Generación de código para lenguajes funcionales perezosos

Alberto de la Encina Vara

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid
13 de septiembre de 2001

Trabajo de Tercer Ciclo
Director del trabajo: Ricardo Peña Marí
Nº de créditos solicitados: 12

Índice General

Agradecimientos

En primer lugar quiero agradecer todo el esfuerzo dedicado por mi director Ricardo Peña, sin el cual hubiera sido imposible llevar a cabo el trabajo de forma correcta. Sus constantes orientaciones han hecho que este trabajo sea el que se presenta aquí.

Por otro lado quisiera agradecer a Clara Segura y a Natalia López, no sólo por estar siempre dispuestas a contestar mis consultas sobre LaTeX, sino también por darme ánimos y estar dispuestas a escucharme.

A Fernando Rubio, por contestar a mis innumerables dudas, sobre el Glasgow Haskell Compiler, Linux, y LaTeX.

A Yolanda Ortega y a Phil Trinder por su interés acerca del trabajo. Su revisión y sus consultas han sido útiles para reformarlo y mejorarlo.

A Olga Marroquín y a Mercedes Hidalgo por aguantar mis charlas y estar dispuestas a ayudarme siempre que han podido.

A todos aquellos que no están mencionados anteriormente y que también mostraron interés por mi trabajo de investigación, ya que gracias a ellos consideré que mi trabajo no era en balde y continué con él.

A todas las personas que han estado apoyándome y se han mantenido cerca, estando siempre dispuestas a escucharme y aguantar a veces mis charlas sobre temas de los que nada comprendían. Este es tanto mi primer trabajo de investigación, como mi primer trabajo en LaTeX, y sin su ayuda no creo que hubiera sido capaz de llevarlo a cabo correctamente.

Por último, aunque muchos de ellos ya han sido mencionados anteriormente, quisiera agradecer de forma especial la revisión de este trabajo a: Clara Segura, Fernando Rubio, Jose Luis Sierra, Laura de la Encina, Ricardo Peña y Yolanda Ortega.

Capítulo 1

Introducción

La programación funcional posee una serie de ventajas [Hug90, Bir98] con respecto a la programación imperativa entre las que caben destacar la ausencia de efectos laterales, el orden superior, el polimorfismo, su semántica simple, la facilidad para realizar transformaciones de forma segura y la evaluación perezosa. Sin embargo, la mayor distancia entre ésta y la arquitectura de las máquinas reales hace que la implementación de dichos lenguajes sea más ineficiente.

Para mejorar la eficiencia y salvar esa distancia, se han desarrollado máquinas abstractas fácilmente implementables sobre la arquitectura tradicional Von Neumann. Las primeras máquinas abstractas para lenguajes funcionales fueron impacientes y utilizaban el paso de parámetros por valor. Hasta finales de los años 80 no se desarrollaron máquinas abstractas para lenguajes funcionales perezosos, implementándose con ello el paso por necesidad, es decir, el orden normal de evaluación junto con la compartición de cómputos ya evaluados. Entre las principales máquinas de esa época se encuentran la máquina-G [Aug84, Joh84, Pey87], la TIM[1] [FW87, PL92], la *STG*[2] [PS89, Pey92] y la máquina de Krivine [Cur91].

Al principio no se ponía énfasis en la corrección, sino solamente en la eficiencia. Es más, no existía una semántica operacional aceptada para la evaluación perezosa. Launchbury en 1993 [Lau93] definió una semántica operacional para la evaluación perezosa y demostró su corrección y completitud respecto a una semántica denotacional del lenguaje. A partir de este momento se convirtió en un estándar *de facto* para dicho modo de evaluación. Esta ha sido aplicada a diferentes áreas tales como el análisis de uso [WP99, GS01] y la evaluación paralela de los lenguajes funcionales [BFKT00].

Sestoft en 1997 [Ses97] definió una serie de máquinas abstractas perezosas y demostró la corrección y completitud de éstas con respecto a la semántica operacional que Launchbury había proporcionado.

Una de las máquinas abstractas más exitosas para la ejecución de lenguajes funcionales perezosos es la máquina *STG* [Pey92], que es el corazón del *Glasgow Haskell Compiler* (*GHC*) [PHH$^+$93, Pey96]. El compilador recibe un programa escrito en Haskell [PH99] y, después de diversos pasos y transformaciones intermedias [San95], produce un programa en un lenguaje funcional muy simple llamado lenguaje STG (STGL). Este lenguaje es la entrada para la máquina

[1] Three Instructions Machine.
[2] Spineless Tagless G-Machine.

STG. El compilador *GHC* genera código imperativo que simula las transiciones de la máquina.

Este trabajo pretende facilitar la comprensión de la compilación eficiente de un lenguaje funcional moderno así como demostrar la corrección de la máquina *STG* con respecto a la semántica operacional de Launchbury. Para ello se va a recorrer el largo camino que parte de una semántica operacional muy abstracta como la de Launchbury y termina en la generación de código imperativo realizada por un compilador optimizado como *GHC*. En cada etapa nuestro interés será doble: por un lado ser didácticos y por otro ser rigurosos. Se definirán propiedades que ayuden a la comprensión de cada etapa y se demostrarán posteriormente. El recorrido se realizará de mayor a menor nivel de abstracción. Se pretende con ello que el lector pueda utilizar este trabajo como guía para introducirse en la generación de código de los lenguajes funcionales perezosos. Para ello estudiaremos en detalle varias máquinas abstractas, fijándonos en ciertas optimizaciones que se realizan sobre ellas, y nos centraremos en la máquina *STG*.

La máquina *STG* ha demostrado ser muy eficiente comparada con algunas otras máquinas para los lenguajes funcionales, tales como la máquina-G o la TIM. Además, ha sido tomada como base para extensiones tales como paralelismo y concurrencia [Hor96, PS98, PPS96, Cha95, BKL$^+$98]. Sin embargo, no ha existido una prueba formal de su corrección hasta hace muy poco tiempo, a diferencia de los lenguajes funcionales impacientes como ML [HMM86] cuyas máquinas poseen una prueba formal de corrección desde mediados de los 80. La única prueba de la corrección de la máquina *STG* ha sido dada por J. Mountjoy en [Mou98]. Allí el autor parte de la semántica natural de Launchbury para la evaluación perezosa [Lau93] y la transforma sucesivamente hasta obtener una semántica más elaborada. Desde esta semántica Mountjoy deriva una máquina muy similar a la máquina *STG* y demuestra su corrección con respecto a la semántica. La máquina tiene una pila (en lugar de las tres que posee la máquina *STG*) y no trata las construcciones como formas normales. Además demuestra que la semántica más elaborada es equivalente a la semántica de Launchbury.

Es debatible si la aproximación correcta para la demostración de la corrección de una máquina abstracta es refinar la especificación para que ésta esté más y más cerca del comportamiento operacional deseado.

En este trabajo de investigación se ha seguido una estrategia diferente. Partimos de la semántica de Launchbury como *la* especificación y de las máquinas abstractas propuestas por Sestoft en [Ses97] las cuales son correctas y completas con respecto a la semántica natural de Launchbury. La idea es continuar con el refinamiento de una de estas máquinas, la *Mark-2*, hasta llegar a la máquina *STG*. Para tener un lenguaje común con el cual poder compararlas, definimos un lenguaje muy similar al STGL para que pueda ser considerado un subconjunto del lenguaje utilizado por las máquinas de Sestoft. Entonces, definimos y probamos la existencia de una *bisimulación* entre la máquina *Mark-2* de Sestoft y la *STG*.

La máquina descrita en [Pey92] tenía tres pilas separadas y la implementada en las primeras versiones del compilador *GHC* tenía dos pilas. La última versión del compilador *GHC* ha modificado la implementación [PMR99] y ahora posee una única pila. La bisimulación se mantiene para una máquina *STG* con una pila simple. Sin embargo, aún no existe una descripción operacional para esta máquina en la dirección de la que se da en [Pey92]. Nosotros demostramos que la máquina con tres pilas *no* es correcta con respecto a la semántica natural para algunos programas mal tipados. Mientras que la semántica es incapaz de alcanzar una forma normal, la máquina puede terminar exitosamente con una.

Otra de las etapas importantes de todo compilador, y donde radica el éxito de muchos de ellos, es la etapa de generación de código. Dicha etapa nunca había sido formalizada anteriormente en el *GHC*. Los únicos documentos que existen al respecto son [PL92, PHH^{+}93, PMR99] o directamente el código de dicho compilador [PMST01]. En todos estos documentos se mezclan muchos y muy diferentes aspectos a la vez: numeración de variables libres, implementación del entorno, reducción de las tres pilas de la máquina *STG* a una sola, diferentes optimizaciones, etc. Esto ha motivado que desarrollemos esquemas formales de traducción para pasar de las expresiones STGL a un código imperativo muy cercano al que genera el compilador.

Este trabajo de investigación se organiza de la siguiente forma: en el Capítulo 2 veremos un resumen de la semántica natural de Launchbury, las modificaciones propuestas por Sestoft, sus máquinas abstractas y por último un resumen de la máquina abstracta *STG* junto con su lenguaje STGL. A partir de aquí comienzan las aportaciones originales de este trabajo. En el Capítulo 3 realizaremos la demostración formal de la corrección de la semántica de Sestoft con respecto a la semántica de Launchbury, siguiendo las indicaciones de Sestoft. Modificaremos la semántica de Sestoft debido a que ésta posee un pequeño error: la propiedad de frescura no se mantiene con la introducción de las expresiones **case**. Además, en este capítulo introduciremos en la semántica algunas consideraciones acerca de la recolección de basura.

Tras el análisis en detalle de la semántica pasaremos al estudio de las máquinas abstractas de Sestoft en el Capítulo 4. En este capítulo propondremos y demostraremos varias propiedades interesantes relativas a las máquinas de Sestoft y formalizaremos la demostración de la corrección entre la máquina *Mark-2* y la máquina *Mark-3*. Para ello se estudiará el paso de una expresión con variables a otra en la que las variables han sido sustituidas por los índices de De Bruijn [Bru72, Bru78].

Posteriormente, en el Capítulo 5 demostraremos la corrección y completitud con respecto a la máquina *Mark-2* de una máquina muy similar a la *STG*, la *STG-1S*, donde la principal diferencia es que posee una única pila. Esta demostración se realizará a través de una bisimulación entre la máquina *STG-1S* y la máquina *Mark-2*. Se demostrará que la máquina *STG* con tres pilas no es correcta con respecto a la semántica, ya que para ciertas expresiones mal tipadas es capaz de llegar a reducir una forma normal, mientras que la semántica no es capaz de alcanzarla. En ese mismo capítulo veremos cómo se podría prescindir de esta exigencia para la expresión de partida de la máquina *STG*, manteniendo las tres pilas pero con una leve modificación en la máquina.

Las aportaciones de los capítulos 3, 4, y 5 han dado lugar a un artículo que se presentará en el *13th International workshop on the Implementation of Functional Languages* (IFL'01) [EP01a] y a otro artículo aceptado en las *Primeras Jornadas sobre Programación y Lenguajes* (PROLE'01) [EP01b].

Una vez demostrada la corrección y completitud de la máquina *STG-1S*, pasaremos a centrarnos en la siguiente etapa del compilador. En el Capítulo 6 formalizaremos una generación de código muy similar a la que realiza *GHC*. Para su mejor comprensión, esta formalización se realizará en sucesivas etapas, partiendo de una generación ingenua hasta llegar a una generación de código optimizada, cercana a la generada por dicho compilador.

El trabajo finaliza con el Capítulo 7 en el que se exponen los logros alcanzados y posibles líneas futuras de trabajo.

Capítulo 2

Semántica natural y máquinas abstractas

2.1 Introducción

En [Lau93] John Launchbury parte de un pequeño lenguaje funcional perezoso con **let** recursivos, pero sin constructores ni **case**. Como es habitual en el λ-cálculo, las funciones tienen un solo argumento. Proporciona una semántica operacional a ese lenguaje y demuestra que esa semántica es correcta y completa con respecto a la semántica denotacional del lenguaje funcional.

Su λ-cálculo pasa previamente por un proceso de normalización. Este proceso de normalización realiza un cambio en el nombre de todas las variables ligadas del programa, de tal manera que todas ellas pasen a tener nombres distintos. Además, para mantener la compartición, este proceso de normalización asegura que todas las aplicaciones se hacen sobre variables . Esto lo realiza añadiendo tantos **let**s como sean necesarios.

Posteriormente añade los constructores, las expresiones **case** y los tipos primitivos junto con las operaciones básicas sobre tipos primitivos, y añade las reglas semánticas correspondientes. Pero no demuestra la corrección y completitud de esta semántica extendida con respecto a la denotacional.

En [Ses97] Peter Sestoft parte del mismo lenguaje funcional perezoso que Launchbury y de su semántica operacional. La semántica de Sestoft es una modificación de la de Launchbury. La principal modificación es el lugar donde se generan variables frescas, que pasa a ser la regla de evaluación del **letrec**. En la semántica de Launchbury se renombra cuando se evalua una variable. El proceso de renombramiento conlleva el cambio de nombre por variables frescas de todas las variables ligadas de la expresión asociada en el montón[1] a esa variable. Parece más cercano al modo en que se implementan los lenguajes funcionales cambiar el nombre de las variables cuando se crean clausuras en el montón (en tiempo de ejecución esto corresponde a la reserva de direcciones de memoria) y, por tanto, cuando pasan de variables ligadas a libres. Con ello se evita el renombramiento de todas las variables ligadas de una expresión, y sólo se renombra cuando la variable pasa de estar ligada a estar libre (es decir, cuando pasa de ser una variable de programa a ser un puntero). Con lo cual se diferencia claramente entre variables

[1] En inglés *heap*.

libres (punteros) y ligadas del programa.

Otra de las ideas principales de Sestoft es que se pueda comprobar de forma local la propiedad de frescura de las variables. En la semántica de Launchbury es necesario recorrer todo el árbol de derivación para averiguar si una variable ya existe o no. Para evitar esto Sestoft añade un nuevo conjunto, denominado A, en el cual se dejan los punteros en evaluación. Con este conjunto Sestoft asegura que se puede comprobar de forma local si una variable es fresca o no. Veremos en el capítulo siguiente que esta propiedad se mantiene para un λ-cálculo básico sin constructores ni **case**, pero que es necesaria una pequeña modificación para que se mantenga cuando se añaden los constructores y los **case**.

A partir de esta semántica modificada Sestoft va derivando distintas máquinas abstractas. La primera máquina abstracta es muy simple, y está basada en la de Krivine [Cur91], que es una máquina para la llamada por nombre, modificada para la evaluación perezosa y extendida a los constructores y expresiones **case**. Demuestra la corrección y completitud de esta máquina con respecto a la semántica operacional modificada.

En la segunda máquina introduce un entorno para evitar las sustituciones explícitas sobre la expresión de control en ejecución e indica cómo se puede demostrar que realiza los mismos cálculos que la primera. Posteriormente, en la tercera máquina cambia las variables por los índices de De Bruijn [Bru72, Bru78] y hace una poda[2] sobre los entornos, para sólo referenciar las variables realmente usadas en las expresiones correspondientes, en lugar de todas las variables en ámbito[3]. Ello facilita la detección de clausuras muertas y, por lo tanto, mejora el uso de la memoria. Finalmente extiende el lenguaje con constructores, expresiones **case** sobre constructores algebraicos, valores básicos y operaciones básicas sobre estos valores.

Posteriormente, veremos la máquina *STG* [Pey92]. Esta máquina es una máquina abstracta basada en un lambda cálculo con una sintaxis bastante alejada de la sintaxis tradicional del lambda cálculo. Además incluye a diferencia de las máquinas de Sestoft una expresión **let** no recursiva, enteros desencapsulados, operaciones sobre valores primitivos y expresiones **case** sobre valores primitivos.

2.2 Semántica natural

Launchbury define en [Lau93] una semántica operacional natural de paso grande para la evaluación perezosa (reducción en orden normal hasta forma normal débil de cabeza y compartición de los argumentos). La única maquinaria necesaria es un montón explícito donde se mantienen las ligaduras. Un montón consiste en una función finita entre variables y expresiones, es decir, no se permiten ligaduras duplicadas para la misma variable. Si en un montón se añadiera una nueva ligadura para una variable ligada previamente, se reemplazaría la expresión asociada a esa variable por la nueva expresión (sin embargo, esto no sucede ni en la semántica de Launchbury ni en la de Sestoft).

[2] En inglés *trimming*.

[3] Entendemos por "ámbito" todas las variables visibles en la expresión. El ámbito de una variable ligada por una λ es el cuerpo de la lambda abstracción; el de una ligada en un **letrec** son todas las partes derechas $\overline{e_i}$ y la expresión principal e; y el de una ligada en una alternativa de un **case** es la parte derecha e_i de la alternativa.

$$
\begin{array}{lcll}
e & \to & x & \text{-- variable} \\
 & | & \lambda x.e & \text{-- lambda abstracción} \\
 & | & e\ x & \text{-- aplicación} \\
 & | & \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e & \text{-- let recursivo} \\
 & | & C\ \overline{x_i} & \text{-- aplicación saturada de constructor} \\
 & | & \textbf{case}\ e\ \textbf{of}\ alts & \text{-- expresión case} \\
alts & \to & \overline{C_i\ \overline{y_{ij}} \to e_i} & \text{-- alternativas}
\end{array}
$$

Figura 2.1: λ-cálculo normalizado de Launchbury

2.2.1 Sintaxis

El lenguaje utilizado tanto por Launchbury y como por Sestoft es un λ-cálculo normalizado, extendido con **let** recursivos, aplicación saturada[4] de constructores, y expresiones **case**. Para asegurar la compartición, se fuerza a que el argumento de toda aplicación sea una variable. La gramática para este lenguaje se puede ver en la Figura 2.1, donde la notación $\overline{A_i}$ denota un vector $A_1, \ldots, A_n$ de valores subindexados, y donde x e y son variables.

Es importante comentar que este lenguaje no pierde ninguna potencia expresiva con respecto al λ-cálculo tradicional, ya que cualquier expresión en este último puede ser transformada a una expresión en este λ-cálculo normalizado. El proceso de normalización implica las siguientes transformaciones:

1. Todas las variables ligadas del programa (ya sean en las expresiones lambda, **let** o **case**) de la expresión inicial deben ser distintas.

2. Toda aplicación debe realizarse a una variable. Para ello lo único necesario es añadir ligaduras **let** para aquellos argumentos que no sean variables.

El punto 1 elimina la captura de variables en la semántica del lenguaje normalizado. El punto 2 surge debido a la compartición. Manteniendo los argumentos como variables, cada vez que se evalúe un argumento y se actualice su valor con su forma normal, su evaluación se mantiene para los cómputos restantes: la siguiente vez que accedamos a ese argumento estará evaluado y utilizaremos directamente su forma normal.

Las formas normales (débiles de cabeza) de este λ-cálculo normalizado, son las lambda abstracciones y las construcciones[5].

Las expresiones **letrec** son potencialmente recursivas, y todas las variables ligadas dentro de un **letrec** han de ser distintas.

La sintaxis de la Figura 2.1 ha sufrido una pequeña modificación con respecto a la original [Lau93]. Allí se utiliza la palabra **let** para las expresiones *let* recursivas. Por claridad hemos preferido utilizar **letrec** para los **let** potencialmente recursivos.

[4]Aplicada a todos sus argumentos.
[5]Aplicaciones de constructores.

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\Gamma : C\,\overline{x_i} \Downarrow \Gamma : C\,\overline{x_i} \qquad \textit{Cons}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow \Theta : w}{\Gamma : e\ x \Downarrow \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow \Delta : w}{\Gamma \cup [x \mapsto e] : x \Downarrow \Delta \cup [x \mapsto w] : \hat{w}} \qquad \textit{Var}$$

$$\frac{\Gamma \cup [\overline{x_i \mapsto e_i}] : e \Downarrow \Delta : w}{\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow \Delta : w} \qquad \textit{Letrec}$$

$$\frac{\Gamma : e \Downarrow \Delta : C_k\ \overline{x_j} \quad \Delta : e_k[\overline{x_j/y_{kj}}] \Downarrow \Theta : w}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i} \Downarrow \Theta : w} \qquad \textit{Case}$$

Figura 2.2: Semántica natural de Launchbury

2.2.2 Semántica de Launchbury

En la Figura 2.2 pueden verse las reglas semánticas. A lo largo de todo el trabajo el símbolo w será utilizado para denotar una forma normal. Un juicio de la forma $\Gamma : e \Downarrow \Theta : w$ significa que la expresión e, con sus variables libres ligadas en el montón Γ, se reduce a una forma normal w y produce el montón final Θ. La notación $\hat{w}$ significa que en la expresión w hemos reemplazado todas las variables ligadas por nombres frescos. Decimos que $\Gamma : e \Downarrow \Theta : w$ es una derivación exitosa si se puede probar usando las reglas. Puede fallar la prueba de una derivación por varios motivos, siendo uno de ellos que el programa no termine y se quede en un bucle infinito sin llegar a derivar una forma normal. Otra posibilidad es que no haya regla para aplicar y que no se pueda alcanzar una forma normal. Entre estas últimas situaciones cabe destacar el caso en que no hay regla para aplicar debido a que se referencia una clausura que actualmente está bajo evaluación. Diremos que la evaluación ha entrado en un *agujero negro*. Esto podría suceder en la regla *Var* cuando aparece una referencia a la variable x mientras estamos reduciendo la expresión e y antes de alcanzar una forma normal. Como el montón Γ no contiene ninguna ligadura para la variable x, no se puede aplicar ninguna regla y, por tanto, la derivación no puede completarse.

El principal teorema en [Lau93] es que la semántica operacional es correcta y completa con respecto a la semántica denotacional para un lenguaje no estricto, es decir, si e es una expresión cerrada, entonces $[\![e]\!]\ \rho_0 = v \neq \perp$ si y sólo si existe Θ y w tales que $\{\,\} : e \Downarrow \Theta : w$ y $[\![w]\!]\ \rho_\Theta = v$, siendo ρ_0 el entorno vacío y ρ_Θ el entorno que se obtiene a partir del montón Θ donde se halla la definición de las variables libres de la expresión w.

2.2.3 Semántica de Sestoft

Sestoft introduce en [Ses97] dos cambios importantes en la semántica operacional de la Figura 2.2:

1. Cambia el renombramiento de variables de la regla *Var* a la regla *Letrec*.

2. Extiende los juicios con un nuevo conjunto A de *variables bajo evaluación* para comprobar la propiedad de frescura de forma local en la regla *Letrec*.

La primera modificación se debe al intento de acercar un poco más la semántica a la implementación de las máquinas abstractas. En una implementación usual, las variables frescas (es decir, los punteros) se crean cuando introducimos clausuras en el montón en la regla *Letrec*. Esta aproximación es también más económica que la consistente en renombrar todas las variables ligadas de una lambda abstracción, ya que ahora sólo se necesita renombrar las variables que han pasado de ligadas a libres. Además, con esta modificación se evita la necesidad de que todas las variables ligadas de la expresión inicial tengan distintos nombres. La segunda modificación hace más precisa la definición de *frescura*: una variable es fresca en un juicio $\Gamma\ : e \Downarrow_A \Theta : w$ si no pertenece ni a *dom* Γ[6] ni a A, y no está ligada ni en *range* Γ[7] ni en e. Las reglas modificadas pueden verse en la Figura 2.3. En la regla *Letrec*, la notación $\hat{e}$ indica que realizamos el renombramiento siguiente $e[\overline{p_i/x_i}]$ donde $\overline{p_i}$ son variables distintas y frescas en el juicio $\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow_A \Delta : w$.

El conjunto A contiene las direcciones de las clausuras actualmente en evaluación. Es decir, cuando vamos a evaluar una variable, metemos en el conjunto A la variable y la quitamos del montón (véase la regla *Var*), de tal manera que en el montón no tenemos ninguna variable en evaluación. Nos interesa que en el montón no se encuentren las variables en evaluación, ya que de esta manera podemos detectar cuándo una expresión en evaluación tiene un puntero a otra clausura en evaluación. Así, si se da el caso de que una clausura en evaluación demande otra clausura bajo evaluación la derivación se detiene en lugar de ciclar. Esto mismo sucede en la semántica de Launchbury, ya que cuando se empieza a evaluar una clausura ésta se elimina del montón(ver regla *Var*, Figura 2.2). Sin embargo, en dicha semántica no se almacenan en ningún conjunto y, por lo tanto, para saber en un momento concreto del cómputo las clausuras en evaluación, es necesario recorrer todo el árbol de derivación.

La diferencia entre las nuevas reglas (ver Figura 2.3) y las dadas en la Figura 2.2 es el lugar donde sucede el renombramiento. Por tanto, lo único que es necesario probar para demostrar la equivalencia entre ambas semánticas es que no existe ninguna captura de variables en la sustitución $e'[x/y]$ en la regla *App*, y que no hay ningún intento de ligar en el montón ninguna variable que estuviera ya ligada en Δ, en la regla *Var*.Esto lo demuestra Sestoft con la siguiente proposición:

Proposición 1 *(Sestoft)*
Sea e una expresión cerrada y $\{\} : e \Downarrow_{\{\}} \Theta : w$ *una derivación. Entonces, en ninguna aplicación de la regla App existe captura de variables en la expresión* $e'[p/x]$*, y* p *no está previamente ligada en* Δ *en ninguna aplicación de la regla Var.*

Además, Sestoft deja indicada la demostración de que en cada árbol de derivación existe una distinción clara entre variables libres y ligadas de las expresiones que aparecen tanto en el juicio como en el montón. Esta distinción se mantiene a lo largo de toda la ejecución. Las variables

[6]Por *dom* Γ entendemos todas las variables que están ligadas a expresiones en Γ.

[7]Por *range* Γ entendemos todas las expresiones ligadas en Γ.

$$\Gamma : \lambda x.e \Downarrow_A \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\Gamma : C\,\overline{p_i} \Downarrow_A \Gamma : C\,\overline{p_i} \qquad \textit{Cons}$$

$$\frac{\Gamma : e \Downarrow_A \Delta : \lambda x.e' \quad \Delta : e'[p/x] \Downarrow_A \Theta : w}{\Gamma : e\ p \Downarrow_A \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow_{A\cup\{p\}} \Delta : w}{\Gamma \cup [p \mapsto e] : p \Downarrow_A \Delta \cup [p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \Downarrow_A \Delta : w}{\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow_A \Delta : w}\ \text{donde } \overline{p_i} \text{ frescas} \qquad \textit{Letrec}$$

$$\frac{\Gamma : e\ \Downarrow_A \Delta : C_k\ \overline{p_j} \quad \Delta : e_k[\overline{p_j/y_{kj}}] \Downarrow_A \Theta : w}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i} \Downarrow_A \Theta : w} \qquad \textit{Case}$$

Figura 2.3: Semántica natural de Sestoft

libres del programa son siempre los punteros (en la Figura 2.3 y a partir de ahora, serán denotadas por p) a direcciones en el montón, o están bajo evaluación y, por lo tanto, pertenecen al conjunto A. Las variables ligadas son variables que pertenecen a la expresión original (en la Figura 2.3 y en lo que sigue serán denotadas por x o y). Este resultado se formaliza en el *Lema 1* (ver pág. 20). En el Capítulo 3 se verán con más detalle estos resultados.

Desafortunadamente, en la Sección 3.1.2 veremos que esta distinción no se mantiene cuando añadimos las expresiones **case**. Este teorema fue propuesto antes de introducir dichas expresiones, y no fue modificado ni ajustado de nuevo para elllas. Con la regla *Case* la propiedad de frescura no se mantiene de forma local, como veremos en el Ejemplo 5(ver pág. 24).

Sin tener en cuenta estos pequeños detalles, Sestoft demuestra que su semántica y la de Launchbury llegan a las mismas deducciones y, por tanto, son equivalentes.

2.3 Máquinas de Sestoft

Tras las modificaciones a las reglas semánticas y la demostración de corrección y completitud con respecto a la semántica de Launchbury, Sestoft introduce en diferentes etapas varias máquinas abstractas *Mark-1*, *Mark-2* y *Mark-3*. Estas van de más cercanas a la semántica operacional hacia más cercanas a la implementación real.

La primera máquina es la *Mark-1* (véase Figura 2.4), que está muy cercana a la semántica. Cada dos líneas separadas por $\Rightarrow$ representan una posible transición de la máquina. El único añadido necesario para esta máquina abstracta es una pila S en la que se almacenan tanto los argumentos por aplicar, como las marcas de actualización, como las alternativas, reglas *app1*, *var1* y *case1* respectivamente. Es decir, la pila almacena el contexto de tal forma que cuando terminamos la evaluación de un juicio en las reglas *App* y *Case* tenemos que acceder a la cima de la pila donde se almacenó el contexto y proseguir la evaluación del segundo juicio con lo que nos indique la cima de la pila. En ella se realizan sustituciones explícitas en la regla *app2* y en la regla *case2*: cuando aplicamos una lambda abstracción a su argumento hacemos ajuste de patrones, realizamos la sustitución $e[p/y]$ o $e_k[\overline{p_i/y_{ki}}]$.

Las reglas semánticas *Var*, *App* y *Case* han sido divididas en las correspondientes reglas *var1*, *var2* y *var3*, *app1* y *app2* y *case1* y *case2*. La división de la regla *Var* se debe a que la actualización, reglas *var2* y *var3*, ha de realizarse después de la evaluación de la expresión, regla *var1*. Sin embargo la división de las reglas *App* y *Case* se debe al cómputo correspondiente a la ejecución de cada juicio en la parte superior de las reglas semánticas correspondientes.

En ciertas reglas es la pila la que toma el control y según lo que se halle en la cima se realiza un cómputo distinto. Así sucede cuando llegamos a una forma normal. Si ésta es una lambda abstracción y en la pila se encuentra una marca de actualización (regla *var2*), se almacena dicha lambda abstracción en el montón en la dirección correspondiente, y si en la pila se encuentra un argumento por aplicar (regla *app2*), se aplica la lambda abstracción a dicho argumento. Si la expresión de control es una construcción y en la pila se encuentra una marca de actualización (regla *var3*), se almacena dicho constructor en el montón en la dirección correspondiente, y si en la pila se encuentran unas alternativas (regla *case2*), se realiza el encaje de patrones y se evalúa la alternativa correspondiente a dicho constructor.

Hemos utilizado la notación $\Gamma \cup [p \mapsto e]$ para indicar que el montón es la unión de Γ con la ligadura $p \mapsto e$. En caso de que existiera previamente en el montón Γ una ligadura del tipo $p \mapsto e'$, ésta se sustituiría por la nueva, y de hecho cuando volviéramos a utilizar Γ habría desaparecido la ligadura previa para p. Sin embargo, esto no va a suceder ya que Sestoft demuestra en su semántica que nunca se liga en el montón una variable que estaba previamente ligada en él.

El comportamiento de esta máquina es exactamente el mismo que el de la semántica, en el sentido de que para toda expresión cerrada e, $\{\,\}\ e \Downarrow \Delta\ w$ es derivable si y sólo si $(\{\,\}, e, [\,]) \Rightarrow^* (\Gamma, w, [\,])$ donde $\Rightarrow^*$ significa una evaluación de cero o más transiciones $\Rightarrow$ de la máquina *Mark-1*. La demostración hace uso de la noción de *evolución equilibrada* de la máquina.

La máquina *Mark-2*, que se muestra en la Figura 2.5, se aleja algo más de la semántica. Elimina las sustituciones explícitas de variables, ya que éstas no son fáciles de realizar en tiempo de ejecución. Para ello necesita un entorno E que es una función finita entre variables del programa y punteros. Ahora cada expresión, tanto en el montón como la expresión de control, lleva asociado un entorno que liga sus variables libres con las direcciones en el montón donde se encuentran almacenadas las clausuras correspondientes. En lugar de realizar la sustitución en las reglas *app2* y *case2*, lo que se hace es añadir en el entorno la ligadura $y \mapsto p$ o las ligaduras $\overline{y_{ki} \mapsto p_i}$, de tal manera que ahora cuando nos encontramos con una variable, regla *var1*, consultamos en el entorno la dirección en la que se encuentra en el montón y accedemos a ella. Cuando se realiza una actualización del montón ahora es necesario almacenar la expresión junto con el entorno asociado a la expresión, y cuando se accede al montón y se extrae una clausura es necesario también sacar con ella su entorno asociado. Demostraremos en la Sección 4.2.1 varias propiedades sobre estos entornos.

Sestoft dejó indicada la demostración de que el comportamiento de esta máquina y el de la máquina *Mark-1* es el mismo y, por tanto, también es correcta y completa con respecto a la semántica.

Hemos utilizado la misma notación para el entorno que la que usamos para el montón, con

	Montón	Control	Pila	Regla
	Γ	$(e\ p)$	S	app1
$\Longrightarrow$	Γ	e	$p : S$	
	Γ	$\lambda y.e$	$p : S$	app2
$\Longrightarrow$	Γ	$e[p/y]$	S	
	$\Gamma \cup [p \mapsto e']$	p	S	var1
$\Longrightarrow$	Γ	e'	$\#p : S$	
	Γ	$\lambda y.e$	$\#p : S$	var2
$\Longrightarrow$	$\Gamma \cup [p \mapsto \lambda y.e]$	$\lambda y.e$	S	
	Γ	$\textbf{letrec}\ \{\overline{x_i = e_i}\}\ \textbf{in}\ e$	S	letrec (∗)
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}]$	$\hat{e}$	S	
	Γ	$\textbf{case}\ e\ \textbf{of}\ alts$	S	case1
$\Longrightarrow$	Γ	e	$alts : S$	
	Γ	$C_k\ \overline{p_i}$	$alts : S$	case2 (∗∗)
$\Longrightarrow$	Γ	$e_k[\overline{p_i/y_{ki}}]$	S	
	Γ	$C_k\ \overline{p_i}$	$\#p : S$	var3
$\Longrightarrow$	$\Gamma \cup [p \mapsto C_k\ \overline{p_i}]$	$C_k\ \overline{p_i}$	S	

(∗) La notación $\hat{e}$ indica que realizamos el renombramiento siguiente $e[\overline{p_i/x_i}]$ donde las variables $\overline{p_i}$ son distintas y frescas con respecto a Γ, $\textbf{letrec}\ \{\overline{x_i = e_i}\}\ \textbf{in}\ e$, y S.

(∗∗) La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*

Figura 2.4: Máquina abstracta *Mark-1*

el mismo significado. Así, $E \cup [x \mapsto p]$ significa que el entorno $E \cup [x \mapsto p]$ es la unión de E con la ligadura $x \mapsto p$. Si ésta existiera se sobreescribe. Posteriormente, en la Sección 4.2.1, demostraremos que si partimos de una expresión cerrada con todas las variables ligadas con nombres distintos, en ningún entorno nunca se liga una variable que ya estuviera ligada previamente.

Tras esto Sestoft introduce la máquina *Mark-3*, ver Figura 2.6, donde las variables han sido sustituidas por los denominados índices de De Bruijn correspondientes. En la Sección 4.3 se formalizará el paso de variables a índices de De Bruijn y se demostrará la equivalencia entre esta máquina y la máquina *Mark-2*.

La principal diferencia ahora es la representación del entorno E, que ha pasado de ser una función a representarse con una lista de variables. Ahora el acceso al puntero relativo a una variable se realiza con el índice de De Bruijn. El índice de De Bruijn es un número entero positivo que nos indica el número de variables ligadas existentes entre la ligadura de dicha variable, es decir su aparición de definición, y las apariciones de uso de la variable. Para realizar este cálculo se tienen en cuenta tanto las variables de las lambda abstracciones como las variables que aparecen

	Montón	Control	Entorno	Pila	Regla	
	Γ	$(e\ x)$	$E \cup [x \mapsto p]$	S	app1	
$\Longrightarrow$	Γ	e	$E \cup [x \mapsto p]$	$p : S$		
	Γ	$\lambda y.e$	E	$p : S$	app2	
$\Longrightarrow$	Γ	e	$E \cup [y \mapsto p]$	S		
	$\Gamma \cup [p \mapsto (e', E')]$	x	$E \cup [x \mapsto p]$	S	var1	
$\Longrightarrow$	Γ	e'	E'	$\#p : S$		
	Γ	$\lambda y.e$	E	$\#p : S$	var2	
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda y.e, E)]$	$\lambda y.e$	E	S		
	Γ	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	E	S	letrec	$(*)$
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto (e_i, E')}]$	e	E'	S		
	Γ	**case** e **of** $alts$	E	S	case1	
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$		
	Γ	$C_k\ \overline{x_i}$	$E \cup [\overline{x_i \mapsto p_i}]$	$(alts, E') : S$	case2	$(**)$
$\Longrightarrow$	Γ	e_k	$E' \cup [\overline{y_{ki} \mapsto p_i}]$	S		
	Γ	$C_k\ \overline{x_i}$	E	$\#p : S$	var3	
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C_k\ \overline{x_i}, E)]$	$C_k\ \overline{x_i}$	E	S		

$(*)$ En la regla *let*, las variables $\overline{p_i}$ deben ser distintas y frescas: No pueden ocurrir ni en Γ, ni en **letrec** $\{\overline{x_i = e_i}\}$ **in** e, ni en S. El nuevo entorno $E' = E \cup [\overline{x_i \mapsto p_i}]$.

$(**)$ La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*.

Figura 2.5: Máquina abstracta *Mark-2*

en las expresiones **case** y **letrec**. Cuando vamos a acceder a una clausura del montón lo hacemos con un entero en la expresión de control, denotado por n en la máquina. Lo único que tenemos que hacer es consultar en el entorno E la posición n y ésta nos indica la dirección del montón donde se encuentra ligada dicha variable.

Esto conlleva un ahorro en ejecución tanto en tiempo como en espacio, ya que ahora no es necesario almacenar las variables del programa en los entornos. Ahora sólo es necesario almacenar los punteros manteniendo un orden, que es el orden temporal en el cual se fueron creando dichas variables libres.

La notación $x : xs$ representa la concatenación por la izquierda de un elemento x a una lista xs. Por otro lado, $\overline{p_i}^n$ representa $p_1 : \ldots : p_n$ y $E[u_i]$ es el valor existente en el entorno E en la posición u_i.

Tras la introducción de estas máquinas, Sestoft propone ciertas mejoras y añadidos. Entre ellos se encuentran:

- La poda en los entornos de las clausuras, ya que como se verá en la Sección 4.2.1, éstos mantienen todas las variables en ámbito en lugar de mantener sólo las variables libres de

	Montón	Control	Entorno	Pila	Regla
	Γ	$(e\ n)$	$\overline{p_i}^n : E$	S	app1
$\Longrightarrow$	Γ	e	$\overline{p_i}^n : E$	$p_n : S$	
	Γ	$\lambda.e$	E	$p : S$	app2
$\Longrightarrow$	Γ	e	$p : E$	S	
	$\Gamma \cup [p_n \mapsto (e', E')]$	n	$\overline{p_i}^n : E$	S	var1
$\Longrightarrow$	Γ	e'	E'	$\#p_n : S$	
	Γ	$\lambda.e$	E	$\#p : S$	var2
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda.e, E)]$	$\lambda.e$	E	S	
	Γ	**letrec** $\{\overline{e_i}\}$ **in** e	E	S	letrec (*)
$\Longrightarrow$	$\Gamma \cup \overline{[p_i \mapsto (e_i, E')]}$	e	E'	S	
	Γ	**case** e **of** $alts$	E	S	case1
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$C_k\ \overline{u_i}^n$	E	$(alts, E') : S$	case2 (**)
$\Longrightarrow$	Γ	e_k	$E[u_n] : \ldots : E[u_1] : E'$	S	
	Γ	$C\ \overline{u_i}$	E	$\#p : S$	var3
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{u_i}, E)]$	$C\ \overline{u_i}$	E	S	

(*) En la regla *let*, las variables $\overline{p_i}$ deben ser distintas y frescas: No pueden ocurrir ni en Γ, ni en **letrec** $\{\overline{x_i\ =\ e_i}\}$ **in** e, ni en S. El nuevo entorno $E' = \overline{p_i} : E$.

(**) La expresión e_k corresponde a la alternativa $C_k \rightarrow e_k$ en *alts*.

Figura 2.6: Máquina abstracta *Mark-3*

la expresión.

- Añade, tanto a su semántica como a sus máquinas, enteros primitivos (desencapsulados) junto con operaciones primitivas sobre estos valores.

- Comenta cómo eliminar actualizaciones innecesarias.

2.4 Máquina *STG*

A continuación expondremos tanto el lenguaje admitido por la máquina abstracta *STG* como sus reglas de ejecución, y comentaremos las diferencias entre el lenguaje de Launchbury y el lenguaje STG (STGL).

2.4.1 Sintaxis del lenguaje STG

El lenguaje STG que se muestra en la Figura 2.7 es también un lambda cálculo que ha sufrido un proceso de normalización diferente al de Launchbury. Las principales diferencias son las siguientes:

- No se permiten aplicaciones de expresiones a variables. Todas las aplicaciones son de variable a variables. Por tanto, el proceso de normalización aún necesitará añadir ligaduras para todas las expresiones aplicadas a variables.
- Las aplicaciones son de una variable a varias, y no de una a una como sucede en el lenguaje de Launchbury (ver Figura 2.1).
- Sólo se admiten lambda abstracciones en las ligaduras de los **letrec**. (En adelante utilizaremos el término *lambda forma* para designar indistintamente a una lambda abstracción y a una expresión).
- Las expresiones **case** tienen una alternativa por defecto.
- Existe una expresión **let** no recursiva.
- Las lambda formas van marcadas con su conjunto de variables libres vs y un indicador $\backslash\pi$ que señala si la lambda forma es actualizable (denotado por **u**) o no lo es (denotado por **n**). Este indicador sólo se utiliza para realizar optimizaciones. Las clausuras en forma normal no necesitan ser actualizadas, por tanto, están marcadas siempre con **n**. Sin embargo, existen clausuras no en forma normal que, tras un análisis de uso, se sabe que sólo van a ser utilizadas una vez, por lo tanto, es seguro tenerlas marcadas como no actualizables también. Todas las demás clausuras son actualizables y, por tanto, su marca es **u**.
- En este lenguaje tenemos valores desencapsulados, operaciones primitivas, y expresiones **case** sobre valores desencapsulados.
- En el lenguaje *STGL* un programa es una lista de ligaduras con una variable distinguida *main* donde comienza la evaluación.

La sintaxis de este lenguaje está un poco lejos de la sintaxis tradicional del lambda cálculo. Por ejemplo $\lambda x_1.\ldots\lambda x_n.e$, donde e tiene además de las $\overline{x_i}$ variables libres $v_1,\ldots,v_m$, se expresa ahora $\{\overline{v_i}^m\}\ \backslash\pi\ \{\overline{x_i}^n\} \rightarrow e$. No obstante, sigue siendo un lambda cálculo al que se le han añadido enteros desencapsulados y operaciones primitivas sobre ellos.

2.4.2 Máquina abstracta *STG*

Las principales diferencias de comportamiento entre la máquina *STG* y las máquinas de Sestoft son las siguientes:

- En la *STG* no se realizan aplicaciones parciales.
- Los entornos asociados a cada expresión, tanto los almacenados en el montón como en la pila, sólo contienen las variables libres de las expresiones asociadas.
- En la máquina *STG* nunca se puede tener como expresión de control una lambda abstracción.

$prog \rightarrow$	$\overline{bind_i}$	programa
$e \rightarrow$	$\textbf{let}\ \overline{bind_i}^n\ \textbf{in}\ e$	$n \geq 1$ let no recursivo
$\mid$	$\textbf{letrec}\ \overline{bind_i}^n\ \textbf{in}\ e$	$n \geq 1$ let recursivo
$\mid$	$\textbf{case}\ e\ \textbf{of}\ alts$	expresión case
$\mid$	$x\ \overline{atom_i}^n$	$n \geq 0$ aplicación o variable
$\mid$	$C\ \overline{atom_i}^n$	$n \geq 0$ constructores saturados
$\mid$	$prim\ \overline{atom_i}^n$	$n \geq 0$ operadores primitivos
$\mid$	$literal$	literales
$bind \rightarrow$	$x = lf$	
$alts \rightarrow$	$\overline{aalt_i}^n ; def$	$n \geq 1$ alternativas algebraicas
	$\overline{palt_i}^n ; def$	$n \geq 1$ alternativas primitivas
$aalt \rightarrow$	$C\ \overline{x_i}^n \rightarrow e$	$n \geq 0$ y la aridad de C es n
$palt \rightarrow$	$literal \rightarrow e$	
$def \rightarrow$	$x \rightarrow e$	alternativa por defecto
$\mid$	$default \rightarrow e$	
$lf \rightarrow$	$vs \setminus \pi\ xs \rightarrow e$	λ-forma
$atom \rightarrow$	$x \mid literal$	
$literal \rightarrow$	$\textbf{0\#} \mid \textbf{1\#} \mid \ldots$	enteros primitivos
	$\mid \ldots$	
$prim \rightarrow$	$+\# \mid -\# \mid *\# \mid /\#$	operaciones sobre enteros primitivos
	$\mid \ldots$	

Figura 2.7: Lenguaje STG (STGL)

- La pila está dividida en tres pilas: la de argumentos, la de continuaciones, y la de marcas de actualización.

Además, la expresión de control de la máquina *STG* no es directamente una expresión del lenguaje, sino que posee tres formas distintas:

- *Eval* e E, donde e es una expresión del lenguaje *STGL* (nótese que esto excluye lambda abstracciones) y E es un entorno que asocia las variables libres de la expresión e a punteros en el montón.

- *Enter* p, donde p es un puntero en el montón. Obsérvese que no existe ningún entorno en este caso.

- *ReturnCon* C $\overline{p_i}$, donde C es un constructor algebraico y $\overline{p_i}$ son sus argumentos dados como un vector de punteros en el montón. En este caso la instrucción tampoco tiene asociado un entorno.

Llamaremos a cada una de estas expresiones *instrucción*, y utilizaremos el símbolo i para denotarlas. Una configuración de la máquina *STG* corresponde con una sextupla de la forma

$$(\Gamma, i, as, rs, us, \sigma)$$

donde Γ es el montón que liga punteros a clausuras, i es la instrucción de control, as es la pila de argumentos, rs es la pila de continuaciones (así es como se denotan las alternativas de un **case** en [PL92]), us es la pila de marcas de actualización y σ es el entorno inicial. En la Figura 2.8 se encuentran las reglas de transición de la máquina *STG*.

La máquina *STG* parte de un entorno inicial σ, donde almacenan las direcciones de las ligaduras de mas alto nivel del programa ($Addr\ a$), y un montón inicial donde almacenan las clausuras de las ligaduras iniciales del programa. Así, para un programa

$$\overline{g_i = vs_i \ \backslash\pi_i \ xs_i \rightarrow e_i}$$

la configuración de partida sería la siguiente:

$$(\overline{a_i \mapsto (vs_i \ \backslash\pi_i \ xs_i \rightarrow e_i) \ (\sigma \ vs_i)},\ Eval\ (main\ \{\ \})\ \{\ \},\ \{\ \},\ \{\ \},\ \{\ \},\ \overline{g_i \mapsto (Addr\ a_i)})$$

Este entorno y montón iniciales son los que surgirían si evaluáramos la expresión

$$\textbf{letrec}\ \overline{g_i = vs_i \ \backslash\pi_i \ xs_i \rightarrow e_i}\ \textbf{in}\ main\ \{\ \}$$

Se mantiene separado para posibilitar diversas optimizaciones. Para más detalles consúltese [PL92].

En las reglas de la Figura 2.8 y en la Sección 5.7 no se muestra este entorno, ya que esto complica bastante la presentación y no se gana nada. Se considera que está incluido en el entorno E.

Aparecen dos reglas que se pueden aplicar indistintamente en los mismos casos: la 17 y la 17a. Ambas producen una actualización en el montón. Mientras que la actualización que se produce en la regla 17 tiene la forma de una lambda abstracción, la correspondiente a la aplicación parcial de la lambda abstracción que se encuentra en Γ en la dirección p a los k argumentos que se encuentran en la pila as, la actualización que se produce en la regla 17a tiene la forma de una aplicación, la correspondiente a la aplicación de la lambda abstracción que se encuentra en Γ en la dirección p a los k argumentos que se encuentran en la pila as. No obstante, los cómputos son equivalentes, y con ambas se llega a los mismos resultados. En principio la regla de actualización más lógica es la 17, sin embargo es imposible generar código en ejecución para una actualización de ese tipo, por eso se utiliza la regla 17a.

	Montón	Control	as	rs	us	Regla
	Γ	$Eval\ (x\ \overline{x_i}^n)\ (E \cup [x \mapsto p] \cup [\overline{x_i \mapsto p_i}^n])$	as	rs	us	1
$\Longrightarrow$	Γ	$Enter\ p$	$\overline{p_i}^n : as$	rs	us	
	$\Gamma \cup [p \mapsto (vs \setminus \mathbf{n}\ \{\overline{x_i}^n\} \text{->} e,\ E)]$	$Enter\ p$	$\overline{p_i}^n : as$	rs	us	2
$\Longrightarrow$	Γ	$Eval\ e\ (E \cup [\overline{x_i \mapsto p_i}^n])$	as	rs	us	
	Γ	$Eval\ (\mathbf{letrec}\ \{\overline{x_i = t_i \setminus \pi_i\ xs_i \text{->} e_i}\}\ \mathbf{in}\ e)\ E$	as	rs	us	3(1)
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto (t_i \setminus \pi_i\ xs_i \text{->} e_i,\ E'\ \vert^{t_i})}]$	$Eval\ e\ E'$	as	rs	us	
	Γ	$Eval\ (\mathbf{case}\ e\ \mathbf{of}\ alts)\ E$	as	rs	us	4
$\Longrightarrow$	Γ	$Eval\ e\ E$	as	$(alts,\ E) : rs$	us	
	Γ	$Eval\ (C\ \overline{x_i})\ (E \cup [\overline{x_i \mapsto p_i}])$	as	rs	us	5
$\Longrightarrow$	Γ	$ReturnCon\ C\ \overline{p_i}$	as	rs	us	
	Γ	$ReturnCon\ C_k\ \overline{p_i}$	as	$(alts,\ E) : rs$	us	6(2)
$\Longrightarrow$	Γ	$Eval\ e_k\ (E \cup [\overline{y_{ki} \mapsto p_i}])$	as	rs	us	
	Γ	$ReturnCon\ C\ \overline{p_i}$	as	$(\overline{C_i\ \overline{vs_i} \text{->} e_i};\ def \text{->} e_d,\ E) : rs$	us	7(3)
$\Longrightarrow$	Γ	$Eval\ e_d\ E$	as	rs	us	
	Γ	$ReturnCon\ C\ \overline{p_i}$	as	$(\overline{C_i\ \overline{vs_i} \text{->} e_i};\ v \text{->} e_d,\ E) : rs$	us	8(3)(4)
$\Longrightarrow$	$\Gamma \cup [a \mapsto (\{\overline{x_i} \setminus \mathbf{n}\ \{\} \text{->} C\ \overline{x_i})\ [\overline{x_i \mapsto p_i}]]$	$Eval\ e_d\ (E \cup [v \mapsto a])$	as	rs	us	
	Γ	$Eval\ k\ E$	as	rs	us	9
$\Longrightarrow$	Γ	$ReturnInt\ k$	as	rs	us	
	Γ	$Eval\ (x\ \{\})\ (E \cup [x \mapsto Int\ k])$	as	rs	us	10
$\Longrightarrow$	Γ	$ReturnInt\ k$	as	rs	us	
	Γ	$ReturnInt\ k$	as	$(\ldots;\ k \text{->} e; \ldots,\ E) : rs$	us	11
$\Longrightarrow$	Γ	$Eval\ e\ E$	as	rs	us	
	Γ	$ReturnInt\ k$	as	$(\overline{k_i \text{->} e_i};\ v \text{->} e_d,\ E) : rs$	us	12
$\Longrightarrow$	Γ	$Eval\ e_d\ (E \cup [v \mapsto Int\ k])$	as	rs	us	
	Γ	$ReturnInt\ k$	as	$(\overline{k_i \text{->} e_i};\ def \text{->} e_d,\ E) : rs$	us	13
$\Longrightarrow$	Γ	$Eval\ e_d\ E$	as	rs	us	
	Γ	$Eval\ (+\ \{x_1, x_2\})\ (E \cup [x_1 \mapsto Int\ i_1, x_2 \mapsto Int\ i_2])$	as	rs	us	14
$\Longrightarrow$	Γ	$ReturnInt\ (i_1 + i_2)$	as	rs	us	
	$\Gamma \cup [p \mapsto (vs \setminus \mathbf{u}\ \{\} \text{->} e,\ E)]$	$Enter\ p$	as	rs	us	15
$\Longrightarrow$	Γ	$Eval\ e\ E$	$\{\}$	$\{\}$	$(as,\ rs,\ p) : us$	
	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	$\{\}$	$(as,\ rs,\ p) : us$	16(4)
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\overline{x_i} \setminus \mathbf{n}\ \{\} \text{->} C\ \overline{x_i},\ [\overline{x_i \mapsto p_i}])]$	$ReturnCon\ C\ \overline{p_i}$	as	rs	us	
	$\Gamma \cup [p \mapsto (vs \setminus \mathbf{n}\ \{\overline{x_i}^n\} \text{->} e,\ E)]$	$Enter\ p$	$\overline{p_i}^k$	$\{\}$	$(as',\ rs,\ p') : us$	17(5)
$\Longrightarrow$	Γ'	$Enter\ p$	$\overline{p_i}^k : as'$	rs	us	
	$\Gamma \cup [p \mapsto (vs \setminus \mathbf{n}\ \{\overline{x_i}^n\} \text{->} e,\ E)]$	$Enter\ p$	$\overline{p_i}^k$	$\{\}$	$(as',\ rs,\ p') : us$	17a(6)
$\Longrightarrow$	Γ'	$Enter\ p$	$\overline{p_i}^k : as'$	rs	us	

(1) En la regla 3, las variables $\overline{p_i}$ son distintas y frescas. $E' = E \cup [\overline{x_i \mapsto p_i}]$

(2) La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en *alts*

(3) $\forall k. C_k \neq C$

(4) Las variables $\overline{x_i}$ son una secuencia de variables arbitrarias distintas.

(5) $k < n$ y $\Gamma' = \Gamma \cup [p \mapsto (vs \setminus \mathbf{n}\ \{\overline{x_i}^n\} \text{->} e,\ E)] \cup [p' \mapsto (vs \cup \{\overline{x_i}^k\} \setminus \mathbf{n}\ \{\overline{x_i}^{n-k}\} \text{->} e,\ E \cup [\overline{x_i \mapsto p_i}^k])]$

(6) $k < n$ y $\Gamma' = \Gamma \cup [p \mapsto (vs \setminus \mathbf{n}\ \{\overline{x_i}^n\} \text{->} e,\ E)] \cup [p' \mapsto (\{x, \overline{x_i}^k\} \setminus \mathbf{n}\ \{\} \text{->} x\ \overline{x_i}^k,\ [x \mapsto p] \cup [\overline{x_i \mapsto p_i}^k])]$, donde x es una variable arbitraria distinta a las $\overline{x_i}$

Figura 2.8: Máquina abstracta *STG*

Capítulo 3

Modificaciones a la semántica natural

En este capítulo desarrollaremos la demostración del *Lema 1* que Sestoft propuso en [Ses97] y para el cual dejó indicado que su demostración se realizaba por inducción sobre la estructura de la derivación. No obstante, los detalles de dicha demostración no aparecen en dicho artículo. Es más, en este capítulo veremos que no es correcta cuando se añaden las reglas semánticas para las expresiones **case**. Este lema prueba la corrección y completitud de su semántica con respecto a la semántica de Launchbury. Como Launchbury demostró que ésta era correcta y completa con respecto a la semántica denotacional, por transitividad tenemos que la de Sestoft es correcta y completa con respecto a la denotacional.

Demostraremos el *Lema 1* para todas las reglas de la semántica exceptuando las reglas *Case* y *Cons*, para las que se demostrará que es necesario una pequeña modificación en la semántica para que se siga manteniendo este lema.

Tras ello motivaremos e introduciremos una semántica en la que en los montones sólo mantendremos las clausuras vivas. Un invariante de la semántica de Sestoft es que los montones van creciendo, y nunca se eliminan las clausuras muertas de ellos, pero nosotros estamos interesados en añadir a la semántica la recolección de basura sólo en los puntos en los que sea necesaria, de tal manera que tengamos en los montones sólo las clausuras vivas en cada momento del cómputo. Con ello se expresa que la recolección de basura podría suceder en cualquier instante, sin alterar la corrección de la evaluación. Además de mantener el montón sólo con las clausuras vivas, mantendremos también el conjunto de las marcas de actualización sólo con las marcas que apunten a clausuras vivas, de manera que evitaremos la actualización de variables que han dejado de estar vivas. Se observará que esta optimización, que se realiza normalmente en el código, también se puede añadir a la semántica sin necesidad de convertirla en una semántica difícil de comprender. Así quedará más claro dónde y cuándo se pueden realizar, y cuáles son las optimizaciones posibles en la máquina abstracta.

Es más, nadie hasta ahora había reflejado en la semántica que el conjunto de marcas de actualización puede ser podado. Normalmente estas ideas aparecían en la implementación de las máquinas abstractas, manteniendo con ello una distancia bastante grande entre la implementación de las máquinas y la semántica. Por tanto, podrían surgir dudas sobre la corrección y completitud de una con respecto a la otra.

3.1 Demostración del *Lema 1*

Un lema importante para la demostración de la corrección de la semántica de Sestoft es el *Lema 1* que Sestoft propone en la página 7 de [Ses97]. Con dicho lema demuestra que en la semántica no hay captura de variables y que no se intenta ligar en el montón un puntero con dos expresiones. Demuestra por tanto que su semántica y la de Launchbury son equivalentes.

Para ello Sestoft plantea varias definiciones:

Definición 2
*Sea A un conjunto de variables. La configuración $\Gamma_0 : e_0$ es A-*bueno *si:*

1. *A y dom Γ_0 son disjuntos.*
2. *fv $(\Gamma_0 : e_0) \subseteq A \cup$ dom Γ_0.*
3. *bv $(\Gamma_0 : e_0)$ y $A \cup$ dom Γ_0 son disjuntos.*

La notación *fv* $(\Gamma : e)$ denota el conjunto de variables libres de la expresión e unido a las variables libres de las expresiones ligadas en *range* Γ. Análogamente, la notación *bv* $(\Gamma : e)$ denota el conjunto de variables ligadas de la expresión e unido a las variables ligadas de las expresiones ligadas en *range* Γ.

Sestoft nos dice que en el contexto de un juicio $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : e_1$, estos requisitos tienen el siguiente significado: (1) no hay ninguna variable ligada en el montón cuyo valor se esté computando actualmente; (2) las variables libres de la expresión e_0 y de todas las de *range* Γ son punteros que están en *dom* Γ_0, o están siendo evaluados (en A); (3) las variables ligadas de la expresión e_0 y de todas las de *range* Γ no están ligadas ni en *dom* Γ_0 ni en A (es decir, actualmente en computación).

Definición 3
El juicio $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : e_1$ es prometedor *si $\Gamma_0 : e_0$ es A-bueno.*

El *Lema 1* propuesto por Sestoft es el siguiente:

Lema 4 (*Lema 1* de Sestoft)
Sea $\Gamma_0 : e_0 \Downarrow_A \Gamma_1 : e_1$ una derivación.

$$\textit{Si } \Gamma_0 : e_0 \textit{ es } A\text{-bueno } \textit{entonces} \begin{cases} \Gamma_1 : e_1 \textit{ es } A\text{-bueno} \\ \textit{dom } \Gamma_0 \subseteq \textit{dom } \Gamma_1 \\ \textit{todo juicio en la derivación es prometedor} \end{cases}$$

La demostración, como deja indicado Sestoft, se hace por inducción sobre la estructura de la derivación. Nosotros la hemos desarrollado completamente.

Demostración:

- Lam

$$\Gamma : \lambda x.e \Downarrow_A \Gamma : \lambda x.e \quad \textit{Lam}$$

$$\Gamma : \lambda x.e \text{ es } A\text{-bueno} \Rightarrow \begin{cases} \Gamma : \lambda x.e \text{ es } A\text{-bueno} \\ \textit{dom}\, \Gamma \subseteq \textit{dom}\, \Gamma \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

- App

$$\frac{\Gamma : e \Downarrow_A \Delta : \lambda y.e' \quad \Delta : e'[p/y] \Downarrow_A \Theta : w}{\Gamma : e\,p \Downarrow_A \Theta : w} \; App$$

a) Hemos de ver que $\Gamma : e$ es A-bueno

1. $\Gamma : e\,p$ es A-bueno
   $\Rightarrow$ {por definición}
   $A \cap dom\,\Gamma = \emptyset$
2. $\Gamma : e\,p$ es A-bueno
   $\Rightarrow$ {por definición}
   $fv(\Gamma : e\,p) \subseteq A \cup dom\,\Gamma$
   $\Rightarrow$ $\{fv(\Gamma : e) \subseteq fv(\Gamma : e\,p)\}$
   $fv(\Gamma : e) \subseteq A \cup dom\,\Gamma$
3. $\Gamma : e\,p$ es A-bueno
   $\Rightarrow$ {por definición}
   $bv(\Gamma : e\,p) \cap (A \cup dom\,\Gamma) = \emptyset$
   $\Rightarrow$ $\{bv(\Gamma : e) = bv(\Gamma : e\,p)\}$
   $bv(\Gamma : e) \cap (A \cup dom\,\Gamma) = \emptyset$

b) Por hipótesis de inducción $\begin{cases} \Delta : \lambda y.e' \text{ es } A\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Delta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

c) Tenemos que ver que $\Delta : e'[p/y]$ es A-bueno

1. $\Delta : \lambda y.e'$ es A-bueno
   $\Rightarrow$ {por definición}
   $A \cap dom\,\Delta = \emptyset$
2. $\Delta : \lambda y.e'$ es A-bueno
   $\Rightarrow$ {por definición}
   $fv(\Delta : \lambda y.e') \subseteq A \cup dom\,\Delta$
   $\Rightarrow$ $\{fv(\Delta : e'[p/y]) \subseteq fv(\Delta : \lambda y.e') \cup \{p\}\}$
   $fv(\Delta : e'[p/y]) \subseteq A \cup dom\,\Delta \cup \{p\}$
   $\Rightarrow$ $\{\Gamma : e\,p$ es A-bueno luego $p \in A \cup dom\,\Gamma$ y $dom\,\Gamma \subseteq dom\,\Delta\}$
   $fv(\Delta : e'[p/y]) \subseteq A \cup dom\,\Delta$
3. $\Delta : \lambda y.e'$ es A-bueno
   $\Rightarrow$ {por definición}
   $bv(\Delta : \lambda y.e') \cap (A \cup dom\,\Delta) = \emptyset$
   $\Rightarrow$ $\{bv(\Delta : e'[p/y]) \subseteq bv(\Delta : \lambda y.e')\}$
   $bv(\Delta : e'[p/y]) \cap (A \cup dom\,\Delta) = \emptyset$

d) Por hipótesis de inducción $\begin{cases} \Theta : w \text{ es } A\text{-bueno} \\ dom\,\Delta \subseteq dom\,\Theta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

d) Recapitulando: $\begin{cases} \Theta : w \text{ es } A\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Delta \subseteq dom\,\Theta \\ \text{Este juicio es prometedor y los de sus derivaciones también.} \end{cases}$

- Var

$$\frac{\Gamma : e \Downarrow_{A \cup \{p\}} \Delta : w}{\Gamma \cup [p \mapsto e] : p \Downarrow_A \Delta \cup [p \mapsto w] : w}\ Var$$

a) Veamos que $\Gamma : e$ es $(A \cup \{p\})$-bueno

1. $\Gamma \cup [p \mapsto e] : p$ es A-bueno
$\Rightarrow$ $\{$por definición$\}$
$A \cap dom\,(\Gamma \cup [p \mapsto e] : p) = \emptyset$
$\Rightarrow$ $\{dom(\Gamma \cup [p \mapsto e]) \subseteq dom\,\Gamma\}$
$A \cap dom\,\Gamma = \emptyset$
$\Rightarrow$ $\{p$ no está ligada en Γ por definición.$\}$
$(A \cup \{p\}) \cap dom\,\Gamma = \emptyset$

2. $\Gamma \cup [p \mapsto e] : p$ es A-bueno
$\Rightarrow$ $\{$por definición$\}$
$fv(\Gamma \cup [p \mapsto e] : p) \subseteq (A \cup dom(\Gamma \cup [p \mapsto e]))$
$\Rightarrow$ $\{fv(\Gamma : e) \subseteq fv(\Gamma \cup [p \mapsto e] : p)\}$
$fv(\Gamma : e) \subseteq (A \cup dom(\Gamma \cup [p \mapsto e])) = \emptyset$
$\Rightarrow$ $\{dom(\Gamma \cup [p \mapsto e]) = dom\,\Gamma \cup \{p\}\}$
$fv(\Gamma : e) \cap (A \cup \{p\} \cup dom\,\Gamma) = \emptyset$

3. $\Gamma \cup [p \mapsto e] : p$ es A-bueno
$\Rightarrow$ $\{$por definición$\}$
$bv(\Gamma \cup [p \mapsto e] : p) \cap (A \cup dom(\Gamma \cup [p \mapsto e])) = \emptyset$
$\Rightarrow$ $\{bv(\Gamma : e) = bv(\Gamma \cup [p \mapsto e] : p)\}$
$bv(\Gamma : e) \cap (A \cup dom(\Gamma \cup [p \mapsto e])) = \emptyset$
$\Rightarrow$ $\{dom(\Gamma \cup [p \mapsto e]) = dom\,\Gamma \cup \{p\}\}$
$bv(\Gamma : e) \cap (A \cup \{p\} \cup dom\,\Gamma) = \emptyset$

b) Por hipótesis de inducción $\begin{cases} \Delta : w \text{ es } (A \cup \{p\})\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Delta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

c) Tenemos que demostrar que $\Delta \cup [p \mapsto w] : w$ es A-bueno

1. $\Delta : w$ es $(A \cup \{p\})$-bueno
$\Rightarrow$ $\{$por definición$\}$
$(A \cup \{p\}) \cap dom\,\Delta = \emptyset$
$\Rightarrow$ $\{dom(\Gamma \cup [p \mapsto e]) \cap A = \emptyset$, por lo tanto $p \notin A\}$
$A \cap dom(\Delta \cup [p \mapsto w]) = \emptyset$

2. $\Delta : w$ es $(A \cup \{p\})$-bueno

$$
\begin{array}{ll}
\Rightarrow & \quad \{\text{por definición}\} \\
& fv(\Delta : w) \subseteq (A \cup \{p\}) \cup dom\,\Delta \\
\Rightarrow & \quad \{fv(\Delta \cup [p \mapsto w] : w) \subseteq fv(\Delta : w) \cup \{p\}\} \\
& fv(\Delta \cup [p \mapsto w] : w) \subseteq A \cup dom(\Delta \cup [p \mapsto w]) \\
3. & \Delta : w \text{ es } (A \cup \{p\})\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
& bv(\Delta : w) \cap (A \cup \{p\} \cup dom\,\Delta) = \emptyset \\
\Rightarrow & \quad \{bv(\Delta \cup [p \mapsto w] : w) = bv(\Delta : w)\} \\
& bv(\Delta \cup [p \mapsto w] : w) \cap (A \cup dom(\Delta \cup [p \mapsto w])) = \emptyset
\end{array}
$$

d) Recapitulando: $\begin{cases} \Delta \cup [p \mapsto w] : w \text{ es } A\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Delta \Rightarrow dom\,\Gamma \cup [p \mapsto e] \subseteq dom\,\Delta \cup [p \mapsto w] \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

- Let

$$
\frac{\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \Downarrow_A \Delta : w}{\Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \Downarrow_A \Delta : w}\ \textit{Let}
$$

Las variables $\overline{p_i}$ son distintas y frescas. No pueden ocurrir ni en A ni en Γ ni en el **letrec**. La notación $\hat{e}$ significa $e[\overline{p_i/x_i}]$.

a) Veamos que $\Gamma[\overline{p_i \mapsto \hat{e}_i}] : \hat{e}$ es A-bueno

$$
\begin{array}{ll}
1. & \Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \text{ es } A\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
& A \cap dom\,\Gamma = \emptyset \\
\Rightarrow & \quad \{\overline{pi} \notin A\} \\
& A \cap dom(\Gamma[\overline{p_i \mapsto \hat{e}_i}]) = \emptyset \\
2. & \Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \text{ es } A\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
& fv(\Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e) \subseteq A \cup dom\,\Gamma \\
\Rightarrow & \quad \{fv(\Gamma[\overline{p_i \mapsto \hat{e}_i}] : \hat{e}) \subseteq fv(\Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e) \cup \{\overline{p_i}\}\} \\
& fv(\Gamma[\overline{p_i \mapsto \hat{e}_i}] : \hat{e}) \subseteq A \cup dom\,\Gamma[\overline{p_i \mapsto \hat{e}_i}] \\
3. & \Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \text{ es } A\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
& bv(\Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e) \cap (A \cup dom\,\Gamma) = \emptyset \\
\Rightarrow & \quad \{bv(\Gamma[\overline{p_i \mapsto \hat{e}_i}] : \hat{e}) \subseteq bv(\Gamma : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e)\} \\
& bv(\Gamma[\overline{p_i \mapsto \hat{e}_i}] : \hat{e}) \cap (A \cup dom\,\Gamma) = \emptyset \\
\Rightarrow & \quad \{\overline{p_i} \text{ son distintas y frescas}\} \\
& bv(\Gamma[\overline{p_i \mapsto \hat{e}_i}] : \hat{e}) \cap (A \cup dom(\Gamma[\overline{p_i \mapsto \hat{e}_i}])) = \emptyset
\end{array}
$$

b) Por hipótesis de inducción $\begin{cases} \Delta : w \text{ es } A\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Gamma[p_i \mapsto e_i] \subseteq dom\,\Delta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

$\square$

3.1.1 Problemas con las expresiones *Case*

Desafortunadamente este lema no se mantiene cuando añadimos la regla del *Case*. El problema que surge al añadir esta regla es que la propiedad de frescura de las variables deja de ser localmente chequeable: mientras se reduce el discriminante del **case** no se dispone de la información relativa a las variables ligadas en las alternativas del **case**. Por tanto, en el juicio $\Gamma : e \Downarrow_A \Delta : C_k\ \overline{p_j}$ se podrían crear variables frescas con esos nombres. A partir de ahí los punteros y las variables del programa dejarían de ser disjuntos. Esto provocaría que en la segunda parte de la premisa pudiera suceder una captura de variable en la expresión $e_k[\overline{p_j/y_{kj}}]$.

Un ejemplo que muestra esto es el siguiente:

Ejemplo 5

Sea la expresión **case** (**letrec** $x = C_0$ **in** $C_1\,x$) **of** $(C_1\,y_1 \Rightarrow \lambda y.y_1)$ *la evaluación de esta expresión según la semántica operacional es:*

$$\dfrac{\dfrac{[p \mapsto C_0] : C_1\,p \Downarrow_{\{\}} [p \mapsto C_0] : C_1\,p \quad Cons}{\{\} : \textbf{letrec}\ x = C_0\ \textbf{in}\ C_1\,x \Downarrow_{\{\}} [p \mapsto C_0] : C_1\,p}\,Let \qquad [p \mapsto C_0] : \lambda y.p \Downarrow_{\{\}} [p \mapsto C_0] : \lambda y.p}{\{\} : \textbf{case}\ (\textbf{letrec}\ x = C_0\ \textbf{in}\ C_1\,x)\ \textbf{of}\ (C_1\,y_1 \Rightarrow \lambda y.y_1) \Downarrow_{\{\}} [p \mapsto C_0] : \lambda y.p}\,Case$$

En la regla Let la variable p que se genera es libre si no aparece ni en $\{\}$, ni en **letrec** $x = C_0$ **in** $C_1\,x$*. Es decir, es fresca si es distinta de la x. Por tanto, p puede ser tanto la y, como la y_1, como cualquier otra variable que existiera en las alternativas del* **case** *y, por tanto, al llegar a la regla del Case y evaluar la alternativa adecuada no se cumple $(bv(e_k) \cap dom\,\Delta) = \emptyset$. La y_1 no genera problemas, pero si elegimos $p = y$ esta intersección dejaría de ser vacía y sería igual a $\{y\}$. Esta elección implicaría la existencia de captura de variables y nos llevaría a una reducción errónea.*

3.1.2 Modificación de la semántica

Para solucionar este problema vamos a realizar una modificación en las reglas semánticas de tal forma que sigamos manteniendo la propiedad de frescura localmente comprobable. Para ello introducimos un multiconjunto C de *continuaciones* asociado con cada juicio. En este multiconjunto almacenamos las alternativas de los **case** durante la evaluación del discriminante. Ahora decimos que una variable es fresca en un juicio $\Gamma : e \Downarrow_{AC} \Delta : w$ si no ocurre ni en A, ni en Γ, ni en e, ni en ninguna alternativa de C.

Esto sólo implica restringir un poco más las variable frescas, ya que ahora para que una variable sea fresca tampoco puede aparecer en C. Las reglas semánticas modificadas se reproducen en la Figura 3.1. Se puede observar que la única regla que modifica el multiconjunto C es la regla *Case*. Las demás reglas sólo lo transmiten.

Definición 6
Sea A un conjunto de variables y C un multiconjunto de alternativas. La configuración $\Gamma_0 : e_0$ es AC-bueno *si:*

1. *A y $dom\,\Gamma_0$ son disjuntos.*

$$\Gamma : \lambda x.e \Downarrow_{A\,C} \Gamma : \lambda x.e \qquad \textit{Lam}$$

$$\Gamma : C'\,\overline{p_i} \Downarrow_{A\,C} \Gamma : C'\,\overline{p_i} \qquad \textit{Cons}$$

$$\frac{\Gamma : e \Downarrow_{A\,C} \Delta : \lambda x.e' \quad \Delta : e'[p/x] \Downarrow_{A\,C} \Theta : w}{\Gamma : e\,p \Downarrow_{A\,C} \Theta : w} \qquad \textit{App}$$

$$\frac{\Gamma : e \Downarrow_{(A \cup \{p\})\,C} \Delta : w}{\Gamma \cup [p \mapsto e] : p \Downarrow_{A\,C} \Delta \cup [p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \Downarrow_{A\,C} \Delta : w}{\Gamma : \textbf{letrec } \{\overline{x_i = e_i}\} \textbf{ in } e \Downarrow_{A\,C} \Delta : w} \text{ donde } \overline{p_i} \text{ son frescas} \qquad \textit{Letrec}$$

$$\frac{\Gamma : e \Downarrow_{A\,(C \cup \{\overline{C_i\ \overline{y_{ij}}}\})} \Delta : C_k\ \overline{p_j} \quad \Delta : e_k[\overline{p_j/y_{kj}}] \Downarrow_{A\,C} \Theta : w}{\Gamma : \textbf{case } e \textbf{ of } \overline{C_i\ \overline{y_{ij}}} \Downarrow_{A\,C} \Theta : w} \qquad \textit{Case}$$

Figura 3.1: Semántica natural de Sestoft revisada

2. $fv\ (\Gamma_0 : e_0) \cup fv\ (C) \subseteq A \cup dom\,\Gamma_0$.

3. $bv\ (\Gamma_0 : e_0) \cup bv\ (C)$ *y* $A \cup dom\,\Gamma_0$ *son disjuntos.*

Donde $fv\,(C)$ *es la extensión de* $fv\,(e)$ *a continuaciones y* $bv\,(C)$ *es la extensión de* $bv\,(e)$ *a continuaciones.*

Es necesario modificar la definición de juicio prometedor de tal forma que refleje la nueva definición.

Definición 7
El juicio $\Gamma_0 : e_0 \Downarrow_{A\,C}\ \Gamma_1 : e_1$ *es prometedor si* $\Gamma_0 : e_0$ *es* AC*-bueno*

El *Lema 1* de Sestoft tal y como estaba la semántica no se cumplía al introducir la regla *Case*. Con la nueva semántica y una pequeña modificación en el lema éste pasa a cumplirse con todas las reglas. La modificación es la siguiente:

Lema 8
Sea $\Gamma_0 : e_0 \Downarrow_{A\,C} \Gamma_1 : e_1$ *una derivación.*

$$\textit{Si } \Gamma_0 : e_0 \textit{ es } AC\text{-bueno } \textit{entonces} \begin{cases} \Gamma_1 : e_1 \textit{ es } AC\text{-bueno} \\ dom\,\Gamma_0 \subseteq dom\,\Gamma_1 \\ \textit{todo juicio en la derivación es prometedor} \end{cases}$$

La demostración del lema apenas sufre variaciones. Por un lado restringimos más la elección de las variables frescas, por lo que eso no influye en la parte que ya teníamos demostrada del lema. Por otro lado añadimos que $fv(C) \subseteq A \cup dom\,\Gamma$ y que $bv(C) \cap (A \cup dom\,\Gamma) = \emptyset$. Es decir, que las variables libres de las alternativas son variables que están en el montón o en evaluación, y que las variables ligadas de las alternativas son distintas de los punteros. Se observa que todas las reglas que no modifican el conjunto C cumplen el lema, ya que se parte de una $\Gamma : e$ que es AC-bueno y, por tanto, $fv(C) \subseteq A \cup dom\,\Gamma$, y como $dom\,\Delta \subseteq dom\,\Gamma$ se llega a la conclusión de que $fv(C) \subseteq A \cup dom\,\Delta$. Por otro lado, como $bv(C) \cap (A \cup dom\,\Gamma) = \emptyset$ y las variables que añadimos en Δ no pertenecen a C, se tiene así que $bv(C) \cap (A \cup dom\,\Delta) = \emptyset$.

Así pues, solamente queda por demostrar el lema para las reglas *Case* y *Cons*, cuya demostración sigue a continuación:

Demostración:

- Cons

$$\Gamma : C'\,\overline{p_i} \Downarrow_{AC} \Gamma : C'\,\overline{p_i} \quad Cons$$

$$\Gamma : C'\,\overline{p_i} \text{ es } AC\text{-bueno} \Rightarrow \begin{cases} \Gamma : C'\,\overline{p_i} \text{ es } AC\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Gamma \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

- Case

$$\frac{\Gamma : e \Downarrow_{A(C \cup \{alts\})} \Delta : C_k\,\overline{p_j} \quad \Delta : e_k[\overline{p_j/y_{kj}}] \Downarrow_{AC} \Theta : w}{\Gamma : \textbf{case}\ e\ \textbf{of}\ alts \Downarrow_{AC} \Theta : w}\ Case$$

a) Veamos que $\Gamma : e$ es $A(C \cup alts)$-bueno

1. $\Gamma : \textbf{case}\ e\ \textbf{of}\ alts$ es AC-bueno
   $\Rightarrow$ {por definición}
   $A \cap dom\,\Gamma = \emptyset$
2. $\Gamma : \textbf{case}\ e\ \textbf{of}\ alts$ es AC-bueno
   $\Rightarrow$ {por definición}
   $fv(\Gamma : \textbf{case}\ e\ \textbf{of}\ alts) \cup fv(C) \subseteq A \cup dom\,\Gamma$
   $\Rightarrow$ $\{fv(\textbf{case}\ e\ \textbf{of}\ alts) = fv(e) \cup fv(alts)\}$
   $fv(\Gamma : e) \cup fv(C \cup alts) \subseteq A \cup dom\,\Gamma$
3. $\Gamma : \textbf{case}\ e\ \textbf{of}\ alts$ es AC-bueno
   $\Rightarrow$ {por definición}
   $(bv(\Gamma : \textbf{case}\ e\ \textbf{of}\ alts) \cup bv(C)) \cap (A \cup dom\,\Gamma) = \emptyset$
   $\Rightarrow$ $\{bv(\textbf{case}\ e\ \textbf{of}\ alts) = bv(e) \cup bv(alts)\}$
   $(bv(\Gamma : e) \cup bv(C \cup alts)) \cap (A \cup dom\,\Gamma) = \emptyset$

b) Por hipótesis de inducción $\Longrightarrow \begin{cases} \Delta : C_k\,\overline{p_j} \text{ es } A(C \cup alts)\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Delta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

c) Tenemos que ver que $\Delta : e_k[\overline{p_j/y_{kj}}]$ es $A(C \cup alts)$-bueno

$$
\begin{array}{ll}
1. & \Delta : C_k\,\overline{p_j} \text{ es } A(C \cup alts)\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
 & A \cap dom\,\Delta = \emptyset \\
2. & \Delta : C_k\,\overline{p_j} \text{ es } A(C \cup alts)\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
 & fv(\Delta : C_k\,\overline{p_j}) \cup fv(C \cup alts) \subseteq A \cup dom\,\Delta \\
\Rightarrow & \quad \{\text{ya que } fv(e_k[\overline{p_j/y_{kj}}]) \subseteq fv(C_k\ \overline{p_j}) \cup fv(alts)\} \\
 & fv(\Delta : e_k[\overline{p_j/y_{kj}}]) \cup fv(C) \subseteq A \cup dom\,\Delta \\
3. & \Delta : C_k\,\overline{p_j} \text{ es } A(C \cup alts)\text{-bueno} \\
\Rightarrow & \quad \{\text{por definición}\} \\
 & (bv(\Delta : C_k\,\overline{p_j}) \cup bv(C \cup alts)) \cap (A \cup dom\,\Delta) = \emptyset \\
\Rightarrow & \quad \{bv(e_k[\overline{p_j/y_{kj}}]) \subseteq bv(alts)\} \\
 & (bv(\Delta : e_k[\overline{p_j/y_{kj}}]) \cup bv(C)) \cap (A \cup dom\,\Delta) = \emptyset
\end{array}
$$

d) Por hipótesis de inducción $\Longrightarrow \begin{cases} \Theta : w \text{ es } AC\text{-bueno} \\ dom\,\Delta \subseteq dom\,\Theta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

e) Recapitulando: $\begin{cases} \Theta : w \text{ es } AC\text{-bueno} \\ dom\,\Gamma \subseteq dom\,\Delta \subseteq dom\,\Theta \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$

□

Demostración de corrección y completitud de la máquina ***Mark-1*****.** Por último, el único cambio necesario que hay que realizar en la demostración de la corrección y completitud que Sestoft realiza de la *Mark-1* con respecto a la semántica operacional es que ahora partimos de una configuración del tipo $\{\} : e \Downarrow_{\{\}\,\{\}} \Theta : w$. Como la máquina *Mark-1* cuando genera variables frescas exige que éstas no aparezcan en la pila S y las continuaciones pendientes están en la pila S ($\forall alts.(alts \in C \Rightarrow alts \in S)$), no existe ningún problema.

Con esta modificación queda demostrado que la nueva semántica operacional es correcta y completa con respecto a la semántica de Launchbury, y queda también demostrado que la máquina *Mark-1* es correcta y completa con respecto a esta semántica operacional.

3.2 Semántica operacional con montones minimales

Un problema no abordado por Sestoft es la recolección de basura. En cada juicio $\Gamma : e \Downarrow_A \Delta : w$ el montón Δ contiene a Γ. Nosotros estamos interesados en tener una semántica que refleje que la recolección de basura puede suceder en cualquier instante sin alterar el resultado de la evaluación. Para llevar a cabo esto, realizaremos una modificación más sobre las reglas semánticas, de tal manera que todos los montones de los juicios contendrán sólo las ligaduras vivas. Se activará el recolector de basura en los puntos adecuados para mantener este invariante. Esto nos obligará a añadir un nuevo conjunto en las derivaciones para encontrar de forma local todas las clausuras vivas en cada derivación sin necesidad de recorrer todo el árbol de derivación.

Para introducir la recolección de basura, derivaremos un nuevo conjunto de reglas en las que todos los montones que aparecerán en los juicios de una derivación serán *minimales*, es decir, sólo incluirán clausuras vivas. Para ello debemos decidir cuáles son los punteros que apuntan a clausuras vivas. Por supuesto, las variables libres de la expresión en evaluación pertenecen a este conjunto de punteros vivos. Observando las reglas, está claro que las continuaciones del multiconjunto C proveen nuevos punteros a clausuras vivas. De otra forma, durante la derivación del discriminante de un **case**, un montón minimal no contendría ligaduras para las variables libres de las alternativas y, al quedarse con la expresión relativa a la correspondiente alternativa, no tendría ligaduras en el montón para las variables libres de esta expresión y, por tanto, no podría continuar. Durante la derivación de la forma normal de la función e, en la regla *App* de la semántica operacional, debemos incluir el argumento p de la aplicación en el conjunto de variables vivas. De otra forma, un montón minimal para esta derivación podría no tener una ligadura para p (esto podría pasar si la aparición de p como argumento fuera la única referencia a p en $range\, \Gamma$, en e, o en C). Por lo tanto, y para evitar esto, añadiremos un nuevo multiconjunto B en los juicios para mantener los argumentos pendientes de aplicación. Un juicio tendrá ahora la siguiente forma:

$$\Gamma : e \Downarrow_{ABC} \Delta : w$$

donde la intención es que Γ sea minimal con respecto a e, B y C, y Δ sea minimal con respecto a w, B y C.

Posteriormente se verá que este nuevo multiconjunto B tiene una representación clara en cualquiera de las máquinas de Sestoft: es el conjunto de argumentos sin aplicar que está en la pila. Obsérvese que tanto B como C son multiconjuntos y que, sin embargo, A es un conjunto.

Como el lector habrá podido observar, el conjunto A no se considera un conjunto de punteros vivos. Este conjunto representa ligaduras actualmente bajo evaluación o, en términos más operacionales, pendientes de actualización. Si la única referencia a una actualización pendiente es la del conjunto A, esto significa que el valor de la variable libre correspondiente no se usará en el futuro. Así que la variable puede ser eliminada del conjunto A sin ningún problema y no es necesario realizar la correspondiente actualización. Además queremos mantener en nuestras derivaciones conjuntos minimales de actualizaciones pendientes. Esto significa que el conjunto A asociado a la expresión inicial de un juicio dado no tiene por qué ser el mismo que el conjunto A' asociado a la expresión final. Para tener esto en cuenta es necesaria una última modificación en los juicios. Su forma final es la siguiente:

$$\Gamma A : e \Downarrow_{BC} \Delta A' : w$$

donde Γ y A son minimales con respecto a e, B y C, y Δ y A' son minimales con respecto a w, B y C. Su significado es que la expresión e se reduce a la forma normal w comenzando con un montón Γ y un conjunto de variables bajo evaluación A y finalizando con un montón Δ y un conjunto A'.

Nuestra intención es que los montones y los conjuntos de clausuras pendientes de evaluación sean minimales. Para ello activaremos la recolección de basura y la poda de las actualizaciones pendientes donde sea necesario, y probaremos que con ello se mantiene dicha propiedad. Las reglas semánticas clarifican cuáles son los puntos en los que es necesario la recolección de basura.

3.2.1 Semántica minimal

Definición 9
Dado un montón Γ, una expresión e, un multiconjunto B de variables y un multiconjunto C de continuaciones, definimos el conjunto de variables vivas de Γ con respecto a B, C y e, denotado $vivas_\Gamma^{BCe}$:

- $vivas_\Gamma^{BCe} = fix\ (\lambda L\, .\, L \cup (fv\ e \cup B \cup fv\ C) \cup \bigcup_{p \in L}\{fv\ e' \mid (p \mapsto e') \in \Gamma\})$
- $\Gamma_{vivas}^{BCe} = \{(p \mapsto e') \in \Gamma \mid p \in vivas_\Gamma^{BCe}\}$
- $A_{vivas}^{BCe} = A \cap vivas_\Gamma^{BCe}$

donde fix denota el menor punto fijo.

El conjunto de variables vivas de un juicio, en un momento dado de la computación, corresponde de forma intuitiva con las clausuras necesarias para la evaluación de la expresión de control. Es decir, toda clausura apuntada por una variable *no* viva es seguro que no se va a utilizar en la computación de la forma normal débil de la expresión de control, ya que no es alcanzable desde ella, y tampoco se utilizará en el futuro ya que no es alcanzable desde los multiconjuntos B y C y, por ello, no será necesaria para continuar con el cómputo.

Γ_{vivas}^{BCe} y A_{vivas}^{BCe} representan, respectivamente, el montón y el conjunto podados, en los que sólo nos hemos quedado con las ligaduras o variables vivas.

En un juicio $\Gamma\ A : e \Downarrow_{BC} \Delta\ A' : w$, si queremos expresar que un montón y un conjunto de actualizaciones deben ser analizados y probablemente podados para que sean minimales, lo denotaremos $\Gamma_{gc}\ A_{gc} : e \Downarrow_{BC} \Delta\ A' : w$, de tal manera que Γ_{gc} representa Γ_{vivas}^{BCe} y A_{gc} representa A_{vivas}^{BCe}. La anotación gc marca los puntos exactos de la evaluación en los que puede ser necesario hacer una recogida de basura y/o podar el conjunto de marcas de actualización porque pueda dejar de estar viva una variable. Es seguro marcar todos los juicios con esta anotación, no obstante, esto no sería de utilidad.

El conjunto de reglas nuevas se muestra en la Figura 3.2. La idea principal es asegurar que en cada juicio $\Gamma\ A : e \Downarrow_{BC} \Delta\ A' : w$, el par (Δ, A') es minimal con respecto a w, B y C, pero sólo marcando aquellos puntos en los que posiblemente deje de ser necesaria alguna ligadura. Así, las reglas de la Figura 3.2 han sido optimizadas al máximo para que la recolección se realice sólo en los casos necesarios.

A continuación explicaremos las diferencias de nuestra semántica con respecto a la de Sestoft y algunas decisiones.

Diferencias principales con respecto a la semántica de Sestoft. Las principales diferencias entre nuestra semántica y la de Sestoft son dos:

1. Las reglas *App* y *Var* de Sestoft han sido divididas en dos en nuestra semántica. En el primer caso la distinción se ha realizado debido a que queremos lanzar el recolector de basura sólo cuando sea necesario. De hecho ambas reglas se pueden combinar en la siguiente regla:

$$\frac{\Gamma A : e \Downarrow_{(B \cup \{p\})C} \Delta A' : \lambda x.e' \quad \Delta_{gc} A'_{gc} : e'[p/x] \Downarrow_{BC} \Theta A'' : w}{\Gamma A : e\ p \Downarrow_{BC} \Theta A'' : w}$$

$$\Gamma A : \lambda x.e \Downarrow_{BC} \Gamma A : \lambda x.e \qquad \mathit{Lam}$$

$$\Gamma A : C' \overline{p_i} \Downarrow_{BC} \Gamma A : C' \overline{p_i} \qquad \mathit{Cons}$$

$$\frac{\Gamma A : e \Downarrow_{(B \cup \{p\})C} \Delta A' : \lambda x.e' \qquad x \in \mathit{fv}\ e' \qquad \Delta A' : e'[p/x] \Downarrow_{BC} \Theta A'' : w}{\Gamma A : e\ p \Downarrow_{BC} \Theta A'' : w} \qquad \mathit{AppA}$$

$$\frac{\Gamma A : e \Downarrow_{(B \cup \{p\})C} \Delta A' : \lambda x.e' \qquad x \notin \mathit{fv}\ e' \qquad \Delta_{gc} A'_{gc} : e' \Downarrow_{BC} \Theta A'' : w}{\Gamma A : e\ p \Downarrow_{BC} \Theta A'' : w} \qquad \mathit{AppB}$$

$$\frac{\Gamma\ (A \cup \{p\})_{gc} : e \Downarrow_{BC} \Delta\ (A' \cup \{p\}) : w}{\Gamma \cup [p \mapsto e] A : p \Downarrow_{BC} \Delta \cup [p \mapsto w]\ A' : w} \qquad \mathit{VarA}$$

$$\frac{\Gamma\ (A \cup \{p\})_{gc} : e \Downarrow_{BC} \Delta A' : w \qquad p \notin A'}{\Gamma \cup [p \mapsto e] A : p \Downarrow_{BC} \Delta A' : w} \qquad \mathit{VarB}$$

$$\frac{\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}]\ A : \hat{e} \Downarrow_{BC} \Delta A' : w}{\Gamma A : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \Downarrow_{BC} \Delta A' : w} \text{ donde } \overline{p_i} \text{ son frescas} \qquad \mathit{Letrec}$$

$$\frac{\Gamma A : e\ \Downarrow_{B(C \cup \{\mathit{alts}\})} \Delta A' : C_k\ \overline{p_j} \qquad \Delta_{gc} A'_{gc} : e_k[\overline{p_j/y_{kj}}] \Downarrow_{BC} \Theta A'' : w}{\Gamma A : \textbf{case}\ e\ \textbf{of}\ \mathit{alts} \Downarrow_{BC} \Theta A'' : w} \qquad \mathit{Case}$$

Figura 3.2: Semántica natural con montones y conjuntos de actualizaciones minimales

En el segundo caso la regla *VarB* no añade al montón la ligadura $[p \mapsto w]$ ya que se sabe que ahora está muerta.

2. Nuestros montones y conjuntos de actualización son siempre minimales en los correspondientes juicios.

Obviando estos detalles, nuestras reglas semánticas son exactamente las mismas.

Eliminando actualizaciones innecesarias. Esto se refleja en la regla *VarB*. Suponiendo que el par (Δ, A') es minimal con respecto a w, B y C, y sabiendo que $p \notin A'$, entonces la actualización para la variable p puede ser descartada con seguridad (comparar con la regla *VarA*).

Manteniendo el conjunto correcto de clausuras vivas. Para mantener el multiconjunto B de forma correcta, cuando se evalúa el cuerpo de una función (ver reglas *AppA* y *AppB*), es necesario añadir a este multiconjunto el argumento al que hay que aplicar ese cuerpo para seguir manteniendo viva la clausura apuntada por el argumento, tal y como hicimos con el multiconjunto C.

Es cierto que en la semántica no es necesario representar B y C como multiconjuntos, y las reglas semánticas siguen funcionando incluso si los consideramos como conjuntos. Si observamos las reglas, los conjuntos de los juicios derechos de los antecedentes hacen referencia al consecuente y nunca disminuyen. No obstante estamos interesados en que la pila S que aparece en las máquinas de Sestoft sea exactamente una reordenación de los conjuntos A, B y C y por ello mantendremos que B y C sean multiconjuntos.

Activando la recolección de basura en los puntos apropiados. La anotación gc debe escribirse en los puntos de las reglas donde pueda existir el peligro de tener ligaduras muertas, para mantener minimales el montón y el conjunto de actualizaciones. Estos puntos son los siguientes:

- En la regla *AppB*, cuando el parámetro formal de la función no aparece en el cuerpo. En este caso existe el peligro de que la ligadura para p en Δ pase a estar muerta.
- En las reglas *VarA* y *VarB*, cuando la referencia a p desaparece de la expresión actual. Puede que no exista otra referencia a p ni en e, ni en B, ni en C.
- En la regla $Case$, cuando se elige una alternativa concreta. Las alternativas que se desechan pueden tener variables libres que ahora pasan a estar muertas.

Si hacemos el análisis desde el punto de vista del montón, en las únicas reglas en que es necesario hacer una recolección es en $Case$ y en *AppB*. En la regla *AppB* puede dejar de ser alcanzable una clausura, y con ella todas o parte de las alcanzables desde ella. En la regla $Case$ pueden quedarse muertas todas las variables libres de las alternativas no seleccionadas y todas las alcanzables desde ellas.

Suponiendo que no existe código muerto en *letrec*. Esto se refleja en la regla *Letrec*, donde no se aplica la recolección de basura, ya que todas las ligaduras del **letrec** son necesarias para la reducción de éste. Por ello escribimos el antecedente de la regla como $\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}]\ A : \hat{e}$ en lugar de $(\Gamma \cup [\overline{p_i \mapsto \hat{e}_i}])_{gc}\ A_{gc} : \hat{e}$. La definición formal de esta exigencia es la siguiente:

Definición 10

- *Sea la expresión* **letrec** $\{\overline{x_i = e_i}\}$ **in** e*; definimos las* ligaduras alcanzables*, denotado por* LA (**letrec** $\{\overline{x_i = e_i}\}$ **in** e)*, como el conjunto* $fix(\lambda X.X \cup (fv(e) \cap \{\overline{x_j}\}) \cup \{(fv(e_i) \cap \{\overline{x_j}\}) \mid x_i \in X\}$ *siendo* fix *el menor punto fijo.*
- *Una expresión* **letrec** $\{\overline{x_i = e_i}\}$ **in** e *es minimal si* LA (**letrec** $\{\overline{x_i = e_i}\}$ **in** e) $= \bigcup_i \{x_i\}$

El conjunto de ligaduras alcanzables corresponde intuitivamente con las variables necesarias para la evaluación de la expresión de control del **letrec**. Así un **letrec** minimal es aquél en el que todas las variables ligadas en él son alcanzables desde la expresión de control.

Vamos a exigir a las expresiones **letrec** que sean minimales. Esto implica que las ligaduras del **letrec** sean alcanzables desde la expresión e o recursivamente desde las expresiones ligadas que son alcanzables. Esta propiedad nos parece razonable y exigible en cualquier compilador de un lenguaje funcional con expresiones de tipo **let**, ya que todo **let** es transformable a uno minimal. Además, la semántica llega al mismo resultado con uno que con otro. Con el **let** minimal se ahorra almacenar en el montón ligaduras para clausuras que no son alcanzables.

Nueva definición de frescura. En el consecuente de la regla *Letrec* se crea el conjunto $\overline{p_i}$ de variables frescas. Una variable es *fresca* en un juicio $\Gamma A : e \Downarrow_{BC} \Delta A' : w$ si no pertenece ni a *dom* Γ ni a A, y no está ligada ni en *range* Γ, ni en e ni en C.

3.2.2 Propiedades de la semántica minimal.

Con esta semántica se ve claramente que se puede realizar una recolección de basura desde cualquier punto de la evaluación sin perder por ello nada necesario para la evaluación. Además se observa de forma detallada dónde y por qué motivos es necesaria la recogida de basura. Es más, esta semántica implica una máquina con recolección de basura incluso de las marcas de actualización, o sea, una recolección completa.

Por otro lado, los conjuntos A, B y C tienen una representación clara en cualquiera de las máquinas de Sestoft. Están representados en la pila S: el conjunto A como los punteros de actualizaciones, el multiconjunto B como los punteros a los argumentos sin aplicar, y el multiconjunto C como las alternativas de los **case**.

Ahora demostraremos que la propiedad de minimalidad de nuestra semántica se satisface.

Definición 11
*Sea A un conjunto de variables, B un multiconjunto de punteros y C un multiconjunto de alternativas. La configuración $\Gamma : e$ es ABC-*bueno *si:*

1. *A y $dom\,\Gamma$ son disjuntos.*

2. *$vivas_{\Gamma}^{BCe} = A \cup dom\,\Gamma$.*

3. *$bv(\Gamma) \cup bv(e) \cup bv(C)$ y $A \cup dom\,\Gamma$ son disjuntos.*

La primera propiedad significa que las variables del programa y los punteros son disjuntos. La segunda propiedad implica que en el montón y en A sólo se encuentran las variables vivas necesarias en ese momento de la evaluación. Esta propiedad es mucho más restrictiva que la de Sestoft. La tercera propiedad expresa que las variables libres son distintas de las ligadas.

Definición 12
Sea $\Gamma\ A : e \Downarrow_{B\,C}\ \Delta\ A' : w$ un juicio. Decimos que es prometedor si $\Gamma\ A : e$ es ABC-bueno

Esta definición asegura que el punto de partida de una derivación cumple los requisitos que queremos para la derivación completa. La siguiente proposición y el corolario establecen que dichas propiedades se mantienen en todos los juicios de la derivación.

Proposición 13
Sea $\Gamma\ A : e \Downarrow_{B\,C}\ \Delta\ A' : w$ una derivación usando las reglas semánticas. Si es un juicio prometedor entonces:

1. *La configuración $\Delta : w$ es $A'BC$-bueno*

2. *$A' \subseteq A$*

3. *Todo juicio en la derivación es prometedor.*

Corolario 14
Sea e *una expresión cerrada y* $\Gamma\{\} : e \Downarrow_{\{\}\{\}} \Delta\{\} : w$ *una derivación. Entonces,*

1. *En ninguna aplicación de las reglas AppA y Case existe captura de variables en las sustituciones de la forma* $e[p/x]$.

2. p *no está previamente ligada en* Δ *en ninguna aplicación de la regla VarA.*

La demostración de esta proposición es bastante similar a la demostración realizada anteriormente para el *Lema 1.* No obstante la vamos a desarrollar en detalle, ya que tanto el conjunto A como el montón pueden ir disminuyendo y, por lo tanto, hay que tener en cuenta que no eliminamos ninguna ligadura necesaria. Para ello utilizaremos el siguiente lema cuya demostración es obvia a partir de la definición de *vivas*.

Lema 15
$\forall\ \Gamma, A, B, C, e$

1. *Si* $vivas_{\Gamma}^{BCe} \subseteq A \cup dom\ \Gamma$ *entonces* $vivas_{\Gamma}^{BCe} = A_{gc} \cup dom\ \Gamma_{gc}$

2. $vivas_{\Gamma}^{BCe} = vivas_{\Gamma_{gc}}^{BCe}$

Demostración: (Proposición 13)
La demostración se realiza por inducción sobre la estructura de la derivación.

- Lam

  $\Gamma : \lambda x.e$ es ABC-bueno, entonces:

$$\left\{\begin{array}{l} \Gamma : \lambda x.e \text{ es } ABC\text{-bueno} \\ A \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{array}\right.$$

- Cons

  $\Gamma : C\ \overline{p_i}$ es ABC-bueno, entonces:

$$\left\{\begin{array}{l} \Gamma : C\ \overline{p_i} \text{ es } ABC\text{-bueno} \\ A \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{array}\right.$$

- AppA

  a) Probaremos que $\Gamma A : e$ es $A(B \cup \{p\})C$-bueno

  1. $\quad \Gamma : e\ p$ es ABC-bueno
     $\Rightarrow \quad \{$por definición$\}$
     $A \cap dom\ \Gamma = \emptyset$

2. $\Gamma : e\ p$ es ABC-bueno
$\Rightarrow$ {por definición}
$vivas_{\Gamma}^{BC(e\ p)} = A \cup dom\ \Gamma$
$\Rightarrow$ {por definición de *vivas* y ya que $(B \cup \{p\}) \cup fv(e) = B \cup fv(e\ p)$}
$vivas_{\Gamma}^{(B\cup\{p\})Ce} = A \cup dom\ \Gamma$

3. $\Gamma : e\ p$ es ABC-bueno
$\Rightarrow$ {por definición}
$(bv(\Gamma : e\ p) \cup bv\ C) \cap (A \cup dom\ \Gamma) = \emptyset$
$\Rightarrow$ $\{bv(\Gamma : e) = bv(\Gamma : e\ p)\}$
$(bv(\Gamma : e) \cup bv\ C) \cap (A \cup dom\ \Gamma) = \emptyset$

b) Por hipótesis de inducción:

$$\left\{ \begin{array}{l} \Delta : \lambda x.e' \text{ es } A'(B \cup \{p\})C\text{-bueno} \\ A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{array} \right.$$

c) Veremos ahora que si $x \in fv\ e'$ entonces $\Delta : e'[p/x]$ es $A'BC$-bueno.

1. $\Delta : \lambda x.e'$ es $A'(B \cup \{p\})C$-bueno
$\Rightarrow$ {por definición}
$A' \cap dom\ \Delta = \emptyset$

2. $\Delta : \lambda x.e'$ es $A'(B \cup \{p\})C$-bueno
$\Rightarrow$ {por definición}
$vivas_{\Delta}^{(B\cup\{p\})C(\lambda x.e')} = A' \cup dom\ \Delta$
$\Rightarrow$ {por definición de *vivas* y debido a que si $x \in fv\ e'$ entonces $(B \cup \{p\}) \cup fv(\lambda x.e') = B \cup fv(e'[p/x])$}
$vivas_{\Delta}^{BC(e'[p/x])} = A' \cup dom\ \Delta$

3. $\Delta : \lambda x.e'$ es $A'(B \cup \{p\})C$-bueno
$\Rightarrow$ {por definición}
$(bv(\Delta : \lambda x.e') \cup bv\ C) \cap (A' \cup dom\ \Delta) = \emptyset$
$\Rightarrow$ $\{bv(\Delta : e'[p/x]) \subseteq bv(\Delta : \lambda x.e')\}$
$(bv(\Delta : e'[p/x]) \cup bv\ C) \cap (A' \cup dom\ \Delta) = \emptyset$

d) Por hipótesis de inducción:

$$\left\{ \begin{array}{l} \Theta : w \text{ es } A''BC\text{-bueno} \\ A'' \subseteq A' \\ \text{todo juicio en la derivación es prometedor.} \end{array} \right.$$

e) Por lo tanto:

$$\left\{ \begin{array}{l} \Theta : w \text{ es } A''BC\text{-bueno} \\ A'' \subseteq A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{array} \right.$$

- <u>AppB</u>

a) La demostración es la misma que en AppA(a).

b) Por hipótesis de inducción tenemos:

$$\begin{cases} \Delta : \lambda x.e' \text{ es } A'(B \cup \{p\})C\text{-bueno} \\ A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

c) Ahora demostraremos que si $x \notin fv\ e$ entonces $\Delta_{gc} : e'$ es $A'_{gc}BC$-bueno.

1. $\Delta : \lambda x.e'$ es $A'(B \cup \{p\})C$-bueno
$\Rightarrow$ $\{$por definición$\}$
$A' \cap dom\ \Delta = \emptyset$
$\Rightarrow$ $\{A'_{gc} \subseteq A',\ dom\ \Delta_{gc} \subseteq dom\ \Delta\}$
$A'_{gc} \cap dom\ \Delta_{gc} = \emptyset$

2. $\Delta : \lambda x.e'$ es $A'(B \cup \{p\})C$-bueno
$\Rightarrow$ $\{$por definición$\}$
$vivas_{\Delta}^{(B\cup\{p\})C(\lambda x.e)} = A' \cup dom\ \Delta$
$\Rightarrow$ $\{$por definición de *vivas* y como $x \notin fv\ e$ entonces $vivas_{\Delta}^{BCe'} \subseteq vivas_{\Delta}^{(B\cup\{p\})C(\lambda x.e')}\}$
$vivas_{\Delta}^{BCe'} \subseteq A' \cup dom\ \Delta$
$\Rightarrow$ $\{$por el Lema 15 (1)$\}$
$vivas_{\Delta}^{BCe')} = A'_{gc} \cup dom\ \Delta_{gc}$
$\Rightarrow$ $\{$por el Lema 15 (2)$\}$
$vivas_{\Delta_{gc}}^{BCe'} = A'_{gc} \cup dom\ \Delta_{gc}$

3. $\Delta : \lambda x.e'$ es $A'(B \cup \{p\})C$-bueno
$\Rightarrow$ $\{$por definición$\}$
$(bv(\Delta : \lambda x.e') \cup bv\ C) \cap (A' \cup dom\ \Delta) = \emptyset$
$\Rightarrow$ $\{bv(\Delta_{gc} : e'[p/x]) \subseteq bv(\Delta : \lambda x.e')\}$
$(bv(\Delta_{gc} : e'[p/x]) \cup bv\ C) \cap (A'_{gc} \cup dom\ \Delta_{gc}) = \emptyset$

d) Por hipótesis de inducción:

$$\begin{cases} \Theta : w \text{ es } A''BC\text{-bueno} \\ A'' \subseteq A'_{gc} \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

e) Por lo tanto:

$$\begin{cases} \Theta : w \text{ es } A''BC\text{-bueno} \\ A'' \subseteq A'_{gc} \subseteq A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

- VarA

a) Veremos que $\Gamma : e$ es $(A \cup \{p\})_{gc}BC$-bueno

1. $\Gamma \cup [p \mapsto e] : p$ es ABC-bueno.
$\Rightarrow$ {por definición}
$A \cap dom\ (\Gamma \cup [p \mapsto e]) = \emptyset$
$\Rightarrow$ $\{dom\ \Gamma \subset dom(\Gamma \cup [p \mapsto e])\}$
$A \cap dom\ \Gamma = \emptyset$
$\Rightarrow$ $\{p$ no está ligada en $\Gamma\}$
$(A \cup \{p\}) \cap dom\ \Gamma = \emptyset$
$\Rightarrow$ $\{(A \cup \{p\})_{gc} \subseteq (A \cup \{p\})\}$
$(A \cup \{p\})_{gc} \cap dom\ \Gamma = \emptyset$
2. $\Gamma \cup [p \mapsto e] : p$ es ABC-bueno.
$\Rightarrow$ {por definición}
$vivas^{BCp}_{\Gamma\cup[p\mapsto e]} = A \cup dom(\Gamma \cup [p \mapsto e])$
$\Rightarrow$ {por definición de *vivas* $vivas^{BCe}_{\Gamma} \cup \{p\} = vivas^{BCp}_{\Gamma\cup[p\mapsto e]}\}$
$vivas^{BCe}_{\Gamma} \cup \{p\} = A \cup \{p\} \cup dom\ \Gamma$
$\Rightarrow$ $\{p$ es la única que puede morir y lo hace si no pertenece a $fv\ (range\ \Gamma)\}$
$vivas^{BCe}_{\Gamma} = (A \cup \{p\})_{gc} \cup dom\ \Gamma$
3. $\Gamma \cup [p \mapsto e] : p$ es ABC-bueno.
$\Rightarrow$ {por definición}
$(bv(\Gamma \cup [p \mapsto e] : p) \cup bv\ C) \cap (A \cup dom(\Gamma \cup [p \mapsto e])) = \emptyset$
$\Rightarrow$ $\{bv(\Gamma : e) = bv(\Gamma[p \mapsto e] : p)\}$
$(bv(\Gamma : e) \cup bv\ C) \cap (A \cup dom(\Gamma \cup [p \mapsto e])) = \emptyset$
$\Rightarrow$ $\{dom(\Gamma \cup [p \mapsto e]) = dom\ \Gamma \cup \{p\}\}$
$(bv(\Gamma : e) \cup bv\ C) \cap (A \cup \{p\} \cup dom\ \Gamma) = \emptyset$
$\Rightarrow$ $\{(A \cup \{p\})_{gc} \subseteq (A \cup \{p\})\}$
$(bv(\Gamma : e) \cup bv\ C) \cap ((A \cup \{p\})_{gc} \cup dom\ \Gamma) = \emptyset$

b) Por hipótesis de inducción:

$$\begin{cases} \Delta : w \text{ es } (A' \cup \{p\})BC\text{-bueno} \\ (A' \cup \{p\}) \subseteq (A \cup \{p\})_{gc} \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

c) Veremos que $\Delta \cup [p \mapsto w] : w$ es $A'BC$-bueno.

1. $\Delta : w$ es $(A' \cup \{p\})BC$-bueno
$\Rightarrow$ {por definición}
$(A' \cup \{p\}) \cap dom\ \Delta = \emptyset$
$\Rightarrow$ $\{(dom\ \Gamma \cup [p \mapsto e]) \cap A = \emptyset$, por esto $p \notin A \supset A'\}$
$A' \cap dom(\Delta \cup [p \mapsto w]) = \emptyset$
2. $\Delta : w$ es $(A' \cup \{p\})BC$-bueno
$\Rightarrow$ {por definición}
$vivas^{BCw}_{\Delta} = (A' \cup \{p\}) \cup dom\ \Delta$
$\Rightarrow$ $\{p$ está viva por tanto $vivas^{BCw}_{\Delta} = vivas^{BCw}_{\Delta\cup[p\mapsto w]}\}$
$vivas^{BCw}_{\Delta\cup[p\mapsto w]} = A' \cup dom\ (\Delta \cup [p \mapsto w])$

3. $\quad \Delta : w$ es $(A' \cup \{p\})BC$-bueno
$\Rightarrow \quad \{$por definición$\}$
$bv(\Delta : w) \cap (A' \cup \{p\} \cup dom\ \Delta) = \emptyset$
$\Rightarrow \quad \{bv(\Delta \cup [p \mapsto w] : w) = bv(\Delta : w)\}$
$bv(\Delta \cup [p \mapsto w] : w) \cap (A' \cup dom(\Delta \cup [p \mapsto w])) = \emptyset$

d) Se cumple que:

$$\begin{cases} \Delta \cup [p \mapsto w] : w \text{ es } A'BC\text{-bueno} \\ A' \subseteq (A' \cup \{p\}) \subseteq (A \cup \{p\})_{gc} \subseteq A \cup \{p\} \text{ como } p \notin A' \text{ entonces } A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

- <u>VarB</u>

a) La demostración es la misma que en VarA(a).

b) Por hipótesis de inducción:

$$\begin{cases} \Delta : w \text{ es } A'BC\text{-bueno} \\ A' \subseteq (A \cup \{p\})_{gc} \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

c) Por tanto, se cumple:

$$\begin{cases} \Delta : w \text{ es } A'BC\text{-bueno} \\ A' \subseteq (A \cup \{p\})_{gc} \subseteq (A \cup \{p\}) \text{ como } p \notin A' \text{ entonces } A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

- <u>Letrec</u>

a) Probaremos que $\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]} : \hat{e}$ es ABC-bueno.

1. $\quad \Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$ es ABC-bueno
$\Rightarrow \quad \{$por definición$\}$
$A \cap dom\ \Gamma = \emptyset$
$\Rightarrow \quad \{\overline{p_i} \notin A\}$
$A \cap dom(\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]}) = \emptyset$

2. $\quad \Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$ es ABC-bueno
$\Rightarrow \quad \{$por definición$\}$
$vivas_{\Gamma}^{BC(\textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e)} = A \cup dom\ \Gamma$
$\Rightarrow \quad \{\textbf{letrec}$ son minimales y
$fv(\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]} : \hat{e}) = fv(\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e) \cup \{\overline{p_i}\}\}$
$vivas_{\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]}}^{BC\hat{e}} = A \cup dom\ (\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]})$

3. $\quad \Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e$ es ABC-bueno
$\Rightarrow \quad \{$por definición$\}$
$(bv(\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e) \cup bv\ C) \cap (A \cup dom\ \Gamma) = \emptyset$
$\Rightarrow \quad \{bv(\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]} : \hat{e}) \subseteq bv(\Gamma : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e)\}$
$(bv(\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]} : \hat{e}) \cup bv\ C) \cap (A \cup dom\ \Gamma) = \emptyset$
$\Rightarrow \quad \{\overline{p_i}$ son distintas y frescas$\}$
$(bv(\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]} : \hat{e}) \cup bv\ C) \cap (A \cup dom(\Gamma \cup \overline{[p_i \mapsto \hat{e}_i]})) = \emptyset$

b) Por hipótesis de inducción:

$$\begin{cases} \Delta : w \text{ es } A'BC\text{-bueno} \\ A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

- <u>Case</u>

a) Veremos ahora que $\Gamma : e$ es $AB(C \cup alts)$-bueno

1. Γ : **case** e **of** $alts$ es ABC-bueno
$\Rightarrow$ {por definición}
$A \cap dom\ \Gamma = \emptyset$

2. Γ : **case** e **of** $alts$ es ABC-bueno
$\Rightarrow$ {por definición}
$vivas_{\Gamma}^{BC(\mathbf{case}\ e\ \mathbf{of}\ alts)} = A \cup dom\ \Gamma$
$\Rightarrow$ $\{fv\ \mathbf{case}\ e\ \mathbf{of}\ alts = fv\ e \cup fv\ alts\}$
$vivas_{\Gamma}^{B(C\cup\{alts\})e} = A \cup dom\ \Gamma$

3. Γ : **case** e **of** $alts$ es ABC-bueno
$\Rightarrow$ {por definición}
$(bv(\Gamma : \mathbf{case}\ e\ \mathbf{of}\ alts) \cup bv\ C)$
$\cap (A \cup dom\ \Gamma) = \emptyset$
$\Rightarrow$ $\{bv\ \mathbf{case}\ e\ \mathbf{of}\ alts = bv\ e \cup bv\ alts\}$
$(bv(\Gamma : e) \cup bv(C \cup \{alts\})) \cap (A \cup dom\ \Gamma) = \emptyset$

b) Por hipótesis de inducción:

$$\begin{cases} \Delta : C_k\ \overline{p_i} \text{ es } A'B(C \cup \{alts\})\text{-bueno} \\ A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

c) Ahora veremos que $\Delta_{gc} : e_k[\overline{p_i/y_{ki}}]$ es $A'_{gc}BC$-bueno.

1. $\Delta : C_k\ \overline{p_i}$ es $A'B(C \cup \{alts\})$-bueno
$\Rightarrow$ {por definición}
$A' \cap dom\ \Delta = \emptyset$
$\Rightarrow$ $\{A'_{gc} \subseteq A'$ y $dom\ \Delta_{gc} \subseteq dom\ \Delta\}$
$A'_{gc} \cap dom\ \Delta_{gc} = \emptyset$

2. $\Delta : C_k\ \overline{p_i}$ es $A'B(C \cup \{alts\})$-bueno
$\Rightarrow$ {por definición}
$vivas_{\Delta}^{B(C\cup\{alts\})(C_k\ \overline{p_i})} = A' \cup dom\ \Delta$
$\Rightarrow$ $\{fv\ e_k[\overline{p_i/y_{ki}}] \subseteq fv\ C_k\ \overline{p_i} \cup fv\ alts\}$
$vivas_{\Delta_{gc}}^{BC(e_k[\overline{p_i/y_{ki}}])} = A'_{gc} \cup dom\ \Delta_{gc}$

3. $\Delta : C_k\ \overline{p_i}$ es $A'B(C \cup \{alts\})$-bueno
$\Rightarrow$ {por definición}
$(bv(\Delta : C_k\ \overline{p_i}) \cup bv(C \cup \{alts\})) \cap (A' \cup dom\ \Delta) = \emptyset$
$\Rightarrow$ $\{bv\ e_k[p_i/y_{ki}] \subseteq bv\ alts, A'_{gc} \subseteq A'$ y $\Delta_{gc} \subseteq \Delta\}$
$(bv(\Delta_{gc} : e_k[p_i/y_{ki}]) \cup bv\ C) \cap (A'_{gc} \cup dom\ \Delta_{gc}) = \emptyset$

d) Por hipótesis de inducción:

$$\begin{cases} \Theta : w \text{ es } A''BC\text{-bueno} \\ A'' \subseteq A'_{gc} \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

e) Por lo tanto:

$$\begin{cases} \Theta : w \text{ es } A''BC\text{-bueno} \\ A'' \subseteq A'_{gc} \subseteq A' \subseteq A \\ \text{todo juicio en la derivación es prometedor.} \end{cases}$$

□

Hemos visto que las variables libres de los juicios están siempre definidas en el montón, o pertenecen al conjunto de actualizaciones pendientes, por tanto, podemos asegurar que ambas semánticas producen exactamente las mismas derivaciones.

Una propiedad interesante que se deduce de cada una de las semánticas es que si partimos de una configuración A-bueno, AC-bueno o ABC-bueno respectivamente, las variables que se añaden a A no se encuentran previamente en A, ya que por ser A-bueno, AC-bueno o ABC-bueno respectivamente, $\mathit{dom}\,\Gamma \cap A = \emptyset$, y las variables que se añaden pertenecen a $\mathit{dom}\,\Gamma$. Por lo tanto en A nunca se va a intentar meter una variable que estuviera previamente allí. Esto parece razonable, ya que el conjunto A representa las clausuras que están actualmente en evaluación, y no parece lógico entrar en una clausura que esté en evaluación hasta que no se reduzca a su forma normal. Por ello A se representa como un conjunto y no como un multiconjunto. Esto, además, nos lleva a la siguiente propiedad de cada una de las máquinas de Sestoft: en la pila S nunca tenemos dos marcas de actualización apuntando a la misma clausura, ya que el conjunto A se representa dentro de la pila como las marcas de actualización.

En una implementación, las variables p son en realidad punteros a direcciones de memoria y, por tanto, no podemos quitar del montón una dirección p. Lo que haremos es hacer que apunte a una clausura especial llamada *agujero negro*. Si la expresión de control intenta acceder a ella, el cómputo queda paralizado. Por tanto, el conjunto A está representado por las clausuras del montón que apuntan a la clausura de tipo agujero negro. Esto es seguro ya que en la evaluación de la parte superior de la regla *Var* no se utiliza la dirección p.

Capítulo 4

Propiedades de las máquinas de Sestoft

Sestoft ya demostró que la máquina *Mark-1* es correcta y completa con respecto a la semántica operacional (y por transitividad con respecto a la denotacional). También demostró que las máquinas *Mark-1* y *Mark-2* son equivalentes.

El análisis de estas máquinas nos ha llevado a demostrar ciertas propiedades interesantes de la .*Mark-2* que clarifican varias ideas relativas a los entornos y a las marcas de actualización que existen en la pila. En este capítulo presentaremos formalmente dichas propiedades junto con su demostración.

Además, una de las cosas más difíciles de comprender es el paso de una máquina abstracta con nombres de variables a una máquina abstracta con las expresiones de De Bruijn equivalentes. Aparentemente este cambio implica una gran dificultad, por eso nosotros demostraremos cómo se puede pasar de la *Mark-2* a la *Mark-3* suavemente. Para ello primeramente formalizaremos el paso de una expresión con variables de programa a otra con índices de De Bruijn, tras lo cual desarrollaremos una máquina *Mark-23* que demostraremos que es equivalente a cada una de las otras dos, *Mark-2* y *Mark-3*.

4.1 Propiedades generales

Algunas propiedades interesantes que se cumplen en cualquiera de las máquinas de Sestoft y que tienen relación con la semántica son las siguientes:

Proposición 16
Sea S la pila de cualquier estado de una derivación de cualquiera de las máquinas de Sestoft. Entonces se cumple lo siguiente:

1. *Sea $\#S$ las marcas de actualizaciones de la pila S. En $\#S$ no existen repeticiones y, por tanto, es un conjunto, y además $\#S$ es idéntico al conjunto A de la semántica extendida de la Sección 3.2 si sobre él no aplicamos la recolección de basura.*

2. *Sea $arg\ S$ el multiconjunto de argumentos de la pila S. Entonces $arg\ S$ es idéntico al multiconjunto B de la semántica operacional de la Sección 3.2.*

3. *Sea cont S el multiconjunto de continuaciones de la pila S. Entonces cont S corresponde fielmente con el multiconjunto correspondiente C de la semántica operacional de la Sección 3.2.*

La demostración de esta correspondencia es trivial y se obtiene directamente a partir de la demostración de la proposición que establecía la corrección y completitud de la semántica operacional con respecto a la máquina *Mark-1*.

Por otro lado, en el capítulo anterior se demostró que en el conjunto A no se añade dos veces el mismo puntero. Recordemos que A representa las clausuras actualmente en evaluación y, por tanto, no se debería entrar en una clausura en evaluación cuando ésta está siendo evaluada. Si esto ocurriera indicaría la presencia de un bucle infinito. Cualquiera de las semánticas expuestas es capaz de detectar estos casos y parar.

4.2 Entornos de la máquina *Mark-2*

En esta sección vamos a demostrar que los entornos de la *Mark-2* son consistentes, es decir, no pueden crecer de forma indefinida, y en ellos se encuentran sólo las variables en ámbito en la expresión asociada al entorno.

En la máquina *Mark-2* una configuración consiste en un montón Γ, una expresión de control e probablemente con variables libres, un entorno E que asocia las variable libres de la expresión de control a punteros del montón, y una pila S. El montón Γ es una función de punteros a clausuras, cada clausura es un par (e, E) formado por una expresión e y un entorno E que liga sus variables libres a punteros. La pila contiene tres tipos de objetos: (1) argumentos pendientes de aplicación, representados por punteros; (2) continuaciones, cada una consistente en un par $(alts, E)$ donde *alts* es un vector de alternativas de un **case** y E es un entorno que liga sus variables libres; y (3) marcas de actualización de la forma $\#p$, donde p es un puntero.

Las reglas operacionales de la máquina *Mark-2* se mostraron en la Figura 2.5 (ver [Ses97] para una explicación mas exhaustiva de su comportamiento).

La máquina *Mark-2* nunca realiza sustituciones de las variables libres del programa por los punteros correspondientes, en comparación con cualquiera de las semánticas, que sí las realizan. Sin embargo, mantiene los entornos que asocian las variables libres del programa con los correspondientes punteros. Si e es una expresión cerrada, la configuración inicial de la máquina es $(\{\}, e, \{\}, [\,])$. La máquina finaliza cuando no hay ninguna regla que pueda ser aplicada. Si la configuración final tiene la forma $(\Gamma, w, E, [\,])$, decimos que w ha sido derivada con éxito desde e y escribimos $(\{\}, e, \{\}, [\,]) \Rightarrow^* (\Gamma, w, E, [\,])$ ($\Rightarrow^*$ es el cierre simétrico, reflexivo y transitivo de $\Rightarrow$). Si tiene la forma $(\Gamma, x, E \cup [x \mapsto p], S)$, con $p \notin dom\ \Gamma$, entonces la derivación falla. El fallo corresponde al intento de entrar en un agujero negro. También puede darse el caso de que la máquina no termine debido a que cicle y no sea capaz de alcanzar una forma normal débil, o que el programa no sea capaz de alcanzar una forma normal por estar mal tipado.

4.2.1 Propiedades de los entornos

El principal teorema demostrado por Sestoft —de hecho sólo lo demostró formalmente para la máquina *Mark-1*, y posteriormente razonó que la máquina *Mark-2* realizaba exactamente los mismos pasos de reducción que la máquina *Mark-1*—, es que las derivaciones exitosas de la máquina son exactamente las mismas que la de la semántica.

Proposición 17 *(Sestoft)*
Para toda expresión cerrada e, tenemos que:

$$(\{\}, e, \{\}, [\,]) \Rightarrow^* (\Gamma, w, E, [\,]) \text{ si y sólo si } \{\}\ e :\Downarrow_{\{\}\{\}} \Gamma : w$$

Nótese que la corrección y completitud de la máquina *Mark-2* con respecto a la semántica operacional no sólo se mantiene con los programas bien tipados. Si un programa está mal tipado como el siguiente

$$\textbf{letrec } x = Nil; y = \lambda z.z \textbf{ in } x\ y$$

ni la máquina ni la semántica pueden derivar una forma normal para él.

A continuación expondremos algunas definiciones previas a las propiedades relativas a los entornos.

Definición 18
Una clausura (e, E)*, donde e es una expresión y E un entorno, es* consistente *si:*

- *Todas la variables de* $bv(e)$ *son distintas.*
- $fv(e) \subseteq dom\ E$
- $bv(e) \cap dom\ E = \emptyset$

Esta definición se extiende de forma fácil a continuaciones del tipo $(alts, E)$.

Definición 19
Γ *es* consistente *si:* $\forall (e, E) \in range\, \Gamma,\ (e, E)$ *es consistente.*

Definición 20
(Γ, e, E, S) *configuración de la Mark-2 es* consistente *si:*

- Γ *es consistente.*
- (e, E) *es consistente.*
- $\forall (alts, E) \in S.\ (alts, E)$ *es consistente.*

Obligamos a que las variables ligadas de la expresión inicial sean distintas entre sí, para que al incluirlas en el entorno no haya que comprobar si ya estaban previamente.

Proposición 21
Sea e una expresión cerrada en la que todas las variables ligadas son distintas, y sea $(\{\}, e, \{\}, [\,]) \Rightarrow^* (\Gamma, e', E, S)$ *una derivación (posiblemente parcial) de la máquina Mark-2. Entonces,*

1. (Γ, e', E, S) *es consistente.*

2. *E liga sólo las variables en ámbito de la expresión e'.*

3. *Toda clausura $(e, E) \in \Gamma$ cumple que E liga exactamente las variables en ámbito de la expresión e.*

4. *Todo par $(alts, E) \in S$ cumple que E liga exactamente las variables en ámbito de la continuación $alts$.*

Demostración:

Por inducción sobre la longitud de la derivación.

<u>**caso base**</u> Cumple trivialmente la proposición.

<u>**paso de inducción**</u> $(\{\}, e, \{\}, [\,]) \Rightarrow \ldots \Rightarrow (\Gamma_n, e_n, E_n, S_n) \Rightarrow (\Gamma_{n+1}, e_{n+1}, E_{n+1}, S_{n+1})$

- Por hipótesis de inducción $(\Gamma_n, e_n, E_n, S_n)$ cumple los puntos 1 al 4 de la proposición.
- Veamos que $(\Gamma_{n+1}, e_{n+1}, E_{n+1}, S_{n+1})$ es consistente y cumple los puntos 2 al 4. Realizaremos análisis por casos sobre la regla aplicada.

  App1 $(\Gamma, e\,x, E \cup [x \mapsto p], S) \Rightarrow (\Gamma, e, E \cup [x \mapsto p], p : S)$
  Lo único que cambia es la expresión de control, y x se mantiene en ámbito en e.

  App2 $(\Gamma, \lambda y.e, E, p : S) \Rightarrow (\Gamma, e, E \cup [y \mapsto p], S)$
  El montón y la pila siguen manteniendo las propiedades y la y pasa a estar en ámbito en la expresión e a la vez que se añade al entorno. Como todas las variables ligadas de la expresión inicial son distintas, y no estaba previamente ligada en E.

  Var1 $(\Gamma \cup [p \mapsto (e', E')], x, E \cup [x \mapsto p], S) \Rightarrow (\Gamma, e', E', \#p : S)$
  Se coloca en el control una clausura consistente que satisface (3), luego, se mantiene la propiedad.

  Var2 $(\Gamma, \lambda y.e, E, \#p : S) \Rightarrow (\Gamma \cup [p \mapsto (\lambda y.e, E)], \lambda y.e, E, S)$
  Se añade al montón la clausura $(\lambda y.e, E)$ que es consistente y satisface (2).

  Var3 $(\Gamma, C_k\,\overline{x_i}, E, \#p : S) \Rightarrow (\Gamma[p \mapsto (C_k\,\overline{x_i}, E)], C_k\,\overline{x_i}, E, S)$
  Se añade al montón la clausura $(C_k\,\overline{x_i}, E)$ que es consistente y satisface (2).

  Letrec $(\Gamma, \textbf{letrec}\ \{\overline{x_i = e_i}\}\ \textbf{in}\ e, E, S) \Rightarrow (\Gamma \cup \overline{[p_i \mapsto (e_i, E')]}, e, E', S)$
  donde $E' = E \cup [\overline{x_i \mapsto p_i}]$. Los pares $\overline{(e_i, E')}$ y (e, E') son consistentes ya que las variables $\overline{x_i}$ se encuentran ahora en ámbito en $\overline{e_i}$ y en e y E contiene todas las variables en ámbito del **letrec** $\{\overline{x_i = e_i}\}$ **in** e. Por tanto, se añaden al montón clausuras consistentes que satisfacen (2) y se pone en el control un par (e, E') consistente que también satisface (2).

  Case1 $(\Gamma, \textbf{case}\ e\ \textbf{of}\ alts, E, S) \Rightarrow (\Gamma, e, E, (alts, E) : S)$
  Los pares (e, E) y $(alts, E)$ son consistentes y satisfacen (2), ya que tiene las mismas variables en ámbito que la expresión **case**.

  Case2 $(\Gamma, C_k\ \overline{x_i}, E \cup [\overline{x_i \mapsto p_i}], (alts, E') : S) \Rightarrow (\Gamma, e_k, E' \cup [\overline{y_{ki} \mapsto p_i}], S)$
  Veamos que $(e_k, E' \cup [\overline{y_{ki} \mapsto p_i}])$ es consistente.

(a) Por hipótesis $bv(alts) \cap fv(alts) = \emptyset$

$\Rightarrow$

$(bv(alts) - \overline{y_{ki}}) \cap (fv(alts) \cup \overline{y_{ki}}) = \emptyset$

$\Rightarrow \quad \{bv(e_k) \subseteq bv(alts) - \overline{y_{ki}} \ \wedge \ fv(e_k) \subseteq fv(alts) \cup \overline{y_{ki}}\}$

$bv(e_k) \cap fv(e_k) = \emptyset$

(b) $bv(e_k)$ no tiene repeticiones ya que e_k es una subexpresión que pertenece a $alts$.

(c) Veamos que $fv(e_k) \subseteq dom\ (E' \cup [\overline{y_{ki} \mapsto p_i}])$
$fv(e_k) \subseteq fv(alts) \cup \overline{y_{ki}} \subseteq dom\ (E' \cup [\overline{y_{ki} \mapsto p_i}])$

(d) Veamos que $bv(e_k) \cap dom\ (E' \cup [\overline{y_{ki} \mapsto p_i}]) = \emptyset$
$bv(e_k) \cap dom\ (E' \cup [\overline{y_{ki} \mapsto p_i}]) \subseteq (bv\ alts - \overline{y_{ki}}) \cap dom\ (E' \cup [\overline{y_{ki} \mapsto p_i}]) = \emptyset$

Además en los entornos correspondientes sólo se encuentran las variables en ámbito, ya que las variables $\overline{y_{ki}}$ están en ámbito en la expresión e_k.

□

Por lo tanto, como los entornos ligan solamente las variables del programa y no contienen dos veces la misma variable, los entornos no crecen de forma ilimitada. Un límite superior para cualquier entorno sería el número total de variables ligadas de la expresión de partida. Pero acabamos de demostrar que en realidad contienen un subconjunto de ellas: las variables en ámbito de la subexpresión correspondiente.

4.3 Índices de De Bruijn

En esta sección vamos a formalizar el paso de una máquina con variables de programa a la representación de dichas variables del programa mediante naturales, que serán los índices de De Bruijn [Bru72, Bru78]. Para más detalles sobre dichos índices consultar [NW98, BP99, ACCL90].

Posteriormente se demostrará que las máquinas *Mark-2* y *Mark-3* son equivalentes y, por tanto, la máquina *Mark-3* es correcta y completa con respecto a la semántica de Sestoft revisada (Figura 3.1).

4.3.1 Formalización del paso a expresiones De Bruijn

En primer lugar vamos a formalizar la transformación de una expresión con nombres de variables (ver Figura 2.1) a una en la que las variables se representen mediante los índices de De Bruijn (ver Figura 4.1) donde los valores u (enteros positivos) son los números de De Bruijn. Estas expresiones corresponden a la categoría $ExpB$.

En esta gramática los nombres de los constructores se siguen manteniendo. No obstante, en las alternativas de un **case** no es necesario tener las variables a las que se aplican los constructores, ya que dichas variables se almacenan en un orden fijo en el entorno. Otras posibilidades a esta representación se plantean en [Ses97], donde los constructores pasan a representarse con una etiqueta, junto con su aridad. Esto implica tener un lenguaje tipado y, para cada constructor sintáctico, añadir una ligadura global con tantas lambdas como la aridad del constructor. Así,

$$
\begin{array}{rcll}
e & \rightarrow & u & \text{-- índice de De Bruijn} \\
 & | & \lambda.e & \text{-- lambda abstracción} \\
 & | & e\ u & \text{-- aplicación} \\
 & | & \mathbf{letrec}\ \overline{e_i}\ \mathbf{in}\ e & \text{-- let recursivo} \\
 & | & C\ \overline{u_i} & \text{-- aplicación de constructor} \\
 & | & \mathbf{case}\ e\ \mathbf{of}\ \overline{C_i \rightarrow e_i} & \text{-- expresión case}
\end{array}
$$

Figura 4.1: λ-cálculo normalizado de Launchbury con índices de De Bruijn

para $Cons$ tendríamos una ligadura $Cons = \lambda.\lambda.C_{2,2}$ en el nivel principal. Sin embargo, nosotros estamos interesados en mostrar la transformación sin añadir restricciones de este tipo a nuestro lenguaje, debido a que el paso de una expresión con variables a una expresión con índices de De Bruijn no implica ninguna restricción sobre la expresión de partida.

Para definir el paso de una expresión con nombres de variables a una expresión con índices de De Bruijn, es necesario que la expresión sea una expresión cerrada, con todas sus variables ligadas con nombres distintos. Hace falta un entorno $\rho : Var \rightarrow Int$, así como las siguientes operaciones definidas sobre entornos:

- $\mapsto$: $Var \rightarrow Int \rightarrow (Var \rightarrow Int)$. Dado un nombre de variable x y un entero n genera un entorno unitario, donde la variable x está asociada al entero n.

- $+ : (Var \rightarrow Int) \rightarrow Int \rightarrow (Var \rightarrow Int)$. Dado un entorno y un entero n incrementa en n el valor asociado a cada variable del entorno.

- $\cup : (Var \rightarrow Int) \rightarrow (Var \rightarrow Int) \rightarrow (Var \rightarrow Int)$. Une dos entornos con dominios disjuntos.

Definición 22
Sea ρ, $\rho' : Var \rightarrow Int$, x, y variables y n un entero, definimos:

- $(x \mapsto n)\ z = \begin{cases} n & si\ z = x \\ \perp & si\ z \neq x \end{cases}$

- $(\rho + n)\ x = \rho\ x + n.$

- $(\rho \cup \rho')\ x = \begin{cases} \rho'\ x & si\ x \in dom\ \rho' \\ \rho\ x & si\ x \in dom\ \rho \end{cases}$

Definición 23
La transformación de una expresión cerrada e en el λ-cálculo normalizado de Launchbury a una expresión equivalente con índices de De Bruijn es la resultante de $bruijn\ [\![e]\!]\ []$, donde la función

bruijn : $Exp \to Env \to ExpB$ *se define:*

$$
\begin{array}{lcl}
bruijn\ [\![x]\!]\ \rho & = & \rho\ x \\
bruijn\ [\![\lambda\ y.e]\!]\ \rho & = & \lambda.bruijn\ [\![e]\!]\ (\rho+1) \cup [y \mapsto 1] \\
bruijn\ [\![e\ x]\!]\ \rho & = & (bruijn\ [\![e]\!]\ \rho)\ \ (bruijn\ [\![x]\!]\ \rho) \\
bruijn\ [\![\mathbf{letrec}\ \overline{x_i\ =\ e_i}^n\ \mathbf{in}\ e]\!]\ \rho & = & \mathbf{letrec}\ \overline{bruijn\ [\![e_i]\!]\ \rho'}\ \mathbf{in}\ bruijn\ [\![e]\!]\ \rho' \\
 & & donde\ \rho'\ =\ (\rho+n) \cup [\overline{x_i \mapsto i}] \\
bruijn\ [\![C\ \overline{x_i}]\!]\ \rho & = & C\ \overline{bruijn\ [\![x_i]\!]\ \rho} \\
bruijn\ [\![\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{y_{ij}}^{n_i} \to e_i}]\!]\ \rho & = & \mathbf{case}\ bruijn\ [\![e]\!]\ \rho\ \mathbf{of}\ \overline{C_i \to bruijn\ [\![e_i]\!]\ \rho'_i} \\
 & & donde\ \rho'_i\ =\ (\rho+n_i)[\overline{y_{ij} \mapsto n_i - j + 1}]
\end{array}
$$

Para una mejor comprensión veamos un ejemplo:

Ejemplo 24
La definición de la función *map* en una notación convencional (e.g. Haskell) es la siguiente:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

La función *map* expresada en nuestro lenguaje sería la siguiente:

$$
\begin{array}{l}
\mathbf{letrec} \\
\quad map = \lambda f.\lambda ys.\ \mathbf{case}\ ys\ \mathbf{of} \\
\qquad\qquad \begin{array}{lcl}
Nil & \to & Nil \\
Cons\ x\ xs & \to & \mathbf{letrec} \\
 & & \quad \begin{array}{l} fx\quad\ = f\ x \\ mfxs = (map\ \ f)\ xs \end{array} \\
 & & \mathbf{in} \\
 & & \quad Cons\ fx\ mfxs
\end{array} \\
\mathbf{in} \\
\quad map
\end{array}
$$

El cambio de variables en esta expresión por los correspondientes índices de De Bruijn produciría la siguiente expresión:

$$
\begin{array}{l}
\mathbf{letrec} \\
\quad \lambda.\lambda.\mathbf{case}\ 1\ \mathbf{of} \\
\qquad\qquad \begin{array}{lcl}
Nil & \to & Nil \\
Cons & \to & \mathbf{letrec} \\
 & & \quad \begin{array}{l} 6\ 4 \\ (7\ 6)\ 3 \end{array} \\
 & & \mathbf{in} \\
 & & \quad Cons\ 1\ 2
\end{array} \\
\mathbf{in} \\
\quad 1
\end{array}
$$

Obsérvese que a la aparición de definición de la variable *map* en el **letrec** le hemos dado el valor 1 mientras que a la aparición de uso de *map* en la expresión $(map\ f)\ xs$ le hemos dado el valor 7, ya que entre la ligadura inicial del *map* y la aparición de uso existen 6 variables ligadas: $\{f, ys, x, xs, mfxs, fx\}$.

□

Los constructores de las alternativas tienen numeradas las variables en orden decreciente. Esto es debido a que si tenemos el patrón $C\ x_1 \ldots x_n$, el primer puntero que añadimos al entorno es el correspondiente a la variable x_1, posteriormente el de la x_2, y así hasta el de la x_n. Por tanto, la dirección relativa a la variable x_i queda en el entorno en la posición $n - i + 1$.

A continuación se expondrá una formalización equivalente a la anterior para la traducción de expresiones con nombres de variables a expresiones con índices de De Bruijn. Pero esta vez el entorno será más simple y se representará por una lista de variables del programa $\rho = [Var]$. La posición, *pos*, de las variables en ese entorno es el índice de De Bruijn que le corresponde a la variable.

```
pos [] y = error
pos (x:xs) y
      | y == x    = 1
      | otherwise = 1 + pos xs y
```

Si partimos de una expresión cerrada, nunca se producirá un error en la transformación de esa expresión a una expresión De Bruijn. Lo demostraremos cuando estudiemos la corrección de la máquina *Mark-2* con respecto a la máquina *Mark-3* (ver pág. 52), ya que para ello relacionaremos los entornos de las máquinas con los entornos que surgen en la computación de los índices de De Bruijn.

La citada formalización es:

$$
\begin{array}{lcl}
bruijn\ [\![x]\!]\ \rho & = & pos\ \rho\ x \\
bruijn\ [\![\lambda\ y.e]\!]\ \rho & = & \lambda.bruijn\ [\![e]\!]\ (y : \rho) \\
bruijn\ [\![e\ x]\!]\ \rho & = & (bruijn\ [\![e]\!]\ \rho)\ \ (bruijn\ [\![x]\!]\ \rho) \\
bruijn\ [\![\mathbf{letrec}\ \overline{x_i\ =\ e_i}\ \mathbf{in\ e}]\!]\ \rho & = & \mathbf{letrec}\ \overline{bruijn\ [\![e_i]\!]\ \rho'}\ \mathbf{in}\ bruijn\ [\![e]\!]\ \rho' \\
 & & donde\ \rho' = \overline{x_i :}\rho \\
bruijn\ [\![C\ \overline{x_i}]\!]\ \rho & = & C\ \overline{bruijn\ [\![x_i]\!]\ \rho} \\
bruijn\ [\![\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{y_{ij}}^{n_i} \rightarrow e_i}]\!]\ \rho & = & \mathbf{case}\ bruijn\ [\![e]\!]\ \rho\ \mathbf{of}\ \overline{C_i\ \rightarrow bruijn\ [\![e_i]\!]\ \rho'_i} \\
 & & donde\ \rho'_i = y_{in_i} : \ldots : y_{i1} : \rho
\end{array}
$$

Donde (:) representa añadir por la izquierda un elemento a una lista, y asocia por la derecha. Dada una expresión cerrada e, la expresión resultante de transformar sus variables por índices de De Bruijn es la que resulta de $bruijn\ [\![e]\!]\ [\,]$.

Este cálculo es completamente equivalente al que hemos visto antes. En lugar de almacenar en el entorno el índice de De Bruijn, lo que hacemos es almacenar las variables en una lista y consideramos la posición de las variables en dicha lista en lugar de consultar el valor de su índice.

En la primera formulación, cuando se añadían n variables al entorno, se sumaba n a todos los índices del mismo. En la actual se añaden por la izquierda n variables a la lista, con lo que las posiciones de las restantes quedan incrementadas en n. El orden en que se añaden las variables coincide con el orden de numeración de la primera definición.

La transformación de una expresión a expresión De Bruijn nos va a definir para una expresión dada e la siguiente relación inductivamente generada $R^e \subseteq Exp \times Var^* \times \mathbb{N}^*$. La expresión de partida e ha de ser una expresión cerrada.

$$\frac{}{(e, [\,], [\,]) \in R^e} \quad \text{Inic}$$

$$\frac{(\lambda x.e', \rho, ns) \in R^e}{(e', x : \rho, 1 : ns) \in R^e} \quad \text{Lambda}$$

$$\frac{(e'\,x, \rho, ns) \in R^e}{(e', \rho, 1 : ns) \in R^e \quad , \quad (x, \rho, 2 : ns) \in R^e} \quad \text{Aplic}$$

$$\frac{(\textbf{letrec } \{\overline{x_i = e_i}\} \textbf{ in } e', \rho, ns) \in R^e}{\forall i.(e_i, \overline{x_i} : \rho, i : ns) \in R^e \quad , \quad (e', \overline{x_i} : \rho, 0 : ns) \in R^e} \quad \text{Letrec}$$

$$\frac{(C\,\overline{x_i}, \rho, ns) \in R^e}{\forall i.(x_i, \rho, i : ns) \in R^e} \quad \text{Constructor}$$

$$\frac{(\textbf{case } e' \textbf{ of } \overline{C_i\,\overline{y_{ij}}^{n_i} \to e_i}, \rho, ns) \in R^e}{\begin{array}{l}(e', \rho, 0 : ns) \in R^e \quad , \quad \forall i.(e_i, \rho_i, i : ns) \in R^e \quad , \quad (\overline{C_i\,\overline{y_{ij}}^{n_i} \to e_i}, \rho, n+1 : ns) \in R^e \\ \textit{donde } \rho_i = y_{in_i} : \ldots : y_{i1} : \rho\end{array}} \quad \text{Case}$$

donde ns representa el camino que nos lleva a la subexpresión concreta.

Esta relación es necesaria para la definición del entorno de De Bruijn correspondiente a una subexpresión concreta de la expresión de partida inicial (Definición 25).

Vamos a utilizar la notación $e' = e\,|_{ns}$ para indicar que e' es la subexpresión de e definida por el camino ns. Esta subexpresión es única ya que los caminos son únicos. Dado un camino valido (es decir, un camino que lleve a una subexpresión de la expresión e) ns en e, es obvio que existe una sola tripla $(e', \rho, ns) \in R^e$, donde $e' = e\,|_{ns}$, por lo que escribiremos $R^e(ns)$ para designar el par (e', ρ).

Definición 25
Sea e una expresión cerrada, y R^e la relación definida por e. Dado un camino válido ns en e, y $(e', \rho) = R^e(ns)$, diremos que ρ es el entorno de De Bruijn asociado a e' en la expresión e.

El entorno de De Bruijn asociado a una subexpresión $e' = e|_{ns}$ de una expresión inicial cerrada e es único, ya que ns es único. Además, para toda subexpresión $e' = e|_{ns}$ de e existe un entorno ρ tal que $(e', \rho, ns) \in R^e$.

Por otro lado, el entorno de De Bruijn de una subexpresión e' es el entorno ρ que se genera al ir transformando la expresión inicial e a una expresión de De Bruijn. De hecho, al ir a transformar la subexpresión e' pasamos por el estado *bruijn* $[\![e']\!]\,\rho$. La demostración de esta propiedad es obvia y surge de las propias definiciones de la relación R^e y de la definición de la transformación *bruijn*, la relación existente es uno a uno entre las reglas de la transformación *bruijn* y las correspondientes de la relación R^e.

4.4 Corrección *Mark-2/Mark-3*

A continuación vamos a demostrar que las máquinas *Mark-2* y *Mark-3* realizan los mismos cálculos sobre una expresión de partida dada. Es decir, que la *Mark-3* es correcta y completa con respecto a la *Mark-2*. Para ello vamos a definir una máquina *Mark-23* (Figura 4.2) que es muy parecida a la *Mark-2* y a la *Mark-3*, con la única diferencia de que el entorno de la *Mark-2* va a pasar a ser una lista en lugar de un conjunto de ligaduras, y con respecto a la *Mark-3* en los entornos va a mantener los nombres de las variables. Esto no crea ningún problema, ya que en el entorno nunca se va a ligar una variable más de una vez, como quedó demostrado en la Sección 4.2.1. Por lo tanto, la representación de dicho entorno con una lista o un conjunto es indiferente. Ahora vamos a ir añadiendo variables en un orden establecido para que las posiciones coincidan con los índices de De Bruijn que les corresponden y, por tanto, la nueva máquina *Mark-23* sea equivalente también a la *Mark-3*. Este orden concreto va a ser el orden cronológico en que van eliminándose las ligaduras. En el caso de que pasen a estar libres varias variables a la vez, elegimos el orden en que se incorporan al entorno para que corresponda con el que se utiliza para los índices de De Bruijn.

Hemos utilizado la notación $E[x \mapsto p]$ para resaltar un vínculo existente en el entorno, y la notación $(x,p) : E$ para indicar que añadimos al entorno E el vínculo $x \mapsto p$.

Hemos de tener en cuenta que todo lo que se almacena en el montón son subexpresiones de la expresión de partida inicial, y poseen un camino único ns para llegar a ellas.

Definición 26
Dado un entorno $E : Var \to Puntero$. Definimos:

- *El dominio de E es $dom\ E = [x|\ (x,p) \in E]$. Si el entorno E está representado por una lista, las variables en $dom\ E$ han de mantener el mismo orden.*
- *El rango de E es $range\ E = [p|\ (x,p) \in E]$. Si el entorno E está representado por una lista, los punteros en $range\ E$ han de mantener el mismo orden.*

En la Figura 4.2 se define un máquina *Mark-23*, cuyas configuraciones (Γ, e, E, S) son las correspondientes a la máquina *Mark-2*, donde los entornos han pasado a representarse con una lista en lugar de un conjunto de ligaduras, y mantienen un orden concreto. La expresión inicial de partida sigue siendo una expresión cerrada del lenguaje de Launchbury.

Definición 27
Una configuración $(\Gamma, e_0|_{ns}, E, S)$ de Mark-23 es De Bruijn-segura con respecto a una expresión de partida inicial e_0 cerrada, con todas sus variables ligadas con nombres distintos si:

	Montón	Control	Entorno	Pila	Regla
	Γ	$(e\ x)$	$E[x \mapsto p]$	S	app1
$\Longrightarrow$	Γ	e	$E[x \mapsto p]$	$p : S$	
	Γ	$\lambda y.e$	E	$p : S$	app2
$\Longrightarrow$	Γ	e	$(y, p) : E$	S	
	$\Gamma \cup [p \mapsto (e', E')]$	x	$E[x \mapsto p]$	S	var1
$\Longrightarrow$	Γ	e'	E'	$\#p : S$	
	Γ	$\lambda y.e$	E	$\#p : S$	var2
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda y.e, E)]$	$\lambda y.e$	E	S	
	Γ	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	E	S	letrec (*)
$\Longrightarrow$	$\Gamma \cup \overline{[p_i \mapsto (e_i, E')]}$	e	E'	S	
	Γ	**case** e **of** $alts$	E	S	case1
$\Longrightarrow$	Γ	e	E	$(alts, E) : S$	
	Γ	$C_k\ \overline{x_i}^{n_k}$	$E[\overline{x_i \mapsto p_i}^{n_k}]$	$(alts, E') : S$	case2 (**)
$\Longrightarrow$	Γ	e_k	E''	S	
	Γ	$C_k\ \overline{x_i}$	E	$\#p : S$	var3
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C_k\ \overline{x_i}, E)]$	$C_k\ \overline{x_i}$	E	S	

($*$) En la regla *let*, las variables $\overline{p_i}$ deben ser distintas y frescas: No pueden ocurrir ni en Γ, ni en **letrec** $\{\overline{x_i = e_i}\}$ **in** e, ni en S. El nuevo entorno E' es $\overline{(x_i, p_i)} : E$.

($**$) La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{kj}}^{n_k} \to e_k$ en *alts*. $E'' = (y_{k\,n_k}, p_{n_k}) : \ldots : (y_{k1}, p_1) : E'$

Figura 4.2: Máquina abstracta *Mark-23*

1. $\forall (e_0\,|_{ns'}, E') \in range\,\Gamma$, *dom* E' *coincide con el entorno de De Bruijn de* $e_0\,|_{ns'}$ *con respecto a* e_0.

2. *dom* E *coincide con el entorno de De Bruijn de* $e_0\,|_{ns}$ *con respecto a* e_0.

3. $\forall (alts, E') \in S$, *siendo* $alts = e_0\,|_{ns}$, *dom* E' *coincide con el entorno de De Bruijn de alts con respecto a* e_0.

Esta definición implica que en la configuración (Γ, e, E, S) todas las clausuras almacenadas en el montón, en la pila o la que se encuentra en el control cumplen que la tripla $(e, dom(E), ns) \in R^{e_0}$, donde ns es la ruta necesaria para llegar a la subexpresión concreta e, es decir $e = e_0|_{ns}$.

Si observamos las reglas de la máquina *Mark-23* es fácil comprobar que tienen el mismo comportamiento que la máquina *Mark-2*. Se añaden las mismas ligaduras en las mismas reglas y su funcionamiento es idéntico a la *Mark-2*. Cada regla de esta máquina se corresponde fielmente con la de la *Mark-2*, con la única salvedad de que el entorno se representa por una lista, pero es el mismo, en el sentido de que contiene los mismos vínculos, ya que quedó demostrado (Proposición

21) que si la expresión de partida era cerrada y con sus variables ligadas con distintos nombres, no se añadían al entorno dos punteros para la misma variable. Por tanto, al resaltar el vínculo $E[x \mapsto p]$ no estamos haciendo ningún tipo de elección en el entorno, solamente estamos resaltando la única ligadura del tipo $(x, \_)$ que existe en él.

Se puede prescindir de la exigencia de que todas las variables ligadas de la expresión de partida sean distintas y permitir que existieran variables ligadas con el mismo nombre y la demostración no daría problemas. En ese caso la notación $E[x \mapsto p]$ significaría quedarnos con el primer par del tipo $(x, \_)$ que existe en el entorno E. Como es una lista, no se pierde el orden en el entorno y, por lo tanto, el último que se vinculó es el que corresponde actualmente a la variable libre. Esto en la semántica operacional de la *Mark-2* implicaría que cuando añadimos una ligadura para la variable x se tiene que actualizar en el entorno con la nueva dirección.

Con la siguiente proposición quedará demostrado que las configuraciones de la máquina *Mark-23* conservan la propiedad de ser De Bruijn-seguras, y quedará también demostrado que nunca llegaremos a encontrarnos con un error al transformar un expresión a una expresión de De Bruijn, ya que en la *Mark-2* habíamos demostrado que en los entornos E de cada clausura (e, E) están las variables alcanzables desde la expresión e.

Proposición 28
Dada una expresión cerrada e con todas sus variables ligadas distintas, todas las configuraciones alcanzables por la Mark-23 a partir de e son De Bruijn-seguras respecto a e.

Demostración:
Por inducción sobre la longitud de la derivación.

Sea e_0 la expresión de partida (expresión cerrada con todos los nombres de sus variables ligadas distintos).

- La configuración inicial $(\{\,\}, e_0, [\,], \{\,\})$, es *De Bruijn-segura* con respecto a e_0, ya que el entorno de De Bruijn con respecto a e_0 asociado a e_0 es $[\,]$.

- Sea $(\Gamma, e, E, S) \implies (\Gamma', e', E', S')$
Si (Γ, e, E, S) es *De Bruijn-segura* con respecto a e_0 entonces (Γ', e', E', S') es *De Bruijn-segura* con respecto a e_0. Lo veremos por casos sobre la regla aplicada.

$$\begin{array}{lllll} \underline{app1} & \Gamma & (e\ x) & E[x \mapsto p] & S \\ \implies & \Gamma & e & E[x \mapsto p] & p : S \end{array}$$

Es trivial, ya que por hipótesis de inducción el entorno $\rho = dom\ E$ es el entorno de De Bruijn asociado a $(e\ x)$ con respecto a e_0 y por definición de R^{e_0}, ρ es el entorno de De Bruijn asociado a e con respecto a e_0. Es más, $pos\ x\ E\ =\ bruijn[\![x]\!]\ \rho$. Por tanto, la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{lllll} \underline{app2} & \Gamma & \lambda y.e & E & p : S \\ \implies & \Gamma & e & (y,p) : E & S \end{array}$$

Por hipótesis tenemos que $dom\, E$ es el entorno de De Bruijn de $\lambda y.e$ con respecto a e_0, por lo tanto $(y : dom\, E)$ es el entorno de De Bruijn de e con respecto a e_0,luego la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{lllll} \underline{var1} & \Gamma \cup [p \mapsto (e', E')] & x & E[x \mapsto p] & S \\ \Longrightarrow & \Gamma & e' & E' & \#p : S \end{array}$$

Γ mantiene la propiedad 1 de la definición 27, y por hipótesis de inducción tenemos que E' es el entorno de De Bruijn de e' con respecto a e_0, ya que $(e', E') \in range(\Gamma \cup [p \mapsto (e', E')])$, luego la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{lllll} \underline{var2} & \Gamma & \lambda y.e & E & \#p : S \\ \Longrightarrow & \Gamma \cup [p \mapsto (\lambda y.e, E)] & \lambda y.e & E & S \end{array}$$

Por hipótesis de inducción E es el entorno de De Bruijn de $\lambda y.e$ con respecto a e_0, luego la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{llll} \underline{letrec} & \Gamma & \mathbf{letrec}\{\overline{x_i = e_i}\} \ \mathbf{in} \ e & E \quad S \\ \Longrightarrow & \Gamma \cup \overline{[p_i \mapsto (e_i, E')]} & e & E' \quad S \\ & \text{donde } E' = \overline{(x_i = p_i)} : E & & \end{array}$$

dom E' es el entorno de De Bruijn de e_i y de e con respecto a e_0, luego la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{lllll} \underline{case1} & \Gamma & \mathbf{case} \ e \ \mathbf{of} \ alts & E & S \\ \Longrightarrow & \Gamma & e & E & (alts, E) : S \end{array}$$

dom E es el entorno de De Bruijn de e y de $alts$ con respecto a e_0, luego la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{lllll} \underline{case2} & \Gamma & C_k \ \overline{x_i}^{n_k} & E[\overline{x_i \mapsto p_i}^{n_k}] & (alts, E') : S \\ \Longrightarrow & \Gamma & e_k & E'' & S \\ & \multicolumn{4}{l}{\text{donde } E'' = (y_{k\,n_k}, p_{n_k}) : \ldots : (y_{k1}, p_1) : E'} \end{array}$$

Por hipótesis de inducción E' es el entorno de De Bruijn de $alts$ con respecto a e_0, por lo tanto, E'' es el entorno de De Bruijn de e_k con respecto a e_0, luego la configuración es *De Bruijn-segura* con respecto a e_0.

$$\begin{array}{lllll} \underline{var3} & \Gamma & C \ \overline{x_i} & E & \#p : S \\ \Longrightarrow & \Gamma \cup [p \mapsto (C \ \overline{x_i}, E)] & C \ \overline{x_i} & E & S \end{array}$$

Por hipótesis de inducción E es el entorno de De Bruijn de $C \ \overline{x_i}$ con respecto a e_0, luego la configuración es *De Bruijn-segura* con respecto a e_0.

□

Por otro lado, también es fácil ver que la máquina *Mark-23* es equivalente a la máquina *Mark-3*. Las únicas diferencias entre ambas máquinas son:

1. En los entornos correspondientes a la máquina *Mark-23* tenemos los punteros asociados con los nombres de las variables. Si nos quedamos solamente con los punteros, el entorno es el mismo de la *Mark-3*, y en ambos entornos las variables mantienen el mismo orden.

2. En la *Mark-3* las expresiones son las correspondientes con los índices de De Bruijn.

Como ya hemos demostrado que todas las configuraciones alcanzables por la máquina *Mark-23* son *De Bruijn seguras* con respecto a la expresión inicial e, es seguro cambiar las expresiones de control de dicha máquina por las expresiones de De Bruijn correspondientes y, por tanto, la máquina que resulta de estas modificaciones es exactamente la máquina *Mark-3*.

Capítulo 5

Corrección de la máquina *STG*

En este capítulo pretendemos demostrar que la máquina *STG* es correcta y completa con respecto a la semántica operacional de Sestoft revisada (Sección 3.1.2). Para ello la compararemos con la máquina de Sestoft *Mark-2* de la cual ya tenemos demostrada la corrección con respecto a la semántica operacional.

Las diferencias en el comportamiento de ambas máquinas son grandes, ya que, por ejemplo, la máquina *STG* hace una aplicación total de las λ-abstracciones a todos sus argumentos, mientras que la máquina *Mark-2* aplica las lambdas a sus argumentos de uno en uno. Además, la máquina *Mark-2* no tiene en cuenta que a veces no son necesarias las actualizaciones, y actualiza siempre. Para relacionar las dos máquinas lo haremos a partir de un lenguaje común, y una máquina *STG-1S* que corresponderá fielmente a la máquina *STG* con la única salvedad de que en lugar de tener tres pilas tendrá una única pila. Veremos que la máquina *STG-1S* es equivalente a la máquina *Mark-2*.

Tras demostrar que la máquina *STG-1S* es correcta y completa con respecto a la semántica de Launchbury y, por transitividad, a la semántica denotacional, demostraremos que la división de la pila en tres pilas puede llevar a reducciones erróneas. De aquí deduciremos que la máquina *STG* no es del todo correcta con respecto a la semántica operacional. En concreto se verá que existen expresiones mal tipadas que pueden llegar a ser reducidas por la máquina y, sin embargo, no serían capaces de alcanzar una forma normal en la semántica.

A partir de aquí realizaremos la modificación necesaria en la máquina *STG* con tres pilas de manera que mantenga la corrección. No obstante, y debido a que el nuevo *RTS*[1] de *GHC* sólo utiliza una única pila, y a que la *STG* con una única pila es equivalente a la de tres para toda expresión bien tipada, a partir de este capítulo nos quedaremos con la máquina *STG-1S*, de manera que la generación de código del Capítulo 6 se derivará directamente sobre la máquina con una única pila.

5.1 El lenguaje común

Para comparar las máquinas *Mark-2* y *STG*, primeramente debemos definir un λ-cálculo común a ambas máquinas. Este lenguaje se presenta en la Figura 5.1 y le llamaremos FUN.

[1] Run time system.

$$
\begin{array}{lcll}
e & \rightarrow & x\ \overline{x_i}^n & \text{-- } n \geq 0, \text{ aplicación/variable} \\
 & | & \textbf{letrec}\ \overline{bind_i}\ \textbf{in}\ e & \text{-- let recursivo} \\
 & | & C\ \overline{x_i} & \text{-- aplicación de constructor} \\
 & | & \textbf{case}\ e\ \textbf{of}\ \overline{alt_i}\ |^t & \text{-- expresión case} \\
bind & \rightarrow & x = lf\ |^t & \\
lf & \rightarrow & \lambda\ \overline{x_i}^n.e & \text{-- } n \geq 0, \text{ lambda abstracción} \\
alt & \rightarrow & C\ \overline{x_j} \rightarrow e &
\end{array}
$$

Figura 5.1: Definición de FUN

Es equivalente al lenguaje STG (STGL) pero expresado en un λ-cálculo más tradicional. Las principales diferencias con respecto al lenguaje STGL son las siguientes:

1. En FUN no existen literales (valores desencapsulados), ni operadores sobre valores primitivos, ni expresiones **case** sobre valores primitivos. Estos han sido excluidos de nuestro estudio.

2. En FUN no se ha añadido al **case** la alternativa por defecto. Esta podría añadirse sin esfuerzo.

3. En las lambda abstracciones de STGL existe una marca $\backslash\pi$ para indicar que algunas actualizaciones pueden eliminarse de forma segura. Esto es sólo un problema de eficiencia. No obstante, modificaremos la máquina *Mark-2* para suprimir actualizaciones en los casos obvios de ligaduras a formas normales.

4. En STGL existe un **let** no recursivo. Es obvio que esto puede simularse en nuestro lenguaje FUN con un **letrec**.

5. En STGL, un programa es una lista de ligaduras con una variable distinguida *main*, por la que comienza la evaluación. Esto puede simularse con la expresión **letrec** $\overline{bind_i}$ **in** *main*.

Los puntos 3, 4 y 5 son diferencias no esenciales y pueden ser simuladas por FUN.

Comparado con el λ-cálculo original de la máquina *Mark-2* (ver Figura 2.1) es evidente que nuestro lenguaje FUN es un subconjunto de él, con las siguientes restricciones:

- Las lambda abstracciones sólo pueden aparecer en las ligaduras de los **letrec**.

- Las aplicaciones son de variable a cero o más variables. La aplicación $x\ \overline{x_i}^n$ en la máquina *Mark-2* se entiende como $(\ldots((x\ x_1)\ x_2)\ldots)\ x_n$.

La notación $\lambda\ \overline{x_i}^n.e$ es una abreviatura de $\lambda x_n.\cdots.\lambda x_1.e$, donde los argumentos han sido numerados en orden decreciente por conveniencia. De esta forma, es fácilmente aceptable que $\lambda\ \overline{x_i}^{n-1}.e$ significa $\lambda x_{n-1}.\cdots.\lambda x_1.e$. Por otro lado, $x\ \overline{x_i}^n$ significa $x\ x_1\ \ldots\ x_n$ y, por tanto, $x\ \overline{x_i}^{n-1}$ significará $x\ x_1\ \ldots\ x_{n-1}$. Cuando $n = 0$ escribiremos e en lugar de $\lambda\ \overline{x_i}^n.e$ y x en lugar de $x\ \overline{x_i}^n$.

Normalmente, las máquinas abstractas no realizan sustituciones explícitas de los nombres de las variables como lo hace la semántica. Por ello es necesario tener asociado a cada expresión un entorno que liga sus variables en ámbito con las direcciones en el montón en las que se encuentran dichas clausuras. Esto conlleva, normalmente, tener entornos más grandes de lo estrictamente necesario, ya que en ellos hay ligadas variables que pueden no estar libres en la expresión asociada. Esto sucede si tenemos, por ejemplo, $\lambda x.e$ y la variable x no aparece en el cuerpo e de la lambda. El entorno asociado a la expresión e ligaría la variable x con su dirección en el montón. Sin embargo, esto no sucede en las reglas semánticas ya que al realizar la sustitución explícita $e[p/x]$, si la x no aparece en e no se realiza ningún cambio en la expresión. Lo ideal sería tener en los entornos sólo las variables libres de la expresión asociada. Aquí surge un compromiso en tiempo de ejecución entre tiempo y espacio, ya que si mantenemos los entornos minimales esto puede acarrear un coste importante en cuestión de tiempo, y si no nos preocupamos de ello, éstos pueden llegar a consumir un gran espacio ligando clausuras que no son necesarias e impidiendo que sea recuperado su espacio por el recolector de basura.

Para acercarnos al entorno ideal hemos añadido al lenguaje FUN los *conjuntos de poda*. La notación $lf\ |^t$ significa que a las lambda abstracciones se les anota en tiempo de compilación con el conjunto t de sus variables libres. Este conjunto t fue llamado *trimmer* en [Ses97]. Será utilizado al construir una clausura en el montón para una lambda abstracción. El entorno almacenado en la clausura sólo ligará las variables contenidas en el conjunto de poda. Esto implica una pequeña penalización en términos de tiempo de ejecución, pero por otro lado conlleva un gran ahorro de espacio. Análogamente, la notación $\overline{alt_i}\ |^t$ tiene el mismo significado: t es el conjunto de variables libres, anotadas en tiempo de compilación, del conjunto de alternativas de la expresión **case**. Cuando se colocan las continuaciones en la pila, el entorno asociado con ellas se poda de acuerdo a t, de tal forma que en la pila tenemos también los entornos sólo con las variables libres. Ambas optimizaciones se realizan en la máquina *STG*, aunque la segunda no se refleja en las reglas de la semántica operacional dadas en [Pey92].

De manera que, tanto en el montón como en la pila, mantenemos los entornos minimales, pero el entorno asociado a la expresión de control no se mantiene minimal, ya que esto conllevaría un notable coste en tiempo de ejecución para un ahorro de espacio poco notable.

Este lenguaje FUN no pierde ninguna potencia expresiva con respecto al λ-cálculo enriquecido de la máquina *Mark-2*, ya que cualquier expresión de aquel λ-cálculo puede ser transformada de modo automático a una expresión en nuestro lenguaje FUN, para lo cual sólo hay que añadir ligaduras cuando sea necesario. En el caso de la aplicación lo que haríamos sería añadir una ligadura para la expresión funcional en caso de que ésta no fuera una variable. Este proceso de normalización es equivalente al necesario para pasar del lenguaje núcleo de Haskell llamado Core [PHH$^+$93] al STGL.

5.2 La máquina *Mark-2* para FUN

En esta sección aprovecharemos para hacer algunas optimizaciones de la máquina *Mark-2* manteniendo la corrección con respecto a la original. De momento, como el lenguaje propuesto FUN es un subconjunto del lenguaje que admite la máquina *Mark-2*, las modificaciones necesarias en la máquina son nulas. No obstante, realizaremos una optimización para evitar las actualizaciones innecesarias.

En la Figura 5.2 se muestran las reglas para la máquina *Mark-2* sobre expresiones del lenguaje FUN. Nótese que la expresión de control es, en general, una lambda forma lf. En particular, podría ser una expresión e cuaquiera, si $lf = \lambda \overline{x_i}^0.e$. Además, suponemos que todas las apariciones de los superíndices n cumplen $n > 0$. Es importante darse cuenta de que, en principio, no existen diferencias entre esta máquina y la máquina *Mark-2* original. Solamente se han aplicado las reglas a una restricción del lenguaje original. Existen además algunas optimizaciones que no afectan al comportamiento original. Éstas son:

- La regla original *var1* ha sido dividida en tres: la primera, correspondiente a la regla original *var1*, ahora se llama *var1c*; las otras dos reglas son casos particulares de la original en la que las expresiones referenciadas por los punteros p están en forma normal. La máquina original *Mark-2* ejecutaría en secuencia la regla *var1* seguida por la regla *var2*, o la regla *var1* seguida por la regla *var3*. Estas secuencias han sido absorbidas respectivamente por las reglas *var1a* y *var1b*.
- Los conjuntos de poda han sido añadidos a las lambda formas y a las continuaciones. Cuando se crean clausuras nuevas en el montón en las reglas *letrec* y *var3*, y cuando las continuaciones se almacenan en la pila (lo que sucede en la regla *case1*) los entornos se podan con el conjunto de variables libres asociado a la expresion. Esta modificación sólo afecta al conjunto de clausuras vivas en el montón, que ahora es más pequeño.

En la máquina de la Figura 5.2 se ha añadido una nueva columna *Última*. El propósito de esta columna es recordar la última regla aplicada por la máquina. Este nuevo campo es importante para definir las configuraciones *estables*. Éstas serán necesarias para comparar la evolución de las máquinas *Mark-2* y *STG-1S* (véase la Sección 5.4).

5.2.1 Configuraciones estables para la máquina *Mark-2*

Para comparar las máquinas *Mark-2* y *STG-1S*, que posteriormente presentaremos, es necesario establecer unos puntos de comparación de los cómputos. Para ello definiremos en ambas máquinas estos puntos, que llamaremos *estables*.

Definición 29
Una configuración (Γ, lf, E, S, l) *de la máquina Mark-2 es* estable *si se cumple una de las siguientes condiciones:*

1. $lf = e \;\wedge\; l \notin \{app1, var3\}$
2. $S = [\,] \;\wedge\; lf = C\ \overline{x_i}$
3. $S = [\,] \;\wedge\; lf = \lambda \overline{x_i}^n.e \;\wedge\; n > 0$

En la máquina *STG* las lambda abstracciones nunca aparecen en las expresiones de control, así que parece natural de las configuraciones estables de la *Mark-2* excluyamos aquellas que poseen lambda abstracciones en las expresiones de control. Si la última regla aplicada es *app1*,

	Montón	Control	Entorno	Pila	Última	Regla
	Γ	$x\ \overline{x_i}^n$	$E \cup [x_n \mapsto p]$	S	l	app1
$\Longrightarrow$	Γ	$x\ \overline{x_i}^{n-1}$	$E \cup [x_n \mapsto p]$	$p : S$	app1	
	Γ	$\lambda\overline{x_i}^n.e$	E	$p : S$	l	app2
$\Longrightarrow$	Γ	$\lambda\overline{x_i}^{n-1}.e$	$E \cup [x_n \mapsto p]$	S	app2	
	$\Gamma \cup [p \mapsto (\lambda\overline{x_i}^n.e', E')]$	x	$E \cup [x \mapsto p]$	S	l	var1a
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda\overline{x_i}^n.e', E')]$	$\lambda\overline{x_i}^n.e'$	E'	S	var1a	
	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E')]$	x	$E \cup [x \mapsto p]$	S	l	var1b
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E')]$	$C\ \overline{x_i}$	E'	S	var1b	
	$\Gamma \cup [p \mapsto (e', E')]$	x	$E \cup [x \mapsto p]$	S	l	var1c $(*)$
$\Longrightarrow$	Γ	e'	E'	$\#p : S$	var1c	
	Γ	$\lambda\overline{x_i}^n.e$	E	$\#p : S$	l	var2
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda\overline{x_i}^n.e, E)]$	$\lambda\overline{x_i}^n.e$	E	S	var2	
	Γ	$\textbf{letrec}\ \overline{x_i = \mathit{lf}_i\ \vert^{t_i}}\ \textbf{in}\ e$	E	S	l	letrec $(**)$
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto (\mathit{lf}_i, E'\ \vert^{t_i})}]$	e	E'	S	letrec	
	Γ	$\textbf{case}\ e\ \textbf{of}\ alts\ \vert^t$	E	S	l	case1
$\Longrightarrow$	Γ	e	E	$(alts, E\ \vert^t) : S$	case1	
	Γ	$C_k\ \overline{x_i}$	$E \cup [\overline{x_i \mapsto p_i}]$	$(alts, E') : S$	l	case2 $(***)$
$\Longrightarrow$	Γ	e_k	$E' \cup [\overline{y_{ki} \mapsto p_i}]$	S	case2	
	Γ	$C_k\ \overline{x_i}$	E	$\#p : S$	l	var3
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C_k\ \overline{x_i}, E\ \vert^{\{\overline{x_i}\}})]$	$C_k\ \overline{x_i}$	E	S	var3	

$(*)$ $e' \neq C\ \overline{x_i}$ y $e' \neq \lambda\overline{x_i}^n.e$

$(**)$ En la regla *letrec*, las variables $\overline{p_i}$ son distintas y frescas con respecto a Γ, $\textbf{letrec}\ \overline{x_i = \mathit{lf}_i\ \vert^{t_i}}\ \textbf{in}\ e$, y S. El entorno E' se define como $E \cup [\overline{x_i \mapsto p_i}]$

$(***)$ La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*

Figura 5.2: Máquina abstracta *Mark-2* para FUN entonces la máquina *Mark-2* está aún colocando argumentos en la pila y aún no se ha puesto a evaluar la variable x correspondiente a la función aplicada. En la máquina *STG* todos estos pasos intermedios se realizan en una sola regla y, por tanto, no existen, así que también los hemos excluido. Por último, si la última regla aplicada es *var3*, entonces la máquina probablemente esté realizando aún actualizaciones, y en cualquier caso aún no se ha realizado el encaje de patrones. Como queremos comparar configuraciones en las que una expresión real de FUN aparezca en el control, todos estos estados deben ser considerados como estados ”internos”. La segunda posibilidad de la definición corresponde a estados finales de cómputo exitoso.

Definición 30
Sean m y m' configuraciones estables de la máquina Mark-2 para FUN. Diremos que m evoluciona *a m' y lo denotaremos por $m \Rightarrow_e^+ m'$ si*

1. $m \Rightarrow^* m'$
2. *existe al menos una transición*
3. *no existe ninguna otra configuración estable entre ambas configuraciones m y m'.*

5.3 La máquina *STG-1S*

En esta sección definiremos una máquina abstracta muy cercana a la máquina *STG* [Pey92] y veremos que es correcta y completa con respecto a la máquina *Mark-2* de la Figura 5.2. La llamaremos *STG-1S* ya que la principal diferencia con respecto a la máquina *STG* original es que tiene sólo una pila en lugar de tres. La pila de la máquina *STG-1S*, al igual que la de la máquina *Mark-2*, contiene los tres tipos de objetos usuales: argumentos para aplicaciones, continuaciones y marcas de actualización. Siendo fieles al lenguaje STGL, la expresión de control de la máquina *STG-1S* puede tener tres formas diferentes:

- *Eval e E*, donde e es una expresión FUN (recalcamos que esto excluye las lambda formas) y E es un entorno que asocia las variables libres de la expresión e a punteros del montón.
- *Enter p*, donde p es un puntero del montón. Nótese que no existe entorno en este caso.
- *ReturnCon C* $\overline{p_i}$, donde C es un constructor de datos y $\overline{p_i}$ son sus argumentos dados como un vector de punteros del montón. Tampoco existe entorno asociado a esta instrucción.

Llamaremos a cada una de estas expresiones *instrucciones*, y usaremos el símbolo i para denotarlas. Para una mejor comparación con la máquina *Mark-2*, consideraremos una configuración de la *STG-1S* como una cuádrupla (Γ, i, E, S), donde Γ es el montón que asocia punteros a clausuras, i es la instrucción de control, E es el entorno asociado con la instrucción i en el caso de que la instrucción sea de la forma *Eval e*, y el entorno vacío $\{\}$ en los demás casos, y S es la pila. En la Figura 5.3 se muestran las reglas de transición de la máquina *STG-1S*.

Hemos numerado las reglas con los mismos números utilizados en [Pey92] para tener una referencia rápida. Como en nuestro lenguaje FUN no tenemos de forma explícita una marca $\backslash\pi$ en las lambda formas para evitar actualizaciones innecesarias, las reglas 2 y $2'$ reflejan que no es

	Montón	Control	Entorno	Pila	Regla
	Γ	$Eval\ (x\ \overline{x_i}^n)$	$E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^n]$	S	1
$\Longrightarrow$	Γ	$Enter\ p$	$\{\}$	$\overline{p_i}^n : S$	
	$\Gamma \cup [p \mapsto (\lambda \overline{x_i}^n.e, E)]$	$Enter\ p$	$\{\}$	$\overline{p_i}^n : S$	2
$\Longrightarrow$	$\Gamma \cup [p \mapsto (\lambda \overline{x_i}^n.e, E)]$	$Eval\ e$	$E \cup [\overline{x_i \mapsto p_i}^n]$	S	
	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$Enter\ p$	$\{\}$	S	$2'$
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$Eval\ (C\ \overline{x_i})$	E	S	
	Γ	$Eval\ (\mathbf{letrec}\ \{\overline{x_i = lf_i\ \vert^{t_i}}\}\ \mathbf{in}\ e)$	E	S	3 (1)
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto (lf_i, E'\ \vert^{t_i})}]$	$Eval\ e$	E'	S	
	Γ	$Eval\ (\mathbf{case}\ e\ \mathbf{of}\ alts\ \vert^{t})$	E	S	4
$\Longrightarrow$	Γ	$Eval\ e$	E	$(alts, E\ \vert^{t}) : S$	
	Γ	$Eval\ (C\ \overline{x_i})$	$E \cup [\overline{x_i \mapsto p_i}]$	S	5
$\Longrightarrow$	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	S	
	Γ	$ReturnCon\ C_k\ \overline{p_i}$	$\{\}$	$(alts, E) : S$	6 (2)
$\Longrightarrow$	Γ	$Eval\ e_k$	$E \cup [\overline{y_{ki} \mapsto p_i}]$	S	
	$\Gamma \cup [p \mapsto (e, E)]$	$Enter\ p$	$\{\}$	S	15 (3)
$\Longrightarrow$	Γ	$Eval\ e$	E	$\#p : S$	
	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	$\#p : S$	16 (4)
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, [\overline{x_i \mapsto p_i}])]$	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	S	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E)]$	$Enter\ p$	$\{\}$	$\overline{p_i}^k : \#p' : S$	17 (5)
$\Longrightarrow$	Γ'	$Enter\ p$	$\{\}$	$\overline{p_i}^k : S$	

(1) En la regla 3, las variables $\overline{p_i}$ son distintas y frescas con respecto a Γ, $\mathbf{letrec}\ \overline{x_i\ =\ lf_i\ \vert^{t_i}}\ \mathbf{in}\ e$, y S. El entorno E' se define como $E \cup [\overline{x_i \mapsto p_i}]$

(2) La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en *alts*.

(3) La expresión $e \neq C\ \overline{x_i}$ y $e \neq \lambda \overline{x_i}^n.e'$.

(4) Las variables $\overline{x_i}$ son arbitrarias y distintas.

(5) $k < n$ y $\Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E), p' \mapsto (\lambda^{n-k} \overline{x_i}.e, E \cup [\overline{x_i \mapsto p_i}^k])]$.

Figura 5.3: Máquina abstracta *STG-1S*

necesario colocar una marca de actualización en la pila cuando se referencia una forma normal en el montón. La regla $2'$ no aparece en [Pey92], pero está implícita en la regla 2.

La regla 17 es la primera propuesta en [Pey92] para realizar la actualización de una aplicación parcial. En términos de implementación, es mejor usar sin embargo la siguiente regla, numerada allí como la 17a:

$$\begin{array}{llll} & \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E)] & \mathit{Enter}\ p & \{\} & (\overline{p_i}^k : \#p' : S) \\ \Longrightarrow & \Gamma' & \mathit{Enter}\ p & \{\} & (\overline{p_i}^k : S) \end{array}$$

donde $k < n$ y $\Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E), p' \mapsto (x\ \overline{x_i}^k, [x \mapsto p, \overline{x_i \mapsto p_i}^k])]$. Para demostrar la corrección de la máquina *STG-1S* con respecto a la máquina *Mark-2*, usaremos la regla 17, ya que con ella es más sencilla la definición de estados equivalentes para ambas máquinas.

Ahora procederemos con la comparación de ambas máquinas. Para ello, y al igual que hicimos con la máquina *Mark-2*, primeramente definiremos las configuraciones *estables* de la máquina *STG-1S*.

5.3.1 Configuraciones estables de la máquina *STG-1S*

Una configuración estable para la máquina *STG-1S* corresponde o bien a la evaluación de una expresión de nuestro lenguaje FUN o a un estado de terminación.

Definición 31
Una configuración $s = (\Gamma, i, E, S)$ de la máquina STG-1S es estable *si se cumple alguna de las siguientes condiciones:*

1. $i = \mathit{Eval}\ e$ *para alguna* e *de FUN*
2. $s = (\Gamma, \mathit{ReturnCon}\ C\ \overline{p_i}, \{\}, [\,])$
3. $s = (\Gamma \cup [p \mapsto (\lambda \overline{x_i}^n.e, E')], \mathit{Enter}\ p, \{\}, [p_1, \ldots, p_k]) \wedge n > k \geq 0.$

Las configuraciones 2 y 3 corresponden a estados de terminación. Obsérvese que la máquina *STG-1S* puede terminar de forma exitosa con una pila no vacía, como sucede en el caso 3. Esto podría suceder cuando la expresión inicial evalúa a una lambda abstracción. Al igual que en la máquina *Mark-2*, utilizaremos $s \Rightarrow_e^+ s'$ para denotar la evolución entre dos configuraciones estables de la máquina *STG-1S* sin ningún punto estable intermedio, y diremos que *s evoluciona* a s'. La noción de configuración *consistente* para la máquina *STG-1S* es la misma que la que se dio en la Definición 20 para la máquina *Mark-2*.

5.4 Comparación de la evolución

Ahora ya tenemos todo lo necesario para comparar la evolución de ambas máquinas. Partiremos de puntos estables equivalentes y veremos que ambas máquinas pasan exactamente a través del mismo número de configuraciones estables, y que las configuraciones estables correspondientes son equivalentes. Esto nos servirá para decir que existe una *bisimulación* entre ambas máquinas.

Para simplificar la noción de configuración equivalente, supondremos que ambas máquinas usan exactamente el mismo generador de nombres de variables frescas en la regla *letrec*. De esta manera, si al generador le damos las mismas entradas (es decir, la misma expresión de control, montón y pila) se obtendrán los mismos nombres de variables frescas.

Definición 32
Una configuración estable $m = (\Gamma, lf, E, S, l)$ *de la máquina Mark-2 y una configuración estable* $s = (\Gamma', i, E', S')$ *de la máquina STG-1S son* equivalentes, *denotado por* $m \equiv s$, *si*

- $\Gamma = \Gamma'$, *y*
- *una de las siguientes condiciones se cumple:*
  1. $i = Eval\ e \ \wedge\ lf = e \ \wedge\ E = E' \ \wedge\ S = S'$
  2. $i = ReturnCon\ C\ \overline{p_i} \ \wedge\ lf = C\ \overline{x_i} \ \wedge\ \overline{p_i} = E\ \overline{x_i} \ \wedge\ S = S' = [\,]$
  3. $i = Enter\ p \wedge \Gamma'\ p = (\lambda \overline{x_i}^n.e, E'') \wedge S' = [p_1, \ldots, p_k] \wedge n > k \geq 0 \wedge lf = \lambda \overline{x_i}^{n-k}.e \wedge E\ [x_n, \ldots, x_{n-k+1}] = [p_1, \ldots, p_k]$

La 1ª condición del segundo punto establece la equivalencia entre dos configuraciones estables intermedias del cómputo. Las condiciones 2 y 3 de ese mismo punto se refieren a configuraciones de terminación con éxito con una forma normal. En el punto 3 la máquina *STG-1S* finaliza con argumentos por aplicar en la pila. Como ya se comentó, esto es debido a que la máquina *STG-1S* no realiza aplicaciones parciales y, sin embargo, la máquina *Mark-2* realiza aplicaciones parciales, de tal manera que la equivalencia entre ambos estados es la que se presenta en la definción. Lo único que se indica en este punto es que la lambda forma con la que termina la máquina *Mark-2* es la misma que si hacemos la aplicación parcial de la lambda forma apuntada por p, a los argumentos que quedan en la pila.

La proposición y el corolario que se exponen a continuación establecen que la máquina *STG-1S* y la máquina *Mark-2* se bisimulan. Por transitividad esto demuestra que la *STG-1S* es correcta y completa con respecto a la semántica natural de Launchbury.

Proposición 33
Dadas dos configuraciones m *y* s *estables y consistentes de las máquinas Mark-2 y STG-1S respectivamente, tales que* $m \equiv s$,

1. *Si* $m \Rightarrow_e^+ m'$, *entonces existe una configuración estable y consistente* s' *tal que* $s \Rightarrow_e^+ s'$ *y* $m' \equiv s'$.
2. *Si* $s \Rightarrow_e^+ s'$, *entonces existe una configuración estable y consistente* m' *tal que* $m \Rightarrow_e^+ m'$ *y* $m' \equiv s'$.

Corolario 34
Si e *es una expresión cerrada del lenguaje FUN, entonces* $(\{\}, e, \{\}, [\,], \perp) \Rightarrow^* m_f$ *en la máquina Mark-2 con* $m_f = (\Delta, w, E, [\,])$ *si y sólo si existe una configuración* s_f *en la máquina STG-1S tal que* $(\{\}, Eval\ e, \{\}, [\,]) \Rightarrow^* s_f$ *y* $m_f \equiv s_f$.

Demostración: (Proposición 33)
La demostración se realiza por casos, teniendo en cuenta solamente configuraciones estables. Por hipótesis, las configuraciones iniciales son consistentes y equivalentes entre sí, así que tenemos todas las variables libres definidas en los entornos asociados. Realizaremos la demostración de los puntos 1 y 2 de forma conjunta.

<u>**Letrec**</u>

Mark-2

$$\begin{array}{rl} & (\Gamma, \textbf{letrec}\ \{\overline{x_i = \mathit{lf}_i\ |^{t_i}}\}\ \textbf{in}\ e, E, S, l) \\ \Rightarrow & (\Gamma \cup [\overline{p_i \mapsto (\mathit{lf}_i, E'\ |^{t_i})}], e, E', S, \mathit{letrec}) \end{array}$$

donde $l \neq var3 \land l \neq app1$ y $E' = E \cup [\overline{x_i \mapsto p_i}]$
Esta transición alcanza directamente una configuración estable. Así que necesitamos ver que la máquina *STG-1S* alcanza una configuración estable equivalente.

STG-1S

$$\begin{array}{rl} & (\Gamma, \mathit{Eval}\ (\textbf{letrec}\ \{\overline{x_i = \mathit{lf}_i\ |^{t_i}}\}\ \textbf{in}\ e), E, S) \\ \Rightarrow & (\Gamma \cup [\overline{p'_i \mapsto (\mathit{lf}_i, E'\ |^{t_i})}], \mathit{Eval}\ e, E', S) \end{array}$$

donde $E' = E \cup [\overline{x_i \mapsto p'_i}]$
Esta es también una configuración estable.

Como ambas máquinas usan el mismo generador de nombres de variables frescas, tenemos que $\overline{p'_i = p_i}$ y entonces ambas configuraciones son equivalentes.

<u>**Case**</u>

Mark-2

$$\begin{array}{rl} & (\Gamma, \textbf{case}\ e\ \textbf{of}\ alts\ |^t, E, S, l) \\ \Rightarrow & (\Gamma, e, E, (alts, E\ |^t) : S, \mathit{case1}) \end{array}$$

donde $l \neq var3 \land l \neq app1$
Esta transición alcanza directamente una configuración estable. Así que necesitamos ver que la máquina *STG-1S* alcanza una configuración estable equivalente.

STG-1S

$$\begin{array}{rl} & (\Gamma, \mathit{Eval}\ (\textbf{case}\ e\ \textbf{of}\ alts\ |^t), E, S) \\ \Rightarrow & (\Gamma, \mathit{Eval}\ e, E, (alts, E\ |^t) : S) \end{array}$$

Esta es también una configuración estable.

Las configuraciones son estables y equivalentes.

<u>**Constructor**</u>

Mark-2 La configuración inicial de partida es:

$$(\Gamma, C_k\ \overline{x_i}, E, S, l)$$

Distinguiremos tres casos de acuerdo a los contenidos de la pila. En todos los casos h podría ser 0:

1. $S = \overline{\#p_j}^h : (alts, E') : S'$.

$$\begin{array}{ll} & (\Gamma, C_k\ \overline{x_i}, E \cup [\overline{x_i \mapsto q_i}], \overline{\#p_j}^h : (alts, E') : S', l) \\ \Rightarrow^h & (\Gamma \cup [\overline{p_j \mapsto (C_k\ \overline{x_i}, E^{\{\overline{x_i}\}})}], C_k\ \overline{x_i}, E \cup [\overline{x_i \mapsto q_i}], (alts, E') : S', var3) \\ \Rightarrow & (\Gamma \cup [\overline{p_j \mapsto (C_k\ \overline{x_i}, E^{\{\overline{x_i}\}})}], e_k, E' \cup [\overline{y_{ki} \mapsto q_i}], S, case2) \end{array}$$

donde la expresión e_k corresponde con la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*. En este caso la máquina alcanza una configuración estable.

2. $S = \overline{\#p_j}^h : p : S'$

$$\begin{array}{ll} & (\Gamma, C_k\ \overline{x_i}, E, \overline{\#p_j}^h : p : S', l) \\ \Rightarrow^h & (\Gamma \cup [\overline{p_j \mapsto (C_k\ \overline{x_i}, E^{\{\overline{x_i}\}})}], C_k\ \overline{x_i}, E, p : S', var3) \end{array}$$

Si $h = 0$ la máquina finaliza en la configuración inicial, y no puede realizar ningún computo. Por otro lado, si $h \neq 0$ entonces la máquina finaliza en una configuración nueva que tampoco es estable. En ambos casos la máquina no puede llegar a otra configuración estable, así que, basta con demostrar que en la otra máquina tampoco se pasa a otro punto estable. Ambas son configuraciones de fallo.

3. $S = [\overline{\#p_j}^h]$

$$\begin{array}{ll} & (\Gamma, C_k\ \overline{x_i}, E \cup [\overline{x_i \mapsto p_i}], [\overline{\#p_j}^h], l) \\ \Rightarrow^h & (\Gamma \cup [\overline{p_j \mapsto (C_k\ \overline{x_i}, E^{\{\overline{x_i}\}})}], C_k\ \overline{x_i}, E \cup [\overline{x_i \mapsto p_i}], [\ ], var3) \end{array}$$

Este es una configuración estable ya que la pila está vacía. Corresponde con una configuración estable de finalización con una forma normal.

Estos tres casos cubren todas las posibles situaciones de la pila.

Ahora veremos las posibles reducciones de la máquina *STG-1S*.

STG-1S

$$\begin{array}{ll} & (\Gamma, Eval\ (C\ \overline{x_i}), E \cup [\overline{x_i \mapsto q_i}], S) \\ \Rightarrow & (\Gamma, ReturnCon\ C\ \overline{q_i}, \{\}, S) \end{array}$$

Distinguiremos los mismos tres casos que en la máquina *Mark-2* para que la comparación sea lo más sencilla posible. En todos los casos h podría ser 0:

1. $S = \overline{\#p_j}^h : (alts, E') : S'$.

$$\begin{array}{ll} & (\Gamma, ReturnCon\ C\ \overline{q_i}, \{\}, \overline{\#p_j}^h : (alts, E') : S') \\ \Rightarrow^h & (\Gamma \cup [\overline{p_j \mapsto (C\ \overline{x'_i}, [\overline{x'_i \mapsto q_i}])}], ReturnCon\ C\ \overline{q_i}, \{\}, (alts, E') : S') \\ \Rightarrow & (\Gamma \cup [\overline{p_j \mapsto (C\ \overline{x'_i}, [\overline{x'_i \mapsto q_i}])}], Eval\ e_k, E' \cup [\overline{y_{ki} \mapsto q_i}], S) \end{array}$$

donde la expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*. En este caso llegamos a una configuración estable.

2. $S = \overline{\#p_j}^h : p : S'$

$$\begin{array}{ll} & (\Gamma, ReturnCon\ C\ \overline{q_i}, \{\}, \overline{\#p_j}^h : p : S') \\ \Rightarrow^h & (\Gamma \cup [\overline{p_j \mapsto (C\ \overline{x_i'}, [\overline{x_i' \mapsto q_i}])}], ReturnCon\ C\ \overline{q_i}, \{\}, p : S') \end{array}$$

La máquina finaliza en una nueva configuración que no es estable. Este es un estado de finalización en una configuración de fallo, ya que en la cima de la pila no tenemos las *alternativas* de un **case** y, por tanto, no existe progreso.

3. $S = [\overline{\#p_j}^h]$

$$\begin{array}{ll} & (\Gamma, ReturnCon\ C\ \overline{q_i}, \{\}, [\overline{\#p_j}^h]) \\ \Rightarrow^h & (\Gamma \cup [\overline{p_j \mapsto (C\ \overline{x_i'}, [\overline{x_i' \mapsto q_i}])}], ReturnCon\ C\ \overline{q_i}, \{\}, [\ ]) \end{array}$$

Esta es una configuración estable ya que la pila está vacía. Esta configuración corresponde con una configuración de finalización con la forma normal.

Estos tres casos cubren todas las posibles configuraciones de la pila.

Ahora tenemos que ver que son equivalentes las configuraciones estables a las que llegamos. Esto surge del hecho de que la máquina *STG-1S* es libre para elegir cualquier conjunto $\overline{x_i'}$ de variables arbitrarias distintas con la única restricción de que no pueden ser iguales. Podemos elegirlas de la siguiente forma $\overline{x_i = x_i'}$, así las configuraciones son equivalentes.

Aplicación

En esté apartado, utilizaremos el superíndice ĵ para denotar lo siguiente $[\overline{s}^{\hat{\jmath}}] = [\overline{s}^m \ldots \overline{s}^j]$. $j \in \{m \ldots 1\}$, donde m es un entero.

Mark-2

$$\begin{array}{ll} & (\Gamma, x\ \overline{x_i}^k, E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^k], S, l) \\ \Rightarrow^k & (\Gamma, x, E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^k], \overline{p_i}^k : S, app1) \end{array}$$

Distinguiremos cuatro casos según los contenidos del montón.

1. $\Gamma\ p = (\lambda \overline{x_i}^n.e', E')$

$$\begin{array}{ll} & (\Gamma, x, E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^k], \overline{p_i}^k : S, l) \\ \Rightarrow & (\Gamma, \lambda \overline{x_i}^n.e', E', \overline{p_i}^k : S, var1a) \end{array}$$

Ahora haremos otra distinción de acuerdo a los contenidos de la pila.

En todos estos casos la notación $\overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^m$ es una abreviatura de $\overline{p_t}^{k_m} : \overline{\#q_t}^{l_m} : \ldots : \overline{p_t}^{k_1} : \overline{\#q_t}^{l_1}$ donde hemos numerado en orden decreciente por conveniencia.

(a) $S = \overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^m : S'$ donde $\forall j \in \{1 \ldots m-1\}.\ k_j > 0$, $k_m \geq 0$, $l_1 = 0$, $\forall j \in \{2 \ldots m\}.\ l_j > 0$ y $k + \sum_{h=1}^{m} k_h = n$.

$$
\begin{array}{ll}
 & (\Gamma, \lambda\overline{x_i}^{n}.e', E', \overline{p_i}^{k} : S, var1a) \\
\Rightarrow^{k} & (\Gamma, \lambda\overline{x_i}^{(n-k)}.e', E' \cup [\overline{x_i \mapsto p_i}^{k}], S, app2) \\
\ldots & \\
\Rightarrow^{k_j} & (\Gamma_{j+1}, \lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_{j+1} \cup [\overline{x_i \mapsto p_t}^{k_j}], \overline{\#q_t}^{l_j} : \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)} \\
 & : S', app2) \\
\Rightarrow^{l_j} & (\Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}], \lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j, \\
 & \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)} : S', var2) \\
\ldots & \\
\Rightarrow^{k_1} & (\Gamma_2, \lambda\overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_2 \cup [\overline{x_i \mapsto p_t}^{k_1}], \overline{\#q_t}^{l_1} : S', app2)
\end{array}
$$

donde

- $E_{m+1} = E' \cup [\overline{x_i \mapsto p_i}^{k}]$ y $\forall j \in \{1 \ldots m\}.\ E_j = E_{j+1} \cup [\overline{x_i \mapsto p_t}^{k_j}]$ así $E_j = E' \cup [\overline{x_i \mapsto p_i}^{k}, \overline{\overline{x_i \mapsto p_t}^{k_h}}^{\hat{j}}]$.
- $\Gamma_{m+1} = \Gamma$ y $\forall j \in \{2 \ldots m\}.\ \Gamma_j = \Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}]$, por tanto, $\Gamma_j = \Gamma \cup [\overline{\overline{q_t \mapsto (\lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_h}}^{\hat{j}}]$

Que finaliza en el estado $(\Gamma_2, e', E_1, \overline{\#q_t}^{l_1} : S', app2)$ el cual es una configuración estable ya que no es una lambda forma y la última regla aplicada es la regla *app2*.

(b) $S = \overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^{m} : (alts, E'') : S'$ donde $\forall j \in \{1 \ldots m-1\}.\ k_j > 0$, $k_m \geq 0$, $l_1 \geq 0$, $\forall j \in \{2 \ldots m\}.\ l_j > 0$ y $k + \sum_{h=1}^{m} k_h < n$.

$$
\begin{array}{ll}
 & (\Gamma, \lambda\overline{x_i}^{n}.e', E', \overline{p_i}^{k} : S, var1a) \\
\Rightarrow^{k} & (\Gamma, \lambda\overline{x_i}^{(n-k)}.e', E' \cup [\overline{x_i \mapsto p_i}^{k}], S, app2) \\
\ldots & \\
\Rightarrow^{k_j} & (\Gamma_{j+1}, \lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_{j+1} \cup [\overline{x_i \mapsto p_t}^{k_j}], \overline{\#q_t}^{l_j} : \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)} \\
 & : (alts, E'') : S', app2) \\
\Rightarrow^{l_j} & (\Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}], \lambda\overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j, \\
 & \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)} : (alts, E'') : S', var2) \\
\ldots & \\
\Rightarrow^{l_1} & (\Gamma_2 \cup [\overline{q_t \mapsto (\lambda\overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_1)}^{l_1}], \lambda\overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_1, \\
 & (alts, E'') : S', var2)
\end{array}
$$

donde

- $E_{m+1} = E' \cup [\overline{x_i \mapsto p_i}^{k}]$ y $\forall j \in \{1 \ldots m\}.\ E_j = E_{j+1} \cup [\overline{x_i \mapsto p_t}^{k_j}]$ así $E_j = E' \cup [\overline{x_i \mapsto p_i}^{k}, \overline{\overline{x_i \mapsto p_t}^{k_h}}^{\hat{j}}]$.

- $\Gamma_{m+1} = \Gamma$ y $\forall j \in \{1 \ldots m\}.\ \Gamma_j = \Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}]$ así $\Gamma_j = \Gamma \cup [\overline{\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_h}}^{\hat{\jmath}}]$

Que finaliza en el estado $(\Gamma_1, \lambda \overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_1, (alts, E'') : S', var2)$ el cual no es una configuración estable ya que tiene una lambda forma en la expresión de control. Por tanto, es una configuración de fallo.

(c) $S = \overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^{m}$ donde $\forall j \in \{1 \ldots m-1\}.\ k_j > 0,\ k_m \geq 0\ ,\ l_1 \geq 0,\ \forall j \in \{2 \ldots m\}.\ l_j > 0$ y $k + \sum_{h=1}^{m} k_h < n$.

$$
\begin{array}{ll}
 & (\Gamma, \lambda \overline{x_i}^{n}.e', E', \overline{p_i}^{k} : S, var1a) \\
\Rightarrow^{k} & (\Gamma, \lambda \overline{x_i}^{(n-k)}.e', E' \cup [\overline{x_i \mapsto p_i}^{k}], S, app2) \\
\ldots & \\
\Rightarrow^{k_j} & (\Gamma_{j+1}, \lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_{j+1} \cup [\overline{x_i \mapsto p_t}^{k_j}], \overline{\#q_t}^{l_j} : \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)}, \\
 & app2) \\
\Rightarrow^{l_j} & (\Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}], \lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j, \\
 & \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)}, var2) \\
\ldots & \\
\Rightarrow^{l_1} & (\Gamma_2 \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_1)}^{l_1}], \lambda \overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_1, [\,], \\
 & var2)
\end{array}
$$

donde

- $E_{m+1} = E' \cup [\overline{x_i \mapsto p_i}^{k}]$ y $\forall j \in \{1 \ldots m\}.\ E_j = E_{j+1} \cup [\overline{x_i \mapsto p_t}^{k_j}]$ así $E_j = E' \cup [\overline{x_i \mapsto p_i}^{k}, \overline{\overline{x_i \mapsto p_t}^{k_h}}^{\hat{\jmath}}]$.
- $\Gamma_{m+1} = \Gamma$ y $\forall j \in \{1 \ldots m\}.\ \Gamma_j = \Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}]$ así $\Gamma_j = \Gamma \cup [\overline{\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_h}}^{\hat{\jmath}}]$

Que finaliza en el estado $(\Gamma_1, \lambda \overline{x_i}^{(n-k-\sum_{h=1}^{m} k_h)}.e', E_1, [\,], var2)$ el cual es una configuración estable ya que tiene una lambda forma en la expresión de control y la pila está vacía.

Estos tres casos cubren todas las posibles situaciones de la pila.

2. $\Gamma\ p = (C\ \overline{x_j}, E')$

$$
\begin{array}{ll}
 & (\Gamma, x, E \cup [x \mapsto p], \overline{p_i}^{k} : S, l') \\
\Rightarrow & (\Gamma, C\ \overline{x_j}, E', \overline{p_i}^{k} : S, var1b)
\end{array}
$$

Esta es una configuración estable.

3. $\Gamma\ p = (e', E')$ where $e' \neq C\ \overline{x_i}$

$$
\begin{array}{ll}
 & (\Gamma' \cup [p \mapsto (e', E')], x, E \cup [x \mapsto p], \overline{p_i}^{k} : S, l) \\
\Rightarrow & (\Gamma', e', E', \#p : \overline{p_i}^{k} : S, var1c)
\end{array}
$$

Esta es una configuración estable.

4. $\Gamma\ p$ no está definida. Entonces esta es una configuración donde la máquina trata de entrar en una clausura bajo evaluación (en términos de cómputo, a un agujero negro). Finaliza en una configuración que podría ser estable si $k = 0$, pero en ese caso sería la de partida. En cualquier caso corresponde a una configuración de fallo, ya que no se puede proseguir.

STG-1S La evaluación de la máquina *STG*:

$$\begin{array}{ll} & (\Gamma, Eval\ (x\ \overline{x_i}^k), E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^k], S) \\ \Rightarrow & (\Gamma, Enter\ p,\ \{\ \}, \overline{p_i}^k : S) \end{array}$$

Al igual que con la máquina *Mark-2*, distinguiremos cuatro casos de acuerdo con los contenidos del montón.

1. $\Gamma\ p = (\lambda \overline{x_i}^n.e', E')$

   Ahora realizaremos la misma distinción que en la *Mark-2* de acuerdo a los contenidos de la pila.

   En todos los casos la notación $\overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^m$ es una abreviatura de $\overline{p_t}^{k_m} : \overline{\#q_t}^{l_m} : \ldots : \overline{p_t}^{k_1} : \overline{\#q_t}^{l_1}$ donde hemos numerado en orden decreciente por conveniencia.

   (a) $S = \overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^m : S'$ donde $\forall j \in \{1 \ldots m-1\}.\ k_j > 0,\ k_m \geq 0\ ,\ l_1 = 0,$ $\forall j \in \{2 \ldots m\}.\ l_j > 0$ y $k + \sum_{h=1}^m k_h = n.$

   $$\begin{array}{ll} & (\Gamma, Enter\ p,\ \{\ \}, \overline{p_i}^k : S) \\ \ldots & \\ \Rightarrow^{l_j} & (\Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^m k_h)}.e', E_j)}^{l_j}], Enter\ p,\ \{\ \}, \overline{p_i}^k \\ & : \overline{\overline{p_t}^{k_h}}^{\hat{\jmath}} : \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)} : S') \\ \ldots & \\ \Rightarrow & (\Gamma_2, Eval\ e', E' \cup [\overline{x_i \mapsto p_i}^k, \overline{\overline{x_i \mapsto p_t}^{k_h}}^m], \overline{\overline{p_t}^{k_h}}^m : \overline{\#q_t}^{l_1} : S') \end{array}$$

   donde

   - $E_{m+1} = E' \cup [\overline{x_i \mapsto p_i}^k]$ y $\forall j \in \{1 \ldots m\}.\ E_j = E' \cup [\overline{x_i \mapsto p_i}^k, \overline{\overline{x_i \mapsto p_t}^{k_h}}^{\hat{\jmath}}].$
   - $\Gamma_{m+1} = \Gamma$ y $\forall j \in \{2 \ldots m\}.\ \Gamma_j = \Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^m k_h)}.e', E_j)}^{l_j}]$

     así $\Gamma_j = \Gamma \cup [\overline{\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^m k_h)}.e', E_j)}^{l_h}}^{\hat{\jmath}}]$

   Que finaliza en el estado $(\Gamma_2, Eval\ e', E_1, S')$ que es una configuración estable.

   (b) $S = \overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^m : (alts, E'') : S'$ donde $\forall j \in \{1 \ldots m-1\}.\ k_j > 0,\ k_m \geq 0\ ,$ $l_1 \geq 0,\ \forall j \in \{2 \ldots m\}.\ l_j > 0$ y $k + \sum_{h=1}^m k_h < n.$

   $$\begin{array}{ll} & (\Gamma, Enter\ p,\ \{\ \}, \overline{p_i}^k : S) \\ \ldots & \\ \Rightarrow^{l_j} & (\Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^m k_h)}.e', E_j)}^{l_j}], Enter\ p,\ \{\ \}, \overline{p_i}^k : \overline{\overline{p_t}^{l_h}}^{\hat{\jmath}} \\ & : \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)} : (alts, E'') : S') \\ \ldots & \\ \Rightarrow & (\Gamma_1, Enter\ p,\ \{\ \}, \overline{p_i}^k : \overline{\overline{p_t}^{l_h}}^m : (alts, E'') : S') \end{array}$$

donde

- $E_{m+1} = E' \cup [\overline{x_i \mapsto p_i}^k]$ y $\forall j \in \{1 \ldots m\}.\ E_j = E' \cup [\overline{x_i \mapsto p_i}^k, \overline{\overline{x_i \mapsto p_t}^{k_h}}^{\hat{\jmath}}]$.
- $\Gamma_{m+1} = \Gamma$ y $\forall j \in \{1 \ldots m\}.\ \Gamma_j = \Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}]$
  así $\Gamma_j = \Gamma \cup [\overline{\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_h}}^{\hat{\jmath}}]$

Que finaliza en el estado $(\Gamma_1, Enter\ p, E_1, \overline{p_i}^k : \overline{\overline{p_t}^{l_h}}^m : (alts, E'') : S')$ que no es una configuración estable, y a partir del cual no se puede continuar. Por tanto, representa una configuración de fallo.

(c) $S = \overline{\overline{p_t}^{k_j} : \overline{\#q_t}^{l_j}}^m$ donde $\forall j \in \{1 \ldots m-1\}.\ k_j > 0,\ k_m \geq 0\ ,\ l_1 \geq 0,\ \forall i \in \{2 \ldots m\}.\ l_j > 0$ y $k + \sum_{h=1}^{m} k_h < n$.

$$
\begin{array}{ll}
 & (\Gamma, Enter\ p, \{\ \}, \overline{p_i}^k : S) \\
\ldots & \\
\Rightarrow^{l_j} & (\Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}], Enter\ p, \{\ \}, \overline{p_i}^k : \overline{\overline{p_t}^{k_h}}^{\hat{\jmath}} \\
 & : \overline{\overline{p_t}^{k_h} : \overline{\#q_t}^{l_h}}^{(j-1)}) \\
\ldots & \\
\Rightarrow & (\Gamma_1, Enter\ p, \{\ \}, \overline{p_i}^k : \overline{\overline{p_t}^{l_h}}^m)
\end{array}
$$

donde

- $E_{m+1} = E' \cup [\overline{x_i \mapsto p_i}^k]$ y $\forall j \in \{1 \ldots m\}.\ E_j = E' \cup [\overline{x_i \mapsto p_i}^k, \overline{\overline{x_i \mapsto p_t}^{k_h}}^{\hat{\jmath}}]$.
- $\Gamma_{m+1} = \Gamma$ y $\forall j \in \{1 \ldots m\}.\ \Gamma_j = \Gamma_{j+1} \cup [\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_j}]$
  así $\Gamma_j = \Gamma \cup [\overline{\overline{q_t \mapsto (\lambda \overline{x_i}^{(n-k-\sum_{h=j}^{m} k_h)}.e', E_j)}^{l_h}}^{\hat{\jmath}}]$

Que finaliza en el estado $(\Gamma_1, Enter\ p, \{\ \}, \overline{p_i}^k : \overline{\overline{p_t}^{l_h}}^m)$ que representa una configuración de finalización.

Estos tres casos cubren todas las posibles situaciones de la pila.

2. $\Gamma\ p = (C\ \overline{x_j}, E')$

$$
\begin{array}{ll}
 & (\Gamma, Enter\ p, \{\}, \overline{p_i}^k : S) \\
\Rightarrow & (\Gamma, Eval\ C\ \overline{x_j}, E', \overline{p_i}^k : S)
\end{array}
$$

Esta es una configuración estable, pero no existe progreso futuro. Corresponde a una configuración de fallo.

3. $\Gamma = \Gamma' \cup [p \mapsto (e', E')]$ donde $e' \neq C\ \overline{x_i}$

$$
\begin{array}{ll}
 & (\Gamma' \cup [p \mapsto (e', E')], Enter\ p, \{\}, \overline{p_i}^k : S) \\
\Rightarrow & (\Gamma', Eval\ e', E', \#p : \overline{p_i}^k : S)
\end{array}
$$

Esta es una configuración estable.

4. $\Gamma\ p$ está indefinido. Entonces esta es una configuración donde la máquina trata de entrar en una clausura bajo evaluación (en términos de cómputo, a un agujero negro). Esta configuración corresponde con una configuración de fallo y no existe progreso futuro.

Cuando una de las máquinas llega a otra configuración (distinta a la inicial) estable, la otra llega a una configuración estable equivalente, y si una de las dos máquinas no puede alcanzar otra configuración estable entonces a la otra máquina le pasa lo mismo.

Estos cuatro casos (*letrec*, *case*, *constructor*, *aplicación*) cubren todos las posibles configuraciones de partida. Por tanto, queda demostrada la proposición.

□

5.5 Algunas propiedades de la máquina *STG-1S*

Una de las propiedades importantes de las máquinas *STG-1S* y *Mark-2* es que los entornos asociados a cada expresión son minimales, si entendemos por minimal que *sólo* tienen asociadas las variables libres de la expresión correspondiente, exceptuando el entorno asociado a la expresión de control y los entornos provenientes de actualizaciones con lambda abstracciones aplicadas parcialmente, ya que en este caso no se realiza la poda.

Esto podría solucionarse indicando, para cada variable de la lambda forma, si es o no es utilizada en el cuerpo de la lambda. Por lo tanto, llegaríamos a entornos minimales en ambas máquinas. No obstante, no vamos a preocuparnos de ello, ya que la máquina *STG* real, en vez de utilizar la regla (17) para las actualizaciones, utiliza la regla (17a). Al realizar las actualizaciones con lambda abstracciones aplicadas parcialmente, con esta regla conseguimos que sean minimales todos los entornos excepto el asociado a la expresión de control.

En la generación de código que se expone en el Capítulo 6 utilizaremos la regla siguiente:

$$\begin{array}{ll} & (\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E)],\ Enter\ p,\ \{\},\ \overline{p_i}^k : \#p' : S) \\ \Longrightarrow & (\Gamma',\ Enter\ p,\ \{\},\ \overline{p_i}^k : S) \\ \text{donde} & k < n \text{ y } \Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E),\ p' \mapsto (x\ \overline{x_i}^k, [x \mapsto p,\ \overline{x_i \mapsto p_i}^k])] \\ & x \text{ es una variable cualquiera distinta de las } \overline{x_i}^k \end{array}$$

que es la regla (17a) adaptada a la máquina *STG-1S*.

5.6 La máquina abstracta *STG*

En la Figura 5.4 reproducimos el subconjunto de la máquina real *STG* para la ejecución de nuestro lenguaje FUN. Como es bien sabido, la máquina *STG* tiene tres pilas: la pila de *argumentos as*, la cual contiene los argumentos pendientes de aplicación y refleja fielmente el multiconjunto B de nuestra semántica; la pila de *retornos rs*, que contiene pares del tipo $(alts, E)$ y corresponde fielmente con el multiconjunto C de nuestra semántica; y la pila de *actualizaciones us* que contiene marcos de actualización. Un marco de actualización es una tripla (as, rs, p) consistente en una pila de argumentos, una pila de retornos y un puntero p a la clausura a actualizar. El conjunto de punteros p de la pila us corresponde fielmente con el conjunto A de la semántica de Sestoft y también de la nuestra, si no aplicamos sobre él la recolección de basura.

Las diferencias principales con la máquina *STG-1S* de la sección anterior son:

- La máquina *STG* tiene tres pilas separadas. La colocación y eliminación se realiza en la pila adecuada de acuerdo con la regla.

- En lugar de colocar punteros de actualización, la máquina *STG* coloca marcos de actualización y deja vacías tanto la pila de argumentos como la pila de continuaciones de la configuración. Cuando se alcanza una forma normal con dichas pilas vacías, se realiza una actualización y se devuelven las pilas al estado almacenado en el marco de la actualización correspondiente, tomado de la pila *us* (ver reglas 16 y 17).

5.7 Comparación de las máquinas *STG* y *STG-1S*

Aparentemente las diferencias expuestas en la sección anterior no son esenciales, y se podría pensar que el comportamiento de ambas máquinas es exactamente el mismo. No obstante, la división de la pila en tres pilas separadas tiene la desafortunada consecuencia de una pérdida del orden temporal entre los elementos de las pilas *as* y *rs*. Como consecuencia, una continuación colocada en la pila *rs antes* de que se coloque un argumento en la pila *as*, puede ser retirada *antes* de que el argumento sea retirado de la pila *as* en lugar de *después*, como debería suceder, y como sucede en la máquina *STG-1S*. Y lo mismo podría ocurrir en el caso contrario: un argumento colocado en la pila *as antes* de que se coloque una continuación en la pila *rs*, puede ser retirado *antes* de que la continuación haya sido retirada de la pila *rs* en lugar de *después*, como debería suceder, y como sucede en la máquina *STG-1S*. Considérese el siguiente programa mal tipado:

$$\begin{array}{l} e = \mathbf{letrec}\ \ \begin{array}{l} y_1 = Nil \\ id = \lambda x.x \end{array} \\ \quad \mathbf{in\ case}\ y_1\, y_1\ \mathbf{of} \\ \qquad\qquad Nil \rightarrow id \end{array}$$

el cual no posee semántica. La máquina *STG* lo reduce como si fuera la expresión:

$$\begin{array}{l} e = \mathbf{letrec}\ \ \begin{array}{l} y_1 = Nil \\ id = \lambda x.x \end{array} \\ \quad \mathbf{in\ case}\ y_1\ \mathbf{of} \\ \qquad\qquad Nil \rightarrow id\ y_1. \end{array}$$

que está bien tipada y, por lo tanto, tiene una semántica de reducción. La secuencia de reducciones para la primera expresión sería la siguiente:

$$\begin{array}{rl} & (\{\},\, e,\, \{\},\, [\,],\, [\,],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Eval\ (\mathbf{case}\ y_1\ y_1\ \mathbf{of}\ Nil \rightarrow id),\, E_1,\, [\,],\, [\,],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Eval\ (y_1\ y_1),\, E_1,\, [\,],\, [(Nil \rightarrow id,\, E_1)],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Enter\ p_1,\, \{\},\, [p_1],\, [(Nil \rightarrow id,\, E_1)],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Eval\ Nil,\, \{\},\, [p_1],\, [(Nil \rightarrow id,\, E_1)],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, ReturnCon\ Nil\ \{\},\, [p_1],\, [(Nil \rightarrow id,\, E_1)],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Eval\ id,\, E_1,\, [p_1],\, [\,],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Enter\ p_2,\, \{\},\, [p_1],\, [\,],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Eval\ x,\, [x \mapsto p_1],\, [\,],\, [\,],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, Eval\ Nil,\, \{\},\, [\,],\, [\,],\, [\,]) \\ \Rightarrow & (\Gamma_1,\, ReturnCon\ Nil\ \{\},\, [\,],\, [\,],\, [\,]) \end{array}$$

	Montón	Control	Entorno	as	rs	us	Regla
	Γ	$Eval\ (x\ \overline{x_i}^n)$	$E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^n]$	as	rs	us	1
$\Rightarrow$	Γ	$Enter\ p$	$\{\}$	$\overline{p_i}^n : as$	rs	us	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E)]$	$Enter\ p$	$\{\}$	$\overline{p_i}^n : as$	rs	us	2
$\Rightarrow$	Γ	$Eval\ e$	$E \cup [\overline{x_i \mapsto p_i}^n]$	as	rs	us	
	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$Enter\ p$	$\{\}$	as	rs	us	$2'$
$\Rightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, E)]$	$Eval\ (C\ \overline{x_i})$	E	as	rs	us	
	Γ	$Eval\ (\mathbf{letrec}\ \overline{x_i = e_i\vert^{t_i}}\ \mathbf{in}\ e)$	E	as	rs	us	3 (1)
$\Rightarrow$	$\Gamma \cup [\overline{p_i \mapsto (e_i, E'\ \vert^{t_i})}]$	$Eval\ e$	E'	as	rs	us	
	Γ	$Eval\ (\mathbf{case}\ e\ \mathbf{of}\ alts\ \vert^t)$	E	as	rs	us	4
$\Rightarrow$	Γ	$Eval\ e$	E	as	$(alts, E\vert^t) : rs$	us	
	Γ	$Eval\ (C\ \overline{x_i})$	$E \cup [\overline{x_i \mapsto p_i}]$	as	rs	us	5
$\Rightarrow$	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	as	rs	us	
	Γ	$ReturnCon\ C_k\ \overline{p_i}$	$\{\}$	as	$(alts, E) : rs$	us	6 (2)
$\Rightarrow$	Γ	$Eval\ e_k$	$E \cup [\overline{y_{ki} \mapsto p_i}]$	as	rs	us	
	$\Gamma \cup [p \mapsto (e, E)]$	$Enter\ p$	$\{\}$	as	rs	us	15 (3)
$\Rightarrow$	Γ	$Eval\ e$	E	$\{\}$	$\{\}$	$(as, rs, p) : us$	
	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	$\{\}$	$\{\}$	$(as, rs, p) : us$	16 (4)
$\Rightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i}, [\overline{x_i \mapsto p_i}])]$	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	as	rs	us	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E)]$	$Enter\ p$	$\{\}$	$\overline{p_i}^k$	$\{\}$	$(as, rs, q) : us$	17 (5)
$\Rightarrow$	Γ'	$Enter\ p$	$\{\}$	$\overline{p_i}^k : as$	rs	us	

(1) En la regla 3, las variables $\overline{p_i}$ son distintas y frescas. $E' = E \cup [\overline{x_i \mapsto p_i}]$.
(2) La expresión e_k corresponde con la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*.
(3) La expresión $e \neq C\ \overline{x_i}$ y $e \neq \lambda \overline{x_i}^n.e'$.
(4) En la regla 16, las variables $\overline{x_i}$ son una secuencia arbitraria de variables distintas.
(5) $k < n$ y $\Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e, E), q \mapsto (\lambda \overline{x_i}^{n-k}.e, E \cup [\overline{x_i \mapsto p_i}^k])]$.

Figura 5.4: Máquina abstracta *STG*

donde $\Gamma_1 = [p_1 \mapsto (Nil,\ \{\}),\ p_2 \mapsto (\lambda x.x,\ \{\})]$ y $E_1 = [y_1 \mapsto p_1,\ id \mapsto p_2]$.

Por tanto, la corrección de la máquina STG con tres pilas se mantiene sólo sobre programas bien tipados. Este no es el caso de la máquina *STG-1S*: si un programa está mal tipado, ni la semántica de Launchbury ni la máquina *STG-1S* son capaces de derivar una forma normal para ella.

Sin embargo la máquina STG es completa, en el sentido de que toda derivación exitosa realizada por la máquina *STG-1S* puede ser computada por la máquina STG. Para cualquier configuración de la máquina *STG-1S* podemos generar un único estado equivalente de la máquina STG. Lo único necesario es dividir la pila en tres pilas, siendo éstas las mismas que surgirían de la ejecución de la máquina STG a partir de la misma expresión inicial de partida.

Veremos a continuación una forma de división de la pila que nos garantiza que las nuevas pilas generadas son exactamente las mismas que se producirían en el cómputo de la máquina STG. Obsérvese que la única diferencia existente entre la máquina STG y la *STG-1S* es la consulta de la pila adecuada en las correspondientes reglas, ya que las reglas son exactamente las mismas.

Partimos de una pila S que ha surgido en el cómputo de la evaluación de una expresión cerrada e, en la máquina *STG1-S*

$$\begin{array}{ll} & (\{\},\ Eval\ e,\ \{\},\ [\,]) \\ \Rightarrow^* & (\Gamma,\ i,\ E,\ S) \end{array}$$

y queremos obtener exactamente las mismas pilas *as*, *rs* y *us* que se hubieran generado si evaluamos e en la máquina STG,

$$\begin{array}{ll} & (\{\},\ Eval\ e,\ \{\},\ [\,],\ [\,],\ [\,]) \\ \Rightarrow^* & (\Gamma,\ i,\ E,\ as,\ rs,\ us). \end{array}$$

Para ello, hemos de tener en cuenta que en la pila S tenemos almacenados los argumentos, los punteros de actualización y las continuaciones en el orden estricto en que fueron introduciéndose en ella. Así, lo que hay en la cima de la pila es lo que se apiló más recientemente, y lo que está abajo de la pila es lo primero que se apiló. Teniendo en cuenta esto, vamos a transformar la pila S en las tres pilas correspondientes *as*, *rs* y *us* siguiendo el proceso de generación que éstas hubieran proseguido si hubiéramos evaluado la expresión en la STG. El algoritmo de transformación parte del elemento que se encuentra al final de la pila y lo coloca en la pila correspondiente, y si se encuentra una marca de actualización genera el marco de actualización correspondiente en la pila *us* y deja vacías las pilas *as* y *rs*.

Definición 35

Definimos la función $Trans :: Pila\_S_{STG-1S} \to (Pila\_as, Pila\_rs, Pila\_us)_{STG}$.

$$\begin{array}{rcl} Trans\ [\![S]\!] & = & TransAcu\ [\![S]\!]\ ([\,],\ [\,],\ [\,]) \\ TransAcu\ [\![S;p]\!]\ (as, rs, us) & = & TransAcu\ [\![S]\!]\ (p : as, rs, us) \\ TransAcu\ [\![S;(alts,E)]\!]\ (as, rs, us) & = & TransAcu\ [\![S]\!]\ (as, (alts,E) : rs, us) \\ TransAcu\ [\![S;\#p]\!]\ (as, rs, us) & = & TransAcu\ [\![S]\!]\ ([\,],\ [\,], (p, as, rs) : us) \\ TransAcu\ [\![[\,]]\!]\ (as, rs, us) & = & (as, rs, us) \end{array}$$

Comparando las reglas realizadas por ambas máquinas y observando la evolución de las correspondientes pilas, es inmediato ver que esta transformación es equivalente al cómputo que realizaría la máquina STG con respecto a las pila. Es posible realizar un cómputo de división de la pila S con el mismo resultado partiendo desde la cima, pero no quedaría tan claro como empezando desde el final.

Con esto queda demostrado que la máquina STG es completa, ya que la máquina *STG-1S* lo es, y que cualquier cálculo que pueda realizar la máquina *STG-1S* lo puede realizar la máquina STG.

5.7.1 Nueva máquina $STGT$

En esta sección vamos a modificar la máquina STG con tres pilas (as, rs, us) para que sea correcta. Para ello mantendremos de alguna manera la información relativa al momento en que se produjeron los distintos acontecimientos en las diferentes pilas.

Una primera aproximación a la solución de dicho problema fue mantener asociado a cada argumento o continuación un índice que nos indicara el momento en que se produjo la colocación de dicho argumento en la pila. Tendríamos un índice global que sólo se incrementaría cuando se produjera una colocación de argumentos en la pila de argumentos as, o cuando se colocasen continuaciones en la pila de continuaciones rs. Este índice disminuiría cuando retiráramos el valor de la pila correspondiente, y nos obligaría a sacar de forma ordenada los argumentos con respecto a las alternativas.

Sin embargo esta modificación nos parece poco elegante y conlleva una gran cantidad de espacio adicional, ya que para cada argumento pendiente de aplicación y para cada continuación necesitaríamos asociar un entero que nos indicaría el momento en que se produjo dicho dato.

Una alternativa más elegante es llevar asociado un puntero a la pila en la que se apiló el último argumento o alternativa, as o rs. Para mantenerlo de forma correcta, cuando se apila algo en una de las dos pilas lo debemos actualizar con la pila en la que se produjo la inserción, y debemos introducir el puntero antiguo en los argumentos o alternativas almacenados en la correspondiente pila, de tal manera que, cuando eliminemos de una de las dos pilas, as o rs, el argumento o alternativa de la cima, podamos recuperar el antiguo puntero. Las reglas para la máquina STG modificada (STGT) se muestran en la Figura 5.5. La columna relativa al puntero p tiene dos posibles valores $''as''$ o $''rs''$ dependiendo de a cuál de las dos pilas apunte: el valor $''as''$ para la pila de argumento y el valor $''rs''$ para la pila de retornos. Esto conlleva almacenar un puntero pS por cada argumento y por cada continuación. El espacio necesario para almacenar este puntero es menor que el necesario para almacenar el entero de la versión anterior, ya que estos punteros pueden representarse por un bit (0 para indicar $''as''$ y 1 para indicar "rs").

Esta máquina que llamaremos $STGT$ (de STG con tiempo) es correcta y completa con respecto a la semántica de Launchbury. Para demostrarlo basta dar una función de transformación inversa que dadas las tres pilas de la máquina $STGT$ produzca una única pila S para la máquina *STG-1S*.

Definición 36
Definimos la función $Trans' :: (Puntero, (Pila\_as, Pila\_rs, Pila\_us))_{STGT} \rightarrow Pila\_S_{STG-1S}$.

	Montón	Control	Entorno	P	as	rs	us	Regla
	Γ	$Eval\ x\ \overline{x_i}^n$	$E \cup [x \mapsto p, \overline{x_i \mapsto p_i}^n]$	pS	as	rs	us	1([1])
$\Longrightarrow$	Γ	$Enter\ p$	$\{\}$	pS'	as	rs	us	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e,\ E)]$	$Enter\ p$	$\{\}$	$''as''$	as	rs	us	2([2])
$\Longrightarrow$	Γ	$Eval\ e$	$E \cup [\overline{x_i \mapsto p_i}^n]$	pS	as'	rs	us	
	Γ	$Eval$ **letrec** $\{\overline{x_i = e_i \vert^{t_i}}\}$ **in** e	E	pS	as	rs	us	3([3])
$\Longrightarrow$	$\Gamma \cup [\overline{p_i \mapsto (e_i,\ E' \vert^{t_i})}]$	$Eval\ e$	E'	pS	as	rs	us	
	Γ	$Eval$ **case** e **of** $alts \vert^t$	E	pS	as	rs	us	4
$\Longrightarrow$	Γ	$Eval\ e$	E	$''rs''$	as	$((alts,\ E \vert^t),\ pS) : rs$	us	
	Γ	$Eval\ C\ \overline{x_i}$	$E \cup [\overline{x_i \mapsto p_i}]$	pS	as	rs	us	5
$\Longrightarrow$	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	pS	as	rs	us	
	Γ	$ReturnCon\ C_k\ \overline{p_i}$	$\{\}$	$''rs''$	as	$((alts,\ E),\ pS) : rs$	us	6([4])
$\Longrightarrow$	Γ	$Eval\ e_k$	$E \cup [\overline{y_{ki} \mapsto p_i}]$	pS	as	rs	us	
	$\Gamma \cup [p \mapsto (\lambda \overline{x_i}^0.e,\ E)]$	$Enter\ p$	$\{\}$	pS	as	rs	us	15([5])
$\Longrightarrow$	Γ	$Eval\ e$	E	pS	$\{\}$	$\{\}$	$(as,\ rs,\ p) : us$	
	Γ	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	pS	$\{\}$	$\{\}$	$(as,\ rs,\ p) : us$	16([6])
$\Longrightarrow$	$\Gamma \cup [p \mapsto (C\ \overline{x_i},\ [\overline{x_i \mapsto p_i}])]$	$ReturnCon\ C\ \overline{p_i}$	$\{\}$	pS	as	rs	us	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e,\ E)]$	$Enter\ p$	$\{\}$	$''as''$	as	$\{\}$	$(as',\ rs,\ p') : us$	17([7])
$\Longrightarrow$	Γ'	$Enter\ p$	$\{\}$	$''as''$	$as +\!\!+\ as'$	rs	us	
	$\Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e,\ E)]$	$Enter\ p$	$\{\}$	$''as''$	as	$\{\}$	$(as',\ rs,\ p') : us$	17a([8])
$\Longrightarrow$	Γ'	$Enter\ p$	$\{\}$	$''as''$	$as +\!\!+\ as'$	rs	us	

([1]) Si $n = 0$ entonces $pS' = pS$ en otro caso $pS' = ''as''$ y $as' = \overline{(p_i,\ ''as'')}^{n-1} : (p_n,\ pS) : as$.
([2]) $as = \overline{(p_i,\ ''as'')}^{n-1} : (p_n,\ pS) : as'$.
([3]) En la regla 3, las variables $\overline{p_i}$ son distintas y frescas. $E' = E \cup [\overline{x_i \mapsto p_i}]$
([4]) La expresión e_k corresponde a la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en *alts*
([5]) La expresión $e \neq C\ \overline{x_i}$
([6]) En la regla 16, las variables $\overline{x_i}$ son una secuencia de variables arbitrarias distintas.
([7]) $as = \overline{(p_i,\ ''as'')}^{k-1} : (p_k,\ pS_k)$ y $k < n$. Si $as = \emptyset$ entonces $pS' = pS$ sino $pS' = pS_k$ y $\Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e,\ E),\ p' \mapsto (\lambda \overline{x_i}^{n-k}.e,\ E \cup [\overline{x_i \mapsto p_i}^k])]$
([8]) $as = \overline{(p_i,\ ''as'')}^{k-1} : (p_k,\ pS_k)$ y $k < n$. Si $as = \emptyset$ entonces $pS' = pS$ sino $pS' = pS_k$ y $\Gamma' = \Gamma \cup [p \mapsto (\lambda^n \overline{x_i}.e,\ E),\ p' \mapsto (\lambda \overline{x_i}^0.x\ \overline{x_i}^k,\ [x \mapsto p, \overline{x_i \mapsto p_i}^k])]$

Figura 5.5: Máquina abstracta *STGT*

$$\begin{array}{rcl}
Trans' \, [\![pS,\, (as,\, rs,\, us)]\!] & = & Trans'Acu \, [\![pS,\, (as, rs, us)]\!] \; [\,] \\
Trans'Acu \, [\![''as'',\, ((p, pS) : as,\, rs,\, us]\!] \; S & = & Trans'Acu \, [\![pS,\, (as,\, rs,\, us)]\!] \; (S; p) \\
Trans'Acu \, [\![''as'',\, ([\,],\, rs,\, (p, as, rs') : us)]\!] \; S & = & Trans'Acu \, [\![''as'',\, (as,\, rs +\!\!+ \, rs',\, us)]\!] \; (S; \#p) \\
Trans'Acu \, [\![''rs'',\, (as,\, ((alts, E), pS) : rs,\, us)]\!] \; S & = & Trans'Acu \, [\![pS,\, (as,\, rs,\, us)]\!] \; (S; (alts, E)) \\
Trans'Acu \, [\![''rs'',\, (as,\, [\,],\, (p, as', rs) : us)]\!] \; S & = & Trans'Acu \, [\![''rs'',\, (as +\!\!+ \, as',\, rs,\, us)]\!] \; (S; \#p) \\
Trans'Acu \, [\![pS, ([\,],\, [\,],\, [\,])]\!] \; S & = & S
\end{array}$$

La transformación de las pilas as, rs y us partiendo del puntero pS corresponde con el cómputo $Trans' \, [\![pS,\, (as, rs, us)]\!]$. *La notación* $S; p$ *indica que en la pila* S *añadimos por el final el elemento p.*

Si hemos partido de una expresión cerrada y del cómputo

$$\begin{array}{rl}
& (\{\},\, Eval\ e,\, \{\}, \bot,\, [\,],\, [\,],\, [\,]) \\
\Rightarrow^* & (\Gamma,\, i,\, E,\, pS,\, as,\, rs,\, us)
\end{array}$$

donde $\bot$ puede valer cualquiera de los dos valores posibles para el puntero $''as''$ o $''rs''$, la transformación de la pila se realizará de forma exitosa, es equivalente a la que surgiría del cómputo de la máquina *STG-1S*

$$\begin{array}{rl}
& (\{\},\, Eval\ e,\, \{\},\, [\,]) \\
\Rightarrow^* & (\Gamma,\, i,\, E,\, S)
\end{array}$$

y no llegaremos a situaciones del tipo $Trans'Acu \, [\![''as'',\, ([\,],\, ((alts, E), pS) : rs,\, [\,])]\!] \; S'$ o del tipo $Trans'Acu \, [\![''rs'',\, ((p, pS) : as,\, [\,],\, [\,])]\!] \; S'$ que implicarían una construcción errónea en las pilas, esto se ve claramente observando las reglas de la máquina $STGT$ y *STG-1S*.

La demostración de la corrección de dicha transformación es trivial, ya que las reglas de ambas máquina son equivalentes. Ambas máquinas aplican las mismas reglas en el mismo orden, y la única diferencia radica en la pila, la *STG-1S* tiene una única pila, mientras que la $STGT$ tiene tres y un puntero, necesario para mantener el orden cronológico de las pilas.

El paso de la pila Sde la máquina *STG-1S* a las tres pilas (as, rs, us) y al puntero pS de la máquina $STGT$ se realiza de forma similar que el paso de ésta a las tres pilas de la máquina STG, pero teniendo en cuenta qué es lo último que se añadió a las pilas, si un argumento o un retorno, y colocando los punteros adecuados cuando se añade algo en alguna de las pilas *as* o *rs*.

La máquina *STG-1S* nos parece más sencilla que la máquina $STGT$ para derivar propiedades y para comparar con las máquinas de Sestoft. Además, es más eficiente en tiempo y en espacio, ya que no almacena los punteros y sólo necesita consultar la cima de la pila para saber cómo actuar. A partir de este momento no nos volveremos a preocupar de la división de la pila en tres y, en el Capítulo 6, generaremos código para la máquina *STG-1S*.

Capítulo 6

Generación de código

En este capítulo nos acercaremos lo máximo a la generación de código que realiza el *Glasgow Haskell Compiler* (*GHC*). Pretendemos con ello no sólo formalizar esta etapa del compilador que fue expuesta en [Pey92], sino también clarificarla, ya que a nuestro modo de ver es una de las etapas más confusas de dicho compilador. Además es el motivo más importante que ha llevado al éxito del compilador *GHC*, ya que en esta fase se realizan muchas optimizaciones y el código que genera es muy eficiente.

Para ello primeramente generaremos un código ingenuo sin optimizaciones, y posteriormente iremos incorporando sucesivamente diversas optimizaciones, acercándonos lo más posible a la generación que realiza *GHC*.

La idea es que cualquier lector pueda seguir el proceso de generación de código sin necesidad de tener que recurrir a la lectura de [PL92, PHH$^+$93, PMR99] donde la generación de código no está formalizada, o directamente al código del compilador [PMST01], ya que éstos son los únicos documentos disponibles relativos a la generación de código. Definiremos un conjunto de funciones mutuamente recursivas que generan código imperativo, en un estilo similar al empleado en [WM95].

6.1 Versión 1: Compilación con un entorno

Uno de los principales motivos del éxito del compilador *GHC* es la representación uniforme de las clausuras. Además, éstas no llevan de forma directa ninguna información relativa a su tamaño, ni al tipo de contenido, ni a la disposición espacial. De ahí el apellido "sin marcas" (tagless) que caracteriza a la máquina. La representación adoptada por dicho compilador es un par (p, ws) donde p es un puntero a una tabla de información en la que se tiene principalmente el código, tamaño, e información necesaria para el recolector de basura, y ws es una lista de punteros que corresponden a las variables libres de la clausura. La única parte de la tabla de información necesaria para la evaluación de la clausura es el código.

Nosotros adoptaremos por ahora la siguiente representación (p, ws) donde la única diferencia con la representación de las clausuras del *GHC* es que en lugar de ser p un puntero a una tabla de información va a ser directamente un puntero al código que hay que ejecutar cuando se entra en la clausura. La lista ws sigue representando las variables libres de la clausura.

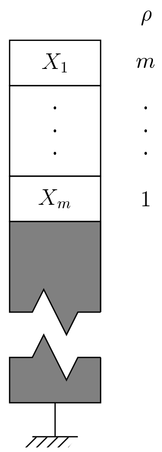

Figura 6.1: Situación del entorno en la pila.

6.1.1 Compilación

Para generar código utilizaremos las siguiente funciones:

$$\begin{array}{lcl} trdExpr & :: & Exp \to Env \to Int \to [Inst] \\ trdAlts & :: & Alts \to Env \to Int \to PCode \\ trdLig & :: & Int \to LForm \to Env \to Int \to (PCode, Int) \end{array}$$

Todas estas funciones necesitan un entorno $Env : Var \to Int$ y un entero m. Éstos son necesarios para saber las posiciones en ejecución, con respecto a la cima de la pila S de las variables libres de la expresión, alternativa o ligadura que están traduciendo. El entero indica el tamaño del entorno, es decir, el número de variables que contiene. La numeración de las variables del entorno se realiza en orden creciente, teniendo la última variable añadida al entorno el valor del tamaño del entorno y la primera el valor 1. Este entorno va a servir para eliminar los nombres de las variables al compilar las expresiones, de tal manera que realizaremos un cálculo parecido al que se hizo para De Bruijn. Queremos una numeración de variables libres relativa al *comienzo* del entorno que sea invariable aunque el entorno y la pila crezcan. Pero el código generado ha de acceder a las variables relativas a la cima de la pila, es decir al *final* del entorno. El cálculo apropiado es $posicion\_relativa\_a\_la\_cima = m - \rho\ x$. En la Figura 6.1 podemos ver una pila con el entorno constituido por las variables $\overline{x_i}^m$ donde la primera que se añadió al entorno fue la variable x_m y la última la x_1. Para evitar problemas, la expresión de partida será una expresión cerrada con todos los nombres de las variables ligadas distintos.

Las funciones *trdExpr* y *trdAlts* generan el código correspondiente a la traducción de una expresión o unas alternativas de un **case**. Para realizar esta generación de código necesitan un entorno y el tamaño del entorno representado aquí por un entero. Con este entorno se numeran las variables de la expresión. La función *trdLig* se encarga de la traducción de las ligaduras de las expresiones **letrec**. Además del entorno necesario para la numeración de las variables, necesita un entero que nos indica la posición relativa a la cima de la pila donde se encuentra en tiempo de ejecución el puntero a la clausura de la ligadura. Esta instrucción produce un puntero al código correspondiente a la ligadura almacenado en un almacén de código cs, junto con un entero que nos indica el tamaño de la clausura que genera esa ligadura.

Para la realización de la traducción es necesario tener un almacén de código cs donde guardar el código generado para cada ligadura de un **letrec**, o para el conjunto de alternativas de un **case**, puesto que en el primer caso se necesita crear clausuras con un puntero a código y en el segundo caso se necesita almacenar en la pila un conjunto de alternativas representadas por un puntero a código. Vamos a considerar que este almacén se mantiene de forma global a la hora de realizar las traducciones. Así, aparecerán en la traducción los efectos laterales que se produzcan sobre este almacén de código (de este modo se simplifica la presentación). Este almacén va a estar estructurado, pudiendo acceder al código de una ligadura a través de un puntero a código. Denotaremos el almacén mediante una función $cs : PCode \to Code$ donde $PCode$ es un puntero a una lista de instrucciones en el almacén de código, y $Code$ es una lista de instrucciones.

Una ligadura de un **letrec** dará lugar en ejecución a varias clausuras diferentes. Todas ellas compartirán el mismo código que será generado una sola vez, y diferirán en el conjunto de variables libres.

En tiempo de ejecución no se podrá escribir en el almacén de código, y todo el código necesario para la ejecución del programa se generará en tiempo de compilación.

6.1.2 Estado de la ejecución

El estado de ejecución es una quíntupla (is, S, nod, h, cs) donde is es una lista de instrucciones bajo ejecución (lista de instrucciones de la clausura que estamos reduciendo a forma normal); S es la pila donde se almacenan tres tipos de objetos diferentes: direcciones de las clausuras en el montón, direcciones del código de las alternativas y marcas de actualización; nod es un puntero a la dirección de la última clausura del montón en la que se ha entrado (la que está ahora mismo computándose); h es el montón (un almacén de clausuras con la estructura $h : PMon \to (PCode, FVars)$); y cs es el almacén de código creado en tiempo de compilación.

6.1.3 Notación

Utilizaremos la siguiente notación:

a, b, c, d	direcciones del montón ($\in PMon$)
p	direcciones de código ($\in PCode$)
$\#a$	marcas de actualización ($a \in PMon$)
$PUSH$	instrucciones de la máquina imperativa
$pushVar$	macros de compilación
i, j, k, l, m, n	enteros
x, y	variables del texto fuente ($\in String$)
as, ws	listas de punteros en el montón ($\in PMon^*$)
C_k^m	constructor de aridad m, k-ésimo de la declaración **data** del tipo algebraico
$S!i$	empezando por la cima, el elemento en la posición i de la pila S (El elemento en la posición 0 es la cima)
η	entorno de las variables que se encuentran en la clausura apuntada por nod

	(variables libres de la clausura)
ρ	entorno de las variables libres que se encuentran en la pila
tm	tamaño de la clausura a almacenar en el montón

La notación $h[a \mapsto (p, ws)]$ se utilizará, como ya se ha dicho, para indicar que en la dirección a del montón tenemos la clausura (p, ws), donde p es una dirección de código y ws son punteros a otras clausuras, y la notación $h \cup [a \mapsto (p, ws)]$ expresa que en la posición a del montón añadimos la clausura (p, ws). Si existía previamente una clausura en la posición a del montón, ésta se sustituye por la nueva. Esta misma notación, con una leve modificación, se utilizará para el almacén de código, de tal forma que la notación $cs \leftarrow cs \cup [p \mapsto is]$ significa que en cs añadimos las instrucciones is en la dirección p. En este caso nunca se sustituyen las instrucciones del almacén de código. Para ello nos aseguraremos de que las direcciones al código sean siempre nuevas. Por otro lado, la notación $cs[p\,] = is$ indica que en la posición p del almacén de código cs tenemos almacenadas las instrucciones is.

El tamaño de una clausura se representará con el espacio necesario para almacenar las variables libres de la clausura ignorando, para simplificar, el puntero a código de la misma. Como en nuestro lenguaje no tenemos enteros desencapsulados o tipos básicos, el tamaño de una clausura va a ser el número de variables libres de dicha clausura, ya que todas ellas corresponden con direcciones de código y, por tanto, ocupan el mismo espacio. Por ejemplo si el tamaño es 2 reservaremos espacio para una clausura de la forma $(p, [a1, a2])$

Por otro lado, nuestro lenguaje no tiene declaraciones de tipos de datos, y la notación C_k^m parece un poco confusa. Supondremos que hemos organizado los constructores por tipos y que les podemos asociar una etiqueta k que indicará el número de orden del constructor C dentro de la declaración de tipo.

Ejemplo 37
Sea un programa donde tenemos las listas y los naturales definidos de la siguiente forma.

$$\begin{array}{lcl} \textbf{data}\ List\ \alpha & = & \textbf{Nil} \mid \textbf{Cons}\ \alpha\ (List\ \alpha) \\ \textbf{data}\ Nat & = & \textbf{Cero} \mid \textbf{Succ}\ Nat \end{array}$$

El constructor **Nil** se representará por $\mathbf{C}_1^0$ y el constructor **Cons** se representará por $\mathbf{C}_2^2$. Por otro lado, los constructores **Cero** y **Succ** se representarán respectivamente mediante $\mathbf{C}_1^0$, $\mathbf{C}_2^1$.

□

Para que esta transformación sea correcta y el código generado realice las mismas reducciones que la máquina abstracta es necesario que el lenguaje esté bien tipado. Supondremos que tal comprobación se ha hecho previamente a esta fase.

También vamos a obligar a que las expresiones **case** tengan una alternativa para cada constructor del tipo de la expresión de control. Además, las alternativas vendrán ordenadas por el número de orden k del constructor. En caso de que previamente no tuviéramos alternativa para alguno de los constructores, generaremos una alternativa con una instrucción de $error$. Veamos un ejemplo de una transformación de una expresión **case**:

Ejemplo 38

La expresión:

$$\begin{array}{l}\textbf{case } Nil \textbf{ of}\\ \quad Nil \ \rightarrow Cero\end{array}$$

ha sido transformada en un paso previo en la siguiente:

$$\begin{array}{l}\textbf{case } Nil \textbf{ of}\\ \quad Nil \qquad\quad \rightarrow Cero\\ \quad Cons\ x\ y \rightarrow error\end{array}$$

Donde *error* es una expresión polimórfica cuya ejecución hará parar el cómputo.

□

Por otro lado, vamos a necesitar tener código prefabricado que se utilizará en tiempo de ejecución. Existen varios motivos para tener código prefabricado para ciertas clausuras. Uno de ellos es que en tiempo de ejecución se actualizan clausuras en el montón (véase regla 17a). Estas clausuras han de apuntar a un código que debe estar previamente generado ya que si no estaríamos generando código en tiempo de ejecución, lo cual sería bastante ineficiente. Ya que es necesario tener código prefabricado para ciertas clausuras, aprovecharemos esto para evitar repetir código en otros casos. Por ello tendremos código prefabricado para las ligaduras de la forma $C_k^m\ \overline{x_i}^m$, que se compartirá por todos los constructores de aridad m, k-ésimos en la declaración **data** del tipo. Otro código prefabricado necesario es el de un *agujero negro*. Esto es necesario ya que cuando en tiempo de ejecución entramos a una clausura actualizable, para evitar volver a entrar en ella hasta que no se haya evaluado hasta su forma normal, hacemos que su puntero a código apunte a un conjunto de instrucciones que llamaremos *agujero negro*. Si alguna vez entramos en una clausura de este tipo, lo que deseamos es que la ejecución se pare, por tanto, el código de un agujero negro debe ser una lista vacía de instrucciones.

Para esta primera versión, el código prefabricado necesario es el siguiente:

$$\begin{array}{lcl} cs[p_{C_k^l}] & = & [\mathit{UNPACK}\ l, \mathit{RETURNCON}\ C_k^l]\\ cs[p_{bh}] & = & [\,]\\ cs[p_{pap^m}] & = & [\mathit{UNPACK}\ m,\ \mathit{ENTER}]\end{array}$$

Donde $p_{C_k^l}$, p_{bh} y p_{pap^m} son los punteros a las posiciones donde se encuentra respectivamente el código generado para los constructores, el agujero negro y las aplicaciones parciales que surgen de la regla 17a de la máquina *STG-1S*.

6.1.4 Invariantes

Es necesario distinguir dos tipos de invariantes. Por un lado, los que se satisfacen en tiempo de ejecución y, por otro, los que se satisfacen en tiempo de compilación.
En tiempo de compilación se cumplen las propiedades siguientes:

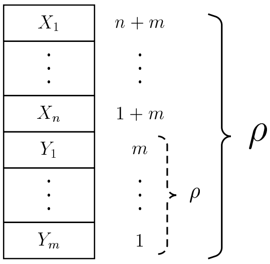

Figura 6.2: Entorno.

- m es el tamaño del entorno ρ en el que tenemos las variables libres de la expresión que estamos traduciendo. Las variables están en el orden especificado en la Sección 6.1.1.
- Al añadir al entorno ρ de tamaño m un conjunto de variables $\overline{x_i}^n$, éstas se añadirán en orden decreciente. Primero añadiremos siempre la x_n y en último lugar la x_1. Así, en tiempo de ejecución la variable x_1 se encontrará en la cima de la pila. Si este conjunto de variables constituyen un entorno consistente con el esquema de numeración descrito en 6.1.1, asignaremos $\rho\ x_1 = n+m, \ldots, \rho\ x_n = 1+m$ (ver Figura 6.2), pasando a tener un entorno de tamaño $n+m$.
- El código de toda clausura termina en una instrucción *RETURNCON* o *ENTER* y es la única vez que aparece en una clausura una instrucción de este tipo.

En tiempo de ejecución se van a cumplir las propiedades siguientes:

- Antes de entrar en una clausura, se encuentran en la pila los argumentos de la misma.
- Al entrar en una clausura se completa el entorno en la cima de la pila con las variables libres de la clausura. Por tanto, al evaluar una expresión con variables libres, todo el entorno necesario está en la cima de la pila.
- En la pila, debajo de cada puntero a código de alternativas, se encuentra el entorno de las alternativas. Los pares $(alts, \overline{[x_i \mapsto a_i]})$ de la pila S de la máquina *STG-1S* están representados en la pila de la siguiente forma $p : \overline{a_i}$, donde $cs\,[p]$ es el código de las alternativas $alts$.

6.1.5 Macros de compilación

Para la generación de código utilizaremos las siguientes macros:

$pushVar\ x\ \rho\ m =$
$\quad PUSHSTACK\ (m - \rho\ x)$

$buildEnv\ \overline{x_i}^n\ \rho\ m =$
$\quad [\ pushVar\ x_n\ \rho\ m,$
$\quad\ \ldots$

$$pushVar\ x_1\ \rho\ (m+n-1)]$$

buildEnv genera las instrucciones que construyen un entorno en la pila. Para ello se apoya en la macro *pushVar* que es la encargada de realizar el cálculo de la distancia de la variable x a la cima de la pila.

6.1.6 Traducción

En algunas reglas de traducción aparece el efecto lateral que se produce sobre el almacén de código, asociado mediante el símbolo &.

La definición de las funciones de traducción está escrita en Haskell, enriquecido con alguna notación matemática autoexplicativa.

$$
\begin{array}{llll}
\multicolumn{4}{l}{trdExpr\ (x\ \overline{x_i}^n)\ \rho\ m\ = is} \\
\quad \text{donde} & & & \\
\qquad is & = & buildEnv\ \overline{x_i}^n\ \rho\ m \mathbin{+\!\!+} & \\
 & & [pushVar\ x\ \rho\ (m+n), & \\
 & & \ SLIDE\ (n+1)\ m, & \text{--elimina el entorno en curso} \\
 & & \ ENTER] &
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{trdExpr\ (C\ \overline{x_i}^n)\ \rho\ m\ = is} \\
\quad \text{donde} & & & \\
\qquad is & = & buildEnv\ \overline{x_i}^n\ \rho\ m \mathbin{+\!\!+} & \\
 & & [SLIDE\ n\ m, & \text{--elimina el entorno en curso} \\
 & & \ RETURNCON\ C_k^n] & \text{--se ha alcanzado una forma normal}
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{trdExpr\ (\mathbf{case}\ e\ \mathbf{of}\ alts\ |^{\overline{x_i}^n})\ \rho\ m = is} \\
\quad \text{donde} & & & \\
\qquad p & = & trdAlts\ alts\ \rho'\ n & \\
\qquad \rho' & = & [\overline{(n+1-i)/x_i}^n] & \\
\qquad is & = & buildEnv\ \overline{x_i}^n\ \rho\ m \mathbin{+\!\!+} & \text{--genera entorno de las } alts \\
 & & [PUSHCODE\ p, & \\
 & & \ SWAP\ (n+1)\ m] \mathbin{+\!\!+} & \text{--intercambia entornos de las } alts \text{ y de } e \\
 & & \ trdExpr\ e\ \rho\ m &
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdExpr\ (\mathbf{letrec}\ \overline{x_i = lf_i\ |^{t_i}}^n\ \mathbf{in}\ e)\ \rho\ m = is} \\
\quad \text{donde} & & \\
\qquad is & = & [ALLOC\ tm_n, \ldots, ALLOC\ tm_1] \mathbin{+\!\!+} \\
 & & \ is_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} is_n \mathbin{+\!\!+} \\
 & & \ trdExpr\ e\ \rho'\ (m+n) \\
\qquad \rho' & = & \rho \cup [\overline{(m+n+1-i)/x_i}^n] \\
\qquad (is_i, tm_i) & = & trdLig\ (i-1)\ (lf_i\ |^{t_i})\ \rho'\ (m+n),\ \ i \in \{1 \ldots n\}
\end{array}
$$

$$trdAlts\ (\overline{alt_i}^n)\ \rho\ m = p \qquad \&\ \ cs \leftarrow cs \cup [p \mapsto \overline{is_i}^n]$$

donde

$$is_i \quad = \quad trdAlt\ alt_i\ \rho\ m, \ \ i \in \{1 \ldots n\}$$

$$trdAlt\ (C\ \overline{x_i}^n \to e)\ \rho\ m = is$$

donde

$$\begin{array}{lcl} is & = & trdExp\ e\ \rho'\ (m+n) \\ \rho' & = & \rho \cup [\overline{m+n+1-i/x_i}^n] \end{array}$$

$$trdLig\ j\ (C\ \overline{x_i}^n\ |^t)\ \rho\ m = (is, n) \qquad \&\ \ cs[p_{C_k^n}] = [UNPACK\ n, RETURNCON\ C_k^n]$$

donde

$$\begin{array}{lcl} is & = & buildEnv\ \overline{x_i}^n\ \rho\ m +\!\!+ \\ & & [PUSHCODE\ p_{C_k^n}, \\ & & \ PACK\ j\ n] \end{array}$$

$$trdLig\ j\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_i}^n})\ \rho\ m = (is', n) \quad \&\ \ cs \leftarrow cs \cup [p_e \mapsto ([ARGCHECK\ l,\ \ UNPACK\ n] +\!\!+ is)]$$

donde

$$\begin{array}{lcl} is' & = & buildEnv\ \overline{y_i}^n\ \rho\ m +\!\!+ \\ & & [PUSHCODE\ p_e \\ & & \ PACK\ j\ n] \\ \rho' & = & [\overline{l+n+1-i/y_i}^n] \cup [\overline{l+1-i/x_i}^l] \\ is & = & trdExpr\ e\ \rho'\ (l+n) \end{array}$$

$$trdLig\ j\ (e\ |^{\overline{x_i}^n})\ \rho\ m = (is', n) \qquad \&\ \ cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK,\ \ UNPACK\ n] +\!\!+ is)]$$

donde

$$\begin{array}{lcl} is' & = & buildEnv\ \overline{x_i}^n\ \rho\ m +\!\!+ \\ & & [PUSHCODE\ p_e \\ & & \ PACK\ j\ n] \\ \rho' & = & [\overline{n+1-i/x_i}^n] \\ is & = & trdExpr\ e\ \rho'\ n \end{array}$$

6.1.7 Observaciones

En esta primera versión las clausuras se generan previamente en la pila para posteriormente pasarlas al montón con la instrucción $PACK$.

La compilación de la instrucción **case** genera primero el entorno necesario para las alternativas, coloca el puntero al código de las alternativas y posteriormente intercambia los entornos con la instrucción $SWAP$.

La numeración de las variables no corresponde fielmente con la numeración de De Bruijn. Esto es debido a que los entornos se generan directamente sobre la pila. A medida que el entorno se va generando en la cima de la pila, las variables ya apiladas van modificando su posición relativa a la cima. Sin embargo, los entornos de la *Mark-2* se generaban como una estructura independiente de la pila.

6.1.8 Comportamiento de las instrucciones

En esta sección se da la semántica operacional de las instrucciones máquina en términos de la transición de estado que produce cada una, simbolizada por $\Rightarrow$.

Utilizaremos notación funcional. En consecuencia, $PUSHSTACK\ i : is$ debe entenderse como $(PUSHSTACK\ i) : is$, etc.

	Instrucciones	Pila	Nodo	Montón	Código	
	$PUSHSTACK\ i : is$	S	nod	h	cs	
$\Longrightarrow$	is	$S!i : S$	nod	h	cs	
	$PUSHCODE\ p : is$	S	nod	h	cs	
$\Longrightarrow$	is	$p : S$	nod	h	cs	
	$ALLOC\ m : is$	S	nod	h	cs	(1)
$\Longrightarrow$	is	$a_m : S$	nod	h'	cs	
	$PACK\ j\ m : is$	$p : \overline{a_i}^m : S$	nod	h	cs	
$\Longrightarrow$	is	S	nod	$h[S!j \mapsto (p, \overline{a_i}^m)]$	cs	
	$UNPACK\ m : is$	S	nod	$h[nod \mapsto (p, \overline{a_i}^m)]$	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	h	cs	
	$SLIDE\ m\ n : is$	$\overline{a_i}^m : \overline{b_j}^n : S$	nod	h	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	h	cs	
	$SWAP\ m\ n : is$	$\overline{a_i}^m : \overline{b_j}^n : S$	nod	h	cs	
$\Longrightarrow$	is	$\overline{b_j}^n : \overline{a_i}^m : S$	nod	h	cs	
	$[ENTER]$	$a : S$	nod	$h[a \mapsto (p, ws)]$	$cs[p \mapsto is]$	
$\Longrightarrow$	is	S	a	h	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : p : S$	nod	h	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow$	is_k	$\overline{a_i}^m : S$	nod	h	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : \#a : S$	nod	$h[a \mapsto (p_{bh}, ws)]$	cs	(2)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : S$	nod	$h[a \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	cs	
	$UPDTMARK : is$	S	nod	$h[nod \mapsto (p, ws)]$	cs	(3)
$\Longrightarrow$	is	$\#nod : S$	nod	$h[nod \mapsto (p_{bh}, ws)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^m : S$	nod	h	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	h	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	$h[a \mapsto (p_{bh}, ws)]$	cs	(4)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	$h[a \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	

(1) a_m es una nueva dirección en el montón donde entra una clausura con m variables libres y h' es el montón resultante después de la reserva.

(2) $cs[p_{C_k^m}] = [UNPACK\ m, RETURNCON\ C_k^m]$

(3) $cs[p_{bh}] = [\,]$

(4) $m > n,\ cs[p_{pap^m}] = [UNPACK\ m,\ ENTER]$

6.1.9 Mejora de la traducción de una ligadura tipo $x\ \overline{x_i}^n$

Una mejora interesante es la traducción de una ligadura del tipo $x\ \overline{x_i}^n$. El orden de sus variables libres influye y hace que sea ineficiente. La traducción de una ligadura de este tipo implica el siguiente código:
$[\mathit{UPDTMARK},\ \mathit{UNPACK}\ (n+1), \mathit{PUSHSTACK}\ ((n+2)-\rho\ x_n), \ldots, \mathit{PUSHSTACK}\ ((n+2)-\rho\ x_1), \mathit{PUSHSTACK}\ ((n+2)-\rho\ x), \mathit{SLIDE}\ (n+1)\ (n+1), \mathit{ENTER}]$
Esta traducción es necesaria ya que nadie nos asegura que el conjunto de variables libres esté ordenado de una forma concreta en el conjunto de poda. Por lo tanto, se generan instrucciones para colocarlas ordenadas en la cima de la pila y además es necesario borrar el entorno previamente generado, cuando lo lógico sería realizar sólo un *UNPACK* $(n+1)$.

Esto se puede solucionar traduciendo las ligaduras de este tipo de forma distinta, y se puede aprovechar y generar un código prefabricado para las clausuras de este tipo. Es posible generar la clausura para una expresión de este tipo almacenando sus variables libres $x\ \overline{x_i}^n$ en ese orden. Primero la x, seguida de las x_i, para que al realizar un *UNPACK* $(n+1)$ queden en el orden adecuado en la pila. Esto se realiza en la siguiente traducción:

$$
\begin{array}{l}
\mathit{trdLig}\ j\ (x\ \overline{x_i}^n\ |^{ws})\ \rho\ m = (is, n+1)\ \ \&\ \ cs[p_{ap^{(n+1)}}] = [\mathit{UPDTMARK},\ \mathit{UNPACK}\ (n+1),\ \mathit{ENTER}] \\
\quad \text{donde} \\
\qquad is \qquad = \mathit{buildEnv}\ [x : \overline{x_i}^n]\ \rho\ m +\!\!+ \\
\qquad\qquad\qquad [\mathit{PUSHCODE}\ p_{ap^{(n+1)}} \\
\qquad\qquad\qquad \mathit{PACK}\ j\ (n+1)]
\end{array}
$$

Cualquier ligadura consistente en una aplicación de una variable a otras n compartirá el siguiente código:

$$cs[p_{ap^m}] = [\mathit{UPDTMARK},\ \mathit{UNPACK}\ m,\ \mathit{ENTER}]$$

donde $m = n+1$.

6.2 Versión 2: Eliminación de la instrucción *PACK*

Hasta ahora, hemos generado las clausuras previamente en la pila para luego pasarlas al montón utilizando la instrucción *PACK*. La mejora principal en esta versión es generarlas directamente sobre el montón, eliminando con ello dicha instrucción. Con este fin tendremos dos instrucciones nuevas, *COPYSTACKHEAP*, y *COPYCODEHEAP*. La primera instrucción copiará las variables libres a una clausura previamente reservada en el montón, y la segunda instrucción copiará el puntero al código de dicha clausura.

La instrucción *COPYSTACKHEAP* tiene el formato *COPYSTACKHEAP* $i\ j\ k$, donde i indica la posición en la pila donde se encuentra el puntero a la clausura en la que ha de colocar la variable libre, j indica el número de orden de la variable libre que hay que copiar y k indica la posición en la pila del argumento que hay que copiar en la posición j de la clausura. Nótese que las variables libres de una clausura se numeran de 1 a n. Una clausura de tamaño n tiene la forma $(p, [a_1, \ldots, a_n])$ donde los subíndices indican la numeración de las variables libres.

La instrucción *COPYCODEHEAP* sólo necesitará un entero y un puntero a código. Tendrá la forma *COPYCODEHEAP* i p, donde el i indica la posición en la pila donde se encuentra la dirección de la clausura en la que se va a colocar el puntero de código p. Como el puntero es siempre lo primero que hay en la clausura, no es necesario ningún otro parámetro en esta instrucción.

6.2.1 Invariantes

Se mantienen los invariantes de la versión 1 (ver Sección 6.1.4).

6.2.2 Nuevas macros de compilación

En esta versión son necesarias dos nuevas macros de compilación:

$$\begin{array}{l} \mathit{pushHeap}\ i\ j\ x\ \rho\ m = \\ \quad \mathit{COPYSTACKHEAP}\ i\ j\ (m - \rho\ x) \\[1em] \mathit{buildHeapEnv}\ i\ \overline{x_i}^n\ \rho\ m = \\ \quad [\ \mathit{pushHeap}\ i\ 1\ x_1\ \rho\ m, \\ \quad\ \ \ldots \\ \quad\ \ \mathit{pushHeap}\ i\ n\ x_n\ \rho\ m] \end{array}$$

La macro *buildHeapEnv*, al igual que *buildEnv* (Sección 6.1.5), genera un entorno pero, en lugar de hacerlo sobre la pila, lo hace directamente sobre una clausura reservada en el montón.

6.2.3 Traducción

Para llevar a cabo estas modificaciones cambiamos la traducción de las ligaduras de los **letrec**.

$$\begin{array}{l} \mathit{trdLig}\ j\ (C\ \overline{x_i}^n\ |^{ws})\ \rho\ m = (is, n) \qquad \&\ \ cs[p_{C_k^n}] = [\mathit{UNPACK}\ n, \mathit{RETURNCON}\ C_k^n] \\ \quad \text{donde} \\ \qquad \begin{array}{lcl} is & = & \mathit{buildHeapEnv}\ j\ \overline{x_i}^n\ \rho\ m +\!\!+ \\ & & [\mathit{COPYCODEHEAP}\ j\ p_{C_k^n}] \end{array} \end{array}$$

$$\begin{array}{l} \mathit{trdLig}\ j\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_i}^n})\ \rho\ m = (is', n) \qquad \&\ \ cs \leftarrow cs \cup [p_e \mapsto ([\mathit{ARGCHECK}\ l,\ \mathit{UNPACK}\ n] +\!\!+ is)] \\ \quad \text{donde} \\ \qquad \begin{array}{lcl} is' & = & \mathit{buildHeapEnv}\ j\ \overline{y_i}^n\ \rho\ m +\!\!+ \\ & & [\mathit{COPYCODEHEAP}\ j\ p_e] \\ \rho' & = & [\overline{l + n + 1 - i / y_i}^n] \cup [\overline{l + 1 - i / x_i}^l] \end{array} \end{array}$$

$$
\begin{array}{lll}
\quad is & = & trdExpr\ e\ \rho'\ (l+n) \\
 & & \\
trdLig\ j\ (x\ \overline{x_i}^n\ |^t)\ \rho\ m = (is, n+1) & \& & cs[p_{ap^{(n+1)}}] = [UPDTMARK, UNPACK\ (n+1), ENTER] \\
\quad \text{donde} & & \\
\quad is & = & buildHeapEnv\ j\ [x, \overline{x_i}^n]\ \rho\ m \mathbin{+\!\!+} \\
 & & [COPYCODEHEAP\ j\ p_{ap^{n+1}}] \\
 & & \\
trdLig\ j\ (e\ |^{\overline{x_i}^n})\ \rho\ m = (is', n) & \& & cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK,\ UNPACK\ n] \mathbin{+\!\!+} is)] \\
\quad \text{donde} & & \\
\quad is' & = & buildHeapEnv\ j\ \overline{x_i}^n\ \rho\ m \mathbin{+\!\!+} \\
 & & [COPYCODEHEAP\ p_e] \\
\quad \rho' & = & [\overline{n+1-i/x_i}^n] \\
\quad is & = & trdExpr\ e\ \rho'\ n
\end{array}
$$

6.2.4 Comportamiento de las instrucciones

La instrucción *PACK* desaparece y se sustituye por las siguientes instrucciones:

	Instrucciones	Pila	Nodo	Montón	Código
	$COPYSTACKHEAP\ i\ j\ k : is$	S	nod	$h[S!i \mapsto (p, a_1 \ldots a_{j-1}, ?, a_{j+1} \ldots a_l)]$	cs
$\Longrightarrow$	is	S	nod	$h[S!i \mapsto (p, a_1 \ldots a_{j-1}, S!k, a_{j+1} \ldots a_l)]$	cs
	$COPYCODEHEAP\ i\ p : is$	S	nod	$h[S!i \mapsto (?, ws)]$	cs
$\Longrightarrow$	is	S	nod	$h[S!i \mapsto (p, ws)]$	cs

6.3 Versión 3: Compilación con dos entornos

Hasta ahora, cada vez que la máquina entra en una clausura, realiza un volcado de todas sus variables libres a la pila, con la instrucción *UNPACK*. Cuando está evaluando una clausura, el puntero *nod* está apuntando a la clausura. *GHC* aprovecha *nod* para no realizar el volcado a la pila. Hasta ahora este puntero sólo se utiliza para colocar una marca de actualización en el caso de que la clausura sea actualizable (véase Sección 6.1.8).

Las variables de la clausura actualmente en evaluación pueden mantenerse en el montón cuando estamos ejecutando el código de dicha clausura. Con ello se elimina bastante trasiego entre la pila y el montón. Así, cuando se necesita acceder a una variable libre de la clausura en evaluación, se hará a través de *nod*. Para ello, es necesario compilar con un nuevo entorno que es el correspondiente al entorno accesible desde *nod*, y una nueva instrucción para acceder a las variables libres de la clausura apuntada desde *nod*. Esta instrucción es *PUSHNODE* j, que colocará la variable libre j de la clausura apuntada por *nod* en la cima de la pila. Su comportamiento es similar a la instrucción *PUSHSTACK* (veasé Sección 6.1.8), la única diferencia es que la coloca desde *nod* en lugar de desde la pila.

Por otro lado, como las ligaduras ahora generan las clausuras directamente sobre el montón, puede que la variable de la clausura que estamos creando en el montón no se encuentre en la pila, sino que aún se encuentre en el montón. Por tanto, es necesario una nueva instrucción que copie desde la clausura apuntada por *nod* a la clausura que está generando la ligadura. Esta instrucción es $COPYNODEHEAP\ i\ j\ k$, donde i indica la posición en la pila del puntero de la clausura que estamos generando, j identifica la variable libre destino y k identifica la variable libre origen.

6.3.1 Compilación

Las funciones de generación de código van a ser las siguientes:

$$\begin{array}{lll} trdExpr & :: & Exp \to Env \to Int \to Env \to [Inst] \\ trdAlts & :: & Alts \to Env \to Int \to Env \to PCode \\ trdLig & :: & Int \to LForm \to Env \to Int \to Env \to (PCode, Int) \end{array}$$

La modificación con respecto a las que aparecen en la Sección 6.1.1 es que se les ha añadido un nuevo entorno que denotaremos η. Corresponde a las variables accesibles desde *nod*, y a diferencia de ρ, del cual necesitamos saber el tamaño para calcular la posición de la variable dada con respecto a la cima, este entorno llevará asociadas las variables accesibles desde *nod* con la posición que les corresponde directamente dentro de la clausura.

6.3.2 Invariantes

Además de los invariantes de la Sección 6.1.4 ahora se cumple que:

- η siempre tiene las variables accesibles desde la clausura apuntada por *nod*, numeradas de 1 a n.

6.3.3 Macros de compilación

Las macros de compilación reflejan ahora el hecho de que las variables libres pueden estar o bien en la pila o bien accesibles a través de *nod*, es decir, respectivamente en el entorno ρ o en el entorno η.

$$pushVar\ x\ \rho\ m\ \eta = \begin{cases} PUSHSTACK\ (m - \rho\ x) & \text{si } x \in dom\ \rho \\ PUSHNODE\ (\eta\ x) & \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildEnv\ \overline{x_i}^n\ \rho\ m\ \eta = \\ \quad [\ pushVar\ x_n\ \rho\ m\ \eta, \\ \quad\quad \ldots \\ \quad\quad pushVar\ x_1\ \rho\ (m + n - 1)\ \eta] \end{array}$$

$$pushHeap\ i\ j\ x\ \rho\ m\ \eta = \begin{cases} COPYSTACKHEAP\ i\ j\ (m - \rho\ x) & \text{si } x \in dom\ \rho \\ COPYNODEHEAP\ i\ j\ (\eta\ x) & \text{en otro caso} \end{cases}$$

$$
\begin{array}{l}
buildHeapEnv\ i\ \overline{x_i}^n\ \rho\ m\ \eta = \\
\quad [\ pushHeap\ i\ 1\ x_1\ \rho\ m\ \eta, \\
\quad\ \ \ldots \\
\quad\ \ pushHeap\ i\ n\ x_n\ \rho\ m\ \eta]
\end{array}
$$

6.3.4 Traducción

Es necesario incorporar el nuevo entorno η. Gran parte de la misma no sufre modificaciones exceptuando la incorporación del entorno. Las reglas de traducción que más cambian son las relacionadas con las ligaduras de los **letrec**.

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdExpr\ (x\ \overline{x_i}^n)\ \rho\ m\ \eta = is} \\
\quad \text{donde} & & \\
\qquad is & = & buildEnv\ \overline{x_i}^n\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [pushVar\ x\ \rho\ (m+n)\ \eta, \\
 & & \ SLIDE\ (n+1)\ m, \\
 & & \ ENTER]
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdExpr\ (C\ \overline{x_i}^n)\ \rho\ m\ \eta = is} \\
\quad \text{donde} & & \\
\qquad is & = & buildEnv\ \overline{x_i}^n\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [SLIDE\ n\ m, \\
 & & \ RETURNCON\ C_k^n]
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdExpr\ (\mathbf{case}\ e\ \mathbf{of}\ alts\ |^{\overline{x_i}^n})\ \rho\ m\ \eta = is} \\
\quad \text{donde} & & \\
\qquad p & = & trdAlts\ alts\ \rho'\ n\ \{\} \\
\qquad \rho' & = & [\overline{(n+1-i)/x_i}^n] \\
\qquad is & = & buildEnv\ \overline{x_i}^n\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [PUSHCODE\ p, \\
 & & \ SWAP\ (n+1)\ m] \mathbin{+\!\!+} \\
 & & \ trdExpr\ e\ \rho\ m\ \eta
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdExpr\ (\mathbf{letrec}\ \overline{x_i = lf_i\ |^{t_i}}^n\ \mathbf{in}\ e)\ \rho\ m\ \eta = is} \\
\quad \text{donde} & & \\
\qquad is & = & [ALLOC\ tm_n, \ldots, ALLOC\ tm_1] \mathbin{+\!\!+} \\
 & & \ is_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} is_n \mathbin{+\!\!+} \\
 & & \ trdExpr\ e\ \rho'\ (m+n)\ \eta \\
\qquad \rho' & = & \rho \cup [\overline{(m+n+1-i)/x_i}^n] \\
\qquad (is_i, tm_i) & = & trdLig\ (i-1)\ (lf_i\ |^{t_i})\ \rho'\ (m+n)\ \eta,\ i \in \{1 \ldots n\}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{trdAlts\ (\overline{alt_i}^n)\ \rho\ m\ \eta = p} & \&\quad cs \leftarrow cs \cup [p \mapsto \overline{is_i}^n] \\
\quad \text{donde} & & \\
\qquad is_i & = & trdAlt\ alt_i\ \rho\ m\ \eta,\ i \in \{1 \ldots n\}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdAlt\ (C\ \overline{x_i}^n \rightarrow e)\ \rho\ m\ \eta = is} \\
\quad \text{donde} & & \\
\qquad is & = & tradExp\ e\ \rho'\ (m+n)\ \eta \\
\qquad \rho' & = & \rho \cup [\overline{m+n+1-i/x_i}^n]
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{trdLig\ j\ (C\ \overline{x_i}^n\ |^{ws})\ \rho\ m\ \eta = (is, n)} & \&\quad cs[p_{C_k^n}] = [UNPACK\ n, RETURNCON\ C_k^n] \\
\quad \text{donde} & & \\
\qquad is & = & buildHeapEnv\ j\ \overline{x_i}^n\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [COPYCODEHEAP\ j\ p_{C_k^n}]
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{trdLig\ j\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_i}^n})\ \rho\ m\ \eta = (is', n)} & \&\quad cs \leftarrow cs \cup [p_e \mapsto ([ARGCHECK\ l] \mathbin{+\!\!+} is)] \\
\quad \text{donde} & & \\
\qquad is' & = & buildHeapEnv\ j\ \overline{y_i}^n\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [COPYCODEHEAP\ j\ p_e] \\
\qquad \eta' & = & [\overline{i/y_i}^n] \\
\qquad \rho' & = & [\overline{l+1-i/x_i}^l] \\
\qquad is & = & trdExpr\ e\ \rho'\ l\ \eta'
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{trdLig\ j\ (x\ \overline{x_i}^n\ |^t)\ \rho\ m\ \eta = (is, n+1)\ \&\ cs[p_{ap^{(n+1)}}] = [UPDTMARK, UNPACK\ (n+1), ENTER]} \\
\quad \text{donde} & & \\
\qquad is & = & buildHeapEnv\ j\ [x, \overline{x_i}^n]\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [COPYCODEHEAP\ j\ p_{ap^{(n+1)}}]
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{trdLig\ j\ (e\ |^{\overline{x_i}^n})\ \rho\ m\ \eta = (is', n)} & \&\quad cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK] \mathbin{+\!\!+} is)] \\
\quad \text{donde} & & \\
\qquad is' & = & buildHeapEnv\ j\ \overline{x_i}^n\ \rho\ m\ \eta \mathbin{+\!\!+} \\
 & & [COPYCODEHEAP\ j\ p_e] \\
\qquad \eta' & = & [\overline{i/x_i}^n] \\
\qquad is & = & trdExpr\ e\ \{\ \}\ 0\ \eta'
\end{array}
$$

6.3.5 Comportamiento de las instrucciones

El comportamiento de las dos instrucciones nuevas que hemos incorporado *PUSHNODE* y *COPYNODEHEAP* es el siguiente:

	Instrucciones	Pila	Nodo	Montón	Código
	$PUSHNODE\ i : is$	S	nod	$h[nod \mapsto (p, ws)]$	cs
$\Longrightarrow$	is	$ws!i : S$	nod	h	cs
	$COPYNODEHEAP\ i\ j\ k : is$	S	nod	$h[nod \mapsto (p, ws),\ S!i \mapsto (p, a_1 \dots a_{j-1}, ?, a_{j+1} \dots a_l)]$	cs
$\Longrightarrow$	is	S	nod	$h[S!i \mapsto (p, a_1 \dots a_{j-1}, ws!k, a_{j+1} \dots a_l)]$	cs

A partir de ahora la instrucción $UNPACK\ m$ se puede ver como una macro de compilación ya que es equivalente a $PUSHNODE\ m \dots PUSHNODE\ 1$. Según las posibilidades de optimización de una u otra, se mantendrá o no dicha instrucción.

6.3.6 Observaciones

Hemos logrado evitar la realización de las instrucciones $UNPACK$ para casi todas las clausuras. También sería interesante evitar esto para las clausuras de tipo constructor, pero en tiempo de compilación no sabemos cuándo un constructor proviene del montón o viene directamente de una expresión. Por lo tanto, no sabemos si tiene o no generada la clausura en el montón. Se podría obligar a que los constructores generen siempre sus clausuras en el montón, como se realizará en la Sección 6.7.

6.4 Versión 4: Compilación con una pila de entornos

En esta versión la mejora principal es la desaparición de la instrucción $SWAP$, que es una de las instrucciones más caras de la máquina.

La instrucción $SWAP$ surge en la traducción de la expresión **case**. En ejecución se apila un puntero al código correspondiente a las alternativas. Debajo de este puntero a código hemos de tener el entorno de las alternativas, ya que cuando la máquina llegue a un constructor y se ponga a evaluar las alternativas necesita ese entorno. Encima de ese puntero hemos de tener también un entorno, ya que el discriminante del **case** lo necesita. Hasta ahora lo que hace la máquina es generar un entorno independiente para las alternativas y para el discriminante del **case**. La traducción que se propone va a utilizar el mismo entorno para ambos. Ahora bien, aparecen dos problemas:

1. Se tendrá más residencia en memoria, ya que no se poda el entorno de las alternativas.

2. Hay que tener en cuenta que ese entorno no hay que borrarlo, y que está situado tras un puntero a código.

El primer problema no se puede solucionar por completo. Una solución parcial sería llevar las expresiones **case** completas anotadas con sus variables libres, y dejar sólo ese entorno en la cima de la pila. La solución es parcial porque puede suceder que el discriminante del **case** tenga variables libres que no se utilicen dentro de las alternativas y, por tanto, ésas se mantendrían en el entorno hasta que se evalúen las alternativas. No obstante se puede anotar cada alternativa con sus variables libres, y así cuando nos pongamos a evaluar la alternativa correspondiente lo primero que realizaremos es una poda sobre el entorno. Estas optimizaciones las realiza el *GHC*, pero debido a que perderíamos claridad no las añadiremos en la generación de código.

El segundo problema se puede solucionar complicando un poco más la traducción. En lugar de llevar un entorno podemos llevar una lista de entornos con sus tamaños. De tal manera que entre entorno y entorno lo que se encontrará en ejecución es un puntero al código de las alternativas. Cuando tengamos que eliminar un entorno lo que haremos será eliminar el último entorno que se ha generado. El último entorno generado siempre se mantiene en la cima de la pila.

Las ventajas principales de esta compilación son: la eliminación de la instrucción $SWAP$ en la instrucción **case**, y que no se duplican entornos para la expresión del discriminante y las alternativas del **case**.

6.4.1 Compilación

La modificación principal en las funciones de generación de código es que ahora en lugar de llevar un entorno y un entero que nos indique el tamaño del entorno, llevaremos una lista de entornos con sus tamaños de manera que el que esté en la cabeza de la lista (es decir, en la cima de la pila de entornos) es el que vamos a quitar en el caso de eliminación, y es en el que vamos a añadir las variables cuando sea necesario:

$$\begin{array}{lll} trdExpr & :: & Exp \to [(Env, Int)] \to Env \to [Inst] \\ trdAlts & :: & Alts \to [(Env, Int)] \to Env \to PCode \\ trdLig & :: & Int \to LForm \to [(Env, Int)] \to Env \to (PCode, Int) \end{array}$$

En la Figura 6.3 podemos observar como se encuentran estos entornos en la pila S.

6.4.2 Invariantes

En tiempo de compilación se cumplen las propiedades siguientes:

- $\forall j.j \in \{1 \dots k\}$, m_j es el tamaño del entorno ρ_j. En $[\overline{(\rho_j, m_j)}^k]$ tenemos las variables libres de la expresión que estamos traduciendo. Las variables están en el orden especificado en la Sección 6.4.1.
- Al añadir al entorno ρ de tamaño m un conjunto de variables $\overline{x_i}^n$, éstas se añadirán en orden decreciente. Primero añadiremos siempre la x_n y en último lugar la x_1. Así, en tiempo de ejecución la variable x_1 se encontrará en la cima de la pila. Si este conjunto de variables constituyen un entorno consistente con el esquema de numeración descrito en 6.1.1, asignaremos $\rho\ x_1 = n + m, \dots, \rho\ x_n = 1 + m$ (ver Figura 6.2), pasando a tener un entorno de tamaño $n + m$.
- El código de toda clausura termina en una instrucción $RETURNCON$ o $ENTER$ y es la única vez que aparece en una clausura una instrucción de este tipo.
- En η siempre se encuentran las variables libres accesibles en tiempo de ejecución desde *nod*, incluyendo los argumentos de un constructor.

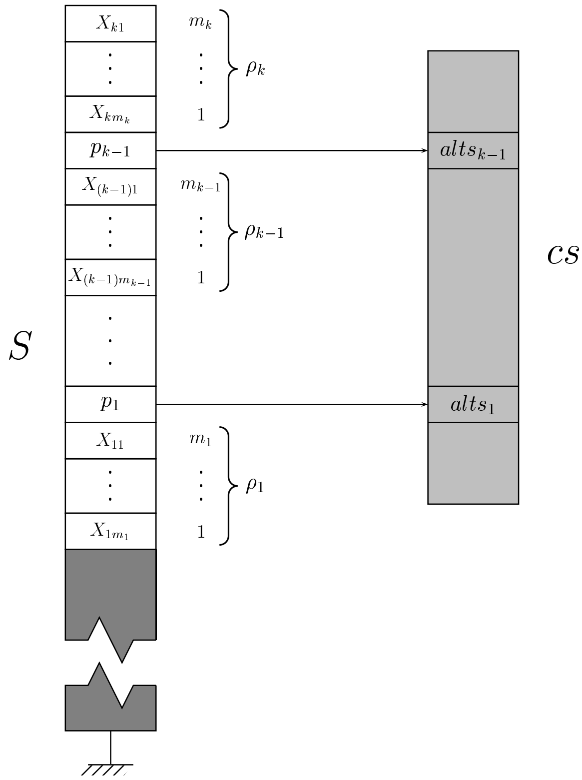

Figura 6.3: Situación del entorno en la pila.

- Una variable no está ligada en dos entornos distintos, debido a que la expresión cerrada de partida tiene todos los nombres de variables distintos.

En tiempo de ejecución se van a cumplir las propiedades siguientes:

- Antes de entrar en una clausura se encuentran en la pila los argumentos de la misma.
- Al entrar en una clausura, todo el entorno necesario está en la cima de la pila, o en el montón apuntado desde *nod*.
- En la pila, debajo de cada puntero a código de alternativas, se encuentra el entorno de las alternativas. Los pares $(alts, \overline{[x_i \mapsto a_i]})$ de la pila S de la máquina *STG-1S* están representados en la pila de la siguiente forma $p : \overline{b_l}^{h_k} : p_{k-1} : \overline{b_l}^{h_{k-1}} : \ldots : p_1 : \overline{b_l}^{h_1}$, donde *cs* $[p]$ es el código de las alternativas *alts*, $\forall j.j \in \{1 \ldots k\}$, $cs[p_j] = alts_j$, y entre los b_l se encuentran los a_i. (Ver Figura 6.3)
- Los entornos se encuentran en la pila y, entre entorno y entorno, se encuentra un puntero a las alternativas de un **case**. (Ver Figura 6.3)
- *nod* apunta a la clausura actualmente en evaluación.

6.4.3 Macros de compilación

$[\overline{(\rho_l, m_l)}^k]$ representa la lista $[(\rho_k, m_k) \ldots (\rho_1, m_1)]$ numerada de forma decreciente por conveniencia.

$$pushVar\ x\ [\overline{(\rho_l, m_l)}^k]\ \eta = \begin{cases} PUSHSTACK\ (\sum_{n=l}^{k} m_n + (k-l) - \rho_l\ x) & \text{si } \exists l.(x \in dom\ \rho_l) \\ PUSHNODE\ (\eta\ x) & \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildEnv\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta = \\ \quad [\ pushVar\ x_n\ [\overline{(\rho_l, m_l)}^k]\ \eta, \\ \quad\ \ pushVar\ x_{n-1}\ [(\rho_k, m_k+1), \overline{(\rho_l, m_l)}^{k-1}]\ \eta, \\ \quad\ \ \ldots \\ \quad\ \ pushVar\ x_1\ [(\rho_k, m_k+n-1), \overline{(\rho_l, m_l)}^{k-1}]\ \eta\ ] \end{array}$$

$$pushHeap\ i\ j\ x\ [\overline{(\rho_l, m_l)}^k]\ \eta = \begin{cases} COPYSTACKHEAP\ i\ j\ (\sum_{n=l}^{k} m_n + (k-l) - \rho_l\ x), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{si } \exists l.(x \in dom\ \rho_l) \\ COPYNODEHEAP\ i\ j\ (\eta\ x), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildHeapEnv\ i\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta = \\ \quad [\ pushHeap\ i\ 1\ x_1\ [\overline{(\rho_l, m_l)}^k]\ \eta, \\ \quad\ \ \ldots \\ \quad\ \ pushHeap\ i\ n\ x_n\ [\overline{(\rho_l, m_l)}^k]\ \eta\ ] \end{array}$$

$$pushVarAlt\ x\ [\overline{(\rho_l, m_l)}^k]\ \eta = \begin{cases} ([\,], (\rho_k, m_k)) & \text{si } \exists l.(x \in dom\ \rho_l) \\ ([PUSHNODE\ (\eta\ x)], (\rho_k \cup [m_k + 1/x], m_k + 1)) & \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildEnvAlt\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, (\rho_1, m_1)) \\ \quad \text{donde} \\ \qquad is = is_n +\!\!+ \ldots +\!\!+ is_1 \\ \qquad (is_i, (\rho_i, m_i)) = pushVarAlt\ x_i\ [(\rho_{i+1}, m_{i+1}),\ \overline{(\rho_l, m_l)}^{k-1}]\ \eta, \quad i \in \{1 \ldots n\} \\ \qquad (\rho_{n+1}, m_{n+1}) = (\rho_k, m_k) \end{array}$$

Por un lado las macros de compilación se han complicado un poco más ya que, no sólo hay que buscar en qué entorno se encuentra la variable x, sino que el cálculo de la distancia hasta la cima de la pila se complica bastante ya que hay que tener en cuenta los tamaños de los entornos que existen en la lista de entornos antes de encontrar el que contiene la variable x. Además, entre un entorno y otro de la pila S, existirá un puntero de alternativas. Todo ello hay que tenerlo en cuenta para calcular la distancia de la variable a la cima de la pila (ver Figura 6.3).

Por otro lado, ahora necesitamos dos macros más: *buildEnvAlt* y *pushVarAlt*. Estas macros generan código para salvar en la pila las variables volátiles, es decir, las que se encuentran en el entorno η. Por tanto, devuelve el nuevo entorno ρ modificado, que contiene esas variables libres accesibles en la pila.

6.4.4 Traducción

Obsérvese que la traducción de la expresión discriminante del **case** se realiza en el entorno modificado. El lector habrá observado que ahora η y ρ_k pueden tener ligada una misma variable, ya que hemos salvado las variables volátiles en la pila. Esto no genera ningún problema, ya que las macros de compilación lo primero que hacen es mirar si una variable está definida en los entornos de la pila. Hemos decidido compilarlo de esta forma porque así se puede ahorrar trasiego entre el montón y la pila. Si la expresión e fuera otro **case**, no sería necesario volcar otra vez desde *nod* las variables volátiles necesarias para el anterior.

$$
\begin{array}{l}
\mathit{trdExpr}\ (x\ \overline{x_i}^n)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is \\
\quad \text{donde} \\
\qquad \begin{array}{lcl}
is & = & \mathit{buildEnv}\ [x, \overline{x_i}^n]\ [\overline{(\rho_l, m_l)}^k]\ \eta \mathbin{+\!\!+} \\
 & & [\mathit{SLIDE}\ (n+1)\ m_k, \\
 & & \ \mathit{ENTER}]
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{trdExpr}\ (C\ \overline{x_i}^n)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is \\
\quad \text{donde} \\
\qquad \begin{array}{lcl}
is & = & \mathit{buildEnv}\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta \mathbin{+\!\!+} \\
 & & [\mathit{SLIDE}\ n\ m_k, \\
 & & \ \mathit{RETURNCON}\ C_k^n]
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{trdExpr}\ (\mathbf{case}\ e\ \mathbf{of}\ alts\ |^{\overline{x_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = is \\
\quad \text{donde} \\
\qquad \begin{array}{lcll}
p & = & \mathit{trdAlts}\ alts\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \{\} & \\
(is', (\rho'_k, m'_k)) & = & \mathit{buildEnvAlt}\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta & \\
is & = & is' \mathbin{+\!\!+} & \text{-- entorno volátil.} \\
 & & [\mathit{PUSHCODE}\ p] \mathbin{+\!\!+}, & \\
 & & \mathit{trdExpr}\ e\ [(\{\ \}, 0), (\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta &
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{trdExpr}\ (\mathbf{letrec}\ \overline{x_i = lf_i\ |^{t_i}}^n\ \mathbf{in}\ e)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is \\
\quad \text{donde} \\
\qquad \begin{array}{lcl}
is & = & [\mathit{ALLOC}\ tm_n, \ldots, \mathit{ALLOC}\ tm_1] \mathbin{+\!\!+} \\
 & & \ is_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} is_n \mathbin{+\!\!+} \\
 & & \ \mathit{trdExpr}\ e\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta \\
(\rho'_k, m'_k) & = & (\rho_k \cup [\overline{(m_k + n + 1 - i)/x_i}^n], m_k + n)
\end{array}
\end{array}
$$

$$
\begin{array}{lll}
(is_i, tm_i) & = & trdLig\ (i-1)\ (lf_i\ |^{t_i})\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta, \qquad i \in \{1 \dots n\}
\end{array}
$$

$trdAlts\ (\overline{alt_i}^n)\ [\overline{(\rho_l, m_l)}^k]\ \eta = p \qquad \&\ cs \leftarrow cs \cup [p \mapsto \overline{is_i}^n]$

donde

$$
\begin{array}{lll}
is_i & = & trdAlt\ alt_i\ [\overline{(\rho_l, m_l)}^k]\ \eta, \qquad i \in \{1 \dots n\}
\end{array}
$$

$trdAlt\ (C\ \overline{x_i}^n \rightarrow e)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is$

donde

$$
\begin{array}{lll}
is & = & tradExp\ e\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta \\
(\rho'_k, m'_k) & = & (\rho_k \cup [\overline{(m_k + n + 1 - i)/x_i}^n], m_k + n)
\end{array}
$$

$trdLig\ j\ (C\ \overline{x_i}^n\ |^t)\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, n) \qquad \&\ cs[p_{C_k^n}] = [UNPACK\ n, RETURNCON\ C_k^n]$

donde

$$
\begin{array}{lll}
is & = & buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_{C_k^n}]
\end{array}
$$

$trdLig\ j\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', n) \quad \&\ cs \leftarrow cs \cup [p_e \mapsto ([ARGCHECK\ l] +\!\!+ is)]$

donde

$$
\begin{array}{lll}
is' & = & buildHeapEnv\ j\ \overline{y_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_e] \\
\eta' & = & [\overline{i/y_i}^n] \\
\rho' & = & [\overline{l + 1 - i/x_i}^l] \\
is & = & trdExpr\ e\ [(\rho', l)]\ \eta'
\end{array}
$$

$trdLig\ j\ (x\ \overline{x_i}^n\ |^t)\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, n+1) \ \ \&\ cs[p_{ap^{(n+1)}}] = [UPDTMARK, UNPACK\ (n+1), ENTER]$

donde

$$
\begin{array}{lll}
is & = & buildHeapEnv\ j\ [x, \overline{x_i}^n]\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_{ap^{n+1}}]
\end{array}
$$

$trdLig\ j\ (e\ |^{\overline{x_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', n) \qquad \&\ cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK] +\!\!+ is)]$

donde

$$
\begin{array}{lll}
is' & = & buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_e] \\
\eta' & = & [\overline{i/x_i}^n] \\
is & = & trdExpr\ e\ [(\{\ \}, 0)]\ \eta'
\end{array}
$$

6.4.5 Comportamiento de las instrucciones

La instrucción $SWAP$ no es necesaria, ya que en compilación no se genera dicha instrucción. No es necesaria ninguna instrucción nueva.

6.5 Versión 5: Añadiendo indirecciones en las clausuras

Otro punto interesante, del que hasta ahora no nos hemos preocupado, son las actualizaciones. Al realizar una actualización en una clausura, puede ocurrir que las clausuras con las que estamos actualizando sean mayores o menores que la clausura que había en esa posición. El problema surge cuando la clausura con la que estamos actualizando es mayor que la que existe, ya que puede que estemos escribiendo en posiciones de memoria ocupadas por otras clausuras.

En esta versión vamos a generar las actualizaciones de forma indirecta. Es decir, antes de realizar una actualización reservaremos una posición nueva en el montón, y la clausura antigua apuntará de forma indirecta a la nueva dirección en el montón donde se ha colocado la actualización. Para ello la clausura antigua tiene que sobrescribirse con una clausura que se encargue de entrar en la actualizada. El siguiente código [*PUSHNODE 1*, *ENTER*] realiza una entrada en la clausura actualizada si tiene como variable libre la dirección de la nueva clausura. Este código es común para todas las clausuras que se actualicen en el montón y, por lo tanto, no es necesario generarlo de forma dinámica. Debido a esto, también podemos tenerlo almacenado desde el inicio en el almacén de código en una posición que denotaremos por p_{ind}, de tal forma que, ahora, $cs[p_{ind}] = [PUSHNODE\ 1,\ ENTER]$.

Esto conlleva un poco más de retraso ya que ahora para entrar en una clausura actualizada lo hacemos de forma indirecta. El código que se generará a lo máximo que llegará es a una indirección simple, de tal forma que el problema no será grave.

Por otro lado, para la realización de indirecciones tenemos que asegurar que en cualquier clausura actualizable quepa una clausura de tipo indirección. Con nuestra convención de tamaños las clausuras actualizables tiene que tener como mínimo el tamaño 1. Así, cuando hacemos la traducción a instrucciones máquina en una ligadura de un **letrec** que sea actualizable tenemos que tenerlo en cuenta. De tal manera que ahora el tamaño de ese tipo de ligaduras será el máximo entre 1 y el espacio que necesitaban antes. Esto supone pérdida de espacio, ya que existen clausuras actualizables sin variables libres, que son aquellas clausuras relativas a expresiones que no son formas normales y son cerradas, o las relativas a constructores de aridad 0. No obstante, la pérdida de tamaño es mínima.

Surge otro problema que es el de los punteros de actualización contiguos en ejecución en la pila S. Se puede actualizar cada clausura con el método anterior, pero entonces por cada puntero de actualización en la pila se generaría una nueva clausura. Hay una opción mejor: si hay varias marcas de actualización contiguas en la pila se puede generar una clausura para la primera y hacer que todas apunten de forma indirecta a ésta. De esta forma se consiguen indirecciones como mucho de nivel 1.

La compilación en esta versión tendrá en cuenta las dos ideas. Posteriormente, en la versión 6, se verá cómo conseguir que si la clausura cabe en donde estaba la vieja, entonces se actualice directamente sobre ella.

Para tener en cuenta todas estas ideas, en esta versión de la máquina imperativa aparece un nuevo puntero que indica la dirección de la clausura que se crea cuando se actualiza por primera vez, a las que las demás apuntan de forma indirecta. Este puntero irá emparejado con el tamaño de la clausura, para así poder utilizarlo con las clausuras que se actualizan a aplicaciones parciales.

6.5.1 Invariantes

Se mantienen los invariantes de la Sección 6.4.2 y además en tiempo de ejecución se cumple:

- Siempre se actualiza en el montón con indirecciones.

6.5.2 Traducción

La traducción se sigue manteniendo, excepto para las ligaduras de una expresión **letrec** que son actualizables y cuyo tamaño podía ser cero. La traducción es la siguiente.

$$trdLig\ j\ (e\ |^{\overline{x_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', max\ 1\ n) \qquad \& \quad cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK] +\!\!+\ is)]$$

donde

$$\begin{array}{lcl} is' & = & buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\ & & [COPYCODEHEAP\ j\ p_e] \\ \eta' & = & [\overline{i/x_i}^n] \\ is & = & trdExpr\ e\ [(\{\ \}, 0)]\ \eta' \end{array}$$

donde max es la función que devuelve el máximo de dos valores.

6.5.3 Comportamiento de las instrucciones

Ya que las actualizaciones se realizan en ejecución, los cambios más importantes se producen en el comportamiento de las instrucciones que realizan las actualizaciones. Éstas son $RETURNCON$ y $ARGCHECK$.

Ahora llevamos en el par *UpdtPtr* un nuevo puntero b junto con el tamaño n de la clausura actualizada, que apunta a la clausura con la que se ha hecho la primera actualización. Utilizamos la notación $\perp$ para indicar que el valor correspondiente está indefinido, y la notación ? para indicar que no es relevante lo que allí se encuentre.

	Instrucciones	Pila	Nodo	UpdtPtr	Montón	Código	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : p : S$	nod	$(?,?)$	h	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow$	is_k	$\overline{a_i}^m : S$	nod	$(\perp,?)$	h	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : \#a : S$	nod	$(\perp,?)$	$h[a \mapsto (p_{bh}, ws)]$	cs	(1)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : S$	nod	$(b,?)$	$h'[a \mapsto (p_{ind}, b), b \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : \#a : S$	nod	$(b,?)$	$h[a \mapsto (p_{bh}, ws)]$	cs	
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : S$	nod	$(b,?)$	$h[a \mapsto (p_{ind}, b)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^m : S$	nod	$(?,?)$	h	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	$(\perp,?)$	h	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	$(\perp,?)$	$h[a \mapsto (p_{bh}, ws)]$	cs	(2)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b,n)	$h'[a \mapsto (p_{ind}, b), b \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	(b,n)	$h[a \mapsto (p_{bh}, ws)]$	cs	(3)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b,n)	$h[a \mapsto (p_{ind}, b)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	(b',n')	$h[a \mapsto (p_{bh}, ws)]$	cs	(4)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b,n)	$h[a \mapsto (p_{ind}, b), b \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	

(1) donde b es una nueva dirección en el montón donde entra una clausura con m variables libres y h' es el montón resultante después de la reserva.
(2) donde b es una nueva dirección en el montón donde entra una clausura con m variables libres h' es el montón resultante después de la reserva, y $n < m$.
(3) $n < m$
(4) $n < m$ y $n' < n$

El comportamiento de las instrucciones depende directamente de lo que nos encontremos en el puntero *UpdtPtr*, de tal forma que si b está indefinido generamos una clausura nueva en el montón. Para ello reservamos previamente espacio en él, dejamos el puntero *UpdtPtr* apuntando a la nueva clausura generada en el montón y realizamos una actualización de forma indirecta a esa clausura. Si b está previamente definido, realizamos una indirección. En el caso de que la indirección se tenga que realizar con una aplicación parcial, previamente comprobamos si el tamaño de la indirección antigua corresponde con los argumentos que tenemos en la pila. En caso afirmativo, realizamos una indirección y en caso negativo, actuamos igual que si estuviera indefinido.

6.6 Versión 6: Controlando el tamaño de las clausuras

Por ahora la máquina realiza actualizaciones por indirección, aún cuando la clausura con la que actualizamos quepa en la clausura a actualizar.

Según los perfiles[1] de la *STG*, gran parte de las actualizaciones con constructores se realiza *in situ*. Por ello parece interesante generarlas *in situ* cuando la clausura nueva quepa en la vieja. Sólo se generarán por indirección cuando no quepa. Para ello es necesario controlar el tamaño de las clausuras.

Uno de los éxitos de la máquina *STG* era que no tenía información del tamaño de las clausuras, y que por tanto no tenía que controlar esa información para ejecutar las instrucciones. Ahora es necesario tener esa información para realizar actualizaciones *in situ*, pero no es necesaria para la ejecución del resto de instrucciones, ya que la traducción que se ha realizado no la necesita. Se

[1] En inglés profiling.

puede representar esta información directamente sobre la clausura, o de forma indirecta, de tal manera que no aparezca en la misma. Si esta información aparece de forma directa, la clausura pasaría a ser de la forma $(p, n, \overline{x_i}^n)$, donde la segunda componente es el tamaño. Esto conlleva mucho más espacio ocupado en el montón de forma innecesaria, ya que cada clausura que se crea en el montón llevaría su tamaño. Por otro lado, la información del tamaño de las clausuras se conoce en tiempo de compilación, por tanto parece innecesario almacenarla en el montón. Los únicos problemas que podrían surgir son con las clausuras con las que actualizamos, pero para estas clausuras generamos el código en tiempo de compilación, y por tanto, podemos emparejarlo con el tamaño que la corresponda.

Por los motivos anteriores y para acercarnos un poco más a la representación de las clausuras en la máquina STG, la información relativa al tamaño de la clausura se llevará de forma indirecta, y no sobre la clausura. Por tanto, ahora la representación de la clausura es de la siguiente forma: (p, ws) donde p es un puntero a un par de la forma (is, n), is es el código de la clausura, n es el tamaño de la clausura y ws sigue representando las variables libres de la clausura.

Además, al igual que hasta ahora las clausuras que en tiempo de ejecución provenían de una misma ligadura compartían el código de dicha ligadura ahora van a compartir el par (is, n).

Para ello, ahora el almacén de código va a sufrir una pequeña modificación, y en lugar de almacenar sólo el código, va a almacenar el código junto con el tamaño correspondiente de la clausura que tenga ese código, en formato de pares de la forma (is, n). Cada par corresponderá en tiempo de ejecución a una o varias clausuras. Las instrucciones relativas a las alternativas de un **case** se siguen manteniendo como una tupla de listas de instrucciones. Por lo tanto, en el almacén de código tenemos dos tipos diferentes de objetos, pares (is, n) y tuplas $\overline{is_i}^n$ correspondientes a las alternativas de un **case**.

6.6.1 Invariantes

Se mantienen los invariantes de la Sección 6.4.2 y además en tiempo de ejecución se cumple:

- Siempre que se actualiza una clausura, si la clausura nueva entra en el montón en donde se encontraba la antigua se actualiza *in situ*. En otro caso, se actualiza a través de una indirección.
- Toda clausura lleva de forma indirecta el tamaño de dicha clausura.

6.6.2 Traducción

La traducción necesita modificarse para que refleje que en el almacén de código ahora guardamos pares del tipo (is, n). Por tanto, las reglas modificadas son aquellas que añadían instrucciones en el almacén de código. Éstas son las siguientes:

$$\begin{array}{lll}
\multicolumn{3}{l}{trdLig\ j\ (C\ \overline{x_i}^n\ |^t)\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, n) \qquad \&\ cs[p_{C_k^n}] = ([UNPACK\ n, RETURNCON\ C_k^n], n)} \\
\quad \text{donde} & & \\
\qquad is & = & buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_{C_k^n}]
\end{array}$$

$$trdLig\ j\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', n)\ \ \&\ \ cs \leftarrow cs \cup [p_e \mapsto ([ARGCHECK\ l] +\!\!+\ is, n)]$$

donde

$$\begin{array}{lll} is' & = & buildHeapEnv\ j\ \overline{y_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\ & & [COPYCODEHEAP\ j\ p_e] \\ \eta' & = & [\overline{i/y_i}^n] \\ \rho' & = & [\overline{l+1-i/x_i}^l] \\ is & = & trdExpr\ e\ [(\rho', l)]\ \eta' \end{array}$$

$$trdLig\ j\ (x\ \overline{x_i}^n\ |^t)\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, n+1)$$
$$\&\ \ cs[p_{ap^{(n+1)}}] = ([UPDTMARK, UNPACK\ (n+1), ENTER], n+1)$$

donde

$$\begin{array}{lll} is & = & buildHeap\ j\ [x, \overline{x_i}^n]\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\ & & [COPYCODEHEAP\ j\ p_{ap^{(n+1)}}] \end{array}$$

$$trdLig\ j\ (e\ |^{\overline{x_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', max\ 1\ n)\qquad \&\quad cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK] +\!\!+\ is, n)]$$

donde

$$\begin{array}{lll} is' & = & buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta +\!\!+ \\ & & [COPYCODEHEAP\ j\ p_e] \\ \eta' & = & [\overline{i/x_i}^n] \\ is & = & trdExpr\ e\ [(\{\ \}, 0)]\ \eta' \end{array}$$

6.6.3 Comportamiento de las instrucciones

Tenemos que tener en cuenta que ahora el código está en un par. Para ello, todas las instrucciones que se referían al código han de reflejar el nuevo cambio. Las instrucciones que tienen modificación en el comportamiento son aquellas relativas a las actualizaciones, ya que ahora controlarán el tamaño de la clausura y siempre que puedan harán la actualización *in situ*.

Por otro lado, ahora necesitamos un agujero negro para cada tamaño de clausura. Dado que no queremos perder la información del tamaño que tenía la clausura porque, si es posible, vamos a utilizarlo para generarlas *in situ*. Luego el código prefabricado necesario es el siguiente:

$$\begin{array}{lll} cs[p_{bh^m}] & = & ([\,], m) \\ cs[p_{ind}] & = & ([PUSHNODE\ 1,\ ENTER], 1) \\ cs[p_{ap^m}] & = & ([UPDTMARK, UNPACK\ m,\ ENTER], m) \\ cs[p_{pap^m}] & = & ([UNPACK\ m,\ ENTER], m) \\ cs[p_{C_k^m}] & = & ([UNPACK\ m,\ RETURNCON\ C_k^m], m) \end{array}$$

El comportamiento de las instrucciones es el que se expresa a continuación:

	Instrucciones	Pila	Nodo	UpdtPtr	Montón	Código	
	$[ENTER]$	$a : S$	nod	$(?, ?)$	$h[a \mapsto (p, ws)]$	$cs[p \mapsto (is, l)]$	
$\Longrightarrow$	is	S	a	$(?, ?)$	h	cs	
	$UPDTMARK : is$	S	nod	$(?, ?)$	$h[nod \mapsto (p, ws)]$	$cs[p \mapsto (is, m)]$	
$\Longrightarrow$	is	$\#nod : S$	nod	$(?, ?)$	$h[nod \mapsto (p_{bh^m}, ws)]$	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : \#a : S$	nod	$(\perp, ?)$	$h[a \mapsto (p_{bh^n}, ws)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(1)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : S$	nod	$(b, ?)$	$h'[a \mapsto (p_{ind}, b), b \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : \#a : S$	nod	$(b, ?)$	$h[a \mapsto (p_{bh^n}, ws)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(2)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : S$	nod	$(b, ?)$	$h[a \mapsto (p_{ind}, b)]$	cs	
	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : \#a : S$	nod	$(?, ?)$	$h[a \mapsto (p_{bh^n}, ws)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(3)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	$\overline{a_i}^m : S$	nod	$(a, ?)$	$h[a \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	$(\perp, ?)$	$h[a \mapsto (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(4)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b, n)	$h'[a \mapsto (p_{ind}, b), b \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	(b, n)	$h[a \mapsto (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(5)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b, n)	$h[a \mapsto (p_{ind}, b)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	(b', n')	$h[a \mapsto (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(6)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b, n)	$h[a \mapsto (p_{ind}, b), b \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	$(?, ?)$	$h[a \mapsto (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(7)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(a, n)	$h[a \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	

(1) donde b es una nueva dirección en el montón donde entra una clausura con m variables libres
h' es el montón resultante después de la reserva, y $n < m$.
(2) $n < m$
(3) $m \leq n$
(4) donde b es una nueva dirección en el montón donde entra una clausura con m variables libres
h' es el montón resultante después de la reserva, $n < m$, y $l < n$.
(5) $m < n$ y $l < n$
(6) $n < m$ y $n' < n$ y $l < n$
(7) $n < m$ y $n \leq l$

Las modificaciones importantes se encuentran en las instrucciones *ARGCHECK* y *RETURNCON* que reflejan el hecho de que si una clausura cabe en la clausura a actualizar, se actualiza directamente sobre ella.

6.7 Versión 7: Eliminación completa de *UNPACK* en constructores

Hasta ahora los constructores siempre se devolvían en la pila y, por tanto, no se aprovechaba de ninguna forma si éstos venían o no de una clausura previamente generada en el montón. Si supiéramos esta información no sería necesario desempaquetar los argumentos de los constructores, ya que podríamos acceder a sus variables libres a través de *nod*, de tal forma que evitaríamos trasiego innecesario, y además sólo accederíamos desde *nod* a las variables del constructor que fueran necesarias en la alternativa concreta.

La última versión del *GHC* (versión 5.0) obliga a que todos los constructores se devuelvan en el montón de tal forma que se pueda aprovechar esta información para realizar una generación de código mejor. Así las clausuras de tipo constructor no necesitan volcar sus argumentos en la cima de la pila, y por tanto nos ahorramos el paso de los argumentos del montón a la pila, y el paso posterior de los argumentos de la pila al montón cuando realizamos las actualizaciones. De tal manera que ahora nunca será necesario generar una clausura nueva cuando tengamos que hacer una actualización, ya que el constructor venía del montón, y debido a esto podemos realizar la actualización con una indirección a la clausura que había en el montón. Dicha clausura estará apuntada por *nod*.

Por otro lado, esta obligación aparentemente conlleva un poco más de código y lo que es peor algo más de tiempo en ejecución, ya que tenemos que generar las clausuras en el montón. Sin embargo no es cierto que conlleve más tiempo, ya que hasta ahora una expresión de tipo constructor tenía que generar su entorno en la pila. Esto implica el mismo esfuerzo que generarlo en el montón, así que no se pierde nada. Es más, se gana, ya que *nod* apunta a la clausura del constructor que estamos evaluando, no se vuelcan las variables libres en la pila y no se generan las indirecciones a *nod* (luego no se crea una clausura nueva para ninguna actualización con constructor). Esto se verá con más claridad en el apartado 6.7.3.

Conseguir que los constructores se devuelvan en el montón se puede realizar de dos formas:

1. Seguir con el mismo lenguaje y, al traducir una expresión de tipo constructor que no esté en una ligadura de un **letrec**, generar código para que se cree una clausura en el montón, y pase al estado correspondiente.

2. Restringir el lenguaje de tal forma que sólo se permitan los constructores en las ligaduras de un **letrec**. Al igual que hicimos con las lambda abstracciones, esto no supone ninguna pérdida expresiva en el lenguaje, ya que cualquier expresión antigua puede ser traducida a una en este lenguaje sin más que ligar la expresión de tipo constructor en un **letrec**. De hecho esto es lo que hace la última versión de *GHC*.

¿Cuál es la más interesante? Ambas son completamente equivalentes y generan un código parecido. No obstante, y para acercarnos más a *GHC*, vamos a quedarnos con la segunda opción, para ello las expresiones de tipo constructor van a ser transformadas en nuestro lenguaje.

6.7.1 Invariantes

Se mantienen los invariantes de la Sección 6.6.1 y ahora además se cumple que:

- Siempre que se evalúa una expresión $C_k^n\ \overline{x_i}^n$, *nod* apunta a una clausura con esta expresión.
- En η siempre se encuentran las variables libres accesibles en tiempo de ejecución desde *nod*, incluyendo los argumentos de un constructor.

6.7.2 Traducción

Como hemos decidido obligar a que todos los constructores estén en una ligadura de una expresión **letrec**, la regla de traducción de una expresión constructor desaparece. En su lugar aparece la siguiente transformación en el lenguaje:

$C_k^n\ \overline{x_i}$ se transforma a **letrec**

$y = C_k^n\ \overline{x_i}^n$

in

y

La traducción de las alternativas cambia, ya que ahora tenemos en el entorno η las variables del constructor.

$$
\begin{array}{l}
trdAlt\ (C\ \overline{x_i}^n \rightarrow e)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is \\
\quad \text{donde} \\
\qquad \begin{array}{lcl}
is & = & tradExpr\ e\ [\overline{(\rho_l, m_l)}^k]\ \eta' \\
\eta' & = & [\overline{i/x_i}^n]
\end{array}
\end{array}
$$

Por otro lado, el código prefabricado para una clausura de tipo constructor debe reflejar este cambio y no debe desempaquetar las variables del constructor. Por tanto, el código es el siguiente:

$$cs[p_{C_k^m}] \quad = \quad ([RETURNCON\ C_k^m], m)$$

6.7.3 Comportamiento de las instrucciones

La instrucción que cambia de comportamiento es la instrucción $RETURNCON\ C_k^m$ que ahora no tendrá en la cima de la pila las variables del constructor, y por otro lado tampoco necesitará el puntero *UpdtPtr* ya que cuando entramos a una clausura de este tipo, *nod* estará apuntando a la clausura del constructor y, por tanto, podemos generar las indirecciones utilizando *nod*.

	Instrucciones	Pila	Nodo	UpdtPtr	Montón	Código	
	$[RETURNCON\ C_k^m]$	$p : S$	nod	$(?,?)$	h	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow$	is_k	S	nod	$(?,?)$	h	cs	
	$[RETURNCON\ C_k^m]$	$\#a : S$	nod	$(?,?)$	$h[a \mapsto (p_{bh^n}, ws)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(1)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	S	nod	$(?,?)$	$h[a \mapsto (p_{ind}, nod)]$	cs	
	$[RETURNCON\ C_k^m]$	$\#a : S$	nod	$(?,?)$	$h[a \mapsto (p_{bh^n}, ws), nod \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(2)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	S	nod	$(?,?)$	$h[a \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	cs	

(1) $n < m$
(2) $m \leq n$

Con esta modificación hemos conseguido eliminar el desempaquetamiento de las variables libres de una clausura del tipo constructor. Ahora sólo se desempaqueta una clausura cuando es estrictamente necesario, mientras que en los demás casos se accede a sus variables libres a través de *nod*. Con ello hemos conseguido eliminar mucho trasiego entre el montón y la pila.

6.8 Versión final con todas las mejoras

En esta versión presentamos la versión definitiva de la generación de código. Aún difiere del código generado por *GHC* pero la distancia es bastante pequeña y se debe a ciertas optimizaciones y a diferencias de representación tanto de la pila como de las clausuras.

6.8.1 Invariantes

En tiempo de compilación se cumplen las propiedades siguientes:

- $\forall j.j \in \{1 \ldots k\}$, m_j es el tamaño del entorno ρ_j. En $[\overline{(\rho_j, m_j)}^k]$ tenemos las variables libres de la expresión que estamos traduciendo. Las variables están en el orden especificado en la Sección 6.4.1.

- Al añadir al entorno ρ de tamaño m un conjunto de variables $\overline{x_i}^n$, éstas se añadirán en orden decreciente. Primero añadiremos siempre la x_n y en último lugar la x_1. Así, en tiempo de ejecución la variable x_1 se encontrará en la cima de la pila. Si este conjunto de variables constituyen un entorno consecuente con el esquema de numeración descrito en 6.1.1, asignaremos $\rho\ x_1 = n + m, \ldots, \rho\ x_n = 1 + m$ (ver Figura 6.2), pasando a tener un entorno de tamaño $n + m$.

- El código de toda clausura termina en una instrucción *RETURNCON* o *ENTER* y es la única vez que aparece en una clausura una instrucción de este tipo.

- En η siempre se encuentran las variables libres accesibles en tiempo de ejecución desde *nod*, incluyendo los argumentos de un constructor.

- Una variable no está ligada en dos entornos distintos, debido a que la expresión cerrada de partida tiene todos los nombres de variables distintos.

En tiempo de ejecución se van a cumplir las propiedades siguientes:

- Antes de entrar en una clausura, se encuentran en la pila los argumentos de la misma.

- Al entrar en una clausura todo el entorno necesario está en la cima de la pila, o en el montón apuntado desde *nod*.

- En la pila, debajo de cada puntero a código de alternativas, se encuentra el entorno de las alternativas. Los pares $(alts, \overline{[x_i \mapsto a_i]})$ de la pila S de la máquina *STG-1S* están representados en la pila de la siguiente forma $p : \overline{b_l}^{h_k} : p_{k-1} : \overline{b_l}^{h_{k-1}} : \ldots : p_1 : \overline{b_l}^{h_1}$, donde $cs\ [p]$ es el código de las alternativas $alts$, $\forall j.j \in \{1 \ldots k\}$, $cs[p_j] = alts_j$, y entre los b_l se encuentran los a_i. (Ver Figura 6.3)

- Los entornos se encuentran en la pila y, entre entorno y entorno, se encuentra un puntero a las alternativas de un **case**. (Ver Figura 6.3)

- *nod* apunta a la clausura actualmente en evaluación.

- Siempre que se actualiza una clausura, si la clausura nueva entra en el montón en donde se encontraba la antigua se actualiza *in situ*. En otro caso, se actualiza a través de una indirección.

- Toda clausura lleva de forma indirecta el tamaño de dicha clausura.

6.8.2 Macros de compilación

Las macros necesarias para la generación de código son las siguientes:

$$pushVar\ x\ [\overline{(\rho_l, m_l)}^k]\ \eta = \begin{cases} PUSHSTACK\ (\sum_{n=l}^{k} m_n + (k-l) - \rho_l\ x) & \text{si } \exists l.(x \in dom\ \rho_l) \\ PUSHNODE\ (\eta\ x) & \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildEnv\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta = \\ \quad [\ pushVar\ x_n\ [\overline{(\rho_l, m_l)}^k]\ \eta, \\ \quad\ \ pushVar\ x_{n-1}\ [(\rho_k, m_k+1), \overline{(\rho_l, m_l)}^{k-1}]\ \eta, \\ \quad\ \ \ldots \\ \quad\ \ pushVar\ x_1\ [(\rho_k, m_k+n-1), \overline{(\rho_l, m_l)}^{k-1}]\ \eta\ ] \end{array}$$

$$pushHeap\ i\ j\ x\ [\overline{(\rho_l, m_l)}^k]\ \eta = \begin{cases} COPYSTACKHEAP\ i\ j\ (\sum_{n=l}^{k} m_n + (k-l) - \rho_l\ x), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{si } \exists l.(x \in dom\ \rho_l) \\ COPYNODEHEAP\ i\ j\ (\eta\ x), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildHeapEnv\ i\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta = \\ \quad [\ pushHeap\ i\ 1\ x_1\ [\overline{(\rho_l, m_l)}^k]\ \eta, \\ \quad\ \ \ldots \\ \quad\ \ pushHeap\ i\ n\ x_n\ [\overline{(\rho_l, m_l)}^k]\ \eta\ ] \end{array}$$

$$pushVarAlt\ x\ [\overline{(\rho_l, m_l)}^k]\ \eta = \begin{cases} ([\,], (\rho_k, m_k)) & \text{si } \exists l.(x \in dom\ \rho_l) \\ ([PUSHNODE\ (\eta\ x)], (\rho_k \cup [m_k + 1/x], m_k + 1)) & \text{en otro caso} \end{cases}$$

$$\begin{array}{l} buildEnvAlt\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, (\rho_1, m_1)) \\ \quad \text{donde} \\ \qquad is = is_n +\!\!+ \ldots +\!\!+ is_1 \\ \qquad (is_i, (\rho_i, m_i)) = pushVarAlt\ x_i\ [(\rho_{i+1}, m_{i+1}),\ \overline{(\rho_l, m_l)}^{k-1}]\ \eta, \quad i \in \{1 \ldots n\} \\ \qquad (\rho_{n+1}, m_{n+1}) = (\rho_k, m_k) \end{array}$$

6.8.3 Traducción

En resumen, la traducción global que se realiza es la siguiente:

$$\begin{array}{l} trdExpr\ (x\ \overline{x_i}^n)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is \\ \quad \text{donde} \\ \qquad is \qquad = \ buildEnv\ [x, \overline{x_i}^n]\ [\overline{(\rho_l, m_l)}^k]\ \eta\ +\!\!+ \\ \qquad\qquad\qquad\ \ [SLIDE\ (n+1)\ m_k, \\ \qquad\qquad\qquad\quad ENTER] \end{array}$$

$trdExpr\ (\mathbf{case}\ e\ \mathbf{of}\ alts\ |^{\overline{x_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = is$

donde

$$
\begin{array}{lcl}
p & = & trdAlts\ alts\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \{\} \\
(is', (\rho'_k, m'_k)) & = & buildEnvAlt\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta \\
is & = & is'\ +\!\!+ \qquad\qquad \text{-- entorno volátil.} \\
 & & [PUSHCODE\ p]\ +\!\!+ \\
 & & trdExpr\ e\ [(\{\ \}, 0), (\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta
\end{array}
$$

$trdExpr\ (\mathbf{letrec}\ \overline{x_i = lf_i\ |^{t_i}}^n\ \mathbf{in}\ e)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is$

donde

$$
\begin{array}{lcl}
is & = & [ALLOC\ tm_n, \ldots, ALLOC\ tm_1]\ +\!\!+ \\
 & & is_1\ +\!\!+ \ldots +\!\!+\ is_n\ +\!\!+ \\
 & & trdExpr\ e\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta \\
(\rho'_k, m'_k) & = & (\rho_k \cup [\overline{(m_k + n + 1 - i)/x_i}^n], m_k + n) \\
(is_i, tm_i) & = & trdLig\ (i-1)\ (lf_i\ |^{t_i})\ [(\rho'_k, m'_k), \overline{(\rho_l, m_l)}^{k-1}]\ \eta, \qquad i \in \{1 \ldots n\}
\end{array}
$$

$trdAlts\ (\overline{alt_i}^n)\ [\overline{(\rho_l, m_l)}^k]\ \eta = p \qquad \&\ cs \leftarrow cs \cup [p \mapsto \overline{is_i}^n]$

donde

$$
\begin{array}{lcl}
is_i & = & trdAlt\ alt_i\ [\overline{(\rho_l, m_l)}^k]\ \eta, \qquad i \in \{1 \ldots n\}
\end{array}
$$

$trdAlt\ (C\ \overline{x_i}^n \to e)\ [\overline{(\rho_l, m_l)}^k]\ \eta = is$

donde

$$
\begin{array}{lcl}
is & = & tradExpr\ e\ [\overline{(\rho_l, m_l)}^k]\ \eta' \\
\eta' & = & [\overline{i/x_i}^n]
\end{array}
$$

$trdLig\ j\ (C\ \overline{x_i}^n\ |^t)\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, n) \qquad \&\ cs[p_{C_k^n}] = ([RETURNCON\ C_k^n], n)$

donde

$$
\begin{array}{lcl}
is & = & buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta\ +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_{C_k^n}]
\end{array}
$$

$trdLig\ j\ (\lambda \overline{x_i}^l.e\ |^{\overline{y_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', n)\ \ \&\ cs \leftarrow cs \cup [p_e \mapsto ([ARGCHECK\ l]\ +\!\!+\ is, n)]$

donde

$$
\begin{array}{lcl}
is' & = & buildHeapEnv\ j\ \overline{y_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta\ +\!\!+ \\
 & & [COPYCODEHEAP\ j\ p_e] \\
\eta' & = & [\overline{i/y_i}^n] \\
\rho' & = & [\overline{l + 1 - i/x_i}^l] \\
is & = & trdExpr\ e\ [(\rho', l)]\ \eta'
\end{array}
$$

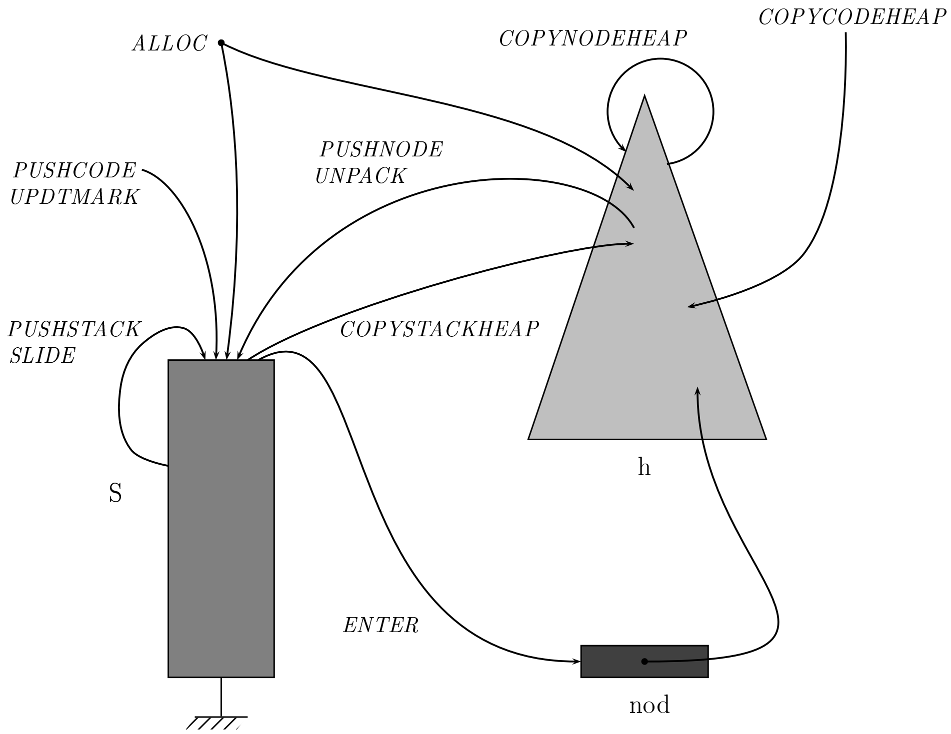

Figura 6.4: Comportamiento de las instrucciones.

$$trdLig\ j\ (x\ \overline{x_i}^n\ |^t)\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is, n+1)$$
$$\&\ \ cs[p_{ap^{(n+1)}}] = ([UPDTMARK, UNPACK\ (n+1), ENTER], m)$$

donde

$$is = buildHeapEnv\ j\ [x, \overline{x_i}^n]\ [\overline{(\rho_l, m_l)}^k]\ \eta\ +\!\!+\ [COPYCODEHEAP\ j\ p_{ap^{(n+1)}}]$$

$$trdLig\ j\ (e\ |^{\overline{x_i}^n})\ [\overline{(\rho_l, m_l)}^k]\ \eta = (is', max\ 1\ n) \quad \& \quad cs \leftarrow cs \cup [p_e \mapsto ([UPDTMARK]\ +\!\!+\ is, n)]$$

donde

$$is' = buildHeapEnv\ j\ \overline{x_i}^n\ [\overline{(\rho_l, m_l)}^k]\ \eta\ +\!\!+\ [COPYCODEHEAP\ j\ p_e]$$
$$\eta' = [\overline{i/x_i}^n]$$
$$is = trdExpr\ e\ [(\{\ \}, 0)]\ \eta'$$

6.8.4 Comportamiento de las instrucciones

En la Figura 6.4 se observa el comportamiento de las instrucciones de forma gráfica para una mejor comprensión. Las instrucciones *RETURNCON*, y *ARGCHECK* no aparecen en esta figura ya que no modifican los estados de la pila, ni del montón, ni de *nod*.

En la tabla que viene a continuación se muestra la semántica operacional de las instrucciones.

	Instrucciones	Pila	Nodo	UpdtPtr	Montón	Código	
	$PUSHSTACK\ i : is$	S	nod	$(?,?)$	h	cs	
$\Longrightarrow$	is	$S!i : S$	nod	$(?,?)$	h	cs	
	$PUSHCODE\ p : is$	S	nod	$(?,?)$	h	cs	
$\Longrightarrow$	is	$p : S$	nod	$(?,?)$	h	cs	
	$ALLOC\ m : is$	S	nod	$(?,?)$	h	cs	(1)
$\Longrightarrow$	is	$a_m : S$	nod	$(?,?)$	h'	cs	
	$COPYSTACKHEAP\ i\ j\ k : is$	S	nod	$(?,?)$	$h[S!i \mapsto (p, a_1 \ldots a_{j-1}, ?, a_{j+1} \ldots a_l)]$	cs	
$\Longrightarrow$	is	S	nod	$(?,?)$	$h[S!i \mapsto (p, a_1 \ldots a_{j-1}, S!k, a_{j+1} \ldots a_l)]$	cs	
	$COPYCODEHEAP\ i\ p : is$	S	nod	$(?,?)$	$h[S!i \mapsto (?, ws)]$	cs	
$\Longrightarrow$	is	S	nod	$(?,?)$	$h[S!i \mapsto (p, ws)]$	cs	
	$PUSHNODE\ i : is$	S	nod	$(?,?)$	$h[nod \mapsto (p, ws)]$	cs	
$\Longrightarrow$	is	$ws!i : S$	nod	$(?,?)$	h	cs	
	$UNPACK\ m : is$	S	nod	$(?,?)$	$h[nod \mapsto (p, \overline{a_i}^m)]$	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	$(?,?)$	h	cs	
	$COPYNODEHEAP\ i\ j\ k : is$	S	nod	$(?,?)$	$h[nod \mapsto (p, ws),\ S!i \mapsto (p, a_1 \ldots a_{j-1}, ?, a_{j+1} \ldots a_l)]$	cs	
$\Longrightarrow$	is	S	nod	$(?,?)$	$h[S!i \mapsto (p, a_1 \ldots a_{j-1}, ws!k, a_{j+1} \ldots a_l)]$	cs	
	$SLIDE\ m\ n : is$	$\overline{a_i}^m : \overline{b_j}^n : S$	nod	$(?,?)$	h	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	$(?,?)$	h	cs	
	$[ENTER]$	$a : S$	nod	$(?,?)$	$h[a \mapsto (p, ws)]$	$cs[p \mapsto (is, l)]$	
$\Longrightarrow$	is	S	a	$(?,?)$	h	cs	
	$UPDTMARK : is$	S	nod	$(?,?)$	$h[nod \mapsto (p, ws)]$	$cs[p \mapsto (is, m)]$	
$\Longrightarrow$	is	$\#nod : S$	nod	$(?,?)$	$h[nod \mapsto (p_{bh^m}, ws)]$	cs	
	$[RETURNCON\ C_k^m]$	$p : S$	nod	$(?,?)$	h	$cs[p \mapsto \overline{is_i}^n]$	
$\Longrightarrow$	is_k	S	nod	$(?,?)$	h	cs	
	$[RETURNCON\ C_k^m]$	$\#a : S$	nod	$(?,?)$	$h[a \mapsto (p_{bh^n}, ws)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(2)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	S	nod	$(?,?)$	$h[a \mapsto (p_{ind}, nod)]$	cs	
	$[RETURNCON\ C_k^m]$	$\#a : S$	nod	$(?,?)$	$h[a \mapsto (p_{bh^n}, ws),\ nod \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	$cs[p_{bh^n} \mapsto ([\,], n)]$	(3)
$\Longrightarrow$	$[RETURNCON\ C_k^m]$	S	nod	$(?,?)$	$h[a \mapsto (p_{C_k^m}, \overline{a_i}^m)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^m : S$	nod	$(?,?)$	h	cs	
$\Longrightarrow$	is	$\overline{a_i}^m : S$	nod	$(\bot,?)$	h	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	$(\bot,?)$	$h[a \mapsto\ (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(4)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b,n)	$h'[a \mapsto\ (p_{ind}, b),\ b \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	(b,n)	$h[a \mapsto\ (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(5)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b,n)	$h[a \mapsto (p_{ind}, b)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	(b',n')	$h[a \mapsto\ (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(6)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(b,n)	$h[a \mapsto\ (p_{ind}, b),\ b \mapsto (p_{pap^{n+1}}, nod : \overline{a_i}^n)]$	cs	
	$ARGCHECK\ m : is$	$\overline{a_i}^n : \#a : S$	nod	$(?,?)$	$h[a \mapsto\ (p_{bh^l}, ws)]$	$cs[p_{bh^l} \mapsto ([\,], l)]$	(7)
$\Longrightarrow$	$ARGCHECK\ m : is$	$\overline{a_i}^n : S$	nod	(a,n)	$h[a \mapsto (p_{pap^{n+1}},\ nod : \overline{a_i}^n)]$	cs	

(1) donde b es una nueva dirección en el montón donde entra una clausura con m variables libres y h' es el montón resultante después de la reserva.
(2) $n < m$
(3) $m \leq n$
(4) donde b es una nueva dirección en el montón donde entra una clausura con m variables libres h' es el montón resultante después de la reserva, $n < m$, y $l < n$.
(5) $m < n$ y $l < n$
(6) $n < m$ y $n' < n$ y $l < n$
(7) $n < m$ y $n \leq l$

Como se comentó en la Sección 6.3.5 la instrucción $UNPACK\ m$ se puede ver como una macro de compilación ya que es equivalente a $PUSHNODE\ m \ldots PUSHNODE\ 1$. Según las posibilidades de optimización de una u otra, se mantendrá o no dicha instrucción.

6.8.5 Código prefabricado

El código prefabricado necesario es el siguiente:

$$
\begin{array}{lcl}
cs[p_{bh^m}] & = & ([\,], m) \\
cs[p_{ind}] & = & ([\mathit{PUSHNODE}\ \mathit{1},\ \mathit{ENTER}], 1) \\
cs[p_{ap^m}] & = & ([\mathit{UPDTMARK}, \mathit{UNPACK}\ m,\ \mathit{ENTER}], m) \\
cs[p_{pap^m}] & = & ([\mathit{UNPACK}\ m,\ \mathit{ENTER}], m) \\
cs[p_{C_k^m}] & = & ([\mathit{RETURNCON}\ C_k^m], m)
\end{array}
$$

6.8.6 Comparación con el código de *GHC*

El código generado aún tiene algunas diferencias con el que genera *GHC*. Las principales son las siguientes:

- La máquina *STG* sólo tiene punteros en la pila. Por tanto no tiene tres tipos distintos de objetos en ella. Esto lo logra ocultando las marcas de actualización como punteros al código encargado de realizar la actualización.
- Cada vez que se borra de la pila no se reorganiza está, sino que cada puntero lleva un bit indicando si está siendo utilizado o es basura. Así que el comportamiento de la instrucción *SLIDE* no es el que aquí se presenta, sino que esta instrucción lo único que hace es colocar los bits para que indiquen que esa parte es basura.
- Para acceder al código de una clausura es necesario una doble indirección, y no como aparece aquí una indirección simple, ya que el puntero de la clausura apunta a una tabla de información generada en tiempo de compilación de la clausura, en la que su primer elemento es un puntero al código.
- Las clausuras llevan de forma indirecta la información de tamaño y otras funciones necesarias para la recolección de basura. Esta información está accesible a través del puntero de la clausura y se encuentra en la tabla de información.
- Las marcas de actualización están organizadas en la pila, pues cada una apunta a la anterior. Con ello, partiendo desde la más actual podemos recorrerlas y eliminar las que hayan pasado a estar muertas.
- La máquina *STG* no tiene *agujeros negros*. Estos desaparecen en un refinamiento. Se sabe cuándo hemos entrado en una clausura bajo evaluación porque tenemos las marcas de actualización organizadas.
- Existen dos formas de realizar una actualización: a través de un vector de actualizaciones o directamente.

Para más detalles ver [Pey92].

A pesar de estas diferencias, nuestro enfoque es equivalente y, por lo tanto, no nos aproximamos más y nos damos por satisfechos.

Capítulo 7

Conclusiones y trabajo futuro

El objetivo principal del presente trabajo de doctorado era dar una demostración formal y fácil de comprender de que la máquina *STG* era correcta y completa con respecto a la semántica denotacional y, por tanto, las reducciones que realiza el actual compilador *GHC* son correctas y completas con respecto a dicha semántica.

Estos objetivos se han logrado partiendo de la semántica de Launchbury el cual demostró que era correcta y completa con respecto a la semántica denotacional. Hemos proseguido con las modificaciones que Sestoft realizó de dicha semántica para acercarla un poco más a la ejecución real de la máquina. Sobre la semántica de Sestoft hemos apreciado una errata que se ha corregido en la Sección 3.1.2. Esta revisión exhaustiva de la semántica de Sestoft nos ha llevado a definir una semántica con montones minimales (véase Figura 3.2) en la que expresamos de forma sencilla la recolección de las ligaduras que han dejado de estar vivas en un momento del cómputo. En la semántica se pueden apreciar los momentos del cómputo en los que una ligadura puede dejar de ser útil. Es la primera ocasión en la que se refleja en la semántica que el conjunto de las marcas de actualización pendientes puede ser podado, y esto nos lleva a los mismos resultados que una semántica en la que no tengamos esta optimización. Normalmente estas ideas aparecían en la etapa de generación de código como optimizaciones en un apartado recóndito. La máquina *STG* realiza esta optimización y, por tanto, hemos creído conveniente mostrarla directamente sobre la semántica. El lector interesado en la recolección de basura en los lenguajes funcionales puede encontrar esta semántica interesante para sus propósitos. Otro resultado es que las pilas de las máquinas de Sestoft surgen como una trasformación natural de los conjuntos A, B y C necesitados en esta semántica.

Posteriormente nos hemos centrado en las máquinas de Sestoft. Sestoft demostró que éstas eran correctas y completas con respecto a su semántica. Sobre estas máquinas hemos realizado varias demostraciones formales de equivalencia, y en el camino hemos demostrado ciertas propiedades interesantes de los entornos (véase el Capítulo 4). Entre ellas el paso de una expresión con nombres de variables a otra en la que las variables se representan por los índices de De Bruijn. Normalmente esta traducción se realiza al generar código y, por tanto, el mecanismo queda oculto.

Posteriormente hemos tomado una de las máquinas de Sestoft, la *Mark-2*, y la hemos comparado con la máquina *STG*. En esta comparación hemos descubierto que la máquina *STG* no es correcta con respecto a la semántica denotacional. Si partimos de una expresión mal tipada los cálculos que realiza pueden llevar a reducir una expresión a una forma normal aún cuando la semántica no sea capaz de reducir nada. Por tanto, los cálculos sólo son equivalentes a los que realiza la semántica si partimos de una expresión cerrada bien tipada, propiedad que no se exige en ninguna de las máquina de Sestoft. Hemos definido una nueva máquina que hemos llamado *STG-1S* cuya principal diferencia con la máquina *STG* es que en lugar de tener tres pilas tiene una solo, que es una mezcla de las tres pilas de la máquina *STG* pero conservando el orden en que se han producido los eventos. Ese es el motivo por el cual la máquina *STG* no era correcta, ya que pierde la disposición temporal de la pila de argumentos con respecto a la pila de continuaciones. No obstante, la máquina *STG* es correcta y completa si partimos de una expresión bien tipada. Hemos demostrado formalmente que la máquina *STG-1S* es equivalente a la máquina *Mark-2* para un subconjunto del lenguaje *STGL*. Este lenguaje es igual de expresivo que el lambda cálculo de Launchbury o Sestoft pero requiere otro proceso de normalización.

Tras esta demostración, hemos bajado un poco más en el nivel de abstracción, y nos hemos centrado en la etapa de generación de código. En el Capítulo 6 hemos formalizado, por primera vez, una generación de código muy próxima a la que realiza el compilador *GHC*. Las diferencias de *GHC* con nuestra generación de código se refieren a pequeñas optimizaciones y han sido enumeradas en la Sección 6.8.6. Hasta ahora los únicos trabajos que hacían referencia a la generación de código son [PL92, PHH$^+$93, PMR99] o directamente el código del compilador. En ninguno de ellos aparece una formalización de la generación de código que realiza *GHC*. Nuestro trabajo pretende una formalización clara, concisa y comprensible. Para ello la hemos realizado en sucesivas etapas, partiendo de una generación de código ingenua y algo alejada de la que realiza *GHC*, y posteriormente refinándola en sucesivas etapas explicando los motivos de dichos refinamientos, y viendo en cada uno de ellos las ventajas e inconvenientes que presentan, hasta acercarnos bastante al código generado por *GHC*.

Los principales resultados de los capítulos 3, 4, y 5 serán presentados y publicados en [EP01a, EP01b].

Se han quedado fuera del estudio los valores desencapsulados, operaciones básicas sobre valores desencapsulados, expresiones **case** sobre estos valores y otros detalles de menor importancia que se pueden simular en el lenguaje que manejamos, como son: las expresiones **let** *no recursivas* y la alternativa por defecto en la expresión **case**. No obstante, para los valores desencapsulados y las operaciones sobres ellos sería necesario incluirlos desde el principio. Habría que comenzar con la semántica demostrando su corrección con respecto a la semántica denotacional, después tendríamos que añadirlos a cada una de las máquinas de Sestoft y por último sería necesario demostrar la equivalencia de éstas con respecto al comportamiento de la máquina *STG* con enteros desencapsulados.

En un futuro próximo nos proponemos abordar la incorporación de los enteros desencapsulados a todas las etapas, seguidas del resto de expresiones. Un punto de partida es [PL91] donde se estudia su semántica denotacional.

Otro aspecto a estudiar es la definición de recolectores de basura incrementales que no produzcan interrupciones prolongadas en la ejecución del programa. La semántica propuesta en la Sección 3.2 puede ser el punto de arranque. Un trabajo reciente en esta linea es [CFM$^+$00] donde se realiza la recolección de basura entrelazada con la ejecución del programa.

Bibliografía

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit Substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[Aug84] Lennart Augustsson. A Compiler for Lazy ML. In *In Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, volume 274 of *LNCS*, pages 218–227. Springer-Verlag, 1984.

[BFKT00] Clem Baker-Finch, David J. King, and Phil Trinder. An Operational Semantics for Parallel Lazy Evaluation. International Conference on Functional Programming, ICFP'00, 2000.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Series in Computer Science. Prentice Hall, 2ª edition, 1998.

[BKL+98] Silvia Breitinger, Ulrike Klusik, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. DREAM: the Distributed Eden Abstract Machine. In *Implementation of Functional Languages, IFL'97. St. Andrews, Scotland*, volume 1467 of *LNCS*, pages 250–269. Springer-Verlag, 1998.

[BP99] Richard Bird and Ross Paterson. De Bruijn Notation as a Nested Datatype. *Journal of Functional Programming, JFP'99*, 9(1):77–91, 1999.

[Bru72] Nicolette De Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation. *Indag. Math.*, 34(5):381–392, 1972.

[Bru78] Nicolette De Bruijn. Lambda Calculus Notation with Namefree Formulas Involving Symbols that Represent Reference Transforming Mapping. *Indag. Math.*, 40:348–356, 1978.

[CFM+00] Andy M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton Jones, and Lyndon While. Non-stop Haskell. In *International Conference on Functional Programming, ICFP'00*, pages 257–267, September 2000.

[Cha95] Manuel M.T. Chakravarty. Integrating Multithreading into the Spineless Tagless G-machine. In *Glasgow Workshop on Functional Programming*, Electronic Workshops in Computing, Ullapool, Scotland, July 1995. Springer-Verlag.

[Cur91] Pierre-Louis Curien. An Abstract Framework for Environment Machines. *Theoretical Computer Science, TCS'91*, 82(2):389–402, 31 May 1991.

[EP01a] Alberto de la Encina and Ricardo Peña. A Proof of Correctness for the STG Machine. In *Implementation of Functional Lenguages, IFL'01*, Stockholm, Sweden, September 2001.

[EP01b] Alberto de la Encina and Ricardo Peña. A Proof of Correctness for the STG Machine. *Primeras Jornadas sobre Programación y Lenguajes, PROLE'2001, Almagro, España*, Noviembre 2001.

[FW87] Jon Fairbairn and Stuart C. Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *Proc. of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.

[GS01] Jörgen Gustavsson and Josef Svenningsson. A Usage Analysis with Bounded Usage Polymorphism and Subtyping. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, volume 2011 of *LNCS*, pages 140–157. Springer-Verlag, 2001.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Edinburgh University, 1986.

[Hor96] Matthias Horn. A Different Integration of Multithreading into the STGM. In W. Kluge, editor, *Implementation of Functional Languages*, 1996.

[Hug90] John Hughes. Why functional programming matters. In *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[Joh84] Thomas Johnsson. Efficient Compilation of Lazy Evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, June 1984.

[Lau93] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. Conference on Principles of Programming Languages, POPL'93*. ACM, 1993.

[Mou98] Jon Mountjoy. The Spineless Tagless G-machine, Naturally. In *Third International Conference on Functional Programming, ICFP'98, Baltimore, USA*. ACM Press, 1998.

[NW98] Gopalan Nadathur and Debra Sue Wilson. A Notation for Lambda Terms: A Generalization of Environments. *Theoretical Computer Science*, 198(1–2):49–98, 30 May 1998. Fundamental Study.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pey92] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine, Version 2.5. *Journal of Functional Programming, JFP'92*, 2(2):127–202, April 1992.

[Pey96] Simon L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *In European Symposium on Programming, ESOP'96, LNCS 1058*, 1996.

[PH99] Simon L. Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98.* URL http://www.haskell.org, February 1999.

[PHH+93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will D. Partain, and Philip L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conference on Functional Programming Languages and Computer Architecture FPCA'91*, volume 523 of *LNCS*, pages 636–666. Springer-Verlag, September 1991.

[PL92] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages, A Tutorial.* Prentice-Hall, 1992.

[PMR99] Simon L. Peyton Jones, Simon Marlow, and Alastair Reid. The STG Runtime System (revised). http://www.haskell.org/ghc/docs, 1999.

[PMST01] Simon L. Peyton Jones, Simon Marlow, Julian Seward, and Reuben Thomas. Glasgow Haskell Compiler Sources (GHC5.0). http://www.haskell.org/ghc/, 2001.

[PPS96] Simon L. Peyton Jones, Will Partain, and Andrew Santos. Let-floating: moving bindings to give faster programs. *International Conference on Functional Programming, ICFP'96*, May 1996.

[PS89] Simon L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. *Journal of Functional Programming Languages and Computer Arquitecture*, September 1989.

[PS98] Simon L. Peyton Jones and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47*, September 1998.

[San95] André Santos. Compilation by Transformation in Non-Strict Functional Languages. Ph.D. thesis. Department of Computing Science. University of Glasgow, 1995.

[Ses97] Peter Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison-Wesley, 1995.

[WP99] Keith Wansbrough and Simon L. Peyton Jones. Once Upon a Polymorphic Type. In *The Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, Jan. 1999.