Efecto de las transformaciones de GHC sobre Edén

Cristóbal Pareja Clara Segura

19 de mayo del 2000

Informe Técnico nº 101-00
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

Resumen

El lenguaje funcional paralelo Edén extiende al lenguaje Haskell con construcciones que definen procesos de forma explícita. Edén está implementado modificando el compilador GHC de Haskell, mediante la ocultación de las construcciones Edén en este lenguaje. GHC transforma Haskell en un lenguaje intermedio llamado Core, sobre el que lleva a cabo numerosas transformaciones, con el objetivo de optimizar el código producido. Se pretende determinar cómo dichas transformaciones afectan a las construcciones Edén que se hallan ocultas en Core. Se tienen en cuenta distintos aspectos de la semántica que se pueden ver alterados: lanzamiento indebido de procesos, cambio de trabajo entre procesos y modificación del comportamiento no determinista. También se estudian los efectos de dichas transformaciones sobre aspectos del rendimiento como el trasiego de información entre procesos y el gasto de memoria.

1 Introducción

El lenguaje funcional paralelo Edén [BLOMP96, BLOP97, KOMP98] extiende al lenguaje Haskell con construcciones que definen procesos de forma explícita. Edén está implementado modificando el compilador de Glasgow de Haskell (GHC) [JHH+93] y su sistema de ejecución paralela GUM [THJP96].

GHC transforma Haskell en un lenguaje intermedio llamado Core, ver Figura 1, sobre el que llevan a cabo numerosas transformaciones, con el objetivo de optimizar el código producido. En el proceso de compilación de Edén, se reutiliza el compilador GHC, escondiendo sus construcciones de abstracción e instanciación de procesos como operadores Haskell, con lo que en Core, dichas construcciones continúan aún escondidas en el lenguaje.

Se define un nuevo lenguaje intermedio, llamado CoreEdén, utilizado por el momento para llevar a cabo un análisis de bypassing [KPS00] y el lanzamiento impaciente de procesos [Rub99]. En él, las construcciones de Edén se hacen explícitas. Una vez terminado el análisis de bypassing volvemos al lenguaje Core, escondiendo de nuevo las construcciones de Edén, ver Figura 1. Después se continúa con el proceso normal de compilación, en el cual se incluiría la realización de las transformaciones que GHC lleva a cabo sobre Core.

Se estudia aquí en qué medida pueden afectar dichas transformaciones a la semántica del programa Edén representado por el correspondiente programa en Core. Para ello debemos tener en cuenta cómo se ha llevado a cabo la traducción de CoreEdén a Core. No se presentan aqui los detalles del lenguaje CoreEdén y la traducción entre CoreEdén y Core pues aqui no son relevantes, pero en la sección 3 se presentan los puntos fundamentales sobre el aspecto que tienen las construcciones Edén al volver de CoreEdén a Core. En la sección 2 presentamos aquellos aspectos del comportamiento de los programas Edén sobre los que podrían tener efectos las transformaciones. En la sección 4 se comentan una a una todas las transformaciones llevadas a cabo sobre

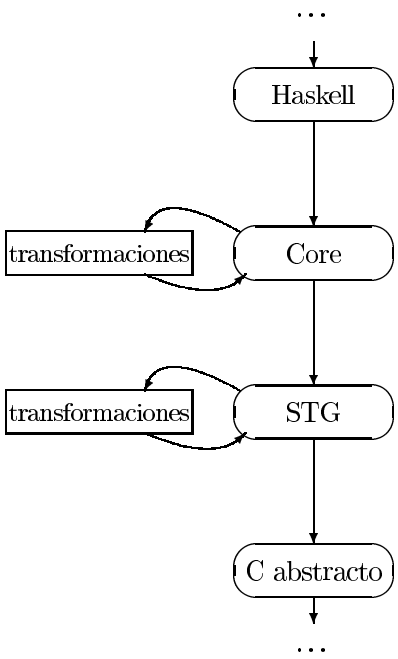

Figura 1: Los lenguajes intermedios del compilador GHC

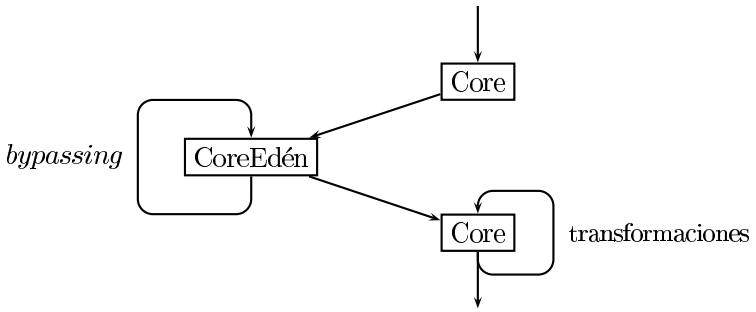

Figura 2: Esquema del paso por CoreEdén

Core [San95] [Jon96] [JPS96] [JS98] [JM99] con las posibles consecuencias para las construcciones de Edén. Finalmente, en la sección 5 presentamos algunos puntos sobre los que continuar trabajando.

2 Puntos de interés

A la hora de observar los efectos de las transformaciones nos centraremos en verificar:

- Que las transformaciones no precipiten el lanzamiento de procesos que, según la semántica de Edén no se lanzarían con seguridad.
- Que las transformaciones no cambien trabajo de un proceso a otro, en concreto de un proceso padre a uno hijo y viceversa.
- Que las transformaciones no cambien el comportamiento no determinista definido en un programa Edén.

  Observamos que la pérdida de no determinismo puede producir consecuencias muy negativas a nivel semántico, como la pérdida de posibilidad de terminación y la pérdida de valores posibles producidos. Un ejemplo de ello se muestra en la transformación de argumentos estáticos, en la sección 4.4.

  Una situación a tener en cuenta es aquélla en la que una variable ligada a una expresión no determinista se evalúa tanto en el padre como en el hijo. Por ejemplo si en Edén tenemos $(f\ a)\ \#\ a$ donde a está ligada a una expresión indeterminista, a se va a evaluar por una parte en el padre para alimentar el canal de entrada del hijo y por otra se va a copiar la clausura al hijo (posiblemente sin evaluar en absoluto) para evaluar en él $f\ a$, de forma que si f necesita la evaluación de a, el valor en el padre y en el hijo es posiblemente diferente. Debemos estudiar si esta situación, o alguna otra, podría generarse a partir de alguna transformación o si éstas solamente mantienen una situación problemática ya existente.

Adicionalmente haremos algunos comentarios sobre los efectos de las transformaciones sobre el trasiego de información entre los procesos, básicamente en relación con la copia de clausuras de las variables libres, allí donde sea procedente, ya que algunas transformaciones podrían respetar la semántica pero en cambio afectar negativamente a la eficiencia. Otro detalle estrechamente relacionado que podríamos tener en cuenta con respecto a la eficiencia es el malgasto de memoria que podrían suponer algunas transformaciones, lo cual se indicará también allí donde sea apropiado.

3 Cómo llega Edén a Core procedente de CoreEdén

Para estudiar los aspectos antes señalados, debemos saber cómo se hallan las construcciones de Edén escondidas en Core. Debemos tener en cuenta los siguientes aspectos:

- Una abstracción de proceso es una λ-abstracción, por lo que tenemos que tener en cuenta aquellas transformaciones que actúan sobre ellas. Todo lo que hay dentro de esa λ-abstracción debe evaluarse en el hijo, por lo que sacar cosas fuera de ellas podría cambiar trabajo del hijo al padre. Y de la misma forma introducir cosas dentro podría cambiar trabajo del padre al hijo. Además, debido a dichas transformaciones se podría cambiar también el proceso en el que se crean las clausuras, aumentando o disminuyendo en cada caso el trasiego de clausuras entre procesos.
- Si aparecen los dos argumentos de una instanciación de proceso, ésta se ha podido transformar en un *createProcessTL* $p\ vs$ y una expresión *case* posterior sobre la variable de salida a la que se encuentra ligada dicha aplicación. p estará ligada en alguna parte a una expresión potencialmente compleja, que debe evaluarse en el hijo, por lo que todas aquellas transformaciones que saquen o metan cosas en los lados derechos de las ligaduras de un *let* han de ser vigiladas estrechamente, pues podrían cambiar trabajo del padre al hijo y viceversa.
- Si no aparecen los dos argumentos de la instanciación de proceso, por ejemplo si tenemos una expresión como $zipWith\ (\#)\ ps\ xs$, no se ha podido llevar a cabo la transformación anterior. En este caso ha quedado la expresión como estaba y además el lanzamiento de procesos se llevará a cabo a medida que se vayan demandando los elementos de esa lista, con lo que no existe expresión *case* alguna.

- No debemos olvidar que en este punto ya se ha llevado a cabo el lanzamiento impaciente de procesos, el cual se refleja en la aparición de una expresión *case* allí donde corresponda. Dicho forzamiento de la evaluación de las salidas de un proceso se lleva a cabo incluso en aquéllos que no se encuentran a nivel superior, pero en ese caso, la expresión *case* que fuerza la evaluación se encuentra en el lado derecho de una ligadura *let* y por tanto hasta que no se demande el valor de la variable, no se lanzará el proceso, manteniéndose así la semántica deseada. Por ejemplo, en una expresión Edén como:

$$\begin{array}{l}\textbf{let}\\ \quad v = \textbf{let } o = p \ \# \ i \ \textbf{in } e'\\ \textbf{in } e\end{array}$$

  la instanciación de p con i no se lanzaría hasta que v fuera demandada en e. En Core tendremos algo semejante a:

$$\begin{array}{l}\textbf{let}\\ \quad v = \textbf{let } o' = \mathit{createProcessTL}\ p\ i \ \textbf{in case } o' \ \textbf{of } (\mathit{Lift}\ o) \rightarrow e'\\ \textbf{in } e\end{array}$$

  De esta forma, cuando en e se demande el valor de v, se llevará a cabo el lanzamiento del proceso. Esto quiere decir que hay que tener cuidado con las reglas de transformación que muevan construcciones *case*, pues se podrían lanzar procesos indebidamente.

Teniendo en cuenta estas consideraciones, hacemos a continuación un recorrido por las transformaciones locales y globales, proporcionando razonamientos sobre su influencia negativa o la ausencia de ella.

4 Efectos de las transformaciones

4.1 Transformaciones locales

4.1.1 β-reducción

Transformación: Esta transformación tiene dos versiones, una para variables y otra para expresiones generales:

- $(\lambda x \rightarrow \mathit{body})\ y \quad \Rightarrow \quad \mathit{body}[y/x]$
- $(\lambda x \rightarrow \mathit{body})\ e \quad \Rightarrow \quad \textbf{let } x = e \textbf{ in } \mathit{body}$

Efectos: Centrémonos en el primer caso. Tenemos que tener en cuenta todas las posibilidades de solapamiento con las construcciones Edén. En principio la λ-abstracción no puede ser la correspondiente a una abstracción de proceso, pues éstas no se aplican directamente, sino que aparecen en una expresión *processBy* f donde f es la λ-abstracción que representa a la abstracción de proceso.

Por otra parte, *body* puede ser una expresión cualquiera con definiciones de procesos e instanciaciones de los mismos. Si la variable x apareciera en *body* dentro de una abstracción de proceso no habría problema, puesto que estaríamos sustituyendo una variable por otra, lo que implicaría solamente que nos ahorraríamos la copia de x al hijo. Si aparece en una instanciación, puede que aparezca en la "parte izquierda" o en la "parte derecha", es decir, en la parte que evalúa el padre o en la que evalúa el hijo, no existiendo problema de nuevo en ninguno de los dos casos.

Si miramos el segundo caso, tampoco hay ningun problema, ya que la verdadera sustitución se llevará a cabo posteriormente mediante la transformación de inlining y ésta se lleva a cabo en ocasiones muy especiales, que más adelante estudiaremos. Esta versión de la β-reducción se limita a meter e en una clausura, lo cual no produce ningún efecto adverso.

En ningún caso hay problemas con el no determinismo. En la primera variante y es una variable, y por tanto no hay problema en que se replique. En la segunda, la expresión e no se replica.

4.1.2 Eliminación de código muerto

Transformación: La siguiente transformación se lleva a cabo cuando e no necesita el valor de x para evaluarse, es decir, la ligadura de x está muerta:

$$\textbf{let } x = e' \textbf{ in } e \quad \Rightarrow \quad e$$

Efectos: Podemos pensar que esta transformación es perjudicial puesto que en Edén hay procesos reactivos que hay que lanzar aunque sus salidas no se utilicen. Pero hay que tener en cuenta que gracias al lanzamiento impaciente de procesos, ya se ha forzado la evaluación de las salidas de estos procesos mediante un *case*, con lo que en realidad esas salidas se están utilizando y no se pueden considerar código muerto. En cuanto a aquellas expresiones con instanciación que no se han podido transformar a un *createProcessTL*, ya hemos dicho antes que solamente se harán los lanzamientos cuando se demanden, de forma que si no se están utilizando sus salidas, se podría eliminar tranquilamente la instanciación, ya que no se lanzaría de todas formas.

4.1.3 Reutilización de un constructor

Transformación: Tenemos dos casos:

- El constructor aparece ligado en un *let*:

$$\begin{array}{l}\textbf{let } v = C\ v_1 \ldots v_n \\ \textbf{in} \cdots C\ v_1 \ldots v_n \cdots\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{let } v = C\ v_1 \ldots v_n \\ \textbf{in} \cdots v \cdots\end{array}$$

- El constructor aparece ligado por una expresión *case*:

$$\begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \cdots \\ \quad C\ v_1 \ldots v_n \rightarrow \cdots C\ v_1 \ldots v_n \cdots \\ \quad \cdots\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \cdots \\ \quad C\ v_1 \ldots v_n \rightarrow \cdots v \cdots \\ \quad \cdots\end{array}$$

Efectos: No parece haber problemas en ninguno de los dos casos. Como máximo, en el primero de los casos podríamos tener que copiar una clausura más al hijo (la de v) si la expresión principal del *let* contuviese una instanciación donde la expresión $C\ v_1 \ldots v_n$ se evaluara en el hijo o una abstracción de proceso donde apareciese esa misma expresión.

No hay problema con el no determinismo porque el constructor ya es conocido y los argumentos de un constructor solamente pueden ser variables, con lo que v y $C\ v_1 \ldots v_n$ valen siempre lo mismo.

4.1.4 Reducción de un *case*

Transformación: Tenemos tres casos:

- El constructor del discriminante ya se conoce:

$$\begin{array}{l}\textbf{case } C\ v_1 \ldots v_n \textbf{ of} \\ \quad \cdots \\ \quad C\ x_1 \ldots x_n \rightarrow e \\ \quad \cdots\end{array} \quad \Rightarrow \quad e[v_i/x_i]$$

- Hay un doble escrutinio sobre una variable:

$$\begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \cdots \\ \quad C\ x_1 \ldots x_n \rightarrow \cdots \left(\begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \cdots \\ \quad C\ y_1 \ldots y_n \rightarrow e \\ \quad \cdots\end{array} \right) \cdots \\ \quad \cdots\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{case } v \textbf{ of} \\ \quad \cdots \\ \quad C\ x_1 \ldots x_n \rightarrow \cdots e[x_i/y_i] \cdots \\ \quad \cdots\end{array}$$

- Se conoce el constructor del escrutinio a partir de una ligadura *let*:

$$\begin{array}{l}\textbf{let } x = C\ x_1 \ldots x_n \\ \textbf{in} \\ \quad \ldots \left(\begin{array}{l} \textbf{case } x \textbf{ of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n \to e \\ \quad \ldots \end{array} \right) \ldots \end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{let } x = C\ x_1 \ldots x_n \\ \textbf{in} \quad \ldots\ e[x_i/y_i]\ \ldots \end{array}$$

Efectos: En ninguno de los tres casos hay problemas. Son casos muy parecidos a la β-reducción. Como se sustituyen variables por variables, no hay problema, y podríamos ahorrarnos la copia de algunas clausuras.

Podríamos pensar que en el segundo caso hay problemas con el no determinismo en el caso de que la v fuera no determinista. Si fuera una expresión cualquiera, no podríamos asegurar que en cada uno de los *case* fuera a escoger la misma rama (es decir, la rama se escogería de forma no determinista), con lo que la transformación no podría realizarse, pues se perderían alternativas. Pero el hecho de que sea una variable hace que, una vez escogida la rama del *case*, v se actualice con el constructor escogido y así aseguramos que en el siguiente *case* vamos a escoger de nuevo la rama de ese constructor y por tanto podemos fundirlas.

4.1.5 Eliminación de un *case*

Transformación: Afecta a expresiones *case* primitivas:

$$\textbf{case } v_1 \textbf{ of } v_2 \to e \quad \Rightarrow \quad e[v_1/v_2]$$

Efectos: No afecta negativamente al comportamiento de Edén, por las mismas razones anteriormente mencionadas para la sustitución de variables por variables.

4.1.6 Mezcla de expresiones *case*

Transformación: Consiste en lo siguiente:

$$\begin{array}{l}\textbf{case } x \textbf{ of} \\ \quad p_1 \to e_1 \\ \quad \ldots \\ \quad p_n \to e_n \\ \quad d \to \textbf{case } x \textbf{ of} \\ \qquad q_0 \to e'_0 \\ \qquad \ldots \\ \qquad q_m \to e'_m \end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{case } x \textbf{ of} \\ \quad p_1 \to e_1 \\ \quad \ldots \\ \quad p_n \to e_n \\ \quad q_0 \to e'_0[x/d] \\ \quad \ldots \\ \quad q_m \to e'_m[x/d] \end{array}$$

Efectos: Por las mismas razones que en la reducción de una construcción *case*, no afecta al comportamiento de Edén. Si en vez de una variable, x pudiera ser una expresión cualquiera, la semántica sería distinta en caso de ser no determinista dicha expresión, puesto que en la primera expresión se evaluaría dos veces la expresión produciendo valores potencialmente distintos y en la segunda solamente una vez.

Supongamos que los p_i junto con los q_j cubren todos los casos. En la primera expresión podría producirse un error, ya que la evaluación del segundo discriminante podría producir un valor que no encajara con ningún q_i pero sí con algún p_i, mientras que en tal caso en la segunda expresión no se produciría el error. Gracias a que tiene que ser una variable no tenemos este problema.

4.1.7 *Case* sobre una expresión de error

Transformación: $\textbf{case } (\textit{error } e) \textbf{ of} \ldots \quad \Rightarrow \quad \textit{error } e$

Efectos: No tiene ningún efecto especial con respecto a Edén.

4.1.8 Eliminación de la ligadura por defecto

Transformación: Consiste en sustituir la variable del caso por defecto por la del escrutinio:

$$\begin{array}{lcl} \textbf{case } v_1 \textbf{ of} & & \textbf{case } v_1 \textbf{ of} \\ \quad \ldots & \Rightarrow & \quad \ldots \\ \quad v_2 \rightarrow e & & \quad v_2 \rightarrow e[v_1/v_2] \end{array}$$

Efectos: Al tratarse de una sustituación de variables, no tiene ningún peligro para Edén, por las mismas razones que la β-reducción.

4.1.9 Eliminación de una alternativa muerta

Transformación: Si sabemos que la variable x no puede ligarse al constructor C_k entonces:

$$\begin{array}{lcl} \textbf{case } x \textbf{ of} & & \textbf{case } x \textbf{ of} \\ \quad C_1 \ldots \rightarrow e_1 & & \quad C_1 \ldots \rightarrow e_1 \\ \quad \ldots & & \quad \ldots \\ \quad C_k \ldots \rightarrow e_k & \Rightarrow & \quad C_{k-1} \ldots \rightarrow e_{k-1} \\ \quad \ldots & & \quad C_{k+1} \ldots \rightarrow e_{k+1} \\ \quad C_n \ldots \rightarrow e_n & & \quad \ldots \\ & & \quad C_n \ldots \rightarrow e_n \end{array}$$

La seguridad de que x no encaja con C_k proviene de que esta expresión, *expr*, aparece dentro de una expresión *case* externa de la siguiente manera:

$$\begin{array}{l} \textbf{case } x \textbf{ of} \\ \quad C_k \ldots \rightarrow \ldots \\ \quad\quad \_ \rightarrow \mathit{expr} \end{array}$$

donde *expr* sería la expresión arriba transformada. En esa alternativa estamos seguros de que x no tiene constructor C_k.

Efectos: No tiene ningún efecto peligroso sobre el comportamiento de Edén, ya que el primer *case* hace que la variable x se actualice con su valor y por tanto, aunque sea indeterminista, seguirá sin poder tener en *expr* como constructor a C_k.

4.1.10 Reflotamiento de un *let* fuera de una aplicación

Transformación: $(\textbf{let } v = e \textbf{ in } e')\ x \quad \Rightarrow \quad \textbf{let } v = e \textbf{ in } (e'\ x)$

Efectos: No tiene ningún efecto pernicioso. En ambos casos se creará primero la clausura para v y después se pasará a evaluar a forma normal débil e' para poder hacer después la aplicación. Es e' quien puede tirar de la evaluación de v, pero en ambos casos, dicha evaluación se llevará a cabo en el mismo sitio (en ese mismo proceso, o en uno hijo, dependiendo de donde aparezca v), ya que el aspecto de e' no cambia. El hecho de que al evaluar v se lancen otros procesos no afecta en nada.

4.1.11 Reflotamiento de un *let* fuera del lado derecho de otro *let*

Transformación: En principio, en GHC se opta por hacer esta transformación solamente cuando se expone una forma normal débil de cabeza o cuando se trata de un *let* estricto, pues entonces nos aseguramos de no crear clausuras inútilmente.

$$\begin{array}{lcl} \textbf{let} & & \textbf{let} \\ \quad x = \textbf{let } \mathit{bind} & & \quad\quad \mathit{bind} \\ \quad\quad\ \textbf{in } e & \Rightarrow & \textbf{in let } x = e \\ \textbf{in } b & & \quad\quad \textbf{in } b \end{array}$$

Para llevar a cabo esta transformación *bind* no debe depender de x.

Efectos: En nuestro caso, podría suceder que la expresión a la que se liga x se evaluara en el hijo (por ejemplo si b fuera en última instancia una instanciación *createProcessTL x is*). En ese caso, si sacamos *bind* hacia fuera, la clausura se guardará en el heap del padre y no en el del hijo, con la consiguiente copia de dicha clausura del padre al hijo. No se cambia trabajo de sitio porque es el hijo quien lleva a cabo su evaluación de todas formas, pero sí que se produce un cambio en el lugar donde se crean las clausuras y un aumento del trasiego entre procesos. Si esto sucede muy a menudo podría suponer un gran gasto de memoria en el padre y un gran trasiego entre él y sus hijos.

De todas formas, esta transformación se lleva a cabo solamente en dos casos: cuando el *let* es estricto, y cuando se expone una forma normal débil de cabeza. En el primer caso, si determinamos que *createProcessTL* no es estricta en sus argumentos, no se aplicaría la transformación, como ya hemos dicho antes, pues no podría entonces determinarse que la expresión es estricta en x. Por otra parte si e fuese una forma normal débil de cabeza (es decir, por ejemplo una abstracción de proceso), sí se haría la transformación con las consecuencias negativas antes descritas.

4.1.12 Flotamiento de un *let* fuera del escrutinio de un *case*

Transformación: Consiste en lo siguiente:

$$\begin{array}{l}\textbf{case } (\textbf{let } v = e \textbf{ in } e') \textbf{ of} \\ \quad alts\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{let } v = e \\ \textbf{in } (\textbf{case } e' \textbf{ of } alts)\end{array}$$

Efectos: No tiene ningún efecto pernicioso sobre las construcciones de Edén, ya que en ambos casos se creará la clausura para v y después se pasará a evaluar e', todo en el mismo proceso. La evaluación de e' puede tirar de la evaluación de v, pero en ambos casos se evaluará en el mismo sitio, ya sea en el mismo proceso padre (no se ve implicada en una instanciación de proceso, o sí lo está pero se ha de mandar por un canal) o en uno hijo (si en e' hay una instanciación de proceso donde v ha de evaluarse en el hijo), puesto que e' no cambia. El hecho de que la evaluación de e provoque el lanzamiento de otros procesos no afecta en nada.

4.1.13 Flotamiento de un *case* fuera de una aplicación

Transformación: Es la siguiente:

$$\left(\begin{array}{l}\textbf{case } e \textbf{ of} \\ \quad p_1 \to e_1 \\ \quad \cdots \\ \quad p_n \to e_n\end{array}\right) x \quad \Rightarrow \quad \begin{array}{l}\textbf{case } e \textbf{ of} \\ \quad p_1 \to e_1\ x \\ \quad \cdots \\ \quad p_n \to e_n\ x\end{array}$$

Efectos: No tiene ningún efecto peligroso para Edén, ya que en ambos casos hay que evaluar e primero para después elegir la alternativa y evaluar la correspondiente e_i llevando a cabo después la aplicación.

4.1.14 Reflotamiento de un *case* fuera del escrutinio de un *case*

Transformación: Consiste en lo siguiente:

$$\begin{array}{l}\textbf{case } \left(\begin{array}{l}\textbf{case } ec \textbf{ of} \\ \quad pc_1 \to ec_1 \\ \quad \cdots \\ \quad pc_m \to ec_m\end{array}\right) \textbf{ of} \\ \quad p_1 \to e_1 \\ \quad \cdots \\ \quad p_n \to e_n\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{case } ec \textbf{ of} \\ \quad pc_1 \to \textbf{case } ec_1 \textbf{ of} \\ \qquad\qquad p_1 \to e_1 \\ \qquad\qquad \cdots \\ \qquad\qquad p_n \to e_n \\ \quad \cdots \\ \quad pc_m \to \textbf{case } ec_m \textbf{ of} \\ \qquad\qquad p_1 \to e_1 \\ \qquad\qquad \cdots \\ \qquad\qquad p_n \to e_n\end{array}$$

Efectos: No tiene ningún efecto peligroso para Edén, ya que en ambos casos se sigue el mismo proceso de evaluación, independientemente de las construcciones relacionadas con los procesos: se evalúa e_c, después la ec_i correspondiente a la alternativa elegida y finalmente la e_j correspondiente. No se pierde ninguna alternativa, aunque alguna de las expresiones implicadas fuera no determinista.

4.1.15 Flotamiento de *case* fuera del lado derecho de un *let*

Transformación: Consiste en lo siguiente:

$$\begin{array}{l} \textbf{let } v = \textbf{case } e_v \textbf{ of} \\ \qquad \ldots \\ \qquad C_i\ x_{i1} \ldots x_{ik} \rightarrow e_i \\ \qquad \ldots \\ \textbf{in } e \end{array} \quad \Rightarrow \quad \begin{array}{l} \textbf{case } e_v \textbf{ of} \\ \qquad \ldots \\ \qquad C_i\ x_{i1} \ldots x_{ik} \rightarrow \textbf{let } v = e_i \textbf{ in } e \\ \qquad \ldots \end{array}$$

Esta transformación se lleva a cabo cuando e es estricta en v.

Efectos: Esta transformación puede cambiar trabajo entre procesos. Ya hemos dicho que todo lo que se encuentra "a la izquierda" en una instanciación de proceso, debe evaluarse en el hijo. Cuando en Full Edén tenemos e_1 # e_2, donde e_1 se evalúa en el hijo, en Core tenemos una variable ligada a la expresión que ha de evaluarse en el hijo (en la transformación v) como ya dijimos en la sección 3. Esto quiere decir, que si sacamos el *case* fuera del lado derecho de una ligadura de este tipo, estaremos cambiando el trabajo de evaluar la expresión del escrutinio, e_v, del hijo al padre, y el hijo sólo evaluará la alternativa que le corresponda e_i.

Podría parecer que también hay problemas en el caso en que la expresión *case* sea la correspondiente a la producida por el lanzamiento impaciente de procesos. Es decir, como dijimos ya en 3, estos *case* también se introducen en aquellas instanciaciones que no se encuentran a nivel superior, pero que sí hay que lanzar en el caso de ser demandada la variable a la que se encuentran ligadas. Si sacamos el *case* fuera de la ligadura estaríamos lanzando la instanciación sin que se nos haya pedido. Pero hay que tener en cuenta que la transformación sólo se hace si la expresión e es estricta en v, en cuyo caso la instanciación se iba a lanzar de todas formas, con lo que no hay problema.

Esta transformación también puede modificar el trasiego entre el padre y el hijo, y afectar al coste en espacio. Si v ha de evaluarse en un proceso hijo, antes de la transformación la clausura para v se crea en el padre y se copia al hijo, posiblemente sin evaluar. Tras la transformación, se evalúa parte de dicha clausura, con lo que el coste en espacio y el trasiego es distinto (puede ser mayor o menor)

Con respecto al no determinismo, si el padre es estricto en v (es decir, e es estricta en v), v ha de evaluarse en un proceso hijo (si en e hay una instanciación de proceso con v como variable libre) y e_v es una expresión no determinista, entonces podemos tener problemas. Este caso se da por ejemplo si e es de la forma $v+(f\ v)\#3$. Antes de la transformación, las dos apariciones de la v pueden tener valores diferentes, pero después de la transformación, una vez elegida la rama del *case*, las dos apariciones de la variable e tienen con seguridad el mismo valor. Con lo cual estamos perdiendo no determinismo.

4.1.16 Otros reflotamientos de *case*

Transformación: Tenemos tres:

Reflotamiento de *case* fuera de un *let*: Es equivalente a introducir un *let* dentro de las ramas de un *case*, que veremos más adelante.

Reflotamiento de *case* fuera de las alternativas de un *case*: Sólo es correcto para *case* con una única rama.

Reflotamiento de *case* fuera de una λ-abstracción: Se trata de la transformación llamada full laziness, que estudiaremos más adelante.

Efectos: Sólo da lugar a comentario el segundo caso, que no tiene ningún efecto. Las otras dos transformaciones se estudiarán más adelante.

4.1.17 Transformación de un *let* en un *case*

Transformación: Cuando la expresión e es estricta en v:

$$\textbf{let } v = e_v \textbf{ in } e \quad \Rightarrow \quad \textbf{case } e_v \textbf{ of } v \rightarrow e$$

Efectos: Esta transformación podría cambiar trabajo de sitio, por las mismas razones que en la transformación de flotamiento de un *case* fuera del lado derecho de un *let*. Si v ha de evaluarse en el hijo, por estar tirando de ella en la parte izquierda una instanciación de proceso en e, estaríamos cambiando trabajo del hijo al padre. Además, puede cambiar el coste en espacio y el trasiego entre padre e hijo, por razones similares a las anteriores.

Desde el punto de vista del no determinismo, sucede lo mismo que en el flotamiento de *case* fuera de un *let*.

4.1.18 Desencapsulamiento de un *let* en un *case*

Transformación: Se lleva a cabo cuando e es estricta en v y e_v es de un solo constructor:

$$\textbf{let } v = e_v \textbf{ in } e \quad \Rightarrow \quad \begin{array}{l}\textbf{case } e_v \textbf{ of} \\ \qquad C\ v_1 \ldots v_n \rightarrow \textbf{let } v = C\ v_1 \ldots v_n \textbf{ in } e\end{array}$$

Efectos: En este caso tenemos el mismo problema que antes, ya que e_v podría evaluarse en el hijo (aunque no puede ser una abstracción de proceso v podría ser un argumento para una función que devolviera como resultado la abstracción) antes de hacer la transformación, y en el padre después de ella.

Además, puede cambiar el coste en espacio y el trasiego entre padre e hijo, por razones similares a las anteriores.

4.1.19 Despliegue de constantes

Transformación: Consiste en desplegar la definición de los operadores primitivos. Por ejemplo:

$$eqInt\ v\ k \quad \Rightarrow \quad \begin{array}{l}\textbf{case } v \textbf{ of} \\ \qquad k \rightarrow True \\ \qquad \_ \rightarrow False\end{array}$$

donde k es una constante explícita.

Efectos: No tiene ningún peligro para Edén, pues no implica a ninguna construcción de Edén.

4.1.20 Eta expansión

Transformación: Consiste en añadir los argumentos que faltan hasta completar la aridad (donde la aridad no es el mayor número de argumentos que puede recibir sino el número de argumentos que se le pueden pasar a la función antes de llegar a una expresión *let* o *case*). En realidad cuando nos encontramos con un *case* sí se hace expansión si e es una variable y las e_i son λ-abstracciones:

$$\begin{array}{l}\textbf{case } e \textbf{ of} \\ \qquad p_1 \rightarrow e_1 \\ \qquad \ldots \\ \qquad p_n \rightarrow e_n\end{array} \quad \Rightarrow \quad \begin{array}{l}\lambda y.\textbf{case } e \textbf{ of} \\ \qquad p_1 \rightarrow e_1\ y \\ \qquad \ldots \\ \qquad p_n \rightarrow e_n\ y\end{array}$$

Efectos: No tiene efectos perjudiciales. En el caso de la *case* $-\eta$-expansión, si la e fuese cualquier expresión no determinista, y no solamente una variable, estaríamos introduciendo no determinismo. Al ser una variable, se actualiza con su valor la primera vez que se aplica la función. Supongamos que la expresión *case* está ligada a una variable z, que se aplica varias veces. Antes de hacer la transformación, la primera vez que se aplica z, se escoge la rama de forma no determinista y z se actualiza con el valor escogido.

Después de hacer la transformación, z es una λ-abstracción, por lo que cada vez que se aplique se evaluará la expresión *case* y por tanto e. Si e no fuera una variable, escogeríamos una rama cada vez, con lo que la función sería más indeterminista. Pero como e es una variable, después de la primera aplicación se ha actualizado y por tanto, aunque se evalúe la expresión *case* cada vez, se escogerá siempre la misma rama.

4.2 Hundimiento de construcciones *let*

4.2.1 Hundimiento

Transformación: Consiste en hundir un *let* dentro de otra expresión para que esté más cerca del punto de uso. Como ejemplo:

$$\begin{array}{l}\textbf{let } x = y + 1 \\ \textbf{in case } z \textbf{ of} \\ \quad [\,] \to x * x \\ \quad (p : ps) \to 1\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{case } z \textbf{ of} \\ \quad [\,] \to \textbf{let } x = y + 1 \\ \qquad \textbf{in } x * x \\ \quad (p : ps) \to 1\end{array}$$

Efectos: No tiene ningún efecto negativo. De hecho, al acercar las clausuras a su punto de uso, podríamos ahorrarnos espacio en el padre y trasiego de clausuras del padre al hijo, pues esta transformación tendría el efecto contrario a las transformaciones locales de reflotamiento.

4.2.2 Full laziness

Transformación: Esta transformación consiste en reflotar un *let* fuera de una λ-abstracción para que su evaluación se comparta entre todas las aplicaciones de la función. Sólo se puede llevar a cabo si expresión del lado derecho de la ligadura no depende de los argumentos de la función. Debajo aparece un ejemplo.

Efectos: Esta transformación tiene dos problemas. Desde el punto de vista del no determinismo el problema surge cuando la ligadura que sacamos fuera de la λ-abstracción es no determinista. Por ejemplo, si tenemos una función:

$$\begin{array}{l}\textbf{let} \\ \quad g = \lambda y.\textbf{let } x = e \\ \qquad\qquad \textbf{in } e' \\ \textbf{in } \ldots\end{array}$$

donde e es no determinista y e' necesita evaluar x, cada vez que se aplique la función g se creará una clausura para x, de forma que en cada aplicación de g, x puede tener un valor diferente. Esta regla la transformaría en:

$$\begin{array}{l}\textbf{let} \\ \quad x = e \\ \textbf{in let} \\ \quad g = \lambda y.e' \\ \quad \textbf{in } \ldots\end{array}$$

Ahora la clausura para x se crea solamente una vez, y la primera vez que g tira de x, ésta se actualiza con su valor ya para siempre, con lo que en sucesivas aplicaciones lo mantiene. Esto quiere decir que hemos perdido no determinismo, lo que altera el comportamiento del programa inicial.

Por otro lado, usando el mismo ejemplo, si e provoca el lanzamiento de algún proceso, entonces, antes de hacer la transformación se lanzarían esos procesos cada vez que se aplicara la función, mientras que después de hacerla, solamente se lanzarían la primera vez, actualizándose x con el valor obtenido. Por lo tanto, se estarían lanzando menos procesos de los especificados por el programador.

4.2.3 Reflotamiento de *case* fuera de una λ-abstracción

Transformación: Consiste en sacar un *case* fuera de una λ-abstracción, siempre que dicho *case* esté haciendo un escrutinio sobre una variable que no esté ligada por la λ-abstracción.

Efectos: Esta transformación no se lleva a cabo en el GHC, pero si se hiciera, podría cambiar trabajo del hijo al padre si la λ-abstracción correspondiese a una abstracción de proceso, ya que habilitaría el posterior flotamiento del case fuera de la ligadura, si éste se pudiera llevar a cabo. Mientras se quedase dentro de la ligadura, no pasaría nada. Como hemos dicho anteriormente que el posterior flotamiento no se llevaría a cabo si determinamos los operadores de instanciación como no estrictos, no causa problemas.

4.3 Inlining

Transformación: Consiste en sutituir una variable de una ligadura por su lado derecho en su punto de uso:

$$\textbf{let } v = e \textbf{ in } body \quad \Rightarrow \quad \textbf{let } v = e \textbf{ in } body[e/v]$$

Efectos: Esta transformación se hace con muchas restricciones que hacen que no tenga ninguna interacción con las construcciones Edén. Si la expresión a sustituir es una variable, un constructor o una λ-abstracción, se hace sin restricciones (sí hay algunas que tienen que ver con el tamaño del código). En este caso no hay ningún problema desde el punto de vista del no determinismo por tratarse de variables o formas normales débiles de cabeza (λ-abstracciones y constructores con variables). Si la expresión a sustituir no es ninguna de las anteriores, se hace un análisis de uso. Si se usa a lo sumo una vez se puede hacer la sustitución, y si no lo podemos asegurar no se hace. Obsérvese que si se permitiera sustituir una expresión no determinista en varios sitios, estaríamos aumentando el no determinismo. Gracias a las restricciones esto no sucede.

Por otra parte, dentro de las restricciones, el inlining reduciría el trasiego de clausuras del padre al hijo.

4.4 Transformación de argumentos estáticos y lambda lifting

4.4.1 Transformación de argumentos estáticos

Transformación: Esta transformación se aplica sobre definiciones recursivas. Si para un argumento de la función, todas las llamadas recursivas actúan sobre el mismo argumento sin modificar, se dice que dicho argumento es estático. Entonces podemos "sacar factor común" definiendo una nueva función que contiene al argumento estático como variable libre y se comporta como la función original. Por ejemplo:

$$\begin{array}{l} foldr\ f\ z\ l = \\ \quad \textbf{case } l \textbf{ of} \\ \qquad [\,] \rightarrow z \\ \qquad (a : as) \rightarrow \textbf{let } v = foldr\ f\ z\ as \\ \qquad\qquad \textbf{in } f\ a\ v \end{array} \quad \Rightarrow \quad \begin{array}{l} foldr\ f\ z\ l = \\ \quad \textbf{let } foldr'\ l = \\ \qquad \textbf{case } l \textbf{ of} \\ \qquad\quad [\,] \rightarrow z \\ \qquad\quad (a : as) \rightarrow \textbf{let } v = foldr\ f\ z\ as \\ \qquad\qquad\quad \textbf{in } f\ a\ v \\ \quad \textbf{in } foldr'\ l \end{array}$$

Efectos: Esta transformación tiene efectos sobre el no determinismo análogos al de la transformación full laziness, es decir, provoca la pérdida de no determinismo. El problema surge cuando la parte no estática de la función, es decir la aplicación parcial de la función a sus argumentos estáticos (que para poder aplicar la transformación deben ser explícitos) no es una forma normal débil de cabeza. Si lo es no hay problema porque la nueva ligadura es también a su vez una λ-abstracción y por tanto el comportamiento de ésta y la aplicación parcial de la función original a sus argumentos estáticos es el mismo. Sin embargo, si para llegar a la λ-abstracción (es decir, a la forma normal débil de cabeza) hace falta previamente evaluar algo no determinista (es decir, la función se genera de forma no determinista), la función original y la nueva ligadura ya no tendrán el mismo comportamiento.

Veamos un ejemplo. Sea la función recursiva

$$
\begin{array}{l}
f\ x = \textbf{let} \\
\quad g = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [] \to x \\
\qquad\qquad a : as \to f\ x\ as \\
\quad h = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [] \to 2 * x \\
\qquad\qquad a : as \to f\ x\ [] \\
\textbf{in} \\
\quad hd(merge\ \#\ [[g],[h]])
\end{array}
$$

donde se ve claramente que x es un parámetro que se mantiene en las llamadas recursivas. En esta función, en cada llamada recursiva se escoge una función entre g y h de forma no determinista. La evaluación de una aplicación de esta función puede terminar aunque la lista sea infinita; basta con que en algún momento se escoja h como valor de la función. La función, en caso de que termine, devuelve x o el doble de x. Si la transformamos en

$$
\begin{array}{l}
f\ x = \textbf{let} \\
\quad f' = \textbf{let} \\
\qquad g = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [] \to x \\
\qquad\qquad a : as \to f'\ as \\
\qquad h = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [] \to 2 * x \\
\qquad\qquad a : as \to f'\ [] \\
\quad\ \textbf{in} \\
\qquad hd(merge\ \#\ [[g],[h]]) \\
\textbf{in}\ f'
\end{array}
$$

perdemos el no determinismo y además la posibilidad de terminación cuando la lista es infinita. Ahora en la primera llamada, f' escogerá entre g y h actualizándose con el valor escogido, fijo ya para todas las llamadas recursivas. Si escoge a g por primera vez y la lista es infinita, con toda seguridad no termina, y si escoge a h va a devolver el doble de x, no pudiendo nunca devolver x.

4.4.2 Lambda lifting

Transformación: Elimina las variables libres de una función pasándolas como argumentos, después de lo cual se pueden sacar al nivel superior. Por ejemplo:

$$
\begin{array}{l}
f\ x\ z = \textbf{let}\ g\ y = y * x \\
\qquad\qquad \textbf{in case}\ x\ \textbf{of} \\
\qquad\qquad\quad 1 \to g\ z \\
\qquad\qquad\quad n \to \textbf{let}\ x' = x - 1 \\
\qquad\qquad\qquad\qquad z' = g\ z \\
\qquad\qquad\qquad \textbf{in}\ f\ x'\ z'
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
g\ x\ y = y * x \\
f\ x\ z = \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad 1 \to g\ x\ z \\
\qquad\qquad n \to \textbf{let}\ x' = x - 1 \\
\qquad\qquad\qquad\qquad z' = g\ x\ z \\
\qquad\qquad\qquad \textbf{in}\ f\ x'\ z'
\end{array}
$$

Efectos: Esta transformación saca ligaduras hacia fuera, con lo que puede aumentar la ocupación del heap en el padre, al igual que en todas las demás transformaciones en las que se sacan ligaduras hacia fuera desde expresiones que se evalúan en el hijo. Además se trata de clausuras de mayor tamaño, puesto que se están añadiendo parámetros. Sin embargo, no se modifica el trasiego de clausuras entre padre e hijo, porque las clausuras a nivel superior son código estático que está disponible para todos los procesos y que por tanto no se copian de uno a otro. Solamente nos ahorramos la creación de la clausura en el hijo.

4.5 Especialización

Transformación: Esta transformación convierte aplicaciones parciales a tipos y diccionarios en ligaduras *let*. Más abajo vemos un ejemplo.

Efectos: Esta transformación también produce pérdida de no determinismo, con todas las consecuencias que se deriven de ello. Veamos un ejemplo. Sea la siguiente función

$$\begin{array}{l} g = \Lambda ty.\lambda dict.\lambda y. \\ \quad \textbf{let } f = \Lambda ty.\lambda dict.e \\ \quad \textbf{in } f\ ty\ dict\ (f\ ty\ dict\ y) \end{array}$$

donde e es una expresión que genera de forma no determinista una función. Esto quiere decir que las dos aplicaciones parciales $f\ ty\ dict$ que aparecen en el cuerpo de g denotan funciones potencialmente distintas. Si aplicamos la especialización obtendríamos

$$\begin{array}{l} g = \Lambda ty.\lambda dict.\lambda y. \\ \quad \textbf{let } f = \Lambda ty.\lambda dict.e \\ \quad \textbf{in} \\ \quad\quad \textbf{let } f' = f\ ty\ dict \\ \quad\quad \textbf{in } f'\ (f'\ y) \end{array}$$

Ahora, la primera vez que se evalúe f', se actualizará con la forma normal débil de cabeza obtenida, con lo que las dos apariciones de f' sí denotan necesariamente la misma función. Esto quiere decir que hemos perdido no determinismo.

5 Trabajo futuro

Hemos de tener en cuenta que el compilador GHC es una herramienta siempre cambiante, que está en continuo proceso de mejora. En los últimos tiempos se han introducido nuevos análisis y transformaciones, que se podrían ver incrementados con el tiempo. Es necesaria por tanto una continua labor de estudio que determine cómo cada una de ellas afecta a Edén en los distintos aspectos ya mencionados.

Pueden existir además variantes de las transformaciones ya existentes u otras nuevas que optimicen las características propias de Edén. Dichas transformaciones podrían llevarse a cabo a nivel del lenguaje CoreEdén, ya que en él se muestran de forma explícita las construcciones de los procesos. En [PPP99] se presentaron ya algunas ideas.

Finalmente, consideramos importante estudiar más a fondo la semántica del no determinismo, y ver cual es el alcance completo de las transformaciones que modifican el comportamiento no determinista de los programas Edén. A tal efecto pretendemos desarrollar una batería de ejemplos en los que se ponga de manifiesto claramente la modificación del comportamiento.

Agradecimientos

Agradecemos a Ricardo Peña y Fernando Rubio sus comentarios y correcciones, que han contribuido a la mejora de este informe. Conocer el aspecto que las construcciones de Edén tienen una vez CoreEdén se traduce de nuevo a Core en el proceso de compilación, ha sido fundamental para desarrollar las ideas aquí expuestas.

Referencias

[BLOMP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report, Bericht 96-10, revised version, Philipps-Universität Marburg, Germany, 1996.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, 1997.

[JHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology, Keele, DTI/SERC*, pages 249–257, 1993.

[JM99] S. L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, September 1999.

[Jon96] S. L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. *6th European Symposium on Programming ESOP'96, LNCS 1058*, April 1996.

[JPS96] S. L. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. *International Conference on Functional Programming ICFP'96*, May 1996.

[JS98] S. L. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47*, September 1998.

[KOMP98] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. LNCS 1595, Springer-Verlag*, pages 1–16, 1998.

[KPS00] U. Klusik, R. Peña, and C. Segura. Bypassing of channels in Eden. In *Trends in Functional Programming*, pages 2–10. Intellect, 2000. Presented at 1st Scottish Functional Programming Workshop, SFP'99. To appear.

[PPP99] C. Pareja, I. Pita, and P. Palao. El efecto de las optimizaciones secuenciales sobre Edén. Technical Report 92.99, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 1999.

[Rub99] F. Rubio. Programación funcional paralela eficiente. Trabajo de Tercer Ciclo, Departamento de Sistemas Informáticos y Programación (Universidad Complutense de Madrid), 1999.

[San95] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Department of Computing Science, 1995.

[THJP96] P. W. Trinder, K. Hammond, J. S. Mattson Jr., and A. S. Partridge. GUM: a portable parallel implementation of Haskell. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, USA*. ACM Press, May 1996.