Estrategias de optimización en compiladores de lenguajes funcionales modernos

Santiago Romero
Estudiante de Ciencias de la Computación
Facultad de Ciencias UNAM, Ciudad de México
santiago.rp@ciencias.unam.mx

Arlet Pinacho
Estudiante de Ciencias de la Computación
Facultad de Ciencias UNAM, Ciudad de México
arlet.pinacho@ciencias.unam.mx

Juvenal Guzmán Condado
Estudiante de Ciencias de la Computación
Facultad de Ciencias UNAM, Ciudad de México
juvenal.guzman@ciencias.unam.mx

Ricardo Iván Martínez Cano
Estudiante de Ciencias de la Computación
Facultad de Ciencias UNAM, Ciudad de México
ricardoivan@ciencias.unam.mx

***Resumen*—Los compiladores de lenguajes funcionales modernos enfrentan el doble desafío de preservar las abstracciones de alto nivel mientras ofrecen una ejecución eficiente en arquitecturas de bajo nivel. En este reporte investigamos técnicas de optimización para compiladores funcionales mediante el diseño e implementación de *subs*, un lenguaje funcional que demuestra una compilación en múltiples etapas. Subs se traduce al cálculo lambda, el cual posteriormente se convierte en un *cálculo de combinadores SKI extendido* enriquecido con combinadores adicionales que permiten una optimización y ejecución práctica y eficiente. A partir de esta representación intermedia, los programas se compilan hacia dos entornos distintos en C++: uno *estricto* y uno *perezoso*. El runtime perezoso aprovecha la técnica de *graph reduction* para modelar la evaluación no estricta, mientras que el runtime estricto proporciona características de rendimiento predecibles. Nuestro trabajo ilustra cómo los supercombinadores pueden servir como una representación intermedia práctica y optimizable, conectando los fundamentos teóricos y prácticos del campo de los compiladores de lenguajes funcionales.**

***Palabras clave*—SKI, Combinadores, Supercombinadores, Cálculo lambda, Optimización de código, Lenguajes funcionales, Graph reduction, Lambda lifting, Generación de código, Compiladores, Lenguajes de programación**

I. Preliminares

En esta sección se presentan las definiciones y convenciones que se emplearán a lo largo del trabajo. El objetivo es establecer un marco común que permita la formulación rigurosa de los resultados posteriores.

- **Términos Lambda**: Decimos que los Términos Lambda son palabras que se definen en el siguiente alfabeto[1]:
  - Variables: $v_0, v_1, \cdots$
  - Abstractor: Λ
  - Parentesis: $(.)$

  Utilizamos las siguientes reglas para definir al conjunto de λ-Terminos de manera inductiva[1]:
  - $x \in \Lambda$: Una variable x es una cadena que representa a un parámetro, por si misma puede ser un término lambda.
  - $M \in \Lambda \Rightarrow (\lambda x M) \in \Lambda$: Una abstracción lambda del tipo $(\lambda x.M)$ es una función la cuál toma como entrada a la variable x y regresa al cuerpo M como resultado. Decimos que esta definición de función es un término lambda.
  - $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$: Una aplicación (MN) es la aplicación de una función M a un argumento N dónde M y N son términos lamda.
- **Gráfica**: Una gráfica es un conjunto de puntos conocidos como vértices, mientras que las lineas que conectan a estos puntos se conocen como aristas.[2]
  Decimos que las aristas pueden representar relaciones entre los vértices que estas conectan.[2]
- **DAG**: Una gráfica dirigida acíclica (DAG por sus siglas en inglés), es una gráfica en dónde cada una de sus aristas cuenta con una dirección en la que pueden ser recorridas.[3] Sumado a esto decimos que esta gráfica es acíclica debido a que no cuenta con ciclos dirigidos.[3]
- **Índices De Brujin**: Son una técnica que nos permite escribir expresiones del cálculo lambda en dónde a cada variable se le reemplaza por un número que representa la cantidad de niveles en la expresión que debemos de tomar "saltar" para encontrar a la variable con la que se encuentra ligada.[1]
- **Efectos secundarios**: Un lenguaje de programación permite la presencia de efectos secundarios si una o más variables cambian su valor durante la ejecución de una función dentro de la cuál no fueron definidas, es decir, no son variables locales [4].

II. Introducción

Existen distintos criterios de clasificación para los lenguajes de programación, uno de los más comunes es la clasificación de acuerdo al paradigma al que pertenecen. Esta permite agrupar los lenguajes en dos grandes categorías: los lenguajes de programación imperativos y los lenguajes de programación declarativos [5]. Si bien, ambas tienen sus propias subcategorías, para el desarrollo de este trabajo nos enfocamos en la subcategoría *funcional* del paradigma declarativo.

Es común mencionar que los lenguajes del paradigma *declarativo* requieren que los programadores en lugar de especificar los pasos a seguir para obtener cierto resultado esperado (el 'cómo' característico del enfoque imperativo), declaren expresiones que reflejen el resultado que se busca obtener (el *qué*) [6]. Sin embargo, otras características que los diferencian son: la recursión como principal estructura de control y generalmente la ausencia de operaciones como la asignación, las cuales pueden dar lugar a *efectos secundarios* [5][7, p. 7].

En particular en el enfoque *funcional* las expresiones declaradas constituyen *funciones* cuyo resultado, al ser aplicadas, únicamente depende del valor de sus argumentos, permitiendo que las mismas expresiones puedan ser utilizadas bajo distintos contextos o *estados*. De manera que se asegura la consistencia del resultado respecto a los argumentos [7, p. 7] [6].

Un punto que resulta de gran interés en este paradigma de los lenguajes de programación es su sólida fundamentación en modelos matemáticos, entre los cuales el *cálculo lambda* constituye el principal [7, p. 9], junto con otros marcos teóricos como la *teoría de categorías* [5]. Gracias a estas bases es posible trasladar a los programas propiedades como la *transparencia referencial*, que a su vez permiten emplear técnicas como el *razonamiento ecuacional*, herramientas resultan fundamentales para facilitar el análisis y la comprensión formal de los programas [8, p. 360].

Ya mencionamos que, en los lenguajes de programación funcional, las funciones pueden aplicarse directamente a sus argumentos. Esto también refleja claramente el fundamento matemático que rige a este paradigma, pues describe la forma en que se deben evaluar las expresiones de un programa. Dicho proceso consiste en realizar una serie de reducciones hacia representaciones equivalentes, hasta obtener la más simple [7, p. 4]. Este esquema se deriva directamente de la β-reducción y de la noción de *forma normal* en las expresiones lambda.

Algunas de las características más destacables que incorporan los lenguajes funcionales modernos son:

- ***Funciones de orden superior***

  Las funciones son tratadas como *ciudadanos de primer nivel*, esto quiere decir que pueden ser almacenadas en estructuras de datos, pasadas como argumentos de otras funciones y regresadas como resultado de alguna aplicación, como cualquier otro valor común (enteros, cadenas y otros) [8, p. 382]. Esta propiedad da la posibilidad de crear nuevas funciones como una combinación de otras ya definidas [9, p. 8].

  También, muchos lenguajes modernos además de permitir definir funciones como *ecuaciones*, permiten su creación como *abstracciones lambda*, eliminando la necesidad de un nombre [8, p. 382].

- ***Evaluación perezosa***

  Debido al esquema de evaluación que sigue la aplicación de funciones, es posible que el orden en el que se realizan las sustituciones genere cómputo repetido, afectando así el tiempo empleado en la ejecución de un programa. Por ello, los lenguajes modernos incorporan la evaluación perezosa, como una forma de posponer la reducción de las expresiones hasta que sea estrictamente necesario [10, p. 142]. Más adelante será necesario notar que este tipo de evaluación, además requiere que su implementación permita asegurar que las expresiones sean evaluadas un sola vez y en caso de volver a aparecer permitir la reutilización del valor ya calculado [11, p. 194] para contribuir en la mejora del tiempo de ejecución.

  Es importante destacar que esta propiedad no puede coexistir tan naturalmente con los *efectos secundarios* presentes en otros paradigmas, debido a la relevancia que el contexto juega en la evaluación [9, p. 9], motivo que si bien no impide su incorporación en otros enfoques, sí la vuelve poco viable.

  Un uso interesante de esta propiedad es detallado en [9, p. 9], pues es posible aprovecharla para coordinar la ejecución de dos programas en el que el primero consume resultados del segundo, sin la necesidad de que este segundo sea un programa que siempre termine. Esto porque al no tener que reducir cada cálculo al instante (*evaluación estricta o ansiosa*), se da lugar al uso de estructuras de datos infinitas [8, p. 385].

Si bien este tipo de lenguajes proveen ventajas, tales como: disminución de fuentes de *bugs*, independencia al contexto, amplia expresividad que deriva en código más breve y facilidad de modularidad. Existen dos grandes desventajas por las que no son el tipo de lenguaje más utilizado: la disminución de la eficiencia y la idoneidad para aplicaciones de naturaleza imperativa [7, p. 10].

Es esta primera desventaja la principal motivación de esta investigación, ya que el objetivo es abordar algunas de las técnicas que emplean los compiladores de los *lenguajes de programación funcionales modernos* para transformar los programas con el fin de retener eficiencia.

Según [7, p. 10] la eficicencia de los programas funcionales se ve afectada principalmente porque las arquitecturas no están diseñadas para realizar este tipo de cómputo, ya sea por la falta de actualizaciones destructivas, que si pueden realizarse

mediante asignaciones, o bien por que el costo de llamadas a funciones es muy alto.

III. Metodología

III-A. Cálculo Lambda

El calculo lambda es un sistema formal de la lógica inventado alrededor de 1928 por Alonzo Church. Su objetivo era crear una fundación para la lógica mas natural que la teoría de tipos de Russell o la teoría de conjuntos de Zermelo-Fraenkel. Dicho sistema esta basado en el concepto de las funciones y sus elementos base son la abstracción ($\lambda x.M$), la operación de formar una función a partir de la variable que la define, y la aplicación (F(X)), la operación de aplicar una función a una variable. [12] [13]

El calculo lambda se define de forma inductiva de la siguiente manera: [14]

1. Toda variable x es un λ-término.
2. Si M y N son λ-términos, entonces (MN) es un λ-término.
3. Si M es un λ-término y x es una variable, entonces (λx.M) es un λ-término.

La abstracción lambda es una operación que va a ligar a las variables x en un λ-término si se encuentran dentro del alcance de la abstracción lambda λx, en otro caso, las variables serán libres. [14]

Denotamos al conjunto de variables libres de un λ-término M como FV(M). Sea x una variable y sean M, N λ-términos, entonces podemos definir dicho conjunto de forma inductiva de la siguiente manera: [15]

1. FV(x) = x
2. FV(MN) = FV(M) $\cup$ FV(N)
3. FV($\lambda x.M$) = FV(M) - x

Denotamos al conjunto de las variables ligadas de un λ-término M como BV(M). Sea x una variable y sean M, N λ-términos, entonces podemos definir dicho conjunto de forma inductiva de la siguiente manera: [15]

1. BV(x) = $\emptyset$
2. BV(MN) = BV(M) $\cup$ BV(N)
3. BV($\lambda x.M$) = FV(M) $\cup$ x

Sea x una variable y M, N λ-términos, definimos a la operación de sustituir por N a las variables libres x en M, cuya notación es M[x := N], de la siguiente manera: [16]

1. x[x := N] $\equiv$ N;
2. y[x := N] $\equiv$ y, si x $\not\equiv$ y;
3. (M_1M_2)[x := N] $\equiv$ (M_1[x := N]M_2[x := N]);
4. ($\lambda y.M_1$)[x := N] $\equiv$ λy.(M_1(x := N))

Sean M y N λ-términos, decimos que existe un cambio de variables desde M hasta N si cualquier abstracción λ x.A en M puede ser reemplazada por λ y.A[x := y] obteniendo a N. Decimos que M y N son α-equivalentes si existe una secuencia de cambio de variables que empiece en M y termine en N. [13] [14]

Sean A y B λ-términos, decimos que existe una β-reducción entre A y B de un solo paso (A $\triangleright\_\beta$,1 B) si existe un subtermino C de A, una variable x y λ-terminos M y N tales que C $\equiv$ (λ x.M)N y tales que cuando reemplazamos C en A por el λ-termino α-equivalente M[x := N] obtenemos a B. [13] [14]

Decimos que existe una secuencia de β-reducciones desde el λ-término M hasta el λ-termino N (M $\triangleright$ N) si existe una secuencia finita $s_1, s_2, ..., s_n$ de λ-términos tales que, empezando desde M y terminando en N, existe una β-reducción $s_k \triangleright\_\beta$,1 s_{k+1} para cada $k \in \{1, ..., n\}$. [13] [14]

Se denomina β-redex a un λ-término de M de la forma (λx.P)Q tal que M sea un candidato a una β-reducción. Al hacer dicha β-reducción decimos que se contrae al β-redex. Un λ-termino es una forma normal si no contiene a ningun β-redex. Decimos que N es la forma normal de M si existe una secuencia de β-reducciones desde M hasta N, tales que N sea una forma normal. [13] [14]

III-B. Combinadores SKI

III-B1. ¿Que es SKI?: SKI es un sistema de cálculo que consta de los operadores S, K e I.[17] Decimos que estos 3 combinadores son supercombinadores.[11]
Un supercombinador S es una expresión lambda de la forma[11]:

$$\lambda x_1, \lambda x_2, \ldots, \lambda x_n.E \text{ con una aridad de } n$$

En dónde se cumplen las siguientes condiciones:[11]

- S no cuenta con variables libres
- Cualquier abstracción lambda en E es un supercombinador
- $n \geq 0$, es decir, no necesariamente cuenta con lambdas.

En otras palabras, un supercombinador es una expresión lambda que no cuenta con variables libres, en ninguna parte y que cuenta con por lo menos un argumento.
Algunos ejemplos de esto son:[11]

$$3$$
$$(+2\ 5)$$
$$\lambda x.x$$
$$\lambda x.(+x\ x)$$
$$\lambda f.\ f(\lambda x.(+x\ x))$$

Estas estructuras nos permiten reducir el uso de memoria de manera innecesaria, pues al no contenter variables libres, no es necesario mantener una pila para almacenar estas.[18]
La CFG de este sistema de cálculo se puede expresar de la siguiente manera en la BNF:[19]:

⟨*Expr*⟩ ::= S
| K
| I
| ⟨*Expr*⟩ ⟨*Expr*⟩
| '(' ⟨*Expr*⟩ ')'

Decimos que el combinador I, es el combinador de la identidad, es decir, de manera formal:[17]

$$I\ x \to x$$

Algunos ejemplos de este combinador son:

$$I\ 4 \to 4$$

Notemos, que también es posible construir la siguiente expresión utilizando a otros combinadores, de la siguiente manera:

$$I\ (K\ y\, x) \to K\ y\, x$$

Para el combinador K, también conocido como la función canceladora, tenemos que la regla es la siguiente[18]:

$$K\ c\, x \to c$$

Decimos que esta función se encarga de tomar de 2 argumentos, y solamente regresar el primero.[18]
Un ejemplo de esto es:

$$K\ 4\, 5 \to 4$$

Notemos, que el concepto de ignorar variables nos es muy útil para definir a los tipos booleanos, pues en estos decimos que necesitamos una función que realice la siguiente operación[19]:

$$\text{Verdadero } V\ x\, y \to x$$
$$\text{Falso } F\ x\, y \to y$$

Esto se puede lograr utilizando al combinador K junto con otro combinador que se analiza más adelante, el combinador S:

$$V = K$$
$$F = S\, K$$

Por último, se tiene al combinador S, el cuál se le conoce como el combinador de distribución, este se encarga de distribuir un solo argumento a 2 funciones, su regla es la siguiente:[18]

$$S\ f\, g\, x \to f\, x\, (g\, x)$$

Un ejemplo de esto compuesto con otros combinadores es:[18]

$$S\ K\, x\, y \to (K\, y)\, (x\, y) \to y \quad \cong \quad F\, x\, y$$

Si se supone que se cuenta con una operación $+$ de suma, podemos visualizar un ejemplo de este combinador en una operación aritmética:

$$S\ +\ I\, 7 \to +7\, (I\, 7) \to +\, (7\, 7) \to 14$$

Notemos, que cada término de este sistema no representa una cadena, si no representa un árbol, un ejemplo de esto es:[19]

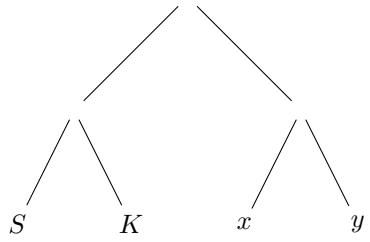


Figura 1. Árbol para la expresión $S\ K\, x\, y$

Decimos que por eso, los paréntesis en este tipo de expresiones nos ayudan a determinar la asociación de unas variables con otras cuando esto es necesario.[19] Decimos que en ausencia de paréntisis, este sistema asocia a la izquierda.[19]
Una característica muy importante del cálculo SKI es que es capaz de representar a todos los programas computables.[18] Es decir, este sistema de cálculo es Turing computable.
Por último, observemos como SKI maneja 3 conceptos fundamentales que nos permiten "programar" en el:[19]

- **Recursión**
  Notemos que la expresión $(S\ I\ I)\ (S\ I\ I)$ es una expresión que se puede reescribir a si misma. Pues:

  $$S\ I\ I\, x \to (I\, x)\, (I\, x) \to x\, (I\, x) \to x\, x$$

  Por lo que se puede definir una llamada a una función recursivamente definiendo a un x como:

  $$x = S(\ K\ f)\ (S(\ I\ I))$$

  De esta manera se puede expandir de la siguiente forma:

  $$S\ I\ I\, x \to xx \to S(\ K\ f)\ (S(\ I\ I))x \to$$
  $$((K\ f)\, x)\ ((S\ I\ I)\, x) \to ((K\ f)\, x)\ (x\, x) \to f\, (x\, x) \to$$
  $$f(f(x, x))$$

- **Condicionales**
  Para lograr tener un comportamiento de ramificación, es necesario contar con booleanos. Es por eso que podemos tomar la definición anterior de booleanos, es decir:

  $$V = K$$
  $$F = S\, K$$

  Para definir las operaciones sobre booleanos tenemos que:

  - $not\, B = B\, F\, V$ en dónde B es un booleano
  - $B_1\, or\, B_2 = B_1\, V\, B_2$ en dónde B es un booleano
  - $B_1\, and\, B_2 = B_1\, B_2\, F = B_1\, B_2\, (S\, K)$ en dónde B es un booleano

  Ahora veamos que se puede definir a un condicional de la siguiente manera:

  $$if\, B\, then\, X\, else\, Y = B\, X\, Y$$

- **Estructuras de datos**
  Observemos que como se realizó con las definiciones

de los booleanos y las estructuras de control, podemos escribir combinadores para representar estructuras de datos, funciones, etc.
Por ejemplo, para definir a la función $swap\,(x\,y) = y\,x$, tenemos que podemos usar al siguiente combinador $swap = S\,(K\,(S\,I))(S\,(K\,K)\,I)$

III-B2. SKI como lenguaje de compilación intermedio: Como se pudo observar en la sección anterior, SKI esa un sistema de cálculo simple, pues en su base solo cuenta con 3 reglas, poderoso, ya que es Turing completo y eficiente, pues debido a que no cuenta con variables no ligadas, no es necesario tener un ambiente en dónde se deban de guardar.
Gracias a su simplicidad, tenemos que el comportamiento de estos combinadores, se puede escribir en un lenguaje de programación que nos permita traducir código SKI a un lenguaje como C++. De tal manera que podemos ejecutar código escrito en C++, concepto explorado más adelante en este trabajo.
Un problema con el que se tiene al intentar escribir programas más complejos en SKI, es que su sintaxis se vuelve muy larga y difícil de escribir, en particular, al intentar introducir múltiples argumentos en combinador.
Tomando en cuenta sus desventajas y ventajas, podemos observar que SKI funciona como un buen candidato para un lenguaje intermedio, pues una computadora no necesita preocuparse por la facilidad de lectura de un programa y tiene mucho que ganar de un sistema de cálculo eficiente.
Entonces es necesario pensar en otra forma de expresar programas que sea de lectura más fácil para humanos y que tenga una traducción no tan difícil a SKI.
Resulta que tal sistema existe y es el cálculo lambda. Esto debido a que existe un algoritmo de transformación directo de cálculo lambda a SKI.[11] Esta transformación, manteniendo el espíritu de SKI, cuenta con 3 simples reglas para cada uno de sus combinadores, las cuáles son:[11]

- Para el combinador S
  Supongamos una abstracción lambda que tiene la siguiente forma:
  $$(\lambda x.e_1\,e_2)$$
  en dónde e_1 y e_2 son expresiones arbitrarias.
  Notemos que el siguiente combinador SKI es equivalente:
  $$S\;(\lambda x.e_1)\,(\lambda x.e_2)$$
  Esto lo podemos apreciar al aplicar ambas funciones a un argumento z.
  Para la abstracción lambda:
  $$(\lambda x.e_1\,e_2)\,z \to (e_1[z/x])\,(e_2[z/x])$$
  Por otro lado, para el combinador SKI:
  $$S\;(\lambda x.e_1)\,(\lambda x.e_2)\,z \to ((\lambda x.e_1)\,z)\,((\lambda x.e_2)\,z) \to (e_1[z/x])\,(e_2[z/x])$$
  Es decir, obtenemos el mismo resultado con ambos.
  A esta transformación se le conoce como la transformación S, y formalmente se define de la siguiente manera:
  $$\lambda x.e_1\,e_2 \Rightarrow S\;(\lambda x.e_1)\,(\lambda x.e_2)$$
- Para el combinador K
  Veamos que en una abstracción lambda $(\lambda x.c)$ en dónde c es una constante o variable distinta de x. Se toma al argumento x y se deshecha, regresando solamente al argumento c. Es decir, el mismo comportamiento que el combinador K.
  A esta transformación se le conoce como la transformación K, la cuál formalmente se define de la siguiente manera:
  $$\lambda x.c \Rightarrow K\;c$$
- Para el combinador I
  En el cáculo lambda, tenemos que la función identidad está definida de la siguiente manera:
  $$(\lambda x.x) \to x$$
  Es decir, el mismo comportamiento que el combinador I.
  A esta transformación se le conoce como la transformación I, y se define formalmente de la siguiente manera:
  $$(\lambda x.x) \Rightarrow I$$

Utilizando estas transformaciones, es posible definir un algoritmo de compilación que transforma cualquier expresión lambda a una expresión SKI.[11]
Este se define de la siguiente manera:[11]

WHILE e contiene una abstracción lambda **DO**

- Escoger cualquier abstracción lambda más interior de e
- Si el cuerpo de esta abstracción es una aplicación, es necesario aplicar la transformación S
- De otra manera, el cuerpo debe de ser una variable o una constante, así que es necesario aplicar la transformación K o I, como sea necesario.

Veamos un ejemplo de esta compilación, tomando como expresión a compilar a $((\lambda x.\;+\;x\,x)5)$:[11]

$$\begin{gathered} S \Rightarrow S\;(\lambda x.\;+\;x)\,(\lambda x.x)\,5 \\ S \Rightarrow S\;(S\;(\lambda x.\;+\;x)\,(\lambda x.x))\,(\lambda x.x)\,5 \\ I \Rightarrow S\;(S\;(\lambda x.\;+\;x)\,I)\,(\lambda x.x)\,5 \\ I \Rightarrow S\;(S\;(\lambda x.\;+\;x)\,I)\,I\,5 \\ K \Rightarrow S\;(S\;(K+)\,I)\,I\,5 \end{gathered}$$

Por último, podemos evaluarlas para asegurarnos de la correctitud de la expresión anterior:[11]

$$\begin{gathered} S\;(S\;(K+)\,I)\,I\,5 \\ \to S\;(K+)\,I\,5\,(I\,5) \\ \to K\;+\;5\,(I\,5)\,(I\,5) \\ \to +\,(I\,5)\,(I\,5) \\ \to +\,5\,(I\,5) \end{gathered}$$

$$\begin{gathered}\to +\,5\,5\\ \to 10\end{gathered}$$

III-C. Generación de código

III-C1. Cálculo Lambda a Cálculo SKI: Un problema con que cuenta la compilación introducida en III-B2, llamada *abstracción bracket* de Schoenfinkel, es que esta tiende a causar que las expresion lambda exploten cuadráticamente en tamaño al ser compiladas a SKI.[17] Esto lo podemos ejemplificar con la expresión $\lambda x.\lambda y.\, y\, x$, pues al realizar la traducción a SKI, tenemos que:[17]

$$\begin{gathered}\lambda x.\lambda y.\, y\, x\\ \to \lambda x.\, S(\lambda y.\, y)\,(\lambda y.\, x)\\ \to \lambda x.\, S(I)\,(K\, x)\\ \vdots\\ S(S(K\, S)\,(K\, I))(S(K\, K)\, I)\end{gathered}$$

Es por esto que en archivo *Kiselyov.hs* se implementa una traducción distinta basada en la versión en *OCaml* descrita en el artículo: *λ to SKI, Semantically* [17].
Para entender esta traducción primero es necesario introducir 2 combinadores nuevos que son casos especiales de la función S, estos se definen de la siguiente manera: [17]

$$\begin{gathered}B\, f\, g\, x \to f\,(g x)\\ C\, f\, g\, x \to f\, x\, g\end{gathered}$$

En palabras más simple, podemos decir que B representa a la composición de funciones, mientras que C representa el combinador que nos permite voltear argumentos.[17]
Un concepto en el cuál se basa esta tradución es en las reglas de tipado para el cálculo lambda escrito con índices de De Brujin.[17] Estas reglas son:[17]

- Para las variables

$$\textsc{Var}\ \frac{}{\tau \ \vdash\ z : \tau}$$

  Esta regla nos indica que en un contexto que solo contiene al tipo τ, la variable que cuenta con el índice 0, denotada por z tiene el tiempo τ.

- Para agregar un nuevo tipo a la izquierda

$$\text{WL}\ \frac{\Gamma \ \vdash\ e : \tau}{\sigma, \Gamma \ \vdash\ e : \tau}$$

  Esta regla nos indica que al momento de agregar un nuevo tipo σ fuera de la expresión e, no se altera el tipo de e.

- Para agregar un nuevo tipo a la derecha

$$\text{WR}\ \frac{\Gamma+ \ \vdash\ e : \tau}{\Gamma+, \sigma \ \vdash\ s\, e : \tau}$$

  Esta regla nos indica que al momento de agregar un nuevo tipo σ, debemos de aplicar la función sucesor a e para corregir el índice, pues ahora $s\, e$ cuenta con el tipo τ.

- Para agregar nuevas funciones

$$\textsc{Abs}\ \frac{\Gamma, \sigma \ \vdash\ e : \tau}{\Gamma \ \vdash\ \lambda e : \sigma \to \tau}$$

  Esta regla nos indica que para agregar un nuevo tipo λe, debemos de añadir a σ al contexto para después proceder a tipar a e. Decimos que el tipo de λ será $\sigma \to \tau$

- Al realizar una aplicación de función

$$\textsc{App}\ \frac{\Gamma \ \vdash\ e_1 : \sigma \to \tau \quad \Gamma \ \vdash\ e_2 : \sigma}{\Gamma \ \vdash\ e_1\ e_2 : \tau}$$

  Esta regla nos indica que una función e_1 cuenta con un tipo $\sigma \to \tau$ y un argumento e_2 con tipo τ, por lo que el tipo resultante será τ.

Por otro lado, para realizar la compilación semántica, utilizamos una serie de reglas similares que producen etiquetas semánticas, la cuáles, dependiendo del caso generan diferentes combinadores SKI, a continuación se presentan las reglas:[17]

- Para las variables

$$\textsc{EVar}\ \frac{}{\tau \ \models\ I}$$

  Esta regla genera la etiqueta semántica I, la cuál simplemente se traduce al combinador SKI I en cualquier caso.

- Al introducir una variable no usada

$$\text{EW}\ \frac{\Gamma+ \ \models\ d}{\Gamma, \sigma \ \models\ (\models K) \coprod (\Gamma+ \models d)}$$

  Esta regla nos indica que al momento de introducir una nueva variable que no es usada en el término, simplemente se ignora, ya que esto se realiza con el combinador K, producimos un objeto sintáctico que aplica K utilizando la función de aplicación $\coprod$.

- En caso de no contar con variables libres

$$\textsc{EAbs0}\ \frac{\models\ d}{\models\ K\, d}$$

  Esta regla nos indica que si no contamos con ninguna variable libre, solamente debemos de aplicar el combinador K, pues se mantiene constante.

- Al remover una variable usada:

$$\textsc{EAbs}\ \frac{\Gamma, \sigma \ \models\ d}{\Gamma \ \models\ d}$$

  Esta regla nos indica que cuando se remueve una variable que es usada, el objeto se mantiene igual, pero se pierde una entrada en Γ.

- Para la aplicación de funciones

$$\textsc{EAbs}\quad \frac{\Gamma_1 \models d_1 \quad \Gamma_2 \models d_2}{\Gamma_1 \sqcup \Gamma_2 \models (\Gamma_1 \models d_1) \coprod (\Gamma_2 \models d_2)}$$

  Esta regla nos indica que al momento de aplicar d_2 a una expresión d_1, debemos de producir un nuevo objeto sintáctico usando la función de aplicación $\coprod$

Por último, tenemos las reglas para la función de aplicación $\coprod$:[17]

- $(\models d') \coprod (\models d) = d'\, d$
- $(\models d') \coprod (\tau_n, \ldots, \tau_1 \models d) = (\models Bd')\,(\tau_n, \ldots, \tau_1 \models d)$
- $(\tau_n, \ldots, \tau_1 \models d') \coprod (\models d) = (\models CCd)\,(\tau_n, \ldots, \tau_1 \models d')$
- $(\tau_n, \ldots, \tau_1 \models d') \coprod (\tau_m, \ldots, \tau_1 \models d') = (\tau_n, \ldots, \tau_1 \models (\models S) \coprod (\tau_n, \ldots, \tau_1 \models d')) \coprod (\tau_m, \ldots, \tau_1 \models d)$

III-C2. Cálculo SKI a C++: Con el fin de poder obtener los archivos ejecutables de nuestros programas, decidimos implementar un módulo encargado de generar el programa equivalente en C++ para entonces aprovechar el compilador g++. Tenemos dos implementaciones de este proceso, una que lleva a cabo una evaluación estricta o ansiosa y otra que permite la evaluación perezosa.

La idea de la generación de código estricto consiste en los siguientes puntos:

- Definir una estructura $Value$ en C++ que puede contener una cadena o una función,
- Función $Apply$ que dadas dos instancias f y x de tipo $Value$ permite aplicar el primero al segundo según sea el caso:
  - Cadena y función → evalúa la función con la cadena vacía como parámetro y si el resultado es una cadena, la concatena a la primera.
  - Cadena y cadena → concatena ambas cadenas.
  - Función y cualquier valor → aplica la función al valor.
  - Otro caso → ocurre un error.
- Implementar el comportamiento de los combinadores S, K, I, B, C y algunas variantes Bs, Cp y Sp, así como las versiones n-arias de S, B y C que sean usadas en el programa SKI dadas por el algoritmo de la sección III-C1.
- Gracias a estas definiciones lo que sigue es reemplazar las expresiones del programa fuente a su definición en C++ que construimos con anterioridad, valor que almacenamos en una variable *ProgramExpr*.
- Se define el método *main* en el que se aplica el programa a cada argumento de línea de comandos mediante la función *Apply* y se imprime el resultado.

La diferencia con la versión de evaluación perezosa es que en lugar de tratar de ejecutar directamente la expresión construida en la traducción a C++, se construye una estructura DAG, con nodos internos de aplicación y hojas de combinadores y constantes, que se *evoluciona* paso por paso. Esto lo vemos con mayor detalle en la sección III-D2.

Se puede enriquecer al cálculo SKI al introducir nuevos combinadores, y al realizar esta evolución expandiendo cada combinador lo más a la izquierda posible, garantizamos la perezosidad (*lazyness*). Por ello, que se dice que SKI es completamente perezoso tras aplicarle dichas optimizaciones descritas por Peyton [11], las cuales discutimos en la siguiente sección.

III-D. Optimizaciones

III-D1. Optimizaciones propias de SKI: La primera optimización importante es cambiar del método de abstracción bracket a la construcción de espacio $O(n)$ de Kiselyov mencionada en la sección III-C1.

En el algoritmo mismo de Kiselyov, se definen estructuras intermedias llamados *operadores abiertos* sobre los cuales se van agrupando combinadores *semánticamente*, con base en los índices de Brujin, para formar versiones n-arias de B, C y S, implementando composición de múltiples funciones (aplicación múltiple hacia el término derecho, $B\,f\,g\,xs \mapsto f(g \leftarrow xs)$), intercambio de orden de múltiples funciones (aplicación múltiple hacia el término izquierdo $C\,f\,g\,xs \mapsto (f \leftarrow xs)g$), y aplicación múltiple a dos funciones (aplicación hacia ambos términos $S\,f\,g\,xs \mapsto (f \leftarrow xs)(g \leftarrow xs)$) [17].

Asimismo, se definen operaciones de *proyección*, los cuales "observan" operadores abiertos para, si así se desea, traducirlos de vuelta al cálculo SKI. [17] Una técnica de optimización es tener estos combinadores n-arios como parte de nuestro SKI enriquecido, y que se proyecten los operadores abiertos directamente a ellos. Una desventaja de esto es que aumenta la complejidad de compilación a C++, pero podemos reducir su impacto al hacer uso extensivo de macros y plantillas, permitiendo además una mayor capacidad de optimización por el compilador de C++.

La segunda optimización importante es el enriquecimiento de SKI con los combinadores B y C como se menciono antes, y los combinadores S', C' y B^*, que aplican algo similar a una *continuación* al incorporar un argumento c al inicio de S, C y B respectivamente [11].

Se definen como [11]

- $S'\,c\,f\,g\,x \to c\,(f\,x)\,(g\,x)$
- $C'\,c\,f\,g\,x \to c\,(f\,(g\,x))$
- $B^*\,c\,f\,g\,x \to c\,(f\,x)\,g$

Así, las reglas de optimización de Peyton son [11]

- $S\,(K\,p)\,(K\,q) \implies K\,(p\,q)$ la K-optimización, que proviene de aplicaciones dentro de una abstracción lambda que no utilizan la variable inmediatamente arriba.
- $S\,(K\,f)\,g \implies B\,f\,g$ la B-optimización, que proviene de aplicaciones dentro de una abstracción lambda cuyo lado derecho es la varible inmediatamente arriba. "Manda" un argumento a la rama izquierda. Es análogo a la η-conversión [11].

  Aplicando un argumento podemos entender mejor el comportamiento:

  $$S\,(K\,f)\,g\,x \to (K\,f\,x)\,(g\,x) \to f\,(g\,x)$$

- $S\,f\,(K\,g) \implies C\,f\,g$ la C-optimización, que es análoga a la B-optimización, pero "manda" el argumento a la rama derecha. Podemos también apreciar esto mejor al darle un argumento:

  $$S\,f\,(K\,g)\,x \to (f\,x)\,(K\,g\,x) \to (f\,x)\,g$$

  Notemos que la regla B y C son versiones de cierto modo débiles de S, la cual manda el argumento a ambas ramas, $S\,f\,g\,x \to (f\,x)\,(g\,x)$. Así estas optimizan este comportamiento que es común en las aplicaciones.
- $S\,(B\,x\,y)\,z \implies S'\,x\,y\,z$ la S'-optimización. Si tenemos muchas expresiones lambdas anidadas, i.e. una subexpresión de muchos argumentos, entonces se aplicará muchas veces la B-optimización junto con el combinador S para repetirlo. Para reducir estas expansiones, podemos introducir "contexto" en la forma de un argumento que domina la expansión.

  Veamos cómo se expande lo anterior

  $$S\,(B\,c\,f)\,g\,x \to (B\,c\,f\,x)\,(g\,a) \to c\,(f\,x)\,(g\,x)$$

  Notemos como esto corresponde a, si c es la abstracción lambda interior en $\lambda x.\lambda y.f_{xy}g_{xy}$, donde denotamos como subíndice de f y g sus dependencias, esto es evaluar f y g en el entorno de x y pasar f_y y g_y al entorno de y.
- $B\,(c\,f)\,g \implies B'\,c\,f\,g$ la B'-optimización. Es análoga a la S'-optimización pero si sólo el lado derecho depende de la abstracción exterior, en el ejemplo anterior x.

  En otras palabras, aplica al caso $\lambda x.\lambda y.f_y g_{xy}$.
- $C\,(B\,c\,f)\,g \implies C'\,c\,f\,g$ la C'-optimización. Es análoga a la S'-optimización pero si sólo el lado izquierdo depende de la abstracción exterior, en el ejemplo anterior x.

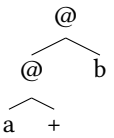
  En otras palabras, aplica al caso $\lambda x.\lambda y.f_{xy}g_y$.

Si se desea saber exactamente cuáles optimizaciones de SKI implementamos, Peyton nos sintetiza estas optimizaciones y otras pocas triviales en su figura 16.5 en la sección 16.2 *Optimizations to the SK scheme* de su libro *The Implementation of Functional Programming Languages* [11].

III-D2. Graph reduction: Para poder evaluar las expresiones del programa vamos a representarlas como arboles de sintaxis, donde las hojas van a representar los valores constantes, como lo son integers, chars y booleans; las funciones predefinidas, como lo son las operaciones aritmeticas o booleanas; y los nombres de las variables. Se utiliza el símbolo '@' como una indicación de que el nodo representa la aplicación de una función sobre un conjunto de variables. Representamos la aplicación de una función f sobre una variable x de la siguiente manera: [7]

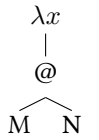
Árbol de sintaxis para f(x)

Si tenemos una función que se aplica sobre mas de una variable, entonces vamos a utilizar la currificación para convertir dicha función en una serie de funciones anidadas. Por ejemplo, para el caso de una suma de dos variables tenemos el siguiente árbol: [7]



Árbol de sintaxis para a+b

Podemos representar una λ-expresión como un árbol de sintaxis de forma que la expresión λx se represente en la raíz del arbol y el cuerpo se represente como la parte inferior del árbol. Podemos entonces representar la aplicación de una λ abstracción de la forma $(\lambda x.M)N$ de la siguiente manera: [7] [11]



Árbol de sintaxis para $\lambda x.M$

Para poder representar los árboles de sintaxis y la forma de implementarlos dentro de un compilador se requiere de una estructura de datos donde cada nodo en los árboles se represente con una celda, la cual en su lado izquierdo va a contener una etiqueta que defina el tipo de celda (aplicación, número, operación, o función), mientras que el lado derecho va a contener dos o más celdas para los argumentos. [18]

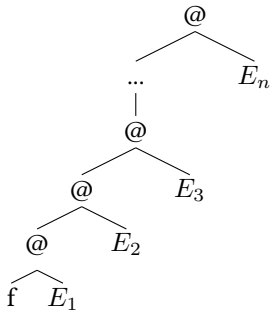
Dicha estructura de datos puede contener la dirección puede incluir dos tipos de instrucciones. Si la celda contiene una dirección a otra celda entonces decimos que es un puntero y que apunta a dicha celda. En otro caso, si la celda no es un puntero entonces debe contener un valor atómico. [18] [11]

Cuando un programa de un lenguaje funcional es compilado en una computadora, dicho programa se representa como una gráfica y se va a evaluar mediante el graph reduction, un proceso que consistirá en ejecutar reducciones sucesivas a la gráfica hasta encontrar una forma normal. [11] Este proceso va a consistir en dos pasos: seleccionar el siguiente redex por reducir y realizar la reducción.

Para seleccionar el siguiente redex hay dos principales estrategias de evaluación: ansiosa y perezosa. En el caso de lenguajes funcionales se suele utilizar la evaluación perezosa, la cual consiste en evaluar las funciones solo cuando solo cuando su valor es necesario y solo debe evaluar cada argumento una sola vez. [11]

Decimos que un λ-termino esta froma normal de cabeza si el cuerpo e la expresión no contiene ningun λ-subtermino al que se le pueda aplicar una β-reducción. Se diferencia de la forma normal dado que el resto de argumentos aun puede tener argumentos con β-reducciones posibles. [20]

Primero veamos como se va a seleccionar el siguiente redex por reducir. La expresión por reducir va a ser de la forma f E_1 E_2 ... E_n y su representación en forma de gráfica es la siguiente:



La expresión f puede ser una constante, una función incorporada del lenguaje, o una abstracción lambda; mientras que tenemos cero o mas argumentos E_i los cuales pueden ser distintas expresiones. Tendremos entonces tres posibilidades:

1. Si f es una constante, entonces la expresión ya se encuentra en forma normal. Sin embargo, si el número de argumentos E_i es mayor a cero, entonces regresamos un error de tipos ya que se esta intentado aplicar una constante a los argumentos.

2. Si f es una función incorportada con k argumentos entonces evaluamos el redex mas al exterior. Sin embargo, si el número de argumentos disponibles E_i es menor a los k argumentos que utiliza la función entonces la expresión ya se encuentra en forma normal.

3. Si f es una abstracción lambda y hay un argumento disponible, entonces el siguiente redex a evaluar es (f E_1). Sin embargo si no tenemos argumentos por evaluar entonces la expresión ya se encuentra en forma normal. [11]

De esta forma para encontrar el siguiente redex por reducir vamos a recorrer el árbol hasta llegar a la hoja mas a la izquierda en el árbol. A la cadena de nodos de aplicación a la izquierda que vamos a recorrer se le denomina la espina de la expresión y acto de descender por esta espina se le denomina "desenrollar"la espina. Así habiendo llegado a la "punta"de la espina y encontrado el siguiente redex por reducir vamos a regresar a su raíz y continuar al siguiente paso. [21]

El siguiente paso consiste en realizar la reducción, este proceso consiste en realizar transformaciones locales a la gráfica que representa a la expresión. Estos pasos se deben repetir hasta llegar a una gráfica en su forma normal final, la cual será el resultado de la evaluación.[11]

Durante la reducción de los árboles en la gráfica vamos a encontrar casos donde va a haber varias copias del mismo argumento, y vamos a querer evitar tener varias copias del mismo argumento en el árbol. Esto debido a que va a desperdiciarse espacio en la memoria y en el caso de que el argumento contenga redex tendremos que desperdiciar tiempo reduciendo el mismo redex multiples veces. Este problema puede solucionarse al reemplazar el argumento por un puntero al parámetro formal. [22]

Notaremos que al utilizar una estructura con multiples punteros al mismo argumento ya no estaremos manejando un árbol binario, sino que tendremos una gráfica que nos provee con una representación mas compacta de la expresión que estamos evaluando. Es de esta manera que al momento de evaluar las expresiones en la gráfica cada subgráfica va a representar más de una subexpresión en la expansión lineal de la gráfica, permitiendonos reducir multiples redex de forma simultanea. [22]

Cuando aplicamos una abstracción lambda es necesario instanciar una nueva copia del cuerpo de la λ-expresión en lugar de actualizar el cuerpo de la función con las substituciones. Esto se debe a que la abstracción puede aplicarse muchas veces y el cuerpo de la función va a utilizarse para construir una nueva instancia cada vez que se aplica la abstracción, por lo que no se debe modificar el cuerpo original. [22]

De esta manera el algoritmo para reducción consiste en repetir los siguientes pasos hasta haber reducido la gráfica a su forma normal.

1. Desenrolla la espina hasta encontrar un nodo que no sea una aplicación.

2. Examina el nodo encontrado en la punta de la espina.

   a) Constante. Verifica que no se haya aplicado a algo, en ese caso es una forma normal de cabeza y se detiene el algoritmo. En otro caso, regresamos un error.

   b) Función incorporada. Verifica el número de argumentos. Si hay muy pocos argumentos tenemos una forma normal de cabeza y detenemos el algoritmo. En otro

caso evaluamos los argumentos necesarios ejecutando la función incorporada y reemplezando la raíz por el resultado.

c) Abstracción lambda. Verifica si existe un argumento. Si no hay un argumento tenemos una forma normal y detenemos el algoritmo. En otro caso instanciamos el cuerpo de la abstracción lambda, sustituyendo los punteros del argumento por el parámetro formal y sobreescribiendo la raíz del redex por el resultado. [11]

III-D3. Lambda Lifting: Dado que los lenguajes funcionales pueden interpretarse como una versión con azúcar sintáctica del cálculo lambda [7, p. 9], el esquema de β-reducción se refleja directamente en la aplicación de funciones dentro de los programas funcionales. Así, al aplicar una función a un argumento, se genera una nueva *instancia* o copia del cuerpo de la función en la que las apariciones, ahora libres tras eliminar la cabecera, del *parámetro formal* (la variable que representa al argumento en la definición) son reemplazadas por dicho argumento.

Este es un proceso costoso computacionalmente pues se requiere: buscar la expresión a reducir en la gráfica del programa, aplicar una serie de verificaciones para reemplazar únicamente el parámetro formal correspondiente al argumento y una buena cantidad de memoria para almacenar cada copia generada. Además, recordemos que es posible tener múltiples argumentos pero hasta este punto la β-reducción solo nos permite aplicar uno a uno.

Por este motivo, surge la *compilación* de las abstracciones lambda como una forma de aligerar su aplicación. La idea que sigue esta optimización, es permitir que el compilador sea capaz de "adelantar" parte de ese trabajo asociando al cuerpo de una función una **serie fija de instrucciones** que permitan dar lugar a la construcción de nuevas instancias [11, p. 220]. Puede ocurrir que existan abstracciones lambda que al aplicarles la β-reducción de un argumento, generen nuevas abstracciones lambda con valores que dependen precisamente de la o las reducciones anteriores pero que además pueden ser aplicadas a otro argumento, lo que representa un problema para la asignación de una secuencia de instrucciones fija.

Un ejemplo es $\lambda x.\lambda y.\lambda z.if\ x\ then\ y\ else\ z$, pues la reducción del primer argumento dará lugar a una nueva abstracción dependiente de y y de z en la que x habrá sido sustituida por el argumento, comportamiento que se repite con el argumento para y y no es hasta la aplicación del tercer argumento que la instancia del cuerpo de la función está completamente construido, listo para su evaluación. De forma general, es posible notar que el problema surge debido a la existencia de variables libres en las expresiones lambda. Como consecuencia, este tipo de expresiones generan instancias intermedias del cuerpo de la función que podrían ahorrarse si realizáramos sustituciones simultáneamente de varios argumentos pues el resultado se mantiene [11, p. 222].

Al tipo de abstracciones lambda que permiten la reducción simultánea se les conoce como *supercombinadores* y se definieron en la sección III-B. A la aplicación de un combinador la llamamos *redex de supercombinador* y como resultado de su *reducción* obtenemos una instancia del cuerpo del supercombinador pero con las ocurrencias de sus variables libres, sustituidas por los argumentos correspondientes. De esta forma una expresión que involucra un supercombinador y solo algunos de sus n argumentos **no** son un *redex de supercombinador* [11, p. 224, p. 225].

Además, a los supercombinadores de aridad cero, los llamamos *formas de aplicación constante* y son especiales porque no requieren compilar código para ellas ya que una sola instancia es suficiente para ser compartida [11, p. 224].

Una vez visto el contexto que origina el proceso de *lambda lifting* y su idea general, lo siguiente es revisar el algoritmo que permite a esta optimización convertir todas las expresiones lambda del programa en supercombinadores.

Algoritmo 1 lambda lifting

1: **Hasta que** no queden más abstracciones lambda:
   1. Elegir cualquier abstracción lambda que **no tenga abstracciones lambda internas** en su cuerpo.
   2. Extraer todas sus variables libres como parámetros adicionales.
   3. Asignar un nombre arbitrario a la abstracción lambda.
   4. Reemplazar la ocurrencia de la abstracción lambda por el nombre asignado, aplicado a las variables libres.
   5. Generar su código de ejecución y registrarlo como una función global del programa bajo el nombre asignado.
2: **Fin**

Ya sabemos que las funciones definidas en un programa funcional son tratadas como cualquier otro valor y que esto brinda un marco de trabajo bastante flexible a la hora de construir nuevas funciones. Una particularidad que hay que notar, es que al momento de declarar funciones es posible definirlas localmente, es decir, dentro de otras declaraciones de funciones. Por ejemplo, en Haskell la siguiente expresión es una declaración válida.

```
normalizarPositivos xs =
  let positivos = filter (> 0) xs
      maximo    = maximum positivos
      normalizar n = n / maximo
  in map normalizar positivos
```

Y si la expresamos con una sintaxis más cercana al cálculo lambda podemos verla como:

$$
\begin{array}{l}
\lambda xs. \\
\quad (\lambda positivos. \\
\quad\quad (\lambda maximo. \\
\quad\quad\quad \text{map}\,(\lambda n.\ (/\ n\ maximo))\ positivos \\
\quad\quad)\ (\text{maximum}\ positivos) \\
\quad)\ (\text{filter}\,(\lambda k.\ (>\ k\,0))\ xs)
\end{array}
$$

Aplicaremos el algoritmo para ilustrar su funcionamiento.

- Comenzamos con $\lambda n.\ (/\ n\ maximo)$
  - Extraemos la variable libre $maximo$ como un parámetro adicional que llamaremos m y obtenemos $\lambda m.\ \lambda n.\ /\ n\ m$
  - Le asignamos el nombre $\$F1$
  - Reemplazamos la abstracción lambda por $\$F1$ aplicado a la variable libre $maximo$.

$$
\begin{array}{l}
\lambda xs. \\
\quad (\lambda positivos. \\
\quad\quad (\lambda maximo. \\
\quad\quad\quad \text{map}\,(\$F1\ maximo)\ positivos \\
\quad\quad)\ (\text{maximum}\ positivos) \\
\quad)\ (\text{filter}\,(\lambda k.\ (>\ k\,0))\ xs)
\end{array}
$$

  - Registramos su código con el nombre $\$F1$

$\$F1\,m\,n\ =\ /\,n\,m$

- Siguiendo con $\lambda k.\ (>\ k\,0)$
  - Como no hay variables libres el siguiente paso es asignar el nombre $\$F2$
  - Reemplazamos la abstracción lambda por $\$F2$

$$
\begin{array}{l}
\lambda xs. \\
\quad (\lambda positivos. \\
\quad\quad (\lambda maximo. \\
\quad\quad\quad \text{map}\,(\$F1\ maximo)\ positivos \\
\quad\quad)\ (\text{maximum}\ positivos) \\
\quad)\ (\text{filter}\,\$F2\ xs)
\end{array}
$$

  - Registramos su código con el nombre $\$F2$

$\$F1\,m\,n\ =\ /\,n\,m$
$\$F2\,k\ =>\ k\,0$

- Al aplicar el algoritmo a $\lambda maximo.\ \text{map}\,(\$F1\ maximo)\ positivos$
  - Extraemos $positivos$ como un parámetro adicional que llamaremos p y obtenemos:

    $\lambda p.\ \lambda maximo.\ \text{map}\,(\$F1\ maximo)\ p$
  - Le asignamos el nombre $F3$
  - Reemplazamos

$$
\begin{array}{l}
\lambda xs. \\
\quad (\lambda positivos. \\
\quad\quad (\$F3\ positivos)\ (\text{maximum}\ positivos) \\
\quad)\ (\text{filter}\,\$F2\ xs)
\end{array}
$$

  - Registramos su código con el nombre $\$F3$

$\$F1\,m\,n\ =\ /\,n\,m$
$\$F2\,k\ =>\ k\,0$
$\$F3\,p\,maximo\ =\ \text{map}\,(\$F1\ maximo)\ p$

- Ahora con:

  $\lambda positivos.\ (\$F3\ positivos)\ (\text{maximum}\ positivos)$
  - No hay variables libres, por lo que solo le asignamos $\$F4$ como nombre y reemplazamos.

    $\lambda xs.\ (\$F4)\ (\text{filter}\,\$F2\ xs)$
  - Registramos su código con el nombre $\$F3$

$\$F1\,m\,n\ =\ /\,n\,m$
$\$F2\,k\ =>\ k\,0$
$\$F3\,p\,maximo\ =\ \text{map}\,(\$F1\ maximo)\ p$
$\$F4\,positivos\ =\ (\$F3\ positivos)\ (\text{maximum}\ positivos)$

- Por último aplicando el algoritmo a $\lambda xs.\ (\$F4)\ (\text{filter}\,\$F2\,xs)$ obtenemos: $\$F5$ y la tabla final es:

$\$F1\,m\,n\ =\ /\,n\,m$
$\$F2\,k\ =>\ k\,0$
$\$F3\,p\,maximo\ =\ \text{map}\,(\$F1\ maximo)\ p$
$\$F4\,positivos\ =\ (\$F3\ positivos)\ (\text{maximum}\ positivos)$
$\$F5\,xs\ =\ (\$F4)\ (\text{filter}\,\$F2\,xs)$

De esta forma al aplicar `normalizarPositivos` a la lista $[2, 4, 6]$ tendríamos:

$\$F1\,m\,n\ =\ /\,n\,m$ $\$F2\,k\ =>\ k\,0$ $\$F3\,p\,maximo\ =\ \text{map}\,(\$F1\ maximo)\ p$ $\$F4\,positivos\ =\ (\$F3\ positivos)\ (\text{maximum}\ positivos)$ $\$F5\,xs\ =\ (\$F4)\ (\text{filter}\,\$F2\,xs)$
$\$F5\ [2,4,6]$

Que se podría reducir como sigue:

$$
\begin{aligned}
&\$F5[2,4,6]\\
&\rightarrow (\$F4)\ (\text{filter}\ \$F2\ [2,4,6])\\
&\rightarrow (\$F4)\ (\text{filter}\ (>\ k\,0)\ [2,4,6])\\
&\rightarrow (\$F4)\ [2,4,6]\\
&\rightarrow ((\$F3\ [2,4,6])\ (\text{maximum}\ [2,4,6]))\\
&\rightarrow (\$F3\ [2,4,6]\ 6)\\
&\rightarrow (\text{map}\ (\$F1\ 6)\ [2,4,6])\\
&\rightarrow (\text{map}\ (/\,n\,6)\ [2,4,6])\\
&\rightarrow [0.3333, 0.6667, 1.0]
\end{aligned}
$$

Así, el espacio de las instancias intermedias se reduce y se hacen sustituciones de forma simultánea sin perder información.

Esta técnica también tiene un área de mejora importante y es que durante el algoritmo es posible generar parámetros redundantes que pueden ser eliminados y para ello es necesario contemplar aspectos como el orden en el que se definen los parámetros adicionales con base en su nivel de profundidad, lo que se puede hacer por ejemplo, con los índices de De Bruijn [11, p. 230].

Esta técnica permite resolver el problema de las variables libres en las declaraciones de funciones locales, reduciendo así la necesidad de crear *closures* durante su evaluación [23]. Es cierto que, en determinados escenarios esto no se traduce necesariamente en una mejora del rendimiento, ya que, dependiendo del patrón de llamadas de la función, puede resultar más eficiente crear un *closure* que introducir un parámetro adicional. A pesar de ello, en términos generales, el uso de esta técnica suele ser beneficioso cuando la función a ser elevada tiene pocas llamadas [23].

En el compilador GHC, el método por defecto para el manejo de variables libres se denomina *closure conversion* [23]. Este proceso consiste en transformar una función con variables libres en una nueva función que recibe explícitamente un ambiente, el cual contiene los valores de dichas variables [24]. De esta manera, la función resultante deja de depender del contexto original y obtiene todos los valores necesarios a partir de su ambiente.

Adicionalmente, GHC cuenta con una implementación de lambda lifting, conocida como *late lambda lifting*, que se aplica justo antes de la fase de generación de código, con el objetivo de no interferir con otras optimizaciones del compilador. Una característica importante de esta implementación es que emplea criterios de selectividad para determinar qué funciones son candidatas a ser transformadas mediante este proceso [23].

III-E. Implementación

Se implementó la solución en Haskell en un proyecto de Cabal, con generación de código en C++.

Se definieron tres lenguajes de programación en dos modos: *estricto* y *perezoso*. Estos lenguajes son, el cálculo SKI con comentarios y cadenas, una versión del cálculo lambda parentizada (similar a Lisp o iswim) con cadenas y comentarios, y un lenguaje original llamado *subs*, que comparte algunas similaridades sintácticas con Haskell, pero sin recursividad directa y con nuestro cálculo lambda parentizado como manera de construir los cuerpos de las funciones. Pensamos en *subs* como un "cálculo lambda de pizarrón", con declaraciones y sustituciones pero respetando que no debe tener recursión, únicamente puntos fijos.

La gramática de *ski* es:

```
<Expresion> ::= <Atomo> { <Atomo> }*
<Atomo> ::= S | s | K | k | I | i | <String>
             | ( <Expresion> )
<String> ::= una cadena multilínea entre "
<Comentario> ::= una línea que inicia en //
<MComentario> ::= texto encerrado entre /* y */
<Espacio> ::= espacio en blanco
```

La gramática de *lambda* es:

```
<Expresion> ::= <Identificador>
             |  <String>
             |  ( <Expresion> <Expresion> )
             |  [ <Identificador> <Expresion> ]
<String> ::= una cadena multilínea entre "
<Identificador> ::= comienza con una letra o .
    y puede continuar con letras, dígitos, ', ?, _, -, .
<Comentario> ::= una línea que inicia en //
<MComentario> ::= texto encerrado entre /* y */
<Espacio> ::= espacio en blanco
```

La gramática de *subs* es:

```
<Programa> ::=
    <Declaracion> { ; { ; }* <Declaracion> }* { ; }*
<Declaracion> ::=
    <Identificador> { <Identificador> }* = <Expresion>
<Expresion> ::= <Identificador>
             |  <String>
             |  ( <Expresion> <Expresion> )
             |  [ <Identificador> <Expresion> ]
<String> ::= una cadena multilínea entre "
<Identificador> ::= comienza con una letra o . y puede
    continuar con letras, dígitos, ', ?, _, -, .
<Comentario> ::= una línea que inicia en //
<MComentario> ::= texto encerrado entre /* y */
<Espacio> ::= espacio en blanco
```

En todo caso, parte del proceso de compilación es la reducción a el lenguaje anterior, *subs* se reduce a una expresión lambda por medio de identificar las definiciones y variables libres, asegurar que no hay definiciones duplicadas, construir una gráfica dirigida de la variable libre hacia su definición, asegurarse que esta gráfica sea una DAG, hacer las sustituciones de las hojas hacia arriba hasta que la declaración *main* tenga

como lado derecho una expresión lambda pura. Asimismo, el lenguaje *lambda* (así como *subs* al llegar a este paso), se reduce a nuestra versión enriquecida de SKI por medio del algoritmo de Kiselyov. Finalmente *ski* se transforma con las optimizaciones de Peyton (así como *subs* y *lambda* al llegar a este paso) y se genera el código ya sea con el generador *strict* o *lazy*.

Así, de manera implícita para *subs* y *lambda*, realizamos *lambda lifting* con nuestra traducción a SKI con el algoritmo de Kiselyov.

Además, se realizaron pruebas al construir, compilar y ejecutar programas en todos estos lenguajes y generadores de código en el archivo *tests.py*.

Estos procesos se ven reflejados en la estructura del proyecto dada a continuación, notando que los *Lexer*, *Parser* y *CodeGenerator* particulares a utilizar son seleccionados al momento de compilación del proyecto, i.e. al designar los distintos objetivos de compilación en el archivo *.cabal*.

```
ski-compiler/
├── app
│   ├── common
│   │   ├── Expr.hs
│   │   ├── Graph.hs
│   │   └── Util.hs
│   ├── lambda-parse
│   │   ├── Lexer.hs
│   │   ├── LexerSpec.txt
│   │   ├── Parser.hs
│   │   └── util
│   │       ├── Kiselyov.hs
│   │       ├── LambdaDesugar.hs
│   │       ├── Lambda.hs
│   │       └── LambdaParser.hs
│   ├── lazy
│   │   └── CodeGenerator.hs
│   ├── Main.hs
│   ├── ski-parse
│   │   ├── Lexer.hs
│   │   ├── LexerSpec.txt
│   │   └── Parser.hs
│   ├── strict
│   │   └── CodeGenerator.hs
│   └── subs-parse
│       ├── Lexer.hs
│       ├── LexerSpec.txt
│       └── Parser.hs
├── CHANGELOG.md
├── LICENSE
├── ski-compiler.cabal
└── tests.py
```

Se optó por no implementar operaciones aritméticas y demás básicas para centralizar el proyecto en las técnicas particulares al paradigma funcional. Como operaciones entre términos no-funcionales usamos las cadenas, y recordemos que podemos codificar números y sus operaciones como numerales de Church, lo cual hacemos extensamente en los programas de ejemplo y al momento de interpretar la entrada de la línea de comandos, utilizando la técnica *duplica e incrementa* para codificarlos en $O(\log n)$ combinadores en el generador perezoso (el estricto lo expandiría inmediatamente a su versión $O(n)$ por lo que no vale la pena esta representación).

Nuestros lexers los diseñamos de modo que *lambda* y *subs* sean compatibles, generando el código en Haskell a partir de una especificación de expresiones regulares utilizando el generador de analizadores léxicos del reporte *Construcción de un Generador de Analizadores Léxicos en Haskell* [25].

El *parsing* en los tres casos se hizo manualmente pues es muy sencilla la gramática de estos lenguajes y además permite una mayor flexibilidad en el manejo de mensajes y errores, así como qué informacion pasa a las siguientes etapas. En el caso de *ski*, el parsing se hace token por token armando el *árbol de sintaxis abstracta* (AST) de arriba hacia abajo por descenso recursivo, construyendo una derivación por la izquierda de izquierda a derecha en ese respecto semejante en un *parser* LL(1), además va modificando el árbol de derivación directamente (con optimización de cola) y no tiene backtracking.

Tanto *lambda* como *subs* comparten la mayor parte de su parser, distinguiéndose en su comportamiento ante variables libres (y las extensiones que realiza *subs* a *lambda* como declaraciones). Esto sigue un esquema de analizador por descenso recursivo más tradicional, teniendo una correspondencia más directa con su gramática.

La estructura más importante es nuestra *Expr*, que corresponde con nuestro SKI enriquecido. Su definición es la siguiente:

```haskell
data Expr = SComb
          | KComb
          | IComb
          | BComb
          | CComb
          | S'Comb
          | C'Comb
          | BsComb
          | BnComb Int
          | CnComb Int
          | SnComb Int
          | Str String
          | EApp Expr Expr
          deriving (Show, Eq)
```

Como un detalle adicional, el código generado perezoso en C++ implementa un recolector de basura primitivo haciendo uso de *apuntadores inteligentes* y reusando referencias lo más posible, en particular en la captura de argumentos mediante una lista ligada que reusa las capturas anteriores aunque el objeto sea uno nuevo, ya que los nodos de la gráfica los implementamos de manera inmutable.

Nuestro trabajo es una base interesante para expandir el lenguaje, ya que podríamos incluír un preámbulo con declaraciones comunes predefinidas como operaciones con numerales de Church, combinadores importantes, booleanos codificados, y demás, así como implementar números (de máquina) con expresiones aritméticas, de punto flotante, enteros, con y sin signo, etc. Además se podría extender para incluir recursión, recursión mutua, tipos, etc. Sabemos como manejar estas construcciones "fuera de SKI/lambda/subs puro" pues lo hemos hecho con nuestras cadenas y sus concatenaciones.

IV. Resultados y Discusión

Ejecutando nuestras pruebas observamos lo que esperábamos; en todo caso se obtienen los resultados correctos o bien el programa estricto no termina o hace un *segfault* por tratar de expandir de manera estricta un término que contiene un operador de punto fijo no-estricto (en particular Y).

Además, aunque los programas son sencillos, se alcanzan resultados de manera no restrictivamente lenta. Si compilamos un programa ski que sólo tiene a I, y en la línea de comandos ejecutamos `100000 /a /b`, se termina casi inmediatamente. El generador estricto termina en $294\,ms$, mientras que el perezoso en $700\,ms$. Esto hace sentido, pues no hay mucho que pueda ganar la evaluación perezosa en este caso, pero sí tiene una carga adicional en mantener y modificar la estructura del árbol así como la recolección de basura.

Esto es en una laptop con las siguientes especificaciones de CPU:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Address sizes:       39 bits physical, 48 bits virtual
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Vendor ID:           GenuineIntel
Model name:          i5-5300U CPU @ 2.30GHz
Thread(s) per core:  2
CPU max MHz:         2900.0000
CPU min MHz:         500.0000
BogoMIPS:            4589.64
L1d cache:           64 KiB (2 instances)
L1i cache:           64 KiB (2 instances)
L2 cache:            512 KiB (2 instances)
L3 cache:            3 MiB (1 instance)
```

A continuación incluimos el programa *twenty.subs* que hace un caso de estudio interesante.

```
// Punto de entrada del programa
main = ((Mul Four) Five);

// Aritmética básica
Zero = [f [x x]];
Succ n = [f [x (f ((n f) x))]];
Add m n = [f [x ((m f) ((n f) x))]];
Four = (Succ (Succ (Succ (Succ Zero))));
Five = (Succ (Succ (Succ (Succ (Succ Zero)))));

// Booleanos
True  = [t [f t]];
False = [t [f f]];
IsZero n = ((n [x False]) True);

// Predecesor
Pred n = [f [x (((n [g [h (h (g f))]]) [u x]) [u u])]];

// Combinador Y
Y f = ([x (f (x x))] [x (f (x x))]);

// Multiplicación recursiva con Y
// Mul m n = if (IsZero m) then Zero
//           else Add n (Mul (Pred m) n)
Mul = (Y [self [m [n (((IsZero m) Zero)
         ((Add n) ((self (Pred m)) n))) ] ] ]);
```

El archivo generado del programa *twenty.subs* con evaluación estricta es de $108\,KB$, y el de evaluación perezosa de $168\,KB$. Ambos son compilados haciendo uso de *stripping*, por lo que procura minimizar el tamaño del ejecutable dentro de lo que aún sea altamente eficiente.

En ninguno de los programas generados se rebasó la aridad 4 en los combinadores Cn, Bn ni Sn, generalmente llegando a la aridad 3 y 2.

Así, aunque en general es más eficiente un sencillo código evaluado de manera estricta, podemos ver en la imposibilidad de ejecutar *twenty.subs* que la evaluación perezosa cuenta con ventajas al poder ejecutar más programas de manera exitosa y optimizar programas que producen una alta cantidad de información que descartarán más adelante.

V. Conclusión

Durante el desarrollo de este trabajo, hemos visitado la definición de cálculo lambda y de esta forma fuimos capaces de profundizar en el concepto del cálculo SKI. Utilizando los conceptos explorados anteriormente pudimos utilizar algoritmos de traducción entre el cálculo lambda y cálculo SKI, así como la traducción de calculo SKI al lenguaje C para desarrollar una implementación concreta del cálculo lambda usando al cálculo SKI como un lenguaje intermedio para la evaluación de los resultados antes de ser pasado finalmente a un lenguaje para su ejecución.

Sumado a esto, exploramos distintas estrategias de optimización para mejorar el rendimiento de nuestro trabajo, dichas estrategias consideraron tanto optimizaciones propias del cálculo SKI, como métodos de reducciones en gráficas para incorporar la evaluación perezosa en la implementación antes mencionada. Asimismo, también se utilizó la técnica de Lambda Lifiting para optimizar tanto el uso del espacio mediante instancias intermedias como la aplicación de operaciones de

sustitución simultanea para mejorar el rendimiento de nuestro compilador.

Habiendo investigado las estrategias de investigación mencionadas anteriormente, nos es posible entender mas claramente el funcionamiento de los compiladores para lenguajes funcionales. Sin embargo, también nos es posible saber que aun hay distintas formas en las que podríamos expandir nuestra investigación y de esta forma optimizar nuestra implementación. Una de estas formas podría ser la incorporación de otros combinadores, como lo son Y, B, C, W, etc., los cuales podrían permitirnos explorar nuevos patrones de programación. Por otro lado, se podría optimizar el proceso de reducción de gráficas mediante la reutilización de subgráficas y técnicas de recolección de basura. Por ultimo, podríamos optimizar el proceso de reducción de gráficas mediante computo paralelo, una idea que se ha explorado anteriormente en varios artículos. [26]

También valdría la pena explorar otros métodos modernos de optimización, así como la generación a una *spineless tagless g-machine* (STG) como lo hace Haskell [27] y abandonar la reducción a supercombinadores en favor de aumentar el nivel semántico con el que se llega a esta etapa. Recordemos que *Miranda*, un precursor a Haskell, hacía su compilación de la misma forma que nosotros aquí, con una máquina ejecutora de SKI [28]. Otra alternativa involucra el hacer uso de algún tipo de caché de expresiones ya computadas para poder ahorrar tiempo de cómputo en expresiones comunes y vistas muchas veces, quizás con la implementación de *hashing* y un caché *LFU*, *LRU*, etc.

Referencias

[1] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam, The Netherlands: Elsevier, 1984, vol. 103, ISBN: 978-0-444-87508-2.

[2] R. Wilson, *Introduction To Graph Theory*, 4.ª ed. Harlow, Essex, England: Longman, 1996, ISBN: 0-582-24993-7.

[3] G. G. Jørgen Bang-Jensen, *Digraphs Theory, Algorithms and Applications*, 4.ª ed. Springer-Verlag, 2007, ISBN: 978-1-84800-997-4.

[4] S. L. Chakrabarti, «States, side effects and multiple contexts in programming languages,» 1985.

[5] K. Ramírez Pulido, M. Soto Romero y J. Enríquez Mendoza, *Lenguajes de Programación. Nota de clase 1: Conceptos generales*, Nota de clase, Facultad de Ciencias, UNAM, curso 2022-1, 23 de sep. de 2021.

[6] R. Awati y A. Bertram. «Declarative Programming: Definition,» TechTarget. dirección: https://www.techtarget.com / searchitoperations / definition / declarative - programming.

[7] R. Plasmeijer y M. van Eekelen, *Functional Programming and Parallel Graph Rewriting*. Berlin, Germany: Springer, 1993.

[8] P. Hudak, «Conception, evolution, and application of functional programming languages,» *ACM Comput. Surv.*, vol. 21, n.º 3, págs. 359-411, sep. de 1989, ISSN: 0360-0300. DOI: 10.1145/72551.72554. dirección: https://doi.org/10.1145/72551.72554.

[9] J. Hughes, «Why Functional Programming Matters,» en *Research Topics in Functional Programming*, D. Turner, ed., Enlace consultado el 28 nov. 2025, Reading, MA, USA: Addison-Wesley, 1990, págs. 17-42. dirección: https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

[10] R. Bird y P. Wadler, *Introduction to Functional Programming*, 2.ª ed. Prentice Hall, 1998, Enlace consultado el 28 nov. 2025. dirección: https://www.cs.kent.ac.uk/people/staff/dat/miranda/ifp.pdf.

[11] S. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall Internaltional (UK) Ltd., abr. de 1987, Chapters also by: Philip Wadler, Programming Research Group, Oxford; Peter Hancock, Metier Management Systems, Ltd.; David Turner, University of Kent, Canterbury. dirección: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages-2/.

[12] F. Cardone y J. R. Hindley, *History of Lambda-calculus and Combinatory Logic*, 2006. dirección: https://www.labouseur.com/courses/tpl/History-of-Lambda-Calculus.pdf.

[13] J. Alama y J. Korbmacher, *The Lambda Calculus*, *The Stanford Encyclopedia of Philosophy*, Winter 2024 Edition por Edward N. Zalta, Uri Nodelman. dirección: https://plato.stanford.edu/archives/win2024/entries/lambda-calculus/.

[14] G. Manzonetto, «Models and theories of λ-calculus,» *arXiv preprint*, 2009. eprint: arXiv:0904.4756. dirección: https://arxiv.org/pdf/0904.4756.

[15] H. Barendsen y E. Barendsen, «Introduction to Lambda Calculus,» 1994. dirección: https : / / jeremybolton . georgetown.domains/courses/pl/barendregt.94.pdf.

[16] H. Barendregt, W. Dekkers y R. Statman, *Lambda Calculus With Types*. Cambridge University Press, 2013, versión electrónica disponible en Google Books, ISBN: 978-0-521-766-142. dirección: https://books.google.com.mx/books?hl=es&lr=&id=qR_NAQAAQBAJ.

[17] O. Kiselyov, *λ to SKI, Semantically — Declarative Pearl*, https://okmij.org/ftp/tagless-final/ski.pdf.

[18] P. J. K. Jr., *An Architecture for Combinator Graph Reduction (TIGRE)*. Carnegie Mellon University, 1990.

[19] A. Aiken, *Lecture 2: Combinator Calculus*, https://web.stanford.edu/class/cs242/materials/lectures/lecture02.pdf.

[20] W. Kluge, *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer-Verlag Berlin Heidelberg, 2005, ISBN: 978-3-540-27359-2. dirección: https://link.springer.com/book/10.1007/b138965.

[21] H. Barendregt, J. Kennaway, J. Klop y M. Sleep, *Needed Reduction and Spine Strategies for the Lambda Calculus*. Springer-Verlag Berlin Heidelberg, 1986. dirección: https://ir.cwi.nl/pub/6258/6258D.pdf.

[22] C. P. Wadsworth, «The Semantics and Pragmatics of the Lambda Calculus,» University of Oxford, 1971. dirección: https://www.computerhistory.org/collections/catalog/102720343/?media=119831.

[23] Haskell Foundation. «Lambda Lifting.» Sección 3.2.3 del *Haskell Optimization Handbook*, visitado 13 de dic. de 2025. dirección: https://haskell.foundation/hs-opt-handbook.github.io/src/Optimizations/GHC_opt/lambda_lifting.html.

[24] Haskell Foundation. «Closure Conversion.» Entrada del glosario, visitado 13 de dic. de 2025. dirección: https://haskell.foundation/hs-opt-handbook.github.io/src/glossary.html#term-Closure-Conversion.

[25] S. Romero, A. Pinacho, M. E. Chávez y A. Escalante, *Construcción de un Generador de Analizadores Léxicos en Haskell*, https://github.com/srp-mx/compiladores-lexer/blob/main/Reporte.pdf, Reporte publicado en GitHub, 2025. visitado 13 de dic. de 2025.

[26] L. Augustsson y T. Johnson, «Parallel Graph Reduction with the <v,G> Machine,» 1990. dirección: https://dl.acm.org/doi/pdf/10.1145/99370.99386.

[27] S. L. P. Jones, «Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine,» *Journal of Functional Programming*, vol. 2, n.º 2, págs. 127-202, abr. de 1992. DOI: 10.1017/S0956796800000319. visitado 14 de dic. de 2025. dirección: https://www.cambridge.org/core/services/aop-cambridge-core/content/view/S0956796800000319.

[28] D. Oisín Kidney. «Fun with Combinators,» visitado 14 de dic. de 2025. dirección: https://doisinkidney.com/posts/2020-10-17-ski.html.