PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Fusión en presencia de acumuladores

Mónica Martínez Amarante

2010

Fusión en presencia de acumuladores
Martínez Amarante, Mónica
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 10-14
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, diciembre de 2010

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Trabajo de tesis para la obtención del grado de Magister en Informática de la Universidad de la República en el programa de Maestría del área Informática del Pedeciba

Fusión en presencia de acumuladores

Mónica Martínez Amarante
mmartine@fing.edu.uy

Orientador de Tesis
Dr. Alberto Pardo

Supervisor de Maestría
Dr. Alberto Pardo

Montevideo, Uruguay
Diciembre 2010

Resumen

En programación funcional es común escribir los programas como la composición de funciones relativamente más sencillas. Esto tiene todas las ventajas asociadas a un estilo de programación modular. Sin embargo los programas obtenidos pueden ver afectada su eficiencia debido a la generación de las estructuras de datos intermedias que son utilizadas como comunicación entre las funciones involucradas en las composiciones.

Existe un conjunto de técnicas de transformación de programas, conocidas como *fusión* o *deforestación*, que apuntan a la eliminación de estas estructuras intermedias y permiten obtener definiciones más eficientes, equivalentes a las originales. El objetivo es posibilitar que el programador pueda continuar escribiendo sus programas de forma composicional y generar un código equivalente, más eficiente, que sea el que efectivamente se ejecute.

En este trabajo se considera el caso particular donde las funciones involucradas son tales que construyen sus resultados utilizando un parámetro adicional de acumulación. Las técnicas de fusión clásicas no suelen ser efectivas en el caso de estas funciones ya que no consiguen eliminar las estructuras intermedias cuando las mismas son generadas en los parámetros de acumulación.

Como primera aproximación a una solución del problema se realiza una revisión del operador *afold* propuesto por Pardo, el cual permite la definición de funciones con acumuladores por recursión estructural sobre tipos polinomiales. Se presenta una modificación al operador que permite ampliar el conjunto de funciones capaz de representar.

Posteriormente la tesis se concentra en la propuesta de un nuevo método de fusión basado en el método de "Short-Cut Fusion" que contempla el caso en que la función productora genera sus resultados utilizando parámetros de acumulación. Se muestra la aplicación del nuevo método a programas Haskell y se realiza una serie de pruebas que permiten comparar la performance desde el punto de vista del tiempo de ejecución y el espacio de memoria requerido entre las versiones originales y transformadas de un conjunto de programas ejemplo.

Índice general

Índice de figuras

Índice de cuadros

1 Introducción

Contenido

1.1. Definición del problema

En programación funcional, la composición de funciones es una herramienta fundamental al momento de escribir los programas ya que mediante su utilización es posible combinar pequeños programas para obtener la solución a un problema mayor. Esto permite obtener beneficios como: aumentar la legibilidad de los programas, facilitar su mantenimiento, además de incentivar el uso de bibliotecas predefinidas [Hug89]. Sin embargo el uso sistemático de la composición de funciones muchas veces también provoca la obtención de programas ineficientes.

Dada una composición de funciones $c \circ p$ donde p es una función que produce una estructura de datos a ser consumida por la función c, dicha estructura cumple únicamente un rol de comunicación de datos entre estas dos funciones. El manejo de la estructura intermedia requiere que sus nodos sean alojados en memoria, recorridos y finalmente descartados, requiriendo por lo tanto tiempo adicional de procesamiento y espacio de memoria para alojar sus nodos y frecuentes llamadas al "garbage collector"para recolectar los nodos liberados. Esto genera un costo que muchas veces puede ser elevado tanto en tiempo de ejecución como en espacio de memoria requerido.

Por ejemplo, consideremos el siguiente programa que dada una lista de enteros y una condición p sobre los enteros, devuelve la suma de los elementos del mayor prefijo tal que se cumple que todos sus elementos satisfacen la condición p. En particular, dada la lista $[7, 21, 49, 6, 14]$ y la condición ser múltiplo de 7 (denotada por $\dot{7}$), el prefijo que cumple la condición es $[7, 21, 49]$, por lo que el resultado que se obtiene de aplicar la función es 77.

Una primera solución en la cual se pone énfasis en la claridad podría ser:

$$
\begin{aligned}
&\mathit{sumTakeW}\ p = \mathit{sum} \circ \mathit{takeWhile}\ p \\
&\\
&\mathit{takeWhile}\ p\ [\,] \qquad = [\,] \\
&\mathit{takeWhile}\ p\ (a : as) = \textbf{if}\ p\ a\ \textbf{then}\ a : \mathit{takeWhile}\ p\ as\ \textbf{else}\ [\,] \\
&\\
&\mathit{sum}\ [\,] \qquad = 0 \\
&\mathit{sum}\ (a : as) = a + \mathit{sum}\ as
\end{aligned}
$$

donde el problema original se descompone en dos subproblemas cada uno de ellos resuelto por una función diferente: *takeWhile p* y *sum* respectivamente. En primer lugar la función *takeWhile p* construye una lista que corresponde al mayor prefijo de la lista de entrada tal que sus elementos cumplen la condición p. A continuación la función *sum* toma dicho prefijo y calcula la suma de los elementos que lo componen. El ejemplo se puede representar gráficamente como:

$$[7, 21, 49, 6, 14] \xrightarrow{\mathit{takeWhile}\ \dot{7}} [7, 21, 49] \xrightarrow{\mathit{sum}} 77$$

donde la flecha representa la aplicación de la función que tiene por nombre. Notar que la lista intermedia se usa simplemente como comunicación de datos entre las dos funciones y que luego es descartada.

Si la prioridad es la eficiencia, es posible pensar otras soluciones para este problema donde no se genere la lista intermedia. Como primer ejemplo es posible considerar la siguiente solución:

$$
\begin{aligned}
&sumTakeW\ p\ [\,] &&= 0\\
&sumTakeW\ p\ (a : as) &&= \textbf{if}\ p\ a\ \textbf{then}\ a + sumTakeW\ p\ as\ \textbf{else}\ 0
\end{aligned}
$$

donde se define *sumTakeW* directamente en una única función que a medida que recorre la lista de entrada determina si el elemento cumple la condición especificada y en caso afirmativo lo considera para la suma final; en caso contrario finaliza la recorrida de la lista.

Otra posible solución es:

$$
\begin{aligned}
&sumTakeW\ p\ xs = sumTakeW'\ p\ 0\ xs\\
&\quad \textbf{where}\\
&\qquad sumTakeW'\ p\ ac\ [\,] &&= ac\\
&\qquad sumTakeW'\ p\ ac\ (a : as) &&= \textbf{if}\ p\ a\ \textbf{then}\ sumTakeW'\ p\ (a + ac)\ as\\
&\qquad\qquad\qquad\qquad &&\quad\ \textbf{else}\ ac
\end{aligned}
$$

En este caso la función *sumTakeW* se define en términos de otra, *sumTakeW′*, que utiliza un parámetro adicional, un acumulador, donde se va generando el resultado final de la función.

Los dos casos anteriores se corresponden con distintas soluciones donde efectivamente se evita la construcción de la estructura intermedia. Esto queda claro si se observa que en las nuevas soluciones no se utiliza ninguna aplicación de los constructores de lista. En general, las distintas versiones que se pueden definir para una función presentan diferentes características desde el punto de vista de las mejoras que permite obtener cada una, tanto en lo que respecta a tiempo de ejecución o espacio consumido, así como las posibilidades de obtener dichas versiones en forma automática. Si bien estas soluciones cumplen con el objetivo de no construir la estructura intermedia presentan otros inconvenientes. Por ejemplo, reutilizan menos código (no se basan en otras funciones preexistentes ya testeadas que puedan estar en bibliotecas), es más difícil comprender qué hacen, y no siempre son tan naturales de escribir.

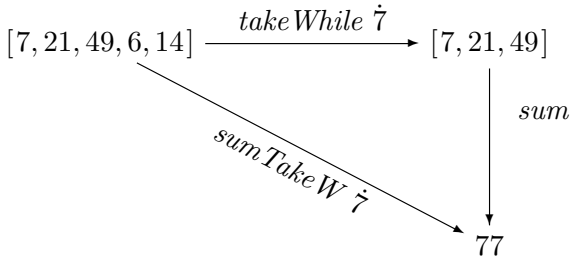
Lo deseable es que el programador escriba los programas de forma modular pero que los programas que efectivamente se ejecuten sean una versión transformada de los mismos de mayor eficiencia. Para esto es necesario que los compiladores sean capaces de realizar esta transformación de manera automática. De esta forma el programador obtendría "lo mejor de dos mundos": claridad en el código y mejor eficiencia en la ejecución.

1.1.1. Fusión

Existe una línea de investigación que se dedica al estudio de cómo transformar la composición de programas funcionales de forma de obtener un programa equivalente pero sin la creación de estructuras intermedias. Debido a que en los lenguajes funcionales estas estructuras en general son arborescentes, se dice que la técnica es de "deforestación" [Wad90]. La eliminación de la estructura intermedia se logra reemplazando las funciones involucradas en la composición por una equivalente que combine sus códigos. Es por esta razón que también se dice que las técnicas son de "fusión". La característica ecuacional de los programas funcionales puros hace que estos sean adecuados para ser transformados aplicando determinadas leyes que garantizan la correctitud de la transformación.

Considerando el ejemplo antes presentado ($sum \circ takeWhile\ p$), la idea que se encuentra por

detrás de la fusión puede representarse gráficamente como:



La composición de las funciones *takeWhile p* y *sum* se reemplaza por una nueva definición de *sumTakeW p* en la que se combinan los códigos de dichas funciones y donde no se construye la lista intermedia.

Una forma de derivar la definición fusionada de *sumTakeW* es a través del método de "plegado/desplegado " [BD77] que consiste en hacer análisis de casos:

Caso $[\,]:$
$$\begin{aligned} & sumTakeW\ p\ [\,] \\ = & \quad \{\ \text{Def. de } sumTakeW\ \} \\ & sum\ (takeWhile\ p\ [\,]) \\ = & \quad \{\ \text{Def. de } takeWhile\ \} \\ & sum\ [\,] \\ = & \quad \{\ \text{Def. de } sum\ \} \\ & 0 \end{aligned}$$

Caso $(a : as)$**:**
$$\begin{aligned} & sumTakeW\ p\ (a : as) \\ = & \quad \{\ \text{Def. de } sumTakeW\ \} \\ & sum\ (takeWhile\ p\ (a : as)) \\ = & \quad \{\ \text{Def. de } takeWhile\ \} \\ & sum\ (\textbf{if}\ p\ a\ \textbf{then}\ a : takeWhile\ p\ as\ \textbf{else}\ [\,]) \\ = & \quad \{\ sum \text{ es estricta}\ \} \\ & \textbf{if}\ p\ a\ \textbf{then}\ sum\ (a : takeWhile\ p\ as)\ \textbf{else}\ sum\ [\,] \\ = & \quad \{\ \text{Def. de } sum\ \} \\ & \textbf{if}\ p\ a\ \textbf{then}\ a + sum\ (takeWhile\ p\ as)\ \textbf{else}\ 0 \\ = & \quad \{\ \text{Def. de } sumTakeW\ \} \\ & \textbf{if}\ p\ a\ \textbf{then}\ a + sumTakeW\ p\ as\ \textbf{else}\ 0 \end{aligned}$$

En resumen, se obtiene la siguiente expresión para la función *sumTakeW*:

$$\begin{aligned} sumTakeW\ p\ [\,] &= 0 \\ sumTakeW\ p\ (a : as) &= \textbf{if}\ p\ a\ \textbf{then}\ a + sumTakeW\ p\ as \\ &\quad\ \ \textbf{else}\ 0 \end{aligned}$$

la cual corresponde a la solución del problema presentada como primera alternativa que evita la construcción de la estructura intermedia.

Existen hoy en día varios métodos muy conocidos que permiten realizar con éxito este tipo de transformaciones para un amplio conjunto de funciones. Estos métodos presentan enfoques diferentes poniendo énfasis en distintos aspectos de las funciones involucradas y en consecuencia pueden obtener resultados con diferentes características.

Uno de los métodos de fusión más conocidos es "Short-Cut Fusion" [GLJ93]. La idea esencial de esta técnica es identificar los lugares de la función productora donde efectivamente se construye la estructura intermedia y sustituir las ocurrencias de los constructores por las operaciones que realiza la función consumidora sobre cada uno de ellos.

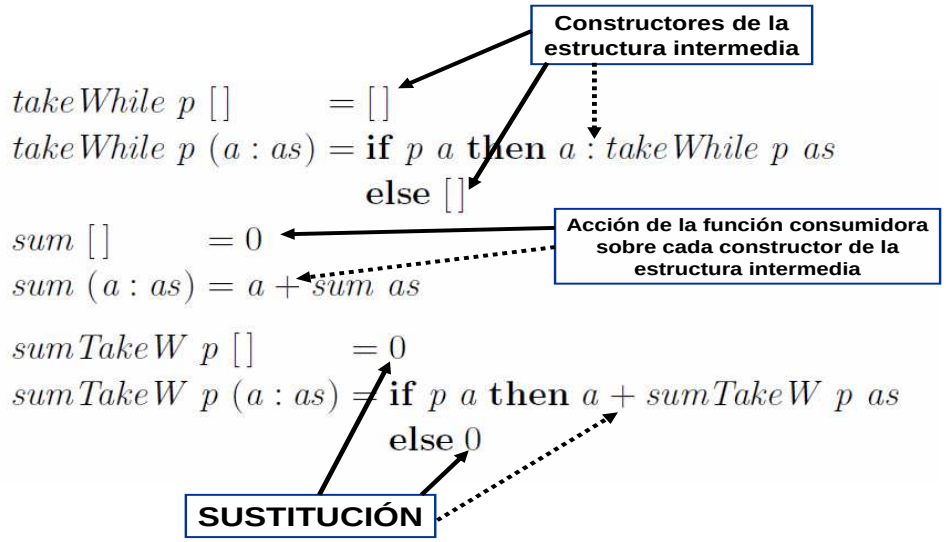


Figura 1.1: Funcionamiento de la técnica de "Short-Cut Fusion"

En la Figura 1.1 se muestra el funcionamiento de esta técnica de fusión para el ejemplo, donde un tipo de flecha diferente se asocia a cada constructor del tipo lista. Primero se identifican los constructores de la lista intermedia en el cuerpo de la definición de la función productora *takeWhile*. A continuación se observa que la función consumidora *sum* para el caso de la lista vacía devuelve el valor 0 y para el caso de una lista no vacía devuelve la suma del primer elemento con la suma del resto de los elementos de la lista. Finalmente la función resultante de la aplicación de fusión se construye sustituyendo en el cuerpo de la función productora las ocurrencias del constructor de lista vacía por el valor 0 y el constructor (:) por la función (+).

Esta técnica de fusión es genérica en el sentido de que puede ser utilizada de manera uniforme para una amplia clase de funciones y tipos de datos.

A pesar de la existencia de diferentes métodos que permiten realizar con éxito la fusión de muchos tipos de composiciones, todavía hay una clase de funciones para las que no ha sido posible encontrar técnicas eficaces para la fusión en forma general. Entre estos casos se encuentran las composiciones que son el objeto de estudio de esta tesis en las que la función productora es una acumulación, entendiendo por tal a una función que construye su resultado en parámetros adicionales llamados parámetros de acumulación.

1.1.2. Fusión con Acumulaciones

El uso de acumulaciones es ampliamente conocido en el contexto de programación funcional y muchas veces son fuentes de mejora en la eficiencia de los programas [Bir84, Bir98]. El ejemplo típico es la versión acumulativa de la función *reverse*. La función *reverse*, que invierte los elementos de una lista, tiene la siguiente definición:

$$
\begin{array}{ll}
reverse & :: [a] \to [a] \\
reverse\ [\,] & = [\,] \\
reverse\ (a : as) & = reverse\ as +\!\!+ [a]
\end{array}
$$

donde $(+\!\!+)$ corresponde a la función que concatena dos listas. El tiempo de ejecución de esta función es cuadrático en el largo de la lista debido a la presencia de la función de concatenación $(+\!\!+)$. Sin embargo es posible obtener una función cuyo tiempo de ejecución es lineal si se usa un acumulador que vaya manteniendo el resultado intermedio.

$$
\begin{array}{l}
reverse\ xs = areverse\ xs\ [\,] \\
\\
\begin{array}{ll}
areverse & :: [a] \to [a] \to [a] \\
areverse\ [\,]\ xs & = xs \\
areverse\ (a : as)\ xs & = areverse\ as\ (a : xs)
\end{array}
\end{array}
$$

La salida se va generando en el acumulador de la función para ser devuelto al llegar al final de la lista de entrada.

Análogamente, es posible definir una función acumulativa, similar a la función *takeWhile*, en la que se obtiene el mismo prefijo que con *takeWhile* pero en el orden inverso:

$$
\begin{array}{l}
takeWhile\ p\ xs = atakeWhile\ p\ xs\ [\,] \\
\\
\begin{array}{ll}
atakeWhile & :: (a \to Bool) \to [a] \to [a] \to [a] \\
atakeWhile\ p\ [\,]\ ys & = ys \\
atakeWhile\ p\ (a : as)\ ys & = \textbf{if}\ p\ a\ \textbf{then}\ atakeWhile\ p\ as\ (a : ys)\ \textbf{else}\ ys
\end{array}
\end{array}
$$

Cuando la función productora es una acumulación las técnicas clásicas de deforestación [Wad90] no consiguen alcanzar los parámetros de acumulación de dicha función y en consecuencia la generación de la estructura intermedia no consigue ser totalmente eliminada. Esto se puede visualizar, por ejemplo, mediante un análisis de casos similar al realizado anteriormente para *sumTakeW*, pero ahora considerando *atakeWhile* en lugar de *takeWhile*. Debido a que la función suma $(+)$ es asociativa y conmutativa, el resultado de sumar el prefijo obtenido con *takeWhile* es el mismo que se resulta al sumar su inverso (que es el obtenido con *atakeWhile*).

$$
\begin{array}{lll}
\textbf{Caso}\ [\,]: & sumTakeW\ p\ [\,]\ ys & \\
= & \{\ \text{Def. de } sumTakeW\ \} & \\
& sum\ (atakeWhile\ p\ [\,]\ ys) & \\
= & \{\ \text{Def. de } atakeWhile\ \} & \\
& sum\ ys &
\end{array}
$$

$$
\begin{aligned}
\textbf{Caso}\quad (a : as) : \quad & sumTakeW\ p\ (a : as)\ ys \\
= \quad & \{\ \text{Def. de } sumTakeW\ \} \\
& sum\ (atakeWhile\ p\ (a : as)\ ys) \\
= \quad & \{\ \text{Def. de } atakeWhile\ \} \\
& sum\ (\textbf{if}\ p\ a\ \textbf{then}\ atakeWhile\ p\ as\ (a : ys)\ \textbf{else}\ ys) \\
= \quad & \\
& \textbf{if}\ p\ a\ \textbf{then}\ sum\ (atakeWhile\ p\ as\ (a : ys))\ \textbf{else}\ sum\ ys \\
= \quad & \{\ \text{Def de } sumTakeW\ \} \\
& \textbf{if}\ p\ a\ \textbf{then}\ sumTakeW\ p\ as\ (a : ys)\ \textbf{else}\ sum\ ys
\end{aligned}
$$

La definición obtenida no es satisfactoria en el sentido de que se continúa construyendo una lista intermedia en el acumulador de las llamadas recursivas, la cual debe ser evaluada por la función *sum* al alcanzar la lista vacía o el primer elemento que no cumpla la condición p.

La aplicación de la técnica de "Short-Cut Fusion" en el caso de que la función productora sea una acumulación requiere de ciertas consideraciones particulares debido a la existencia del parámetro de acumulación y el hecho de que el mismo puede contener parte de la estructura intermedia. Por ejemplo, en la versión acumulativa de *takeWhile* el parámetro de acumulación (ys) directamente aloja parte de la estructura intermedia que luego debe ser alcanzada por la función consumidora. La construcción de toda o parte de la estructura intermedia en el acumulador es en muchos casos ignorada por los métodos de fusión (como ocurrió en la derivación anterior) y requiere de consideraciones particulares por parte de los métodos para alcanzar sus constructores.

Una posible solución para $sumTakeW$ que considera en forma explícita el acumulador de la función productora es:

$$
\begin{aligned}
& sumTakeW\ p\ as && = sumTakeW'\ p\ as\ 0 \\
& sumTakeW'\ p\ [\,]\ n && = n \\
& sumTakeW'\ p\ (a : as)\ n && = \textbf{if}\ p\ a\ \textbf{then}\ sumTakeW'\ p\ as\ (a + n)\ \textbf{else}\ n
\end{aligned}
$$

Este trabajo se propone avanzar en la solución al problema de la fusión cuando la función productora es una acumulación poniendo énfasis en una técnica basada en "Short-Cut Fusion". Aplicando esta nueva técnica se obtendrá esta definición acumulativa para $sumTakeW$.

1.2. Contribución

La principal contribución de este trabajo consiste en la propuesta de una nueva regla de fusión basada en el método de "Short-Cut Fusion" [GLJ93] que contempla el caso en que la función productora genera sus resultados utilizando parámetros de acumulación, situación que no es considerada en la propuesta original del método.

Junto con esta propuesta se llevan a cabo una serie de pruebas con un conjunto de programas

tomados como ejemplos. Se realizan comparaciones entre los programas originales y los transformados tanto en términos de tiempo de ejecución como de consumo de memoria. Esto permite observar el impacto real del uso de la regla de fusión propuesta en ejemplos concretos ya que en muchos casos la eliminación de la estructura intermedia no es una garantía en sí misma de que se logren mejores resultados. Las programas utilizados fueron escritos en Haskell y compilados con GHC (Glasgow Haskell Compiler) versión 6.8.2 [GHC].

En forma adicional se realiza un estudio del operador *afold*, propuesto en [Par03], el cual permite la definición de acumulaciones por recursión estructural. Se realiza una pequeña modificación al mismo, eliminando ciertas restricciones en su definición, lo que permite ampliar el conjunto de las acumulaciones representables en forma directa. La mayoría de las leyes de fusión asociadas a este operador continúan siendo válidas en la versión modificada y como consecuencia es posible resolver la fusión de un conjunto mayor de composiciones en forma efectiva. Tomando como inspiración el trabajo realizado en [Gib06] se usan estas leyes de fusión para mostrar un uso no tradicional de las mismas en el contexto de *fisión de programas*, esto es, el proceso inverso a la fusión donde se descompone una función en la composición de funciones más sencillas.

Parte de los resultados de esta tesis fueron publicados en [MP09].

1.3. Organización del documento

Este documento está organizado en cinco capítulos (incluido éste) y dos anexos cuyos contenidos se describen a continuación.

El Capítulo 2 presenta los conceptos y herramientas formales necesarias como base para el desarrollo del trabajo. Se formalizan conceptos tales como tipos de datos inductivos, operadores recursivos y leyes de fusión, entre otros.

En el Capítulo 3 se hace una revisión del operador *afold* y sus leyes de fusión. Se define una extensión al mismo, *afoldE*, que permite capturar el comportamiento de un nuevo grupo de acumulaciones. Se muestra que el operador *foldl*, de amplio uso en programación funcional, es un caso particular del operador *afold*. Finalmente, usando las leyes asociadas al operador *afold* se desarrolla un ejemplo de fisión (lo inverso a fusión) para un ejemplo basado en lo presentado por Gibbons [Gib06].

En el Capítulo 4 se presenta la propuesta de la ley de "Short-Cut Fusion" acumulativa siguiendo el enfoque de la ley de "Short-Cut Fusion" estándar. Se realiza la definición de la nueva ley, se da su justificación formal y se muestra una serie de ejemplos que permiten visualizar su uso para distintos tipos de datos. Finalmente, se presentan los resultados obtenidos en las pruebas comparativas entre las versiones originales y transformadas (resultantes de aplicar la ley propuesta) para un conjunto de programas desde el punto de vista del tiempo y el espacio de memoria requerido para la ejecución de los mismos.

En el Capítulo 5 se presenta un breve resumen del trabajo realizado, el vínculo con otros trabajos relacionados que se consideran representativos y finalmente se mencionan posibles trabajos futuros.

En el Anexo A se presentan los códigos fuentes de las funciones utilizadas como funciones

consumidoras y productoras en los ejemplos de las pruebas realizadas, así como la instancia de la ley de fusión para cada uno de los tipos de datos utilizados. Finalmente, se presenta la definición recursiva de cada una las funciones obtenidas como resultado de aplicar la ley de fusión a los distintos ejemplos considerados.

En el Anexo B se presentan los valores de tiempo de ejecución y espacio de memoria requeridos por los distintos programas considerados en las pruebas realizadas. Estos valores son los que justifican los datos de los gráficos presentados en el Capítulo 4.

2 Marco Formal

En este capítulo se presentan los conceptos y herramientas formales que sirven de base para el desarrollo del trabajo. Interesa poder representar los programas recursivos en forma genérica con respecto al tipo de dato que manipulan así como poder razonar acerca de los mismos. Para ello se toma un modelado algebraico para la descripción de los tipos de datos y los programas, lo que permite una representación suficientemente abstracta (independiente de aspectos sintácticos) y adecuada para una manipulación formal de los programas.
En especial se introduce el operador recursivo *fold*, el cual captura definiciones de funciones por recursión estructural y puede ser definido para cualquier tipo de dato inductivo. Se presentan leyes de fusión asociadas al operador *fold* poniendo especial énfasis en la ley conocida como *fold*/*build*.

En las siguientes secciones se presentan únicamente las nociones que son necesarias para este trabajo. Se trabaja únicamente con funciones totales por lo que se usa una semántica basada en conjuntos donde los tipos de datos son interpretados como conjuntos y las funciones son funciones entre los mismos. Por más detalles ver, por ejemplo, [BdM97, TM95] .

Contenido

2.1. Functores

El modelado de los tipos de datos está fuertemente relacionado con el concepto de functor por lo que se comienza por la presentación de este último.

Definición 2.1.1 (**Functor**)**.** Un functor F es un operador formado por dos componentes:

- un constructor de tipos $F :: * \to *$;
- una función $F :: (a \to b) \to (F\ a \to F\ b)$ tal que se cumplen las propiedades:

$$\begin{aligned} F\ id &= id \\ F\ (f \circ g) &= F\ f \circ F\ g \end{aligned}$$

♦

La primera propiedad dice que las identidades son preservadas por un functor, mientras que la segunda establece la preservación de composiciones.

Un ejemplo clásico de functor es el que está dado por el constructor de tipos lista y la función *map*:

$$\begin{aligned} &map && :: (a \to b) \to [a] \to [b] \\ &map\ f\ [\,] && = [\,] \\ &map\ f\ (x : xs) && = f\ x : map\ f\ xs \end{aligned}$$

Al comenzar a trabajar con functores surge casi inmediatamente la necesidad de ampliar este concepto para llegar a hablar de functores sobre más de un argumento.

Definición 2.1.2 (**Bifunctor**)**.** Un bifunctor es un functor en dos variables dado por las componentes:

- un constructor de tipos binario $F :: * \to * \to *$;
- una función $F :: (a \to b) \to (c \to d) \to (F\ a\ c \to F\ b\ d)$ tal que se cumplen las siguientes propiedades:

$$\begin{aligned} F\ id_a\ id_b &= id_{Fab} \\ F\ (f \circ g)\ (h \circ k) &= F\ f\ h \circ F\ g\ k \end{aligned}$$

♦

A continuación se presentan, a modo de ejemplo, aquellos functores que resultan más relevantes para el resto del trabajo.

Functor Producto El constructor de tipos de este functor corresponde al constructor de pares. La otra componente es una función que toma dos funciones como argumento las cuales son aplicadas a cada componente del par de entrada. Este functor se suele escribir en forma infija.

$$
\begin{aligned}
&\textbf{type } a \times b &&= (a, b)\\
&(\times) &&:: (a \to c) \to (b \to d) \to a \times b \to c \times d\\
&(f \times g)\ (a, b) &&= (f\ a, g\ b)
\end{aligned}
$$

Asociado a este functor es posible definir las funciones de proyección y la llamada *split* ($\langle \_\ ,\ \_ \rangle$):

$$
\begin{aligned}
&\pi_1 &&:: a \times b \to a\\
&\pi_1\ (a, b) &&= a\\
&\pi_2 &&:: a \times b \to b\\
&\pi_2\ (a, b) &&= b\\
&\langle \_\ ,\ \_ \rangle &&:: (c \to a) \to (c \to b) \to c \to (a, b)\\
&\langle f, g \rangle\ a &&= (f\ a, g\ a)
\end{aligned}
$$

Estas funciones tienen las siguientes propiedades:

$$
\begin{aligned}
f \circ \pi_1 &= \pi_1 \circ (f \times g)\\
g \circ \pi_2 &= \pi_2 \circ (f \times g)\\
\langle f, g \rangle \circ h &= \langle f \circ h, g \circ h \rangle
\end{aligned}
$$

Más aún, es posible definir la acción del functor sobre funciones a partir de la función *split*: $f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle$. El functor producto puede ser generalizado en forma inmediata para n componentes.

Functor Suma El constructor de tipos de este functor corresponde al constructor de la unión disjunta. La otra componente es una función que toma dos funciones como argumento y aplica la que corresponde según el origen de la entrada.

$$
\begin{aligned}
&\textbf{data } a + b &&= \mathit{Left}\ a \mid \mathit{Right}\ b\\
&(+) &&:: (a \to c) \to (b \to d) \to (a + b) \to (c + d)\\
&(f + g)\ (\mathit{Left}\ a) &&= \mathit{Left}\ (f\ a)\\
&(f + g)\ (\mathit{Right}\ b) &&= \mathit{Right}\ (g\ b)
\end{aligned}
$$

Asociada a este functor es posible definir la función que realiza un análisis de casos:

$$
\begin{aligned}
&(\triangledown) &&:: (a \to c) \to (b \to c) \to (a + b) \to c\\
&(f \triangledown g)\ (\mathit{Left}\ a) &&= f\ a\\
&(f \triangledown g)\ (\mathit{Right}\ b) &&= g\ b
\end{aligned}
$$

Adicionalmente las siguientes propiedades se satisfacen:

$$
\begin{aligned}
h \circ (f \triangledown g) &= (h \circ f) \triangledown (h \circ g)\\
(f + g) \circ \mathit{Right} &= \mathit{Right} \circ g\\
(f + g) \circ \mathit{Left} &= \mathit{Left} \circ f
\end{aligned}
$$

Al igual que con el producto, este functor también puede ser generalizado para n componentes.

Estrechamente ligado al concepto de functor está el de transformación natural.

Definición 2.1.3 (**Transformación Natural**)**.** Dados dos functores F y G, una transformación natural $\tau : F \Rightarrow G$ es una familia de funciones $\tau_a :: F\ a \to G\ a$ tal que para toda $f :: a \to b$ se cumple la condición:

$$G\ f \circ \tau_a \quad = \quad \tau_b \circ F\ f \tag{2.1}$$

que es equivalente a decir que el siguiente diagrama conmuta:

$$\begin{array}{ccc} F\ a & \xrightarrow{\tau_a} & G\ a \\ {\scriptstyle F\ f}\downarrow & & \downarrow{\scriptstyle G\ f} \\ F\ b & \xrightarrow[\tau_b]{} & G\ b \end{array}$$

♦

Una transformación natural es una función que recibe una estructura de datos y devuelve otra conteniendo elementos del mismo tipo. A la condición (2.1) se le llama condición de naturalidad.

En el contexto de los lenguajes de programación el concepto de naturalidad es sinónimo al de polimorfismo paramétrico. Una relación entre transformaciones naturales y funciones polimórficas se describe en [Wad89], donde se establece que cada función polimórfica satisface un conjunto de propiedades, conocidas como "free theorems", que se deducen directamente de su tipo. Por ejemplo, la condición de naturalidad de la función *concat*:

$$\begin{array}{ll} concat & :: [[a]] \to [a] \\ concat\ [\,] & = [\,] \\ concat\ (a : as) & = a +\!\!+ concat\ as \end{array}$$

establece por ejemplo que, para toda función $f :: a \to b$,

$$\begin{array}{ccc} [[a]] & \xrightarrow{concat} & [a] \\ {\scriptstyle map\ (map\ f)}\downarrow & & \downarrow{\scriptstyle map\ f} \\ [[b]] & \xrightarrow[concat]{} & [b] \end{array}$$

$$map\ f \circ concat = concat \circ map\ (map\ f)$$

Esto es, el resultado es independiente del orden en que se apliquen las funciones involucradas; se obtiene lo mismo si primero se concatenan las listas y luego se aplica la función f a los elementos de la lista resultado que si se aplica la función f a los elementos de cada una de las listas originales y luego se concatenan éstas. Así, la función *concat* no depende de la estructura de los elementos de las listas que se están concatenando.

2.2. Álgebras

El modelado de tipos de datos a ser presentado se basa en el concepto de álgebra para un determinado functor, el cual se introduce a continuación. En particular, resulta relevante la definición del concepto de álgebra inicial.

Definición 2.2.1 (**F-álgebra**)**.** Dado un functor F, una F-álgebra es una función $h :: F\ a \to a$. Al tipo a se le llama conjunto soporte del álgebra. ♦

Un álgebra es un conjunto de operaciones donde el functor representa la signatura de las mismas.

Ejemplo 2.2.2.

1. Dado el functor F definido como:
   $F\ a\ =\ a\ \times\ a$
   $F\ f\ =\ f\ \times\ f$

   Las siguientes funciones son F-álgebras:

   $+\ ::\ Nat\ \times\ Nat \to Nat$
   $\oplus\ ::\ Nat_p\ \times\ Nat_p \to Nat_p$

   donde $+$ se corresponde con la suma de naturales, $Nat_p = \{0, 1, 2, \cdots, p-1\}$ y

   $n\ \oplus\ m = (n + m)\ mod\ p$

2. Sea el functor G definido como:
   $G\ a\ =\ a + a\ \times\ a$
   $G\ f\ =\ f + f\ \times\ f$

   La siguiente función es una G-álgebra:

   $(not\ \triangledown\ \&\&) :: Bool + Bool\ \times\ Bool \to Bool$

3. Sea el functor H definido como:
   $H\ a\ =\ 1 + a\ \times\ a$

   $H\ f\ =\ id + f\ \times\ f$

   La siguiente función es una H-álgebra:

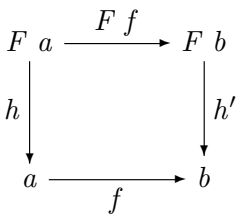
   $(const\ [\,]\ \triangledown\ +\!\!+) :: 1 + [\,a\,]\ \times\ [\,a\,] \to [\,a\,]$

   donde $+\!\!+$ es la función que concatena dos listas.

□

Nota 2.2.3. En las álgebras las constantes de tipo t son representadas como funciones de tipo $1 \to t$, esto es, una función que tomando () devuelve t, por ejemplo $const\ [\,] : 1 \to [\,a\,]$.

Definición 2.2.4 (**F-homomorfismo**)**.** Dadas dos F-álgebras : $h :: F\ a \to a$ y $h' :: F\ b \to b$, un homomorfismo entre ellas es una función $f :: a \to b$ tal que el siguiente diagrama conmuta



♦

Ejemplo 2.2.5. Sean las F-álgebras presentadas en el Ejemplo 2.2.2 (1). Es sencillo ver que

$$h \quad :: Nat \to Nat_p$$
$$h\ n = n\ mod\ p$$

es un F-homomorfismo entre ellas dado que el siguiente diagrama conmuta:

$$\begin{array}{ccc} Nat \times Nat & \xrightarrow{h \times h} & Nat_p \times Nat_p \\ \downarrow{\scriptstyle +} & & \downarrow{\scriptstyle \oplus} \\ Nat & \xrightarrow[h]{} & Nat_p \end{array}$$

□

Definición 2.2.6 (**Álgebra inicial**)**.** Una F-álgebra es **inicial** si y sólo si existe un único F-homomorfismo entre ella y cualquier otra F-álgebra. ♦

El álgebra inicial es también conocida como el álgebra de términos.

Definición 2.2.7 (**Transformador**)**.** Un transformador de álgebras es una función T, $T :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to a)$, polimórfica en a, que transforma F-álgebras en G-álgebras.♦

A partir del tipo de un transformador T es posible derivar la siguiente propiedad usando la técnica de los "free theorems" [Wad89] :

$$\begin{array}{ccc} F\ a & \xrightarrow{F\ f} & F\ b \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle h'} \\ a & \xrightarrow[f]{} & b \end{array} \quad \Rightarrow \quad \begin{array}{ccc} G\ a & \xrightarrow{G\ f} & b \\ \downarrow{\scriptstyle T\ h} & & \downarrow{\scriptstyle T\ h'} \\ a & \xrightarrow[f]{} & b \end{array}$$

Esta propiedad dice que todo homomorfismo entre F-álgebras lo es también entre las álgebras transformadas.

Ejemplo 2.2.8.

Sea una función $f :: b \to a$ y la función T definida como:

$$T :: ((1 + a\ \times\ c) \to c) \to ((1 + b\ \times\ c) \to c)$$

$$T\ (h) = \lambda x\ .\ \textbf{case}\ x\ \textbf{of}$$
$$\qquad\qquad\qquad\qquad Left\ () \to h_1$$
$$\qquad\qquad\qquad\qquad Right\ (n, l) \to h_2\ (f\ n, l)$$

$$h = (h_1 \bigtriangledown h_2)$$

Es posible demostrar que T es efectivamente un transformador. (En el Ejemplo 2.5.3 se mostrará un uso del mismo). □

2.3. Tipos de datos inductivos

En esta sección se muestra cómo es posible modelar el concepto de tipo de datos a partir del concepto de functor antes presentado. Para esta tesis se considera sólo el caso de los tipos inductivos, los que corresponden a declaraciones de tipos de datos de la forma:

$$\textbf{data}\ \tau = C_1\ \tau_{1,1}\ \cdots\ \tau_{1,k_1}\ \mid\ \cdots\ \mid\ C_n\ \tau_{n,1}\ \cdots\ \tau_{n,k_n}$$

donde los $\tau_{i,j}$ están restringidos a:

- tipos constantes (por ej. Int, Char);
- variables de tipo;
- el propio tipo τ;
- constructores de tipos D (ej. List) aplicados a expresiones $\tau'_{i,j}$ con las mismas restricciones.

A partir de una declaración de tipo de esta forma es posible derivar un functor F que tiene como función capturar la signatura del tipo. Dicha derivación procede de la siguiente forma:

- Se sustituyen las alternativas por sumas (esto es, se reemplazan los | por +).
- cada lista de parámetros $\tau_{i,1}\ \cdots\ \tau_{i,k_i}$ se visualiza como un producto $\tau_{i,1}\ \times\ \cdots\ \times\ \tau_{i,k_i}$
- Las ocurrencias del tipo τ se sustituyen por la variable del functor.
- Los constructores de tipos nularios se sustituyen por el tipo unit (denotado por 1).

Aplicando este procedimiento se obtiene el constructor de tipo del functor:

$$F\ a = \sigma_{1,1}\ \times\ \cdots\ \times\ \sigma_{1,k_1} + \cdots + \sigma_{n,1}\ \times\ \cdots\ \times\ \sigma_{n,k_n}$$

donde $\sigma_{i,j} = \tau_{i,j}\ [a\ /\ \tau]$, o sea, el resultado de sustituir las ocurrencias de τ por la variable a en todos los $\tau_{i,j}$.

La acción sobre funciones del functor se obtiene en forma similar a lo realizado para la obtención del constructor de tipos F con la diferencia de que las ocurrencias de la variable de tipo a ahora son reemplazadas por una función f y el resto de las posiciones se reemplazan por la correspondiente función identidad. Esto es,

$$F\ f = g_{1,1} \times \cdots \times g_{1,k_1} + \cdots + g_{n,1} \times \cdots \times g_{n,k_n}$$

$$\text{donde } g_{i,j} = \begin{cases} f & si\ \ \sigma_{i,j} = a \\ id & si\ \ \sigma_{i,j} = t\ \ para\ algún\ tipo\ t \\ D\ g'_{i,j} & si\ \ \sigma_{i,j} = D\ \sigma'_{i,j} \end{cases}$$

donde D en $D\ g'_{i,j}$ corresponde a la función $map_D :: (a \to b) \to (D\ a \to D\ b)$ asociada al constructor de tipo D.

A continuación se presentan los functores que capturan la estructura de algunos tipos de datos, obtenidos por la aplicación del procedimiento expuesto anteriormente.

Ejemplo 2.3.1.

1. Números naturales:

   Dada la definición del tipo:

   $$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

   se puede derivar el functor N

   $$\begin{aligned} N\ a &= 1 + a \\ N &:: (a \to b) \to (N\ a \to N\ b) \\ N\ f &= id + f \end{aligned}$$

2. Expresiones aritméticas:

   Dada la definición del tipo:

   $$\textbf{data}\ Exp = Num\ Int \mid Add\ Exp\ Exp$$

   se puede derivar el functor E

   $$\begin{aligned} E\ a &= Int + a\ \times\ a \\ E &:: (a \to b) \to (E\ a \to E\ b) \\ E\ f &= id + f\ \times\ f \end{aligned}$$

□

En el caso de los tipos paramétricos se hace necesario modificar la expresión del functor asociado al tipo de datos de forma de explicitar las posiciones correspondientes al parámetro del tipo. La signatura de un tipo paramétrico debe cargar con el nombre del parámetro por lo que es necesario trabajar con un bifunctor F. El primer argumento del bifunctor va a representar el parámetro y el segundo la variable que indica las posiciones recursivas del tipo de dato. Fijando el primer argumento en un objeto a, se obtiene el functor $F\ a\ \_$, que usualmente se representa por F_a, tal que:

$$\begin{aligned} F_a\ b &= F\ a\ b \\ F_a\ f &= F\ id_a\ f \end{aligned}$$

Ejemplo 2.3.2.

A continuación se presentan los functores que capturan la estructura de algunos tipos de datos paramétricos.

1. Listas

   Dada la definición del tipo:

   $$\textbf{data}\ List\ a = Nil\ \mid\ Cons\ a\ (List\ a)$$

   se obtiene el functor L_a:

$$
\begin{aligned}
L_a\ b &= 1 + a \times b \\
L_a &:: (b \to c) \to (L_a\ b \to L_a\ c) \\
L_a\ f &= id + id \times f
\end{aligned}
$$

2. Árboles binarios con información en las hojas

   Dada la definición del tipo:

   data *Btree a* = *Leaf a* | *Join* (*Btree a*) (*Btree a*)

   se obtiene el functor B_a:

$$
\begin{aligned}
B_a\ c &= a + c \times c \\
B_a &:: (c \to d) \to (B_a\ c \to B_a\ d) \\
B_a\ f &= id + f \times f
\end{aligned}
$$

3. Árboles binarios con información en los nodos

   Dada la definición del tipo:

   data *Tree a* = *Empty* | *Node* (*Tree a*) *a* (*Tree a*)

   se obtiene el functor T_a:

$$
\begin{aligned}
T_a\ c &= 1 + c \times a \times c \\
T_a &:: (c \to d) \to (T_a\ c \to T_a\ d) \\
T_a\ f &= id + f \times id \times f
\end{aligned}
$$

□

Finalmente, la derivación del functor F que captura la signatura de un tipo recursivo τ, lleva a que cada tipo τ pueda ser interpretado como una solución (un punto fijo) a la ecuación:

$$x \cong F\ x \tag{2.2}$$

donde por solución se entiende a un par dado por un objeto x y un isomorfismo. De todas las posibles soluciones consideraremos la mínima que es la que se corresponde al tipo inductivo. Dicha solución está dada por el tipo denotado por μF y por dos funciones $in_F :: F\ \mu F \to \mu F$ y su inversa.

$$F\ \mu F \underset{in_F^{-1}}{\overset{in_F}{\rightleftarrows}} \mu F$$

El tipo μF corresponde al conjunto de términos finitos del tipo. Y el álgebra in_F, que empaqueta a los constructores del tipo, satisface la propiedad de ser inicial.

A continuación se presentan las funciones in_F para algunos de los tipos de datos presentados en los ejemplos anteriores.

Ejemplo 2.3.3.

1. Naturales

$$\begin{array}{lcl} \mu N & = & Nat \\ in_N & :: & N\ Nat \to Nat \\ in_N\ (Left\ ()) & = & Zero \\ in_N\ (Right\ n) & = & Succ\ n \end{array}$$

2. Expresiones Aritméticas

$$\begin{array}{lcl} \mu E & = & Exp \\ in_E & :: & E\ Exp \to Exp \\ in_E\ (Left\ n) & = & Num\ n \\ in_E\ (Right\ (e, e')) & = & Add\ e\ e' \end{array}$$

□

En el caso un tipo paramétrico D ocurre que $D\ a = \mu F_a$, siendo F_a el functor que captura su signatura. Los constructores son entonces dados por el álgebra inicial $in_{F_a} :: F_a\ (D\ a) \to D\ a$.

Ejemplo 2.3.4.

1. Listas

$$\begin{array}{lcl} \mu L_a & = & List\ a \\ in_{L_a} & :: & L_a\ List\ a \to List\ a \\ in_{L_a}\ (Left\ ()) & = & Nil \\ in_{L_a}\ (Right\ (x, xs)) & = & Cons\ x\ xs \end{array}$$

2. Árboles binarios con información en las hojas

$$\begin{array}{lcl} \mu B_a & = & Btree\ a \\ in_{B_a} & :: & B_a\ Btree\ a \to Btree\ a \\ in_{B_a}\ (Left\ a) & = & Leaf\ a \\ in_{B_a}\ (Right\ (i, d)) & = & Join\ i\ d \end{array}$$

□

2.4. El Operador Fold

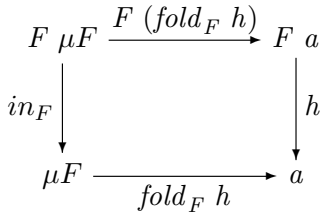
A partir de la propiedad que define el álgebra inicial es posible asociar a cada tipo de dato inductivo τ un operador recursivo que captura el comportamiento de las funciones definidas por recursión estructural sobre τ.

Definición 2.4.1 (Fold). Dada una F-álgebra: $h :: F\ a \to a$, se define fold [Bir98](o catamorfismo [MFP91]), denotado por $fold_F\ h$, al único homomorfismo entre in_F y h. ♦

Por ser $fold_F h$ un homomorfismo se cumple la ecuación:

$$fold_F\ h \circ in_F \quad = \quad h \circ F\ fold_F\ h \tag{2.3}$$

Diagramáticamente,



A continuación se presenta una serie de ejemplos donde se deja en evidencia las caracteristicas del operador *fold* para distintos tipos de datos. Al presentar las instancias del operador *fold* para tipos de datos concretos, vamos a escribir las operaciones de un álgebra $h_1 \triangledown h_2 \triangledown ... \triangledown h_n$ como una tupla $(\overline{h_1}, \overline{h_2}, ..., \overline{h_n})$ donde cada $\overline{h_i} :: \sigma_{i_1} \to \sigma_{i_2} \to \cdots \to \sigma_{i_k} \to a$ es la versión currificada de $h_i :: \sigma_{i_1} \times \sigma_{i_2} \times \cdots \times \sigma_{i_k} \to a$. Operaciones de tipo $1 \to a$ serán representadas por el correspondiente valor de tipo a.

Ejemplo 2.4.2.

1. Naturales

   Considerando la definición del functor para los naturales (N) y su álgebra inicial (in_N), $fold_N$ es la función tal que:

   $$\begin{aligned} &fold_N && :: (a, a \to a) \to Nat \to a \\ &fold_N\ (h_1, h_2) &&= f_N \\ &\quad \textbf{where} \\ &\quad\quad f_N\ Zero && = h_1 \\ &\quad\quad f_N\ (Succ\ n) && = h_2\ (f_N\ n) \end{aligned}$$

   En particular, la función $add\ m : Nat \to Nat$,

   $$\begin{aligned} &add\ m\ Zero && = m \\ &add\ m\ (Succ\ n) && = Succ\ (add\ m\ n) \end{aligned}$$

   se puede expresar en términos del operador fold:

   $$add\ m = fold_N\ (m, Succ)$$

2. Expresiones aritméticas

   Considerando la definición del functor para las expresiones aritméticas (E) y su álgebra inicial (in_E), $fold_E$ es la función tal que:

   $$fold_E \qquad :: (Int \to a, a \to a \to a) \to Exp \to a$$

$$
\begin{aligned}
&fold_E\ (h_1, h_2) = f_E \\
&\quad \textbf{where} \\
&\quad\quad f_E\ (Num\ i) \quad = h_1\ i \\
&\quad\quad f_E\ (Add\ e\ e') = h_2\ (f_E\ e)\ (f_E\ e')
\end{aligned}
$$

En particular, la función que evalúa una expresión aritmética:

$$
\begin{aligned}
&eval \qquad\qquad :: Exp \to Int \\
&eval\ (Num\ i) \quad = i \\
&eval\ (Add\ e\ e') = add\ (eval\ e)\ (eval\ e')
\end{aligned}
$$

se puede expresar en términos del operador fold:

$$eval = fold_E\ (id, add)$$

3. Listas

Considerando la definición del functor para las listas (L_a) y su álgebra inicial (in_{L_a}), $fold_{L_a}$ es la función tal que:

$$
\begin{aligned}
&fold_{L_a} :: \qquad (b, a \to b \to b) \to List\ a \to b \\
&fold_{L_a}\ (h_1, h_2) = f_L \\
&\quad \textbf{where} \\
&\quad\quad f_L\ Nil \qquad\quad = h_1 \\
&\quad\quad f_L\ (Cons\ x\ xs) = h_2\ x\ (f_L\ xs)
\end{aligned}
$$

Por ejemplo, la función que calcula la cantidad de elementos de una lista:

$$
\begin{aligned}
&length \qquad\qquad :: List\ a \to Nat \\
&length\ Nil \qquad\quad = Zero \\
&length\ (Cons\ x\ xs) = Succ\ (length\ xs)
\end{aligned}
$$

se puede expresar en términos del operador fold:

$$length = fold_{L_a}\ (Zero, \lambda a\ b\ .\ Succ\ b)$$

Otro ejemplo es la función que suma los elementos de una lista:

$$
\begin{aligned}
&suma \qquad\qquad :: Num\ a \Rightarrow List\ a \to a \\
&suma\ Nil \qquad\quad = 0 \\
&suma\ (Cons\ x\ xs) = x + suma\ xs
\end{aligned}
$$

la cual se puede expresar en términos del operador fold como:

$$suma = fold_{L_a}\ (0, (+))$$

4. Árbol binario con información en las hojas

Considerando la definición del functor para los árboles binarios con información en las hojas (B_a) y su álgebra inicial (in_{B_a}), $fold_{B_a}$ es la función tal que:

$$
\begin{aligned}
&fold_{B_a} :: (a \to b, b \to b \to b) \to Btree\ a \to b \\
&fold_{B_a}\ (h_1, h_2) = f_B \\
&\quad \textbf{where} \\
&\qquad f_B\ (Leaf\ a) = h_1\ a \\
&\qquad f_B\ (Join\ i\ d) = h_2\ (f_B\ i)\ (f_B\ d)
\end{aligned}
$$

Por ejemplo, la función $leaves :: Btree\ a \to List\ a$:

$$
\begin{aligned}
&leaves\ (Leaf\ a) = wrap\ a \\
&leaves\ (Join\ i\ d) = leaves\ i +\!\!+ leaves\ d
\end{aligned}
$$

$$
\begin{aligned}
&wrap :: a \to List\ a \\
&wrap\ a = Cons\ a\ Nil
\end{aligned}
$$

siendo $(+\!\!+) :: List\ a \to List\ a \to List\ a$ la función que concatena dos listas, se puede expresar en términos del operador *fold*:

$$leaves = fold_{B_a}\ (wrap, (+\!\!+))$$

La función que obtiene el máximo valor de un árbol:

$$
\begin{aligned}
&maxtree :: Ord\ a \Rightarrow (Btree\ a) \to a \\
&maxtree\ (Leaf\ a) = a \\
&maxtree\ (Join\ l\ r) = max\ (maxtree\ l)\ (maxtree\ r)
\end{aligned}
$$

también se puede expresar en términos del operador *fold*:

$$maxtree = fold_{B_a}\ (id, max)$$

□

Ejemplo 2.4.3.

La función *map* presentada al comienzo de este capítulo puede ser expresada en términos del operador *fold*:

$$
\begin{aligned}
&map :: (a \to b) \to List\ a \to List\ b \\
&map\ f = fold_{L_a}\ (Nil, \lambda a\ bs \to Cons\ (f\ a)\ bs)
\end{aligned}
$$

Es posible extender la definición de una función *map* a otros tipos de datos. En el caso de los árboles binarios con información en las hojas la correspondiente función *map* es:

$$
\begin{aligned}
&mapBt\ f\ (Leaf\ a) = Leaf\ (f\ a) \\
&mapBt\ f\ (Join\ i\ d) = Join\ (mapBt\ f\ i)\ (mapBt\ f\ d)
\end{aligned}
$$

Al igual que la función *map* para listas, esta función puede ser expresada en términos del operador *fold* asociado a este tipo de datos:

$$mapBt \;\; :: \;\; (a \to b) \to (Btree\ a \to Btree\ b)$$
$$mapBt\ f \;\; = \;\; fold_{B_a}\ (Leaf \circ f, Join)$$

Esta situación se puede generalizar a cualquier tipo de datos polimórfico D con functor base F:

$$mapD \;\; :: \;\; (a \to b) \to (D\ a \to D\ b)$$
$$mapD\ f \;\; \stackrel{def}{=} \;\; fold_{F_a}(in_{F_b} \circ F\ f\ id)$$

Es simple probar que el constructor de tipos D junto con la función *mapD* forman un functor usualmente llamado **functor de tipo** [BdM97]. □

2.5. Fusión

En la sección anterior se presentó la definición genérica del operador *fold*. Dicho operador está dado por el único homomorfismo que existe entre el álgebra inicial y cualquier otra. Esta propiedad permite derivar propiedades algebraicas para fold, conocidas como **leyes de fusión**, las cuales resultan de utilidad práctica para la transformación de programas. Las leyes que se enuncian a continuación son de particular interés ya que han servido de inspiración para la formulación de algunas leyes sobre acumulaciones que se presentan en este trabajo.

Ley 2.5.1 (Fusión de fold)**.**

$$f \circ h = k \circ F\ f \quad \Rightarrow \quad f \circ fold_F\ h = fold_F\ k$$

Esta ley establece que la composición de un fold con un homomorfismo resulta en un fold.

Ley 2.5.2 (Fusión fold/fold - Acid Rain)**.**

$$T :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to a) \quad \Rightarrow \quad fold_F\ h \circ fold_G\ (T\ in_F) = fold_G\ (T\ h)$$

Esta ley permite fusionar la composición de dos folds obteniendo como resultado un nuevo fold. Para esto el álgebra del primer fold debe ser el resultado de la aplicación de un transformador a los constructores del tipo intermedio entre los dos folds.

Ejemplo 2.5.3.

Sea la siguiente composición de funciones sobre listas:

$$parlist \circ map\ f$$

donde la función *parlist* se define como:

$$\begin{aligned} &parlist && :: List\ a \rightarrow List\ (a, a) \\ &parlist\ Nil && = Nil \\ &parlist\ (Cons\ x\ xs) && = Cons\ (x, x)\ (parlist\ xs) \end{aligned}$$

Como fue presentado en el Ejemplo 2.4.3, la función $map\ f$ puede expresarse como un *fold*:

$$map\ f = fold_{L_a}\ (Nil, \lambda a\ bs\ .\ Cons\ (f\ a)\ bs)$$

Dado que $map\ f$ genera la lista intermedia usando esencialmente los constructores de listas, es simple expresar el álgebra del *fold* en términos de un transformador haciendo abstracción de dicho constructores. El transformador corresponde al presentado en el Ejemplo 2.2.8 :

$$map\ f = fold_{L_a}\ (T\ (Nil, Cons))$$

donde $T\ (h_1, h_2) = (h_1, \lambda a\ b\ .\ h_2\ (f\ a)\ b)$

Por otro lado, es posible expresar la función *parlist* en términos del operador *fold*:

$parlist = fold_{L_a}\ (Nil, \lambda c\ d\ .\ Cons\ (c, c)\ d)$

Es posible entonces simplificar la composición original haciendo uso de la Ley 2.5.2:

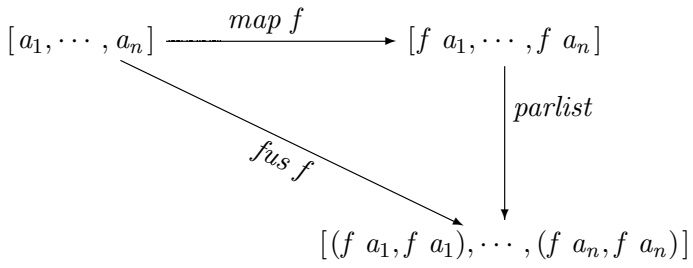
$$\begin{aligned} & parlist \circ map\ f \\ = & \quad \{\ \text{Definición de las funciones}\ \} \\ & fold_{L_a}\ (Nil, \lambda c\ d\ .\ Cons\ (c, c)\ d) \circ fold_{L_a}\ (T\ (Nil, Cons)) \\ = & \quad \{\ \text{Ley 2.5.2}\ \} \\ & fold_{L_a}\ (T\ (Nil, \lambda c\ d\ .\ Cons\ (c, c)\ d)) \\ = & \quad \{\ \text{Definición de } T\ \} \\ & fold_{L_a}\ (Nil, \lambda a\ b\ .\ Cons\ (f\ a, f\ a)\ b) \end{aligned}$$

que se corresponde con la definición recursiva:

$$\begin{aligned} &fus\ f\ Nil && = Nil \\ &fus\ f\ (Cons\ a\ as) && = Cons\ (f\ a, f\ a)\ (fus\ f\ as) \end{aligned}$$

□

El ejemplo anterior permite visualizar claramente el efecto obtenido por la aplicación de la ley de fusión al evitar la construcción de la estructura de datos intermedia. En la composición original la función $map\ f$ crea una nueva lista resultado de aplicar la función f a cada uno de los elementos de la lista argumento. Luego esta lista es consumida, por la función *parlist* para crear una nueva lista donde cada elemento se obtiene de duplicar los elementos de la lista generada por $map\ f$. Por el contrario, en la función resultante de la aplicación de la ley de fusión se toma la lista argumento y directamente se construye una nueva lista donde cada elemento se forma como un par donde cada una de sus componentes es el resultado de aplicar la función f a cada uno de los elementos de la lista argumento.



De esta forma se evita la construcción de la lista intermedia cuyo único sentido es la comunicación entre las funciones involucradas en la composición original. Resultados de este tipo son los que se buscan por la aplicación de las leyes de fusión.

2.6. "Short-Cut Fusion"

Otro ejemplo conocido de ley algebraica para la transformación de programas es la llamada **"Short-Cut Fusion"** [Gil96, GLJ93, TM95], o regla **fold/build**. Es una técnica que permite la eliminación de la estructura de datos intermedia generada en composiciones $c \circ p$ entre una función consumidora c y una función productora p. A diferencia de las Leyes 2.5.1 y 2.5.2, esta ley es consecuencia de propiedades de parametricidad ("free theorems" [Wad89]) asociadas a una función polimórfica. Al mismo tiempo, la ley de Short-Cut Fusion puede ser explicada desde un punto de vista categórico [GUV04].

Para su aplicación "Short-Cut Fusion" requiere que tanto la función consumidora como la productora cumplan ciertas condiciones. La función consumidora debe ser expresable en términos del operador *fold*, mientras que la función productora debe ser tal que la generación de la estructura intermedia sea realizada utilizando únicamente los constructores del tipo. Por ejemplo, si p produce una lista como resultado, debe garantizar que su construcción se realiza en términos de los constructores *Nil* y *Cons*. Para lograr el cumplimiento de esta condición se requiere que la función productora sea expresable en términos de una función llamada *build* [GLJ93], la cual utiliza un "template" la cual deja en evidencia las ocurrencias de los constructores de tipo de la estructura intermedia.

La idea de esta transformación consiste en reemplazar las ocurrencias de los constructores de la estructura intermedia en el cuerpo de la función productora, por las funciones del álgebra del fold correspondiente a la función consumidora.

Se define la función $build_F$ como:

$$
\begin{aligned}
&build_F \quad :: (\forall\ a\ .\ (F\ a \to a) \to b \to a) \to b \to \mu F \\
&build_F\ g = g\ in_F
\end{aligned}
$$

Comúnmente la función *build* se presenta de tipo $(\forall\ a\ .\ (F\ a \to a) \to a) \to \mu F$. En esta tesis se opta por trabajar con esta definición alternativa ya que permite la manipulación de las

propiedades a nivel funcional. En [GUV04] esta definición es usada para dar una justificación alternativa de *build*.

Con esta definición se llega a la expresión habitual de la regla:

Ley 2.6.1 (Regla Fold/Build). *Para φ estricta,*

$$\begin{array}{c} g :: \forall\ a\ .\ (F\ a \to a) \to b \to a \\ \Rightarrow \\ fold_F\ \varphi \circ build_F\ g = g\ \varphi \end{array}$$

Prueba: Al considerar la función $g :: \forall\ a\ .\ (F\ a \to a) \to b \to a$, un "free theorem" asociado con el tipo de g es:

$$f \circ \psi = \varphi \circ F\ f\ \Rightarrow\ f \circ g\ \psi = g\ \varphi$$

con $f :: a \to c$, $\psi :: F\ a \to a$ y $\varphi :: F\ c \to c$.

Tomando f como $fold_F\ \varphi$ y ψ como in_F la regla anterior se instancia como:

$$fold_F\ \varphi \circ in_F = \varphi \circ F\ fold_F\ \varphi\ \Rightarrow\ fold_F\ \varphi \circ g\ in_F = g\ \varphi$$

Se observa que el antecedente de esta expresión resulta ser la propiedad de la definición del operador *fold* (Ecuación 2.3), por lo que siempre se cumple, por lo que se obtiene como expresión final:

$$fold_F\ \varphi \circ g\ in_F = g\ \varphi$$

A continuación se presenta una serie de ejemplos donde se da la instancia de esta ley para distintos tipos de datos.

Ejemplo 2.6.2.

1. Listas

   $$fold_{L_a}\ (h_1, h_2) \circ build_{L_a}\ g = g\ (h_1, h_2)$$

   donde

   $$\begin{array}{l} build_{L_a} \quad :: (\forall\ b\ .\ (b, a \to b \to b) \to c \to b) \to c \to List\ a \\ build_{L_a}\ g = g\ (Nil, Cons) \end{array}$$

   Reconsiderando la composición *parlist* $\circ$ *map* f presentada en el Ejemplo 2.5.3, ahora se muestra cómo es posible fusionarla usando la regla *fold*/*build*. Primero es necesario expresar la función *map* f en términos de *build*

   $$\begin{array}{l} map\ f = build_{L_a}\ \ gmap \\ \quad \textbf{where} \\ \quad\quad gmap\ (fnil, fcons)\ Nil \qquad\qquad = fnil \\ \quad\quad gmap\ (fnil, fcons)\ (Cons\ a\ as) = fcons\ (f\ a)\ (gmap\ (fnil, fcons)\ as) \end{array}$$

   La expresión de *parlist* en términos de *fold* es la dada en el Ejemplo 2.5.3:

$$parlist = fold_{L_a}\ (Nil, \lambda c\ d\ .\ Cons\ (c, c)\ d)$$

Por lo tanto, al aplicar la Ley 2.6.1 se obtiene como resultado:

$$parlist \circ map\ f = gmap\ (Nil, \lambda c\ d\ .\ Cons\ (c, c)\ b)$$

que corresponde a la definición recursiva:

$$\begin{aligned} fus\ Nil &= Nil \\ fus\ (Cons\ a\ as) &= Cons\ (a, a)\ (fus\ as) \end{aligned}$$

la cual coincide con la función antes calculada por aplicación de las otras leyes de fusión.

2. Árboles binarios con información en las hojas

$$fold_{B_a}\ (h_1, h_2) \circ build_{B_a}\ g = g\ (h_1, h_2)$$

donde

$$\begin{aligned} build_{B_a} &:: (\forall\ b\ .\ (a \to b, b \to b \to b) \to c \to b) \to c \to Btree\ a \\ build_{B_a}\ g &= g\ (Leaf, Join) \end{aligned}$$

Es posible usar esta ley para fusionar la siguiente composición:

$$\begin{aligned} fus &:: Btree\ a \to List\ a \\ fus &= leaves \circ espejo \end{aligned}$$

$$\begin{aligned} espejo &:: Btree\ a \to Btree\ a \\ espejo\ (Leaf\ a) &= Leaf\ a \\ espejo\ (Join\ i\ d) &= Join\ (espejo\ d)\ (espejo\ i) \end{aligned}$$

$$\begin{aligned} leaves &:: Btree\ a \to List\ a \\ leaves\ (Leaf\ a) &= wrap\ a \\ leaves\ (Join\ i\ d) &= leaves\ i +\!\!+ leaves\ d \end{aligned}$$

$$\begin{aligned} wrap &:: a \to List\ a \\ wrap\ a &= Cons\ a\ Nil \end{aligned}$$

Para esto, primero es necesario expresar la función *espejo* en términos de *build*:

$$\begin{aligned} &espejo = build_{B_a}\ gespejo \\ &\quad \textbf{where} \\ &\qquad gespejo\ (fleaf, fjoin)\ (Leaf\ a) = fleaf\ a \\ &\qquad gespejo\ (fleaf, fjoin)\ (Join\ i\ d) = fjoin\ (gespejo\ (fleaf, fjoin)\ d) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (gespejo\ (fleaf, fjoin)\ i) \end{aligned}$$

La función *leaves* en términos de *fold* fue presentada en el Ejemplo 2.4.2:

$$leaves = fold_{B_a}\ (wrap, (+\!\!+))$$

Aplicando la Ley 2.6.1 se obtiene como resultado:

$$leaves \circ espejo = gespejo\ (wrap, +\!\!+)$$

que corresponde a la definición recursiva:

$$\begin{aligned} fus\ (Leaf\ a) &= wrap\ a \\ fus\ (Join\ i\ d) &= fus\ d +\!\!+ fus\ i \end{aligned}$$

□

Al observar el enunciado de las leyes 2.5.2 y 2.6.1 es posible reconocer que la primera es un caso particular de la segunda. Esto se debe a que las funciones productoras de estructuras de tipo μF de la forma $fold_G\ (T\ in_F)$ pueden expresarse en términos de $build_F$. Para esto es necesario determinar la función g del *build*. Recordando que $build_F\ g = g\ in_F$, es sencillo ver que la función g requerida en este caso no es otra que $g\ \varphi = fold_G\ (T\ \varphi)$.

Propiedad 2.6.3.

$$fold_G\ (T\ in_F) = build_F\ g \qquad para \quad g\ \varphi = fold_G\ (T\ \varphi)$$

Prueba:

$$\begin{aligned} & build_F\ g \\ = & \quad \{\ \text{Definición de } build_F\ \} \\ & g\ in_F \\ = & \quad \{\ \text{Definición de } g\ \} \\ & fold_G\ (T\ in_F) \end{aligned}$$

□

Usando esta propiedad se puede probar, por ejemplo, la propia Ley 2.5.2:

$$\begin{aligned} & fold_F\ h \circ fold_G\ (T\ in_F) \\ = & \quad \{\ \text{Definición de } g\ \} \\ & fold_F\ h \circ g\ in_F \\ = & \quad \{\ \text{Definición de } build_F\ \} \\ & fold_F\ h \circ build_F\ g \\ = & \quad \{\ \text{Ley 2.6.1}\ \} \\ & g\ h \\ = & \quad \{\ \text{Definición de } g\ \} \\ & fold_G\ (T\ h) \end{aligned}$$

□

El Operador afold

3

Enfrentados al problema de trabajar con acumulaciones, se presenta como primera aproximación la revisión de un operador, llamado *afold* [Par03], que captura el comportamiento de acumulaciones definidas por recursión estructural.

Se propone luego una extensión al operador *afold* que será presentado para el caso de listas y árboles binarios. Junto a la definición de la extensión se presenta un conjunto seleccionado de leyes de fusión para la misma.

Dentro del contexto de programación funcional uno de los operadores más conocidos es *foldl*, el cual permite capturar el comportamiento de un tipo de acumulaciones sobre listas. En este capítulo se muestra que *foldl* es un caso particular de *afold* y que leyes conocidas de *foldl* se obtienen a partir de las de *afold*. Se realiza una extensión de *foldl* similar a la presentada para el operador *afold*.

Analizar el caso particular de *foldl* resulta de utilidad para mostrar una aplicación de las leyes de fusión en forma inversa (fisión de programas) en la que se reconstruye el estudio presentado en [Gib06].

Durante el capítulo se marcan las fortalezas del operador *afold* y sobre el final del mismo se presentan algunas limitaciones que dan lugar a los desarrollos que se consideran en los posteriores capítulos.

Contenido

3.1. Definición

Sean las siguientes funciones sobre listas:

$$
\begin{aligned}
&areverse && :: (List\ a, List\ a) \rightarrow List\ a\\
&areverse\ (Nil, ys) && = ys\\
&areverse\ (Cons\ x\ xs, ys) && = areverse\ (xs, Cons\ x\ ys)
\end{aligned}
$$

$$
\begin{aligned}
&sumaAc && :: Num\ a \Rightarrow (List\ a, a) \rightarrow a\\
&sumaAc\ (Nil, n) && = n\\
&sumaAc\ (Cons\ x\ xs, n) && = sumaAc\ (xs, x + n)
\end{aligned}
$$

$$
\begin{aligned}
&sumAnt && :: Num\ a \Rightarrow (List\ a, a) \rightarrow List\ a\\
&sumAnt\ (Nil, n) && = Nil\\
&sumAnt\ (Cons\ x\ xs, n) && = Cons\ (x + n)\ (sumAnt\ (xs, x + n))
\end{aligned}
$$

Es posible observar que estas definiciones poseen un conjunto de características similares:

- Están definidas por recursión estructural en el primer argumento
- Pasan información a las llamadas recursivas a través de un acumulador
- En el paso recursivo es posible utilizar el valor del acumulador y el contenido en la cabeza de la lista.

Dichas características se pueden generalizar en el siguiente esquema para ciertas funciones h_1, h_2 y ψ:

$$
\begin{aligned}
&f && :: (List\ a, x) \rightarrow c\\
&f\ (Nil, x) && = h_1\ x\\
&f\ (Cons\ a\ as, x) && = h_2\ a\ (f\ (as, \psi\ a\ x))\ x
\end{aligned}
$$

La función *areverse* se obtiene instanciando en f, $h_1 = id, h_2\ n\ r\ x = r, \psi = Cons$; en el caso de *sumaAc*, $h_1 = id, h_2\ n\ r\ x = r, \psi = (+)$; y para *sumAnt*, $h_1\ x = Nil, h_2\ n\ r\ x = Cons\ (n + x)\ r, \psi = (+)$.

Con el objetivo de capturar el comportamiento de este tipo de funciones se ha propuesto la definición de un operador recursivo llamado *afold* [Par03]. Dicho operador es presentado como alternativa para capturar el comportamiento de funciones con acumulaciones frente al uso de un *fold* de alto orden. La idea es extender el comportamiento del operador *fold* de forma de permitir el manejo explícito del acumulador. La fortaleza de este operador se concentra en el agregado de una función de acumulación que es la encargada de modificar el acumulador al ser pasado en las sucesivas llamadas recursivas.

Por ejemplo, para el tipo de las listas el operador tiene la siguiente definición:

$$
\begin{aligned}
&afold_{L_a} :: (x \rightarrow c, a \rightarrow c \rightarrow x \rightarrow c) \rightarrow (a \rightarrow x \rightarrow x) \rightarrow (List\ a, x) \rightarrow c\\
&afold_{L_a}\ (h_1, h_2)\ \psi = f\\
&\quad \textbf{where}\ f\ (Nil, x) && = h_1\ x\\
&\qquad\quad f\ (Cons\ a\ as, x) && = h_2\ a\ (f\ (as, \psi\ a\ x))\ x
\end{aligned}
$$

A continuación se enumera una serie de definiciones que permiten concluir con la definición genérica del operador *afold*. Estas definiciones son las dadas en [Par01, Par03].

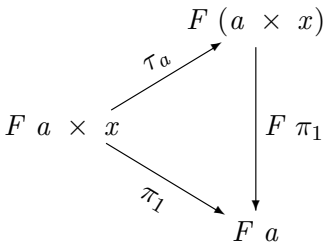
Debido a la variedad de posibles formas de acumulación (es decir, de manipular el acumulador) no es posible dar una expresión general, única para la función de acumulación pero sí es posible establecer un conjunto de condiciones que debe cumplir, determinando de esta forma lo que se llama una ***función de acumulación apropiada***. En esta función se representan las modificaciones que se realizan sobre el parámetro de acumulación.

Definición 3.1.1 (**Función de acumulación apropiada**)**.** Sea F un functor, una función $\tau :: F\ a\ \times\ x \to F\ (a\ \times\ x)$ es una función de acumulación apropiada si cumple las siguientes condiciones:

Naturalidad τ debe ser natural en a, esto es, para toda $f : a \to b$ se satisface el siguiente diagrama:

$$\begin{array}{ccc} F\ a\ \times\ x & \xrightarrow{\tau_a} & F\ (a\ \times\ x) \\ \Big\downarrow{\scriptstyle F\ f\ \times\ id_x} & & \Big\downarrow{\scriptstyle F\ (f\ \times\ id_x)} \\ F\ b\ \times\ x & \xrightarrow{\tau_b} & F\ (b\ \times\ x) \end{array}$$

Preservación de datos y estructura



♦

La condición de naturalidad establece una restricción sobre la información que puede ser utilizada al momento de modificar el acumulador: además del acumulador en sí mismo, los únicos valores disponibles para la acumulación son aquellos contenidos en los nodos de la estructura de datos que se está manipulando. La condición de preservación de datos y estructura establece que la función τ no puede modificar ni la estructura del tipo de datos ni los valores de los datos originales contenidos en la misma.

Definición 3.1.2 (**Extensión de un functor**)**.** Dado un functor F y una función de acumulación apropiada τ_a, se define su extensión $\overline{F}$ de la siguiente forma: $\overline{F}\ a = F\ a$, y para cada función $f : a\ \times\ x \to b$,

$$\overline{F}\ f = F\ a\ \times\ x \xrightarrow{\tau_a} F\ (a\ \times\ x) \xrightarrow{F\ f} F\ b$$

♦

Esta extensión del functor, que también involucra a la función de acumulación, es la responsable de distribuir los valores modificados del acumulador en las sucesivas llamadas recursivas.

El siguiente paso consiste en extender el concepto de álgebra inicial al de **álgebra inicial con acumulaciones** para dar origen al nuevo operador *afold*.

Definición 3.1.3 (**Álgebra inicial con acumulaciones**). Un álgebra inicial in_F se dice inicial con acumulaciones si para cada objeto x, para cada $\tau : F\ a\ \times\ x \to F\ (a\ \times\ x)$ apropiada para acumulaciones y $h : F\ a\ \times\ x \to a$ existe una única función $f : \mu F\ \times\ x \to a$ que hace que el siguiente diagrama conmute:

$$\begin{array}{ccc} F\ \mu F\ \times\ x & \xrightarrow{\langle \overline{F}\ f, \pi_2 \rangle} & F\ a\ \times\ x \\ in_F\ \times\ id_x \downarrow & & \downarrow h \\ \mu F\ \times\ x & \xrightarrow[f]{} & a \end{array}$$

La existencia de este tipo de álgebras esta garantizada en los contextos de alto orden. (ver por ejemplo [Par03]) ♦

Definición 3.1.4 (***afold***). Se llama *afold* a la única función que resulta del álgebra inicial con acumulaciones y se denota por $afold_F\ (h, \tau) : \mu F\ \times\ x \to a$. ♦

Ejemplo 3.1.5. Para el caso del tipo de las listas, a partir de su functor L_a, la función de acumulación τ_b puede expresarse de la siguiente forma:

$$\tau_b = (\pi_1 + \Phi) \circ d$$

donde

$$\begin{array}{rcl} \Phi & :: & (a\ \times\ b)\ \times\ x \to a\ \times\ (b\ \times\ x) \\ \Phi\ ((a, b), x) & = & (a, (b, \psi\ (a, x))) \end{array}$$

para alguna función $\psi :: a\ \times\ x \to x$ y $d :: (1 + a\ \times\ b)\ \times\ x \to (1\ \times\ x + (a\ \times\ b)\ \times\ x)$ función distributiva.

Para una L_a-álgebra $h = (h_1\ \triangledown\ h_2) \circ d :: (1 + a\ \times\ c)\ \times\ x \to c$, la función $f = afold_{L_a}\ (h, \tau)$ es tal que:

$$\begin{array}{rcl} f \circ (Nil\ \times\ id_x) & = & h_1 \\ f \circ (Cons\ \times\ id_x) & = & h_2 \circ \langle (id_a\ \times\ f) \circ \Phi, \pi_2 \rangle \end{array}$$

□

A continuación se presenta la instancia de este operador para algunos tipos de datos más frecuentemente usados. Con el objetivo de simplificar la notación se opta por una representación funcional de los mismos (tipo Haskell). De la misma forma que lo realizado para el operador *fold* para tipos de datos concretos, vamos a escribir las operaciones de la función h del operador *afold* como una tupla $(\overline{h_1}, \overline{h_2}, ..., \overline{h_n})$ donde cada $\overline{h_i} :: \sigma_{i_1} \to \sigma_{i_2} \to \cdots \to \sigma_{i_k} \to x \to a$ es la versión currificada de $h_i :: \sigma_{i_1}\ \times\ \sigma_{i_2}\ \times\ \cdots\ \times\ \sigma_{i_k}\ \times\ x \to a$. Operaciones de tipo $1 \to x \to a$ serán representadas por operaciones de tipo $x \to a$. En las funciones involucradas en la acumulación también se utiliza una versión currificada de las mismas. Por más detalles ver [Par03].

Ejemplo 3.1.6.

1. Listas

   A partir de este cambio de notación y lo presentado en el Ejemplo 3.1.5 se obtiene la expresión vista al comienzo de esta Sección para el *afold* sobre listas:

$$
\begin{aligned}
&afold_{L_a} :: (x \to c, a \to c \to x \to c) \to (a \to x \to x) \to (List\ a, x) \to c\\
&afold_{L_a}\ (h_1, h_2)\ \psi = f\\
&\quad \textbf{where}\\
&\quad\quad f\ (Nil, x) \quad\quad\ = h_1\ x\\
&\quad\quad f\ (Cons\ a\ as, x) = h_2\ a\ (f\ (as, \psi\ a\ x))\ x
\end{aligned}
$$

   - Sea la función que calcula el largo de una lista:

$$
\begin{aligned}
&largo \quad :: List\ a \to Int\\
&largo\ xs = aLargo\ (xs, 0)
\end{aligned}
$$

$$
\begin{aligned}
&aLargo \ :: (List\ a, Int) \to Int\\
&aLargo\ (Nil, n) \quad\quad = n\\
&aLargo\ (Cons\ x\ xs, n) = aLargo\ (xs, n + 1)
\end{aligned}
$$

     Es posible expresar aLargo como un *afold* de la siguiente forma:

$$
\begin{aligned}
&aLargo = afold_{L_a}\ (id, mdl)\ \psi\\
&\quad\quad\quad \textbf{where}\\
&\quad\quad\quad\quad \psi\ a\ b = b + 1
\end{aligned}
$$

     donde $mdl\ a\ b\ c = b$.

   - Las funciones consideradas al comienzo de esta Sección se pueden expresar en términos del operador *afold* de la siguiente forma:

$$areverse = afold_{L_a}\ (id, mdl)\ Cons$$

$$sumaAc\ = afold_{L_a}\ (id, mdl)\ (+)$$

$$
\begin{aligned}
&sumAnt\ = afold_{L_a}\ (const\ Nil, h_2)\ (+)\\
&\quad\quad\quad \textbf{where}\\
&\quad\quad\quad\quad h_2\ a\ b\ c = Cons\ (a + c)\ b
\end{aligned}
$$

2. Árboles binarios con información en las hojas

$$\textbf{data}\ Btree\ a = Leaf\ a \mid Join\ (Btree\ a)\ (Btree\ a)$$

$$
\begin{aligned}
&afold_{B_a} :: (a \to x \to c, c \to c \to x \to c) \to (x \to x) \to (x \to x) \to (Btree\ a, x) \to c\\
&afold_{B_a}\ (h_1, h_2)\ \psi\ \psi' = f\\
&\quad \textbf{where}\\
&\quad\quad f\ (Leaf\ a, z) \quad = h_1\ a\ z\\
&\quad\quad f\ (Join\ t\ u, z) = h_2\ (f\ (t, \psi\ z))\ (f\ (u, \psi'\ z))\ z
\end{aligned}
$$

   Observar que las funciones de acumulación (ψ, ψ') usadas en ambas llamadas recursivas pueden ser distintas entre sí.

a) Sea *down* la función que sustituye el valor de cada hoja por su profundidad en el árbol.

$$
\begin{aligned}
&down && :: (Btree\ a, Int) \to Btree\ Int\\
&down\ (Leaf\ a, z) && = Leaf\ z\\
&down\ (Join\ t\ u, z) && = Join\ (down\ (t, z+1))\ (down\ (u, z+1))
\end{aligned}
$$

Es posible expresar *down* como un *afold* donde:

$$
\begin{aligned}
&down = afold_{B_a}\ (h_1, h_2)\ \psi\ \psi\\
&\quad \textbf{where}\\
&\qquad h_1\ a\ z && = Leaf\ z\\
&\qquad h_2\ i\ d\ z && = Join\ i\ d\\
&\qquad \psi\ z && = z+1
\end{aligned}
$$

b) Sea *cConst* la función que calcula la cantidad de constructores de un árbol.

$$
\begin{aligned}
&cConst && :: (Btree\ a, Int) \to Int\\
&cConst\ (Leaf\ a, z) && = z+1\\
&cConst\ (Join\ t\ u, z) && = cConst\ (t, 0) + cConst\ (u, z+1)
\end{aligned}
$$

Es posible expresar *cConst* como un *afold* donde:

$$
\begin{aligned}
&cConst = afold_{B_a}\ (h_1, h_2)\ \psi\ \psi'\\
&\quad \textbf{where}\\
&\qquad h_1\ a\ z && = z+1\\
&\qquad h_2\ i\ d\ z && = i+d\\
&\qquad \psi\ z && = 0\\
&\qquad \psi'\ z && = z+1
\end{aligned}
$$

3. Árboles binarios con información en los nodos

$$\textbf{data}\ Tree\ a = Empty \mid Node\ (Tree\ a)\ a\ (Tree\ a)$$

$$
\begin{aligned}
&afold_{T_a} :: (x \to c, a \to c \to c \to x \to c) \to (a \to x \to x) \to (a \to x \to x)\\
&\qquad\qquad \to (Tree\ a, x) \to c\\
&afold_{T_a}\ (h_1, h_2)\ \psi\ \psi' = f\\
&\quad \textbf{where}\\
&\qquad f\ (Empty, z) && = h_1\ z\\
&\qquad f\ (Node\ t\ e\ u, z) && = h_2\ e\ (f\ (t, \psi\ e\ z))\ (f\ (u, \psi'\ e\ z))\ z
\end{aligned}
$$

Sea la función que calcula la suma de los elementos de un árbol.

$$
\begin{aligned}
&sumElemTree && :: Num\ a \Rightarrow Tree\ a \to a\\
&sumElemTree\ a && = aSumElemTree\ (a, 0)\\
&aSumElemTree && :: Num\ a \Rightarrow (Tree\ a, a) \to a\\
&aSumElemTree\ (Empty, n) && = n\\
&aSumElemTree\ (Node\ i\ r\ d, n) && = aSumElemTree\ (i, 0) + aSumElemTree\ (d, r+n)
\end{aligned}
$$

Es posible expresar *aSumElemTree* como un *afold* donde:

$$
\begin{aligned}
aSumElemTree = {} & afold_{T_a}\ (id, h_2)\ \psi\ (+) \\
& \textbf{where} \\
& \quad h_2\ e\ i\ d\ a = i + d \\
& \quad \psi\ a\ b = 0
\end{aligned}
$$

□

Los anteriores ejemplos muestran una serie de funciones que pueden ser representadas en forma sencilla y directa en término de *afold*. Sin embargo es posible encontrar otras funciones que no son representables como *afold*, en forma directa, debido a la estructura de su definición. Este punto será tratado en la Sección 3.2.

A continuación se enumeran algunas leyes de fusión del operador *afold*, las cuales son relevantes en conexión con lo que va a ser presentado en el Capítulo 4; una lista completa de las leyes de *afold* se encuentra en [Par01, Par03].

Ley 3.1.7 (Afold Pure Fusion)**.**

$$f \circ h = h' \circ (F\ f\ \times\ id) \quad \Rightarrow \quad f \circ afold_F\ (h, \tau) = afold_F\ (h', \tau)$$

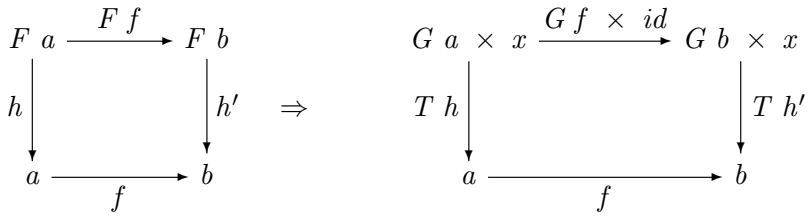
Ley 3.1.8 (Acid Rain: Afold-Fold Fusion)**.**

$$T : \forall\ a\ .\ (F\ a \to a) \to (G\ a\ \times\ x \to a) \quad \Rightarrow \quad fold_F h \circ afold_G\ (T\ in_F, \tau) = afold_G\ (T\ h, \tau)$$

Ley 3.1.9 (Morph-Afold Fusion)**.** *Para* $f : x \to x'$

$$F\ (id\ \times\ f) \circ \tau_a = \tau'_a \circ (id\ \times\ f) \quad \Rightarrow \quad afold_F\ (h, \tau') \circ (id\ \times\ f) = afold_F\ (h \circ (id\ \times\ f), \tau)$$

En la Ley 3.1.8, T representa un transformador entre F-álgebras y funciones de la forma $G\ a\ \times\ x \to a$. Su característica de naturalidad se refleja en la siguiente condición:



La Ley 3.1.9 relaciona dos acumulaciones cuyos acumuladores tienen diferentes tipos; su precondición establece una condición de coherencia entre los acumuladores. La prueba de estas leyes se encuentran en [Par01]

A partir de la posibilidad de representar acumulaciones en términos del operador *afold* es posible fusionar la composición de funciones que tienen acumulaciones como funciones productoras usando las leyes antes presentadas como se muestra en el siguiente ejemplo.

Ejemplo 3.1.10.

1. Sea la función *aheight* que calcula la altura de un árbol binario en dos pasos: primero reemplaza el valor de cada hoja por su profundidad en el árbol (*down*) y en un segundo paso se calcula el máximo del árbol (*maxtree*).

$$aheight\ t = maxtree\ (down\ (t, 0)).$$

Las funciones *maxtree* y *down* son las definidas en los Ejemplos 2.4.2 y 3.1.6 respectivamente. La función *aheight* produce un árbol binario como estructura de datos intermedia que puede ser eliminado por la aplicación de fusión.

Para ello como primer paso es conveniente visualizar que la función *down* genera la estructura intermedia a partir de los constructores del tipo árbol, lo que permite expresar el álgebra del *afold* en términos de un transformador:

$$\begin{aligned} &down = afold_{B_a}\ (T\ in_{B_a})\ succ\ succ \\ &\quad \textbf{where}\ succ\ n = n + 1 \end{aligned}$$

donde $T\ (h_1, h_2) = (k_1, k_2)$, tal que:

$$\begin{aligned} k_1\ a\ z &= h_1\ z \\ k_2\ i\ d\ z &= h_2\ i\ d \end{aligned}$$

Recordando la definición de *maxtree* en términos de *fold* dada en el Ejemplo 2.4.2, la función *aheight* se puede simplificar de la siguiente forma:

$$\begin{aligned} & aheight \\ = & \quad \{\ \text{Definición de } aheight\ \} \\ & maxtree \circ down \\ = & \quad \{\ \text{Definición de } maxtree \text{ y } down\ \} \\ & fold_{B_{Nat}}\ (id, max) \circ afold_{B_A}\ (T\ in_{B_a})\ succ\ succ \\ = & \quad \{\ \text{Ley 3.1.8}\ \} \\ & afold_{B_{Nat}}\ (T\ (id, max))\ succ\ succ \\ = & \quad \{\ \text{Definición de } T\ \} \\ & afold_{B_{Nat}}\ (j1, j2)\ succ\ succ \\ & \quad \{\ \text{donde } j1\ a\ z = z, j2\ i\ d\ z = max\ i\ d\ \} \end{aligned}$$

El *afold* que resulta corresponde a la siguiente definición recursiva:

$$\begin{aligned} f\ (Leaf\ a, z) &= z \\ f\ (Join\ i\ d, z) &= max\ (f\ (i, z + 1))\ (f\ (d, z + 1)) \end{aligned}$$

2. Sea la función $ld = leaves \circ down$, siendo *leaves* y *down* las funciones definidas en los Ejemplos 2.4.2 y 3.1.6 respectivamente.

$$leaves = fold_{B_a}\ (wrap, (+\!\!+))$$

$$\begin{aligned} &down = afold_{B_a}\ (T\ in_{B_a})\ succ\ succ \\ &\quad \textbf{where}\ succ\ n = n + 1 \end{aligned}$$

donde $T\ (h_1, h_2) = (k_1, k_2)$, tal que:

$$\begin{aligned} k_1\ a\ z &= h_1\ z \\ k_2\ i\ d\ z &= h_2\ i\ d \end{aligned}$$

Aplicando fusión se obtiene que:

$$
\begin{aligned}
& fold_{B_{Nat}}\ (wrap, (+\!\!+)) \circ afold_{B_a}\ (T\ in_{B_a})\ succ\ succ \\
= & \quad \{\ \text{Ley 3.1.8}\ \} \\
& afold_{B_{Nat}}\ (T\ (wrap, (+\!\!+)))\ succ\ succ \\
= & \quad \{\ \text{Definición de } T\ \} \\
& afold_{B_{Nat}}\ (j1, j2)\ succ\ succ \\
& \quad \{\ \text{donde } j1\ a\ z = wrap\ z, j2\ i\ d\ z = i +\!\!+ d\ \}
\end{aligned}
$$

que corresponde a la siguiente definición recursiva:

$$
\begin{aligned}
ld\ (Leaf\ a, z) &= wrap\ z \\
ld\ (Join\ i\ d, z) &= ld\ (i, z+1) +\!\!+ ld\ (d, z+1)
\end{aligned}
$$

□

3.2. Extensión del operador

Si bien la definición original del operador *afold* permite capturar el comportamiento de una amplia gama de funciones con acumulaciones, existen casos que no son cubiertos.

Por ejemplo, dada la siguiente definición de la función que cuenta la cantidad de elementos de una lista sin considerar los repetidos:

$$
\begin{aligned}
&contar && :: Eq\ a \Rightarrow List\ a \to Int \\
&contar\ xs && = acontar\ (xs, 0) \\
\\
&acontar && :: Eq\ a \Rightarrow (List\ a, Int) \to Int \\
&acontar\ (Nil, n) && = n \\
&acontar\ (Cons\ x\ xs, n) && = acontar\ (xs, gcon\ x\ n\ xs) \\
& && \quad \textbf{where}\ gcon && :: Eq\ a \Rightarrow a \to Int \to List\ a \to Int \\
& && \qquad gcon\ x\ n\ xs \mid elem\ x\ xs = n \\
& && \qquad\qquad \mid otherwise = n + 1
\end{aligned}
$$

Con esta definición la función *acontar* no puede representarse directamente en términos de *afold* debido a que es necesario considerar la cola de la lista para la modificación del acumulador. Es posible hacerlo para el caso de una definición similar en la que se incorpora la cola de la lista como parte del acumulador:

$$
\begin{aligned}
&acontar' && :: Eq\ a \Rightarrow (List\ a, (Int, List\ a)) \to Int \\
&acontar'\ (Nil, (n, \_)) && = n \\
&acontar'\ (Cons\ x\ xs, (n, ys)) && = acontar'\ (xs, gcon'\ x\ (n, ys)) \\
& && \quad \textbf{where}\ gcon'\ x\ (n, ys) \mid elem\ x\ ys = (n, tl\ ys) \\
& && \qquad\qquad\qquad\qquad \mid otherwise = (n + 1, tl\ ys)
\end{aligned}
$$

$$
\begin{array}{ll}
tl & :: List\ a \rightarrow List\ a \\
tl\ Nil & = Nil \\
tl\ (Cons\ x\ xs) & = xs
\end{array}
$$

la expresión en términos del operador *afold* de esta definición es:

$$
\begin{array}{l}
acontar' :: Eq\ a \Rightarrow (List\ a, (Int, List\ a)) \rightarrow Int \\
acontar' = afold_{L_a}\ (fst, mdl)\ gcon' \\
\qquad \textbf{where} \\
\qquad\quad gcon'\ x\ (n, ys) \mid elem\ x\ ys = (n, tl\ ys) \\
\qquad\qquad\qquad\qquad \mid otherwise = (n+1, tl\ ys)
\end{array}
$$

En consecuencia, la definición de la función *contar* original puede expresarse como:

$$contar\ xs = acontar'\ (xs, (0, tl\ xs))$$

□

Otra forma de lograr representar funciones con definiciones de este estilo en términos de *afold* es modificando la definición del operador de forma tal que al momento de manipular el acumulador sea posible considerar tanto el valor contenido en el nodo de la estructura como sus subestructuras. La definición del operador en sí misma no cambia siendo la única modificación la eliminación de la restricción de naturalidad sobre la función de acumulación τ. O sea, el cambio queda encapsulado en las funciones consideradas apropiadas para la acumulación.

Se nombrará al nuevo operador como *afold extendido* (se lo denotará *afoldE*) en el sentido de que extiende el conjunto de funciones para las cuales puede capturar su comportamiento con respecto a las capturadas por el operador *afold*. A continuación se presenta la definición del operador extendido para listas y árboles binarios de forma tal de visualizar explícitamente el cambio en el dominio de las funciones de acumulación.

Ejemplo 3.2.1.

1. Listas

$$
\begin{array}{l}
afoldE_{L_a} : (x \rightarrow c, a \rightarrow c \rightarrow x \rightarrow c) \rightarrow (a \rightarrow x \rightarrow List\ a \rightarrow x) \rightarrow (List\ a, x) \rightarrow c \\
afoldE_{L_a}\ (h_1, h_2)\ \psi = f \\
\quad \textbf{where} \\
\quad\quad f\ (Nil, z) = h_1\ z \\
\quad\quad f\ (Cons\ a\ as, z) = h_2\ a\ (f\ (as, \psi\ a\ z\ as))\ z
\end{array}
$$

En la definición anterior queda en evidencia la modificación realizada con respecto a la definición original. La función de acumulación ψ manipula el elemento de la lista, el acumulador y como novedad también manipula la cola de la lista.

a) En particular,

$acontar = afoldE_{L_a}\ (id, mdl)\ gcon$, donde $mdl\ a\ b\ c = b$

2. Árboles binarios con información en las hojas

$$
\begin{aligned}
&afoldE_{B_a} :: (a \to x \to c, c \to c \to x \to c) \to (x \to Btree\ a \to Btree\ a \to x) \\
&\qquad \to (x \to Btree\ a \to Btree\ a \to x) \to (Btree\ a, x) \to c
\end{aligned}
$$

$$
\begin{aligned}
&afoldE_{B_a}\ (h_1, h_2)\ \psi\ \psi' = f \\
&\quad \textbf{where} \\
&\qquad f\ (Leaf\ a, z) \quad = h_1\ a\ z \\
&\qquad f\ (Join\ t\ u, z) = h_2\ (f\ (t, \psi\ z\ t\ u))\ (f\ (u, \psi'\ z\ t\ u))\ z
\end{aligned}
$$

En este caso la modificación realizada sobre el operador *afold* queda en evidencia en las funciones ψ y ψ' que manipulan tanto el valor del acumulador como los subárboles en cada paso de la recursión.

3. Árboles binarios con información en los nodos

$$
\begin{aligned}
&afoldE_{T_a} :: (x \to c, a \to c \to c \to x \to c) \to (a \to x \to Tree\ a \to Tree\ a \to x) \to \\
&\quad (a \to x \to Tree\ a \to Tree\ a \to x) \to (Tree\ a, x) \to c
\end{aligned}
$$

$$
\begin{aligned}
&afoldE_{T_a}\ (h_1, h_2)\ \psi\ \psi' = f \\
&\quad \textbf{where} \\
&\qquad f\ (Empty, z) \quad\ = h_1\ z \\
&\qquad f\ (Node\ t\ e\ u, z) = h_2\ e\ (f\ (t, \psi\ e\ z\ t\ u))\ (f\ (u, \psi'\ e\ z\ t\ u))\ z
\end{aligned}
$$

Nuevamente es posible visualizar la modificación realizada en los argumentos de las funciones ψ y ψ'.

Sea la siguiente definición de la función *sumDif* que dado un árbol binario, suma los valores contenidos en el mismo que no coinciden con el valor de ninguna de las raíces de sus subárboles inmediatos:

$$
\begin{aligned}
&sumDif && :: Num\ a \Rightarrow (Tree\ a, a) \to a \\
&sumDif\ (Empty, n) && = n \\
&sumDif\ (Node\ i\ a\ d, n) && = sumDif\ (i, 0) + sumDif\ (d, acSD\ a\ n\ i\ d)
\end{aligned}
$$

$$
\begin{aligned}
&acSD :: Num\ a \Rightarrow a \to a \to Tree\ a \to Tree\ a \to a \\
&acSD\ a\ n\ Empty\ Empty && && = a + n \\
&acSD\ a\ n\ Empty\ (Node\ i\ x\ d) && \mid x == a && = n \\
&&& \mid otherwise && = a + n \\
&acSD\ a\ n\ (Node\ i\ x\ d)\ Empty && \mid x == a && = n \\
&&& \mid otherwise && = a + n \\
&acSD\ a\ n\ (Node\ i\ x\ d)\ (Node\ i'\ x'\ d') \\
&&& \mid (a == x) \vee (a == x') && = n \\
&&& \mid otherwise && = a + n
\end{aligned}
$$

Esta función puede expresarse en términos de *afoldE* de la siguiente forma:

$$sumDif = afoldE_{T_a}\ (id, h_2)\ \psi\ acSD$$

donde $h_2\ a\ ri\ rd\ ac = ri + rd$ y $\psi\ a\ z\ t\ u = 0$. □

Como consecuencia de la eliminación de la restricción de naturalidad sobre la función de acumulación (τ),las leyes de fusión de *afold* que se mantienen vigentes para *afoldE* son aquellas que no requieren de la propiedad de naturalidad para su validez. Por ejemplo, las tres leyes de fusión de *afold* mostradas en la Sección 3.1 mantienen su vigencia. El resto de las leyes que aparecen en [Par03] se comportan en forma variada.

A modo de ilustración se enuncian a continuación las leyes de fusión de *afoldE* para el caso particular de listas y árboles binarios con información en las hojas debido a que serán utilizadas en secciones posteriores.

Ley 3.2.2 (AfoldE Pure Fusion)**.**

1. *Listas*

$$\begin{array}{l} \quad f\ (h_2\ a\ b\ c) = h_2'\ a\ (f\ b)\ c \wedge f\ (h_1\ x) = h_1'\ x \\ \Rightarrow \\ \quad f \circ afoldE_{L_a}\ (h_1, h_2)\ \psi\ as = afoldE_{L_a}\ (h_1', h_2')\ \psi\ as \end{array}$$

2. *Árboles binarios*

$$\begin{array}{l} \quad f\ (h_2\ i\ d\ p) = h_2'\ (f\ i)\ (f\ d)\ p \wedge f\ (h_1\ a\ x) = h_1'\ a\ x \\ \Rightarrow \\ \quad f \circ afoldE_{B_a}\ (h_1, h_2)\ \psi\ \psi' = afoldE_{B_a}\ (h_1', h_2')\ \psi\ \psi' \end{array}$$

Ley 3.2.3 (Acid Rain: AfoldE-Fold Fusion)**.**

1. *Listas*

$$\begin{array}{l} \quad T :: \forall\ b\ .\ (F\ b \to b) \to ((L_a\ b, x) \to b) \\ \Rightarrow \\ \quad fold_F\ (h) \circ afoldE_{L_a}\ (T\ in_F)\ \psi = afoldE_{L_a}\ (T\ h)\ \psi \end{array}$$

2. *Árboles binarios*

$$\begin{array}{l} \quad T :: \forall\ b\ .\ (F\ b \to b) \to ((B_a\ b, x) \to b) \\ \Rightarrow \\ \quad fold_F\ (h) \circ afoldE_{B_a}\ (T\ in_F)\ \psi\ \psi' = afoldE_{B_a}\ (T\ h)\ \psi\ \psi' \end{array}$$

Ley 3.2.4 (Morph-AfoldE Fusion)**.** *Para* $f : x \to x'$

1. *Listas*

$$\begin{array}{l} \quad f\ (\psi\ a\ b\ c) = \psi'\ a\ (f\ b)\ c \\ \Rightarrow \\ \quad afoldE_{L_a}\ (h_1, h_2)\ \psi'\ (as, f\ z) = afoldE_{L_a}\ (h_1', h_2')\ \psi\ (as, z) \\ \quad\quad \textbf{where} \\ \quad\quad h_1'\ x = h_1\ (f\ x) \\ \quad\quad h_2'\ a\ b\ c = h_2\ a\ b\ (f\ c) \end{array}$$

2. Árboles binarios

$$
\begin{array}{l}
\quad f\ (\psi_1\ z\ t\ u) = \psi_2\ (f\ z)\ t\ u \wedge f\ (\psi_1'\ z\ t\ u) = \psi_2'\ (f\ z)\ t\ u \\
\Rightarrow \\
\quad afoldE_{B_a}\ (h_1, h_2)\ \psi_2\ \psi_2'\ (b, f\ z) = afoldE_{B_a}\ (h_1', h_2')\ \psi_1\ \psi_1'\ (b, z) \\
\quad\quad \textbf{where} \\
\quad\quad h_1'\ a\ z = h_1\ a\ (f\ z) \\
\quad\quad h_2'\ i\ d\ z = h_2\ i\ d\ (f\ z)
\end{array}
$$

Nuevamente estas leyes pueden ser utilizadas para fusionar composiciones de funciones donde la función productora es expresable en términos del operador *afoldE*, como se muestra en el siguiente ejemplo.

Ejemplo 3.2.5.

Sean las siguientes funciones:

$$
\begin{array}{ll}
biggers & :: (Num\ a, Ord\ a) \Rightarrow List\ a \rightarrow List\ a \\
biggers\ xs & = abiggers\ (xs, Nil) \\
\\
abiggers & :: (Num\ a, Ord\ a) \Rightarrow (List\ a, List\ a) \rightarrow List\ a \\
abiggers\ (Nil, ys) & = ys \\
abiggers\ (Cons\ a\ as, ys) & = abiggers\ (as, phi\ a\ ys\ as) \\
 & \quad \textbf{where} \\
 & \quad\quad phi\ a\ ys\ as = \textbf{if}\ a > sum\ as\ \textbf{then}\ a : ys\ \textbf{else}\ ys \\
\\
sum & :: Num\ a \Rightarrow List\ a \rightarrow a \\
sum\ Nil & = 0 \\
sum\ (Cons\ a\ as) & = a + sum\ as
\end{array}
$$

donde la función *biggers* retorna una lista con aquellos elementos de la lista de entrada que son mayores que la suma de los elementos que los que siguen en la lista. Este cálculo se realiza mediante una función acumulativa *abiggers*. Por ejemplo:
$abiggers\ [1, 107, 22, 15, 3, 11, 9, 0]\ [\,] = [9, 11, 107]$.

La función *abiggers* se puede expresar en términos del operador *afoldE* de la siguiente forma:

$abiggers = afoldE_{L_a}\ (id, mdl)\ phi$, donde $mdl\ a\ b\ c = b$.

A partir de esta representación de la función *abiggers* es posible utilizar las leyes de fusión para simplificar, por ejemplo, la composición $sum \circ abiggers$:

$$
\begin{array}{ll}
 & sum \circ abiggers \\
= & \quad \{ \text{ Definición de } abiggers \ \} \\
 & sum \circ afoldE_{L_a}\ (id, mdl)\ phi \\
= & \quad \{ \text{ pure fusion } \} \\
 & afoldE_{L_a}\ (sum, mdl)\ phi \\
= & \quad \{ \text{ morph-afoldE fusion } \} \\
 & afoldE_{L_a}\ (id, mdl)\ \psi' \circ (id\ \times\ sum) \\
 & \textbf{where}\ \psi'\ a\ b\ c = \textbf{if}\ a > sum\ c\ \textbf{then}\ a + b\ \textbf{else}\ b
\end{array}
$$

Por lo tanto $sum \circ abiggers = f \circ (id\ \times\ sum)$ donde f tiene la siguiente definición recursiva:

$$
\begin{array}{ll}
f & :: (Num\ a, Ord\ a) \Rightarrow (List\ a, a) \rightarrow a \\
f\ (Nil, y) & = xs \\
f\ (Cons\ x\ xs, y) & = f\ (xs, \psi'\ x\ y\ xs) \\
 & \quad \textbf{where}\ \psi'\ a\ b\ c = \textbf{if}\ a > sum\ c\ \textbf{then}\ a + b\ \textbf{else}\ b
\end{array}
$$

□

3.3. El operador *foldl* y su extensión

Muchos casos prácticos de acumulaciones sobre listas son instancias del operador *foldl* [Bir98]. Este operador, de amplio uso en programación funcional, tiene la característica de manipular un acumulador y de estar definido por una recursión de cola ("tail recursion"). En esta sección se analiza el operador *foldl* desde la óptica de *afold* y se ve la formulación de su correspondiente extensión. Por último, se presenta una aplicación.

La definición del operador *foldl* es la siguiente:

$$
\begin{array}{ll}
foldl & :: (a \rightarrow b \rightarrow b) \rightarrow (List\ a, b) \rightarrow b \\
foldl\ (\oplus) & = f \\
\quad \textbf{where}\ f\ (Nil, z) & = z \\
\quad\quad f\ (Cons\ a\ as, z) & = f\ (as, a\ \oplus\ z)
\end{array}
$$

Si bien esta definición es levemente diferente a la tradicional, en el orden de sus parámetros y el hecho de que no se encuentra currificada, se va a considerar por compatibilidad con el operador *afold*.

Analizando la definición de *foldl* es posible identificar rápidamente que es un caso particular del operador *afold* y que varias de sus conocidas propiedades surgen directamente de las leyes de fusión de *afold*. La definición en términos de *afold* es:

$$
\begin{array}{l}
foldl\ (\oplus) \stackrel{def}{=} afold_{L_a}\ (id, mdl)\ (\oplus) \\
\quad \textbf{where}\ mdl\ a\ b\ c = b
\end{array}
$$

El hecho de que *foldl* es una definición recursiva de cola se refleja en que la función *mdl* es simplemente la proyección de la llamada recursiva.

Ejemplo 3.3.1.

1. Las funciones *sumaAc* y *aLargo* presentadas en el Ejemplo 3.1.6 pueden expresarse en término del operador *foldl* como sigue:

$$\begin{aligned} &sumaAc = foldl\ (+) \\ &aLargo\ \ = foldl\ (\oplus)\ \textbf{where}\ a\ \oplus\ b = b + 1 \end{aligned}$$

2. Dada la siguiente definición:

$$\begin{aligned} &pack\ (Nil, z) &&= z \\ &pack\ (Cons\ a\ as, z) &&= pack\ (as, z * 10 + a) \end{aligned}$$

que dada una lista de dígitos obtiene el número que representa en notación decimal, la misma puede expresarse en términos del operador *foldl* como:

$$pack = foldl\ (\oplus)\ \textbf{where}\ a\ \oplus\ z = a + z * 10$$

□

Es posible instanciar las leyes de fusión presentadas en la Sección 3.1 para este caso particular del operador *afold*, obteniendo expresiones en términos del operador *foldl* en la medida de lo posible.

Ley 3.3.2 (Foldl Pure Fusion)**.**

$$f \circ foldl\ (\oplus) \quad = \quad afold_{L_a}\ (f, mdl)\ (\oplus)$$

Ley 3.3.3 (Morph-Foldl Fusion)**.** *Para* $f : x \to x'$

$$\begin{aligned} & f\ (a\ \oplus\ b) = a\ \oplus'\ f\ b \\ \Rightarrow & \\ & foldl\ (\oplus')\ (as, f\ z) = afold_{L_a}\ (f, mdl)\ (\oplus)\ (as, z) \end{aligned}$$

Las expresión particular del álgebra del *afold* que define al operador *foldl* tiene como resultado que la Ley 3.3.2 no requiera de hipótesis. Por otro lado, dicha expresión del álgebra hace que no tenga sentido el planteo de una ley del tipo Acid-Rain para *foldl* ya que el álgebra no es expresable en términos de un transformador.

Ejemplo 3.3.4. Sea la siguiente definición:

$$\begin{aligned} &aconcat &&:: (List\ (List\ a), List\ a) \to List\ a \\ &aconcat\ (Nil, xs) &&= xs \\ &aconcat\ (Cons\ a\ as, xs) &&= aconcat\ (as, a +\!\!+ xs) \end{aligned}$$

que corresponde a una variante acumulativa de la función *concat*. Esta función es posible expresarla en término del operador *foldl* de la siguiente forma: $aconcat = foldl\ (+\!\!+)$.

A partir de esta definición interesa calcular la siguiente composición: $f = length \circ aconcat$.

$$
\begin{array}{ll}
 & length \circ aconcat \\
= & \quad \{\ \text{Definición de } aconcat\ \} \\
 & length \circ foldl\ (+\!\!+) \\
= & \quad \{\ \text{Ley 3.3.2}\ \} \\
 & afold_{L_a}\ (length, mdl)\ (+\!\!+) \\
= & \quad \{\ \text{Ley 3.3.3 con } as\ \oplus'\ n = length\ as + n\ \} \\
 & foldl\ (\oplus') \circ (id\ \times\ length)
\end{array}
$$

Para la correcta aplicación de la ley 3.3.3 se debe cumplir que:

$$length\ (as +\!\!+ xs) = as\ \oplus'\ length\ xs = add\ (length\ as, length\ xs)$$

lo cual es sencillo de probar. □

En el ejemplo anterior la aplicación sucesiva de las leyes Foldl Pure Fusion y Morph Foldl Fusion permitió mover la función posterior al *foldl* en la composición a la posición del acumulador realizando únicamente un pequeño cambio en la función de acumulación. Si miramos la expresión inicial y final de la derivación anterior vemos que es posible expresar la composición de una función con un *foldl* en términos de otro *foldl* sin necesidad de mencionar el operador *afold*. A partir de esta situación se propone explicitar una ley que combina las leyes Foldl Pure Fusion y Morph Foldl Fusion.

Ley 3.3.5 (Foldl - Regla Combinada)**.** *Para* $f : x \to x'$

$$
\begin{array}{ll}
 & f\ (a\ \oplus\ b) = a\ \oplus'\ f\ b \\
\Rightarrow & \\
 & f \circ foldl\ (\oplus)\ (as, z) = foldl\ (\oplus')\ (as, f\ z)
\end{array}
$$

Prueba:

$$
\begin{array}{ll}
 & f \circ foldl\ (\oplus) \\
= & \quad \{\ \text{Ley 3.3.2}\ \} \\
 & afold_{L_a}\ (f, mdl)\ (\oplus) \\
= & \quad \{\ \text{Ley 3.3.3}\ \} \\
 & foldl\ (\oplus')\ as\ (f\ z)
\end{array}
$$

□

Como consecuencia de la combinación realizada se obtuvo una regla expresada totalmente en términos del operador *foldl* sin tener que recurrir al caso más general del operador *afold*

sobre las listas como sí sucede en las leyes anteriores. Esta ley se corresponde con el Teorema de Fusión sobre *foldl* presentado en [Bir98] y una derivación de esta ley en términos del operador *afold* se encuentra en [Jas04].

La fusión presentada en el ejemplo 3.3.4 puede reescribirse en una forma más concisa utilizando esta nueva ley:

$$
\begin{array}{ll}
 & length \circ aconcat \\
= & \quad \{ \text{ Definición de } aconcat \ \} \\
 & length \circ foldl\ (+\!\!+) \\
= & \quad \{ \text{ Ley 3.3.5, } as \oplus' \ n = length\ as + n \ \} \\
 & foldl\ (\oplus') \circ (id \times length)
\end{array}
$$

3.3.1. El operador *foldl* extendido

Con la misma idea de extensión realizada al operador *afold* es posible obtener una extensión del operador *foldl* que permite utilizar las subestructuras en la actualización del acumulador. Su definición es la siguiente:

$$
\begin{array}{l}
foldlE :: (a \to b \to List\ a \to b) \to (List\ a, b) \to b \\
foldlE\ \psi = f \\
\quad \textbf{where} \\
\qquad f\ (Nil, z) \qquad\quad = z \\
\qquad f\ (Cons\ a\ as, z) = f\ (as, \psi\ a\ z\ as)
\end{array}
$$

Como consecuencia de haber realizado la extensión del operador *foldl* con la misma idea que la utilizada en la extensión del operador *afold* y siendo *foldl* un caso particular de este, el nuevo operador se puede expresar en términos de la extensión de *afold*:

$$
\begin{array}{l}
foldlE\ \psi \stackrel{def}{=} afoldE_{L_a}\ (id, mdl)\ \psi \\
\quad \textbf{where}\ mdl\ a\ b\ c = b
\end{array}
$$

Ejemplo 3.3.6.

1. La función *contar* presentada en la Sección 3.2 corresponde a un caso de *foldlE*:

   $$contar = foldlE\ acSD$$

2. La función *abiggers* presentada en el Ejemplo 3.2.5 corresponde también a un caso de *foldlE*:

   $$
   \begin{array}{l}
   abiggers = foldlE\ \psi \\
   \quad \textbf{where}\ \psi\ a\ ys\ as = \textbf{if}\ a > sum\ as\ \textbf{then}\ a : ys\ \textbf{else}\ ys
   \end{array}
   $$

3. Sea la siguiente definición:

$$\begin{array}{ll} menResto & :: Ord\ a \Rightarrow (List\ a, List\ Bool) \rightarrow List\ Bool \\ menResto\ (Nil, ys) & = ys \\ menResto\ (Cons\ a\ as, ys) & = menResto\ (as, ys +\!\!+ (Cons\ (all\ (>a)\ as)\ Nil)) \end{array}$$

donde *all p as* indica si todos los elementos de la lista *as* cumplen con la condición P. Dada una lista de elementos, *menResto* devuelve una lista de booleanos donde el elemento en la posición i indica si el elemento en esa posición en la lista de entrada es menor que el resto de los elementos que le siguen en la lista. Por ejemplo,

$$menResto\ [1,5,2,3,4,7]\ [\,] = [\,True, False, True, True, True, True]$$

Es posible expresar la función *menResto* en términos del operador *foldlE*:

$$\begin{array}{l} menResto = foldlE\ \psi \\ \qquad\qquad \textbf{where}\ \psi\ a\ ys\ as = ys +\!\!+ (Cons\ (all\ (>a)\ as)\ Nil) \end{array}$$

□

A continuación se presentan algunas de las leyes de fusión para el caso particular del operador *foldlE*.

Ley 3.3.7 (FoldE Pure Fusion)**.**

$$f \circ foldlE\ \psi \quad = \quad afoldE_{L_a}\ (f, mdl)\ \psi$$

Ley 3.3.8 (Morph-FoldE Fusion)**.** *Para* $f : x \rightarrow x'$

$$\begin{array}{ll} & f\ (\psi\ a\ b\ c) = \psi'\ a\ (f\ b)\ c \\ \Rightarrow & \\ & foldlE\ \psi'\ (as, f\ z) = afoldE_{L_a}\ (f, mdl)\ \psi\ (as, z) \end{array}$$

Al igual que para el operador *foldl* es de utilidad en este caso establecer una regla combinada entre las leyes anteriores. La prueba de esta regla es análoga a la realizada para la regla combinada para el operador *foldl*.

Ley 3.3.9 (FoldE - Regla combinada)**.** *Para* $f : x \rightarrow x'$

$$\begin{array}{ll} & f\ (\psi\ a\ b\ c) = \psi'\ a\ (f\ b)\ c \\ \Rightarrow & \\ & foldlE\ \psi'\ (as, f\ z) = f \circ foldlE\ \psi\ (as, z) \end{array}$$

Ejemplo 3.3.10.

1. Sea la siguiente definición de la función *cantTrue* que cuenta la cantidad de elementos *True* en una lista de booleanos:

$$\begin{aligned} &cantTrue && :: List\ Bool \to Int \\ &cantTrue\ Nil && = 0 \\ &cantTrue\ (Cons\ a\ as) && = \textbf{if}\ a\ \textbf{then}\ 1 + cantTrue\ as\ \textbf{else}\ cantTrue\ as \end{aligned}$$

   La composición de funciones $cantTrue \circ menResto$, siendo *menResto* la función definida en el Ejemplo 3.3.6, es posible fusionarla aplicando la Ley 3.3.9:

$$\begin{aligned} & cantTrue \circ menResto \\ = & \quad \{\ \text{Definición de } menResto\ \} \\ & cantTrue \circ foldlE\ \psi \\ = & \quad \{\ \text{Ley 3.3.9}\ \} \\ & foldlE\ \psi' \circ (id\ \times\ cantTrue) \end{aligned}$$

   donde se debe cumplir la siguiente condición $cantTrue\ (\psi\ a\ b\ c) = \psi'\ a\ (cantTrue\ b)\ c$. Recordando que $\psi\ a\ b\ c = b +\!\!+ (Cons\ (all\ (>a)\ c)\ Nil)$ es posible determinar una expresión para la función ψ':

$$\psi'\ a\ b\ c = \textbf{if}\ (all\ (>a)\ c)\ \textbf{then}\ 1 + b\ \textbf{else}\ b$$

2. Es posible reproducir la fusión realizada en el Ejemplo 3.2.5 tomando como punto de partida la definición de la función de *abiggers* en términos del operador *foldlE* (Ejemplo 3.3.6) y aplicando la Ley 3.3.9.

$$\begin{aligned} & sum \circ abiggers \\ = & \quad \{\ \text{Definición de } abiggers\ \} \\ & sum \circ foldlE\ phi \\ = & \quad \{\ \text{Ley 3.3.9}\ \} \\ & foldlE\ \psi' \circ (id\ \times\ sum) \\ & \psi'\ a\ b\ c = \textbf{if}\ a > sum\ c\ \textbf{then}\ a + b\ \textbf{else}\ b \end{aligned}$$

□

3.3.2. Aplicación: fisión de programas

En la mayoría de los casos las leyes de fusión son utilizadas en la dirección en que dos componentes son fusionadas para obtener una nueva función con igual comportamiento a la composición de las funciones originales. Sin embargo también es posible utilizarlas en la dirección contraria permitiendo descomponer una función en la composición de dos funciones; a este proceso se le llama **fisión**.

```
#include <stdio.h>
#define  IN   1  /* inside a word */
#define  OUT  0  /* outside a word */

/* count lines, words, and characteres in input */

main ()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0;
    while  ( ( c = getchar() ) != EOF) {
           ++nc;
           if (c == '\n')
               ++ nl;
           if (c == ' ' || c == '\n' || c == '\t')
               state = OUT;
           else  if (state == OUT) {
                     state = IN;
                     ++nw;
           }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Figura 3.1: Programa wc de Kernighan y Richie

Al realizar fusión se busca ganar en eficiencia, pero en muchos casos se pierde en claridad. En cambio al realizar fisión se busca ganar en comprensión de la función original obteniendo componentes que muestren más claramente el comportamiento. Es un proceso típico de reingeniería que en general presenta mayores dificultades que el proceso de fusión, ya que implica la introducción de componentes y la selección de éstas no es sencilla.

En esta sección se presenta una aplicación en este sentido que se basa en el ejemplo usado por Gibbons [Gib06]. Se muestra cómo se puede llevar adelante el proceso de fisión con las herramientas formales descriptas en este capítulo realizando una manipulación totalmente análoga a la efectuada por Gibbons.

Gibbons considera la función *wc* de Unix que permite contar la cantidad de líneas, palabras y caracteres contenidos en un texto. Su definición en C dada por Kernighan y Richie [KR88] se encuentra en la Figura 3.1. Del programa descripto en la Figura 3.1 se considera únicamente la parte que permite contar las palabras, por lo que se realiza una primera descomposición del mismo mostrada en la Figura 3.2.

Como punto de partida para el proceso de fisión se considera un programa funcional equivalente al programa C de la Figura 3.2 donde el loop se convierte en una función que realiza recursión de cola y utiliza el operador *foldl* para su definición:

$$
\begin{array}{ll}
wc_1 & :: List\ Char \rightarrow Int \\
wc_1\ xs & = fst\ (foldl\ step_1\ (xs, (0, False))) \\
\\
step_1 & :: Char \rightarrow (Int, Bool) \rightarrow (Int, Bool) \\
step_1\ c\ (n, b) \mid blank\ c & = (n, False)
\end{array}
$$

```
#include <stdio.h>
#define  IN   1  /* inside a word */
#define  OUT  0  /* outside a word */

int  blank(int c) {
     return (c == ' ' || c == '\n' || c == '\t');
}
/* count words in input */
main ()
{
    int c, nw, state;
    state = OUT;
    nw = 0;
    while  ( ( c = getchar() ) != EOF) {
           if (blank(c))
               state = OUT;
           else  if (state == OUT) {
                     state = IN;
                     ++nw;
           }
    }
    printf("%d\n", nw);
}
```

Figura 3.2: Extracción de contar palabras de *wc*

$$
\begin{array}{ll}
step_1\ c\ (n, True) & = (n, True) \\
step_1\ c\ (n, False) & = (n+1, True) \\
\\
blank\ c & = (c == \texttt{' '}) \vee (c == \texttt{'\textbackslash n'}) \vee (c == \texttt{'\textbackslash t'})
\end{array}
$$

La función *foldl* $step_1$ opera sobre un acumulador formado por un par tal que en la primera componente se calcula la cantidad de palabras (lo que será devuelto por *wc* a través de la función *fst*) y en la segunda componente se modela una noción de estado que indica si se está en el medio de una palabra (*True*), o en un intervalo entre palabras (*False*). El acumulador se inicializa con el par $(0, False)$ indicando que aún no se ha contado ninguna palabra y no se está en el medio de una palabra.

A partir de esta definición Gibbons construye diferentes implementaciones de wc_1, todas con el mismo comportamiento pero trabajando sobre diferentes estructuras, obteniendo en todas ellas una descomposición de wc_1 que involucra la función *length*. Dichas implementaciones permiten diferentes niveles de comprensión del comportamiento de *wc*. Para realizar este proceso Gibbons trabaja sobre la estructura de datos *snoc* (lista invertida) y el operador *folds* que permite realizar recursión de cola sobre este tipo de datos:

$$
\begin{array}{l}
\textbf{data}\ \langle a \rangle = SNil \mid Snoc\ \langle a \rangle\ a \\
\\
folds \quad :: (b \to a \to b) \to (\langle a \rangle, b) \to b \\
folds\ (\oplus) = f \\
\quad \textbf{where}\ f\ (SNil, e) \quad = e \\
\qquad\qquad f\ (Snoc\ xs\ a, e) = f\ (xs, e)\ \oplus\ a
\end{array}
$$

Para manipular funciones definidas en términos de *folds* se puede utilizar su ley de fusión:

$$h\ (b\ \oplus\ a) = h\ b\ \oplus'\ a \quad \Rightarrow \quad h \circ \mathit{folds}\ (\oplus)\ (xs, e) = \mathit{folds}\ (\oplus')\ (xs, h\ e)$$

Nuestro desarrollo va a ser análogo al de Gibbons, pero con la diferencia de que nos vamos a basar en el operador *foldl* en lugar de *folds*. Como en [Gib06] vamos a conseguir descomponer wc_1 en términos de otras funciones más sencillas solo que ahora usando leyes de *foldl*.

Dado que la definición de wc_1 involucra el cálculo del largo de una estructura, como primer paso en el proceso de fisión se extrae de dicha definición un factor que involucra la función *length*. En este sentido se reescribe wc_1 como:

$$\begin{aligned} wc_1\ xs &= \mathit{fst}\ (wc'_1\ (xs, ([\,], \mathit{False}))) \\ wc'_1\ (xs, (as, bl)) &= \mathit{foldl}\ \mathit{step}_1\ (xs, f\ (as, bl)) \\ f\ (zs, bl) &= (\mathit{length}\ zs, bl) \end{aligned}$$

Si se observa la expresión de wc'_1 es posible reconocer una expresión como la presentada en la ley combinada asociada al operador *foldl* (Ley 3.3.5). Por lo tanto se utiliza esta ley de forma de factorizar la expresión de wc'_1 en dos funciones:

$$\begin{aligned} & wc'_1\ (xs, (as, bl)) \\ = & \quad \{\ \text{Definición de } wc'_1\ \} \\ & \mathit{foldl}\ \mathit{step}_1\ (xs, f\ (as, bl)) \\ = & \quad \{\ \text{Foldl - Regla combinada}\ \} \\ & f \circ \mathit{foldl}\ g\ (xs, (as, bl)) \end{aligned}$$

La aplicación de la ley lleva a la introducción de una nueva función:

$$g :: \mathit{Char} \to (\mathit{List}\ (\mathit{List}\ \mathit{Char}), \mathit{Bool}) \to (\mathit{List}\ (\mathit{List}\ \mathit{Char}), \mathit{Bool})$$

En este punto es necesario recurrir a un proceso creativo, pero ahora mucho más limitado, dado que hay mayor información aportada por las condiciones que debe cumplir la nueva función para que la ley sea aplicable. La condición que resulta de la ley es:

$$f\ (g\ a\ b) = \mathit{step}_1\ a\ (f\ b)$$

Esta condición caracteriza una relación de refinamiento entre las funciones g y step_1, donde g es la función abstracta, step_1 la función concreta y f representa la función de abstracción. La función g manipula secuencias mientras que la función step_1 sus largos. En la mayoría de los casos se realiza el proceso de refinamiento de forma tal de obtener funciones concretas a partir de funciones abstractas pero en este caso se debe realizar en sentido contrario. Como step_1 manipula números es necesario construir una función g que manipule secuencias de esos largos, en cierto sentido una función "*unlength*". El problema radica en que *unlength* no es una función: dado un número no existe una única secuencia de ese largo. Por lo tanto, si bien la condición planteada sirve de guía para construir la función g, no la determina, siendo necesario recurrir a la creatividad. Para esto se analizan las ecuaciones que definen la función step_1:

- En la primera ecuación de la definición de la función $step_1$ se establece:

  $$step_1\ c\ (n, b) \mid blank\ c = (n, False)$$

  O sea, en el caso en que el carácter considerado, c, sea un blanco la función $g\ c\ (ws, b)$ debe devolver un par $(ws', False)$ tal que ws y ws' tienen el mismo largo. La forma más sencilla de resolver este punto es considerar $ws' = ws$.
  Por lo tanto se tiene una primera ecuación que permite definir la función g:

  $$g\ c\ (ws, b)\ \mid\ blank\ c = (ws, False)$$

- Al considerar la segunda ecuación que define $step_1$:

  $$step_1\ c\ (n, True) = (n, True)$$

  se observa que en el caso en que el carácter c no sea blanco es necesario que $g\ c\ (ws, True)$ devuelva un par $(ws', True)$ donde nuevamente ws' tiene el mismo largo que ws. En este caso es posible proceder de la misma forma que en la primera ecuación, tomando $ws' = ws$ pero en cambio se busca una forma de usar c, combinándola con ws de forma de obtener una secuencia con el mismo largo que la secuencia original.

  Si ws es una secuencia no vacía de secuencias de caracteres es posible agregar el carácter c a la primera secuencia de ws y de esta forma obtener una nueva secuencia del mismo largo que la original. En este caso es posible asegurar que ws es una secuencia no vacía ya que por la definición de $step_1$ si la componente booleana del par es $True$ entonces el natural del par es mayor que cero; por lo tanto la secuencia correspondiente tiene largo mayor que cero.

  La segunda ecuación correspondiente a la definición de la función g es entonces:

  $$g\ c\ (w : ws, True) = ((c : w) : ws, True)$$

- Por último al considerar la tercera ecuación de la definición de $step_1$:

  $$step_1\ c\ (n, False) = (n + 1, True)$$

  se observa que es necesario que la función $g\ \ c\ \ (ws, False)$ devuelva un par $(ws', True)$ donde el largo de ws' es uno más que el largo de ws. La forma más directa de construir esta nueva secuencia es agregando una secuencia a ws formada por el carácter c.

  $$g\ c\ (ws, False) = ([\,c\,] : ws, True)$$

De esta forma se completa la definición de la función g y es posible llegar a la descomposición de wc_1:

$$\begin{array}{lll}
wc_1\ xs & = fst\ (wc'_1\ (xs, ([\,], False))) & \\
wc'_1\ (xs, (as, bl)) = (length\ ws, b) & & \\
\quad \textbf{where}\ (ws, b) & & = foldl\ g\ (xs, (as, bl)) \\
\qquad g\ c\ (ws, b)\ \mid\ blank\ c & & = (ws, False) \\
\qquad g\ c\ (w : ws, True) & & = ((c : w) : ws, True) \\
\qquad g\ c\ (ws, False) & & = ([\,c\,] : ws, True)
\end{array}$$

```
#include <stdio.h>

/* count words in input */

int  blank(int c) {
     return (c == ' ' || c == '\n' || c == '\t');
}

main ()
{
    int c, d, nw;
    nw = 0;
    d  = ' ';
    c = getchar();
    while   (c != EOF) {
           if (!blank(c) && blank(d)) {
              ++nw;
           }
           d = c; c = getchar();
    }
    printf("%d\n", nw);
}
```

Figura 3.3: Segunda version de contar palabras

A continuación, al igual que en [Gib06], se vuelve a realizar el proceso de fisión tomando en este caso como punto de partida una función diferente que también permite calcular la cantidad de palabras en un texto. En lugar de considerar una función que manipula un estado que indica si es necesario contar una palabra o no, se opta por manipular la estructura directamente y observar el carácter siguiente en el texto para determinar si es el comienzo de una nueva palabra y decidir entonces si corresponde contar una nueva palabra. El programa C correspondiente se muestra en la Figura 3.3.

En [Gib06] se escribe un programa funcional equivalente a dicho programa C en términos del operador *paramorfismo* [Mee92] para las listas invertidas ($\langle a \rangle$):

$$
\begin{array}{l}
paras \quad :: (a \to b \to \langle a \rangle \to b) \to (\langle a \rangle, b) \to b \\
paras\ \psi = f \\
\qquad\qquad \textbf{where} \\
\qquad\qquad f\ (SNil, z) \qquad\ = z \\
\qquad\qquad f\ (Snoc\ as\ a, z) = \psi\ a\ (f\ (as, z))\ as
\end{array}
$$

El operador paramorfismo permite capturar el comportamiento de funciones definidas por recursión primitiva con la característica de que en cada paso recursivo de la definición además del resultado de la llamada recursiva se cuenta con las subestructuras sobre la que se realizan estas llamadas. Usando la siguiente ley de fusión para *paras*:

$$f\ (\psi\ a\ e\ sa) = \psi'\ a\ (f\ e)\ sa \quad \Rightarrow \quad f \circ paras\ \psi\ (sx, e) = paras\ \psi'\ (sx, f\ e)$$

Gibbons obtiene una descomposición de la función original. Nosotros, en cambio, vamos a representar el programa C por una función sobre listas que utiliza el operador *foldlE*:

$$
\begin{aligned}
&wc_2 \quad :: List\ Char \to Int\\
&wc_2\ xs = foldlE\ step_2\ (xs, 0)\\
\\
&step_2 \quad :: Char \to Int \to List\ Char \to Int\\
&step_2\ c\ n\ x \qquad\quad \mid blank\ c = n\\
&step_2\ c\ n\ (d : xs) \mid blank\ d = n + 1\\
&step_2\ c\ n\ (d : xs) \qquad\quad = n\\
&step_2\ c\ n\ [\,] \qquad\qquad\quad = n + 1\\
\\
&blank\ c = (c == \texttt{' '}) \vee (c == \texttt{'\textbackslash n'}) \vee (c == \texttt{'\textbackslash t'})
\end{aligned}
$$

El proceso a seguir es totalmente análogo al realizado en el primer caso. Por lo tanto como primer paso se extrae un factor de la definición de wc_2 que involucra la función *length*:

$$
\begin{aligned}
wc_2\ xs &= wc'_2\ (xs, [\,])\\
wc'_2\ (xs, as) &= foldlE\ step_2\ (xs, length\ as)
\end{aligned}
$$

En la expresión de wc'_2 es posible reconocer una estructura como la presentada en la ley combinada asociada al operador *foldlE*. (Ley 3.3.9). Utilizando esta ley es posible entonces factorizar la expresión de wc'_2 en dos funciones:

$$
\begin{aligned}
&wc'_2\ (xs, as)\\
=&\quad \{\ \text{Definición de } wc'_2\ \}\\
&foldlE\ step_2\ (xs, length\ as)\\
=&\quad \{\ \text{FoldlE - Regla combinada}\ \}\\
&length \circ foldlE\ g_2\ (xs, as)
\end{aligned}
$$

siendo la condición para la aplicación de esta ley la que permite guiar la definición de la función g_2 a introducir:

$$length\ (g_2\ a\ b\ c) = step_2\ a\ (length\ b)\ c$$

Tomando en cuenta esta condición, la definición de la función $step_2$, y realizando manipulaciones del mismo tipo a las realizadas en el primer caso, se llega a una definición de g_2 y con ello a la definición de wc'_2:

$$
\begin{aligned}
&wc'_2 = length \circ foldlE\ g_2\\
&\quad \textbf{where}\\
&\quad\ g_2\ c\ ws\ x \qquad\quad \mid blank\ c = ws\\
&\quad\ g_2\ c\ ws\ (d : x) \mid blank\ d = [c] : ws\\
&\quad\ g_2\ c\ ws\ (d : x) \qquad\quad = ws\\
&\quad\ g_2\ c\ ws\ [\,] \qquad\qquad\quad = [c] : ws
\end{aligned}
$$

En resumen, es posible llegar a la siguiente descomposición de wc_2:

$$
\begin{aligned}
&wc_2\ xs = wc'_2\ (xs, [\,])\\
&wc'_2 = length \circ foldlE\ g_2
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{where} \\
\quad g_2\ c\ ws\ x \qquad\quad \mid blank\ c = ws \\
\quad g_2\ c\ ws\ (d : x) \mid blank\ d = [\,c\,] : ws \\
\quad g_2\ c\ ws\ (d : x) \qquad\quad = ws \\
\quad g_2\ c\ ws\ [\,] \qquad\qquad\quad = [\,c\,] : ws
\end{array}
$$

De esta forma mostramos cómo utilizando las reglas combinadas de *foldl* y *foldlE* fue posible realizar el proceso de fisión. En ambos casos los resultados que se obtienen coinciden con los obtenidos en [Gib06].

3.4. Limitaciones del operador *afold*

En las secciones anteriores se mostró qué tanto el operador *afold* como su extensión son apropiados para capturar el comportamiento de un amplio conjunto de acumulaciones. De todas formas, existen acumulaciones que no pueden ser representadas mediante estos operadores.

Un ejemplo es la función *flatten*:

$$
\begin{array}{l}
flatten \qquad\qquad\quad :: (Btree\ a, List\ a) \rightarrow List\ a \\
flatten\ (Leaf\ a, xs) \quad = Cons\ a\ xs \\
flatten\ (Join\ t\ u, xs) = flatten\ (t, flatten\ (u, xs))
\end{array}
$$

En efecto, recordando las definiciones de los operadores *afold* y *afoldE* para árboles binarios con información en las hojas:

$$
\begin{array}{l}
afold_{B_a}\ (h_1, h_2)\ \psi\ \psi' = f \\
\quad \textbf{where} \\
\qquad f\ (Leaf\ a, z) \quad = h_1\ a\ z \\
\qquad f\ (Join\ t\ u, z) = h_2\ (f\ (t, \psi\ z))\ (f\ (u, \psi'\ z))\ z
\end{array}
$$

$$
\begin{array}{l}
afoldE_{B_a}\ (h_1, h_2)\ \psi\ \psi' = f \\
\quad \textbf{where} \\
\qquad f\ (Leaf\ a, z) \quad = h_1\ a\ z \\
\qquad f\ (Join\ t\ u, z) = h_2\ (f\ (t, \psi\ z\ t\ u))\ (f\ (u, \psi'\ z\ t\ u))\ z
\end{array}
$$

se puede ver que el operador $afold_{B_a}$ no puede capturar el comportamiento de la función *flatten* ya que, es necesario manipular la subestructura en el acumulador y esto no está contemplado en el operador. Esto sí puede lograrse con el operador $afoldE_{B_a}$:

$$
\begin{array}{l}
flatten = afoldE_{B_a}\ (Cons, ffst)\ \psi\ \psi' \\
\qquad\quad \textbf{where} \\
\qquad\qquad \psi\ ac\ t\ u = flatten\ (u, ac)
\end{array}
$$

donde *ffst* $a\ b\ c = a$, y ψ' es una función arbitraria dado que nunca se utiliza. Sin embargo se presenta el incoveniente de que la propia función *flatten* ocurre dentro de ψ, o sea, es un caso donde la manipulación del acumulador es realizada por la propia función que se está definiendo. Es posible identificar varias funciones sobre árboles que tienen un esquema de definición similar al de *flatten* por lo que sería deseable capturar el comportamiento de estas funciones y contar con leyes de fusión que las manipulen.

Como posible forma de eliminar la ocurrencia de la llamada recursiva del acumulador se analiza la posibilidad de representar este tipo de funciones por un par de funciones mutuamente recursivas que no posean llamadas recursivas en el acumulador las que luego vamos a representar en términos del operador *afold*.

Para separarnos del caso particular de *flatten* se considera la siguiente función f, que generaliza el esquema de esta función para un h_1 arbitrario:

$$\begin{aligned} f\ (Leaf\ a, z) &= h_1\ a\ z \\ f\ (Join\ t\ u, z) &= f\ (t, f\ (u, z)) \end{aligned}$$

Como primer paso se busca encontrar un par de funciones mutuamente recursivas con las características antes mencionadas. Con el objetivo de eliminar la llamada recursiva en el acumulador se recurre a la utilización de una función auxiliar g, definida como $g\ (u, z) = f\ (u, z)$:

$$\begin{aligned} f\ (Leaf\ a, z) &= h_1\ a\ z \\ f\ (Join\ t\ u, z) &= f\ (t, g\ (u, z)) \end{aligned}$$

Es trivial derivar una definición recursiva para g:

$$\begin{array}{ll} & g\ (Leaf\ a, z) \\ = & \quad \{\ \text{Definición de } g\ \} \\ & f\ (Leaf\ a, z) \\ = & \quad \{\ \text{Definición de } f\ \} \\ & h_1\ a\ z \end{array}$$

$$\begin{array}{ll} & g\ (Join\ t\ u, z) \\ = & \quad \{\ \text{Definición de } g\ \} \\ & f\ (Join\ t\ u, z) \\ = & \quad \{\ \text{Definición de } f\ \} \\ & f\ (t, f\ (u, z)) \\ = & \quad \{\ \text{Definición de } g\ \} \\ & g\ (t, f\ (u, z)) \end{array}$$

Por lo tanto, es posible expresar la función original como un par de funciones mutuamente recursivas:

$$\begin{aligned} f\ (Leaf\ a, z) &= h_1\ a\ z \\ f\ (Join\ t\ u, z) &= f\ (t, g\ (u, z)) \end{aligned}$$

$$g\ (Leaf\ a, z) \quad = h_1\ a\ z$$
$$g\ (Join\ t\ u, z) = g\ (t, f\ (u, z))$$

Ahora cada una de estas funciones puede ser expresada en términos del operador $afoldE_{B_a}$:

$$f = afoldE_{B_a}\ (h_1, \mathit{ffst})\ \psi_1\ \psi'$$
where
$$\psi_1\ z\ t\ u = g\ (u, z)$$

$$g = afoldE_{B_a}\ (h_1, \mathit{ffst})\ \psi_2\ \psi'$$
where
$$\psi_2\ z\ t\ u = f\ (u, z)$$

donde ψ' es una función arbitraria.

Una vez que se cuenta con la definición de estas funciones en términos del operador $afoldE_{B_a}$ es posible pensar en fusionar composiciones que las involucren utilizando las leyes de fusión presentadas en la Sección 3.2.

Ejemplo 3.4.1. Se considera la composición $p \circ f$ siendo f la función definida anteriormente y p una función adecuada para la composición:

$$\begin{array}{ll} & p \circ f \\ = & \quad \{\ \text{Definición de } f\ \} \\ & p \circ afoldE_{B_a}\ (h_1, \mathit{ffst})\ \psi_1\ \psi' \\ = & \quad \{\ \text{pure fusion } afoldE\ \} \\ & afoldE_{B_a}\ (p \circ h_1, \mathit{ffst})\ \psi_1\ \psi' \end{array}$$

Llegado a este punto no se puede considerar completa la fusión ya que la función ψ_1 se define en términos de la función g que tiene en su definición a la función f, por lo que no es posible afirmar que la función p haya alcanzado todos los componentes deseados. Es necesario entonces buscar una forma que permita avanzar en este sentido. Tomando como referencia lo realizado al obtener las reglas combinadas para el operador *foldl* y su extensión sería deseable poder aplicar algo similar de forma de que la función p o alguna función derivada de esta alcance al acumulador de la función f. Para que esto sea posible es necesario encontrar funciones m_1,m_2 y φ_1 de forma tal que se cumplan las siguientes condiciones que involucran a las funciones del *afold*:

$$p\ (h_1\ a\ b) \quad = m_1\ a\ (p\ b) = m'_1\ a\ b$$
$$\mathit{ffst}\ a\ b\ c \quad = m_2\ a\ b\ (p\ c)$$
$$p\ (\psi_1\ z\ t\ u) = \varphi_1\ (p\ z)\ t\ u$$

La función m_2 no es otra que la misma función *ffst* pero la existencia y definición de las funciones m_1 y φ_1 dependen de cada caso en particular. En los casos que existan estas funciones es posible continuar con la fusión utilizando la Ley Morph fusion de *afoldE*.

$$
\begin{array}{ll}
 & afoldE_{B_a}\ (p \circ h_1, ffst)\ \psi_1\ \psi' \\
= & \quad \{\ \text{reescritura de las funciones}\ \} \\
 & afoldE_{B_a}\ (m'_1, ffst)\ \psi_1\ \psi' \\
= & \quad \{\ \text{Ley Morph } afoldE\ \} \\
 & (afoldE_{B_a}\ (m_1, ffst)\ \varphi_1\ \psi') \circ (id\ \times\ p)
\end{array}
$$

Es importante observar que para obtener la definición de φ_1 es necesario considerar la composición $p \circ \psi_1$. La función ψ_1 se encuentra expresada en términos del operador $afoldE_{B_a}$ por lo que entonces es necesario recurrir nuevamente a las leyes de fusión de este operador para derivar φ_1. □

Ejemplo 3.4.2. A continuación se considera un caso particular que permite visualizar lo presentado en el ejemplo anterior para el caso general. Se considera la composición $sum \circ flatten$, siendo sum la función definida en el Ejemplo 3.2.5.

La expresión de *flatten* como dos funciones mutuamente recursivas es:

$$
\begin{array}{ll}
fflaten\ (Leaf\ a, z) & = Cons\ a\ z \\
fflaten\ (Join\ t\ u, z) & = fflatten\ (t, gflatten\ (u, z))
\end{array}
$$

$$
\begin{array}{ll}
gflatten\ (Leaf\ a, z) & = Cons\ a\ z \\
gflatten\ (Join\ t\ u, z) & = gflatten\ (t, fflatten\ (u, z))
\end{array}
$$

Sus respectivas expresiones en términos del operador $afoldE_{B_a}$ son:
para ψ' arbitrario,

$$
\begin{array}{l}
fflatten = afoldE_{B_a}\ (Cons, ffst)\ \psi_1\ \psi' \\
\qquad \textbf{where} \\
\qquad\quad \psi_1\ z\ t\ u = gflatten\ (u, z)
\end{array}
$$

$$
\begin{array}{l}
gflatten = afoldE_{B_a}\ (Cons, ffst)\ \psi_2\ \psi' \\
\qquad \textbf{where} \\
\qquad\quad \psi_2\ z\ t\ u = fflatten\ (u, z)
\end{array}
$$

A partir de estas expresiones es posible plantear la composición $sum \circ fflatten$ y aplicar las leyes de fusión del *afold*:

$$
\begin{array}{ll}
 & sum \circ fflatten \\
= & \quad \{\ \text{Definición de } fflatten\ \} \\
 & sum \circ afoldE_{B_a}\ (Cons, ffst)\ \psi_1\ \psi' \\
= & \quad \{\ \text{Ley pure fusion } afoldE\ \} \\
 & afoldE_{B_a}\ (sum \circ Cons, ffst)\ \psi_1\ \psi'
\end{array}
$$

Para continuar con la fusión es necesario encontrar m_1 tal que:
$sum\ (Cons\ a\ xs) = m_1\ a\ (sum\ xs) = m'_1\ a\ xs$. En este caso se puede afirmar que $m_1 = (+)$ y permite continuar la fusión de la siguiente forma:

$$
\begin{aligned}
& afoldE_{B_a}\ (sum \circ Cons, \mathit{ffst})\ \psi_1\ \psi' \\
= & \quad \{\ \text{reescritura de las funciones}\ \} \\
& afoldE_{B_a}\ (m'_1, \mathit{ffst})\ \psi_1\ \psi' \\
= & \quad \{\ \text{Ley Morph } afoldE\ \} \\
& (afoldE_{B_a}\ ((+), \mathit{ffst})\ \varphi_1\ \psi') \circ (id\ \times\ sum) \\
& \varphi_1\ (sum\ z)\ t\ u = sum\ (\mathit{gflatten}\ (u, z))
\end{aligned}
$$

En este es posible encontrar φ_1 que cumpla la condición establecida:
$\varphi_1\ a\ b\ c = a + sum\ (\mathit{gflatten}\ (c, Nil))$. En este definición se observa que está involucrada la composición de $sum \circ \mathit{gflatten}$, y como *gflatten* se encuentra expresada en términos del operador *afold* es necesario recurrir a las leyes de fusión de este operador para derivar φ_1.

La forma de proceder para resolver $sum \circ \mathit{gflatten}$ es totalmente análoga a la realizada para la composición $sum \circ \mathit{fflaten}$, obteniendo como resultado:

$$
\begin{aligned}
& sum \circ \mathit{gflatten} \\
= & \ (afoldE_{B_a}\ ((+), \mathit{ffst})\ \varphi_2\ \psi') \circ (id\ \times\ sum) \\
& \varphi_2\ (sum\ z)\ t\ u = sum\ (\mathit{fflatten}\ (u, z)) \\
& \varphi_2\ a\ b\ c = a + sum\ (\mathit{fflatten}\ (c, Nil))
\end{aligned}
$$

Por lo tanto, si tomamos $\mathit{ffus} = sum \circ \mathit{fflatten}$ y $\mathit{gfus} = sum \circ \mathit{gflatten}$ se obtienen las siguientes funciones mutuamente recursivas:

$$
\begin{aligned}
& \mathit{ffus} :: Num\ a \Rightarrow (Btree\ a, a) \to a \\
& \mathit{ffus} = afoldE\ ((+), \mathit{ffst})\ \varphi_1\ \psi' \\
& \quad \textbf{where} \\
& \qquad \varphi_1\ a\ b\ c = a + \mathit{gfus}\ (c, sum\ Nil)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{gfus} :: Num\ a \Rightarrow (Btree\ a, a) \to a \\
& \mathit{gfus} = afoldE\ ((+), \mathit{ffst})\ \varphi_2\ \psi' \\
& \quad \textbf{where} \\
& \qquad \varphi_2\ a\ b\ c = a + \mathit{ffus}\ (c, sum\ Nil)
\end{aligned}
$$

$$
sumflatten\ (term, ac) = \mathit{ffus}\ (term, sum\ ac)
$$

Este es un ejemplo donde la alternativa de descomponer la definición de la función original *flatten* en dos funciones mutuamente recursivas resultó válida en el sentido que permitió la definición de la función en términos del operador *afold* y su fusión con la función *sum* aplicando las leyes de fusión del operador. Al mismo tiempo muestra la complejidad de optar por este camino y las fuertes condiciones para su aplicación.

□

De los precedentes ejemplos se puede observar que si bien se busca ampliar el conjunto de acumulaciones de las que se pretende capturar el comportamiento, se está tomando un camino que lleva a imponer mayores condiciones en la estructura de las funciones a considerar. Esto fue uno de los motivos que llevó a modificar la línea de trabajo, descartando posibles extensiones

del operador *afold* y pasando a buscar otras alternativas que sean más libres en cuanto a la estructura de las funciones que consideran. En este sentido en el capítulo siguiente se presenta una nueva forma de enfrentar el problema de fusión con acumulaciones.

4 "Short-Cut Fusion" Acumulativa

Uno de los problemas del abordaje de acumulaciones basado en *afold* es su alta dependencia respecto a la estructura de las funciones involucradas. En este capítulo se presenta un enfoque diferente al problema de fusión de acumulaciones en el cual se hace énfasis en la identificación del proceso de producción de la estructura intermedia a eliminar, pero no así en la estructura interna de dicho proceso. Esto se logra a través de un abordaje basado en "Short-Cut Fusion". Se muestra la aplicación de este abordaje a programas Haskell y se realizan corridas de prueba que permiten comparar la performance de los programas originales con las respectivas versiones transformadas.

Este capítulo es una versión extendida de [MP09].

Contenido

4.1. Acumulaciones y "Short-Cut Fusion"

El enfoque presentado en el Capítulo 3 requiere que, en caso que la función productora de la estructura intermedia sea una acumulación, la misma debe ser expresable en términos del operador *afold* para poder realizar la fusión. Sin embargo, como fue presentado en la Sección 3.4, esto no siempre es posible ya que existen acumulaciones que no son expresables en términos del operador *afold*. Se busca entonces un esquema que permita capturar el comportamiento de acumulaciones que jueguen el rol de productoras, que se concentre solamente en la forma de construcción de la estructura intermedia, y al que se le pueda determinar alguna ley de fusión. Esto implica, buscar una alternativa que permita representar las acumulaciones en forma independiente de la estructura de su definición y enfocada al proceso de construcción de la estructura de datos intermedia. Para esto se opta por un enfoque del tipo "Short-Cut Fusion" (presentado en el Capítulo 2) ya que cumple con estas condiciones.

A continuación se analiza la posibilidad de aplicar "Short-Cut Fusion" en el caso de que alguna de las funciones involucradas en la composición sea una acumulación. Se consideran por separado los casos en que las acumulaciones se comportan como consumidoras o productoras en la composición a resolver.

4.1.1. Acumulaciones como funciones consumidoras

El enfoque de "Short-Cut Fusion" requiere que la función consumidora se exprese en términos del operador *fold*. Como este operador no maneja en forma explícita un parámetro adicional es necesario entonces expresar las acumulaciones como un *fold* de alto orden.

A modo de ejemplo se considera la composición $reverse \circ map\ f$ para el caso en que la función *reverse* es dada por su versión acumulativa:

$$
\begin{array}{ll}
reverse\ as & = areverse\ as\ Nil \\
areverse & :: List\ a \to List\ a \to List\ a \\
areverse\ Nil\ zs & = zs \\
areverse\ (Cons\ a\ as)\ zs & = areverse\ as\ (Cons\ a\ zs)
\end{array}
$$

La función *areverse* puede escribirse como un *fold* de alto orden:

$$
\begin{array}{l}
areverse = fold_{L_a}\ (id, hrev) \\
\quad \textbf{where} \\
\quad\quad hrev \quad :: a \to (List\ a \to b) \to (List\ a \to b) \\
\quad\quad hrev\ a\ rec = rec \circ (Cons\ a)
\end{array}
$$

mientras que la función $map\ f$ puede expresarse en términos de la función *build*:

$$
\begin{array}{l}
map\ f = build_{L_a}\ \ (gmap\ f) \\
\quad \textbf{where} \\
\quad\quad gmap\ f\ (fnil, fcons)\ Nil \quad\quad\quad\quad = fnil \\
\quad\quad gmap\ f\ (fnil, fcons)\ (Cons\ a\ as) = fcons\ (f\ \ a)\ (gmap\ f\ (fnil, fcons)\ as)
\end{array}
$$

Por lo tanto se está en condiciones de aplicar la Ley 2.6.1, obteniendo como resultado $gmap\ f\ (id, hrev)$ lo que se corresponde con la siguiente definición recursiva:

$$\begin{aligned} fus\ f\ Nil &= id \\ fus\ f\ (Cons\ a\ as) &= fus\ f\ as \circ (Cons\ (f\ a)) \end{aligned}$$

Si bien la aplicación de la ley de fusión en estos casos no presenta dificultades es necesario detenerse en la representación de la función consumidora como un *fold* de alto orden. El problema con este tipo de definiciones es que se genera como salida una lista de llamadas pendientes a funciones la cual es isomorfa a la lista de entrada a la función. Esto es, en el ejemplo, para calcular el reverse de la lista de entrada el fold primero construye una lista de llamadas pendientes, la cual finalmente es aplicada al valor inicial del acumulador. Para observar esta situación se considera la evaluación de la función *areverse* para el caso de la lista de entrada $[1, 2, 3]$:

$$\begin{aligned} & areverse\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))) \\ = & \quad \{\ \text{Definición de } areverse \text{ en términos del operador } fold\ \} \\ & areverse\ (Cons\ 2\ (Cons\ 3\ Nil)) \circ (Cons\ 1) \\ = & \quad \{\ \text{Definición de } areverse \text{ en términos del operador } fold\ \} \\ & areverse\ (Cons\ 3\ Nil) \circ (Cons\ 2) \circ (Cons\ 1) \\ = & \quad \{\ \text{Definición de } areverse \text{ en términos del operador } fold\ \} \\ & areverse\ Nil \circ (Cons\ 3) \circ (Cons\ 2) \circ (Cons\ 1) \\ = & \quad \{\ \text{Definición de } areverse \text{ en términos del operador } fold\ \} \\ & id \circ (Cons\ 3) \circ (Cons\ 2) \circ (Cons\ 1) \\ = & \quad \{\ \text{Aplicación de la función } id\ \} \\ & (Cons\ 3) \circ (Cons\ 2) \circ (Cons\ 1) \end{aligned}$$

Esta computación suele ser menos eficiente que la versión original de la función *areverse* y esta pérdida de eficiencia en la representación de la función consumidora impacta en forma negativa en la eficiencia de la función resultante de la fusión. En efecto, la lista de llamadas pendientes generada por el *fold* de alto orden se traslada directamente al cuerpo de las función productora (en el lugar de los constructores) como resultado de la aplicación de la ley de fusión. Como consecuencia el programa resultante produce valores funcionales isomorfos con la estructura intermedia.

$$
\begin{aligned}
& fus\ f\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))) \\
= & \quad \{\ \text{Definición de } fus \text{ en términos del operador } fold\ \} \\
& fus\ f\ (Cons\ 2\ (Cons\ 3\ Nil)) \circ (Cons\ (f\ 1)) \\
= & \quad \{\ \text{Definición de } fus\ \} \\
& fus\ f\ (Cons\ 3\ Nil) \circ (Cons\ (f\ 2)) \circ (Cons\ (f\ 1)) \\
= & \quad \{\ \text{Definición de } fus\ \} \\
& fus\ f\ Nil \circ (Cons\ (f\ 3)) \circ (Cons\ (f\ 2)) \circ (Cons\ (f\ 1)) \\
= & \quad \{\ \text{Definición de } fus\ \} \\
& id \circ (Cons\ (f\ 3)) \circ (Cons\ (f\ 2)) \circ (Cons\ (f\ 1)) \\
= & \quad \{\ \text{Aplicación de la función } id\ \} \\
& (Cons\ (f\ 3)) \circ (Cons\ (f\ 2)) \circ (Cons\ (f\ 1))
\end{aligned}
$$

En la evaluación anterior es posible observar la traslación del efecto originado por la representación de la función consumidora como una función de alto orden a la función resultado de aplicar la ley de fusión: *fus f* para la lista de entrada $[1, 2, 3]$.

Este tipo de incoveniente se presenta en forma insalvable cuando la función productora es una acumulación, de hecho, una acumulación de determinado tipo. Este punto será considerado nuevamente en la Sección 4.3.1 al realizar la comparación de los tiempos de ejecución entre los resultados de aplicar la ley de fusión y los programas originales. Si no se está frente a este caso el incoveniente puede ser subsanado realizando la saturación en la definición de la función obtenida como resultado. En el ejemplo considerado se obtiene:

$$
\begin{aligned}
& fus\ f\ Nil\ zs = zs \\
& fus\ f\ (Cons\ a\ as)\ zs = fus\ f\ as\ (f\ a : zs)
\end{aligned}
$$

4.1.2. Acumulaciones como funciones productoras

La eficacia de la aplicación de fusión en el caso en que la función productora es una acumulación depende del tipo de acumulación. Es posible distinguir dos casos según el comportamiento del parámetro de acumulación:

- cuando el parámetro adicional mantiene un valor auxiliar que se puede usar en la construcción del resultado final pero que no forma parte del resultado final explícitamente (acumulador de contexto).

- cuando el resultado de la función, o parte del mismo, es construido directamente en el parámetro adicional. Como consecuencia se tiene la restricción de que el parámetro de acumulación sea del mismo tipo que el resultado final de la función (acumulador como parte de la estructura).

A continuación se analiza cada uno de estos casos. Las acumulaciones que poseen parámetros de acumulación de ambos tipos requieren de un tratamiento combinado.

Acumulador como contexto En este caso la existencia del parámetro adicional en la función productora no es relevante para la función consumidora, en el sentido de que su valor no forma parte de la estructura intermedia y por lo tanto no se presentan dificultades en la aplicación de la ley de fusión estándar. Podría decirse que desde el punto de vista de la ley de fusión no es necesario distinguir a la función productora como una acumulación.

Para ilustrar esta situación se considera el siguiente ejemplo donde se calcula la altura de un árbol binario (*aheight*) en dos pasos: primero se reemplaza el valor de cada hoja por su altura en el árbol (*down*) y en un segundo paso se calcula el máximo valor contenido en el árbol (*maxtree*).

$$
\begin{aligned}
&aheight && :: (Btree\ a, Int) \to Int\\
&aheight && = maxtree \circ down
\end{aligned}
$$

$$
\begin{aligned}
&down && :: (Btree\ a, Int) \to Btree\ Int\\
&down\ (Leaf\ a, n) && = Leaf\ n\\
&down\ (Join\ i\ d, n) && = Join\ (down\ (i, n+1))\ (down\ (d, n+1))
\end{aligned}
$$

$$
\begin{aligned}
&maxtree && :: Ord\ a \Rightarrow Btree\ a \to a\\
&maxtree\ (Leaf\ a) && = a\\
&maxtree\ (Join\ i\ d) && = max\ (maxtree\ i)\ (maxtree\ d)
\end{aligned}
$$

Las funciones *maxtree* y *down* pueden expresarse en términos del operador *fold* y la función *build*, respectivamente, de la siguiente forma:

$$
\begin{aligned}
&down = build_{B_a}\ gdown\\
&\quad \textbf{where}\\
&\qquad gdown\ (leaf, join)\ (Leaf\ a, n) && = leaf\ n\\
&\qquad gdown\ (leaf, join)\ (Join\ i\ d, n) && = join\ (gdown\ (leaf, join)\ (i, n+1))\\
&\qquad\qquad && \qquad\quad (gdown\ (leaf, join)\ (d, n+1))
\end{aligned}
$$

$$maxtree = fold_{B_a}\ (id, max)$$

Al aplicar la regla *fold*/*build* (Ley 2.6.1) se obtiene la siguiente expresión:

$$aheight = gdown\ (id, max)$$

que da origen a la siguiente definición recursiva:

$$
\begin{aligned}
&aheight\ (Leaf\ a, n) && = n\\
&aheight\ (Join\ i\ d, n) && = max\ (aheight\ (i, n+1))\ (aheight\ (d, n+1))
\end{aligned}
$$

El hecho de que *down* es una acumulación no impacta en la forma en que se realiza la fusión. El acumulador mantiene simplemente un entero que es usado para colocar en las hojas del árbol con su correspondiente altura y la computación de este valor debe ser mantenido por la fusión.

Acumulador como parte de la estructura Se analiza ahora el caso en que el acumulador de la función productora forma parte del resultado de dicha función. Este es el caso en que los métodos tradicionales de fusión suelen fallar por no conseguir alcanzar el parámetro de acumulación. En particular, ésta es la situación presentada en la Sección 1.1.2. Ejemplos de acumulaciones de este tipo son la función *atakeWhile* (Sección 1.1.2), la versión acumulativa de

la función *reverse*, *areverse*, vista anteriormente en este capítulo y la definición de la función *flatten* presentada en la Sección 3.4:

$$\begin{aligned}
&\mathit{flatten} && :: (\mathit{Btree}\ a, \mathit{List}\ a) \to \mathit{List}\ a\\
&\mathit{flatten}\ (\mathit{Leaf}\ a, xs) && = \mathit{Cons}\ a\ xs\\
&\mathit{flatten}\ (\mathit{Join}\ i\ d, xs) && = \mathit{flatten}\ (i, \mathit{flatten}\ (d, xs))
\end{aligned}$$

La función *flatten* resulta particularmente interesante ya que realiza una llamada recursiva en el propio acumulador de la función; esta característica se repite frecuentemente en acumulaciones que manipulan estructuras arborescentes.

Para visualizar las dificultades que se presentan en este caso se considera la función *sumArb* que dado un árbol binario con información en las hojas devuelve la suma de los elementos que lo componen.

$$\begin{aligned}
&\mathit{sumArb} && :: \mathit{Num}\ a \Rightarrow \mathit{Btree}\ a \to a\\
&\mathit{sumArb}\ t && = \mathit{asumArb}\ (t, \mathit{Nil})\\
&\mathit{asumArb} && = \mathit{suma} \circ \mathit{flatten}
\end{aligned}$$

donde se considera la función *suma* con la siguiente definición:

$$\begin{aligned}
&\mathit{suma} && :: \mathit{Num}\ a \Rightarrow \mathit{List}\ a \to a\\
&\mathit{suma}\ \mathit{Nil} && = 0\\
&\mathit{suma}\ (\mathit{Cons}\ a\ as) && = a + \mathit{suma}\ as
\end{aligned}$$

Al intentar derivar una definición recursiva de *asumArb* por análisis de casos se obtiene:

$$\begin{aligned}
\mathit{asumArb}\ (\mathit{Leaf}\ a, xs) &= \mathit{suma}\ (\mathit{flatten}\ (\mathit{Leaf}\ a, xs))\\
&= \mathit{suma}\ (\mathit{Cons}\ a\ xs)\\
&= a + \mathit{suma}\ xs\\
\mathit{asumArb}\ (\mathit{Join}\ i\ d, xs) &= \mathit{suma}\ (\mathit{flatten}\ (\mathit{Join}\ i\ d, xs))\\
&= \mathit{suma}\ (\mathit{flatten}\ (i, \mathit{flatten}\ (d, xs)))\\
&= \mathit{asumArb}\ (i, \mathit{flatten}\ (d, xs))
\end{aligned}$$

De igual forma que en el ejemplo presentado en la Sección 1.1.2 la definición obtenida no es satisfactoria ya que se continúa construyendo una lista intermedia ahora en el acumulador de la llamada recursiva. Dicha lista acumula el valor de las hojas de los subárboles derechos y debe ser evaluada por la función *suma* al alcanzar el paso base de la función.

Por el tipo de acumulación que se está considerando el valor contenido en el acumulador forma parte de la estructura generada por la función productora. Por lo tanto al momento de realizar la fusión ésta parte de la estructura debe ser necesariamente transformada por la función consumidora.

La versión que uno debería obtener por aplicación de fusión es la siguiente:

$$\begin{aligned}
&\mathit{asumArb'}\ (\mathit{Leaf}\ a, n) && = a + n\\
&\mathit{asumArb'}\ (\mathit{Join}\ i\ d, n) && = \mathit{asumArb'}\ (i, \mathit{asumArb'}\ (d, n))
\end{aligned}$$

junto con la siguiente relación:

$$asumArb\ (t, xs) = asumArb'\ (t, suma\ xs)$$

que establece que el valor inicial del acumulador original debe ser convertido al nuevo formato por la aplicación de la función consumidora. De hecho esta relación se cumple siempre que se considera una acumulación de este tipo como función productora. O sea, dada una función consumidora *cons* y una acumulación de este tipo como función productora *prod* el resultado de la fusión de las mismas *fus* cumple la siguiente relación:

$$cons \circ prod \quad = \quad fus \circ (id \ \times \ cons) \tag{4.1}$$

La función *asumArb′* puede obtenerse por la aplicación de la regla *fold/build* (Ley 2.6.1). Para ello se requiere que: (i) la función consumidora (en el ejemplo, *suma*) sea expresable en términos del operador *fold*, y (ii) la función productora (en el ejemplo, *flatten*) sea expresable en términos de la función *build*.

El punto (i) no presenta inconvenientes ya que es posible expresar la función *suma* como un *fold* de la siguiente forma:

$$suma = fold_{L_a}\ (0, (+))$$

Al considerar el punto (ii), recordamos que para obtener la expresión de una función en términos de la función *build* es necesario poder abstraer de su definición los constructores de la estructura de salida. Observando la definición de *flatten*, se ve que en el caso de una hoja se construye el valor de salida por la aplicación del constructor *Cons*, por lo que debe ser abstraído. En la ecuación correspondiente a un *Join* no hay constructor a abstraer dado que se realiza una llamada recursiva de cola ("tail-recursion") y una llamada anidada. Por lo tanto, la definición de *flatten* en términos de *build* sería en principio la siguiente:

$$\begin{array}{l} flatten = build_{L_a}\ gflat \\ \quad \textbf{where} \\ \quad\quad gflat\ (fnil, fcons)\ (btree, acum) \ = gflat'\ (fnil, fcons)\ (btree, acum) \\ \quad\quad gflat'\ (fnil, fcons)\ (Leaf\ a, ac) \ = fcons\ a\ ac \\ \quad\quad gflat'\ (fnil, fcons)\ (Join\ i\ d, ac) = gflat'\ (fnil, fcons)\ (i, gflat'\ (fnil, fcons)\ (d, ac)) \end{array}$$

Si bien en esta definición se observa que las funciones *gflat* y *gflat′* coinciden se opta, por considerar esta definición ya que permite visualizar con mayor claridad los pasos siguientes del proceso que se está realizando. Sin embargo, esta definición no es correcta, ya que *gflat*, la función del *build*, no está bien tipada. En efecto, según la definición de *build*, dicha función debería tener el siguiente tipo:

$$\forall\ b\ .\ (b, a \to b \to b) \to (Btree\ a, List\ a) \to b$$

y esto no se cumple debido a que, por un lado, la llamada $gflat'\ (fnil, fcons)\ (d, ac)$ no es de tipo $List\ a$ sino de tipo polimórfico b y, por otra parte, $fcons\ a\ ac$ espera que ac sea de tipo b, y no de tipo *Lista* como lo es. Este error es causado por el hecho de que, a pesar de haber intentado abstraer todos los constructores de la estructura intermedia en la definición de *flatten*, esto no se realizó en su totalidad ya que en el valor inicial del acumulador persisten constructores que no han sido considerados.

Con el objetivo de lograr abstraer estos constructores, se introduce primeramente la función identidad reemplazando *acum* por *id acum*.

$$
\begin{array}{l}
\mathit{flatten} = \mathit{build}_{L_a}\ \mathit{gflat} \\
\quad \textbf{where} \\
\quad\quad \mathit{gflat}\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{btree}, \mathit{acum}) \ = \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{btree}, \mathit{id}\ \mathit{acum}) \\
\quad\quad \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{Leaf}\ a, \mathit{ac}) \ = \mathit{fcons}\ a\ \mathit{ac} \\
\quad\quad \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{Join}\ i\ d, \mathit{ac}) = \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (i, \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (d, \mathit{ac}))
\end{array}
$$

El siguiente paso consiste en representar la función identidad como $\mathit{fold}_{L_a}\ (\mathit{Nil}, \mathit{Cons})$.

$$
\begin{array}{l}
\mathit{flatten} = \mathit{build}_{L_a}\ \mathit{gflat} \\
\quad \textbf{where} \\
\quad\quad \mathit{gflat}\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{btree}, \mathit{acum}) \ = \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{btree}, \mathit{fold}_{L_a}\ (\mathit{Nil}, \mathit{Cons})\ \mathit{acum}) \\
\quad\quad \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{Leaf}\ a, \mathit{ac}) \ = \mathit{fcons}\ a\ \mathit{ac} \\
\quad\quad \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{Join}\ i\ d, \mathit{ac}) = \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (i, \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (d, \mathit{ac}))
\end{array}
$$

De esta forma se logra la visibilidad requerida de los constructores del acumulador pudiendo realizar ahora la abstracción completa:

$$
\begin{array}{l}
\mathit{flatten} = \mathit{build}_{L_a}\ \mathit{gflat} \\
\quad \textbf{where} \\
\quad\quad \mathit{gflat}\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{btree}, \mathit{acum}) \ = \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{btree}, \mathit{fold}_{L_a}\ (\mathit{fnil}, \mathit{fcons})\ \mathit{acum}) \\
\quad\quad \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{Leaf}\ a, \mathit{ac}) \ = \mathit{fcons}\ a\ \mathit{ac} \\
\quad\quad \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (\mathit{Join}\ i\ d, \mathit{ac}) = \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (i, \mathit{gflat}'\ (\mathit{fnil}, \mathit{fcons})\ (d, \mathit{ac}))
\end{array}
$$

Mientras $\mathit{gflat} :: \forall\ b \circ (b, a \to b \to b) \to (\mathit{Btree}\ a, \mathit{List}\ a) \to b$ observar que $\mathit{gflat}' :: \forall\ b \circ (b, a \to b \to b) \to (\mathit{Btree}\ a, b) \to b$ pues $\mathit{fold}_{L_a}\ (\mathit{fnil}, \mathit{fcons}) :: \mathit{List}\ a \to b$.

A partir de esta formulación, es posible aplicar la Ley 2.6.1 a $\mathit{asumArb} = \mathit{suma} \circ \mathit{flatten}$ obteniendo una función equivalente que no genera la estructura intermedia:

$$
\begin{array}{l}
\mathit{asumArb}\ (\mathit{btree}, \mathit{acum}) = \mathit{asumArb}'\ (\mathit{btree}, \mathit{fold}_{L_a}\ (0, (+))\ \mathit{acum}) \\
\quad \textbf{where} \\
\quad\quad \mathit{asumArb}'\ (\mathit{Leaf}\ a, \mathit{ac}) \ = a + \mathit{ac} \\
\quad\quad \mathit{asumArb}'\ (\mathit{Join}\ i\ d, \mathit{ac}) = \mathit{asumArb}'\ (i, \mathit{asumArb}'\ (d, \mathit{ac}))
\end{array}
$$

lo que a su vez es equivalente a :

$$
\begin{array}{l}
\mathit{asumArb}\ (\mathit{btree}, \mathit{acum}) = \mathit{asumArb}'\ (\mathit{btree}, \mathit{suma}\ \mathit{acum}) \\
\quad \textbf{where} \\
\quad\quad \mathit{asumArb}'\ (\mathit{Leaf}\ a, \mathit{ac}) \ = a + \mathit{ac} \\
\quad\quad \mathit{asumArb}'\ (\mathit{Join}\ i\ d, \mathit{ac}) = \mathit{asumArb}'\ (i, \mathit{asumArb}'\ (d, \mathit{ac}))
\end{array}
$$

De esta forma se obtiene la definición deseada de la función $\mathit{asumArb}'$ que satisface la relación antes mencionada con la función *asumArb* .

4.2. Definición de la Ley "Short-Cut Fusion" Acumulativa

Si bien el uso de la Ley 2.6.1 resultó efectiva, en el caso anterior, la misma no resalta la existencia de un acumulador tanto en la preparación para su uso como en su aplicación. Por otra parte, a pesar de que el acumulador requiere de una manipulación adicional que es uniforme en todos los casos, la misma debe ser considerada cada vez que se define la función productora en términos de la función *build*. Este tipo de consideraciones nos lleva al planteo de intentar incorporar esta manipulación uniforme como parte de la ley de fusión. En este sentido, nuestro objetivo es la formulación de una ley de fusión del estilo de la Ley 2.6.1 que contemple explícitamente la manipulación requerida para el acumulador de la función productora. Para lograrlo se propone modificar la definición de la función *build*, introduciendo una nueva función *buildA* que tiene modificado el tipo de su función g de forma de hacer explícita la existencia de un parámetro adicional que representa al acumulador. Al igual que en la función *build*, se mantiene la restricción de que la construcción de la estructura de salida sea realizada únicamente en términos de los constructores del tipo, pero ahora también incluyendo al acumulador.

Para el caso de listas la función *buildA* propuesta es la siguiente:

$$\begin{aligned} &buildA_{L_a} \quad :: (\forall\ b\ .\ (b, a \to b \to b) \to (c, b) \to b) \to (c, List\ a) \to List\ a \\ &buildA_{L_a}\ g = g\ (Nil, Cons) \end{aligned}$$

donde el primer argumento de la función g corresponde como antes a una L_a-álgebra, mientras que el segundo argumento es un par formado por el eventual valor de entrada junto con el acumulador.

Como se muestra a continuación, la manipulación esperada del acumulador surge en forma automática como parte de la ley de fusión por aplicación de un "free theorem" [Wad89] asociado con el nuevo tipo de la función g.

Al igual que *build*, la función *buildA* se generaliza para cualquier tipo de dato (con functor asociado F) de la siguiente forma:

$$\begin{aligned} &buildA_F \quad :: (\forall\ a\ .\ (F\ a \to a) \to (b, a) \to a) \to (b, \mu F) \to \mu F \\ &buildA_F\ g = g\ in_F \end{aligned}$$

A partir de esta nueva definición para la función *build* se propone la siguiente regla de fusión la cual satisface – como era deseable – la Ecuación (4.1):

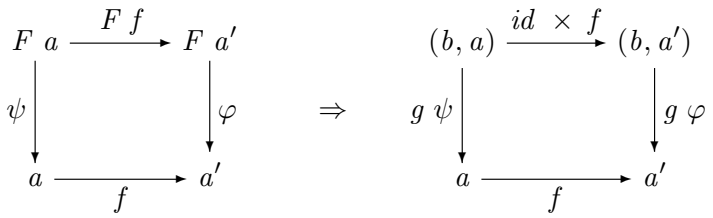
Ley 4.2.1 (**Regla Fold/Build Acumulativa**)**.**

$$\begin{aligned} & g :: \forall\ a\ .\ (F\ a \to a) \to (b, a) \to a \\ \Rightarrow \\ & fold_F\ \varphi \circ buildA_F\ g = g\ \varphi \circ (id\ \times\ fold_F\ \varphi) \end{aligned}$$

Prueba: Un "free theorem" asociado con el tipo de la función g establece que:

$$f \circ \psi = \varphi \circ F\ f\ \Rightarrow\ f \circ g\ \psi = g\ \varphi \circ (id\ \times\ f)$$

lo que se puede expresar en términos de los siguientes diagramas:



Tomando f como $fold_F\ \varphi$ y ψ como in_F la ley anterior se instancia como:

$$fold_F\ \varphi \circ in_F = \varphi \circ F\ fold_F\ \varphi\ \Rightarrow\ fold_F\ \varphi \circ g\ in_F = g\ \varphi \circ (id\ \times\ fold_F\ \varphi)$$

Se observa que el antecedente de esta implicación es verdadero por la definición del operador *fold* (Ecuación (2.3)), por lo que siempre se cumple. Por lo tanto,

$$fold_F\ \varphi \circ g\ in_F\ =\ g\ \varphi \circ (id\ \times\ fold_F\ \varphi)$$

□

A continuación se presenta una serie de ejemplos donde se aplica esta ley para distintos tipos de datos.

Ejemplo 4.2.2. Listas

La instancia correspondiente de la Ley 4.2.1 es la siguiente:

$$fold_{L_a}\ (fnil, fcons) \circ buildA_{L_a}\ g = g\ (fnil, fcons) \circ (id\ \times\ fold_{L_a}\ (fnil, fcons))$$

Es posible usar esta ley para completar la fusión de la composición de funciones presentada en la sección anterior: $asumArb = suma \circ flatten$. Como se vio anteriormente la expresión de la función *suma* en términos del operador *fold* es: $suma = fold_{L_a}\ (0, (+))$. El siguiente paso es expresar la función *flatten* en términos de *buildA*:

$flatten = buildA_{L_a}\ gflat$
where
$gflat\ (fnil, fcons)\ (Leaf\ a, ac)\ \ = fcons\ a\ ac$
$gflat\ (fnil, fcons)\ (Join\ i\ d, ac) = gflat\ (fnil, fcons)\ (i, gflat\ (fnil, fcons)\ (d, ac))$

Por lo tanto es posible escribir la composición original como: $fold_{L_a}\ (0, (+)) \circ buildA_{L_a}\ gflat$

Aplicando la Ley 4.2.1 se obtiene como resultado:

$$gflat\ (0, (+)) \circ (id\ \times\ fold_{L_a}\ (0, (+)))$$

En resumen, si se define $fus = gflat\ (0, (+))$, es posible expresar el resultado de la siguiente forma:

$$
\begin{array}{l}
asumArb\ (btree, acum) = fus\ (btree, suma\ acum) \\
\quad \textbf{where} \\
\quad\quad fus\ (Leaf\ a, ac) = a + ac \\
\quad\quad fus\ (Join\ i\ d, ac) = fus\ (i, fus\ (d, ac))
\end{array}
$$

□

Ejemplo 4.2.3. En este caso se considera una composición de funciones donde tanto la productora como la consumidora son acumulaciones:

$$
\begin{array}{ll}
sumarev\ xs\ ys\ ac & = sumaAc\ (areverse\ (xs, ys))\ ac \\
 & \\
sumaAc & :: Num\ a \Rightarrow List\ a \rightarrow a \rightarrow a \\
sumaAc\ Nil\ n & = n \\
sumaAc\ (Cons\ a\ l)\ n & = sumaAc\ l\ (a + n) \\
 & \\
areverse & :: (List\ a, List\ a) \rightarrow List\ a \\
areverse\ (Nil, ys) & = ys \\
areverse\ (Cons\ x\ xs, ys) & = areverse\ (xs, Cons\ x\ ys)
\end{array}
$$

Primero es necesario expresar la función *sumaAc* en términos del operador *fold*. Como se trata de una acumulación será expresada como un *fold* de alto orden.

$$
\begin{array}{l}
sumaAc = fold_{L_a}\ (id, hsum) \\
\quad \textbf{where} \\
\quad\quad hsum\ :: Num\ a \Rightarrow a \rightarrow (a \rightarrow a) \rightarrow (a \rightarrow a) \\
\quad\quad hsum\ a\ rec = \lambda x \rightarrow rec\ (a + x)
\end{array}
$$

El siguiente paso consiste en expresar la función *areverse* en términos de *buildA*:

$$
\begin{array}{l}
areverse = buildA_{L_a}\ grev \\
\quad \textbf{where} \\
\quad\quad grev\ (fnil, fcons)\ (Nil, ys) = ys \\
\quad\quad grev\ (fnil, fcons)\ (Cons\ x\ xs, ys) = grev\ (fnil, fcons)\ (xs, fcons\ x\ ys)
\end{array}
$$

Aplicando la Ley 4.2.1 se obtiene como resultado:

$$
\begin{array}{ll}
 & sumarev\ xs\ ys\ ac \\
= & grev\ (id, hsum)\ (xs, fold_{L_a}\ (id, hsum)\ ys)\ ac \\
= & grev\ (id, hsum)\ (xs, sumaAc\ ys)\ ac
\end{array}
$$

Si se define $fus = grev\ (id, hsum)$ se obtiene que :

$$
\begin{array}{l}
sumarev\ xs\ ys\ ac = fus\ (xs, sumaAc\ ys)\ ac \\
\quad \textbf{where} \\
\quad\quad fus\ (Nil, s) = s \\
\quad\quad fus\ (Cons\ x\ xs, s) = fus\ (xs, hsum\ x\ s)
\end{array}
$$

Haciendo el inlining de la llamada a *hsum* queda en evidencia que el acumulador de la función *fus* es efectivamente una función.

$$fus\ (Nil, s) = s$$
$$fus\ (Cons\ x\ xs, s) = fus\ (xs, \lambda a \rightarrow s\ (x + a))$$

□

Ejemplo 4.2.4. Árboles binarios con información en las hojas

La instancia de la Ley 4.2.1 para este tipo de datos es la siguiente:

$$fold_{B_a}\ (fleaf, fjoin) \circ buildA_{B_a}\ g = g\ (fleaf, fjoin) \circ (id\ \times\ fold_{B_a}\ (fleaf, fjoin))$$

donde

$$buildA_{B_a} :: (\forall\ b\ .\ (a \rightarrow b, b \rightarrow b \rightarrow b) \rightarrow (c, b) \rightarrow b) \rightarrow (c, Btree\ a) \rightarrow Btree\ a$$
$$buildA_{B_a}\ g = g\ (Leaf, Join)$$

Consideremos una función que permite construir un árbol en forma acumulativa de la siguiente forma:

$$constB :: (Int, Btree\ a) \rightarrow Btree\ a$$
$$constB\ (0, t) = t$$
$$constB\ (n + 1, t) = constB\ (n, Join\ t\ t)$$

Se pretende componer esta función con otra que cuenta la cantidad de constructores de un árbol definida como:

$$cantConst :: Num\ a \Rightarrow Btree\ b \rightarrow a$$
$$cantConst\ (Leaf\ a) = 1$$
$$cantConst\ (Join\ i\ d) = 1 + cantConst\ i + cantConst\ d$$

Es posible aplicar la instancia correspondiente de la Ley 4.2.1 para fusionar la composición de estas funciones ya que la función *constB* se puede expresar en términos de la función $buildA_{B_a}$ y la función *cantConst* en términos del operador *fold* de la siguiente forma:

$$constB = buildA_{B_a}\ gconst$$
where
$$gconst\ (fleaf, fjoin)\ (0, t) = t$$
$$gconst\ (fleaf, fjoin)\ (n + 1, t) = gconst\ (fleaf, fjoin)\ (n, fjoin\ t\ t)$$

$$cantConst = fold_{B_a}\ (const\ 1, contar)$$
where $contar\ i\ d = 1 + i + d$

La aplicación de la ley da como resultado: $cantConst\ (constB\ (n, t)) = gconst\ (const\ 1, contar)\ (n, cantConst\ t)$ cuya expresión recursiva es:

$$cantConst\ (constB\ (n, t)) = fus\ (n, cantConst\ t)$$
where $fus\ (0, c) = c$
$$fus\ (n + 1, c) = fus\ (n, 1 + c + c)$$

□

Ejemplo 4.2.5. Árboles de estructura

Consideremos la siguiente definición de árboles binarios que no contienen información ni en los nodos ni en las hojas.

$$\textbf{data}\ \mathit{ShapeTree} = \mathit{SLeaf} \mid \mathit{SJoin}\ \mathit{ShapeTree}\ \mathit{ShapeTree}$$

El functor base correspondiente a este tipo es:

$$\begin{aligned} S\ a &= 1 + a \times a \\ S &:: (a \to b) \to (S\ a \to S\ b) \\ S\ f &= id + f \times f \end{aligned}$$

El operador *fold* tiene la siguiente definición:

$$\begin{aligned} &\mathit{fold}_S :: (a, a \to a \to a) \to \mathit{ShapeTree} \to a \\ &\mathit{fold}_S\ (\mathit{fsleaf}, \mathit{fsjoin}) = f_S \\ &\quad \textbf{where}\ f_S\ \mathit{SLeaf} = \mathit{fsleaf} \\ &\qquad\quad f_S\ (\mathit{SJoin}\ i\ d) = \mathit{fsjoin}\ (f_S\ i)\ (f_S\ d) \end{aligned}$$

En este caso la regla *fold*/*buildA* es dada por:

$$\mathit{fold}_S\ (\mathit{fsleaf}, \mathit{fsjoin}) \circ \mathit{buildA}_S\ g = g\ (\mathit{fsleaf}, \mathit{fsjoin}) \circ (id \times \mathit{fold}_S\ (\mathit{fsleaf}, \mathit{fsjoin}))$$

donde

$$\begin{aligned} &\mathit{buildA}_S :: (\forall\ a\ .\ (a, a \to a \to a) \to (b, a) \to a) \to (b, \mathit{ShapeTree}) \to \mathit{ShapeTree} \\ &\mathit{buildA}_S\ g = g\ (\mathit{SLeaf}, \mathit{SJoin}) \end{aligned}$$

Dadas las siguientes definiciones:

$$\begin{aligned} &\mathit{rp} :: (\mathit{ShapeTree}, \mathit{ShapeTree}) \to \mathit{ShapeTree} \\ &\mathit{rp}\ (\mathit{SLeaf}, w) = w \\ &\mathit{rp}\ (\mathit{SJoin}\ i\ d, w) = \mathit{SJoin}\ (\mathit{rp}\ (i, \mathit{SLeaf}))\ (\mathit{rp}\ (d, w)) \\ &\mathit{cantNodos} :: \mathit{ShapeTree} \to \mathit{Int} \\ &\mathit{cantNodos}\ \mathit{SLeaf} = 0 \\ &\mathit{cantNodos}\ (\mathit{SJoin}\ i\ d) = 1 + \mathit{cantNodos}\ i + \mathit{cantNodos}\ d \end{aligned}$$

es posible usar la regla *fold*/*buildA* (Ley 4.2.1) para fusionar la siguiente composición:

$$\mathit{cantRp} = \mathit{cantNodos} \circ \mathit{rp}$$

Para ello, primero es necesario expresar la función *rp* en términos de *buildA*:

$$\begin{aligned} &\mathit{rp} = \mathit{buildA}_{\mathit{ShapeTree}}\ \mathit{grp} \\ &\quad \textbf{where} \\ &\qquad \mathit{grp}\ (\mathit{fsleaf}, \mathit{fsjoin})\ (\mathit{SLeaf}, w) = w \\ &\qquad \mathit{grp}\ (\mathit{fsleaf}, \mathit{fsjoin})\ (\mathit{SJoin}\ i\ d, w) = \mathit{fsjoin}\ (\mathit{grp}\ (\mathit{fsleaf}, \mathit{fsjoin})\ (i, \mathit{fsleaf})) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{grp}\ (\mathit{fsleaf}, \mathit{fsjoin})\ (d, w)) \end{aligned}$$

y *cantNodos* en término del operador *fold*:

$$\begin{array}{l} cantNodos = fold_S\ (0, hcant) \\ \quad \textbf{where}\ hcant\ i\ d = 1 + i + d \end{array}$$

Aplicando la Ley 4.2.1 se obtiene como resultado:

$$\begin{array}{ll} & cantRp \\ = & \\ & grp\ (0, hcant) \circ (id\ \times\ fold_S\ (0, hcant)) \\ = & \\ & grp\ (0, hcant)\ (id\ \times\ cantNodos) \end{array}$$

Este resultado se puede expresar en términos de una función recursiva de la siguiente forma:

$$\begin{array}{l} cantRp\ (st, ac) = fus\ (st, cantNodos\ ac) \\ \quad \textbf{where} \\ \quad\quad fus\ (SLeaf, w) \qquad = w \\ \quad\quad fus\ ((SJoin\ i\ d), w) = 1 + fus\ (i, 0) + fus\ (d, w) \end{array}$$

□

Por lo expuesto anteriormente es simple ver que se cumple la siguiente relación entre *buildA* y *build*:

Propiedad 4.2.6.

$$\begin{array}{rl} & g :: \forall\ a\ .\ (F\ a \to a) \to (b, a) \to a \\ \Rightarrow & \\ & buildA_F\ g = build_F\ g' \end{array}$$

donde $g' :: \forall\ a\ .\ (F\ a \to a) \to (b, \mu F) \to a$ *es dado por* $g'\ \varphi = g\ \varphi \circ (id\ \times\ fold_F\ \varphi)$.

Prueba:

$$\begin{array}{ll} & buildA_F\ g \\ = & \quad \{\ \text{Definición de } buildA\ \} \\ & g\ in_F \\ = & \quad \{\ fold_F\ in_F = id\ \} \\ & g\ in_F \circ (id\ \times\ fold_F\ in_F) \\ = & \quad \{\ \text{Definición de } g'\ \} \\ & build_F\ g' \end{array}$$

□

4.3. Pruebas realizadas

Con la aplicación de la regla *fold/build acumulativa* (Ley 4.2.1) efectivamente se previene la creación de la estructura intermedia. Sin embargo, esta eliminación por sí sola no garantiza una mejora en la eficiencia de los programas. Con el objetivo de observar el impacto práctico de la aplicación de esta ley, se realizaron pruebas para un conjunto de programas en las cuales se tomaron medidas de tiempo y espacio requerido en la ejecución de los mismos. Se usó un conjunto de programas simples pero que consideramos representativos, los cuales corresponden a la composición de una función productora con acumulador y una función consumidora (en algunos casos también una acumulación). No se han considerado ejemplos donde la función productora sea una acumulación de contexto ya que, como se mencionó en la Sección 4.1.2, desde el punto de vista de la fusión para este caso no es necesario distinguir a la función productora como una acumulación. En el Apéndice A se presentan los códigos de los ejemplos considerados.

Los programas fueron escritos en Haskell y se compilaron con GHC (Glasgow Haskell Compiler) versión 6.8.2, siendo ejecutados en un Laptop Dell Inspiron 1420 Intel Core2 Duo CPU T5750 2.00GHz, 2GB RAM, ejecutando Linux Ubuntu 8.04

La aplicación de la regla *fold/build acumulativa* a los programas fue a través del uso de reglas de reescritura (RULES pragma) de GHC. Para ello es necesario especificar una regla de reescritura para cada una de las instancias de la ley a ser utilizada. Queda fuera del alcance de este trabajo la incorporación de dichas reglas en forma automática.

Para la aplicación de la ley de fusión es necesario que las funciones a componer estén expresadas en términos de *fold* y *buildA*. Aplicando las ideas presentadas en [Chi99, Chi00, LS95] podría implementarse un procedimiento que permita inferir la representación de las funciones involucradas en la composición en términos de *fold* y *buildA*. Este desarrollo se considera fuera del alcance de este trabajo. Esto tiene como consecuencia que sea necesario definir para cada tipo de dato involucrado el operador *fold* y la función *buildA* asociados al mismo y expresar las funciones a fusionar en términos de los mismos.

Como primer paso se comparó, para cada programa, la performance en tiempo y espacio entre la composición de las funciones originales y la función obtenida por aplicar la ley de fusión. Esto lleva a considerar las siguientes versiones, las cuáles se muestran tomando como base la composición $suma \circ flatten$ presentada en el Ejemplo 4.2.2:

Original(OR) Las funciones involucradas en la composición tienen una definición recursiva y el programa consiste en la composición directa de las funciones definidas de esta forma.

$$main = suma\ (flatten\ (term, ac))$$

$$\begin{aligned} suma &:: Num\ a \Rightarrow List\ a \rightarrow a \\ suma\ Nil &= 0 \\ suma\ (Cons\ a\ s) &= a + suma\ s \end{aligned}$$

$$\begin{aligned} flatten &:: (Btree\ a, List\ a) \rightarrow List\ a \\ flatten\ (Leaf\ a, w) &= Cons\ a\ w \\ flatten\ (Join\ i\ d, w) &= flatten\ (i, flatten\ (d, w)) \end{aligned}$$

Regla (SCA) Las funciones involucradas en la composición están definidas en términos del operador *fold* y la función *buildA*. El programa consiste en la composición de las funciones definidas de esta forma junto con la definición de la regla. El programa que finalmente se ejecuta es el resultante de aplicar la regla de fusión.

$$
\begin{aligned}
&\{-\ \#\ RULES\ \texttt{"fold/buildac"}\\
&\quad forall\ k\ z\ t\ ac\ (g :: forall\ b\ .\ (b, a \to b \to b)) \to (c, b) \to b)\ .\\
&\quad fold_{L_a}\ (z, k)\ (buildA_{L_a}\ g\ (t, ac)) = g\ (z, k)\ (t, fold_{L_a}\ (z, k)\ ac)\\
&\#\ -\}
\end{aligned}
$$

$$main = suma\ (flatten\ (term, ac))$$

$$suma = fold_{L_a}\ (0, (+))$$

$$
\begin{aligned}
&flatten = buildA_{L_a}\ gflat\\
&\quad \textbf{where}\\
&\quad gflat\ (fnil, fcons)\ (Leaf\ a, ac) \quad = fcons\ a\ ac\\
&\quad gflat\ (fnil, fcons)\ (Join\ i\ d, ac) = gflat\ (fnil, fcons)\ (i, gflat\ (fnil, fcons)\ (d, ac))
\end{aligned}
$$

Manual (MN) Para disparar las reglas de reescritura en GHC es necesario compilar los programas con la opción de optimización. Esto provoca que no se visualice en forma clara el efecto de aplicar fusión ya que se involucran otras optimizaciones propias del compilador. Como alternativa a esta situación se consideró una versión donde el programa que efectivamente se ejecuta es el resultado de aplicar la ley de fusión en forma manual (sin uso de las reglas de reescritura) permitiendo de esta forma realizar una ejecución del programa sin optimizaciones propias del compilador. En el ejemplo corresponde a:

$$main = gflat\ (0, (+))\ (term, fold_{L_a}\ (0, (+))\ ac)$$

En esta versión las funciones resultantes de la aplicación de la ley de fusión se expresan en términos de los componentes que forman parte de la codificación de las funciones originales en términos de los operadores *fold* y *build* (funciones del álgebra del *fold* y la función del *build*). Como se constató en las medidas de tiempo de ejecución, esto puede implicar un deterioro en la eficiencia de dichas funciones.

Final (FN) Por último se considera una nueva versión donde se realizan los inlining, simplificaciones y las aplicaciones parciales son eliminadas en la medida de lo posible. Estas transformaciones fueron realizadas manualmente. En el ejemplo esta versión corresponde a:

$$main = fus\ (term, suma\ ac)$$

$$
\begin{aligned}
&fus :: Num\ a \Rightarrow (Btree\ a, a) \to a\\
&fus\ (Leaf\ a, ac) = a + ac\\
&fus\ (Join\ i\ d, ac) = fus\ (i, fus\ (d, ac))
\end{aligned}
$$

Los ejemplos considerados para las pruebas de performance fueron divididos en dos grupos, a saber:

Grupo I La función consumidora tiene un argumento adicional que corresponde a un acumulador o a un parámetro de contexto.

Grupo II La función consumidora no es una acumulación.

En la Sección 4.3.1 se presentan los resultados obtenidos al realizar las comparaciones entre estas diferentes versiones desde el punto de vista del tiempo de ejecución. Como complemento a este trabajo en la Sección 4.3.2 se presentan los resultados obtenidos en dichas comparaciones desde el punto de vista del espacio de almacenamiento requerido.

4.3.1. Medidas de Tiempo de Ejecución

Con el objetivo de aislar el efecto de aplicar la fusión los tiempos de ejecución se obtienen de las estadísticas generadas utilizando la opción `-sstderr` en la ejecución del programa y sin optimizaciones del compilador(esto es, compiladas sin activar las opciones de optimización propias del compilador). Los tiempos incluyen el proceso de generación de la entrada al programa pero se considera un factor constante que se agrega en todos los casos y que no afecta los resultados obtenidos. En el Apéndice B se presentan los datos obtenidos en las corridas los cuales son usados para generar los valores presentados en todos los gráficos de este capítulo.

Primeramente se comparan los resultados obtenidos para las versiones manual y original mediante la fórmula:

$$\%Mejora \quad = \quad (T_{OR} - T_{MN})/T_{OR}$$

donde T_{OR} representa el tiempo de ejecución de la versión original y T_{MN} representa el tiempo de ejecución de la versión manual. La Figura 4.1 presenta gráficamente los resultados obtenidos para cada uno de los grupos considerados. La primera parte del gráfico corresponde a las funciones del grupo I y la segunda parte al grupo II. Los nombres de las funciones se forman por la concatenación de los nombres de la función consumidora y la función productora, por ejemplo *LargoReverse*. En el caso de consumidoras acumulativas se agrega el sufijo "Ac" a su nombre, por ejemplo, *LargoAcReverse*. En el caso de los nombres con sufijo "Rp" corresponden a composiciones donde la función productora es la función presentada en el Ejemplo 4.2.5. Para cada programa se consideran 3 valores: MUT (tiempo empleado efectivamente en la ejecución del programa), GC (tiempo empleado en garbage collector) y TOTAL (MUT+GC).

En la Figura 4.1 es posible observar que, salvo para *RempAcAsc*, todos los casos donde la función consumidora es un acumulación (parte izquierda del gráfico) el porcentaje de mejora es negativo, lo que indica que los tiempos de las versiones manuales son mayores que los tiempos de las versiones originales. Como se mencionó en la Sección 4.1.1, cuando la función consumidora es una acumulación, la misma se expresa como un *fold* de alto orden y la aplicación de la ley de fusión provoca la inserción de las operaciones del álgebra del fold en el cuerpo de la función productora. Como consecuencia de esto, cuando la función productora es una acumulación de segundo tipo (de acuerdo a la clasificación de la Sección 4.1.2), la función resultante de la fusión genera una lista de llamadas pendientes de funciones, en general inevitable. A pesar de esto se puede observar que el tiempo destinado al garbage collector (GC) disminuye en la mayoría de los casos.

Un caso de estos se presenta en el Ejemplo 4.2.3 donde se considera la composición $sumarev = sumaAc \circ areverse$ y cuyo resultado de aplicar la ley de fusión se corresponde con la siguiente función:

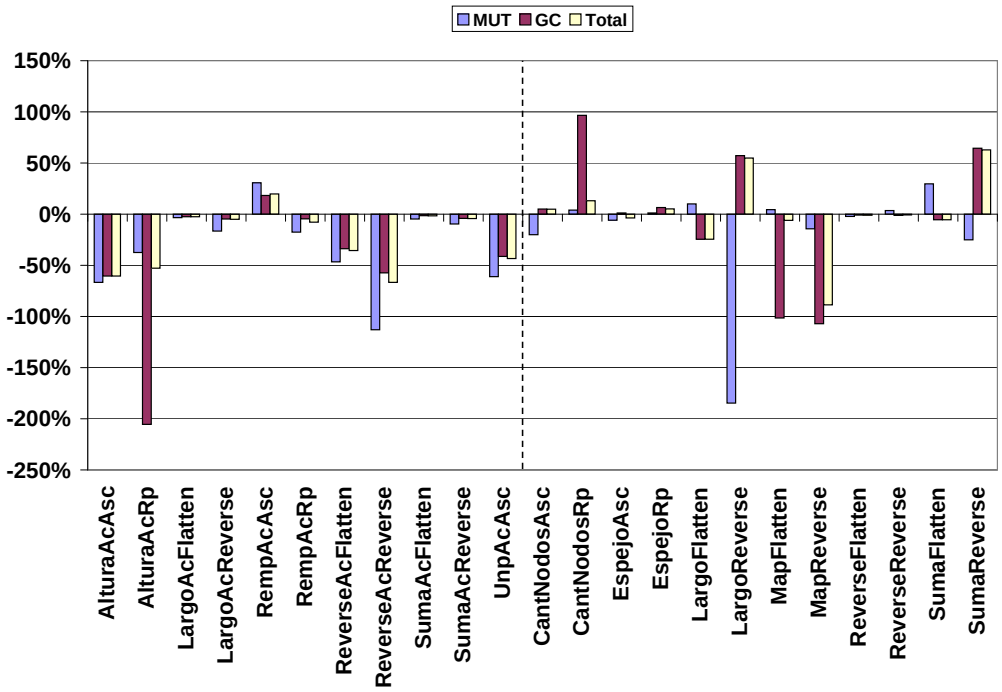


Figura 4.1: Relación T_{MN}/T_{OR}

$$
\begin{array}{l}
sumarev\ as = fus\ (as, suma\ Nil)\ 0 \\
\qquad \textbf{where} \\
\qquad\quad fus\ (Nil, s) = s \\
\qquad\quad fus\ (Cons\ x\ xs, s) = fus\ (xs, s \circ (x+))
\end{array}
$$

Es posible entonces observar que en este caso la evaluación de la función *fus* genera una lista de llamadas pendientes isomorfa con la lista de entrada:

$$
\begin{array}{ll}
 & fus\ (Cons\ x_1\ (\cdots\ (Cons\ x_n\ Nil)\ \cdots), zs)\ ac \\
= & \qquad \{\ \text{Aplicación de } fus\ \} \\
 & fus\ (Cons\ x_2\ (\cdots\ (Cons\ x_n\ Nil)), zs \circ (x_1+))\ ac \\
= & \qquad \{\ \text{Aplicación sucesiva de } fus\ \} \\
 & (zs \circ (x_1+) \circ \cdots \circ (x_n+))\ ac
\end{array}
$$

Este tipo de situaciones se da siempre cuando la función productora es una acumulación definida por "tail recursion" o por "recursión anidada". Sin duda la generación de esta lista de llamadas pendientes causa un deterioro en el tiempo de ejecución.

Con el objetivo de comparar los programas originales con las mejores soluciones que se pueden obtener por aplicación de la ley de fusión, se compara la versión final con la versión original. En la Figura 4.2 se presentan los resultados obtenidos, donde nuevamente la primera parte del gráfico corresponde a composiciones del grupo I y la segunda parte al grupo II.

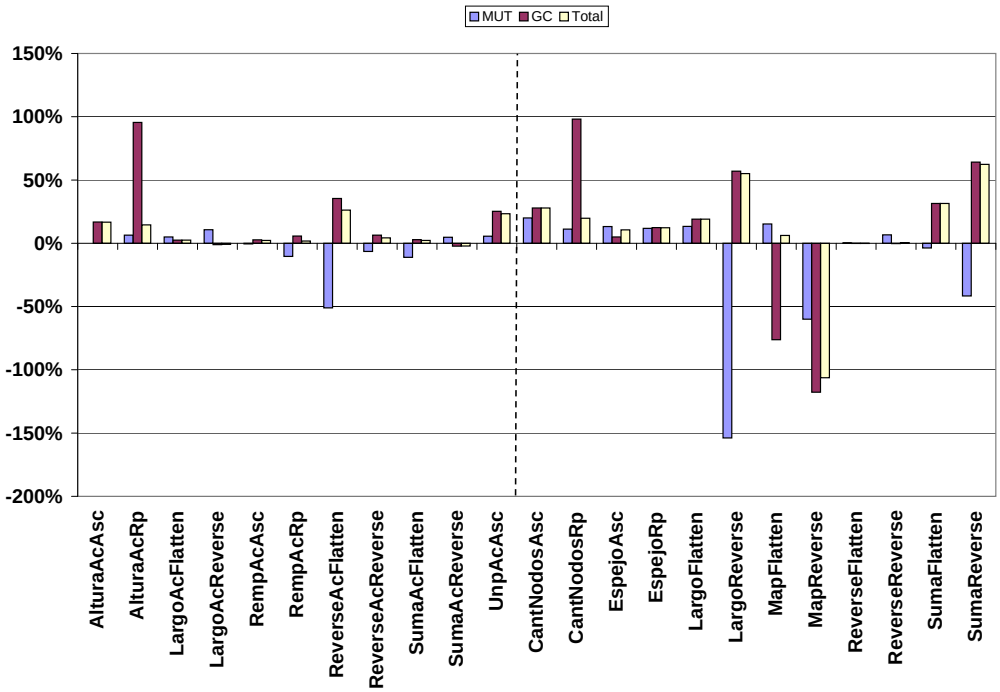


Figura 4.2: Relación T_{FN}/T_{OR}

En este caso se calcula:

$$\%Mejora \quad = \quad (T_{OR} - T_{FN})/T_{OR}$$

donde T_{OR} representa el tiempo de ejecución de la versión original y T_{FN} representa el tiempo de ejecución de la versión final.

En las Figuras 4.3 y 4.4 se presenta la comparación de los tiempos de ejecución en segundos para cada uno de los programas en su versión final y original para los grupos I y II respectivamente. El subfijo "O" representa la versión original y el sufijo "F" a la versión final. En los casos del grupo I donde los resultados son negativos, *LargoAcReverse* y *SumaAcReverse*, el porcentaje de mejora si bien es negativo los valores son pequeños (-1 % y -2 % respectivamente). Esta situación se puede considerar como un caso en que no hay cambios entre las versiones ya que al considerar la diferencia absoluta entre los tiempos de ejecución de estas versiones (0,4s y 0,95s) es menor que la diferencia en el tiempo que se da entre dos ejecuciones de un mismo programa en una misma versión.

Al considerar los programas del grupo II se observa que prácticamente la totalidad de los resultados son positivos tanto en la comparación de la versión manual como en la versión final, lo que indica que la aplicación de la regla es beneficiosa, incluso existen mejoras mayores al 50 % por ejemplo los programas *LargoReverse* y *SumaReverse*.

La excepción está dada por la función *MapReverse* ($map\ f \circ reverse$) donde se tiene un deterioro del orden del 100 % en el tiempo total (Ver Figura 4.2). Este comportamiento no puede explicarse por características propias de alguna de las funciones involucradas en la composición ya que hay otros ejemplos que involucran a estas funciones cuyo comportamiento es distinto

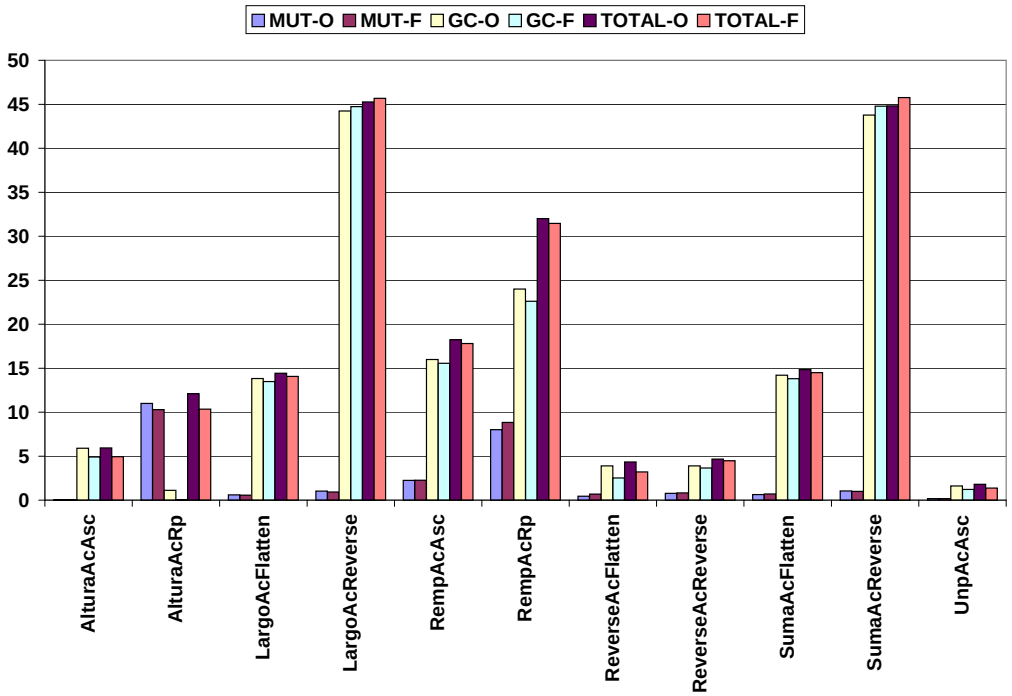


Figura 4.3: Tiempos versión final sobre original - Grupo I

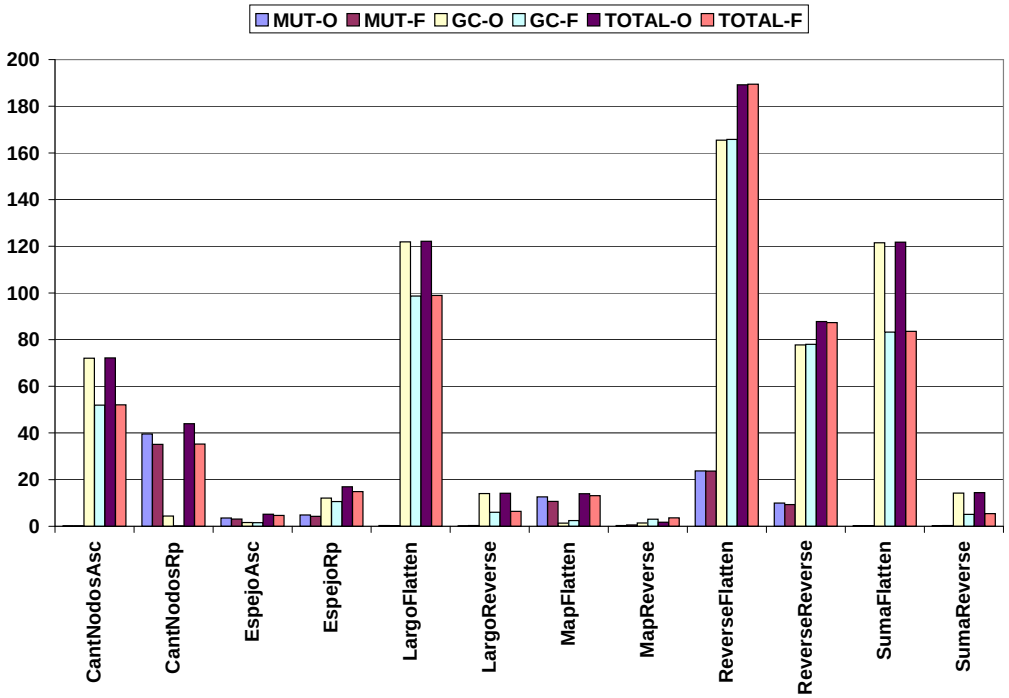


Figura 4.4: Tiempos versión finales sobre original - Grupo II

(por ejemplo las funciones *MapFlatten* y *LargoReverse*). Por lo tanto es necesario buscar una explicación directamente relacionada con la interacción de estas funciones y no en cada una de ellas.

El porcentaje negativo obtenido en la comparación indica que el tiempo total de ejecución requerido por la versión final es el doble del tiempo total requerido por la versión original pero si se analiza los tiempos totales discriminados se observa que el mayor aumento se da en el tiempo correspondiente al manejo de garbage collector (ver Figura 4.2). El tiempo de ejecución ocupado por el garbage collector se vincula directamente con el uso de heap por parte de los programas, por lo que resulta de interés analizar el uso del mismo para cada una de las versiones. Para esto se consideró la reseña biográfica (biographical profile) de cada versión de *MapReverse*, ya que esta permite observar los distintos tipos de celdas ocupadas en el heap en distintos momentos de la ejecución del programa. Por otro lado se analiza la reseña histórica de las clausuras (closure profile) con el objetivo de analizar qué funciones son las responsables de las celdas que se encuentran en el heap [RR96].

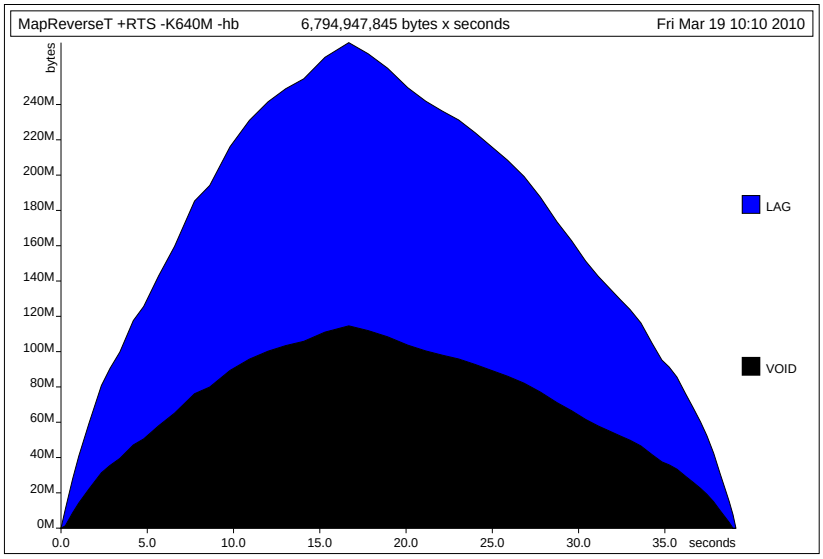


Figura 4.5: Reseña Biográfica - MapReverse - versión final

Para la obtención de estas reseñas se realiza la compilación de los programas con la opción `-prof -auto-all` y se ejecutan con la opción `-hb` para la obtención de la reseña biográfica y la opción `-hd` para la obtención de la reseña histórica de las clausuras.

En la reseña biográfica correspondiente a la versión final (Figura 4.5) se observa una importante cantidad de celdas de tipo **lag**, o sea celdas que han sido creadas y esperan para ser usadas por primera vez. Existen dos fases bien marcadas en el uso del heap durante la ejecución del programa: una primera fase de crecimiento, en cuanto al espacio requerido y una segunda fase de decrecimiento. Esto indica que existen muchas celdas que se crean, quedan pendientes de uso por un tiempo importante de la ejecución del programa y finalmente son usadas y eliminadas

del heap.

Sin embargo en la reseña biográfica de la versión original (Figura 4.6) prácticamente no se observa la existencia de este tipo de celdas; su número es constante y mínimo durante todo el tiempo de ejecución, además de tener un tamaño menor que el correspondiente al de la versión final. Todo esto marca una diferencia importante en el uso del heap por parte de estas versiones.

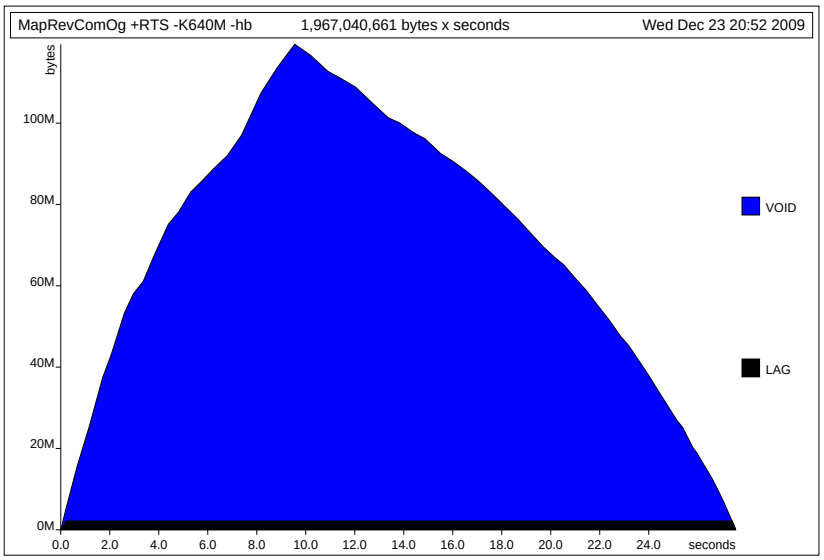


Figura 4.6: Reseña Biográfica - MapReverse - versión original

En la reseña histórica de clausuras correspondiente a la versión original (Figura 4.7) se observa que las celdas que ocupan el heap son generadas por la función (:). Sin embargo en la reseña correspondiente a la versión final (Figura 4.8) se observa que además de las celdas que corresponden a la función (:) (iguales a la versión original) se agregan celdas generadas por una nueva función.

Los códigos correspondientes a ambas versiones son:

Versión original:

$$mapReverse\ f\ term\ ac = map\ f\ (areverse\ (term, ac))$$

$$\begin{aligned} areverse &:: List\ a \to List\ a \to List\ a \\ areverse\ Nil\ zs &= zs \\ areverse\ (Cons\ a\ as)\ zs &= areverse\ as\ (Cons\ a\ zs) \end{aligned}$$

$$\begin{aligned} map &:: (a \to b) \to List\ a \to List\ b \\ map\ f\ Nil &= Nil \\ map\ f\ (Cons\ x\ xs) &= Cons\ (f\ x)\ (map\ f\ xs) \end{aligned}$$

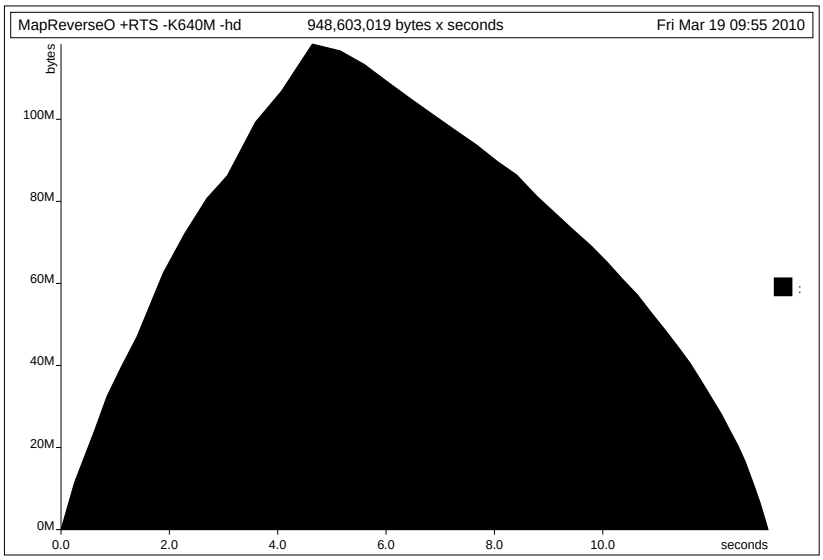


Figura 4.7: Reseña histórica de clausuras - MapReverse - versión original

Versión final

$$
\begin{aligned}
&\mathit{mapReverse}\ f\ \mathit{term}\ \mathit{ac} = \mathit{fus}\ f\ (\mathit{term}, \mathit{map}\ f\ \mathit{ac}) \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad\quad \mathit{fus}\ f\ (\mathit{Nil}, \mathit{ac}) = \mathit{ac} \\
&\qquad\qquad\quad \mathit{fus}\ f\ (\mathit{Cons}\ a\ x, \mathit{ac}) = \mathit{fus}\ f\ (x, \mathit{Cons}\ (f\ a)\ \mathit{ac})
\end{aligned}
$$

La diferencia entre estas dos versiones radica en que en el acumulador de la versión original se encuentra simplemente la aplicación de la función *Cons* a los elementos de la lista de entrada mientras que en el acumulador de la versión final se agregan aplicaciones de la función f del *map* a dichos elementos.

Una forma de visualizar el efecto que esta diferencia genera en la ejecución del programa es observar el desarrollo de la ejecución de cada una de las versiones sobre una cierta lista de entrada hasta generar el primer elemento de la lista de salida. Para una mejor visualización se utiliza la notación de Haskell para las listas.
En el caso de la versión original se realiza la siguiente reducción:

$$
\begin{aligned}
& mapReverse\ f\ [x_1, x_2\ \cdots, x_n]\ ac \\
= \quad & map\ f\ (areverse\ ([x_1, x_2, \cdots, x_n], ac)) \\
= \quad & map\ f\ (areverse\ ([x_2, \cdots, x_n], x_1 : ac)) \\
& \cdots\ \cdots\ \cdots \\
= \quad & map\ f\ (areverse\ ([\,], x_n : \cdots : x_2 : x_1 : ac)) \\
= \quad & map\ f\ (x_n : \cdots : x_2 : x_1 : ac) \\
= \quad & f\ x_n : map\ f\ x_{n-1} : \cdots : x_2 : x_1 : ac
\end{aligned}
$$

Para el caso de la versión final la reducción es la siguiente:

$$
\begin{aligned}
& mapReverse\ f\ [x_1, x_2, \cdots, x_n]\ ac \\
= \quad & fus\ f\ ([x_1, x_2, \cdots, x_n], map\ f\ ac) \\
= \quad & fus\ f\ ([x_2, \cdots, x_n], f\ x_1 : map\ f\ ac) \\
& \cdots\ \cdots\ \cdots \\
= \quad & fus\ f\ ([\,], f\ x_n : \cdots : f\ x_1 : map\ f\ ac) \\
= \quad & f\ x_n : \cdots : f\ x_1 : map\ f\ ac
\end{aligned}
$$

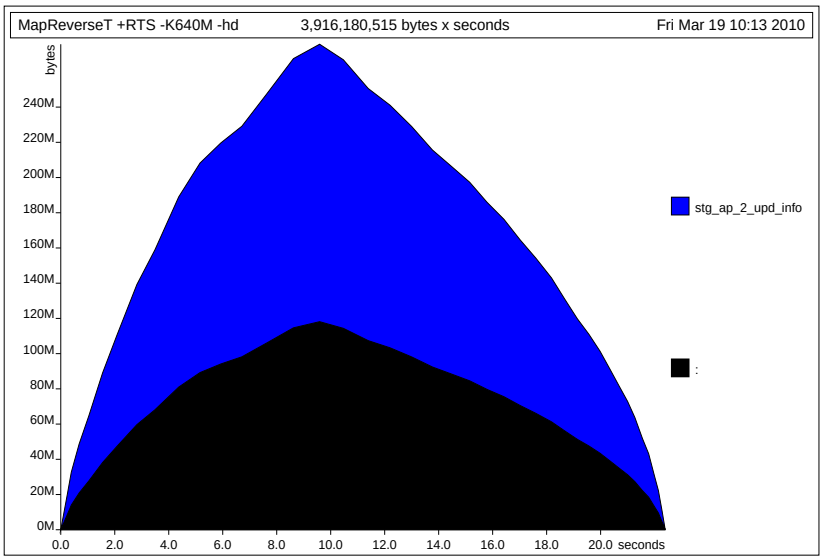


Figura 4.8: Reseña histórica de clausuras - MapReverse - versión final

En las reducciones anteriores se observa cómo en la versión final se generan aplicaciones pendientes de la función f (tantas como el largo de la lista de entrada) antes de poder generar el primer valor del resultado. Sin embargo esto no sucede en la evaluación de la versión original,

donde no quedan evaluaciones pendientes de la función f, sino que se van generando a medida que se genera el resultado.

Las celdas lags que se visualizan en la reseña biográfica de la versión final se corresponden por lo tanto con estas evaluaciones pendientes. La diferencia entre la versión original y la versión final en cuanto al uso del heap, y en consecuencia el tiempo necesario para su manipulación, sería la causante del deterioro en el tiempo de ejecución de la versión final.

Esta situación se origina en la fusión al insertar la evaluación de la función f en el cuerpo de la función productora *areverse*, generando de esta forma aplicaciones pendientes de f que afectan drásticamente el comportamiento de la versión final. Esto no sucede si en la función productora ya existe una función adicional a aplicar. Para ilustrar esto se considera una modificación a la función *areverse*, *areverseMod*, que agrega la aplicación de una función (por ejemplo $(+1)$) a los elementos de la lista de entrada al invertirlos.

$$
\begin{array}{ll}
areverseMod & :: Num\ a \Rightarrow (List\ a, List\ a) \rightarrow List\ a \\
areverseMod\ (Nil, w) & = w \\
areverseMod\ (Cons\ a\ x, w) & = areverseMod\ (x, Cons\ (a+1)\ w)
\end{array}
$$

A partir de esta modificación se espera que tanto en la versión original como en la final de la composición $map\ f \circ areverseMod$ se generen evaluaciones pendientes de funciones provocando un comportamiento similar en ambas versiones. La versión original en este caso es:

$$mapReverseM\ f\ term\ ac = map\ f\ (areverseMod\ (term, ac))$$

mientras que la versión final es:

$$
\begin{array}{l}
mapReverseMod\ f\ term\ ac = fus\ f\ (term, map\ f\ ac) \\
\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\quad fus\ f\ (Nil, ac) \qquad = ac \\
\qquad\qquad\qquad\qquad\quad fus\ f\ (Cons\ a\ x, ac) = fus\ f\ (x, Cons\ (f\ a+1)\ ac)
\end{array}
$$

Si se analiza la ejecución de la composición para las dos versiones se obtiene en el caso de la versión original:

$$
\begin{array}{ll}
 & mapReverseMod\ f\ [x_1, x_2\ \cdots, x_n]\ ac \\
= & map\ f\ (areverseMod\ ([x_1, x_2, \cdots, x_n], ac)) \\
= & map\ f\ (areverseMod\ ([x_2, \cdots, x_n], (x_1+1) : ac)) \\
 & \cdots\ \cdots\ \cdots \\
= & map\ f\ (areverseMod\ ([\,], (x_n+1) : \cdots : (x_2+1) : (x_1+1) : ac)) \\
= & map\ f\ ((x_n+1) : \cdots : (x_2+1) : (x_1+1) : ac) \\
= & f\ (x_n+1) : map\ f\ ((x_{n-1}+1) : \cdots : (x_2+1) : (x_1+1) : ac)
\end{array}
$$

y para la versión final:

$$
\begin{aligned}
& mapReverseMod\ f\ [x_1, x_2, \cdots, x_n]\ ac \\
= \ & fus\ f\ ([x_1, x_2, \cdots, x_n], map\ f\ ac) \\
= \ & fus\ f\ ([x_2, \cdots, x_n], (f\ x_1 + 1) : map\ f\ ac) \\
& \cdots\ \cdots\ \cdots \\
= \ & fus\ f\ ([\,], (f\ x_n + 1) : \cdots : (f\ x_1 + 1) : map\ f\ ac \\
= \ & (f\ x_n + 1) : \cdots : (f\ x_1 + 1) : map\ f\ ac
\end{aligned}
$$

donde se evidencia entonces la existencia de llamadas pendientes a funciones en ambas versiones, provocando así la aparición de celdas lags en el heap en ambas versiones. Como esta situación se repite en las dos versiones se espera que la comparación entre las mismas mejore sustancialmente.

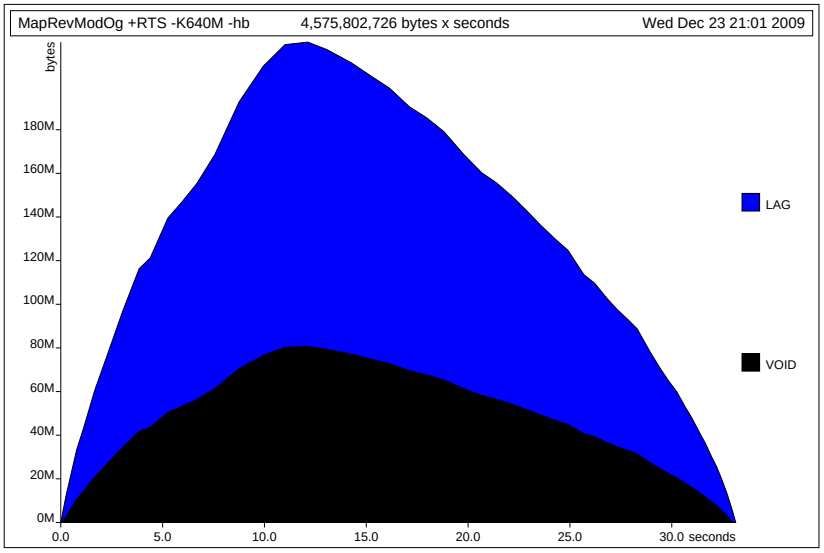


Figura 4.9: Reseña Biográfica - MapReverse Modificado - versión original

En las Figuras 4.9 y 4.10 se presentan los gráficos correspondientes a las reseñas biográficas de la versión original y final respectivamente. En estas figuras es posible observar la similitud de uso del heap en ambas versiones, como se esperaba.

Al realizar la comparación de los tiempos de ejecución entre estas versiones se obtiene un porcentaje de mejora de 21 %, lo que marca una diferencia importante con el caso de la composición original con la función *areverse*.

A partir de lo analizado para la composición *MapReverse* llama la atención el comportamiento de la composición *MapFlatten* que coincide en la función consumidora con *MapReverse* pero sin embargo su versión final presenta una mejora del 6 % con respecto a la versión original. Considerando los códigos correspondientes a la versión original y final de *MapFlatten*:

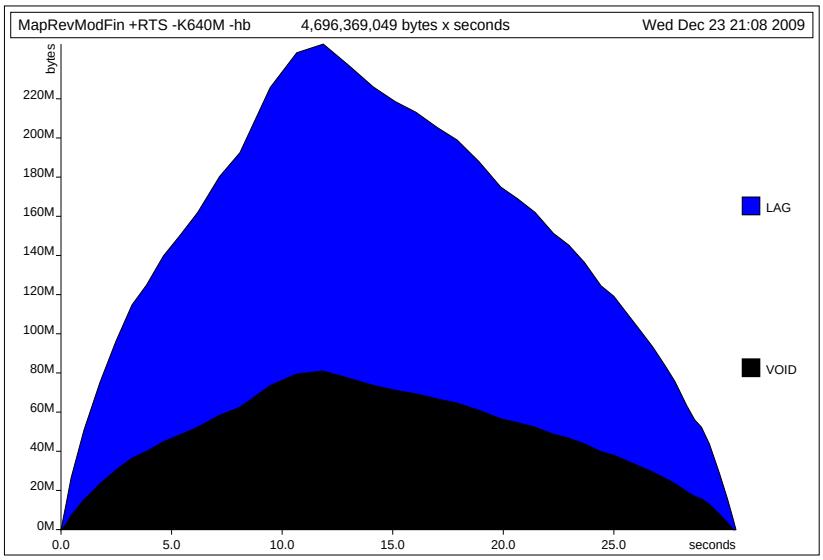


Figura 4.10: Reseña Biográfica - MapReverse Modificado - versión final

Versión original

$$mapFlatten\ f\ term\ ac = map\ f\ (flatten\ (term, ac))$$

Versión final

$$\begin{aligned} mapFlatten\ f\ term\ ac &= fus\ f\ (term, map\ f\ ac) \\ &\quad \textbf{where} \\ &\quad\quad fus\ f\ (Leaf\ a, ac) \quad = Cons\ (f\ a)\ ac \\ &\quad\quad fus\ f\ (Join\ i\ d, ac) = fus\ f\ (i, fus\ f\ (d, ac)) \end{aligned}$$

las evaluaciones necesarias para la obtención del primer elemento del resultado en cada caso, se observa que en esta composición el comportamiento de ambas versiones es igual.

En resumen, a partir de los resultados obtenidos en las pruebas realizadas, se puede afirmar que con la excepción de la composición *MapReverse* la aplicación de la ley de fusión es efectiva ya que en la mayoría de los casos se obtiene una mejora en el tiempo de ejecución y en solo unos pocos casos no se aprecian cambios.

4.3.2. Medidas del espacio de almacenamiento

Junto a las comparaciones realizadas entre los tiempos de ejecución de las distintas versiones y como complemento a las mismas se realizaron también comparaciones del espacio requerido

para la ejecución. Al igual que los tiempos de ejecución, el espacio requerido se obtuvo de las estadísticas generadas utilizando la opción `-sstderr` en la ejecución de los programas. Los datos obtenidos se presentan en el Apéndice B.

En primer lugar se comparó el consumo de espacio entre la versión manual y la versión original mediante el siguiente cálculo:

$$\%Mejora \quad = \quad (E_{OR} - E_{MN})/E_{OR}$$

donde E_{OR} representa el espacio utilizado en la ejecución de la versión original y E_{MN} representa el espacio utilizado en la ejecución de la versión manual.

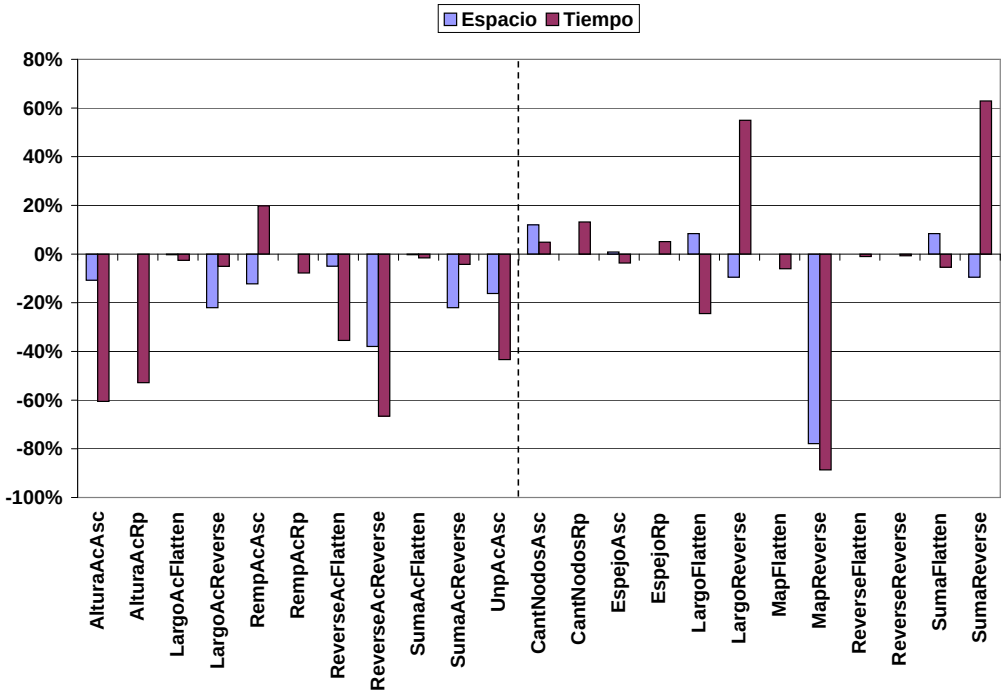


Figura 4.11: Relación Espacio - Tiempo versión manual sobre versión original

La siguiente comparación realizada fue entre las versiones finales y originales. En este caso se calcula:

$$\%Mejora \quad = \quad (E_{OR} - E_{FN})/E_{OR}$$

donde E_{FN} representa el espacio utilizado en la ejecución de la versión final.

Considerando que la ley de fusión aplicada provoca la eliminación de la estructura de datos intermedia, se espera que las posibles mejoras en el tiempo de ejecución sean un reflejo de las mejoras obtenidas en cuanto al espacio de memoria requerido para la ejecución. Por lo tanto resulta de interés considerar estos valores en forma conjunta. En la Figura 4.11 se presenta la comparación entre la versión manual y la original y en la Figura 4.12 se presenta la comparación entre la versión final y la original.

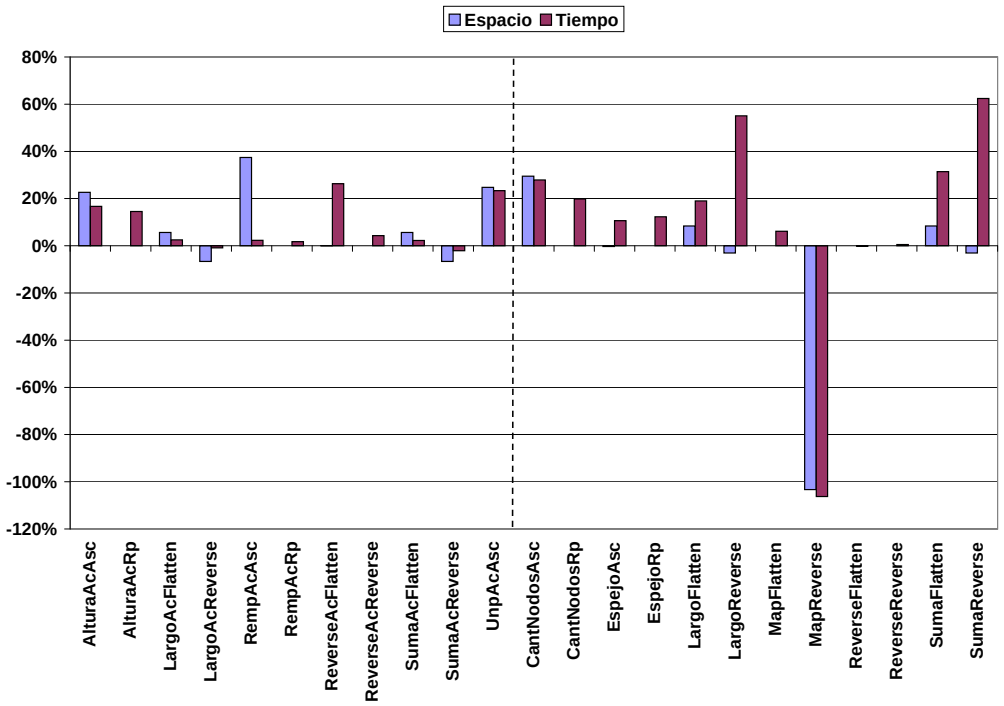


Figura 4.12: Relación Espacio - Tiempo versión final sobre original

En la comparación de la versión manual con la original, salvo unas pocas excepciones las mejoras en el espacio siguen el sentido de la mejora del tiempo. Del total de los programas considerados solo se obtienen mejoras en tres de los casos y son muy pequeñas.

Por otro lado en la comparación de la versión final con la versión original las mejoras desde el punto de vista del espacio son acompañadas por mejoras desde el punto de vista del tiempo de ejecución. En la amplia mayoría de los casos las mejoras del tiempo siguen el sentido de la mejora del espacio. Las excepciones están dadas por *LargoReverse* y *SumaReverse* donde se logra una mejora en el tiempo sin lograr una mejora en el espacio de memoria. De todas formas el deterioro en el espacio es mínimo (apenas un 3 %, podría considerarse dentro de los márgenes de error). Esta situación es análoga a la presentada en la Sección 4.3.1 (página 81) con respecto a la mejora en el tiempo.

En resumen, en las pruebas realizadas se verifica la relación esperada entre las mejoras en el espacio de memoria requerido y el tiempo de ejecución de los programas.

4.3.3. Interacción de optimizaciones

Las comparaciones realizadas en las secciones anteriores corresponden a ejecuciones no optimizadas (esto es, compiladas sin activar las opciones de optimización propias del compilador) de forma de poder evaluar el efecto de la aplicación de las leyes de fusión en forma aislada sin mezclar con otras optimizaciones. Como última etapa de las pruebas se repiten las comparaciones antes realizadas activando dichas opciones de forma de observar el comportamiento

al hacer interactuar la ley de fusión con otras optimizaciones. Como en los casos anteriores se realizaron las comparaciones tanto del punto de vista del tiempo de ejecución como del espacio de almacenamiento requerido.

En primera instancia se comparó la versión SCA con la versión original. En este caso no es necesario considerar la versión manual ya que la misma se obtiene por la aplicación de la regla en forma automática. Para esto se realizaron los siguientes cálculos:

$$\begin{aligned} \%Mejora &= (T_{OROp} - T_{SCA})/T_{OROp} \\ \%Mejora &= (E_{OROp} - E_{SCA})/E_{OROp} \end{aligned}$$

donde T_{OROp} y E_{OROp} representan el tiempo de ejecución y el espacio de memoria requerido por la versión original optimizada y T_{SCA} y E_{SCA} representan el tiempo de ejecución y el espacio de memoria de la versión obtenida por aplicación de la regla, versión SCA.

Nuevamente con el objetivo de considerar la mejor expresión para el resultado de aplicar la ley de fusión se consideró la versión final y se realizaron las comparaciones entre esta versión y la versión original, ahora con optimizaciones. En este caso se realizaron los siguientes cálculos:

$$\begin{aligned} \%Mejora &= (T_{OROp} - T_{FNOp})/T_{OROp} \\ \%Mejora &= (E_{OROp} - E_{FNOp})/E_{OROp} \end{aligned}$$

donde T_{OROp} y E_{OROp} representan el tiempo de ejecución y el espacio de memoria requerido por la versión original optimizada y T_{FNOp} y E_{FNOp} representan el tiempo de ejecución y el espacio de memoria requerido por la versión final optimizada.

Los datos obtenidos en todas las comparaciones realizadas se encuentran en el Apéndice B. A modo de ejemplo se presentan las gráficas correspondientes a las comparaciones entre la versión final y original en las Figuras 4.13 y 4.14.

Los resultados obtenidos en estos casos muestran que el agregado de las optimizaciones propias del compilador afectan en forma heterogénea a los distintos programas.

En primer lugar si se busca en qué porcentaje afectan las optimizaciones propias del compilador en el tiempo de ejecución de cada una de las versiones,original y final, se observa que en la mayoría de los casos el porcentaje de mejora es mayor en la versión original. Estos resultados se verifican tanto para el tiempo de ejecución como para el espacio de memoria requerido. Las Figuras 4.15 y 4.16 corresponden a la representación gráfica de los mismos. Cabe destacar que, en los casos donde los porcentajes de mejora son negativos, si se observan los valores absolutos se verifica que la diferencia que existe entre ellos es pequeña, se encuentran dentro de los márgenes de error de las pruebas.

Por otro parte el estudio del porcentaje de mejora de la versión final sobre la original entre las ejecuciones no optimizadas u optimizadas también se encuentran resultados heterogéneos. Por ejemplo en el caso de las composiciones *ReverseAcFlatten* y *LargoReverse* se obtienen mejores resultados en las ejecuciones optimizadas: 71 % contra 26 % en el primer caso y 100 % contra 55 % en el segundo caso. Sin embargo en las composiciones *ReverseReverse* y *UnpAsc* son más convenientes las ejecuciones no optimizadas: 1 % contra -2 % en el primer caso y 23 % contra 9 % en el segundo. Nuevamente este comportamiento se repite en forma homogénea desde el punto de vista del tiempo de ejecución y el espacio de memoria requerido.

Las situaciones antes planteadas no guardan relación con el tipo de función consumidora, con ninguna función productora en particular, ni con el tipo de la estructura intermedia eliminada; en todos estos grupos se obtienen resultados distintos. Por lo tanto a partir de los datos

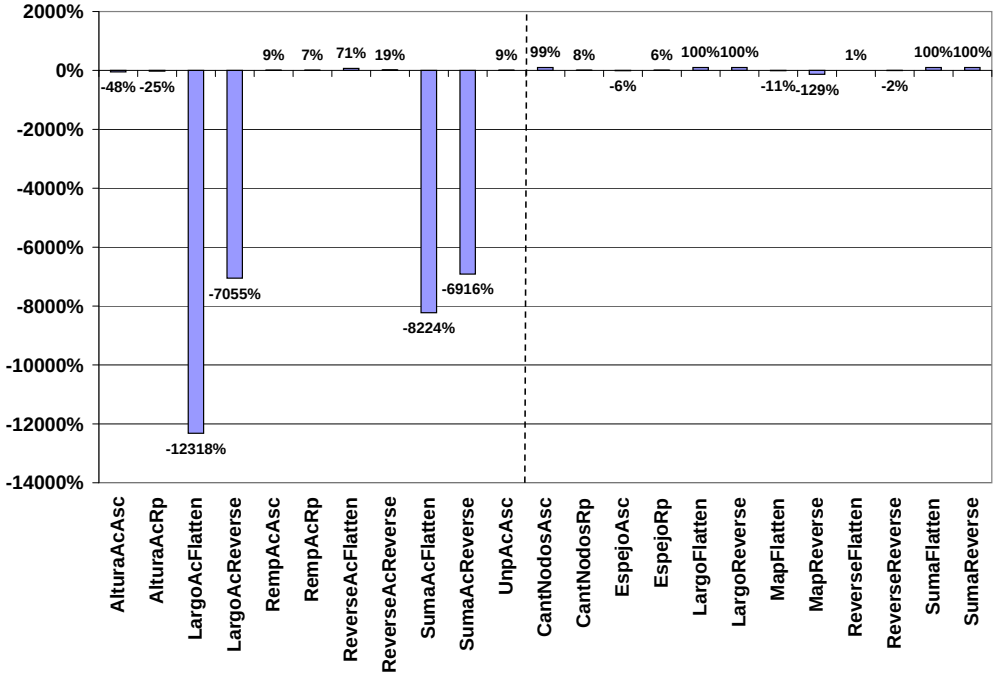


Figura 4.13: Relación T_{FNOp}/T_{OROp}

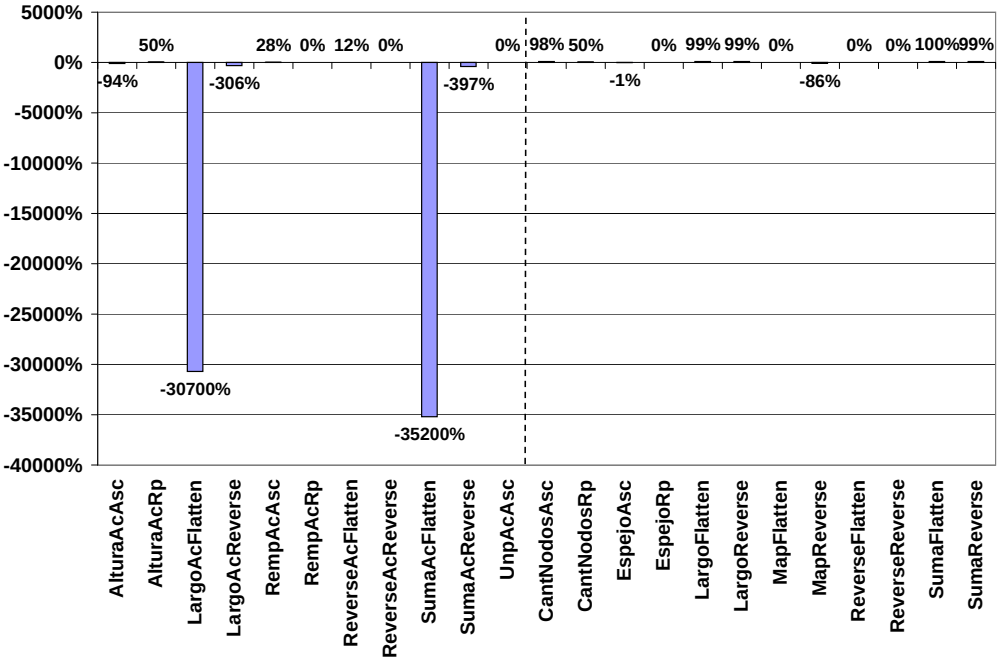


Figura 4.14: Relación E_{FNOp}/E_{OROp}

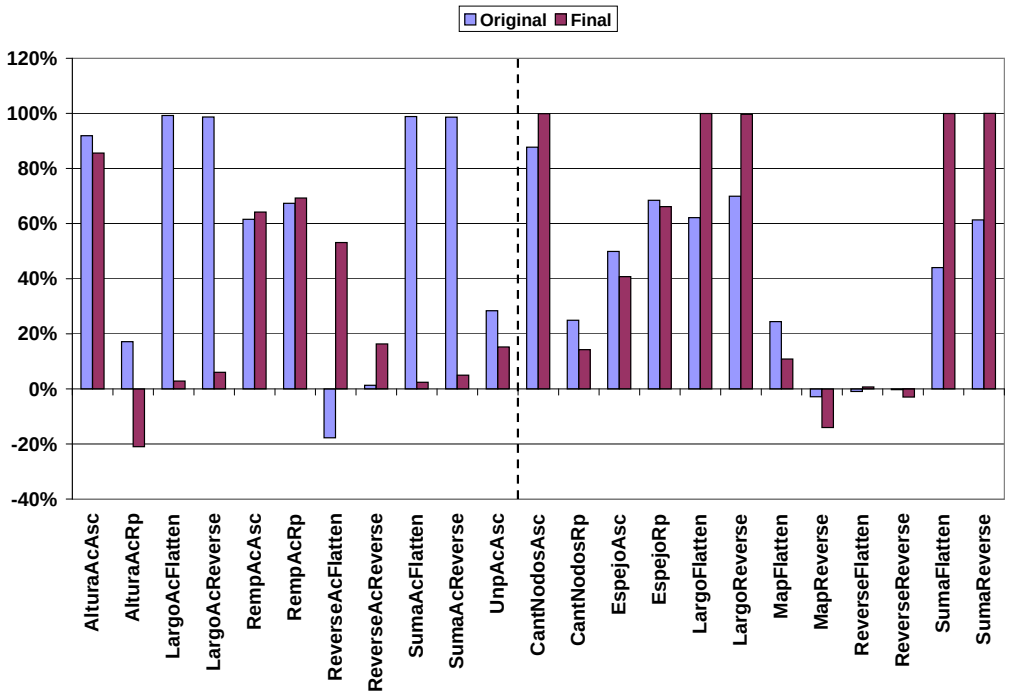


Figura 4.15: % Mejora de las optimizaciones propias del compilador en el tiempo

obtenidos en los ejemplos considerados no es posible obtener una conclusión sobre el efecto combinado de la aplicación de las optimizaciones propias del compilador y la aplicación de la ley de fusión. Determinar las formas de interacción entre las distintas optimizaciones es un punto importante que debe ser considerado para obtener conclusiones definitivas sobre la efectividad de la aplicación de la ley de fusión pero que se encuentra fuera del alcance de esta tesis.

4.4. Resumen

En este capítulo se presentó una ley de fusión que sigue el enfoque de Short-Cut Fusion y al mismo tiempo contempla en forma explícita el uso de acumulaciones como funciones productoras. Junto a la definición de la ley se realizaron pruebas para un conjunto de programas ejemplos, desde el punto de vista del tiempo de ejecución y del espacio de memoria requerido para la misma. Los programas fueron divididos en dos grupos: un primer grupo donde las funciones consumidoras son acumulaciones y el segundo grupo donde no lo son.

Para las pruebas se consideraron ejecuciones de los programas donde no se activaron las optimizaciones propias del compilador. En las pruebas se consideraron distintas versiones de cada uno de los programas, encontrando que los resultados más relevantes corresponden a las comparaciones realizadas entre la versión original y la versión final.

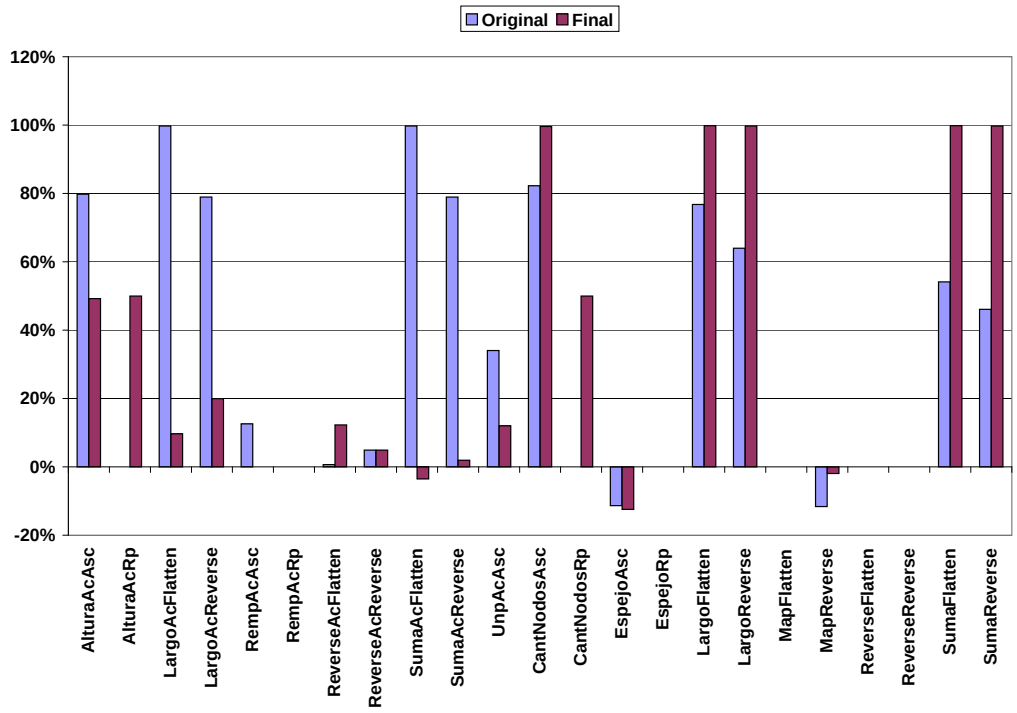


Figura 4.16: % Mejora de las optimizaciones propias del compilador en el espacio

Con respecto al tiempo de ejecución, se observó que en los programas del primer grupo los tiempos de ejecución de las versiones finales se mantienen similares a los de las versiones originales. Estos resultados eran esperados por ser las funciones consumidoras acumulaciones y en consecuencia la codificación de éstas en términos de un fold de alto orden provoca un costo adicional en la versión que resulta de la fusión debido a la manipulación de valores de alto orden los cuales no siempre son posibles de eliminar. Los programas correspondientes al segundo grupo presentan mejoras de hasta un 50 % en los tiempos de ejecución. En este grupo se identifica un programa que se comporta en forma excepcional, *MapReverse*, donde el resultado de aplicar la fusión es una función que empeora el tiempo de ejecución en un 100 %. A partir de observaciones realizadas sobre el uso del heap por parte de este programa en ambas versiones concluimos que su comportamiento se debe a la combinación específica de las funciones consideradas y no a cada una de ellas en particular.

Con respecto al espacio de memoria requerido para las ejecuciones se observó que las diferencias entre las distintas versiones siguen la misma línea de comportamiento que las observadas en los tiempos de ejecución.

Por último se observó que al incluir las optimizaciones propias del compilador en las ejecuciones de los programas los comportamientos que se obtienen son dispares, no siguiendo una tendencia clara, lo que pone en evidencia la necesidad de un estudio más profundo de este punto que debe ser considerado en los trabajos futuros si se pretende aplicar la ley de fusión en general.

5 Conclusiones

En este capítulo se presenta una descripción de otros trabajos en temas afines a esta tesis y que se consideran representativos. Asimismo se da un breve resumen del trabajo realizado y se mencionan posibles trabajos futuros.

Contenido

5.1. Trabajos relacionados

En esta sección se presentan brevemente otros trabajos que también se ocupan del tema de la fusión de funciones y que de una forma u otra consideran a las acumulaciones como posibles funciones involucradas en la composición original.

5.1.1. Lazy composition

Voigtländer [Voi04] desarrolla un algoritmo llamado "lazy composition"que permite transformar composiciones de funciones, entre las cuales pueden figurar acumulaciones, con el objetivo de eliminar las estructuras intermedias. La aplicación de este algoritmo requiere que las funciones involucradas cumplan un conjunto de restricciones sintácticas sobre la forma de su definición (condiciones de linealidad).

El proceso de derivación de la función resultado pasa por diferentes pasos donde se realiza "plegado/desplegado" de definiciones, agregado de parámetros adicionales que representan el alcance de la función consumidora a los parámetros adicionales de la función productora, creación de una tupla resultado y la introducción de definiciones circulares. Este proceso se realiza a partir de un conjunto de reglas de transformación que se basan en las siguientes ideas:

Dada una composición $f_2\ (f_1\ \theta\ \Phi_1\ \cdots\ \Phi_r)\ \Psi_1\ \cdots\ \Psi_s$, donde f_1 tiene r parámetros de contexto y f_2 tiene s, la transformación deriva una función $\overline{f_1 f_2}$ que evita la construcción de la estructura intermedia. Para cada ecuación de f_1, $f_1\ (C\ x_1 \ldots x_k)\ y_1 \ldots y_r = rhs_{f_1,C}$, se define una ecuación para $\overline{f_1 f_2}$:

$$\overline{f_1 f_2}\ (C\ x_1 \ldots x_k)\ y_1 \ldots y_r\ z_1 \ldots z_s = f_2\ rhs_{f_1,C}\ z_1 \ldots z_s$$

que se puede simplificar usando las ecuaciones que definen a la función f_2. De acuerdo a las condiciones requeridas para las funciones involucradas en la composición en la definición de $\overline{f_1 f_2}$ sólo es posible encontrar los siguientes tipos de llamadas a f_2:

1. $f_2\ (D\ \Phi_1 \ldots \Phi_r)\ \Psi_1 \ldots \Psi_s$, siendo D un constructor
2. $f_2\ x_i\ \Psi_1 \ldots \Psi_s$, siendo x_i una variable recursiva
3. $f_2\ y_h\ \Psi_1 \ldots \Psi_s$, siendo y_h una variable de contexto de f_1
4. $f_2\ (f_1\ x_i\ \Phi_1 \ldots \Phi_r)\ \Psi_1 \ldots \Psi_s$, siendo x_i una variable recursiva

En los dos primeros casos no se presentan dificultades. En el primero simplemente se aplica la ecuación que define a f_2 para el constructor D, eliminando de esta forma la construcción de la estructura intermedia. En el segundo caso se mantiene la llamada de f_2 sobre x_i ya que en este caso no hay estructura intermedia para eliminar. Lo distintivo del método se encuentra en la forma en que se resuelven los últimos casos.

En el tercer caso la llamada a la función f_2 debe eliminarse ya que es posible que el parámetro de contexto contenga parte de la estructura intermedia que no se estaría eliminando. En este caso se continuará con el proceso de "plegado/desplegado" pero agregando a $\overline{f_1 f_2}$ nuevos parámetros

que se corresponden con la traducción de los parámetros de contexto de f_1 según f_2. Suponiendo que los resultados de estas traducciones se almacenan en variables y'_1 .... y'_r entonces el lado izquierdo de las ecuaciones que definen a $\overline{f_1 f_2}$ pasan a ser:

$$\overline{f_1 f_2}\ (C\ x_1\ ...\ x_k)\ y_1\ ....\ y_r\ z_1\ ....\ z_s\ y'_1\ ....\ y'_r$$

y en consecuencia, el resultado del caso 3 es el parámetro y'_h.

En el último caso, manteniendo las ideas anteriores esta llamada debe remplazarse por una llamada a $\overline{f_1 f_2}$ sobre x_i con los parámetros de contexto de f_1, los parámetros de contexto de f_2 y al igual que en el caso anterior la traducción de los parámetros de contexto de f_1 según f_2. Esto es la aplicación de la función f_2 a los parámetros de contexto de f_1. Los parámetros necesarios para estas aplicaciones son a su vez parte del resultado de la función $\overline{f_1 f_2}$, introduciendo de esta forma una definición circular. En resumen, la función $\overline{f_1 f_2}$ devuelve una tupla cuya primera componente es el resultado de la composición y las restantes componentes de la tupla corresponden a los parámetros de contexto necesarios para la aplicación de f_2 a los parámetros de contexto de f_1, tal como se muestra en la siguiente definición:

$$(v, v_{1,1}, ...v_{r,s}) = \overline{f_1 f_2}\ x_i\ \Phi_1\ \cdots\ \Phi_r\ \Psi_1\ \cdots\ \Psi_s\ (f_2\ \Phi_1\ v_{1,1}\ ...\ v_{1,s})\ \cdots\ (f_2\ \Phi_r\ v_{r,1}\ ...\ v_{r,s})$$

Este agregado requiere de una adaptación de los casos anteriores y en particular en los casos de tipo 3 es donde se resuelven los valores para estos parámetros.

Por ejemplo, el resultado de aplicar "lazy composition" a la composición *sumaAc* ∘ *areverse* (Ejemplo 4.2.3) es el siguiente:

$$\begin{aligned}
&\mathit{sumrev}\ \mathit{xs}\ y\ z \quad = \mathbf{let}\ (c, c') = \mathit{ars}\ \mathit{xs}\ y\ z\ (\mathit{sumaAc}\ y\ c')\ \mathbf{in}\ c\\
&\mathit{ars}\ [\,] \qquad y\ z\ y' = (y', z)\\
&\mathit{ars}\ (a : \mathit{as})\ y\ z\ y' = \mathbf{let}\ (c, v)\ = \mathit{ars}\ \mathit{as}\ (a : y)\ z\ y'\ \mathbf{in}\ (c, a + v)
\end{aligned}$$

Notar el parámetro adicional en la función *ars* y la circularidad en c'.

Bajo determinadas condiciones sintácticas relativas a la definición de las funciones involucradas al resultado de la transformación se le puede aplicar un postprocesamiento que permite obtener programas más eficientes ya que con la transformación inicial no siempre se logra la eficiencia buscada. Las restricciones de linealidad sobre las funciones productoras excluyen funciones que sí son expresables en términos del operador *buildA*, y por lo tanto válidas como productoras para la aplicación de "Short-Cut Fusion" acumulativa. Una comparación entre "lazy composition" y "Short-Cut Fusion" estándar puede encontrarse en [Voi04]. Las consideraciones en dicha comparación son igualmente válidas para el uso de "Short-Cut Fusion" acumulativa.

En las medidas presentadas en [Voi04] se observa que los resultados obtenidos por la aplicación de "Short-Cut Fusion" son más eficientes que los obtenidos por la aplicación de "lazy composition" pero esto se invierte en el caso de realizar postprocesamiento.

5.1.2. Fusión Algebraica

En [KN08] se presenta una solución uniforme al problema de fusión de funciones en donde tanto la función productora como la consumidora pueden ser acumulaciones. Dada una función

productora $prod :: A \to B \to B$ y una función consumidora $cons :: B \to D$, la idea es definir una función $fuse :: A \to D \to D$ que satisface la ecuación:

$$fuse\ t\ (cons\ u)\ =\ cons\ (prod\ t\ u)$$

y que calcula directamente valores en D a partir de los valores de A sin crear los valores intermedios que son pasados entre *prod* y *cons*.

El método de fusión propuesto, llamado *fusión algebraica*, se basa en la teoría de los monoides. La función productora es fusionada con un homomorfismo de monoides el cual es derivado de la función consumidora. Como resultado se obtiene una función de alto orden que computa sobre el monoide del espacio de las funciones $(D \to D, id, \circ)$.

Los monoides considerados durante el desarrollo del método son: i) el monoide de los contextos sobre B, $(C_B, [-], \bullet)$, donde C_B se obtiene agregando un elemento distinguido $[-]$ a B, un *hueco*, el cual se utiliza para representar la información del acumulador y $\bullet$, operación de sustitución (dados dos contextos sustituye los huecos del primer contexto por el segundo), que cumple el rol de la composición en el monoide; ii) el monoide del espacio de las funciones sobre D.

Para su aplicación, el método requiere que las funciones involucradas en la composición cumplan con ciertas condiciones: la función consumidora debe ser expresable como un *fold*, $cons = fold\ \delta$, y la función productora se debe poder expresar en términos de un *fold* sobre el monoide de los contextos, $prod = fill \circ fold\ \beta$ donde β es un álgebra polinomial y $fill :: C_B \to B \to B$ es un homomorfismo entre el monoide de los contextos y el espacio de las funciones que permite la factorización de la función productora.

Se define una función $\overline{cons}$ que se obtiene extendiendo el dominio y codominio de la función consumidora de forma tal que es posible manipular contextos. Para esto se agrega un parámetro adicional a las funciones del álgebra δ y se agrega una nueva función al álgebra para el caso del símbolo especial de los contextos.

La función $\overline{cons}$ resulta ser un homomorfismo entre el monoide de los contextos y el de las funciones. Dado que la función β es un polinomio es posible calcular la imagen de este polinomio según el homomorfismo: $\overline{cons}\ \beta$. Como resultado de estas construcciones se tiene que $fold\ (\overline{cons}\ \beta)$ corresponde a la función *fuse* buscada.

Existe una gran similitud entre este método de fusión y el método de "Short-Cut Fusion". De hecho en muchos casos se obtienen los mismos resultados. Las diferencias entre los métodos se originan en el enfoque utilizado para la resolución de la fusión, y se refleja en las condiciones exigidas para las funciones productoras. En el caso de "Short-Cut Fusion" las condiciones son menos restrictivas por lo que acepta un conjunto más amplio de funciones como productoras. Por ejemplo la función *rp* que cumple el rol de función productora en el Ejemplo 4.2.5 no es una productora válida para el método de fusión algebraica.

A continuación se presenta el resultado de aplicar este método a la composición $sumaAc \circ areverse$ presentada en el Ejemplo 4.2.3, donde se evidencia la igualdad en los resultados obtenidos por el método de fusión algebraica y "Short-Cut Fusion":

$$fuse\ xs\ acrev\ acsum = sumarev\ xs\ (sumaAc\ acrev)\ acsum$$

$$\begin{aligned} &sumarev && :: Num\ a \Rightarrow [a] \to (a \to b) \to a \to b \\ &sumarev\ [\,]\ w && = w \end{aligned}$$

$$sumarev\ (a : l)\ w \quad = sumarev\ l\ (\lambda x \rightarrow (w\ (a + x)))$$

Por otro lado existe un valor agregado al considerar un conjunto de productoras restringido. El método de Fusión algebraica preserva la estructura de la función productora en forma explícita permitiendo de esta forma conocer la estructura de la función resultado. Esto permite la realización de postprocesamiento luego de la fusión, situación que usualmente no es realizable a partir del resultado de aplicar "Short-Cut Fusion" por no tener por lo general información sobre la estructura del resultado. En esta dirección, bajo determinadas condiciones en la forma de la definición de la función consumidora, en el contexto de fusión algebraica se propone postprocesamiento. Este consiste en factorizar la función resultante de la aplicación del método *fuse* en dos funciones f y h tales que $fuse = f \circ h$ de forma tal de reducir el costo de computar sobre el monoide del espacio de funciones, que involucra la manipulación de clausuras de funciones, pasando a computar en un monoide donde las operaciones sean menos costosas. Si bien existen casos donde este mecanismo permite obtener mejoras con respecto a la función resultante de la aplicación de la fusión algebraica no existe un método que permita realizarla de forma de garantizar su eficacia para todas las funciones resultantes de la aplicación.

5.1.3. Derivación de acumulaciones

A diferencia de nuestro trabajo, en [HIT99] el foco principal es la derivación y manipulación de definiciones acumulativas para funciones dadas en términos de *fold*. En particular, se consideran acumulaciones dadas en términos de *folds* de alto orden. Por ejemplo, la derivación de una acumulación a partir de una función expresada en términos del operador *fold* es la dada por la siguiente regla: Dado un fold de primer orden $fold_F\ \varphi$, un operador binario $\oplus$ con neutro izquierdo e y $\psi = (\oplus) \circ g$,

$$\varphi = g \circ F\ (\oplus) \quad \Rightarrow \quad fold_F\ \varphi\ xs = fold_F\ \psi\ xs\ e$$

Relacionado con la manipulación de acumulaciones se presentan dos reglas: una que permite la fusión de la composición de una función y una acumulación en una nueva acumulación, y la otra que permite realizar transformaciones sobre los parámetros de acumulación. Sean $fold_F\ \varphi$ y $fold_F\ \psi$ dos acumulaciones.

$$(h\circ) \circ \varphi = \psi \circ F\ (h\circ) \quad \Rightarrow \quad h \circ fold_F\ \varphi\ xs = fold_F\ \psi\ xs$$
$$(\circ g) \circ \varphi = \psi \circ F\ (\circ g) \quad \Rightarrow \quad fold_F\ \varphi\ xs \circ g = fold_F\ \psi\ xs$$

Estas reglas presentan similitudes con las reglas de fusión para el operador *afold* definidas en la Sección 3.1.

5.1.4. *destroy / unfold*

Existe una formulación dual de la regla de "Short-Cut Fusion" llamada *destroy / unfold* en la cual se hace abstracción de los destructores de la estructura intermedia. En [Sve02] se hace un análisis de esta regla para el caso en que la estructura intermedia es una lista. Se muestra

también como es posible resolver fusiones con funciones productoras que realizan recursión simultánea sobre múltiples listas, del estilo de la clásica función *zip*, y eventualmente con funciones consumidoras que son acumulaciones del tipo *foldl*.

Al igual que en el caso de la regla *fold / build* la regla *destroy / unfold* requiere que tanto la función productora como la consumidora cumplan ciertas condiciones. La función productora debe ser expresable en términos del operador *unfold*, dual del operador *fold* significando que construye una estructura en forma uniforme. Su definición para listas es:

$$
\begin{aligned}
&\mathit{unfoldr} \quad :: (b \to \mathit{Maybe}\ (a, b)) \to b \to \mathit{List}\ a \\
&\mathit{unfoldr}\ f\ b = \mathbf{case}\ f\ b\ \mathbf{of} \\
&\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nil} \\
&\qquad\qquad\qquad \mathit{Just}\ (a, b') \to a : \mathit{unfoldr}\ f\ b'
\end{aligned}
$$

Por su parte, la función consumidora debe ser expresable en términos de la función *destroy*, que para el caso de las listas está dada por la siguiente definición:

$$
\begin{aligned}
&\mathit{destroy} \quad :: (\forall\ b\ .\ (b \to \mathit{Maybe}\ (a, b)) \to b \to c) \to \mathit{List}\ a \to c \\
&\mathit{destroy}\ g\ xs = g\ \mathit{out}\ xs \\
&\qquad\qquad \mathbf{where} \\
&\qquad\qquad\quad \mathit{out} \qquad\qquad :: \mathit{List}\ a \to \mathit{Maybe}\ (a, \mathit{List}\ a) \\
&\qquad\qquad\quad \mathit{out}\ \mathit{Nil} \qquad\quad = \mathit{Nothing} \\
&\qquad\qquad\quad \mathit{out}\ (\mathit{Cons}\ x\ xs) = \mathit{Just}\ (x, xs)
\end{aligned}
$$

En forma dual a *build*, la función *destroy* especifica como se consume una lista usando para ello el destructor *out*. A partir de estas definiciones se plantea la siguiente ley:

$$\mathit{destroy}\ g\ (\mathit{unfoldr}\ \psi\ e) = g\ \psi\ e$$

Es posible utilizar esta ley para fusionar casos donde, por ejemplo, la función consumidora es una acumulación expresable en términos del operador *foldl*. La definición de *foldl* en términos del operador *destroy* es la siguiente:

$$
\begin{aligned}
&\mathit{foldl}\ f\ (xs, ac) = \mathit{destroy}\ (g\ ac)\ xs \\
&\qquad\qquad\quad \mathbf{where} \\
&\qquad\qquad\quad\ g\ \mathit{acc}\ \mathit{phi}\ xs = \mathbf{case}\ \mathit{phi}\ xs\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \quad\ \to \mathit{acc} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ (a, ys) \to g\ (f\ a\ \mathit{acc})\ \mathit{phi}\ ys
\end{aligned}
$$

Sea por ejemplo la siguiente definición:

$$
\begin{aligned}
&\mathit{bar}\ f\ b\ n\ m \qquad\quad = \mathit{foldl}\ f\ (\mathit{enumFromTo}\ n\ m, b) \\
&\mathit{enumFromTo}\ n\ m = \mathit{unfoldr}\ (\lambda i \to \mathbf{if}\ i > m \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{Just}\ (i, i+1))\ n
\end{aligned}
$$

Considerando la definición anterior de *foldl* en términos de *destroy* es posible aplicar la ley *destroy / unfold* obteniendo:

$$
\begin{array}{l}
bar\ f\ b\ n\ m = g\ b\ n \\
\quad \textbf{where}\ g\ acc\ p = \textbf{if}\ p > m \\
\qquad\qquad\qquad\qquad\ \textbf{then}\ acc \\
\qquad\qquad\qquad\qquad\ \textbf{else}\ g\ (f\ acc\ p)\ (p+1)
\end{array}
$$

5.1.5. *dmap*

En [KGF01] se ataca el problema de la fusión de acumulaciones para un subconjunto de funciones capturadas por un esquema de recursión, llamado *dmap*, que es una extensión a la clásica función *map* obtenida agregando dos nuevos parámetros: una función g que construye los elementos del acumulador y una lista que corresponde con la última parte de la lista a construir. Intuitivamente la definición de *dmap* es:

$$dmap\ f\ g\ [x_1, \cdots, x_n]\ [y_1, \cdots, y_m] = [f\ x_1, \cdots, f\ x_n] +\!\!+ [g\ x_n, \cdots, g\ x_1] +\!\!+ [y_1, \cdots, y_m]$$

Concretamente su definición es:

$$
\begin{array}{l}
dmap\ f\ g\ [\,]\ \ \ \ \ \ \ ys = ys \\
dmap\ \#\ \#(x:xs)\ ys = ys \\
dmap\ \#\ g\ (x:xs)\ ys = dmap\ \#\ g\ xs\ (g\ x:ys) \\
dmap\ f\ \#\ (x:xs)\ ys = f\ x : dmap\ f\ \#\ xs\ ys \\
dmap\ f\ g\ (x:xs)\ \ ys = f\ x : dmap\ f\ g\ xs\ (g\ x:ys)
\end{array}
$$

donde se utiliza el símbolo # como una marca especial, llamada función nula. Frente a la ocurrencia de la función nula *dmap* evita la construcción de ciertas listas en la expresión final. Por ejemplo la función *reverse* puede expresarse como: $reverse\ xs = dmap\ \#\ id\ xs\ [\,]$.

Funciones definidas en términos de *dmap* se pueden fusionar mediante una ley que sigue la línea de la composición de funciones definidas en términos de *map*.

$$
\begin{array}{l}
dmap\ f\ g\ (dmap\ h\ k\ [x_1, \cdots, x_n]\ [y_1, \cdots, y_m])\ [z_1, \cdots, z_l] \\
\quad = dmap\ (f \circ h)\ (f \circ k)\ [x_1, \cdots, x_n] \\
\qquad\quad (dmap\ f\ g\ [y_1, \cdots, y_m] \\
\qquad\qquad (dmap\ (g \circ k)\ (g \circ h)\ [x_1, \cdots, x_n]\ [z_1, \cdots, z_l]))
\end{array}
$$

La aplicación de esta ley evita efectivamente la construcción de la lista intermedia. Se observa que las funciones h y k se aplican dos veces a la lista xs lo que puede ser contraproducente si el costo de estas aplicaciones es relativamente alto y su composición no puede optimizarse. A partir de esta regla general se define un conjunto de simplificaciones para los casos donde alguno de los parámetros de *dmap* corresponde a un valor constante o la función nula. Por ejemplo se plantea la siguiente regla de composición para el caso en que las funciones involucradas tengan a la función nula como primer parámetro :

$$dmap\ \#\ f\ (dmap\ \#\ g\ xs\ [\,])\ zs = (dmap\ (f \circ g)\ \#\ xs\ zs)$$

Utilizando esta simplificación es muy sencillo resolver la composición $reverse \circ reverse$:

$$reverse\ (reverse\ xs) = dmap\ \#\ id\ (dmap\ \#\ id\ xs\ [\,])\ [\,]$$

$$= dmap\ (id \circ id)\ \#\ xs\ [\,]$$
$$= dmap\ id\ \#\ xs\ [\,]$$

Al igual que "Short-Cut Fusion", este método permite fusionar composiciones donde la función productora es una acumulación. Sin embargo tiene la desventaja de ser muy restrictivo en su poder expresivo ya que hay funciones sobre listas que no son expresables en términos de *dmap*.

5.2. Conclusiones

En esta tesis se estudió el problema de fusión de funciones que involucran acumulaciones. Se analizaron dos abordajes con el objetivo de encontrar soluciones que permitan realizar la fusión en forma sencilla, directa y en la medida de lo posible con características que favorezcan su automatización.

En primera instancia se hizo una revisión del operador *afold* propuesto en [Par03], considerando las posibles acumulaciones que se pueden definir en términos de este operador así como las leyes de fusión asociadas al mismo. Se observó que el conocido operador *foldl*, que permite representar funciones con recursión de cola sobre listas, es un caso particular del operador *afold*. Se propuso una modificación en la definición del operador *afold* que permite extender el conjunto de acumulaciones que se pueden representar directamente con el operador. Finalmente se utilizaron las leyes asociadas al operador *afold* para mostrar un ejemplo de fisión, proceso inverso a la fusión, en el estilo de lo realizado en [Gib06].

El énfasis de esta tesis se encuentra en la propuesta de una nueva ley de fusión basada en la técnica de "Short-Cut Fusion" [GLJ93]. En este sentido se definió una modificación a la ley de "Short-Cut Fusion" estándar llamada "Short-Cut Fusion" acumulativa, que maneja explícitamente el acumulador de las funciones productoras. Con el objetivo de analizar la eficacia de la ley propuesta se realizaron una serie de pruebas comparativas relativas al tiempo de ejecución y espacio de memoria requerido entre las versiones originales y transformadas por la aplicación de la ley a un conjunto de programas ejemplo. Dichas pruebas permiten concluir que la aplicación de la ley de fusión es efectiva en la eliminación de las estructuras intermedias, pero este hecho no siempre se refleja en una mejora del tiempo o espacio requerido para la ejecución de los programas. Las mayores dificultades se presentan en el caso en que las funciones consumidoras son acumulaciones ya que las condiciones de aplicación de la ley de fusión hacen que estas se representen mediante un *fold* de alto orden provocando que el resultado de la transformación deba manipular clausuras de funciones. De todas formas se comprobó que al considerar la mejor versión de los programas transformados (versiones recursivas, saturadas en la medida de lo posible) en una amplia mayoría de los casos se obtienen mejoras o por lo menos no se empeoran los resultados al compararlos con la versión original.

Las mejoras encontradas en el espacio de memoria requerido por los programas transformados siguen la dirección de las mejoras en el tiempo de ejecución. Dadas la característica de la ley que apunta a la eliminación de la estructura intermedia la mejora que se obtiene se encuentra muy relacionada con el espacio de memoria requerido por los programas, ya que se estaría realizando

un menor uso del heap y por lo tanto reduciendo el trabajo del garbage collector.

Esta tesis fue desarrollada considerando una semántica de conjuntos siendo posible extenderla a una semántica en términos de órdenes parciales (*cpos*) con el consecuente agregado de ciertas precondiciones a las leyes referentes a la manipulación de valores no definidos.

5.3. Trabajo Futuro

A partir del trabajo realizado se identifican distintos puntos que se consideran interesantes como para avanzar en su estudio.

Automatización Para la aplicación de la ley de fusión es necesario expresar las funciones involucradas en términos de los operadores *fold* y *build*. Por el momento la traducción de las definiciones recursivas dadas inicialmente a sus respectivas expresiones en términos de estos operadores se realiza en forma manual y sería interesante avanzar en una traducción automática en un estilo similar al presentado en [LS95, Chi99].

Programas de mayor complejidad Es necesario estudiar la eficacia de la ley de fusión al considerar programas de mayor dimensión. Los programas considerados en las pruebas realizadas son sencillos ya que se pretendió realizar únicamente una primera evaluación de la ley.

Optimizaciones al programa transformado Otro punto a considerar es la posibilidad de realizar un postprocesamiento al resultado obtenido por la aplicación de la ley de fusión con el objetivo de continuar el proceso de optimización. Como consecuencia de la libertad en la forma de la definición de las funciones productoras que permite la función *build*, al fusionar usando "Short-Cut Fusion" no se tiene información suficiente sobre la estructura de la función resultante. Esto dificulta la posibilidad de definir algún tipo de postprocesamiento del estilo realizado en fusión algebraica [KN08] sobre el programa resultante de la transformación que permita continuar con el proceso de optimización. En este sentido se realizó un intento en la búsqueda de una posible optimización posterior pero sin obtener un avance significativo ya que por el momento no se encontraron propuestas generales.

Se consideraron los casos especiales donde la función productora es o bien la función *areverse* o la función *flatten* y una función consumidora de la forma: $cons = foldl\ (\oplus)$ de acuerdo a la definición de *foldl* vista en la Sección 3.3. Una posible función consumidora en este caso es la función que calcula la suma de los elementos de una lista en forma acumulativa: *sumaAc*.

El resultado de la fusión de este tipo de función consumidora con algunas de estas productoras es de la forma:

$$cons\ (prod\ (t, ac), w) = cons\ (ac, improv\ t\ w)$$

Para el caso de la función *areverse* la función *improv* corresponde a la siguiente función:

$$improv\ [\,]\ w \qquad = w$$

$$improv\ (a : as)\ w = a\ \oplus\ (improv\ as\ w)$$

Considerando el caso de la función consumidora *sumaAc* el resultado de la fusión es:

$$\begin{aligned}
&fus\ (xs, ac)\ w = sumaAc\ (ac, improvRev\ xs\ w)\\
&\quad \textbf{where}\\
&\qquad improvRev\ [\,]\ w \qquad = w\\
&\qquad improvRev\ (a : as)\ w = a + improvRev\ as\ w
\end{aligned}$$

En el Ejemplo 4.2.3 se presentó el resultado de la fusión para esta composición aplicando la ley de "Short-Cut Fusion" acumulativa. En este ejemplo se observó que el acumulador de la función *fus* es efectivamente una función situación que no se repite considerando esta función *improv*.

En el caso de la función *flatten*, si la función $\oplus$ es asociativa y conmutativa la función *improv* es la siguiente:

$$\begin{aligned}
&improv\ (Leaf\ a)\ w\ \ = a\ \oplus\ w\\
&improv\ (Join\ i\ d)\ w = improv\ i\ (improv\ d\ w)
\end{aligned}$$

Interacción de optimizaciones Un punto pendiente es el análisis de la interacción de distintas optimizaciones. Al analizar el efecto del uso de optimizaciones propias del compilador se observó que estas no se realizan en un porcentaje similar en los programas transformados por la ley "Short-Cut Fusion" acumulativa a lo que se obtiene para las versiones originales. Esta situación marca que el resultado de la aplicación de la ley de fusión es tal que se inhibe la aplicación de las optimizaciones propias del compilador.

Bibliografía

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. In *Journal of the ACM*, volume 24(1), pages 44–67, January 1977.

[BdM97] Richard Bird and Oege de Moor. *Algebra of programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[Bir98] Richard S. Bird. *Introduction to Functional Programming Using Haskell.* Prentice-Hall, 1998.

[Chi99] Olaf Chitil. Typer inference builds a short cut to deforestation. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 249–260, New York, NY, USA, 1999. ACM Press.

[Chi00] Olaf Chitil. *Type-Inference Based Deforestation of Functional Programs.* PhD thesis, RWTH Aachen, October 2000.

[GHC] The glasgow haskell compiler. `http://haskell.org/ghc/`.

[Gib06] Jeremy Gibbons. Fission for program comprehension. In Tarmo Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, 2006.

[Gil96] Andrew Gill. *Cheap Deforestation for Non strict Functional Languages.* PhD thesis, University of Glasgow, 1996.

[GLJ93] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, FPCA'93*, page 223–232. ACM Press, 1993.

[GUV04] Neil Ghani, Tarmo Uustalu, and Varmo Vene. Build, augment and destroy. universally. In *In Asian Symposium on Programming Languages, Proceedings*, pages 327–347. Springer-Verlag, 2004.

[HIT99] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Calculating accumulations. *NEW GENERATION COMPUTING*, 17(2):153–173, 1999.

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

[Jas04] Mauro Jaskelioff. Generic accumulations for program calculation. Master's thesis, Universidad Nacional de Rosario, Rosario, Argentina, 2004.

[KGF01] Kazuhiko Kakehi, Robert Glück, and Yoshihiko Futamura. On deforesting parameters of accumulating maps. In *LOPSTR '01: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*, pages 46–56, London, UK, 2001. Springer-Verlag.

[KN08] Shin-Ya Katsumata and Susumu Nishimura. Algebraic fusion of functions with an accumulating parameter and its improvement. *Journal of Functional Programming*, 18(Special Double Issue 5-6):781–819, 2008.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 1988.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *In Conf. on Func. Prog. Languages and Computer Architecture*, pages 314–323. ACM Press, 1995.

[Mee92] Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

[MP09] Mónica Martínez and Alberto Pardo. A shortcut fusion approach to accumulations. In *Anales del XIII Simposio Brasileño de Lenguajes de Programación (SBLP)*, Gramado, Agosto 19-21, Sociedad Brasileña de Computación, 2009.

[Par01] Alberto Pardo. *A Calculational Approach to Recursive Programs with Effects*. PhD thesis, Darmstadt University of Technology, Germany, 2001.

[Par03] Alberto Pardo. Generic accumulations. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 49–78, Deventer, The Netherlands, The Netherlands, 2003. Kluwer, B.V.

[RR96] Colin Runciman and Niklas Röjemo. Heap profiling for space efficiency. In *2nd Intl. School on Advanced Functional Programming*, pages 159–183. Springer LNCS, 1996.

[Sve02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 124–132, New York, NY, USA, 2002. ACM Press.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.

[Voi04] Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17:129–163, 2004. Previous version appeared in *ASIA-PEPM 2002*, Proceedings, pages 126–137, ACM Press, 2002.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings 4th Conference on Functional Programming Languages and Computer Architecture, FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science (Special issue of selected papers from 2nd European Symposium on Programming)*, number 73, pages 231–248, 1990.

Códigos Fuentes

A

En este anexo se presentan los códigos correspondientes a los ejemplos usados en las pruebas realizadas. Se presentan los distintos tipos de datos usados con sus respectivas definiciones de los operadores *fold* y *build*, a continuación se enuncian las instancias de las reglas de reescritura que permiten definir la regla de fusión para cada tipo de datos utilizados. A continuación se presentan las funciones usadas como productoras dando el código correspondiente a la versión original y su expresión en términos de la función *build* así como las funciones utilizadas como consumidoras dando el código de la versión original y su expresión en términos del operador *fold*. Por último se presentan las composiciones utilizadas como ejemplo, dando la expresión recursiva correspondiente al resultado de aplicar la regla de fusión.

Contenido

A.1. Tipos de datos

A continuación se presentan los tipos de datos involucrados en los programas usados como ejemplos para las pruebas. Para cada uno de ellos se da su definición, operador *fold* y función *build* asociada, así como la regla de reescritura que permite expresar la instancia de la ley de "Short Cut Fusion" acumulativa para este tipo.

A.1.1. Lista

- ```
  foldList :: (b,a -> b -> b) -> [a] -> b
  foldList (fnil,fcons) = foldr fcons fnil
  ```

- ```
  buildaList :: (forall b.(b,a -> b -> b) -> (c,b) -> b)
                                          -> (c,[a]) -> [a]
  buildaList g = g ([],(:))
  ```

- ```
  {-# RULES "fold/buildac"
   forall k z t ac
         (g :: forall c. (c,(a -> c -> c)) -> (b,c) -> c).
         foldList (z,k) (buildaList g (t,ac)) =
                                  g (z,k) (t, (foldList (z,k) ac)) #-}
  ```

A.1.2. Term

```
data Term = Num Int | Add Term Term  deriving (Show)
```

- ```
  foldTerm               :: (Int -> b, b -> b -> b) -> Term -> b
  foldTerm (fnum, fadd)  =  fT
                            where fT (Num x)   = fnum x
                                  fT (Add x y) = fadd (fT x) (fT y)
  ```

- ```
  buildaTerm     :: (forall b.(Int -> b,b -> b -> b) -> (c,b) -> b)
                                                     -> (c,Term) -> Term
  buildaTerm g   = g (Num,Add)
  ```

- ```
  {-# RULES "foldTerm/buildacTree"
    forall k z t ac
           (g ::forall c.(Int -> c,c -> c -> c) -> (b,c) -> c) .
           foldTerm (k,z) (buildaTerm g (t,ac)) =
                                    g (k,z) (t,foldTerm (k,z) ac) #-}
  ```

A.1.3. ShapeTree

```
data ShapeTree = SLeaf | SJoin ShapeTree ShapeTree  deriving (Show)
```

- ```
  foldShapeTree :: (b,b -> b -> b) -> ShapeTree -> b
  foldShapeTree (fSLeaf,fSJoin) = fT
                                  where fT SLeaf = fSLeaf
                                        fT (SJoin x y) = fSJoin (fT x) (fT y)
  ```
- ```
  buildAShapeTree :: (forall b.(b,b -> b -> b) -> (c, b) -> b)
                                                  -> (c,ShapeTree) -> ShapeTree
  buildAShapeTree g = g (SLeaf,SJoin)
  ```
- ```
  {-# RULES "fold/buildAShapeTree"
   forall k z t ac
   (g :: forall b. (b,b -> b -> b) -> (c,b) -> b).
    foldShapeTree (k,z) (buildAShapeTree g (t,ac)) =
                                         g (k,z) (t,foldShapeTree (k,z) ac) #-}
  ```

A.2. Funciones productoras

A.2.1. asc

Original

```
asc               :: (Term,Term) -> Term
asc (Num x,z)     = Add z (Num x)
asc (Add x y,z)   = asc (x, asc (y,z))
```

Función en términos de *build*

```
asc  :: (Term,Term) -> Term
asc  = buildaTerm gasc
          where
           gasc (fnum,fadd) (Num x,z)   = fadd z (fnum x)
           gasc (fnum,fadd) (Add x y,z) =
                            gasc (fnum,fadd) (x,gasc (fnum,fadd) (y,z))
```

A.2.2. flatten

Original
```

```
flatten               :: (Btree a,[a]) -> [a]
flatten (Leaf a,w)    = a:w
flatten (Join i d, w) = flatten (i, flatten (d, w))
```

En términos de la función *build*

```
flatten :: (Btree a, [a]) -> [a]
flatten = buildaList gflat
           where
            gflat (fnil,fcons) (Leaf a, ac)   = fcons a ac
            gflat (fnil,fcons) (Join i d, ac) = gflat (fnil,fcons)
                                                      (i,gflat (fnil,fcons) (d,ac))
```

A.2.3. reverse

Original

```
reverseA         :: ([a],[a]) -> [a]
reverseA ([],w)  = w
reverseA (a:x,w) = reverseA (x,a:w)
```

En términos de la función *build*

```
reverseA :: ([a],[a]) -> [a]
reverseA = buildaList grev
              where
                grev (fnil,fcons) ([],ac)  = ac
                grev (fnil,fcons) (a:x,ac) = grev (fnil,fcons) (x,fcons a ac)
```

A.2.4. reverseMod

Original

```
areverseMod          :: ([a] ,[a]) -> [a]
areverseMod ([],w)   = w
areverseMod (a:x,w)  = areverseMod (x,(a+1):w)
```

En términos de la función *build*

```
areverseMod  :: ([a],[a]) -> [a]
areverseMod  = buildList grevM
                  where
                    grevM (fnil,fcons) ([],ac)    = ac
                    grevM (fnil,fcons) (a:xs,ac)  = grevM (fnil,fcons) (xs,fcons (a+1) ac)
```

A.2.5. rp

Original

```
rp                :: (ShapeTree,ShapeTree) -> ShapeTree
rp (SLeaf,w)      = w
rp (SJoin l r,w) = SJoin (rp (l,SLeaf)) (rp (r,w))
```

En términos de la función *build*

```
rp :: (ShapeTree,ShapeTree) -> ShapeTree
rp = buildAShapeTree grp
      where
       grp (fSLeaf,fSJoin) (SLeaf,w)     = w
       grp (fSLeaf,fSJoin) (SJoin i d,w) = fSJoin (grp (fSLeaf,fSJoin) (i,fSLeaf))
                                                  (grp (fSLeaf,fSJoin) (d,w))
```

A.3. Funciones consumidoras

A.3.1. Altura Acumulativa

Original

```
alturaAc               :: (Num a, Ord a) => Term -> a -> a
alturaAc (Num x) ac    = ac
alturaAc (Add x y) ac  = 1 + max (alturaAc x ac) (alturaAc y ac)
```

En términos del operador *fold*

```
alturaAc  :: (Num a, Ord a) => Term -> a -> a
alturaAc  = foldTerm (galt,halt)

halt          :: (Num a, Ord a) => (b -> a) -> (b -> a) -> b -> a
halt fx fy z  = 1 + max (fx z) (fy z)
galt          :: a -> b -> b
galt x ac     = ac
```

A.3.2. Cantidad de Nodos

Original

```
cantNodos           :: Num a => Term -> a
cantNodos (Num x)   = 1
cantNodos (Add l r) = 1 + (cantNodos l) + (cantNodos r)
```

En términos del operador *fold*

```
cantNodos :: Term -> Integer
cantNodos = foldTerm (hnum,contar)

contar     :: Num a => a -> a -> a
contar i d = 1 + i + d

hnum       :: Num a => b -> a
hnum a     = 1
```

A.3.3. Espejo

Original

```
espejo           :: Term -> Term
espejo (Num x)   = Num x
espejo (Add l r) = Add (espejo r) (espejo l)
```

En términos del operador *fold*

```
espejo      :: Term -> Term
espejo      = foldTerm (Num,hespejo)

hespejo     :: Term -> Term -> Term
hespejo i d = Add d i
```

A.3.4. Largo

Original

```
largo        :: Num a => [b] -> a
largo []     = 0
largo (a:s)  = 1 + largo s
```

En términos del operador *fold*

```
largo :: Num a => [b] -> a
largo = foldList (0,suc2)

suc2     :: Num a => b -> a -> a
suc2 x y = y + 1
```

A.3.5. Largo acumulativo

Original

```
largoAc         :: Num a => [b] -> a -> a
largoAc [] w    = w
largoAc (a:s) w = largoAc s (1+w)
```

En términos del operador *fold*

```
largoAc :: Num a => [b] -> a -> a
largoAc = foldList (id,hlargo)

hlargo         :: Num a => b -> (a -> c) -> a -> c
hlargo a rec x = rec (1+x)
```

A.3.6. Map

Original

```
map          :: (a -> b) -> [a] -> [b]
map f []     = []
map f (a:as) = f a : mapMo f as
```

En términos del operador *fold*

```
map    :: (a -> b) -> [a] -> [b]
map f  = foldList ([],hmap f)
            where hmap g a rec = g a : rec
```

A.3.7. Remplazar acumulativo

Original

```
rempAc              :: Term -> Term -> Term
rempAc (Num x) ac   = ac
rempAc (Add x y) ac = Add (rempAc x ac) (rempAc y ac)
```

En términos del operador *fold*

```
rempAc :: Term -> Term -> Term
rempAc = foldTerm (gremp,hremp)

gremp     :: a -> b -> b
```

```
gremp a b = b

hremp         :: (a -> Term) -> (a -> Term) -> a -> Term
hremp fx fy z = Add (fx z) (fy z)
```

A.3.8. Reverse

Original

```
reverse        :: [a] ->  [a]
reverse []     = []
reverse (a:x)  = rev x ++ [a]
```

En términos del operador *fold*

```
reverse :: [a] -> [a]
reverse = foldList ([],hrev)

hrev       :: a -> [a] -> [a]
hrev a rec = rec ++ [a]
```

A.3.9. Reverse acumulativo

Original

```
reverseAc         :: [a] -> [a] -> [a]
reverseAc [] w    = w
reverseAc (a:x) w = reverseAc x (a:w)
```

En términos del operador *fold*

```
reverseAc :: [a] -> [a] -> [a]
reverseAc = foldList (id,hrev)

hrev          :: a -> ([a] -> b) -> [a] -> b
hrev a rec x  =  rec (a:x)
```

A.3.10. Suma

Original

```
suma        :: Num a => [a] -> a
suma  []    = 0
suma (a:s)  = a + suma s
```

En términos del operador *fold*

```
suma :: Num a => [a] -> a
suma = foldList (0,(+))
```

A.3.11. Suma acumulativa

Original

```
sumaAc          :: Num a => [a] -> a -> a
sumaAc [] n     = n
sumaAc (a:l) n = sumaAc l (a+n)
```

En términos del operador *fold*

```
sumaAc :: Num a => [a] -> a -> a
sumaAc = foldList (id,hsum)

hsum           :: Num a => a -> (a -> b) -> (a -> b)
hsum a rec x   =  rec (a+x)
```

A.3.12. Unp acumulativo

Original

```
unp              :: Term -> String -> String
unp (Num x) z    = shows x z
unp (Add x y) z =  unp x ('+':unp y z)
```

En términos del operador *fold*

```
unp :: Term -> String -> String
unp = foldTerm (gunp,hunp)

gunp   :: Int -> (String -> String)
gunp n = shows n

hunp           :: (String -> String) -> (String -> String) -> (String -> String)
hunp fx fy s   = fx  ('+': (fy s))
```

A.4. Ejemplos

A continuación se presentan cada una de las composiciones tomadas como ejemplo para las pruebas. Para cada una de ellas se presenta la composición original y como se resuelve la composición usando la función recursiva correspondiente al resultado de aplicar la regla de fusión.

AlturaAcAsc

Composición de altura acumulativa con asc definidas en A.3.1 y A.2.1 respectivamente.

```
comp term1 ac1 ac2 = alturaAc (asc (term1,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2    = fus (term, alturaAc ac1) ac2

fus                  :: (Num a, Ord a) => (Term,a -> a) -> a -> a
fus (Num x,z) ac     = 1 + max (z ac) ac
fus (Add x y, z) ac  = fus (x,fus (y,z)) ac
```

AlturaAcRp

Composición de altura acumulativa con Rp definidas en A.3.1 y A.2.5 respectivamente.

```
comp term ac1 ac2 = alturaAc (rp (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2     = fus (term,alturaAc ac1) ac2

fus                   :: (Num a, Ord a) => (ShapeTree,a -> a) -> a -> a
fus (SLeaf,w) ac      = w ac
fus (SJoin i d,w) ac  = 1 + max (fus (i,id) ac) (fus (d,w) ac)
```

CantNodosAsc

Composición de la función cantidad de nodos y asc definidas en A.3.2 y A.2.1 respectivamente.

```
comp term ac = cantNodos (asc (term,ac))
```

Versión final recursiva:

```
comp term ac    = fus (term, cantNodos ac)

fus             :: Num a => (Term,a) -> a
fus (Num x,z)   = 1 + z + 1
fus (Add x y,z) = fus (x,fus (y,z))
```

CantNodosRp

Composición de la función cantidad de nodos y rp definidas en A.3.2 y A.2.5 respectivamente.

```
comp term ac = cantNodos (rp (term,ac))
```

Versión final recursiva:

```
comp term ac      = fus (term, cantNodos ac)

fus               :: Num a => (ShapeTree,a) -> a
fus (SLeaf,w)     = w
fus (SJoin i d,w) = 1 + (fus (i,0)) +  (fus (d,w))
```

EspejoAsc

Composición de la función espejo y asc definidas en A.3.3 y A.2.1 respectivamente.

```
comp term ac = espejo (asc (term,ac))
```

Versión final recursiva:

```
comp term ac    = fus (term, espejo ac)

fus             :: (Term,Term) -> Term
fus (Num x,z)   = Add (Num x) z
fus (Add x y,z) = fus (x,fus (y,z))
```

EspejoRp

Composición de la función espejo y rp definidas en A.3.3 y A.2.5 respectivamente.

```
comp term ac = espejo (rp (term,ac))
```

Versión final recursiva:

```
comp term ac      = fus (term, espejo ac)

fus               :: (ShapeTree,ShapeTree) -> ShapeTree
fus (SLeaf,w)     = w
fus (SJoin i d,w) = SJoin (fus (d,w)) (fus (i,SLeaf))
```

LargoAcFlatten

Composición de la función largo acumulativo y flatten definidas en A.3.5 y A.2.2 respectivamente.

```
comp term ac1 ac2 = largoA (flatten (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2      = fus (term, largoA ac1) ac2

fus                    :: Num a => (Btree b,a -> c) -> a -> c
fus (Leaf a, ac) ac2   = ac (1 + ac2)
fus (Join i d, ac) ac2 = fus (i,fus (d,ac)) ac2
```

LargoAcReverse

Composición de la función largo acumulativo y reverse definidas en A.3.5 y A.2.3 respectivamente.

```
comp term ac1 ac2 = largoA (reverseA (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2 = fus (term, largoA ac1) ac2

fus               :: Num a => ([b],a -> c) -> a -> c
fus ([],ac) ac2   = ac ac2
fus (a:x,ac) ac2  = fus (x,hlargo a ac) ac2

hlargo            :: Num a => b -> (a -> c) -> a -> c
hlargo a rec x    = rec (1+x)
```

LargoFlatten

Composición de la función largo y flatten definidas en A.3.4 y A.2.2 respectivamente.

```
comp term ac = largo (flatten (termino,ac))
```

Versión final recursiva:

```
comp term ac       = fus (term, (largo ac))

fus                :: Num a => (Btree b,a) -> a
fus (Leaf a, ac)   =  ac + 1
fus (Join i d, ac) = fus (i, fus(d,ac))
```

LargoReverse

Composición de la función largo y reverse definidas en A.3.4 y A.2.3 respectivamente.

```
comp term ac = largo (reverseA (term,ac))
```

Versión final recursiva:

```
comp term ac = fus (term, (largo ac))

fus          :: Num a => ([b],a) -> a
fus ([],ac)  = ac
fus (a:x,ac) = fus (x, ac + 1)
```

MapFlatten

Composición de la función map y flatten definidas en A.3.6 y A.2.2 respectivamente.

```
comp f term ac = mapMo f (flatten (term,ac))
```

Versión final recursiva:

```
comp f term ac       = fus f (term, mapMo f ac)

fus                  :: (a -> b) -> (Btree a,[b]) -> [b]
fus f (Leaf a, ac)   = f a : ac
fus f (Join i d, ac) = fus f (i,fus f (d,ac))
```

MapReverse

Composición de la función map y reverse definidas en A.3.6 y A.2.3 respectivamente.

```
comp f term ac = mapMo f (reverseA (term,ac))
```

Versión final recursiva:

```
comp f term ac = fus f (term, mapMo f ac)

fus             :: (a -> b) -> ([a],[b]) -> [b]
fus f ([],ac)   = ac
fus f (a:x,ac)  = fus f (x,f a: ac)
```

MapReverseMod

Composición de la función map y aeverseMod definidas en A.3.6 y A.2.4 respectivamente.

```
comp f term ac = mapMo f (areverseMod (term,ac))
```

Versión final recursiva:

```
comp f term ac = fus f (term, mapMo f ac)

fus :: Num a => (a -> b) -> ([a],[b]) -> [b]
fus f ([],ac)  = ac
fus f (a:x,ac) = fus f (x, (f a)+1 : ac)
```

RempAcAsc

Composición de la función reemplazar y asc definidas en A.3.7 y A.2.1 respectivamente.

```
comp term ac1 ac2 = rempAc (asc (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2  = fus (term, rempAc ac1) ac2

fus                :: (Term,Term -> Term) -> Term -> Term
fus (Num x, z) ac  = Add (z ac) ac
fus (Add x y,z) ac = fus (x,fus (y,z)) ac
```

RempAcRp

Composición de la función reemplazar y rp definidas en A.3.7 y A.2.5 respectivamente.

```
comp term ac1 ac2 = rempAc (rp (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2    = fus (term, rempAc ac1) ac2

fus                  :: (ShapeTree,ShapeTree -> ShapeTree) -> ShapeTree -> ShapeTree
fus (SLeaf,w) ac     = w ac
fus (SJoin i d,w) ac = SJoin ((fus (i,id)) ac) ((fus (d,w)) ac)
```

ReverseAcFlatten

Composición de la función reverse acumulativa y flatten definidas en A.3.9 y A.2.2 respectivamente.

```
comp term ac1 ac2 = reverseAc (flatten (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2 = fus (term, reverseAc ac1) ac2

fus                      :: (Btree a,[a] -> b) -> [a] -> b
fus (Leaf a, ac)   ac2   = ac (a:ac2)
fus (Join i d, ac) ac2   = fus (i,fus (d,ac)) ac2
```

ReverseAcReverse

Composición de la función reverse acumulativo con si mismo definida en A.3.9 y A.2.3.

```
comp term ac1 ac2 = reverseAc (reverseA (term, ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2 = fus (term, reverseAc ac1) ac2

fus                :: ([a],[a] -> b) -> [a] -> b
fus ([],ac)  ac2   = ac  ac2
fus (a:x,ac) ac2   = fus (x,hrev a ac) ac2

hrev               :: a -> ([a] -> b) -> [a] -> b
hrev a rec x       = rec (a:x)
```

ReverseFlatten

Composición de la función reverse y flatten definidas en A.3.8 y A.2.2 respectivamente.

```
comp term ac = rev (flatten (term,ac))
```

Versión final recursiva:

```
comp term ac       = fus (term, rev ac)

fus                :: (Btree a,[a]) -> [a]
fus (Leaf a, ac)   = ac ++ [a]
fus (Join i d, ac) = fus (i,fus(d,ac))
```

ReverseReverse

Composición de la función reverse y reverse acumulativo definidas en A.3.8 y A.2.3 respectivamente.

```
comp term ac = rev (reverseA (term, ac))
```

Versión final recursiva:

```
comp term ac = fus (term, rev ac)

fus          :: ([a],[a]) -> [a]
fus ([],ac)  = ac
fus (a:x,ac) = fus (x,ac ++ [a])
```

SumaAcFlatten

Composición de la función suma acumulativa y flatten definidas en A.3.11 y A.2.2 respectivamente.

```
comp term ac1 ac2 = sumaA (flatten (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2      = fus (term, sumaA ac1) ac2

fus                    :: Num a => (Btree a,a -> b) -> a -> b
fus (Leaf a, ac)   ac2 = ac (a + ac2)
fus (Join i d, ac) ac2 = fus (i, fus (d,ac)) ac2
```

SumaAcReverse

Composición de la función suma acumulativa y reverse definidas en A.3.11 y A.2.3 respectivamente.

```
comp term ac1 ac2 = sumaA (reverseA (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2 = fus (term, sumaA ac1) ac2

fus               :: Num a => ([a],a -> b) -> a -> b
fus ([],ac)  ac2  = ac ac2
fus (a:x,ac) ac2  = fus (x,hsum a ac) ac2

hsum              :: Num a => a -> (a -> b) -> (a -> b)
hsum a rec x      = rec (a+x)
```

SumaFlatten

Composición de la función suma y flatten definidas en A.3.10 y A.2.2 respectivamente.

```
comp term ac = suma (flatten (term,ac))
```

Versión final recursiva:

```
comp term ac       = fus (term, suma ac)

fus                :: Num a => (Btree a,a) -> a
fus (Leaf a, ac)   =  a + ac
fus (Join i d, ac) = fus (i,fus (d,ac))
```

SumaReverse

Composición de la función suma y reverse definidas en A.3.10 y A.2.3 respectivamente.

```
comp term ac = suma (reverseA (term,ac))
```

Versión final recursiva:

```
comp term ac = fus (term, suma ac)

fus          :: Num a => ([a],a) -> a
fus ([],ac)  = ac
fus (a:x,ac) = fus (x, a + ac)
```

UnpAcAsc

Composición de la función cantidad de nodos y rp definidas en A.3.2 y A.2.5 respectivamente.

```
comp term ac1 ac2 = unp (asc (term,ac1)) ac2
```

Versión final recursiva:

```
comp term ac1 ac2   = fus (term, unp ac1) ac2

fus                 :: (Term,String -> String) -> String -> String
fus (Num x,z)  ac   = z ('+':((shows x) ac))
fus (Add x y,z) ac  = fus (x,fus (y,z)) ac
```

Datos de las Medidas

B

En este anexo se presentan los valores del tiempo y espacio de memoria requerido para la ejecución de las distintas pruebas realizadas que permiten obtener las gráficas presentadas en el capítulo 4.

Contenido

B.1. Medidas del Tiempo de ejecución

B.1.1. Valores Absolutos

Los valores que se presentan a continuación están dados en segundos.

Composición	Tiempos No Optimizados			Tiempos Optimizados		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
AlturaAcAsc	0,03	5,9	5,93	0,16	0,32	0,48
AlturaAcRp	10,99	1,11	12,1	7,97	2,06	10,03
LargoAcFlatten	0,6	13,83	14,42	0,06	0,04	0,11
LargoAcReverse	1,03	44,24	45,27	0,09	0,51	0,6
RempAcAsc	2,25	15,98	18,23	0,85	6,16	7,01
RempAcRp	8,01	23,99	32	4,14	6,3	10,44
ReverseAcFlatten	0,45	3,9	4,34	0,56	4,55	5,11
ReverseAcReverse	0,77	3,9	4,67	0,49	4,12	4,61
SumaAcFlatten	0,63	14,2	14,83	0,12	0,05	0,17
SumaAcReverse	1,05	43,77	44,82	0,07	0,55	0,62
UnpAcAsc	0,18	1,62	1,8	0,17	1,12	1,29

Cuadro B.1: Tiempos Absolutos - Versión Original - Consumidoras Acumulativas

Composición	Tiempos No Optimizados			Tiempos Optimizados		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
CantNodosAsc	0,15	72,03	72,18	4	4,85	8,85
CantNodosRp	39,57	4,37	43,94	26,74	6,25	32,99
EspejoAsc	3,57	1,62	5,19	1,07	1,53	2,6
EspejoRp	4,82	12,11	16,94	2,06	3,28	5,34
LargoFlatten	0,3	121,86	122,16	29,84	16,36	46,2
LargoReverse	0,13	14,06	14,18	2,62	1,64	4,26
MapFlatten	12,58	1,39	13,97	9,2	1,36	10,56
MapReverse	0,35	1,41	1,76	0,3	1,51	1,81
ReverseFlatten	23,71	165,48	189,19	23,34	167,66	191
ReverseReverse	9,98	77,75	87,73	10,01	77,98	87,99
SumaFlatten	0,27	121,52	121,79	30,85	37,31	68,16
SumaReverse	0,24	14,2	14,44	2,58	3	5,58

Cuadro B.2: Tiempos Absolutos - Versión Original- Consumidoras No Acumulativas

Composición	Tiempos No Optimizados			Tiempos Optimizados		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
AlturaAcAsc	0,03	4,91	4,94	0,02	0,69	0,71
AlturaAcRp	10,29	0,05	10,34	12,5	0,01	12,51
LargoAcFlatten	0,57	13,49	14,06	0,47	13,19	13,66
LargoAcReverse	0,92	44,74	45,67	0,5	42,43	42,93
RempAcAsc	2,26	15,55	17,81	0,95	5,44	6,38
RempAcRp	8,84	22,61	31,45	3,82	5,86	9,67
ReverseAcFlatten	0,68	2,52	3,2	0,52	0,97	1,5
ReverseAcReverse	0,82	3,65	4,47	0,58	3,16	3,74
SumaAcFlatten	0,7	13,8	14,5	0,42	13,73	14,15
SumaAcReverse	1	44,78	45,77	0,64	42,86	43,5
UnpAcAsc	0,17	1,21	1,38	0,15	1,02	1,17

Cuadro B.3: Tiempos Absolutos - Versión Final - Consumidoras Acumulativas

Composición	Tiempos No Optimizados			Tiempos Optimizados		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
CantNodosAsc	0,12	51,92	52,04	0,06	0	0,06
CantNodosRp	35,14	0,08	35,23	30,2	0,01	30,21
EspejoAsc	3,1	1,54	4,64	1,21	1,54	2,75
EspejoRp	4,25	10,61	14,86	1,84	3,19	5,03
LargoFlatten	0,26	98,65	98,91	0,07	0	0,07
LargoReverse	0,33	6,04	6,37	0,02	0	0,02
MapFlatten	10,66	2,45	13,11	9,05	2,64	11,69
MapReverse	0,56	3,07	3,63	0,36	3,78	4,14
ReverseFlatten	23,67	165,8	189,47	24,19	163,99	188,18
ReverseReverse	9,32	77,97	87,29	10,04	79,82	89,87
SumaFlatten	0,28	83,25	83,52	0,06	0	0,06
SumaReverse	0,34	5,09	5,43	0	0	0

Cuadro B.4: Tiempos Absolutos - Versión Final - Consumidoras No Acumulativas

Composición	Ejecución No Optimizados		
Nombre	MUT	GC	TOTAL
AlturaAcAsc	0,05	9,47	9,52
AlturaAcRp	15,1	3,39	18,49
LargoAcFlatten	0,62	14,18	14,79
LargoAcReverse	1,2	46,35	47,55
RempAcAsc	1,56	13,06	14,62
RempAcRp	9,41	25,08	34,49
ReverseAcFlatten	0,66	5,22	5,88
ReverseAcReverse	1,64	6,14	7,78
SumaAcFlatten	0,66	14,4	15,06
SumaAcReverse	1,15	45,58	46,73
UnpAcAsc	0,29	2,29	2,58

Cuadro B.5: Tiempos Absolutos - Versión Manual - Consumidoras Acumulativas

Composición	Ejecución No Optimizados		
Nombre	MUT	GC	TOTAL
CantNodosAsc	0,18	68,46	68,64
CantNodosRp	38	0,15	38,15
EspejoAsc	3,78	1,6	5,38
EspejoRp	4,76	11,32	16,07
LargoFlatten	0,27	151,77	152,04
LargoReverse	0,37	6,02	6,39
MapFlatten	12,02	2,8	14,81
MapReverse	0,4	2,92	3,32
ReverseFlatten	24,25	166,86	191,1
ReverseReverse	9,62	78,72	88,34
SumaFlatten	0,19	128,16	128,35
SumaReverse	0,3	5,05	5,36

Cuadro B.6: Tiempos Absolutos - Versión Manual - Consumidoras No Acumulativas

Composición	Tiempos Optimizados		
Nombre	MUT	GC	TOTAL
AlturaAcAsc	0,75	1,44	2,19
AlturaAcRp	12,98	5,3	18,29
LargoAcFlatten	0,62	13,45	14,07
LargoAcReverse	1,02	45,29	46,31
RempAcAsc	0,63	5,96	6,6
RempAcRp	4,2	6,62	10,82
ReverseAcFlatten	0,91	5,68	6,59
ReverseAcReverse	1,04	6,45	7,49
SumaAcFlatten	0,58	14,29	14,87
SumaAcReverse	0,97	45,03	46
UnpAcAsc	0,2	2,28	2,48

Cuadro B.7: Tiempos Absolutos - Versión SCA - Consumidoras Acumulativas

Composición	Tiempos Optimizados		
Nombre	MUT	GC	TOTAL
CantNodosAsc	0,06	15,06	15,11
CantNodosRp	28,12	0,04	28,17
EspejoAsc	0,99	1,8	2,79
EspejoRp	2,21	3,08	5,29
LargoFlatten	0,25	61,17	61,42
LargoReverse	0,3	5,88	6,18
MapFlatten	10,57	2,62	13,19
MapReverse	0,32	2,46	2,78
ReverseFlatten	24,97	165,62	190,59
ReverseReverse	9,58	80,01	89,59
SumaFlatten	0,26	51,76	52,02
SumaReverse	0,31	4,96	5,27

Cuadro B.8: Tiempos Absolutos - Versión SCA - Consumidoras No Acumulativas

B.1.2. Porcentajes de Mejora

Ejecución No Optimizada						
Composición	Manual/Original			Final/Original		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
AlturaAcAsc	-67 %	-61 %	-61 %	0 %	17 %	17 %
AlturaAcRp	-37 %	-205 %	-53 %	6 %	95 %	15 %
LargoAcFlatten	-3 %	-3 %	-3 %	5 %	2 %	2 %
LargoAcReverse	-17 %	-5 %	-5 %	11 %	-1 %	-1 %
RempAcAsc	31 %	18 %	20 %	0 %	3 %	2 %
RempAcRp	-17 %	-5 %	-8 %	-10 %	6 %	2 %
ReverseAcFlatten	-47 %	-34 %	-35 %	-51 %	35 %	26 %
ReverseAcReverse	-113 %	-57 %	-67 %	-6 %	6 %	4 %
SumaAcFlatten	-5 %	-1 %	-2 %	-11 %	3 %	2 %
SumaAcReverse	-10 %	-4 %	-4 %	5 %	-2 %	-2 %
UnpAcAsc	-61 %	-41 %	-43 %	6 %	25 %	23 %

Cuadro B.9: %Mejora Tiempo - Consumidoras Acumulativas

Ejecución No Optimizada						
Composición	Manual/Original			Final/Original		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
CantNodosAsc	-20 %	5 %	5 %	20 %	28 %	28 %
CantNodosRp	4 %	97 %	13 %	11 %	98 %	20 %
EspejoAsc	-6 %	1 %	-4 %	13 %	5 %	11 %
EspejoRp	1 %	7 %	5 %	12 %	12 %	12 %
LargoFlatten	10 %	-25 %	-24 %	13 %	19 %	19 %
LargoReverse	-185 %	57 %	55 %	-154 %	57 %	55 %
MapFlatten	4 %	-101 %	-6 %	15 %	-76 %	6 %
MapReverse	-14 %	-107 %	-89 %	-60 %	-118 %	-106 %
ReverseFlatten	-2 %	-1 %	-1 %	0 %	0 %	0 %
ReverseReverse	4 %	-1 %	-1 %	7 %	0 %	1 %
SumaFlatten	30 %	-5 %	-5 %	-4 %	31 %	31 %
SumaReverse	-25 %	64 %	63 %	-42 %	64 %	62 %

Cuadro B.10: %Mejora Tiempo - Consumidoras No Acumulativas

Ejecución Optimizada						
Composición	SCA/Original			Final/Original		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
AlturaAcAsc	-369 %	-350 %	-356 %	88 %	-116 %	-48 %
AlturaAcRp	-63 %	-157 %	-82 %	-57 %	100 %	-25 %
LargoAcFlatten	-933 %	-33525 %	-12691 %	-683 %	-32875 %	-12318 %
LargoAcReverse	-1033 %	-8780 %	-7618 %	-456 %	-8220 %	-7055 %
RempAcAsc	26 %	3 %	6 %	-12 %	12 %	9 %
RempAcRp	-1 %	-5 %	-4 %	8 %	7 %	7 %
ReverseAcFlatten	-63 %	-25 %	-29 %	7 %	79 %	71 %
ReverseAcReverse	-112 %	-57 %	-62 %	-18 %	23 %	19 %
SumaAcFlatten	-383 %	-28480 %	-8647 %	-250 %	-27360 %	-8224 %
SumaAcReverse	-1286 %	-8087 %	-7319 %	-814 %	-7693 %	-6916 %
UnpAcAsc	-18 %	-104 %	-92 %	12 %	9 %	9 %

Cuadro B.11: %Mejora Tiempo - Ej. Optimizadas - Consumidoras Acumulativas

Ejecución Optimizada						
Composición	SCA/Original			Final/Original		
Nombre	MUT	GC	TOTAL	MUT	GC	TOTAL
CantNodosAsc	99 %	-211 %	-71 %	99 %	100 %	99 %
CantNodosRp	-5 %	99 %	15 %	-13 %	100 %	8 %
EspejoAsc	7 %	-18 %	-7 %	-13 %	-1 %	-6 %
EspejoRp	-7 %	6 %	1 %	11 %	3 %	6 %
LargoFlatten	99 %	-274 %	-33 %	100 %	100 %	100 %
LargoReverse	89 %	-259 %	-45 %	99 %	100 %	100 %
MapFlatten	-15 %	-93 %	-25 %	2 %	-94 %	-11 %
MapReverse	-7 %	-63 %	-54 %	-20 %	-150 %	-129 %
ReverseFlatten	-7 %	1 %	0 %	-4 %	2 %	1 %
ReverseReverse	4 %	-3 %	-2 %	0 %	-2 %	-2 %
SumaFlatten	99 %	-39 %	24 %	100 %	100 %	100 %
SumaReverse	88 %	-65 %	6 %	100 %	100 %	100 %

Cuadro B.12: %Mejora Tiempo - Ej. Optimizadas - Consumidoras No Acumulativas

B.2. Medidas del espacio de memoria requerido

B.2.1. Valores Absolutos

Los valores que se presentan a continuación están dados en Mega bytes.

Composición	Ejecución No Optimizados			Ejecución Optimizados		
Nombre	Original	Manual	Final	Original	SCA	Final
AlturaAcAsc	84	93	65	17	36	33
AlturaAcRp	2	2	2	2	2	1
LargoAcFlatten	723	725	682	2	670	616
LargoAcReverse	922	1125	983	194	954	787
RempAcAsc	778	873	487	680	873	487
RempAcRp	2	2	2	2	2	2
ReverseAcFlatten	921	967	920	915	968	807
ReverseAcReverse	798	1101	798	759	1213	759
SumaAcFlatten	723	725	682	2	678	706
SumaAcReverse	922	1125	983	194	1047	964
UnpAcAsc	420	488	316	277	522	278

Cuadro B.13: Espacio Absolutos - Consumidoras Acumulativas

Composición	Ejecución No Optimizados			Ejecución Optimizados		
Nombre	Original	Manual	Final	Original	SCA	Final
CantNodosAsc	366	322	258	65	162	1
CantNodosRp	2	2	2	2	2	1
EspejoAsc	361	358	362	402	401	407
EspejoRp	2	2	2	2	2	2
LargoFlatten	560	513	513	130	513	1
LargoReverse	358	392	369	129	356	1
MapFlatten	2	2	2	2	2	2
MapReverse	483	859	982	539	728	1001
ReverseFlatten	10	10	10	10	10	10
ReverseReverse	7	7	7	7	7	7
SumaFlatten	560	513	513	257	513	1
SumaReverse	358	392	369	193	356	1

Cuadro B.14: Espacio Absolutos - Consumidoras No Acumulativas

B.2.2. Porcentaje de Mejora

Composición	Ejecución No Optimizados		Ejecución Optimizados	
Nombre	Manual/Original	Final/Original	SCA/Original	Final/Original
AlturaAcAsc	-11 %	23 %	-112 %	-94 %
AlturaAcRp	0 %	0 %	0 %	50 %
LargoAcFlatten	0 %	6 %	-33400 %	-30700 %
LargoAcReverse	-22 %	-7 %	-392 %	-306 %
RempAcAsc	-12 %	37 %	-28 %	28 %
RempAcRp	0 %	0 %	0 %	0 %
ReverseAcFlatten	-5 %	0 %	-6 %	12 %
ReverseAcReverse	-38 %	0 %	-60 %	0 %
SumaAcFlatten	0 %	6 %	-33800 %	-35200 %
SumaAcReverse	-22 %	-7 %	-440 %	-397 %
UnpAcAsc	-16 %	25 %	-88 %	0 %

Cuadro B.15: %Mejora Espacio - Consumidoras Acumulativas

Composición	Ejecución No Optimizados		Ejecución Optimizados	
Nombre	Manual/Original	Final/Original	SCA/Original	Final/Original
CantNodosAsc	12 %	30 %	-149 %	98 %
CantNodosRp	0 %	0 %	0 %	50 %
EspejoAsc	1 %	0 %	0 %	-1 %
EspejoRp	0 %	0 %	0 %	0 %
LargoFlatten	8 %	8 %	-295 %	99 %
LargoReverse	-9 %	-3 %	-176 %	99 %
MapFlatten	0 %	0 %	0 %	0 %
MapReverse	-78 %	-103 %	-35 %	-86 %
ReverseFlatten	0 %	0 %	0 %	0 %
ReverseReverse	0 %	0 %	0 %	0 %
SumaFlatten	8 %	8 %	-100 %	100 %
SumaReverse	-9 %	-3 %	-84 %	99 %

Cuadro B.16: %Mejora Espacio - Consumidoras No Acumulativas