Simulación de Programas Paralelos en Haskell

Martín A. Ceresa

CIFASIS, Rosario, Argentina

Resumen El estudio de 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 ha 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 herramienta permite al programador observar directamente qué computaciones se van a paralelizar, dándole un mayor entendimiento sobre la ejecución de su programa.

Keywords: parallel functional programming, Haskell, Klytius

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 ha 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 ha 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 tanto al momento de dar soporte a la experimentación con nuevas formas de incorporar paralelismo, introducir nuevos estudiantes a la programación paralela, o para comprender el comportamiento de los programas.

En este trabajo se le otorga al programador la posibilidad de 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 artículo se encuentra estructurado de la siguiente forma: sección 1, introducción al paralelismo y sus problemáticas; sección 2, introducción al paralelismo básico en Haskell y a Klytius; sección 3, implementación de Klytius; sección 4, trabajo relacionado; sección 5, conclusiones y trabajo futuro.

1.1. Problemáticas del Paralelismo

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. configuración del hardware disponible, cantidad de núcleos disponibles, cómo distribuir las diferentes tareas y establecer canales de comunicación entre ellos.

Debido a limitaciones físicas no es posible continuar aumentando la velocidad de procesamiento de los microprocesadores. Esto produce que el programador deba incorporar el paralelismo dentro del diseño de sus programas, o modificar sus algoritmos.

El programador debe dedicar tiempo y esfuerzo en pensar cómo paralelizar sus programas, y en el caso de que sean modificados, garantizar que no se introducen nuevos errores.

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, en la práctica no es tan fácil obtener un mayor rendimiento. Esto sucede principalmente por dos razones:

- No todo es paralelizable. 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.
- Paralelizar no es *gratis*. Aún dados algoritmos que son totalmente paralelizables, deberemos saber cómo dividiremos los datos, 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.

1.2. 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 utilizar lenguajes funcionales puros obtenemos las siguientes ventajas (Roe 1991):

- Nos permite razonar sobre los programas paralelos de la misma manera que los secuenciales.
- Nos permite delegar todos los aspectos de coordinación de computaciones al *runtime*.
- No existe la posibilidad de que haya *deadlock*, excepto en condiciones donde su versión secuencial no termine debido a dependencias cíclicas.

1.3. Contribuciones

Hemos desarrollado un lenguaje de dominio específico embebido en Haskell para la simulación simbólica de programas paralelos. A su vez, hemos asignado una representación gráfica a las diferentes construcciones del lenguaje de dominio específico, que en conjunto permite al programador observar directamente qué computaciones se van a paralelizar, dándole un mayor entendimiento sobre la ejecución de su programa.

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.

El presente artículo es una versión reducida del trabajo de finalización de la carrera de grado de Licenciatura en Ciencias de la Computación. El trabajo fue realizado bajo la supervisión de Mauro Jakelioff y Ezequiel Rivas, a los cuales estoy muy agradecido por su dedicación.

2. Paralelismo Puro en Haskell

2.1. Particionamiento Semi-Explícito.

Dentro de los lenguajes funcionales se encuentran dos enfoques para paralelismo: *a*) particionamiento implícito, donde el compilador es el encargado de paralelizar los programas; *b*) particionamiento explícito, donde el programador es el encargado no solo de detectar computaciones a paralelizar, sino que además debe proveer mecanismos de organización de computaciones, como ser, cómo distribuir las tareas en los procesadores (Hammond 1994).

El lenguaje de programación Haskell presenta un enfoque intermedio, donde el programador es quien indica cuáles son las computaciones que se pueden paralelizar, mientras que el *runtime* es el encargado de la coordinación de las computaciones (Loidl y col. 2008). Este enfoque se denomina *particionamiento semi-explícito* de programas. Esta división no es arbitraria, sino que se representan dos niveles de abstracción diferentes: *a*) un nivel bajo que depende del hardware de la computadora, como ser, cantidad de procesadores, tecnología disponible para la comunicación entre los procesadores, donde el compilador y

el entorno de ejecución son los encargados de utilizar dichos recursos; *b*) un nivel alto el cual es especificado por el programador, y es inherente al programa, independiente del hardware en el cual se vaya a ejecutar o del estado del sistema en el momento de la ejecución.

2.2. Primitivas Básicas de Paralelismo.

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 puro 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, dejando la coordinación de computaciones al compilador en conjunto con el *runtime*.

Definiremos como *comportamiento dinámico* a los efectos generados por el uso de los combinadores básicos de paralelismo, los cuales son reflejados en tiempo de ejecución.

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.

Los combinadores básicos **par** y **pseq**, nos permiten expresar paralelismo de manera fácil, pero no siempre de manera correcta. Por ejemplo en el fragmento de Código 1 se muestra cómo paralelizar la evaluación de un entero (x), con la evaluación del resultado de la expresión (x+y). Pero el programa mostrado en el fragmento de Código 1, 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

[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)
```

Código 1: Suma de enteros utilizando **par**.

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

Código 2: Suma de enteros paralelos.

evaluar ((+) x y), **eliminando** toda posibilidad de paralelismo. La forma correcta de definir una función suma que evalúe sus argumentos en paralelo y luego realice la suma de sus valores es la que se muestra en el fragmento de Código 2, donde se utiliza el combinador **pseq** para forzar la evaluación de y, dejando la posibilidad que el *spark* destinado a la evaluación de x pueda realizar su trabajo.

2.3. Observando Paralelismo

Utilizar los combinadores básicos de paralelismo resulta sencillo, pero no nos otorga ninguna garantía que los programas se ejecuten efectivamente de forma paralela. Por ejemplo, en el caso de sumar dos enteros (ver Código 1) se elimina el paralelismo debido a que el operador (+) es estricto en su primer argumento.

Para estudiar de forma más profunda el paralelismo dentro de Haskell, ya sea al momento de diseñar un programa, o al experimentar nuevas abstracciones de paralelismo, es útil contar con una estructura visible del comportamiento de nuestros programas.

Proponemos la utilización de la herramienta *Klytius* la cual permite observar directamente la estructura dinámica de los programas, e incluye la posibilidad de representar gráficamente dicha estructura. Por ejemplo, para el caso de la suma paralela de enteros (ver Código 2) podemos representarlo como se muestra en la Figura 3, donde la elipse representa al combinador básico **par**, del cual se desprenden dos nodos, con la línea segmentada marcamos la computación que es evaluada en paralelo, mientras que con la línea continua marcamos la computación resultante. El hexágono representa al combinador **pseq**, y con una línea punteada la computación que es evaluada primero, y con una linea continua la computación resultante. Los valores se representan dentro de cajas.

Para representar un programa dentro de *Klytius*, se deben realizar pequeñas modificaciones al código de la función. A continuación, se puede observar a modo de comparación los dos códigos de la función parsuma, a izquierda utilizando la

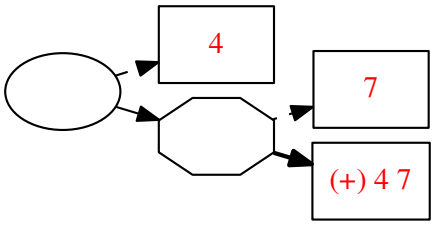

Figura 3. Representación gráfica de parsuma 4 7.

librería Control.Parallel y a derecha a *Klytius*.

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

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

El constructor de tipo **TPar**, nos permite indicar que estamos ante la presencia de cierto comportamiento dinámico adicional, el cual se encuentra encapsulado dentro de **TPar**. La función **mkVars** de tipo a → **TPar** a nos permite insertar elementos dentro del tipo **TPar**, sin comportamiento dinámico adicional, pero con la capacidad de adquirirlo.

Paralelizar la suma de enteros. Un ejemplo un poco más complejo es sumar una lista de enteros. Mostraremos dos aproximaciones a la solución.

Una primera aproximación es pensar en paralelizar la evaluación del elemento a la cabeza de la lista, con la evaluación de una llamada recursiva sobre el resto de la lista, para luego sumar los resultados. De forma similar al ejemplo anterior, se puede implementar como se muestra en el fragmento de Código 4.

Podemos obtener una representación gráfica que se muestra en la Figura 5, donde se observa una repetición del comportamiento dinámico al momento de realizar un *spark* de las computaciones, por ejemplo, el gráfico comienza con una elipse, y continúa a un hexágono, para luego bifurcarse en dos computaciones que aparentemente tienen la misma estructura. Esto se debe a que no es exactamente la misma estructura, sino que el valor resultante es diferente, y se puede diferenciar mediante la etiqueta de la elipse, (+) 3. El patrón se origina en la expresión **pseq** xs' (x'+xs'), donde la suma de dos valores con comportamiento dinámico, es evaluar el comportamiento dinámico de x', luego el de xs',

```
sumal :: [Int] → TPar Int
sumal [] = mkVars 0
sumal [x] = mkVars x
sumal (x:xs) = par x' (pseq xs' (x' + xs'))
    where
        x' = mkVars x
        xs' = sumal xs
```

Código 4: Función sumal.

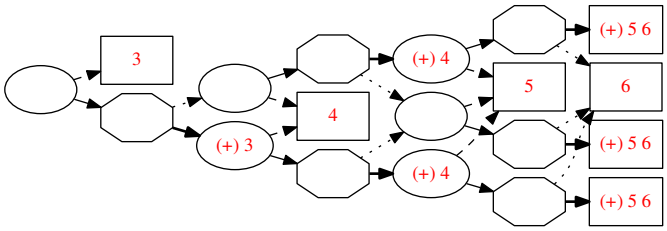

Figura 5. Representación gráfica de sumal [3,4,5,6]

y por ultimo realizar la suma de los valores que representan. Como x' no posee comportamiento dinámico, sólo observamos el comportamiento de xs'. Esto da origen a la repetición de estructuras en el gráfico, pero aquí se observa que en realidad contienen diferentes valores, en xs' representa el resultado de sumar la *cola* de la lista, mientras que en x'+xs' representa el resultado de sumar toda la lista, aunque ambas contienen el mismo comportamiento dinámico.

Podemos observar a simple vista el efecto de las computaciones paralelas, el cual evalúa los valores que comparten con el resto de las computaciones, como ser, la evaluación de las expresiones dentro de las cajas.

Las etiquetas del gráfico en conjunto con la gran cantidad de **pseq** que se observan, nos indican que estamos ante la presencia de un proceso secuencial. De la manera en que implementamos sumal, utilizamos el paralelismo para evaluar las expresiones de los enteros que tomamos como argumento, pero **no** paralelizamos la aplicación de la suma. Dada una lista [x1, x2, ..., xn], sumal [x1, x2, ..., xn] evalúa las expresiones x1, x2, ..., xn en paralelo, pero devuelve el resultado x1 + (x2 + ... (xn−1 + (xn)) ...). Si seguimos las etiquetas de las elipses, podemos observar de forma simple cuando van ocurriendo estas sumas.

Para explotar todo el paralelismo de nuestro algoritmo podemos dividir la lista en dos, evaluar las llamadas recursivas en paralelo, y luego sumarlas. Una forma de implementar ésta idea es como se muestra en el fragmento de Código 6.

```
dqsuma :: [Int] → TPar Int
dqsuma xs = dqsuma' (length xs) xs

dqsuma' :: Int → [Int] → TPar Int
dqsuma' _ []  = mkVars 0
dqsuma' _ [x] = mkVars x
dqsuma' 0 xs = error (show xs)
dqsuma' n xs  = par lxs (pseq rxs (lxs + rxs))
    where
        n2 = div n 2
        (lxs',rxs') = splitAt n2 xs
        (lxs, rxs) = (dqsuma' n2 lxs', dqsuma' (n − n2) rxs')
```

Código 6: Algoritmo de suma de lista de enteros estilo divide & conquer

Podemos observar gráficamente la estructura dinámica del algoritmo dqsuma en la Figura 7, donde se ve directamente cómo la lista es divida en dos partes, y mientras un *spark* da origen a la evaluación de la primer mitad ([3,4]), se fuerza la evaluación del *spark* destinado a evaluar la segunda mitad ([5,6]).

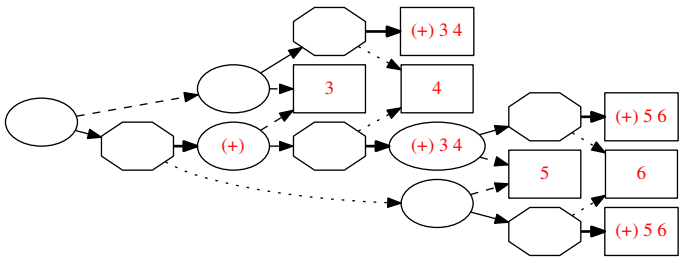

Figura 7. Representación gráfica de dqsuma [3,4,5,6]

3. Klytius

3.1. Lenguajes de Dominio Específico

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).

Los DSL tienen la desventaja de que necesitan del desarrollo de herramientas especiales, como ser, editores, compiladores, etc. 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. Al operar directamente sobre los valores obtenemos como resultado el valor 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 este árbol, 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.

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, lo que permite analizar dicha información de diferentes maneras. Así mismo al tener el árbol abstracto ya no se manipulan valores. 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.

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

Código 8: 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 9: Suma de enteros paralela

3.2. 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, Maier y col. 2010), 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 del lenguaje, definimos un tipo de datos como se muestra en el fragmento de Código 8.

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 (internos) 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 se ha mencionado, **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 9), obtenemos el siguiente árbol abstracto:

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

De esta manera es posible construir el árbol abstracto 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.

3.3. 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.

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*[2].

Particularmente al desarrollar programas paralelos en Haskell, es muy común que se compartan la mayoría de las computaciones. Por ejemplo, dentro de una expresión como `par p q` el objetivo de generar un *spark* para la evaluación de `p` es que su valor es necesario en `q`, donde en el caso que se pueda evaluar en paralelo, el programa se ejecutará más rápido.

Andy Gill presenta (Gill 2009) 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.

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.

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

- 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. Trabajo Relacionado

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 *runtime*. 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 herramienta gráfica 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, ligado al estado general del sistema y al hardware subyacente.

La extensión de Haskell, Eden (Loogen 2012), presenta una herramienta 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.

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

tes 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).

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, ni del hardware subyacente, sino que es inherente al código escrito. Ninguna de las herramientas antes mencionadas permiten realizar éste análisis.

5. Conclusiones

El estudio del comportamiento dinámico de los programas es difícil, particularmente porque se plantea como un efecto observable sólo al momento en que los programas son ejecutados. Por lo que se plantea un EDSL que permite observar de forma más detenida la estructura de dicho comportamiento.

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

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 observable 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.

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. Por lo que se propone como trabajo futuro, utilizar esta información en conjunto con Klytius para permitirle al usuario observar directamente la estructura generada por el comportamiento dinámico en *Klytius* desde GHCi.

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.

Referencias

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

Bentley, Jon (1986). "Programming Pearls: Little Languages". En: *Commun. ACM* 29.8, págs. 711-721. ISSN: 0001-0782.

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

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.

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.

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.

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

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.

— (2014). "Domain-specific Languages and Code Synthesis Using Haskell". En: *Queue* 12.4, 30:30-30:43. ISSN: 1542-7730.

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

Hudak, Paul (1996). "Building Domain-specific Embedded Languages". En: *ACM Comput. Surv.* 28.4es. ISSN: 0360-0300.

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.

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.

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.

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

Marlow, Simon, Patrick Maier y col. (2010). "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.

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.

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

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.

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.