03-2015).

Gansner, Emden R. y Stephen C. North (2000). "An open graph visualization system and its applications to software engineering". En: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11, págs. 1203-1233.

Gill, Andy (2009). "Type-safe Observable Sharing in Haskell". En: *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. Edinburgh, Scotland: ACM, págs. 117-128. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596653. URL: http://doi.acm.org/10.1145/1596638.1596653.

— (2014). "Domain-specific Languages and Code Synthesis Using Haskell". En: *Queue* 12.4, 30:30-30:43. ISSN: 1542-7730. DOI: 10.1145/2611429.2617811. URL: http://doi.acm.org/10.1145/2611429.2617811.

Graph Visualization Software (2015). http://www.haskell.org/haskellwiki/Ghc-events. (Visitado 27-03-2015).

Hammond, Kevin (1994). "Parallel Functional Programming: An Introduction". En: *International Symposium on Parallel Symbolic Computation*. World Scientific.

Harris, Tim, Simon Marlow y Simon Peyton Jones (2005). "Haskell on a Shared-memory Multiprocessor". En: *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. Haskell '05. Tallinn, Estonia: ACM, págs. 49-61. ISBN: 1-59593-071-X. DOI: 10.1145/1088348.1088354. URL: http://doi.acm.org/10.1145/1088348.1088354.

Hudak, Paul (1986). "Para-Functional Programming". En: *Computer* 19.8, págs. 60-70. ISSN: 0018-9162. DOI: 10.1109/MC.1986.1663309. URL: http://dx.doi.org/10.1109/MC.1986.1663309.

— (1996). "Building Domain-specific Embedded Languages". En: *ACM Comput. Surv.* 28.4es. ISSN: 0360-0300. DOI: 10.1145/242224.242477. URL: http://doi.acm.org/10.1145/242224.242477.

Hudak, Paul y Benjamin Goldberg (1985). "Serial Combinators: "Optimal" Grains of Parallelism". En: *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag New York, Inc., págs. 382-399. ISBN: 3-387-15975-4. URL: http://dl.acm.org/citation.cfm?id=5280.5304.

Johnsson, Thomas (1984). "Efficient Compilation of Lazy Evaluation". En: *SIGPLAN Not.* 19.6, págs. 58-69. ISSN: 0362-1340. DOI: 10.1145/502949.502880. URL: http://doi.acm.org/10.1145/502949.502880.

Jones Jr, Don, Simon Marlow y Satnam Singh (2009). "Parallel Performance Tuning for Haskell". En: *ACM SIGPLAN 2009 Haskell Symposium*. (to appear). Association for Computing Machinery, Inc. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=80976.

Keller, Gabriele y col. (2010). "Regular, Shape-polymorphic, Parallel Arrays in Haskell". En: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, págs. 261-272. ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863582. URL: http://doi.acm.org/10.1145/1863543.1863582.

Kieburtz, Richard B. (1985). "The G-machine: A Fast, Graph-reduction Evaluator". En: *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag New York, Inc., págs. 400-413. ISBN: 3-387-15975-4. URL: http://dl.acm.org/citation.cfm?id=5280.5305.

Loidl, Hans-Wolfgang y col. (2008). "Semi-Explicit Parallel Programming in a Purely Functional Style: GpH". En: *Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development*. Ed. por Michael Alexander y Bill Gardner. Chapman y Hall, págs. 47-76. URL: http://www.macs.hw.ac.uk/~trinder/papers/PAPDP.pdf.

Loogen, Rita (2012). "Eden – Parallel Functional Programming with Haskell". English. En: *Central European Functional Programming School*. Ed.

por Viktória Zsók, Zoltán Horváth y Rinus Plasmeijer. Vol. 7241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, págs. 142-206. ISBN: 978-3-642-32095-8. DOI: 10.1007/978-3-642-32096-5_4. URL: http://dx.doi.org/10.1007/978-3-642-32096-5_4.

Marlow, Simon (2010). *Haskell 2010 Language Report*.

Marlow, Simon, Ryan Newton y Simon P. Jones (2011). "A monad for deterministic parallelism". En: *Proceedings of the 4th ACM symposium on Haskell*. Haskell '11. Tokyo, Japan: ACM, págs. 71-82. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034685. URL: http://dx.doi.org/10.1145/2034675.2034685.

Marlow, Simon, Ryan Newton y Simon Peyton Jones (2011). "A Monad for Deterministic Parallelism". En: *Proceedings of the 4th ACM Symposium on Haskell*. Haskell '11. Tokyo, Japan: ACM, págs. 71-82. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034685. URL: http://doi.acm.org/10.1145/2034675.2034685.

Runtime Support for Multicore Haskell (2009). Association for Computing Machinery, Inc. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=79856.

Marlow, Simon y col. (2010a). "Seq No More: Better Strategies for Parallel Haskell". En: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, págs. 91-102. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863535. URL: http://doi.acm.org/10.1145/1863523.1863535.

Marlow, Simon y col. (2010b). "Seq no more: Better Strategies for Parallel Haskell". En: *Haskell '10: Proceedings of the Third ACM SIGPLAN Symposium on Haskell*. Baltimore, USA: ACM. URL: http://community.haskell.org/~simonmar/papers/strategies.pdf.

Matthew Sackman, Ivan Lazar Miljenovic (2015). *GraphViz Haskell Library*. http://hackage.haskell.org/package/graphviz. (Visitado 27-03-2015).

Mazor, Stanley, Senior Member y Robert Noyce (1995). "The History of the Microcomputer-Invention and Evolution". En: *Proceedings of the IEEE*, págs. 1601-1607.

McBride, Conor y Ross Paterson (2007). "Applicative programming with effects". En: *Journal of Functional Programming* 18.01, págs. 1-13. DOI: 10.1017/s0956796807006326. URL: http://dx.doi.org/10.1017/s0956796807006326.

Moore, Shirley y col. (2001). "Review of Performance Analysis Tools for MPI Parallel Programs". En: *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, págs. 241-248. ISBN: 3-540-42609-4. URL: http://dl.acm.org/citation.cfm?id=648138.746655.

Noguchi, Kenichiro, Isao Ohnishi e Hiroshi Morita (1975). "Design Considerations for a Heterogeneous Tightly-coupled Multiprocessor System". En:

Proceedings of the May 19-22, 1975, National Computer Conference and Exposition. AFIPS '75. Anaheim, California: ACM, págs. 561-565. DOI: 10.1145/1499949.1500062. URL: http://doi.acm.org/10.1145/1499949.1500062.

O'Sullivan, Bryan, John Goerzen y Don Stewart (2008). *Real World Haskell*. 1st. O'Reilly Media, Inc. ISBN: 0596514980, 9780596514983.

Peyton Jones, S. L. (1989). "Parallel Implementations of Functional Programming Languages". En: *Comput. J.* 32.2, págs. 175-186. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.175. URL: http://dx.doi.org/10.1093/comjnl/32.2.175.

Roe, Paul (1991). *Parallel Programming using Functional Languages*.

Skillicorn, David B. y Domenico Talia (1998). "Models and Languages for Parallel Computation". En: *ACM Comput. Surv.* 30.2, págs. 123-169. ISSN: 0360-0300. DOI: 10.1145/280277.280278. URL: http://doi.acm.org/10.1145/280277.280278.

Strategies (2015). http://hackage.haskell.org/package/parallel-3.2.0.6/docs/Control-Parallel-Strategies.html. (Visitado 27-03-2015).

Svensson, Joel, Mary Sheeran y Koen Claessen (2011). "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors". En: *Implementation and Application of Functional Languages*. Ed. por Sven-Bodo Scholz y Olaf Chitil. Vol. 5836. Lecture Notes in Computer Science. Springer Berlin Heidelberg, págs. 156-173. ISBN: 978-3-642-24451-3. DOI: 10.1007/978-3-642-24452-0_9. URL: http://dx.doi.org/10.1007/978-3-642-24452-0_9.

Trinder, P. W., K. Hammond y col. (1998). "Algorithm + Strategy = Parallelism". En: *J. Funct. Program.* 8.1, págs. 23-60. ISSN: 0956-7968. DOI: 10.1017/S0956796897002967. URL: http://dx.doi.org/10.1017/S0956796897002967.

Trinder, P.W., H-W. Loidl y R.F. Pointon (2002). "Parallel and Distributed Haskells". En: *Journal of Functional Programming* 12.4&5, págs. 469-510.